

華中科技大學

課程報告

課程名稱：區塊鏈技術與應用

專業班級：計算機 2020 級 11 班

學 號：M202073600

姓 名：周金偉

指導教師：肖江

報告日期：2020 年 12 月 24 日

計算機科學與技術學院

目录

1 Merkle tree 介绍.....	3
1.1 简介	3
1.2 特点	4
2 Merkle tree 实现.....	4
2.1 Merkle Node.....	4
2.2 哈希方法	4
2.3 Merkle Tree	5
2.4 实现结果	5
3 优化 Merkle tree 的出发点	6
4 优化方案	7
4.1 Trie Node	7
4.2 Trie Tree	7
5 优化效果	8
5.1 实验方案	8
5.2 实验结果	8

1 Merkle tree 介绍

1.1 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。

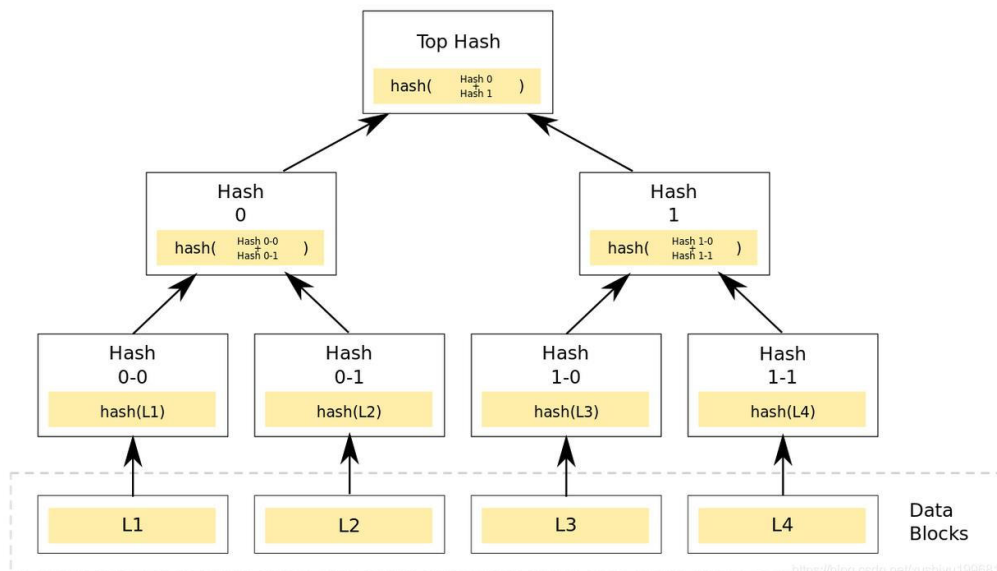


图 1 Merkle Tree

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。

1.2 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

2 Merkle tree 实现

编程语言：C++

编程环境：ubutu20.04, g++9.3.0

2.1 Merkle Node

```
struct Node {  
    std::string hash;  
    Node *left;  
    Node *right;  
    Node(std::string data);  
};
```

2.2 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;  
}
```

```
}
```

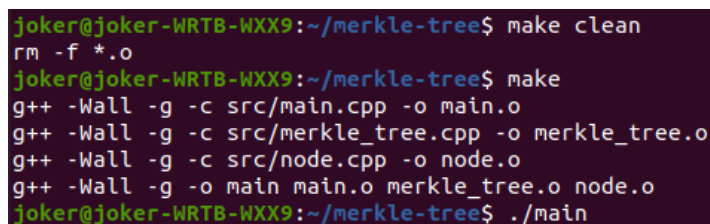
2.3 Merkle Tree

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
};
```

2.4 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:



```
joker@joker-WRTB-WXX9:~/merkle-tree$ make clean
rm -f *.o
joker@joker-WRTB-WXX9:~/merkle-tree$ make
g++ -Wall -g -c src/main.cpp -o main.o
g++ -Wall -g -c src/merkle_tree.cpp -o merkle_tree.o
g++ -Wall -g -c src/node.cpp -o node.o
g++ -Wall -g -o main main.o merkle_tree.o node.o
joker@joker-WRTB-WXX9:~/merkle-tree$ ./main
```

图 2 运行程序

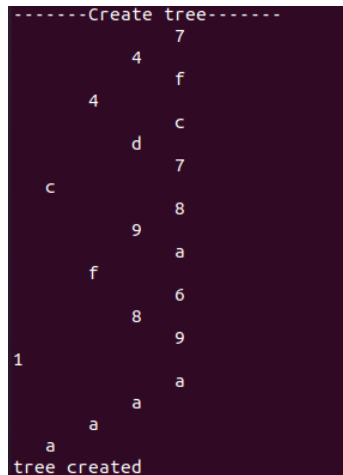


图 3 创建并打印（只打印了首字符）

```
-----Search node-----
9b6df9398983b1f4b9624d5bc2f712a187e5cb75f7ef3893a021c46dd092f094: true
a85f492c19143b93c5a01ca1215a6fe6afec9a994865caa4fcca4d62c976ff32: true
9c9bf1925176bc911a121170a07f01b0894e0e3d4b55f471fd679f92d1473d6a: false
```

图 4 查找节点



图 5 插入和删除节点

3 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 $O(1)$ 。

4 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。

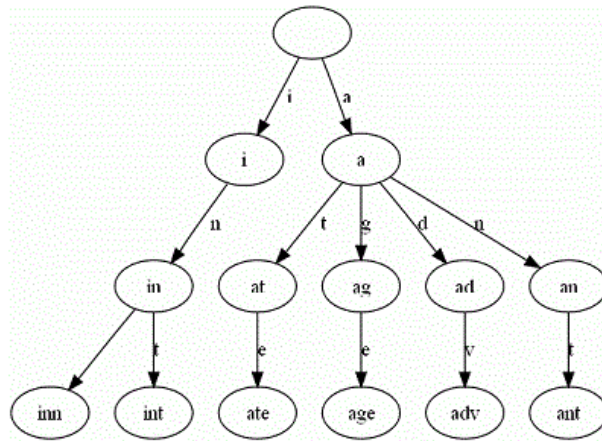


图 6 字典树

4.1 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);
};
```

4.2 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);
};

```

5 优化效果

5.1 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。

5.2 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：

Node Num	Create Merkle Time	Create Tire Time	SlowDown of Creation	Search Merkle Tree	Search Tire Tree	Speedup of Search
100	0.002622	0.000902	0.344012	0.002	0.001	2
1000	0.024373	0.007953	0.326304	0.00808	0.00095	8.50526
10000	0.191427	0.066671	0.348284	0.058107	0.000787	73.8335
100000	1.93691	0.611104	0.315505	0.655068	0.0008539	767.148
1000000	19.2969	6.82632	0.353752	6.66456	0.00085326	7810.7

图 7 实验结果

Node Num	Create Merkle Time	Create Tire Time	SlowDown of Creation
100	0.002622	0.000902	0.344012
1000	0.024373	0.007953	0.326304
10000	0.191427	0.066671	0.348284
100000	1.93691	0.611104	0.315505
1000000	19.2969	6.82632	0.353752

Search Merkle Tree	Search Tire Tree	Speedup of Search
0.002	0.001	2
0.00808	0.00095	8.50526
0.058107	0.000787	73.8335
0.655068	0.0008539	767.148
6.66456	0.00085326	7810.7

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。