Merkle Patricia Trie Tree

创新创业实践

曹文馨：202000460034

Contents

1 Merkle Tree

Merkle Tree 是一种数据结构，用来验证计算机之间存储和传输数据的一致性，如果不使用这一数据结构，一致性的验证需要消耗大量的存储和网络资源，如比对计算机之间的所有数据；使用 Merkle Tree，只需要比对 merkle root（根节点）就可以达到相同的效果。整个过程，简单的描述如下[1]：

1. 将数据通过哈希之后放置在叶子节点之中。
2. 将相邻两个数据的哈希值组合在一起，得出一个新的哈希值。
3. 依次类推，直到只有一个节点也就是根节点。
4. 在验证另外的计算机拥有和本机相同的数据时，只需验证其提供的根节点和自己的根节点一致即可。

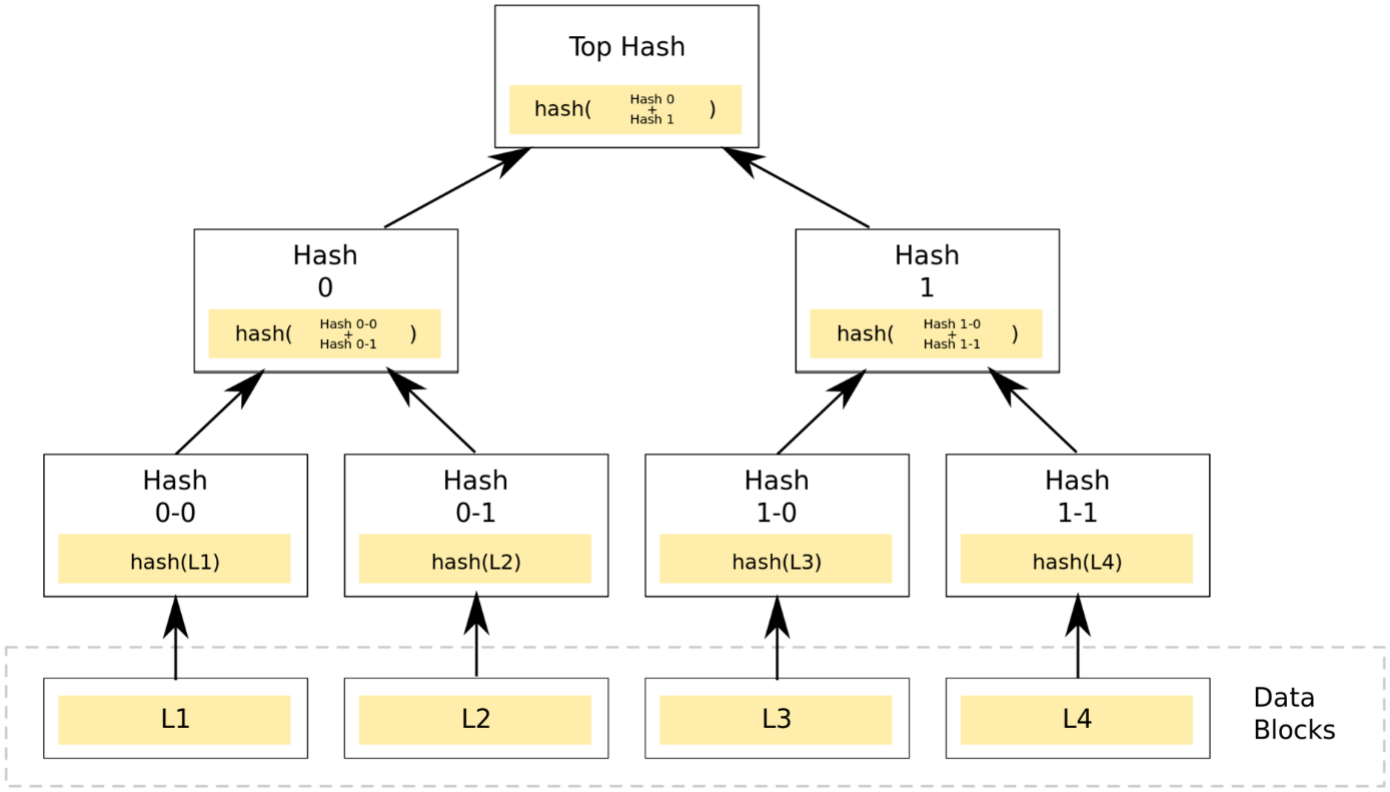

Figure 1: Merkle Tree

1.1 Merkle Tree 在区块链中的应用

1. 在 Bitcoin 等区块链平台，使用 Merkle tree 来对区块中的交易进行汇总和验证，并将 Merkle root 作为对所有交易的汇总嵌入到 block header 中。

   在每个区块中，包含有：

(a) 当前 block ID value：为当前区块 header 域的 hash 值，Merkle root 为该 header 域的一部分。

(b) previous block ID：上图中的 Parent 是指前一区块的 block ID 值。

在 Bitcoin 中，将区块中所有交易的 Merkle root 包含在 block header 中，借此保证没有任何交易被篡改过。

任何交易数据的变动都将造成 Merkle root 的修改，使得其不再与 block header 中所记录的 Merkle root 匹配。借助 block header 的链接，这种交易的不可篡改性可传递至整个区块链网络。

2. 在 blockchain pruning 中借助 Merkle tree 实现了对空间的裁剪——将本地不再需要的交易数据移除。

3. 此外，SPV，简单支付验证，是一种不用运行全节点、只需保存所有的区块头，就可以验证支付的技术手段。是一个在轻客户端环境下，就能验证支付有效性的过程。在 SPV 中，轻量级区块链客户端仅存储 block headers，但也希望验证它在区块链中收到的付款是否为有效交易。由于缺少完整的交易细节，SPV 客户端使用 Merkle tree 与完整节点协作来有效地验证交易细节。

2 Patricia Trie Tree

Trie Tree（字典树或前缀树），又名单词查找树，是哈希树的变种。

Trie Tree 的典型应用有：

1. 用于统计、排序和保存大量的字符串（但不局限于字符串）

2. 常被搜索引擎系统用于文本词频统计。

基本性质如下：

1. 根节点不包含字符，除根节点外每个节点都只包含一个字符。

2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。

3. 每个节点的所有子节点包含的字符都不相同。

Trie Tree 的优点主要是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希树高。

具体原因是：相比于哈希表，使用前缀树来查询拥有共同前缀 key 的数据时十分高效。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[2]。

Trie Tree 的缺点主要是：

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。

3 Merkle Patricia Trie Tree

MPT 是默克尔树和帕特里夏树的结合缩写，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，以太坊使用 mpt 存储所有账户状态，以及每个区块中的交易和收据数据。是一种典型的用空间换时间的数据结构。

以太坊 MPT 树中，每个节点最多有 16 个子节点。与前缀树相同，MPT 同样是把 key-value 数据项的 key 编码在树的路径中，但是 key 的每一个字节值的范围太大（[0-127]），因此在以太坊中，在进行树操作之前，首先会进行一个 key 编码的转换：将一个字节的高低四位内容分拆成两个字节存储。通过编码转换，key' 的每一位的值范围都在 [0, 15] 内。因此，一个分支节点的孩子至多只有 16 个。以太坊通过这种方式，减小了每个分支节点的容量，但是在一定程度上增加了树高。

主要功能为:

1. 存储任意长度的 key-value 键值对数据。

2. 快速计算所维护数据集哈希标识。

3. 快速状态回滚。

4. 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。

3.1 MPT 的删改操作

3.1.1 插入键值对

参数列表[3]:

- 当前节点 n。
- 节点与待插入的 key 的共同前缀 prefix。
- 待插入的 key。
- 待插入的 value。

返回值列表:

- 是否是脏数据。
- 新节点引用。
- 错误消息。

操作步骤:

1. 如果待插入的 key 的长度为 0，那么意味着在当前节点上更新待插入的 value。

2. 判断当前节点 n 的类型：

   (a) 压缩节点：

      i. 计算当前节点 n 的 key 与待插入的 key 的相同前缀下标 +1。

      ii. 如果相同前缀下标与当前节点 n 的 key 长度一致，也就意味着待插入的 key 与当前节点 n 的 key 完全匹配，就是更新当前节点 n 的 value。递归调用当前方法，参数为当前节点 n 的 value，prefix+ 之前计算的相同前缀，待插入的 key 剩下的部分，以及 value。完成后将返回的节点作为当前节点的 value 返回。如果不一致，也就意味着有了分支。新建 fullNode，分别对当前节点 n 的 key 和待插入的 key 的不一致下标开始递归调用插入后续 key 及 value，返回值为 fullNode 的两个分支。节点 n 插入的是 n 的 value，另一分支为待插入的 value。递归完成后，当前调用返回 shortNode，key 为相同前缀，value 为新建的 fullNode。还有一种情况，如果最开始节点 key 就不匹配，直接就返回 fullNode，因为没有共同前缀 key。

   (b) 分支节点：因为是分支节点，那么每个 child 的 key 只有一位，那么只要将 value 插入跟待插入的 key 的第一位相同的 child 位置就可以了。

   (c) 哈希节点：哈希节点先去数据库中 load 相关节点的数据，之后再递归调用。

   (d) 其他：直接将 key，value 包装成压缩节点返回。

3.1.2 删除键

参数列表[3]**：**

- 当前节点 n。
- 节点与待删除的 key 的共同前缀 prefix。
- 待删除的 key。

返回值列表：

- 是否是脏数据。
- 新节点引用。
- 错误消息。

操作步骤：

判断节点类型：

1. 压缩节点：

   (a) 寻找节点 n 的 key 与待删除的 key 的共同前缀下标 +1。

   (b) 如果下标小于节点 n 的 key 的长度，表示 key 没匹配上，直接返回；如果下标等于节点 n 的 key 的长度，表示该节点 n 正是需要被删除的节点，删除操作就是返回的新节点引用为 nil，当前节点 n 在内存中就成了野节点，被 mpt 树排除在外了；剩下的情况，表示待删除的 key 存在当前节点 n 的子树，递归调用删除方法，返回后，这里再判断一次返回的新节点 child 的类型：

i. 压缩节点：父节点 n 与子节点 child 都是一种节点类型，直接合并 key，value 为子节点的 value。

ii. 其他：也是返回压缩节点，其 key 为当前节点 n 的 key，value 是递归调用返回的子节点。

2. 分支节点：和插入类似，将待删除的 key 的首位与该节点对应的 child 带入递归调用，返回后更新对应 child 的 value；删除之后需要检查分叉节点的所有 child 是否有多于 2 个非 nil 的 child，如果少于 2 个，就可以合并 child。

3. 哈希节点：先从数据库 load 数据，再递归调用。

4. 值节点：直接删除当前节点。

3.1.3 获取键值对

参数列表[3]：

- 当前节点 n。
- 已经过滤的子 key 在 key 中的 pos。
- 待查找的 key。

返回值列表：

- 查找的 value。
- 是否从数据库中 load 数据。
- 新节点引用。
- 错误消息。

操作步骤：

判断当前节点的类型：

1. 值节点：直接返回节点 n 的 value。

2. 压缩节点：如果剩余的 key（查找的 key 的长度-pos）的长度小于节点 n 的 key 的长度或者两个 key 的前缀不匹配，表示在树种没有找到对应的 key，直接返回；待查找的 key 与该节点 n 的 key 是匹配的，那么只需要将节点 n 的 value 和剩余的待查找的子 key 带入递归。

3. 分支节点：找到对应 key 的 child 递归调用。

4. 哈希节点：先载入数据，再递归调用。

4 参考文献

References

[1] https://blog.csdn.net/shangsongwww/article/details/119272573

[2] https://blog.csdn.net/mutourend/article/details/114384264

[3] https://zhuanlan.zhihu.com/p/32924994