chain_indexer 区块链索引

chain_indexer.go 源码解析

chain_indexer 顾名思义， 就是用来给区块链创建索引的功能。 BloomIndexer的功能，其实BloomIndexer是chain_indexer的一个特殊的实现， 可以理解为派生类， 主要的功能其实在chain_indexer这里面实现的。虽说是派生类，但是chain_indexer其实就只被BloomIndexer使用。也就是给区块链的布隆过滤器创建了索引，以便快速的响应用户的日志搜索功能。 下面就来分析这部分的代码。

数据结构

```
// ChainIndexerBackend defines the methods needed to process chain segments in
// the background and write the segment results into the database. These can be
// used to create filter blooms or CHTs.
// ChainIndexerBackend定义了处理区块链片段的方法，并把处理结果写入数据库。 这些可以用来创建布隆过滤器或者CHTs.
// BloomIndexer 其实就是实现了这个接口 ChainIndexerBackend 这里的CHTs不知道是什么东西。
type ChainIndexerBackend interface {
    // Reset initiates the processing of a new chain segment, potentially terminati
ng
    // any partially completed operations (in case of a reorg).
    // Reset 方法用来初始化一个新的区块链片段，可能会终止任何没有完成的操作。
    Reset(section uint64)

    // Process crunches through the next header in the chain segment. The caller
    // will ensure a sequential order of headers.
    // 对区块链片段中的下一个区块头进行处理。 调用者将确保区块头的连续顺序。
    Process(header *types.Header)

    // Commit finalizes the section metadata and stores it into the database.
    完成区块链片段的元数据并将其存储到数据库中。
    Commit() error
}

// ChainIndexer does a post-processing job for equally sized sections of the
// canonical chain (like BlooomBits and CHT structures). A ChainIndexer is
// connected to the blockchain through the event system by starting a
// ChainEventLoop in a goroutine.
// ChainIndexer 对区块链进行 大小相等的片段 进行处。 ChainIndexer在ChainEventLoop方法中通过事件系统与区块链通信，
// Further child ChainIndexers can be added which use the output of the parent
// section indexer. These child indexers receive new head notifications only
// after an entire section has been finished or in case of rollbacks that might
// affect already finished sections.
//更远可以添加使用父section索引器的输出的更多子链式索引器。 这些子链式索引器只有在整个部分完成后或在可能影响已完成部分的回滚的情况下才接收新的头部通知。

type ChainIndexer struct {
    chainDb  ethdb.Database      // Chain database to index the data from 区块链所在的数据库
    indexDb  ethdb.Database      // Prefixed table-view of the db to write index metadata into 索引存储的数据库
    backend  ChainIndexerBackend // Background processor generating the index data content  索引生成的后端。
    children []*ChainIndexer     // Child indexers to cascade chain updates to  子索引

    active uint32          // Flag whether the event loop was started
    update chan struct{}   // Notification channel that headers should be processed  接收到的headers
    quit   chan chan error // Quit channel to tear down running goroutines

    sectionSize uint64 // Number of blocks in a single chain segment to process section的大小。 默认是4096个区块为一个section
    confirmsReq uint64 // Number of confirmations before processing a completed segment   处理完成的段之前的确认次数

    storedSections uint64 // Number of sections successfully indexed into the database  成功索引到数据库的部分数量
    knownSections  uint64 // Number of sections known to be complete (block wise) 已知完成的部分数量
    cascadedHead   uint64 // Block number of the last completed section cascaded to subindexers 级联到子索引的最后一个完成部分的块号

    throttling time.Duration // Disk throttling to prevent a heavy upgrade from hogging resources 磁盘限制，以防止大量资源的大量升级

    log  log.Logger
    lock sync.RWMutex
}
```

构造函数NewChainIndexer,

```
这个方法是在eth/bloombits.go里面被调用的
const (
    // bloomConfirms is the number of confirmation blocks before a bloom section is
    // considered probably final and its rotated bits are calculated.
    // bloomConfirms 用来表示确认区块数量，  表示经过这么多区块之后， bloom section被认为是已经不会更改了。
    bloomConfirms = 256

    // bloomThrottling is the time to wait between processing two consecutive index
    // sections. It's useful during chain upgrades to prevent disk overload.
    // bloomThrottling是处理两个连续索引段之间的等待时间。 在区块链升级过程中防止磁盘过载
是很有用的。
    bloomThrottling = 100 * time.Millisecond
)

func NewBloomIndexer(db ethdb.Database, size uint64) *core.ChainIndexer {
    backend := &BloomIndexer{
        db:   db,
        size: size,
    }
    // 可以看到indexDb和chainDb实际是同一个数据库， 但是indexDb的每个key前面附加了一个Bl
oomBitsIndexPrefix的前缀。
    table := ethdb.NewTable(db, string(core.BloomBitsIndexPrefix))

    return core.NewChainIndexer(db, table, backend, size, bloomConfirms, bloomThrot
tling, "bloombits")
}


// NewChainIndexer creates a new chain indexer to do background processing on
// chain segments of a given size after certain number of confirmations passed.
// The throttling parameter might be used to prevent database thrashing.

func NewChainIndexer(chainDb, indexDb ethdb.Database, backend ChainIndexerBackend,
section, confirm uint64, throttling time.Duration, kind string) *ChainIndexer {
    c := &ChainIndexer{
        chainDb:     chainDb,
        indexDb:     indexDb,
        backend:     backend,
        update:      make(chan struct{}, 1),
        quit:        make(chan chan error),
        sectionSize: section,
        confirmsReq: confirm,
        throttling:  throttling,
        log:         log.New("type", kind),
    }
    // Initialize database dependent fields and start the updater
    c.loadValidSections()
    go c.updateLoop()

    return c
}
```

loadValidSections,用来从数据库里面加载我们之前的处理信息， storedSections表示我们已经处理到哪里了。

```
// loadValidSections reads the number of valid sections from the index database
// and caches is into the local state.
func (c *ChainIndexer) loadValidSections() {
    data, _ := c.indexDb.Get([]byte("count"))
    if len(data) == 8 {
        c.storedSections = binary.BigEndian.Uint64(data[:])
    }
}
```

updateLoop,是主要的事件循环，用于调用backend来处理区块链section，这个需要注意的是，所有的主索引节点和所有的 child indexer 都会启动这个goroutine 方法。

```
func (c *ChainIndexer) updateLoop() {
    var (
        updating bool
        updated  time.Time
    )
    for {
        select {
        case errc := <-c.quit:
            // Chain indexer terminating, report no failure and abort
            errc <- nil
            return

        case <-c.update:  //当需要使用backend处理的时候，其他goroutine会往这个channel上
面发送消息
            // Section headers completed (or rolled back), update the index
            c.lock.Lock()
            if c.knownSections > c.storedSections { // 如果当前以知的Section 大于已经
存储的Section
                // Periodically print an upgrade log message to the user
                // 每隔8秒打印一次日志信息。
                if time.Since(updated) > 8*time.Second {
                    if c.knownSections > c.storedSections+1 {
                        updating = true
                        c.log.Info("Upgrading chain index", "percentage", c.storedS
ections*100/c.knownSections)
                    }
                    updated = time.Now()
                }
                // Cache the current section count and head to allow unlocking the
mutex
                section := c.storedSections
                var oldHead common.Hash
                if section > 0 { // section - 1 代表section的下标是从0开始的。
                    // sectionHead用来获取section的最后一个区块的hash值。
                    oldHead = c.sectionHead(section - 1)
                }
                // Process the newly defined section in the background
                c.lock.Unlock()
                // 处理 返回新的section的最后一个区块的hash值
                newHead, err := c.processSection(section, oldHead)
                if err != nil {
                    c.log.Error("Section processing failed", "error", err)
                }
                c.lock.Lock()

                // If processing succeeded and no reorgs occcurred, mark the sectio
n completed
                if err == nil && oldHead == c.sectionHead(section-1) {
                    c.setSectionHead(section, newHead) // 更新数据库的状态
                    c.setValidSections(section + 1)    // 更新数据库状态
                    if c.storedSections == c.knownSections && updating {
                        updating = false
                        c.log.Info("Finished upgrading chain index")
                    }
                    // cascadedHead 是更新后的section的最后一个区块的高度
                    // 用法是什么 ?
                    c.cascadedHead = c.storedSections*c.sectionSize - 1
                    for _, child := range c.children {
                        c.log.Trace("Cascading chain index update", "head", c.casca
dedHead)

                        child.newHead(c.cascadedHead, false)
                    }
                } else { //如果处理失败，那么在有新的通知之前不会重试。
                    // If processing failed, don't retry until further notification
                    c.log.Debug("Chain index processing failed", "section", section
, "err", err)

                    c.knownSections = c.storedSections
                }
            }
            // If there are still further sections to process, reschedule
            // 如果还有section等待处理，那么等待throttling时间再处理。避免磁盘过载。
            if c.knownSections > c.storedSections {
                time.AfterFunc(c.throttling, func() {
                    select {
                    case c.update <- struct{}{}:
                    default:
                    }
                })
            }
            c.lock.Unlock()
        }
    }
}
```

Start方法。 这个方法在eth协议启动的时候被调用,这个方法接收两个参数，一个是当前的区块头，一个是事件订阅器，通过这个订阅器可以获取区块链的改变信息。

```
eth.bloomIndexer.Start(eth.blockchain.CurrentHeader(), eth.blockchain.SubscribeChainEvent)

// Start creates a goroutine to feed chain head events into the indexer for
// cascading background processing. Children do not need to be started, they
// are notified about new events by their parents.

// 子链不需要被启动。 以为他们的父节点会通知他们。
func (c *ChainIndexer) Start(currentHeader *types.Header, chainEventer func(ch chan<- ChainEvent) event.Subscription) {
    go c.eventLoop(currentHeader, chainEventer)
}

// eventLoop is a secondary - optional - event loop of the indexer which is only
// started for the outermost indexer to push chain head events into a processing
// queue.

// eventLoop 循环只会在最外面的索引节点被调用。 所有的Child indexer不会被启动这个方法。

func (c *ChainIndexer) eventLoop(currentHeader *types.Header, chainEventer func(ch chan<- ChainEvent) event.Subscription) {
    // Mark the chain indexer as active, requiring an additional teardown
    atomic.StoreUint32(&c.active, 1)

    events := make(chan ChainEvent, 10)
    sub := chainEventer(events)
    defer sub.Unsubscribe()

    // Fire the initial new head event to start any outstanding processing
    // 设置我们的其实的区块高度，用来触发之前未完成的操作。
    c.newHead(currentHeader.Number.Uint64(), false)

    var (
        prevHeader = currentHeader
        prevHash   = currentHeader.Hash()
    )
    for {
        select {
        case errc := <-c.quit:
            // Chain indexer terminating, report no failure and abort
            errc <- nil
            return

        case ev, ok := <-events:
            // Received a new event, ensure it's not nil (closing) and update
            if !ok {
                errc := <-c.quit
                errc <- nil
                return
            }
            header := ev.Block.Header()
            if header.ParentHash != prevHash { //如果出现了分叉，那么我们首先
                //找到公共祖先， 从公共祖先之后的索引需要重建。
                c.newHead(FindCommonAncestor(c.chainDb, prevHeader, header).Number.
Uint64(), true)
            }
            // 设置新的head
            c.newHead(header.Number.Uint64(), false)

            prevHeader, prevHash = header, header.Hash()
        }
    }
}
```

newHead方法,通知indexer新的区块链头，或者是需要重建索引，newHead方法会触发

```
// newHead notifies the indexer about new chain heads and/or reorgs.
func (c *ChainIndexer) newHead(head uint64, reorg bool) {
    c.lock.Lock()
    defer c.lock.Unlock()

    // If a reorg happened, invalidate all sections until that point
    if reorg { // 需要重建索引 从head开始的所有section都需要重建。
        // Revert the known section number to the reorg point
        changed := head / c.sectionSize
        if changed < c.knownSections {
            c.knownSections = changed
        }
        // Revert the stored sections from the database to the reorg point
        // 将存储的部分从数据库恢复到索引重建点
        if changed < c.storedSections {
            c.setValidSections(changed)
        }
        // Update the new head number to te finalized section end and notify childr
en
        // 生成新的head 并通知所有的子索引
        head = changed * c.sectionSize

        if head < c.cascadedHead {
            c.cascadedHead = head
            for _, child := range c.children {
                child.newHead(c.cascadedHead, true)
            }
        }
        return
    }
    // No reorg, calculate the number of newly known sections and update if high en
ough
    var sections uint64
    if head >= c.confirmsReq {
        sections = (head + 1 - c.confirmsReq) / c.sectionSize
        if sections > c.knownSections {
            c.knownSections = sections

            select {
            case c.update <- struct{}{}:
            default:
            }
        }
    }
}
```

父子索引数据的关系
父Indexer负载事件的监听然后把结果通过newHead传递给子Indexer的updateLoop来处理。

setValidSections方法，写入当前已经存储的sections的数量。 如果传入的值小于已经存储的数量，那么从数据库里面删除对应的section

```
// setValidSections writes the number of valid sections to the index database
func (c *ChainIndexer) setValidSections(sections uint64) {
    // Set the current number of valid sections in the database
    var data [8]byte
    binary.BigEndian.PutUint64(data[:], sections)
    c.indexDb.Put([]byte("count"), data[:])

    // Remove any reorged sections, caching the valids in the mean time
    for c.storedSections > sections {
        c.storedSections--
        c.removeSectionHead(c.storedSections)
    }
    c.storedSections = sections // needed if new > old
}
```

processSection

```
// processSection processes an entire section by calling backend functions while
// ensuring the continuity of the passed headers. Since the chain mutex is not
// held while processing, the continuity can be broken by a long reorg, in which
// case the function returns with an error.

//processSection通过调用后端函数来处理整个部分，同时确保传递的头文件的连续性。 由于链接互
斥锁在处理过程中没有保持，连续性可能会被重新打断，在这种情况下，函数返回一个错误。
func (c *ChainIndexer) processSection(section uint64, lastHead common.Hash) (common
.Hash, error) {
    c.log.Trace("Processing new chain section", "section", section)

    // Reset and partial processing
    c.backend.Reset(section)

    for number := section * c.sectionSize; number < (section+1)*c.sectionSize; numb
er++ {
        hash := GetCanonicalHash(c.chainDb, number)
        if hash == (common.Hash{}) {
            return common.Hash{}, fmt.Errorf("canonical block #%d unknown", number)
        }
        header := GetHeader(c.chainDb, hash, number)
        if header == nil {
            return common.Hash{}, fmt.Errorf("block #%d [%x…] not found", number, h
ash[:4])
        } else if header.ParentHash != lastHead {
            return common.Hash{}, fmt.Errorf("chain reorged during section processi
ng")
        }
        c.backend.Process(header)
        lastHead = header.Hash()
    }
    if err := c.backend.Commit(); err != nil {
        c.log.Error("Section commit failed", "error", err)
        return common.Hash{}, err
    }
    return lastHead, nil
}
```