research report on MPT

摘要

MPT 是基于 **Merkle Trie** 和前缀树实现的，本文从 **Merkle Trie** 和前缀树这两方面开始介绍了 **MPT** 的实现以及性质等相关内容。

目录

一：Merkle Trie

1.1 概念介绍：

Merkle Trie 是一种基于梅克尔树（Merkle Tree）的数据结构，它是密码学和区块链中常用的一种技术。梅克尔树是一种二叉树，其中每个非叶节点都是其子节点的哈希值的哈希，而每个叶节点包含了一份原始数据。这种树的构建方式使得可以高效地验证数据的完整性。

1.2 Python 实现中变量和函数的作用

<1> calculate_hash(data): 这个函数用于计算给定数据的哈希值。在实际应用中，可能会使用更强大的哈希算法，如 SHA-256。
<2> MerkleTrieNode: 这是梅克尔树节点的类定义。每个节点包含数据、哈希值以及左右子节点。
<3> build_merkle_tree(data_list): 这个函数用于构建 Merkle Trie。它接受一个数据列表作为输入，并返回根节点，即 Merkle Trie 的根。

1.3 Python 实现示例：

下图代码只实现了基本的 Merkle Trie 的功能。

```
import hashlib

def calculate_hash(data):
    # 使用 SHA-256 哈希算法计算数据的哈希值
    return hashlib.sha256(data.encode()).hexdigest()

class MerkleTrieNode:
    def __init__(self, data):
        self.data = data
        self.hash_value = calculate_hash(data)
        self.left_child = None
        self.right_child = None

def build_merkle_tree(data_list):
    nodes = [MerkleTrieNode(data) for data in data_list]

    while len(nodes) > 1:
        new_level = []
        for i in range(0, len(nodes), 2):
            node1 = nodes[i]
```

```
            node2 = nodes[i + 1] if i + 1 < len(nodes) else node1
            combined_data = node1.data + node2.data
            combined_hash = calculate_hash(combined_data)

            new_node = MerkleTrieNode(combined_data)
            new_node.left_child = node1
            new_node.right_child = node2
            new_node.hash_value = combined_hash

            new_level.append(new_node)

        nodes = new_level

    return nodes[0]
```

1.4 在密码学中的应用：

Merkle Trie 在密码学中有广泛的应用，特别是在区块链技术中。它用于验证大量数据的完整性，使得区块链数据的安全和高效验证成为可能。

在区块链中，每个区块包含了多个交易的数据。这些交易数据构成了一个 Merkle Trie。区块头中的 Merkle 根哈希是这个 Merkle Trie 的根节点的哈希值，通过它可以验证整个区块中的所有交易是否未被篡改。这种设计使得区块链的数据具有高度的不可篡改性和透明性。

除了区块链之外，Merkle Trie 还可以用于其他需要验证数据完整性的场景，例如 P2P 网络、分布式存储系统等。

二：前缀树

2.1 概念介绍：

前缀树，是一种用于存储和快速查找字符串的树状数据结构。前缀树的特点是将共同的前缀部分合并在一起，从而节省了存储空间和查找时间。它常用于处理字符串的自动补全、拼写检查、字典查找等场景。

2.2 变量和函数的作用：

<1> TrieNode: 前缀树节点的类定义。
每个节点包含一个子节点字典 children 和一个布尔变量 is_end_of_word，用于标记该节点

是否是一个单词的结尾。

<2> Trie: 前缀树的类定义。它包含一个根节点 root，并提供了插入单词、查找单词和查找前缀的方法。
<3> insert(word): 将一个单词插入到前缀树中。
<4> search(word): 查找给定的单词是否在前缀树中。
<5> starts_with(prefix): 查找是否有以给定前缀开头的单词在前缀树中。

2.3Python 实现示例：

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

2.4 在 Merkle Patricia Trie 中的应用：

Merkle Patricia Trie 是一种基于前缀树的数据结构，用于在以太坊（Ethereum）区块链中存储账户和账户状态。它是 Merkle 树的变种，其中 Merkle 树中的每个节点都是一个前缀树，用于存储不同的键值对。

在 Merkle Patricia Trie 中，每个节点都代表一个字符串键值对。树的叶节点包含实际的键值对，而内部节点包含根据其子节点的内容计算得出的哈希值。这样的设计使得可以高效地验证树中特定数据是否存在以及完整性。

Merkle Patricia Trie 在以太坊中用于存储账户和账户状态。以太坊中的每个账户都有一个唯一的地址，Merkle Patricia Trie 将这些地址映射到账户的状态数据，如账户余额、合约代码等。通过使用 Merkle Patricia Trie，以太坊可以高效地查找和验证账户状态，同时确保数据的完整性和安全性。

三：Merkle Patricia Trie

3.1 概念介绍：

Merkle Patricia Trie（MPT）是一种基于前缀树（Trie）和 Merkle 树的数据结构，在密码学和区块链领域中得到广泛应用。MPT 是以太坊（Ethereum）区块链中用于存储账户和账户状态的核心数据结构。

MPT 使用了前缀树的优势，将具有共同前缀的键值对合并，从而节省了存储空间。同时，它还利用了 Merkle 树的特性，通过哈希值验证确保数据的完整性和安全性。这使得 MPT 能够高效地存储大量数据，并且能够进行高效的数据验证和查找操作。

3.2 函数变量解释

<1> calculate_hash(data) 函数：

这个函数用于计算给定数据的哈希值，这里使用 SHA-256 哈希算法来进行计算。SHA-256 是一种常用的密码学哈希算法，它可以将任意长度的数据映射成一个固定长度的哈希值（通常是 256 位，即 32 字节）。这个哈希值在 Merkle Patricia Trie 中用于验证数据的完整性和安全性。

<2> MerkleTrieNode 类：

这是 Merkle Trie 节点的类定义。每个节点包含以下属性：

data：节点保存的数据，对于叶节点来说，数据是实际的字符串数据；对于内部节点来说，数据是经过哈希计算后的键值对。

is_leaf：布尔变量，用于标识该节点是否为叶节点。叶节点表示存储真实的数据，而内部节点表示存储经过哈希计算的键值对。

hash_value：保存节点的哈希值。对于叶节点，哈希值是对实际数据计算得到的；对于内部节点，哈希值是对左右子节点哈希值的哈希计算得到的。

children：节点的子节点字典，用于保存该节点的左右子节点（内部节点）。对于叶节

点，子节点字典为空，因为它们没有子节点。

<3>build_merkle_patricia_trie(data_list) 函数：

这个函数用于构建 Merkle Patricia Trie。它接受一个数据列表 data_list 作为输入，并返回根节点，即 MPT 的根。

首先，将数据列表中的每个数据创建为 Merkle Trie 的叶节点，因为叶节点表示存储真实的数据。

然后，通过不断合并节点构建 Merkle Patricia Trie。首先将所有叶节点作为初始节点列表 nodes。

迭代直到只剩下一个节点。在每次迭代中，每两个节点合并成一个新的内部节点，并对这两个节点的数据进行哈希计算。新的内部节点作为下一次迭代的节点列表。

最后，返回最终的根节点，这就是 MPT 的根。

3.3 部分代码实现

Merkle Trie 和 Merkle Patricia Trie 在代码层面上有一些区别，尤其是在节点的设计和哈希计算上。Merkle Patricia Trie 的节点会包含更多的信息，并且使用特定的哈希计算方式。

```
import hashlib

def calculate_hash(data):
    # 使用 SHA-256 哈希算法计算数据的哈希值
    return hashlib.sha256(data.encode()).hexdigest()

class MerkleTrieNode:
    def __init__(self, data, is_leaf=False):
        self.data = data
        self.is_leaf = is_leaf
        self.hash_value = None
        self.children = {}

def build_merkle_patricia_trie(data_list):
    nodes = [MerkleTrieNode(data, is_leaf=True) for data in data_list]

    while len(nodes) > 1:
        new_level = []
        for i in range(0, len(nodes), 2):
            node1 = nodes[i]
            node2 = nodes[i + 1] if i + 1 < len(nodes) else node1
            combined_data = node1.data + node2.data
            combined_hash = calculate_hash(combined_data)

            new_node = MerkleTrieNode(combined_data)
            new_node.children['left'] = node1
            new_node.children['right'] = node2
            new_node.hash_value = combined_hash
```

```
                new_node.is_leaf = False

                new_level.append(new_node)

            nodes = new_level

    return nodes[0]
```

3.4 在密码学中的应用

Merkle Patricia Trie 在密码学中主要用于实现以太坊区块链中的账户和账户状态存储。以太坊中的每个账户都有一个唯一的地址，MPT 将这些地址映射到账户的状态数据，如账户余额、合约代码等。

MPT 的设计使得可以高效地验证区块链数据的完整性。每个区块包含了多个交易的数据，这些交易数据构成了一个 MPT。区块头中的 Merkle 根哈希是这个 MPT 的根节点的哈希值，通过它可以验证整个区块中的所有交易是否未被篡改。这保障了区块链数据的安全性和不可篡改性。

3.5 Merkle Patricia Trie 的相关性质

<1> 前缀合并： MPT 使用前缀树的数据结构，它将具有共同前缀的键值对合并在一起，从而节省存储空间和提高存储效率。

<2> 唯一性保证： 每个节点的哈希值是根据其数据和子节点的哈希值计算得出的，因此每个节点的哈希值都是唯一的，从而确保数据的唯一性。

<3> 默克尔证明： MPT 的每个节点都包含一个哈希值，这些哈希值构成了一种"默克尔证明"，允许在未完全揭示整个树的情况下验证树上的特定数据。这对于数据完整性的验证非常有用。

<4> 不可变性： 由于哈希值是由节点的数据和子节点的哈希值计算得出的，一旦 MPT 构建完成后，数据和结构是不可变的。任何尝试更改数据或结构的操作都会导致哈希值的变化，从而证明数据被篡改。

<5> 高效查询： 由于 MPT 使用了前缀树的结构，它可以在 O(log n)的时间内高效地查询数据，其中 n 是树中的节点数。这使得在区块链中查找账户状态等信息变得非常高效。