以太坊源码解析：trie下篇

2019-01-18 fatcat22 ethereum 原创 ethereum trie 源码解析

> 本篇文章分析的源码地址为：https://github.com/ethereum/go-ethereum
> 分支：master
> commit id: 257bfff316e4efb8952fbeb67c91f86af579cb0a

引言

在上篇文章中，我们讲解了以太坊中Trie的主要原理。在这篇文章里，我们通过探索源代码来进一步了解以太坊trie模块。

以太坊trie模块位于项目目录下的trie目录。下面我们先学习下模块导出的功能和用法，然后结合源代码介绍一下目录的组织结构和框架。

trie模块使用方法

trie模块提供了四个主要的对象和接口：Trie、SecureTrie、NodeIterator、TrieSync、Database。下面分别介绍。

Trie

Trie对象实现了Merkle Patricia Tree的全部功能，包括(key, value)对的增删改查、计算默克尔哈希，以及将整个树写入数据库中。

你可以使用trie.New或trie.NewTrieWithPrefix来创建或打开一个Trie对象。trie.NewTrieWithPrefix相对于trie.New的不同之处在于，你需要提供一个前缀串，每次查询或写入(key, value)对时，都会自动将这个前缀串加到key的前面，然后再执行相应操作。

Trie对象提供的方法名都很简洁易懂，几乎所有方法都可以根据名字知道其功能。这里我们只是简单罗列一下，不一一详细说明。

```
func (t *Trie) NodeIterator(start []byte) NodeIterator
func (t *Trie) PrefixIterator(prefix []byte) NodeIterator
func (t *Trie) Get(key []byte) []byte
func (t *Trie) TryGet(key []byte) ([]byte, error)
func (t *Trie) Update(key, value []byte)
func (t *Trie) TryUpdate(key, value []byte)
func (t *Trie) Delete(key []byte)
func (t *Trie) TryDelete(key []byte) error
func (t *Trie) Root() []byte
func (t *Trie) Hash() common.Hash
func (t *Trie) Commit() (root common.Hash, err error)
func (t *Trie) CommitTo(db DatabaseWriter) (root common.Hash, err error)
func (t *Trie) SetCacheLimit(l uint16)
func (t *Trie) Prove(key []byte, fromLevel uint, proofDb DatabaseWriter) error
```

SecureTrie

SecureTrie对象实现了Trie相同的功能，实际上它内部几乎都是调用Trie对象的方法实现的。唯一不同的是，SecureTrie中的所有方法会将传入的key计算一个哈希，然后把这个哈希当作key进行操作。因此你无法通过根结点到叶子结点的路径上的信息拼凑出key的内容，这也是它叫作"Secure"的原因。

Content

SecureTrie提供的方法与Trie类似，这里不再细说。唯一需要注意的是，对 `SecureTrie` 节点进行枚举时，想要获取原始的key值，需要多调用一步 `SecureTrie.GetKey` 。因为 `NodeIterator.LeafKey` 和 `Iterator.Key` 得到的是加密后的key，需要调用 `SecureTrie.GetKey` 得到原始key。

NodeIterator

NodeIterator是一个接口，如名字所示，它提供了遍历树内部所有结点的功能。它提供的方法如下：

> func (it NodeIterator) Next(bool) bool
> func (it NodeIterator) Error() error
> func (it NodeIterator) Hash() common.Hash
> func (it NodeIterator) Parent() common.Hash
> func (it NodeIterator) Path() []byte
> func (it NodeIterator) Leaf() bool
> func (it NodeIterator) LeafBlob() []byte
> func (it NodeIterator) LeafKey() []byte

NodeIterator的核心是Next方法，每调用一次这个方法，NodeIterator对象当前代表的结点就会更新至下一个结点，此时你可以调用其它方法获取这个结点的信息。当所有结点遍历结束，Next方法返回false。所以使用NodeIterator接口的代码一般形式都像这样：

```
for it.Next(true) {
  ... // 现在it代表一了一个新的结点，你可以调用其它方法如it.Hash等，获取这个结点的信息
}
```

需要注意的是，NodeIterator接口对结点的遍历是有序的。其实Trie就是一棵有序树，而遍历的内部实现就是按照Key从短到长、从小到大的顺序进行的。（然而貌似并没有文档写明并保证遍历是有序的）

生成NodeIterator结口的方法有3种。我们分别说明一下。

1.调用和Trie.NodeIterator或Trie.PrefixIterator
如果你想遍历某个Trie对象的所有结点，可以调用Trie对象的方法 `NodeIterator()` 或 `PrefixIterator()` ，这俩个方法都返回一个NodeIterator接口用来进行遍历。NodeIterator()的start参数可以让你有选择的指定从哪个路径开始遍历，如果为nil则从头到尾按顺序遍历。PrefixIterator()方法能过参数指定前缀后，其返回的NodeIterator接口只会遍历路径中包含了你指定前缀的结点。

2.调用NewDifferenceIterator
`NewDifferenceIterator` 函数的功能如名字所示，只遍历不同的部分。当你调用NewDifferenceIterator(a, b NodeIterator)后，生成的NodeIterator只遍历存在于b但不存在于a中的结点。

3.调用NewUnionIterator
`NewUnionIterator` 也如名字所示，会把多个NodeIterator当成一个合集进行遍历。当你调用NewUnionIterator(its []NodeIterator)后，生成的NodeIterator遍历的结点是所有传入的结点的合集。

另外我们不得不强调一下NewIterator函数，因为我觉得这个函数更常用到。这个函数用来枚举树中的所有(key, value)对，而不是每个结点（多数情况下我们并不关心中间结点的数据）。这个函数声明如下：

> func NewIterator(it NodeIterator) *Iterator

它并不返回NodeIterator，而是返回Iterator指针对象。Iterator是一个结构体，只有一个方法。定义如下：

```
type Iterator struct {
  nodeIt NodeIterator

  Key   []byte // Current data key on which the iterator is positioned on
  Value []byte // Current data value on which the iterator is positioned on
  Err   error
}
```

```
//Iteartor的Next方法
func (it *Iterator) Next() bool
```

可以看到Iterator结构体导出了Key、Value、Err三个字段。每调用一次Next方法，Key、Value和Err字段就会被更新。以下是一段使用NewIterator的示例：

```
//假设tr是一个Trie变量，并且包含了("hi", "lili"), ("hello", "world"), ("good", "morning")三对数据
it := trie.NewIterator(tr.NodeIterator(nil))
for it.Next {
  if it.Err != nil {
    fmt.Println(it.Err)
    continue
  }
  fmt.Println(it.Key, it.Value)
}

//输出:
//good morning
//hello world
//hi lili
```

注意上面例子的三个输出是按Key从小到大排序的，这并非偶然。刚才已经说过NodeIterator的遍历是有序的，因此这里的输出也肯定是有序的。

TrieSync

TrieSync不是trie模块的主要功能，应用得也比较少，只在以太坊的downloader模块中用来辅助同步代表State的trie对象。与TrieSync有关的有以下几个方法：

> func NewTrieSync(root common.Hash, database DatabaseReader, callback TrieSyncLeafCallback) *TrieSync
> func (s *TrieSync) AddSubTrie(root common.Hash, depth int, parent common.Hash, callback TrieSyncLeafCallback)
> func (s *TrieSync) AddRawEntry(hash common.Hash, depth int, parent common.Hash)
> func (s *TrieSync) Missing(max int) []common.Hash
> func (s *TrieSync) Process(results []SyncResult) (bool, int, error)
> func (s *TrieSync) Commit(dbw DatabaseWriter) (int, error)
> func (s *TrieSync) Pending() int

其中`NewTrieSync`用来生成一个新的TrieSync对象并调用AddSubTrie方法为同步root结点作准备。Peinding用来获取当前的Trie对象有几个结点待同步；而Missing则返回待同步结点的哈希。当调用者通过网络获取到这些结点的数据时，它会调用Process进行处理。在Process内部会解析并记录这些结点，如果解析结果表示它们还有子结点，则加入missing和pending队列里。当调用者再次调用Missing方法时，就会获取到这些缺失的子结点的哈希。这样不断循环，直到Pending队列为空，表示所有结点同步完成。整个过程的示意代码如下：

```
trSync := trie.NewTrieSync(root, db, callback)
for trSync.Pending() != 0 {
  missHashs := trSync.Missing(0)

  results := downloadTrieNode(missHashs)

  trSync.Process(results)
}

trSync.Commit(db)
```

我们可以看到TrieSync并不具备同步的功能，它只是对结点的解析和组装。所以我觉得这个名字起得很有迷惑性，如果是我，我会把它叫做`TrieBuilder`。

Database

`Database`是trie模块对真正数据库的缓存层，其目的是对缓存的节点进行引用计数，从而实现区块的修剪功能。`Database`对外提供的方法有：

```
func NewDatabase(diskdb ethdb.Database) *Database
func NewDatabaseWithCache(diskdb ethdb.Database, cache int) *Database
func (db *Database) DiskDB() DatabaseReader
func (db *Database) InsertBlob(hash common.Hash, blob []byte)
func (db *Database) Node(hash common.Hash) ([]byte, error)
func (db *Database) Nodes() []common.Hash
func (db *Database) Reference(child common.Hash, parent common.Hash)
func (db *Database) Dereference(root common.Hash)
func (db *Database) Cap(limit common.StorageSize) error
func (db *Database) Commit(node common.Hash, report bool) error
func (db *Database) Size() (common.StorageSize, common.StorageSize)
```

其中`NewDatabase`或`NewDatabaseWithCache`用来创建一个`Database`对象，其参数`diskdb`是一个数据库接口。

在以太坊早期的trie模块中是没有`Database`对象的，加上这个就是为了增加节点引用计数功能，实现区块的修剪。详细信息我们后面再进行介绍。

目录结构

trie模块功能的实现代码全部位于以太坊项目的trie目录中，且没有子目录。下面我们对主要的源代码文件作简单的说明。

- encoding.go
  在上篇文章中我们提到过在Trie内部key的最小单位是nibble而不是byte，以及key在从数据库存取的时候是如何编码的。这个文件里的几个函数实现了这两大块功能。（nibble就是将一个byte的高4位和低4位拆成俩字节得到的值，比如将0xab拆成[0xa 0xb]，具体请参看trie上篇）

- hasher.go
  hasher.go中的代码实现了从某个结点开始计算子树的哈希的功能。可以说这个文件里的代码实现了以太坊的Trie的默克尔树特性。

- node.go
  在上篇文章中我们介绍过为了缩短无分支路径上的结点数量，以太坊使用了不同的结点类型和结构。这个文件里的代码就是定义了一个Trie树中所有结点的类型和解析的代码。

- sync.go
  sync.go中的代码实现了前面说的SyncTrie对象的定义和所有方法。

- iterator.go
  iterator.go中的代码定义了所有枚举相关接口和实现。

- secure_trie.go
  secure_trie.go中的代码实现了SecureTrie对象。

- errors.go
  errors.go中只定义了一个结构体：MissingNodeError。当找不到相应的结点时，就会返回这个错误。

- proof.go
  proof.go中只包含了Prove和VerifyProof两个函数，它们只在轻量级以太坊子协议（LES）中被使用。这两个函数被用来提供自己拥有某一对(key, value)的证明数据，和对数据进行验证。

- trie.go
  trie.go实现了Trie对象的主要逻辑功能。

- database.go
  database.go实现了`Database`对象的主要逻辑功能。

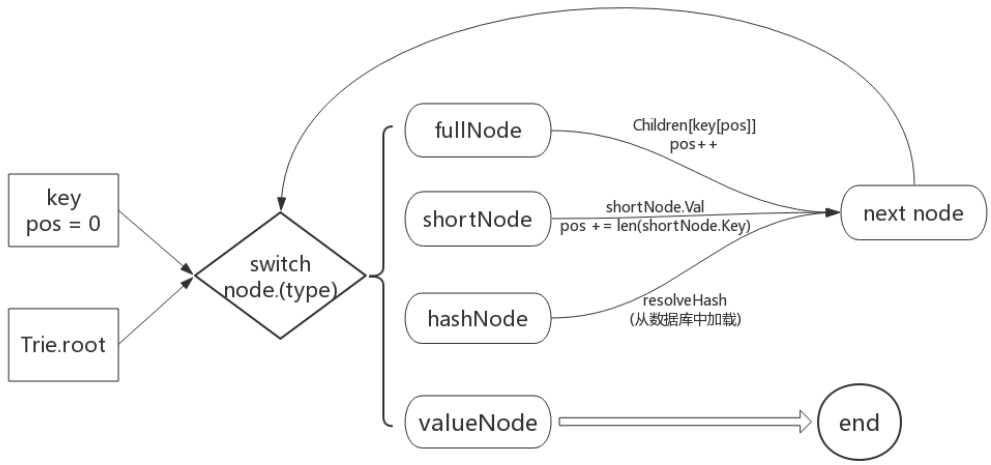
实现框架

前面我们说过，以太坊的trie模块提供了4个主要的对象和接口：Trie、SecureTrie、SyncTrie和NodeIterator。然而trie模块的核心功能是Trie对象，所以我们这里仅针对Trie作介绍（SecureTrie与Trie是类似的，实际上SecureTrie基本上是调用了Trie的方法）。

Trie对象的功能总得来看可以分成两部分：(key,value)的增删改查和Hash与Commit功能。下面我们分别对其介绍。

Trie对象的增删改查

Trie对象的Get、Update、Delete等方法实现了其增删改查的功能（当然也包含TryXXX等几个类似的方法）。其实Trie对象本质上是一棵树，对于树的增删改查操作，也就是对树的某一路径上所有结点的访问。所以在Get/Delete等方法的实现代码里，主体结构就是对节点类型的type switch，并根据不同的节点类型和目的（新增或删除）对节点进行操作。我对这一过程进行了一下规纳，如下图：



从图中可以看出，在key的驱动下，我们循环获取当前部分key所对应的结点（不断的“查字典”），并根据节点类型的不同使用不同的方式获取下一个结点。具体来说，如果是shortNode，则其value存储的是下一个结点；如果是fullNode，则Children数组中存储的是下一个节点——你需要根据key的值来确定到底是数组中的哪个元素；如果是hashNode，则说明这个结点还没有被从数据库中加载，只是存了一个结点哈希而已，因此要先载入并解析，再进行处理；如果是valueNode，则说明找到了key对应的value值，停止查找。

从以上的流程中也可以看出，要理解Trie以太坊中Trie的实现，关键要理解这4个结点类型以及它们是如何被使用的。在上篇中我们已经详细介绍过shortNode和fullNode的设计原理，这里我们再结合代码，详细地介绍一下这四个节点以及它们在代码中的用法。

在介绍4个结点类型之前，我们必须先说明一下node类型。这是一个接口类型，是4个结点类型的抽象（上面提到的type switch就是针对这个node接口类型进行的）。在shortNode和fullNode中，使用node接口代表其子结点，而不管子结点是什么类型。

fullNode

我们首先来看fullNode的定义：

```
// code in node.go
fullNode struct {
  Children [17]node
  flags    nodeFlag
}
```

结构体中flags字段记录了一些结点的其它信息，可以忽略不看，我们重点关注Children字段。上篇中我们提到过根据数据的元素个数来区分fullNode和shortNode类型，这里可以看到Children字段恰好就是一个17个元素的数据（是的，写入数据库时只写入Children，而不写入flags字段）。

相信你现在已经可以很好地理解对fullNode的相关操作了。当遇到一个fullNode时，我们将key[pos]作为索引，从Children数组中取出下一个结点。这里不得不提一下key[pos]的值为16的情况。上篇中我们提到过两个情况：一是[]byte类型的key在Trie内部会被转换成nibble数组，并在nibble数组最后添加一个值16代表key已到达结尾（参考keybytesToHex函数）;二是fullNode的17个元素中最后一个元素代表的是value而不是子结点。因此这里如果key[pos]的值是16，则取到的是Children数组的最后一个元素，也就是value。相信到这里，你可以完全理解为什么要在key尾部加一个值，以及为什么这个值是16而不是17或18等其它值。

shortNode

shortNode的定义如下：

```
//code in node.go
shortNode struct {
  Key   []byte
  Val   node
  flags nodeFlag
}
```

同样的我们忽略flags字段。可以看到shortNode有两个元素：Key和Val，这与我们之前介绍的一致（17个元素的是fullNode，2个元素的是shortNode）。

当遇到一个shortNode时，首先要对比key是否有shortNode.Key保存的前缀，如果是则Val字段代表了下一个结点，如果不是，则代表树中根本存在key这条路径。

shortNode稍微特殊一点的处理是新加分支。比如一个shortNode的Key保存的是"hello"，现在要新增一个key为"held"，那么就要把shortNode一分为二，中间插入一个fullNode。这个过程虽然麻烦一些但不难理解，我们这里就不详细展开了。

hashNode

hashNode的定义如下：

```
type hashNode []byte
```

hashNode就是一个简单的byte slice，它的内容就是某个结点的哈希（所以才叫做hashNode）。为什么会有这个结点类型呢？

这个类型的存在是由结点从数据库中延迟加载的机制造成的。之前我们说过，Trie的数据是存储在数据库中的。当我们使用某个root哈希创建一个新的Trie对象时，不会将所有子结点都一下子加载到内存中，而是用到再去加载。例如当我们创建新的Trie对象时，会根据root哈希从数据库中加载根结点。假设根结点是一个fullNode，那我们接下来是否要继续将它的16个子结点加载到内存中呢？答案是不需要的，我们只需要让根结点的Children数组保存子结点的哈希（其类型当然是hashNode），在需要时使用这个哈希从数据库中加载，并替换掉原来的哈希。其它结点的处理方法也是这样。这就是hashNode的意义。

valueNode

valueNode的定义如下：

```
type valueNode []byte
```

与hashNode类似，valueNode也是一个byte slice。它的意义是代表value数据，而不再是一个结点了（虽然名字里仍有node这个词）。如果在访问结点时遇到这个类型的结点，说明已经找到了key所对应的value。

Trie的Hash与Commit

Trie的另一个非常有特色的地方在于，它可以做永久性存储，也可以像默克尔树那样，获取根结点的哈希。这一小节里我们对这两个特性做个简单的介绍。

其实的树类数据结构想要做永久性存储，面临的问题都差不多。我觉得一个最主要的问题是，在数据库中没有了内存指针，如何对子结点进行引用。解决这个问题的方法有多种，比如为每个结点编号且存储时将指针改为编号；或者像以太坊这样使用结点哈希引用结点。这些方法的道理都是类似的。

之所以将这两个功能放在一起讲，也是因为它们内部的实现实际上用的是同一段代码。Trie.Hash和Trie.Commit方法最终都会调用内部方法Trie.hashRoot（Trie对象中还有两个方法Trie.Root和Trie.CommitTo，但与Trie.Hash、Trie.Commit都是一样的代码）。在hashRoot方法中，会调用内部对象hasher的hash方法。这个方法的就是遍历当前树在内存中每一个结点，将所有shortNode和fullNode结点变成hashNode（代码中叫压缩，collapsed）。直接看代码会更容易理解这个过程（下面的代码都是经过简化的，目的是为了更清晰的描述计算Trie的Hash和写入数据库的方法）

Trie.Hash

```
func (t *Trie) Hash() common.Hash {
  //注意hashRoot的参数为nil，即只计算哈希不保存到数据库中
  hash, cached, _ := t.hashRoot(nil)
  t.root = cached
  return hash
}
```

Trie.Commit

```
func (t *Trie) Commit() (root common.Hash, err error) {
  hash, cached, err := t.hashRoot(db)
  t.root = cached
  return hash
}
```

Trie.hashRoot

```
func (t *Trie) hashRoot(db DatabaseWriter) (node, node, error) {
  h := newHasher()
  return h.hash(t.root, db, true)
}
```

hasher.hash及相关代码：

```
//hash返回的第一个node为hashNode，第二个node为传入参数n的缓存（其实也可以理解成拷贝）
//collapsed为「压缩的」树，即它的所有子孙结点都是 hashNode。
func (h *hasher) hash(n node, db DatabaseWriter, force bool) (node, node, error) {
  collapsed, cached, err := h.hashChildren(n, db)

  hashed, err := h.store(collapsed, db, force)
  return hashed, cached, err
}

//hashChildren 以给定的结点为根，将这个子树「压缩」。
//所谓「压缩」，就是将所有子孙结点变成 hashNode 类型（并将原始节点存储到数据库中）。
//hashChildren 的第一个返回值正是「压缩」树的根；第二个node为传入参数original的缓存
func (h *hasher) hashChildren(original node, db DatabaseWriter) (node, node, error) {
  switch n := original.(type) {
```

```
    case *shortNode:
      collapsed := n.copy()
      collapsed.Key = hexToCompact(n.Key)

      cached := n.copy()
      cached.Key = common.CopyBytes(n.Key)

      if _, ok := n.Val.(valueNode); !ok {
        collapsed.Val, cached.Val, err = h.hash(n.Val, db, false)
      }
      return collapsed, cached, nil

    case *fullNode:
      // Hash the full node's children, caching the newly hashed subtrees
      collapsed, cached := n.copy(), n.copy()

      for i := 0; i < 16; i++ {
        collapsed.Children[i], cached.Children[i], err = h.hash(n.Children[i], db, false)
      }
      cached.Children[16] = n.Children[16]
      return collapsed, cached, nil

    default:
      // Value and hash nodes don't have children so they're left as were
      return n, original, nil
    }
  }


  //h.store计算collapsed结点的哈希；如果db不为nil，则将collapsed结点保存到db中。
  //Trie.Hash和Trie.Commit的区别也仅仅体现在这里。将普通结点变为hashNode这一操作也是
  //发生在这里
  func (h *hasher) store(n node, db DatabaseWriter, force bool) (node, error) {
    h.tmp.Reset()
    if err := rlp.Encode(h.tmp, n); err != nil {
      panic("encode error: " + err.Error())
    }

    h.sha.Reset()
    h.sha.Write(h.tmp.Bytes())
    hash = hashNode(h.sha.Sum(nil)) //将普通结点转变成对应的hashNode

    if db != nil {
      return hash, db.Put(hash, h.tmp.Bytes())
    }
    return hash, nil
  }
```

注意上面hasher.hash和hasher.hashChildren的实现。这两个方法通过相互的递归调用，从叶子结点开始逐步地将所有结点变成hashNode。

`hasher.hash` 始终维护着两棵树，它的前两个返回值分别为这两棵树的根。第一棵树的所有结点都是 `hashNode` 类型，包括根结点，因此使用这棵树可以很快的获得整棵树的哈希；第二棵树是原始的树，其结点可能是 `fullNode` 或 `shortNode` （当然也可能是其它类型）。使用这棵树可以防止每次调用 `Trie.Hash` 或 `Trie.Commit` 后，所有结点都被变成了 `hashNode` ，后续再次调用 `Trie.Get` 等方法时又得从数据库中重新加载。

`hasher.hashChildren` 用来将某个结点的所有子结点变成 `hashNode` 类型。作为 `hasher.hash` 的辅助方法， `hasher.hashChildren` 也维护了两棵树，不同的是它的第一棵树的根（也就是第一个返回值）并不是 `hashNode` 类型，只是这个根的所有子孙结点是 `hashNode` 类型而已。

Database

前面我们说过，在早期的以太坊trie版本中是没有`Database`对象的，后来加上这个对象就是用来实现区块的修剪功能（关于区块的修剪可以参看这里）。那么`Database`是怎么实现的呢？那就是对内存中的trie树节点进行引用计数，当引用计数为时，从内存中删除此节点。

在`Database`结构体中，对trie树节点实现引用计数功能的字段是`dirties`，它的类型是`map[common.Hash]*cachedNode`。其中`cachedNode`代表的是一个有引用计数的节点，它的定义如下：

```
type cachedNode struct {
  node node   // node接口
  size uint16

  parents  uint32                // 引用当前节点的节点数量
  children map[common.Hash]uint16 // 当前节点的子结点的引用计数

  flushPrev common.Hash // flush-list列表的字段
  flushNext common.Hash
}
```

在这个结构体中，`parents`和`children`实现了引用计数功能。而`flushPrev`和`flushNext`将当前节点加入到了flush-list链表中。

insert

我们先看看写入节点时会发生什么。当调用trie的Commit方法时，最终会调用`Database.insert`方法，且`Database.InsertBlob`其实也是调用insert方法，因此我们从这个方法看起：

```
func (db *Database) insert(hash common.Hash, blob []byte, node node) {
  //已存在立即返回
  if _, ok := db.dirties[hash]; ok {
    return
  }
  //构造cacheNode
  entry := &cachedNode{
    node:      simplifyNode(node),
    size:      uint16(len(blob)),
    flushPrev: db.newest,
  }
  //引用子节点，将所有子节点的parents加1
  for _, child := range entry.childs() {
    if c := db.dirties[child]; c != nil {
      c.parents++
    }
  }
  db.dirties[hash] = entry

  //加入flush-list链表
  if db.oldest == (common.Hash{}) {
    db.oldest, db.newest = hash, hash
  } else {
    db.dirties[db.newest].flushNext, db.newest = hash, hash
  }
  db.dirtiesSize += common.StorageSize(common.HashLength + entry.size)
}
```

`Database.insert`的主要逻辑非常简单，就是构造一个新加入的`cacheNode`节点，然后增加所有子节点的引用计数（parents字段）。注意这里并没有增加新节点自身的parents计数，因为这里只是往数据库里加入一个新节点，没有证据显示有人引用了这个节点。

reference

以太坊在进行区块的修剪时会调用 `Database.Reference` 和 `Database.Dereference` 两个方法。为了分析的完整一些，我们先来看看修剪时的调用：

```
func (bc *BlockChain) writeBlockWithState(block *types.Block, receipts []*types.Receipt, state
*state.StateDB) (status WriteStatus, err error) {
  ......
  if bc.cacheConfig.Disabled {
    ......
  } else {
    triedb.Reference(root, common.Hash{}) // metadata reference to keep trie alive
    ......

      for !bc.triegc.Empty() {
        ......
        triedb.Dereference(root.(common.Hash))
      }
  }
}
```

这里先引用一整棵树，经过一些判断和处理，再找合适机会解引用这棵树（详细分析请参看这里）。

对节点进行引用的功能主要在 `Database.reference` 中实现，我们来看看这个方法：

```
func (db *Database) reference(child common.Hash, parent common.Hash) {
  //如果child节点不在缓存中，立即返回
  node, ok := db.dirties[child]
  if !ok {
    return
  }
  //如果children字段不存在则构造并继续后面的执行。
  //如果children已存在说明已经引用过子节点了，那么如果child变量代表的不是根节点，就直接返回
  if db.dirties[parent].children == nil {
    db.dirties[parent].children = make(map[common.Hash]uint16)
  } else if _, ok = db.dirties[parent].children[child]; ok && parent != (common.Hash{}) {
    return
  }

  //增加引用计数
  node.parents++
  db.dirties[parent].children[child]++
}
```

这个方法就是增加 `child` 和 `parent` 节点的各自的引用计数。但这里有一个特殊的地方，就是如果 `child` 已经被 `parent` 引用过了，且 `child` 代表的不是一个根节点，那么就不继续增加 `parent` 对 `child` 的引用了；如果 `child` 代表一个根节点，还是要继续增加 `parent` 对 `child` 的引用。

这个处理有些难以理解，我们多解释一下。当父节点已经引用过某个子节点时，不再增加对子节点的引用是合理的，因为一个父节点只能引用 **某个特定的子节点** 一次，不存在引用多次的情况。但为什么 `parent` 为 `common.Hash{}` 时，还要继续引用呢？

这是因为在调用 `Database.Reference` 时，如果 `child` 参数是一个根节点，那么 `parent` 的值肯定是 `common.Hash{}`，也即 `common.Hash{}` 是任一trie树的根节点的父节点；所以这里判断 `parent` 是否是 `common.Hash{}`，也就是在判断 `child` 参数是否是一个根节点。对根节点的引用与对普通节点引用的不同之处在于，普通节点的引用发生在trie树的内部，因此刚才说了，一个父节点只能引用 **某个特定的子节点** 一次；而根节点是可以被trie树以外的地方引用的，比如在miner模块中引用了某个trie树的根节点，然后blockchain模块又对这个根节点引用了一次。所以这种情况不存在 `common.Hash{}` 只能引用某个根节点一次的情况。

deference

下面我们再来看看解引用的代码。解引用主要在 `Database.dereference` 中实现：

```
func (db *Database) dereference(child common.Hash, parent common.Hash) {
  // Dereference the parent-child
  node := db.dirties[parent]

  //首先解除父节点对子节点的引用
  if node.children != nil && node.children[child] > 0 {
    node.children[child]--
    if node.children[child] == 0 {
      delete(node.children, child)
    }
  }

  //如果child不存在，立即返回
  node, ok := db.dirties[child]
  if !ok {
    return
  }

  //减小对child参数代表的节点的引用
  if node.parents > 0 {
    node.parents--
  }

  //如果没人再引用这个节点，则删除它
  if node.parents == 0 {
    //将这个节点从flush-list中删除
    switch child {
      ......
    }

    //解除对所有子节点的引用，然后删除节点
    for _, hash := range node.childs() {
      db.dereference(hash, child)
    }
    delete(db.dirties, child)
    db.dirtiesSize -= common.StorageSize(common.HashLength + int(node.size))
  }
}
```

这段代码逻辑也比较简单，已经用中文注释进行了说明，这里就不再赘述了。但需要多说一点的时，只有某个节点将要被删除时，才会解引用所有子节点，而不是解引用某个节点的同时也解引用所有子节点。想象一下存在一个引用计数为2的节点A，它有一个引用计数为1的节点B（也即B只被A引用了）。当我们对A进行解引用时，A的计数变成了1，如果我们此时同时解引用A的子节点，那么B的计数将变成0从而被删除，但A仍然需要这么节点。因此只有当A的引用计数变为0将要被删除时，才应该解引用它的所有子节点。

flush-list

前面我们多次提到"flush-list"这个概念，现在我们就来看看它是什么。要了解flush-list，首先要了解`Database.Cap`的功能。这个方法将缓存中的数据刷到真实的数据库中，直到缓存占用的内存量达到参数的要求。flush-list就是决定在刷新缓存时，先刷哪个节点、最后刷哪个节点的一个双向链表。`Database.oldest`和`Database.newest`定义了这个链表的头和尾，链表的元素是`cachedNode`，而将`cacheNode`加入到这个链表中的字段就是`cacheNode.flushNext`和`cacheNode.flushPrev`。

我们直接看一下`Database.Cap`是如何使用这个链表的：

```
func (db *Database) Cap(limit common.StorageSize) error {
  ......

  oldest := db.oldest
  for size > limit && oldest != (common.Hash{}) {
    // Fetch the oldest referenced node and push into the batch
```

```
      node := db.dirties[oldest]
      if err := batch.Put(oldest[:], node.rlp()); err != nil {
        db.lock.RUnlock()
        return err
      }

      ......

      size -= common.StorageSize(3*common.HashLength + int(node.size))
      oldest = node.flushNext
    }

    ......
  }
```

可以看到这是一个典型的链表的遍历。至于其它对flush-list插入和删除的代码，相信在了解了它的功能以后，会很容易看懂，因此这里就不再详细说明了。

总结

在这篇文章中，我们结合源代码，介绍了以太坊中trie模块的功能与实现。trie模块的代码组织还是非常工整的，主要文件有trie.go、node.go、hasher.go、encoding.go和iterator.go。

trie模块的主要对象就是Trie。想要理解Trie的实现原理，关键是弄明白代码中的4种node类型，以及key的编码规则（encoding.go）。

唯一我觉得不太好的是，在trie.go及其它源文件中，到处都是关于node的type switch。我觉得这几乎完全可以放到一个函数中去做，至少可以不像现在这样，在Get/Upate/Delete等方法中都有一套type switch。

以上我是对以太坊的trie模块的分析，如有不对的地方还望不吝指正。

Similar Posts

- 初识联盟链1: Fabric 是什么
- PBFT代码篇：fabric 中的 PBFT 实现
- 实用拜占庭容错算法（PBFT）
- 拜占庭将军问题
- 数据结构与算法：B树
- 以太坊源码解析：evm

上一篇 以太坊源码解析：trie上篇

下一篇 以太坊源码解析：accounts

Comments

Comments

Login

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name

Email

Website (optional)

Displayed next to your comments.

Not displayed publicly.

If you have a website, link to it here.

Subscribe to None

Submit Comment

Contact me at:

Site powered by Jekyll & Github Pages. Theme designed by HyG.