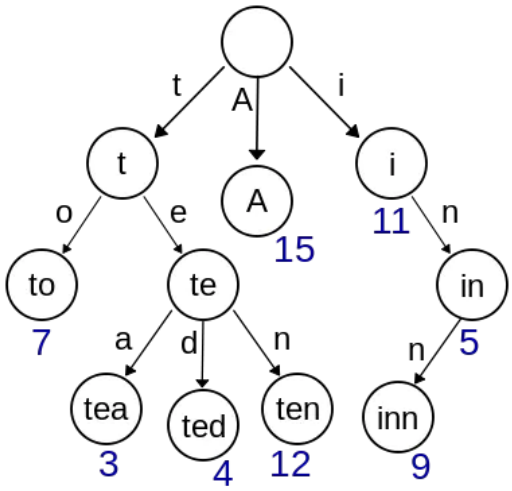
前言

Merkle Patricia Trie（简称MPT）是以太坊区块链中的核心数据结构，其用于处理用户的交易数据，收据数据，状态数据和存储数据，这些功能分别对应值以太坊区块链和用户信息中心存储的4颗MPT。

基础概念

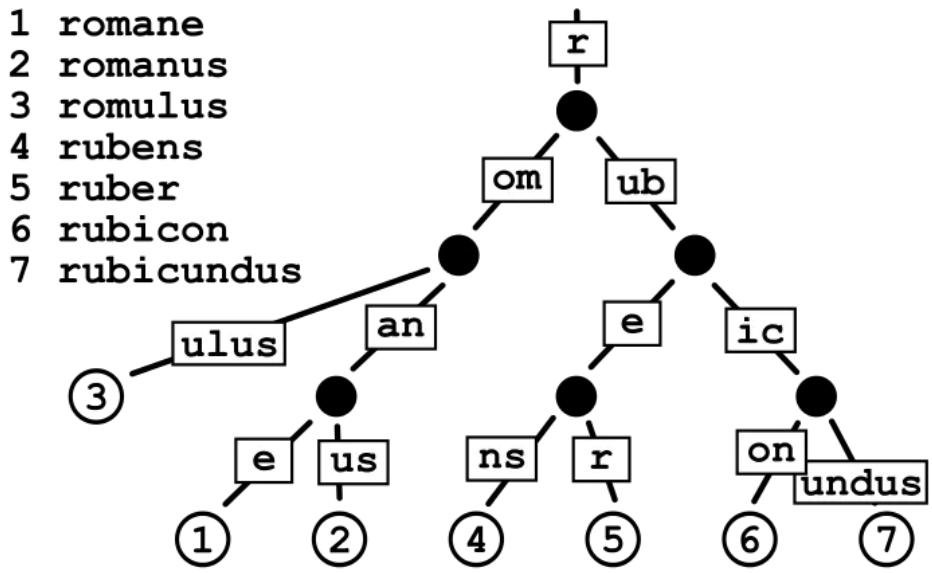
MPT树是一种对基础的数据结构——Trie树的扩展。

Trie树是将字符串按照从前到后的顺序进行构建属性结构的过程，也叫前缀树。通过这样的构建，对于制定长度上限的字符串，可以以常数上限时间内进行字符串哈希，并且完全不用担心哈希碰撞，因为每个value都对应着树结构的节点。另外，同步搜索前缀相同的字符串速度会大大提升，因为搜索节点可以从共同前缀下的节点，而不是根节点开始，图示如下：



> 图源：https://www.jianshu.com/p/a4c3d8bf0b05

而trie树的缺点是,如果存在大量前缀相似度低的字符串，树就会变得极其臃肿，并且单位存储空间的有效存储会下降。Patricia Trie是对trie树的改进，其思路也较为简单，就是将不重叠的字符串前序序列折叠成为一个树节点，这样一来树结构的查询性能不会因稀疏字符串明显下降，图示如下：



> 图源：https://www.jianshu.com/p/a4c3d8bf0b05

而MPT作为PT的扩展，用十六进制哈希结果的字符串作为PT的索引（这也意味着一般意义上的MPT的度是16），并且用PT 整合了Merkle Hash的功能，使得查询和处理记录可以在保证安全性的前提下保持高效。

以下是对以太坊源码中关于MPT实现部分的分析。

源码分析

本次所使用的源码是go语言版本的以太坊源码（似乎也是唯一被维护的版本），地址为https://github.com/ethereum/go-ethereum。关于MPT结构的源码大部分位于go-ethereum/trie目录下：

数据结构

首先分析树整体的数据结构：

```
type Trie struct {
    db   *Database
    root node
    // Keep track of the number leafs which have been inserted since the last
    // hashing operation. This number will not directly map to the number of
    // actually unhashed nodes
    unhashed int
}
```

其中db是作用与以太坊和文件系统中间的一个抽象层，用于周期性地将树内容flush入数据库。而root指示树根，此时研究node类型的构造，发现node是接口类型，声明了转字符串和cache的接口方法：

```
type node interface {
    fstring(string) string
    cache() (hashNode, bool)
}
```

而加下来定义了4中实现类型：

```
// 定义了MPT的四种节点类型,是上述接口的实现定义,因为这四种类型都实现了node声明的方法
type (
    // 代表前缀树的节点
    fullNode struct {
        // 前16个是子节点，最后一个存储值
        Children [17]node // Actual trie node data to encode/decode (needs
custom encoder)
        // 通过find usage发现，nodeFlag变量用于指示DB周期性将node写入文件，可以视作dirty
'bit'
        flags nodeFlag
    }
    // 代表叶子结点，也代表扩展节点
    shortNode struct {
        Key   []byte
        Val   node // 叶子结点和扩展节点的Val值有不同的含义
        flags nodeFlag
    }
    // 哈希结点就是被折叠的节点
    hashNode  []byte
```

```
    // 存储数据的节点
    valueNode []byte
)
```

可以看到，这里定义了4中类型的node，他们的作用和差别会在之后操作MPT的routine中体现。

算法分析

go语言通过首字母大写暴露保重定义的包外可访问方法。

1. 新建MPT的方法：New

```
func New(root common.Hash, db *Database) (*Trie, error) {// 只能从已有的rootnode中创建trie树
   // 要求DB存在
   if db == nil {
      panic("trie.New called without a database")
   }
   // 新建trie变量，首先要指定db
   trie := &Trie{
      db: db,
   }
   // 如果root是zero hash，或是空树的根的hash，就报错
   if root != (common.Hash{}) && root != emptyRoot {
      // 将哈希值通过上述生成的trie中的db的方法构建node
      rootnode, err := trie.resolveHash(root[:], nil)
      if err != nil {
         return nil, err
      }
      // 赋值trie树的根节点
      trie.root = rootnode
   }
   return trie, nil
}
```

其中，emtpyRoot是预定义已知的空字符串的哈希树根。New操作比较简单，就是根据已存在的一个node和db创建一个以此节点为根节点的trie树。

2. 更新方法和对应的插入和删除

实际使用的Update方法简单调用TryUpdate，并在获得非nil的error值时返回错误

```
func (t *Trie) TryUpdate(key, value []byte) error {
   t.unhashed++ // 此次update必然改变一个节点的状态，所以unhashed数增加
   k := keybytesToHex(key)
   if len(value) != 0 { // 要寻找的value不存在，就将其插入trie树
      _, n, err := t.insert(t.root, nil, k, valueNode(value))
      if err != nil {
         return err
      }
      t.root = n
   } else { // 如果存在就将其删除
      _, n, err := t.delete(t.root, nil, k)
      if err != nil {
         return err
```

```
        }
        t.root = n
    }
    return nil
}
```

可以看到，Update方法调用了2个关键的方法：insert和delete

3. 插入方法：insert

源码和大部分行内分析如下：

```
/*
    递归调用函数，将元素插入树的指定位置
    @param n 当前处理节点位置
    @param prefix 当前节点代表的共同前缀
    @param key 哈希值除去共同前缀的剩余前缀
    @param value 要插入的值

    @return 分别指是否成功插入，插入后的根节点，产生的错误
*/
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node,
error) {
    // key长度为0意味着找到了插入位置
    if len(key) == 0 {
        if v, ok := n.(valueNode); ok {
            return !bytes.Equal(v, value.(valueNode)), value, nil
        }
        return true, value, nil
    }
    // 根据节点类型选择策略，这里稍微更换了一下位置
    switch n := n.(type) {

    case *fullNode:
        // 直接递归调用，寻找插入位置
        dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]),
key[1:], value)
        if !dirty || err != nil {
            return false, n, err
        }
        n = n.copy() // 注意这里返回的n应该是原来的n副本而不能是引用，否则可能导致数据错误
        n.flags = t.newFlag()
        n.Children[key[0]] = nn
        return true, n, nil

    case *shortNode: // 表示达到了叶子节点或者扩展结点
        matchlen := prefixLen(key, n.Key)
        // If the whole key matches, keep this short node as is
        // and only update the value.
        if matchlen == len(n.Key) { // 叶子节点key值和给出的key值长度匹配，则说明这个哈
希值存在，更新即可
            // 递归调用，此时传入的n.Val的类型是valueNode
            dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...),
key[matchlen:], value)
            if !dirty || err != nil {
                return false, n, err
            }
```

```
            return true, &shortNode{n.Key, nn, t.newFlag()}, nil
        }
        // Otherwise branch out at the index where they differ.
        // 如果不完全匹配，说明有新的分支产生，而当前的叶子节点也将变为fullNode
        branch := &fullNode{flags: t.newFlag()}
        var err error
        // 以下两步，分别是将原来叶节点存储的key值和将要拆入的新key值分别写成2个叶子节点
        // 并插入到这个节点的Children中，使之成为它们的父节点

        // 将原来的节点取出后缀，剩下部分产生新叶子节点
        _, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix,
n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
        if err != nil {
            return false, nil, err
        }
        // 将新插入的节点取出后缀，剩下部分产生新叶子节点
        _, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix,
key[:matchlen+1]...), key[matchlen+1:], value)
        if err != nil {
            return false, nil, err
        }
        // Replace this shortNode with the branch if it occurs at index 0.
        // 完全不匹配的情况，直接用fullNode替换当前节点
        if matchlen == 0 {
            return true, branch, nil
        }
        // Otherwise, replace it with a short node leading up to the branch.
        // 否则，用指向branch的shortNode替代当前节点，此时这个节点变成了扩展结点
        return true, &shortNode{key[:matchlen], branch, t.newFlag()}, nil


    case nil:
        // 碰到空节点是因为之前的前缀已经被完全遍历，此时key长度仍然不为0，这个时候在当前位置
构建short节点即可
        return true, &shortNode{key, value, t.newFlag()}, nil

    case hashNode:
        // 这表示，访问的到的node不是完整的node，而是一串指向缓存在db中node的hash值
        // 那么要做的就是从db中resolve出这块node，并且在这里继续执行插入操作
        // We've hit a part of the trie that isn't loaded yet. Load
        // the node and insert into it. This leaves all child nodes on
        // the path to the value in the trie.
        rn, err := t.resolveHash(n, prefix)
        if err != nil {
            return false, nil, err
        }
        dirty, nn, err := t.insert(rn, prefix, key, value)
        if !dirty || err != nil {
            return false, rn, err
        }
        return true, nn, nil

    default:
        panic(fmt.Sprintf("%T: invalid node: %v", n, n))
    }
}
```

这里不难分析出，其实fullNode对应的就是传统trie树的节点，是根据16bit字符串位（也即是）Children的前16位；而shortNode除了代表叶子节点外，还代表了Patricia Trie中折叠的路径，也即文中所说的扩展结点。如果Key值不能覆盖整个key，那它其实就代表Patricia Trie折叠的串，如果能，那么它其实就代表叶子节点除去prefix最后的key值。

另外还可以看出，MPT用到了db为中间层的缓存技术。

4. 删除方法：delete

其实了解了MPT的原理，删除方法可以和插入方法对应起来。

源码和大部分行内分析如下：

```
/*
   delete returns the new root of the trie with key deleted.
   It reduces the trie to minimal form by simplifying
   nodes on the way up after deleting recursively.

   @param n 当前处理节点位置
   @param prefix 当前节点代表的共同前缀
   @param key 哈希值除去共同前缀的剩余前缀

   @return 分别指是否成功进行了delete，删除后的根节点，产生的错误

*/
func (t *Trie) delete(n node, prefix, key []byte) (bool, node, error) {
   switch n := n.(type) {
   // 这里shortNode开头更容易理解，因为delete是一个从下到上的过程
   case *shortNode:
      matchlen := prefixLen(key, n.Key)
      if matchlen < len(n.Key) { // 说明无法完全匹配
         return false, n, nil // don't replace n on mismatch
      }
      if matchlen == len(key) { // 正好匹配，而这个node完全被删掉后也变成了nil
         return true, nil, nil // remove n entirely for whole matches
      }
      // The key is longer than n.Key. Remove the remaining suffix
      // from the subtrie. Child can never be nil here since the
      // subtrie must contain at least two other values with keys
      // longer than n.Key.
      // 运行到这其实说明了key比nKey长，n是个扩展结点，还是要找他指向的子节点

      // 在n的子结点中找，而prefix则append上了被n折叠的部分，key去掉了这些部分
      // 这里的...应该是把slice类型解包成可变长参数
      dirty, child, err := t.delete(n.Val, append(prefix, key[:len(n.Key)]...),
key[len(n.Key):])
      if !dirty || err != nil {
         return false, n, err
      }
      // 这里还需要对产生的child再次进行一次判断
      // 原因是，如果删除了fullNode中的一个分支使得fullNode变成了shortNode，
      // 那就要把2个shortNode进行合并
      // 但令人疑惑的是这里难道不能用if-else？？
      switch child := child.(type) {
      case *shortNode:
         return true, &shortNode{concat(n.Key, child.Key...), child.Val,
t.newFlag()}, nil
```

```
        default:
            return true, &shortNode{n.Key, child, t.newFlag()}, nil
        }

    case *fullNode:
        // 在fullNode层面就一次步进一个fullNode进行递归调用，直到达到了其他类型的node
        dirty, nn, err := t.delete(n.Children[key[0]], append(prefix, key[0]),
key[1:])
        if !dirty || err != nil {
            return false, n, err
        }
        // 这里依然要做值拷贝
        n = n.copy()
        n.flags = t.newFlag()
        n.Children[key[0]] = nn

        // When the loop is done, pos contains the index of the single
        // value that is left in n or -2 if n contains at least two
        // values.g

        // 这里也是一个非常重要的检查
        // 当fullNode产生deletion而只有1个子节点时，他就不能称为一个fullNode，
        // 就需要降级变成一个shortNode从而与之前或者之后的node合并
        // 这里英文注释阐述了pos的含义
        // 而正是因为有只有1个子节点时fullNode就会被处理掉，所以fullNode不可能会在deletion后
没有子节点
        pos := -1
        for i, cld := range &n.Children {
            if cld != nil {
                if pos == -1 {
                    pos = i
                } else {
                    pos = -2
                    break
                }
            }
        }


        if pos >= 0 {
            if pos != 16 {// 代表pos指向的是一个，并且是仅存的一个节点
                // If the remaining entry is a short node, it replaces
                // n and its key gets the missing nibble tacked to the
                // front. This avoids creating an invalid
                // shortNode{..., shortNode{...}}.  Since the entry
                // might not be loaded yet, resolve it just for this
                // check.
                cnode, err := t.resolve(n.Children[pos], prefix)
                if err != nil {
                    return false, nil, err
                }
                // 如果子节点是一个shortNode，就把这个子节点和当前降级成shortNode的节点合并成
新的shortNode
                if cnode, ok := cnode.(*shortNode); ok {
                    k := append([]byte{byte(pos)}, cnode.Key...)
                    return true, &shortNode{k, cnode.Val, t.newFlag()}, nil
                }
            }
```

```
            // 如果剩下的是这个fullNode存储的值，那就直接把它变成shortNode
            return true, &shortNode{[]byte{byte(pos)}, n.Children[pos],
t.newFlag()}, nil
        }
        // n still contains at least two values and cannot be reduced.
        return true, n, nil

    case valueNode:// 找到valueNode直接删除
        return true, nil, nil

    case nil:// 找不到删除不成功
        return false, nil, nil

    case hashNode:
        // hashNode也是类似的情况
        rn, err := t.resolveHash(n, prefix)
        if err != nil {
            return false, nil, err
        }
        dirty, nn, err := t.delete(rn, prefix, key)
        if !dirty || err != nil {
            return false, rn, err
        }
        return true, nn, nil

    default:
        panic(fmt.Sprintf("%T: invalid node: %v (%v)", n, n, key))
    }
}
```

可以看到，删除大体意义上和插入是逆过程，但仍然采取了一系列的算法机制，保证了只有一个子节点的fullNode会被降级成shortNode，连续的shortNode也会被合并，算法设计非常值得学习。

5. 取值方法：tryGet

暴露出的最终方法Get最终调用了包内方法tryGet，后者的源码和大部分行内分析如下：

```
/**
    从MPT中取出数据
    @param origNode 取值的起始处
    @param key 从node处开始要取值的key
    @param pos 跳过prefix，真正的key值开始的位置，这里没有使用prefix

    @return value 取出的值
    @return newnode 取值后的新root（可能因为有hashNode的存在，所以取个值也可能让root变化）
    @return didResolve 是否从db中resolve出值，与newnode共同使用
    @return err 错误
 */
func (t *Trie) tryGet(origNode node, key []byte, pos int) (value []byte, newnode
node, didResolve bool, err error) {
    switch n := (origNode).(type) {
    case nil:
        return nil, nil, false, nil
    case valueNode:// 碰到了valueNode，说明找到了要取的值，将这个node返回
        return n, n, false, nil
    case *shortNode:
        // 如果剩下用于索引的key值长度比n.Key小，那么必然出错
```

```
        // 如果key值对应的位置的n.Key不完全匹配，也说明了必然找不到
        if len(key)-pos < len(n.Key) || !bytes.Equal(n.Key,
key[pos:pos+len(n.Key)]) {
            // key not found in trie
            return nil, n, false, nil
        }
        // 继续递归寻找
        value, newnode, didResolve, err = t.tryGet(n.Val, key, pos+len(n.Key))
        if err == nil && didResolve {// 如果有newnode产生，就把当前的node备份一次，并将信
息返回
            n = n.copy()
            n.Val = newnode
        }
        return value, n, didResolve, err
    case *fullNode:
        // fullNode操作与之前类似
        value, newnode, didResolve, err = t.tryGet(n.Children[key[pos]], key,
pos+1)
        if err == nil && didResolve {
            n = n.copy()
            n.Children[key[pos]] = newnode
        }
        return value, n, didResolve, err
    case hashNode:
        // 遇到了hashNode，就在递归完成后返回不是nil的newnode
        child, err := t.resolveHash(n, key[:pos])
        if err != nil {
            return nil, n, true, err
        }
        value, newnode, _, err := t.tryGet(child, key, pos)
        return value, newnode, true, err
    default:
        panic(fmt.Sprintf("%T: invalid node: %v", origNode, origNode))
    }
}
```

6. 拾遗与其他

通过查阅资料可以得知，对于shortNode：

```
shortNode struct {
    Key   []byte
    Val   node // 叶子结点和扩展节点的Val值有不同的含义
    flags nodeFlag
}
```

如果它是当前MPT树的叶子结点，那么Val字段实际上就指向valueNode，而这个valueNode和其他valueNode中存储的字节序列，是名为RLP的数据编码，它用于MPT数据库的序列化和反序列化，也就代表着数据本身。

总结

上述分析的其实主要是Patricia Trie这一数据结构的routine，而其中Merkle的含义则是，每个valueNode在MPT中从下到上的哈希值，都是一个以太坊区块头部中的根Merkle Hash，这就建立了根Merkle Hash到区块数据记录的映射关系。以太坊通过在运行环境和用户账户端分别建立存储数据的MPT树，就构成了分布式存储和验证的体系。

如此一来，一方面，对交易信息的查询就可以归结为对区块的查询，再归结为MPT上的检索，而它理论上是O(1)的复杂度，所以这样的查询可以一定程度上保证效率；另一方面，使用Merkle Hash，而不是每个区块的实际物理地址，能够很好地保存安全性，之前的实验也进行过验证，区块信息的小幅度修改就会引起Merkle Hash的变化，这使得对区块链结构进行篡改的行为无法通过MPT定位到正确的区块，保证了安全性。