Lab2: 以太坊 MPT 源代码分析

> 学号：3190104783
> 姓名：欧翌昕
> 日期：2021 年 10 月 12 日

在以太坊源代码(https://github.com/ethereum/go-ethereum)的包 trie 中实现了Merkle Patricia Tries (MPT)。MPT 是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构，实际上是三种数据结构的组合，分别是 Trie 树， Patricia Trie，和 Merkle 树。

每一个以太坊的区块头包含三颗 MPT 树，分别是：

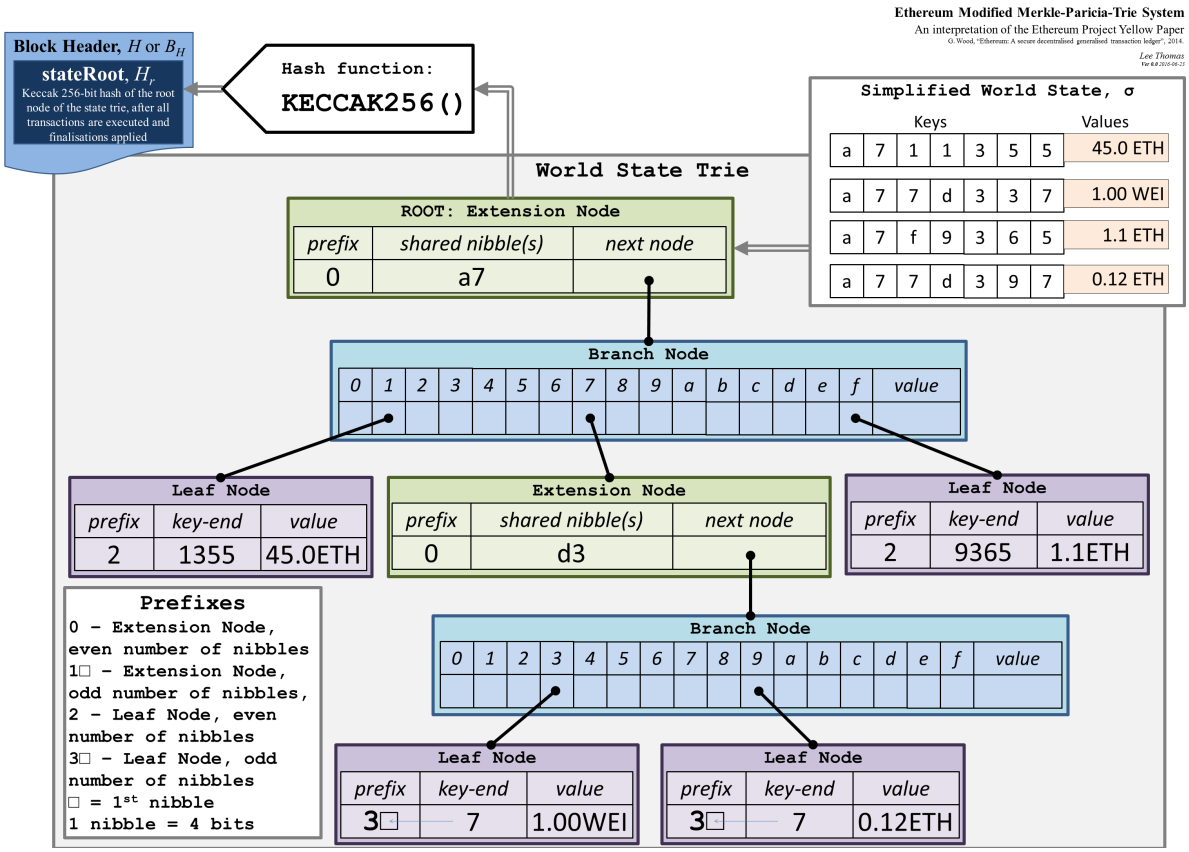
- 交易树：记录交易的状态和变化。每个块都有各自的交易树，且不可更改
- 收据树：记录交易执行过程中的一些数据
- 状态树：记录账号中各种状态信息

MPT 的数据结构

在 MPT 中存在四种节点：

- **空节点**：表示空，在代码中用空串实现
- **叶子节点(Leaf)**：包含两个字段， 第一个字段 Key 是关键字的一种特殊十六进制编码，第二个字段是 Value 是 Value 的 RLP 编码
- **扩展节点(Extention)**：也是 [key, value] 的一个键值对，但是这里的 value 是其他节点的 hash 值，通过 hash 可以链接到其他节点
- **分支节点(Branch)**：分支节点包含了 17 个字段，其前 16 个字段对应于这些点在其遍历中的键的十六个可能的半字节值中的每一个，第17个字段是存储那些在当前节点结束了的节点

MPT 的结构大致如下图所示：



在以太坊源代码中，Go 语言实现的`node`数据结构如下：

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

type (
  fullNode struct {
    Children [17]node // Actual trie node data to encode/decode (needs custom encoder)
    flags    nodeFlag
  }
  shortNode struct {
    Key   []byte
    Val   node
    flags nodeFlag
  }
  hashNode  []byte
  valueNode []byte
)
```

可以看到 node 一共有 4 种类型, 其中：

- `fullNode`对应形式化定义中的分支节点

- `shortNode` 对应形式化定义中的扩展节点和叶子节点，需要通过 `shortNode.Val` 的类型来判断当前节点是叶子节点还是拓展节点, 的类型表示当前节点是叶子节点还是扩展节点
  - `shortNode.Val` 为 `valueNode` 表示当前节点是叶子节点
  - `shortNode.Val` 指向下一个 node 表示当前节点是扩展节点

MPT 节点有个 `nodeFlag` 字段，用来记录一些辅助数据：

- 节点哈希：若该字段不为空，则当需要进行哈希计算时，可以跳过计算过程而直接使用上次计算的结果（当节点变脏时，该字段被置空）
- 脏标志：当一个节点被修改时，该标志位被置为1

```
// nodeFlag contains caching-related metadata about a node.
type nodeFlag struct {
  hash  hashNode // cached hash of the node (may be nil)
  dirty bool     // whether the node has changes that must be written to the database
}
```

Go 语言实现的 `Trie` 数据结构如下：

```
// Trie is a Merkle Patricia Trie.
// The zero value is an empty trie with no database.
// Use New to create a trie that sits on top of a database.
//
// Trie is not safe for concurrent use.
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：

- `root` 是当前的 root 节点
- `db` 是后端的 KV 储存

MPT 的操作流程

Trie 树的初始化

Go 语言实现的 Trie 树初始化函数如下：

```
// New creates a trie with an existing root node from db.
//
// If root is the zero hash or the sha3 hash of an empty string, the
// trie is initially empty and does not require a database. Otherwise,
// New will panic if db is nil and returns a MissingNodeError if root does
// not exist in the database. Accessing the trie loads nodes from db on demand.
func New(root common.Hash, db *Database) (*Trie, error) {
  if db == nil {
    panic("trie.New called without a database")
  }
  trie := &Trie{
    db: db,
  }
  if root != (common.Hash{}) && root != emptyRoot {
    rootnode, err := trie.resolveHash(root[:], nil)
    if err != nil {
      return nil, err
    }
    trie.root = rootnode
  }
  return trie, nil
}
```

Trie 树的初始化调用`New()`函数，函数接受一个 hash 值和一个 Database 参数，如果 hash 值不是空值，则说明是从数据库加载一个已经存在的 Trie 树， 就调用`trie.resolveHash()`方法来加载整颗 Trie 树，如果 root 是空，则新建一颗 Trie 树返回。

Trie 树的插入

Go 语言实现的 Trie 树插入函数如下：

```
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
  if len(key) == 0 {
    if v, ok := n.(valueNode); ok {
      return !bytes.Equal(v, value.(valueNode)), value, nil
    }
    return true, value, nil
  }
  switch n := n.(type) {
  case *shortNode:
    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) {
      dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchle
n:], 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.
    branch := &fullNode{flags: t.newFlag()}
    var err error
    _, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix, n.Key[:mat
chlen+1]...), n.Key[matchlen+1:], n.Val)
    if err != nil {
      return false, nil, err
    }
    _, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix, key[:matchle
n+1]...), key[matchlen+1:], value)
    if err != nil {
      return false, nil, err
    }
    // Replace this shortNode with the branch if it occurs at index 0.
    if matchlen == 0 {
      return true, branch, nil
    }
    // Otherwise, replace it with a short node leading up to the branch.
    return true, &shortNode{key[:matchlen], branch, t.newFlag()}, nil

  case *fullNode:
    dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]), key[1:], va
lue)
    if !dirty || err != nil {
      return false, n, err
    }
    n = n.copy()
    n.flags = t.newFlag()
    n.Children[key[0]] = nn
    return true, n, nil

  case nil:
    return true, &shortNode{key, value, t.newFlag()}, nil

  case hashNode:
    // 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))
  }
}
```

Trie 树的插入是一个递归调用的方法，从根节点开始往下找，直到找到可以插入的点进行插入操作。参数`node`是当前插入的节点，`prefix`是当前已经处理完的部分 key，`key`是还没有处理玩的部分key。`value`是待插入的值。 返回值`bool`表示操作是否改变了 Trie 树，`node`是插入完成后的子树的根节点，`error`是错误信息。

具体操作为：

- 如果节点类型是`nil`说明整棵树是空的，直接返回`shortNode{key, value, t.newFlag()}`，然后整棵树的根就有了一个`shortNode`节点
- 如果当前节点是`shortNode`(叶节点)，首先计算公共前缀，如果公共前缀相同，说明两个 key 相同，如果value也相同，则返回一个错误；如果没有错误就更新`shortNode`的值然后返回；如果公共前缀不完全匹配，就需要把公共前缀提取出来形成一个独立的节点(拓展节点)，拓展节点后面连接一个branch节点
- 如果当前节点是`fullNode`，直接向对应子节点调用`insert()`方法, 然后把对应子节点指向新的节点
- 如果当前节点是`hashNode`，说明对应的节点还未加载到内存，先调用`t.resolveHash`加载节点，然后对其调用`insert()`方法

Trie 树的查询

Go 语言实现的 Trie 树查询函数如下：

```
// Get returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
func (t *Trie) Get(key []byte) []byte {
  res, err := t.TryGet(key)
  if err != nil {
    log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
  }
  return res
}

// TryGet returns the value for key stored in the trie.
// The value bytes must not be modified by the caller.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *Trie) TryGet(key []byte) ([]byte, error) {
  value, newroot, didResolve, err := t.tryGet(t.root, keybytesToHex(key), 0)
  if err == nil && didResolve {
    t.root = newroot
  }
  return value, err
}

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:
    return n, n, false, nil
  case *shortNode:
    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
    }
    value, newnode, didResolve, err = t.tryGet(n.Val, key, pos+len(n.Key))
    if err == nil && didResolve {
      n = n.copy()
      n.Val = newnode
    }
    return value, n, didResolve, err
  case *fullNode:
    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:
    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))
  }
}
```

Trie 树查询首先是通过`Get()`函数和`TryGet()`函数进入，然后调用`tryGet()`函数遍历 Trie 树，来获取 Key 的信息。在遍历时需要依次判断该节点的类型：

- 如果是分支节点，就会按照字典树的方式，去查找下一位对应的路径
- 如果是叶子节点，内容匹配，返回查找结果；内容不匹配，返回错误信息
- 如果是扩展结点，继续对关联的分支节点进行查找
- 如果是`hashNode`，先加载节点到内存，然后进行查找

Trie 树的删除

Go 语言实现的 Trie 树删除函数如下：

```
// Delete removes any existing value for key from the trie.
func (t *Trie) Delete(key []byte) {
  if err := t.TryDelete(key); err != nil {
    log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
  }
```

```
}

// TryDelete removes any existing value for key from the trie.
// If a node was not found in the database, a MissingNodeError is returned.
func (t *Trie) TryDelete(key []byte) error {
  t.unhashed++
  k := keybytesToHex(key)
  _, n, err := t.delete(t.root, nil, k)
  if err != nil {
    return err
  }
  t.root = n
  return nil
}

// 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.
func (t *Trie) delete(n node, prefix, key []byte) (bool, node, error) {
  switch n := n.(type) {
  case *shortNode:
    matchlen := prefixLen(key, n.Key)
    if matchlen < len(n.Key) {
      return false, n, nil // don't replace n on mismatch
    }
    if matchlen == len(key) {
      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.
    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
    }
    switch child := child.(type) {
    case *shortNode:
      // Deleting from the subtrie reduced it to another
      // short node. Merge the nodes to avoid creating a
      // shortNode{..., shortNode{...}}. Use concat (which
      // always creates a new slice) instead of append to
      // avoid modifying n.Key since it might be shared with
      // other nodes.
      return true, &shortNode{concat(n.Key, child.Key...), child.Val, t.newFlag()}, ni
l
    default:
      return true, &shortNode{n.Key, child, t.newFlag()}, nil
    }

  case *fullNode:
    dirty, nn, err := t.delete(n.Children[key[0]], append(prefix, key[0]), key[1:])
    if !dirty || err != nil {
      return false, n, err
    }
    n = n.copy()
    n.flags = t.newFlag()
```

```
      n.Children[key[0]] = nn

      // Because n is a full node, it must've contained at least two children
      // before the delete operation. If the new child value is non-nil, n still
      // has at least two children after the deletion, and cannot be reduced to
      // a short node.
      if nn != nil {
        return true, n, nil
      }
      // Reduction:
      // Check how many non-nil entries are left after deleting and
      // reduce the full node to a short node if only one entry is
      // left. Since n must've contained at least two children
      // before deletion (otherwise it would not be a full node) n
      // can never be reduced to nil.
      //
      // 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.
      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 {
          // 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
          }
          if cnode, ok := cnode.(*shortNode); ok {
            k := append([]byte{byte(pos)}, cnode.Key...)
            return true, &shortNode{k, cnode.Val, t.newFlag()}, nil
          }
        }
        // Otherwise, n is replaced by a one-nibble short node
        // containing the child.
        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:
      return true, nil, nil

    case nil:
```

```
    return false, nil, nil

  case hashNode:
    // We've hit a part of the trie that isn't loaded yet. Load
    // the node and delete from 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.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))
  }
}
```

Trie 的删除操作与插入操作类似，即从根节点开始往下找，找到需要删除的节点进行删除操作。

MPT 的作用

MPT 具有以下几个作用：

- 存储任意长度的key-value键值对数据，符合以太坊的 state 模型
- 提供了一种快速计算所维护数据集哈希标识的机制
- 提供了快速状态回滚的机制
- 提供了一种称为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