

**课程报告**

**课程名称：区块链技术与应用**

**专业班级： 计算机2020级11班**

**学 号： M202073600**

**姓 名： 周金伟**

**指导教师： 肖江**

**报告日期： 2020年12月24日**

**计算机科学与技术学院**

目录

[1 Merkle tree介绍 3](#_Toc59740170)

[1.1 简介 3](#_Toc59740171)

[1.2 特点 4](#_Toc59740172)

[2 Merkle tree实现 4](#_Toc59740173)

[2.1 Merkle Node 4](#_Toc59740174)

[2.2 哈希方法 4](#_Toc59740175)

[2.3 Merkle Tree 5](#_Toc59740176)

[2.4 实现结果 5](#_Toc59740177)

[3 优化Merkle tree的出发点 6](#_Toc59740178)

[4 优化方案 7](#_Toc59740179)

[4.1 Trie Node 7](#_Toc59740180)

[4.2 Trie Tree 7](#_Toc59740181)

[5 优化效果 8](#_Toc59740182)

[5.1 实验方案 8](#_Toc59740183)

[5.2 实验结果 8](#_Toc59740184)

# Merkle tree介绍

## 简介

**默克尔树，Merkle Tree可以看做Hash List的泛化（Hash List可以看作一种特殊的Merkle Tree，即树高为2的多叉Merkle Tree）。**

**在最底层，和哈希列表一样，我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做Merkle Root。**

**在p2p网络下载网络之前，先从可信的源获得文件的Merkle Tree树根。一旦获得了树根，就可以从其他从不可信的源获取Merkle tree。通过可信的树根来检查接受到的Merkle Tree。如果Merkle Tree是损坏的或者虚假的，就从其他源获得另一个Merkle Tree，直到获得一个与可信树根匹配的Merkle Tree。**

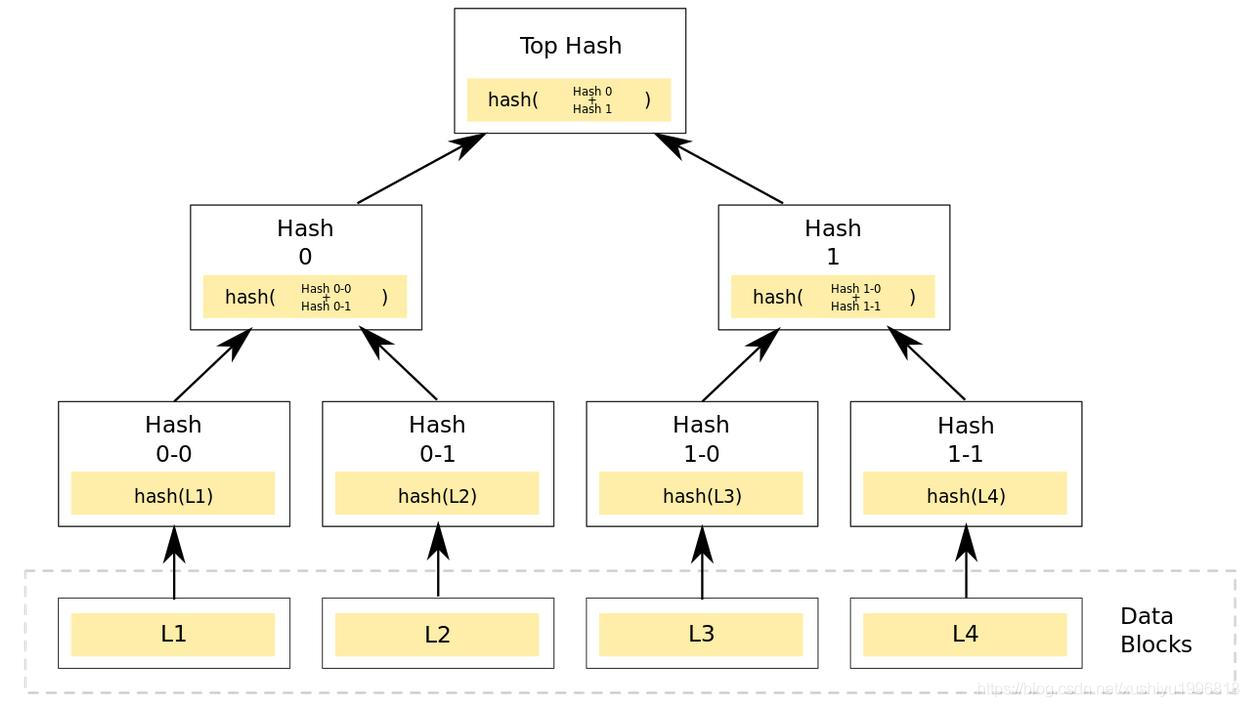


图 1 Merkle Tree

**Merkle Tree和Hash List的主要区别是，可以直接下载并立即验证Merkle Tree的一个分支。因为可以将文件切分成小的数据块，这样如果有一块数据损坏，仅仅重新下载这个数据块就行了。如果文件非常大，那么Merkle tree和Hash list都很到，但是Merkle tree可以一次下载一个分支，然后立即验证这个分支，如果分支验证通过，就可以下载数据了。而Hash list只有下载整个hash list才能验证。**

## ****特点****

**MT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；**

**Merkle Tree的叶子节点的value是数据集合的单元数据或者单元数据HASH。默克尔树的基础数据不是固定的，想存什么数据由你说了算，因为它只要数据经过哈希运算得到的hash值。**

**非叶子节点的value是根据它下面所有的叶子节点值，然后按照Hash算法计算而得出的。默克尔树是从下往上逐层计算的，就是说每个中间节点是根据相邻的两个叶子节点组合计算得出的，而根节点是根据两个中间节点组合计算得出的，所以叶子节点是基础**

# Merkle tree实现

编程语言：C++

编程环境：ubutu20.04, g++9.3.0

## Merkle Node

struct Node {

    std::string hash;

    Node \*left;

    Node \*right;

    Node(std::string data);

};

## 哈希方法

本文使用picosha2封装的方法实现哈希方法，只需要调用接口中的hash256\_hex\_string(string src\_str)即可，具体实现原理这里不再赘述。为了方便用户和保证底层实现程序的封装性，我又重新封装了该方法，用户调用hash\_sha256(string src\_str)即可。

inline std::string hash\_sha256(std::string src\_str) {

    std::string hash\_hex\_str = picosha2::hash256\_hex\_string(src\_str);

    return hash\_hex\_str;

}

## Merkle Tree

封装在类MerkleTree中，该类包含根节点root和一系列方法，比如创建Merkle树、打印Merkle树、删除Merkle树、增加、删除、修改、查找节点等。

class MerkleTree {

Node\* root;

    MerkleTree(const std::vector<std::string>& blocks);

    ~MerkleTree();

    void printMerkleTree();

    void deleteTree(Node \*n, bool deleaf);

    // 判断某一节点是否在merkele-tree中，传递该数据的hash值

    bool containsNode(const std::string& data);

    Node\* getErrorNode(Node\* root2); // 找出root1和root2中不相同的节点   这个感觉没啥用

    void insertNode(const std::string& data);//插入节点

    void deleteNode(const std::string& data);//删除节点

private:

    void printTree(Node \*n, int indent);

    std::vector<Node\*> getLeaves(); // 获取叶节点的列表

    void createMerkleTree(std::vector<Node\*> blocks);// 创建Merkle树

};

## 实现结果

在main函数中测试实现的Merkle Tree，依次创建Merkle Tree、打印树结构、查找节点、插入和删除节点，最后删除Merkle Tree，具体运行图如下：

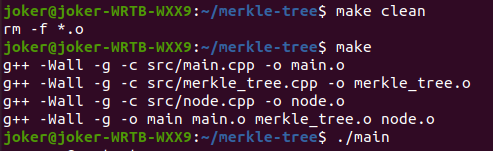


图 2 运行程序

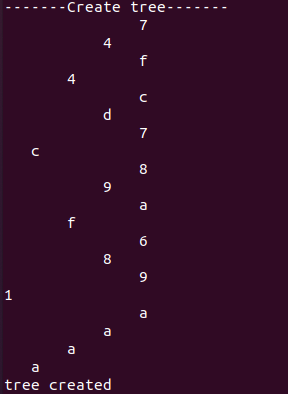


图 3 创建并打印（只打印了首字符）

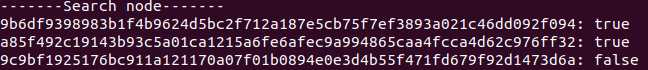


图 4 查找节点

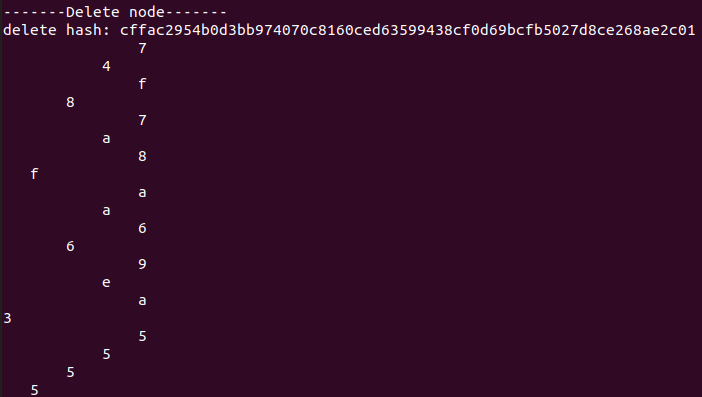
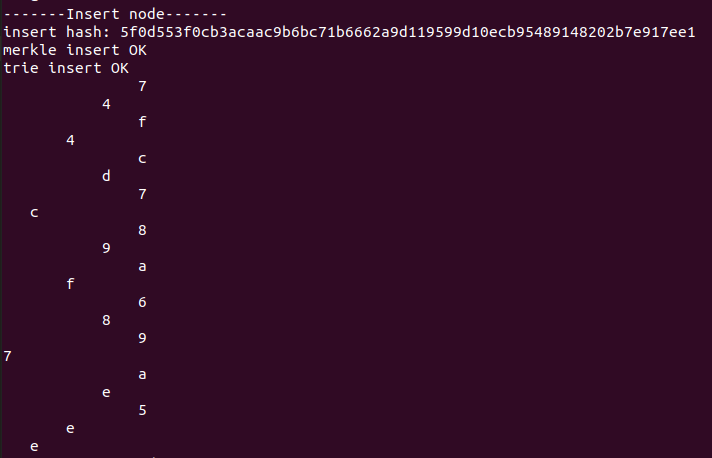


图 5 插入和删除节点

# 优化Merkle tree的出发点

Merkle Tree查找的效率仍然不够高，Merkle Tree的结点并不是按序排列的，所以遍历时需要采用层次遍历的方法查找相应的叶子结点，时间复杂度为O(n)，这种遍历的方法效率较低，为了查找更加快捷，引入一棵字典树，在对Merkle Tree进行操作的同时，维护这棵字典树。引入字典树，可以将时间复杂度降低到O(1),因为通过哈希之后的哈希字符串的最大长度为32，且字典树有序，所以查找时间复杂度为O(1)。

# 优化方案

在对Merkle Tree进行操作的同时维护一棵字典树，字典树根据哈希值进行排列，由于哈希之后的值用16进制的字符表示，即32个’0’-‘f’的字符串。

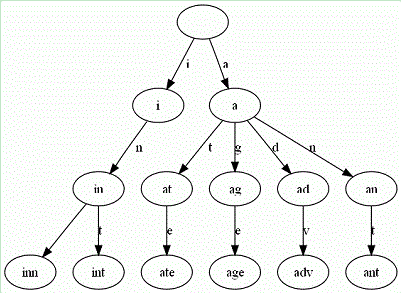


图 6 字典树

## Trie Node

每个节点都维护一个孩子节点数组child（SIZE=16），因为子字符串的首字符有16种可能的情况。

struct TrieTreeNode {

    char val;

    bool isEnd;

    int childCnt;

    TrieTreeNode \*child[SIZE];

    TrieTreeNode(char \_val);

};

## Trie Tree

为了减少程序的复杂性，将字典树的有关方法合并到Merkle Tree的类中。

class MerkleTree {

public:

// MerkleTree

   …

    // TrieTree

    TrieTreeNode \*trieroot;

    void Insert(const std::string& word); // 插入节点

    bool Remove(TrieTreeNode \*treeNode, const std::string& word,int pos, int n);// 删除节点

    bool Find(const std::string& word); // 查找结点

    void LevelOrderTraverse(); //层次遍历

    void BuildTrieTree(const std::vector<std::string>& words); // 创建字典树

    void PreOrderTraverse(); // 前序遍历

    void PostOrderTraverse(); // 后序遍历

private:

  …

    void PreTraverse(TrieTreeNode \*treeNode);

    void PostTraverse(TrieTreeNode \*treeNode);

    void MakeEmpty(TrieTreeNode \*treeNode);

};

# 优化效果

## 实验方案

分别插入100、1000、10000、100000、1000000个节点，然后分别计算各自情况下Merkle Tree、Tire Tree的创建时间和查找10%节点元素的平均时间（比如10000个节点查找其中100个节点，然后计算平均值），然后计算出改进前后Creation的Slowdown和Search Speedup值。

## 实验结果

得到的结果如下图所示，其中*Create Merkle Time*和*Create Tire Time*是创建Merkle树和Tire树的总时间，*Search Merkle Tree*和*Search Tire Tree*是放大了100倍的查找一个节点的平均时间：

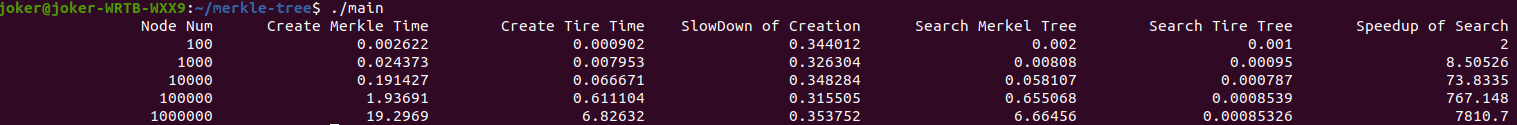
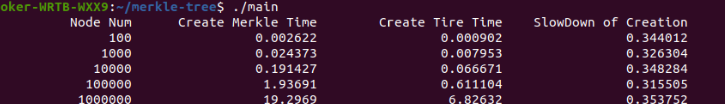


图 7 实验结果



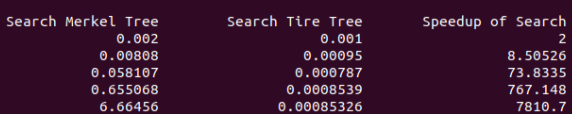


图 8 优化结果（放大）

通过上图中的可以看到，通过引入字典树，当节点数量增加时， Merkle Tree由于需要层次遍历导致时间相应增长，而字典树的查找时间永远是O(1)，查找时间基本不变。因此*Speedup of Search = Search Merkle Tree*/*Search Tire Tree*也相应成反比减少，课件引入字典树对查找效率的提升是十分可观的，当节点达到1,000,000时，查找的效率甚至提升了将近8000倍。

当然，字典树的引入除了空间上的开销外，还带来了创建相应树的时间的增长，通过上图中的*SlowDown of Creation*，可以看到Merkle树带来的创建时间上的代价大概是原来的1.3倍。

总体来说，通过较小的空间开销和创建时间开销的代价，可以带来搜索时间的巨幅提升，在Merkle树的结构中维护一个字典树是很有价值的。