StateSync源码分析

1、简介

statesync是一个同步状态的模块，最直接的联系为block中的stateRoot，stateRoot就是使用statesync模块进行下载的。

2、源码分析

万物之始，首先介绍statesync本身的数据结构

```
// stateSync schedules requests for downloading a particular state trie defined
// by a given state root.
// stateSync通过调度发送请求下去载一个特定状态根的状态树
type stateSync struct {
    d *Downloader // Downloader instance to access and manage current peerset 组合downloader模块
     //调度器
    sched  *trie.Sync                 // State trie sync scheduler defining the tasks
    keccak hash.Hash                  // Keccak256 hasher to verify deliveries with
    // 请求任务队列
    tasks  map[common.Hash]*stateTask // Set of tasks currently queued for retrieval
    // 统计位
    numUncommitted   int
    bytesUncommitted int
    // 处理数据相关的通道
    deliver    chan *stateReq // Delivery channel multiplexing peer responses
    cancel     chan struct{}  // Channel to signal a termination request
    cancelOnce sync.Once      // Ensures cancel only ever gets called once
    done       chan struct{}  // Channel to signal termination completion
    err        error          // Any error hit during sync (set before completion)
}
```

以上即是statesync的主要数据结构，下面紧接着开始进入源码引导分析之旅

```
    // 1、首先看一下状态同步如何开启，以及如何触发状态同步
    // 状态同步线程开启：
    // downloader下的new方法
    // New creates a new downloader to fetch hashes and blocks from remote peers.
func New(mode SyncMode, stateDb ethdb.Database, mux *event.TypeMux, chain BlockChain, lightchain
LightChain, dropPeer peerDropFn) *Downloader {
    ......................
    go dl.stateFetcher()
    return dl
}
// 进入到stateFetcher方法以后，就开始轮训通道，等待状态同步任务到来
// stateFetcher manages the active state sync and accepts requests
// on its behalf.
func (d *Downloader) stateFetcher() {
    for {
        select {
```

```
         // 从d.stateSyncStart通道读取同步任务
        case s := <-d.stateSyncStart:
            for next := s; next != nil; {
                 // 调用runStateSync方法进行状态同步
                next = d.runStateSync(next)
            }
        case <-d.stateCh:
            // Ignore state responses while no sync is running.
        case <-d.quitCh:
            return
        }
    }
}
```

上面主要介绍了下状态同步线程stateFetcher何时开启，开启后做了什么事情。下面开始讲解何时触发了状态同步

```
// 1、首先回忆downloader处理快速同步内容的方法processFastSyncContent
func (d *Downloader) processFastSyncContent(latest *types.Header) error {
    // Start syncing state of the reported head block. This should get us most of
    // the state of the pivot block.
    // 在processFastSyncContent方法的第一行调用了d.sycnState方法，传入bestPeer的最新区块的stateRoot
    stateSync := d.syncState(latest.Root)
    ..................
}
// 2、下面我们分析一下d.sycnState(latest.Root)做了什么工作
func (d *Downloader) syncState(root common.Hash) *stateSync {
    // 先调用newStateSync构造同步任务
    s := newStateSync(d, root)
    select {
     // 将同步任务放入stateFetcher轮训的通道中，交给stateFether处理，stateFetcher在上面已经提到
    case d.stateSyncStart <- s:
    case <-d.quitCh:
        s.err = errCancelStateFetch
        close(s.done)
    }
    return s
}
// 3、先来分析newStateSync做了什么工作
func newStateSync(d *Downloader, root common.Hash) *stateSync {
    // 实际就是构造stateSync结构体，其中state.NewStateSync我们在下面再继续分析
    return &stateSync{
        d:       d,
        sched:   state.NewStateSync(root, d.stateDB),
        keccak:  sha3.NewLegacyKeccak256(),
        tasks:   make(map[common.Hash]*stateTask),
        deliver: make(chan *stateReq),
        cancel:  make(chan struct{}),
        done:    make(chan struct{}),
    }
}
func NewStateSync(root common.Hash, database trie.DatabaseReader) *trie.Sync {
    var syncer *trie.Sync
     // 定义callback方法
    callback := func(leaf []byte, parent common.Hash) error {
        var obj Account
        if err := rlp.Decode(bytes.NewReader(leaf), &obj); err != nil {
```

```
            return err
        }
         // 随后在下面分析一下AddSubTrie方法
        syncer.AddSubTrie(obj.Root, 64, parent, nil)
        syncer.AddRawEntry(common.BytesToHash(obj.CodeHash), 64, parent)
        return nil
    }
    // 调用trie.NewSync方法
    syncer = trie.NewSync(root, database, callback)
    return syncer
}
// NewSync creates a new trie data download scheduler.
func NewSync(root common.Hash, database DatabaseReader, callback LeafCallback) *Sync {
    ts := &Sync{
         // 数据库
        database: database,
         // 内存缓存
        membatch: newSyncMemBatch(),
        // 请求map
        requests: make(map[common.Hash]*request),
        // 优先级队列
        queue:    prque.New(nil),
    }
    // 下面分析AddSubTrie
    ts.AddSubTrie(root, 0, common.Hash{}, callback)
    return ts
}
func (s *Sync) AddSubTrie(root common.Hash, depth int, parent common.Hash, callback LeafCallback)
{
    // 判断根是不是为空
    if root == emptyRoot {
        return
    }
    // 判断内存中是否已经存在此root，存在的话直接返回
    if _, ok := s.membatch.batch[root]; ok {
        return
    }
    // 从本地数据库中获取数据
    key := root.Bytes()
    blob, _ := s.database.Get(key)
     // 如果本地解析出来这个root trie节点，那么证明已经存在此根，直接返回
    if local, err := decodeNode(key, blob, 0); local != nil && err == nil {
        return
    }
    // 执行到此开始装配同步请求
    req := &request{
        hash:     root,
        depth:    depth,
        callback: callback,
    }
    // 如果这个子树有指定的父节点，把他们链接在
    if parent != (common.Hash{}) {
        ancestor := s.requests[parent]
        if ancestor == nil {
            panic(fmt.Sprintf("sub-trie ancestor not found: %x", parent))
        }
        ancestor.deps++
```

```
        req.parents = append(req.parents, ancestor)
    }
    // 开始将请求放入调度器的任务切片中
    s.schedule(req)
}
func (s *Sync) schedule(req *request) {
    // If we're already requesting this node, add a new reference and stop
    if old, ok := s.requests[req.hash]; ok {
        old.parents = append(old.parents, req.parents...)
        return
    }
    // Schedule the request for future retrieval
    s.queue.Push(req.hash, int64(req.depth))
    s.requests[req.hash] = req
}
```

以上主要是同步任务的构建，直到最后被放入调度器会拿取数据的切片中，下面我们开始分析如何将同步请求发送出去及将请求结果落入本地数据库。

```
// 1、上面已经构建好一个stateSync结构体并且将其传送到d.stateSyncStart通道，stateFetcher中监听此通道数据，拿到数据后会调用d.runStateSync()方法，那么下面就开始分析runStateSync方法
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    var (
        active   = make(map[string]*stateReq) // 处理中的请求
        finished []*stateReq                  // 完成或者失败的请求
        timeout  = make(chan *stateReq)       // 过期的请求
    )
    defer func() {
         // 方法解释如下英文
        // Cancel active request timers on exit. Also set peers to idle so they're
        // available for the next sync.
        for _, req := range active {
            req.timer.Stop()
            req.peer.SetNodeDataIdle(len(req.items))
        }
    }()
    // 另起线程执行状态同步主循环，后面分析
    go s.run()
    defer s.Cancel()

    // 下面监听peer断开事件，断开后则不给断开的peer分配任务
    peerDrop := make(chan *peerConnection, 1024)
    peerSub := s.d.peers.SubscribePeerDrops(peerDrop)
    defer peerSub.Unsubscribe()
    // 下面for循环处理各种通道数据
    for {
        // Enable sending of the first buffered element if there is one.
        var (
            deliverReq   *stateReq
            deliverReqCh chan *stateReq
        )
        if len(finished) > 0 {
            deliverReq = finished[0]
            deliverReqCh = s.deliver
        }
```

```
        select {
        // 如果在状态同步期间另一个同步请出现，则此方法返回新的stateSync对象，重新调用runStateSync方法
        case next := <-d.stateSyncStart:
            return next
        // 如果收到同步完成的消息则返回
        case <-s.done:
            return nil

        // 从finished中取出第一个完成的请求，发送请求完成的结果给deliverReqCh并删除finised第一个元素，缩短finished切片
        case deliverReqCh <- deliverReq:
            // Shift out the first request, but also set the emptied slot to nil for GC
            copy(finished, finished[1:])
            finished[len(finished)-1] = nil
            finished = finished[:len(finished)-1]

        // 处理收到的请求回应包
        case pack := <-d.stateCh:
            //如果此回应包不是我们请求的，则忽略
            req := active[pack.PeerId()]
            if req == nil {
                log.Debug("Unrequested node data", "peer", pack.PeerId(), "len", pack.Items())
                continue
            }
            // 先终止此请求的定时器
            req.timer.Stop()
             // 将回应结果赋值给请求的response
            req.response = pack.(*statePack).states
            // 将此请求添加到已完成切片中国
            finished = append(finished, req)、
             // 从正在处理请求map结构中删除此请求
            delete(active, pack.PeerId())

        // 处理peerDrop消息
        case p := <-peerDrop:
            // 如果此peer对应的请求为空，则直接跳过此次处理
            req := active[p.id]
            if req == nil {
                continue
            }
            // 若此peer对应有相应请求，则停止其计时器并将dropped标志置为true
            req.timer.Stop()
            req.dropped = true
            // 将此请求加入finished切片中，finished切片存放完成的请求或者失败的请求，并从active中删除此peer对应的请求
            finished = append(finished, req)
            delete(active, p.id)

        // 处理超时请求
        case req := <-timeout:
            // 如果此peer已经有了其他状态请求，则跳过
            if active[req.peer.id] != req {
                continue
            }
            // 要不然按照失败请求处理
            finished = append(finished, req)
            delete(active, req.peer.id)
```

```
        // 跟踪外发的状态请求
        case req := <-d.trackStateReq:
            // If an active request already exists for this peer, we have a problem. In
            // theory the trie node schedule must never assign two requests to the same
            // peer. In practice however, a peer might receive a request, disconnect and
            // immediately reconnect before the previous times out. In this case the first
            // request is never honored, alas we must not silently overwrite it, as that
            // causes valid requests to go missing and sync to get stuck.
             // 此段理解请看上述英文
            if old := active[req.peer.id]; old != nil {
                log.Warn("Busy peer assigned new state fetch", "peer", old.peer.id)

                // Make sure the previous one doesn't get siletly lost
                old.timer.Stop()
                old.dropped = true

                finished = append(finished, old)
            }
             // 给请求添加计时器
            // Start a timer to notify the sync loop if the peer stalled.
            req.timer = time.AfterFunc(req.timeout, func() {
                select {
                case timeout <- req:
                case <-s.done:
                    // Prevent leaking of timer goroutines in the unlikely case where a
                    // timer is fired just before exiting runStateSync.
                }
            })
             // 将此请求放入活跃请求Map中
            active[req.peer.id] = req
        }
    }
}
```

以上就是runstateSync代码，期间加的注释很方便大家理解，主要就是在for循环处理各个通道的消息，下面来分析下s.run()方法，这个方法是状态同步的主循环。

```
func (s *stateSync) run() {
    s.err = s.loop()
    close(s.done)
}
func (s *stateSync) loop() (err error) {
    // 监听新peer产生事件，以便给新peer安排任务
    newPeer := make(chan *peerConnection, 1024)
    peerSub := s.d.peers.SubscribeNewPeers(newPeer)
    defer peerSub.Unsubscribe()
    defer func() {
        cerr := s.commit(true)
        if err == nil {
            err = cerr
        }
    }()

    // 若s.request中有状态同步请求（也就是上面s.schedule()方法中放入的请求）那么开始处理
    for s.sched.Pending() > 0 {
```

```
        // 先执行一次数据提交，参数表示是否强制提交，方法后面分析
        if err = s.commit(false); err != nil {
            return err
        }
        // 开始安排任务
        s.assignTasks()
        // Tasks assigned, wait for something to happen
        select {
        case <-newPeer:
            // New peer arrived, try to assign it download tasks

        case <-s.cancel:
            return errCancelStateFetch

        case <-s.d.cancelCh:
            return errCancelStateFetch
        // 处理状态请求返回的结果，这个结果的传送过程在此讲解下：
         // 首先给相关节点发送GetNodeDataMsg，对方回复NodeDataMsg，ProtocolManager.handlerMsg调用
pm.downloader.DeliverNodeData(p.id, data)将消息进行分发，实际调用d.deliver(id, d.stateCh,
&statePack{id, data}, stateInMeter, stateDropMeter)进行消息分发，将消息写入d.stateCh
         // d.stateCh在runStateSync方法的for循环中处理此消息，将消息放入finised切片中
         // for循环开始时候会将deliverReq = finished[0] deliverReqCh = s.deliver
         // 再select deliverReqCh <- deliverReq
         // 最后下面代码将受到状态请求的结果
        case req := <-s.deliver:
            // Response, disconnect or timeout triggered, drop the peer if stalling
            log.Trace("Received node data response", "peer", req.peer.id, "count",
len(req.response), "dropped", req.dropped, "timeout", !req.dropped && req.timedOut())
            if len(req.items) <= 2 && !req.dropped && req.timedOut() {
                // 2 items are the minimum requested, if even that times out, we've no use of this
peer at the moment.
                log.Warn("Stalling state sync, dropping peer", "peer", req.peer.id)
                s.d.dropPeer(req.peer.id)
            }
            // 开始处理状态同步请求的结果
            delivered, err := s.process(req)
            if err != nil {
                log.Warn("Node data write error", "err", err)
                return err
            }
             // 设置此peer为空闲
            req.peer.SetNodeDataIdle(delivered)
        }
    }
    return nil
}
```

以上就是s.run()主循环做的事情，主要事情是：

1、检测是否有新的状态请求，若无则运行一次状态强制提交，若有则进行下面处理

2、先执行一次非强制提交，看看内存是否够用，若够用则不做任何动作，要不然进行状态提交

3、给空闲peer安排状态请求任务

4、处理状态请求结果

commit(bool)

```
func (s *stateSync) commit(force bool) error {
    // 若为非强制提交模式且为提交的字节小于100 * 1024 则返回
    if !force && s.bytesUncommitted < ethdb.IdealBatchSize {
        return nil
    }
    // 若上面条件不满足，则开始进行数据持久化
    start := time.Now()、
    // 创建一个Batch
    b := s.d.stateDB.NewBatch()
    // 将 s.membatch 数据放入etgdb.Putter中
    if written, err := s.sched.Commit(b); written == 0 || err != nil {
        return err
    }
    // 调用 batch的Write()方法，实际将Putter的数据写入数据库
    if err := b.Write(); err != nil {
        return fmt.Errorf("DB write error: %v", err)
    }
    // 将相关状态打印给用户看
    s.updateStats(s.numUncommitted, 0, 0, time.Since(start))
    // 将相关状态置0
    s.numUncommitted = 0
    s.bytesUncommitted = 0
    return nil
}
```

以上就是状态同步部分做的全部事情，期间还有许多代码细节需要大家自己去了解，本次分析就到这里。