downloader主要负责区块链最开始的同步工作，当前的同步有两种模式，一种是传统的fullmode,这种模式通过下载区块头，和区块体来构建区块链，同步的过程就和普通的区块插入的过程一样，包括区块头的验证，交易的验证，交易执行，账户状态的改变等操作，这其实是一个比较消耗CPU和磁盘的一个过程。 另一种模式就是 快速同步的fast sync模式， 这种模式有专门的文档来描述。请参考fast sync的文档。简单的说 fast sync的模式会下载区块头，区块体和收据， 插入的过程不会执行交易，然后在一个区块高度(最高的区块高度 - 1024)的时候同步所有的账户状态，后面的1024个区块会采用fullmode的方式来构建。 这种模式会加区块的插入时间，同时不会产生大量的历史的账户信息。会相对节约磁盘， 但是对于网络的消耗会更高。 因为需要下载收据和状态。

downloader 数据结构

```
type Downloader struct {
    mode SyncMode       // Synchronisation mode defining the strategy used (per syn
c cycle)
    mux  *event.TypeMux // Event multiplexer to announce sync operation events
    // queue 对象用来调度 区块头，交易，和收据的下载，以及下载完之后的组装
    queue   *queue   // Scheduler for selecting the hashes to download
    // 对端的集合
    peers   *peerSet // Set of active peers from which download can proceed
    stateDB ethdb.Database
    // fast sync 中的 Pivot point区块的头
    fsPivotLock  *types.Header // Pivot header on critical section entry (cannot ch
ange between retries)
    fsPivotFails uint32        // Number of subsequent fast sync failures in the cr
itical section
    // 下载的往返时延
    rttEstimate   uint64 // Round trip time to target for download requests
    rttConfidence uint64 // Confidence in the estimated RTT (unit: millionths to al
low atomic ops)  估计RTT的信心(单位：允许原子操作的百万分之一)

    // Statistics 统计信息,
    syncStatsChainOrigin uint64 // Origin block number where syncing started at
    syncStatsChainHeight uint64 // Highest block number known when syncing started
    syncStatsState       stateSyncStats
    syncStatsLock        sync.RWMutex // Lock protecting the sync stats fields

    lightchain LightChain
    blockchain BlockChain

    // Callbacks
    dropPeer peerDropFn // Drops a peer for misbehaving
```

```
    // Status
    synchroniseMock func(id string, hash common.Hash) error // Replacement for sync
hronise during testing
    synchronising   int32
    notified        int32

    // Channels
    headerCh      chan dataPack        // [eth/62] Channel receiving inbound block
headers  header的输入通道，从网络下载的header会被送到这个通道
    bodyCh        chan dataPack        // [eth/62] Channel receiving inbound block
bodies   bodies的输入通道，从网络下载的bodies会被送到这个通道
    receiptCh     chan dataPack        // [eth/63] Channel receiving inbound receip
ts       receipts的输入通道，从网络下载的receipts会被送到这个通道
    bodyWakeCh    chan bool            // [eth/62] Channel to signal the block body
 fetcher of new tasks    用来传输body fetcher新任务的通道
    receiptWakeCh chan bool            // [eth/63] Channel to signal the receipt fe
tcher of new tasks      用来传输receipt fetcher 新任务的通道
    headerProcCh  chan []*types.Header // [eth/62] Channel to feed the header proce
ssor new tasks       通道为header处理者提供新的任务

    // for stateFetcher
    stateSyncStart chan *stateSync   //用来启动新的 state fetcher
    trackStateReq  chan *stateReq    // TODO
    stateCh        chan dataPack // [eth/63] Channel receiving inbound node state d
ata     state的输入通道，从网络下载的state会被送到这个通道
    // Cancellation and termination
    cancelPeer string        // Identifier of the peer currently being used as the
master (cancel on drop)
    cancelCh   chan struct{} // Channel to cancel mid-flight syncs
    cancelLock sync.RWMutex  // Lock to protect the cancel channel and peer in deli
vers

    quitCh   chan struct{} // Quit channel to signal termination
    quitLock sync.RWMutex  // Lock to prevent double closes

    // Testing hooks
    syncInitHook     func(uint64, uint64)  // Method to call upon initiating a new
sync run
    bodyFetchHook    func([]*types.Header) // Method to call upon starting a block
body fetch
    receiptFetchHook func([]*types.Header) // Method to call upon starting a receip
t fetch
    chainInsertHook  func([]*fetchResult)  // Method to call upon inserting a chain
 of blocks (possibly in multiple invocations)
}
```

构造方法

```
// New creates a new downloader to fetch hashes and blocks from remote peers.
func New(mode SyncMode, stateDb ethdb.Database, mux *event.TypeMux, chain BlockChai
n, lightchain LightChain, dropPeer peerDropFn) *Downloader {
    if lightchain == nil {
        lightchain = chain
    }
    dl := &Downloader{
        mode:           mode,
        stateDB:        stateDb,
        mux:            mux,
        queue:          newQueue(),
        peers:          newPeerSet(),
        rttEstimate:    uint64(rttMaxEstimate),
        rttConfidence:  uint64(1000000),
        blockchain:     chain,
        lightchain:     lightchain,
        dropPeer:       dropPeer,
        headerCh:       make(chan dataPack, 1),
        bodyCh:         make(chan dataPack, 1),
        receiptCh:      make(chan dataPack, 1),
        bodyWakeCh:     make(chan bool, 1),
        receiptWakeCh:  make(chan bool, 1),
        headerProcCh:   make(chan []*types.Header, 1),
        quitCh:         make(chan struct{}),
        stateCh:        make(chan dataPack),
        stateSyncStart: make(chan *stateSync),
        trackStateReq:  make(chan *stateReq),
    }
    go dl.qosTuner()  //简单 主要用来计算rttEstimate和rttConfidence
    go dl.stateFetcher() //启动stateFetcher的任务监听，但是这个时候还没有生成state fetc
her的任务。
    return dl
}
```

同步下载

Synchronise试图和一个peer来同步，如果同步过程中遇到一些错误，那么会删除掉Peer。然后会被重试。

```
// Synchronise tries to sync up our local block chain with a remote peer, both
// adding various sanity checks as well as wrapping it with various log entries.
func (d *Downloader) Synchronise(id string, head common.Hash, td *big.Int, mode Syn
```

```
cMode) error {
    err := d.synchronise(id, head, td, mode)
    switch err {
    case nil:
    case errBusy:

    case errTimeout, errBadPeer, errStallingPeer,
        errEmptyHeaderSet, errPeersUnavailable, errTooOld,
        errInvalidAncestor, errInvalidChain:
        log.Warn("Synchronisation failed, dropping peer", "peer", id, "err", err)
        d.dropPeer(id)

    default:
        log.Warn("Synchronisation failed, retrying", "err", err)
    }
    return err
}
```

synchronise

```
// synchronise will select the peer and use it for synchronising. If an empty strin
g is given
// it will use the best peer possible and synchronize if it's TD is higher than our
 own. If any of the
// checks fail an error will be returned. This method is synchronous
func (d *Downloader) synchronise(id string, hash common.Hash, td *big.Int, mode Syn
cMode) error {
    // Mock out the synchronisation if testing
    if d.synchroniseMock != nil {
        return d.synchroniseMock(id, hash)
    }
    // Make sure only one goroutine is ever allowed past this point at once
    // 这个方法同时只能运行一个， 检查是否正在运行。
    if !atomic.CompareAndSwapInt32(&d.synchronising, 0, 1) {
        return errBusy
    }
    defer atomic.StoreInt32(&d.synchronising, 0)

    // Post a user notification of the sync (only once per session)
    if atomic.CompareAndSwapInt32(&d.notified, 0, 1) {
        log.Info("Block synchronisation started")
    }
    // Reset the queue, peer set and wake channels to clean any internal leftover s
tate
    // 重置queue和peer的状态。
    d.queue.Reset()
    d.peers.Reset()
```

```
    // 清空d.bodyWakeCh, d.receiptWakeCh
    for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
        select {
        case <-ch:
        default:
        }
    }
    // 清空d.headerCh, d.bodyCh, d.receiptCh
    for _, ch := range []chan dataPack{d.headerCh, d.bodyCh, d.receiptCh} {
        for empty := false; !empty; {
            select {
            case <-ch:
            default:
                empty = true
            }
        }
    }
    // 清空headerProcCh
    for empty := false; !empty; {
        select {
        case <-d.headerProcCh:
        default:
            empty = true
        }
    }
    // Create cancel channel for aborting mid-flight and mark the master peer
    d.cancelLock.Lock()
    d.cancelCh = make(chan struct{})
    d.cancelPeer = id
    d.cancelLock.Unlock()

    defer d.Cancel() // No matter what, we can't leave the cancel channel open

    // Set the requested sync mode, unless it's forbidden
    d.mode = mode
    if d.mode == FastSync && atomic.LoadUint32(&d.fsPivotFails) >= fsCriticalTrials
 {
        d.mode = FullSync
    }
    // Retrieve the origin peer and initiate the downloading process
    p := d.peers.Peer(id)
    if p == nil {
        return errUnknownPeer
    }
    return d.syncWithPeer(p, hash, td)
}
```

syncWithPeer

```
// syncWithPeer starts a block synchronization based on the hash chain from the
// specified peer and head hash.
func (d *Downloader) syncWithPeer(p *peerConnection, hash common.Hash, td *big.Int)
 (err error) {
    ...
    // Look up the sync boundaries: the common ancestor and the target block
    // 使用hash指来获取区块头，这个方法里面会访问网络
    latest, err := d.fetchHeight(p)
    if err != nil {
        return err
    }
    height := latest.Number.Uint64()
    // findAncestor试图来获取大家共同的祖先，以便找到一个开始同步的点。
    origin, err := d.findAncestor(p, height)
    if err != nil {
        return err
    }
    d.syncStatsLock.Lock()
    if d.syncStatsChainHeight <= origin || d.syncStatsChainOrigin > origin {
        d.syncStatsChainOrigin = origin
    }
    d.syncStatsChainHeight = height
    d.syncStatsLock.Unlock()

    // Initiate the sync using a concurrent header and content retrieval algorithm
    pivot := uint64(0)
    switch d.mode {
    case LightSync:
        pivot = height
    case FastSync:
        // Calculate the new fast/slow sync pivot point
        // 如果pivot这个点没有被锁定。
        if d.fsPivotLock == nil {
            pivotOffset, err := rand.Int(rand.Reader, big.NewInt(int64(fsPivotInter
val)))
            if err != nil {
                panic(fmt.Sprintf("Failed to access crypto random source: %v", err)
)
            }
            if height > uint64(fsMinFullBlocks)+pivotOffset.Uint64() {
                pivot = height - uint64(fsMinFullBlocks) - pivotOffset.Uint64()
            }
        } else { // 如过这个点已经被锁定了。那么就使用这个点
            // Pivot point locked in, use this and do not pick a new one!
            pivot = d.fsPivotLock.Number.Uint64()
```

```
        }
        // If the point is below the origin, move origin back to ensure state downl
oad
        if pivot < origin {
            if pivot > 0 {
                origin = pivot - 1
            } else {
                origin = 0
            }
        }
        log.Debug("Fast syncing until pivot block", "pivot", pivot)
    }
    d.queue.Prepare(origin+1, d.mode, pivot, latest)
    if d.syncInitHook != nil {
        d.syncInitHook(origin, height)
    }
    // 启动几个fetcher 分别负责header,bodies,receipts,处理headers
    fetchers := []func() error{
        func() error { return d.fetchHeaders(p, origin+1) }, // Headers are always
retrieved
        func() error { return d.fetchBodies(origin + 1) },   // Bodies are retrieve
d during normal and fast sync
        func() error { return d.fetchReceipts(origin + 1) }, // Receipts are retrie
ved during fast sync
        func() error { return d.processHeaders(origin+1, td) },
    }
    if d.mode == FastSync {  //根据模式的不同，增加新的处理逻辑
        fetchers = append(fetchers, func() error { return d.processFastSyncContent(
latest) })
    } else if d.mode == FullSync {
        fetchers = append(fetchers, d.processFullSyncContent)
    }
    err = d.spawnSync(fetchers)
    if err != nil && d.mode == FastSync && d.fsPivotLock != nil {
        // If sync failed in the critical section, bump the fail counter.
        atomic.AddUint32(&d.fsPivotFails, 1)
    }
    return err
}
```

spawnSync给每个fetcher启动一个goroutine, 然后阻塞的等待fetcher出错。

```
// spawnSync runs d.process and all given fetcher functions to completion in
// separate goroutines, returning the first error that appears.
func (d *Downloader) spawnSync(fetchers []func() error) error {
    var wg sync.WaitGroup
    errc := make(chan error, len(fetchers))
```

```
    wg.Add(len(fetchers))
    for _, fn := range fetchers {
        fn := fn
        go func() { defer wg.Done(); errc <- fn() }()
    }
    // Wait for the first error, then terminate the others.
    var err error
    for i := 0; i < len(fetchers); i++ {
        if i == len(fetchers)-1 {
            // Close the queue when all fetchers have exited.
            // This will cause the block processor to end when
            // it has processed the queue.
            d.queue.Close()
        }
        if err = <-errc; err != nil {
            break
        }
    }
    d.queue.Close()
    d.Cancel()
    wg.Wait()
    return err
}
```

headers的处理

fetchHeaders方法用来获取header。 然后根据获取的header去获取body和receipt等信息。

```
// fetchHeaders keeps retrieving headers concurrently from the number
// requested, until no more are returned, potentially throttling on the way. To
// facilitate concurrency but still protect against malicious nodes sending bad
// headers, we construct a header chain skeleton using the "origin" peer we are
// syncing with, and fill in the missing headers using anyone else. Headers from
// other peers are only accepted if they map cleanly to the skeleton. If no one
// can fill in the skeleton - not even the origin peer - it's assumed invalid and
// the origin is dropped.
```

fetchHeaders不断的重复这样的操作，发送header请求，等待所有的返回。直到完成所有的header请求。 为了提高并发性，同时仍然能够防止恶意节点发送错误的header，我们使用我们正在同步的“origin”peer构造一个头文件链骨架，并使用其他人填充缺失的header。 其他peer的header只有在干净地映射到骨架上时才被接受。 如果没有人能够填充骨架 - 甚至origin peer也不能填充 - 它被认为是无效的，并且origin peer也被丢弃。

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64) error {
    p.log.Debug("Directing header downloads", "origin", from)
    defer p.log.Debug("Header download terminated")
```

```
    // Create a timeout timer, and the associated header fetcher
    skeleton := true            // Skeleton assembly phase or finishing up
    request := time.Now()       // time of the last skeleton fetch request
    timeout := time.NewTimer(0) // timer to dump a non-responsive active peer
    <-timeout.C                 // timeout channel should be initially empty
    defer timeout.Stop()

    var ttl time.Duration
    getHeaders := func(from uint64) {
        request = time.Now()

        ttl = d.requestTTL()
        timeout.Reset(ttl)

        if skeleton { //填充骨架
            p.log.Trace("Fetching skeleton headers", "count", MaxHeaderFetch, "from
", from)
            go p.peer.RequestHeadersByNumber(from+uint64(MaxHeaderFetch)-1, MaxSkel
etonSize, MaxHeaderFetch-1, false)
        } else { // 直接请求
            p.log.Trace("Fetching full headers", "count", MaxHeaderFetch, "from", f
rom)
            go p.peer.RequestHeadersByNumber(from, MaxHeaderFetch, 0, false)
        }
    }
    // Start pulling the header chain skeleton until all is done
    getHeaders(from)

    for {
        select {
        case <-d.cancelCh:
            return errCancelHeaderFetch

        case packet := <-d.headerCh: //网络上返回的header会投递到headerCh这个通道
            // Make sure the active peer is giving us the skeleton headers
            if packet.PeerId() != p.id {
                log.Debug("Received skeleton from incorrect peer", "peer", packet.P
eerId())
                break
            }
            headerReqTimer.UpdateSince(request)
            timeout.Stop()

            // If the skeleton's finished, pull any remaining head headers directly
 from the origin
            if packet.Items() == 0 && skeleton {
                skeleton = false
                getHeaders(from)
```

```
                continue
            }
            // If no more headers are inbound, notify the content fetchers and retu
rn
            // 如果没有更多的返回了。 那么告诉headerProcCh通道
            if packet.Items() == 0 {
                p.log.Debug("No more headers available")
                select {
                case d.headerProcCh <- nil:
                    return nil
                case <-d.cancelCh:
                    return errCancelHeaderFetch
                }
            }
            headers := packet.(*headerPack).headers

            // If we received a skeleton batch, resolve internals concurrently
            if skeleton { // 如果是需要填充骨架，那么在这个方法里面填充好
                filled, proced, err := d.fillHeaderSkeleton(from, headers)
                if err != nil {
                    p.log.Debug("Skeleton chain invalid", "err", err)
                    return errInvalidChain
                }
                headers = filled[proced:]
                // proced代表已经处理完了多少个了。  所以只需要proced:后面的headers了
                from += uint64(proced)
            }
            // Insert all the new headers and fetch the next batch
            if len(headers) > 0 {
                p.log.Trace("Scheduling new headers", "count", len(headers), "from"
, from)
                //投递到headerProcCh 然后继续循环。
                select {
                case d.headerProcCh <- headers:
                case <-d.cancelCh:
                    return errCancelHeaderFetch
                }
                from += uint64(len(headers))
            }
            getHeaders(from)

        case <-timeout.C:
            // Header retrieval timed out, consider the peer bad and drop
            p.log.Debug("Header request timed out", "elapsed", ttl)
            headerTimeoutMeter.Mark(1)
            d.dropPeer(p.id)

            // Finish the sync gracefully instead of dumping the gathered data thou
```

```
gh
            for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                select {
                case ch <- false:
                case <-d.cancelCh:
                }
            }
            select {
            case d.headerProcCh <- nil:
            case <-d.cancelCh:
            }
            return errBadPeer
        }
    }
}
```

processHeaders方法，这个方法从headerProcCh通道来获取header。并把获取到的header丢入到queue来进行调度，这样body fetcher或者是receipt fetcher就可以领取到fetch任务。

```
// processHeaders takes batches of retrieved headers from an input channel and
// keeps processing and scheduling them into the header chain and downloader's
// queue until the stream ends or a failure occurs.
// processHeaders批量的获取headers， 处理他们，并通过downloader的queue对象来调度他们。直到错误发生或者处理结束。
func (d *Downloader) processHeaders(origin uint64, td *big.Int) error {
    // Calculate the pivoting point for switching from fast to slow sync
    pivot := d.queue.FastSyncPivot()

    // Keep a count of uncertain headers to roll back
    // rollback 用来处理这种逻辑，如果某个点失败了。那么之前插入的2048个节点都要回滚。因为安全性达不到要求， 可以详细参考fast sync的文档。
    rollback := []*types.Header{}
    defer func() { // 这个函数用来错误退出的时候进行回滚。 TODO
        if len(rollback) > 0 {
            // Flatten the headers and roll them back
            hashes := make([]common.Hash, len(rollback))
            for i, header := range rollback {
                hashes[i] = header.Hash()
            }
            lastHeader, lastFastBlock, lastBlock := d.lightchain.CurrentHeader().Number, common.Big0, common.Big0
            if d.mode != LightSync {
                lastFastBlock = d.blockchain.CurrentFastBlock().Number()
                lastBlock = d.blockchain.CurrentBlock().Number()
            }
            d.lightchain.Rollback(hashes)
            curFastBlock, curBlock := common.Big0, common.Big0
```

```
            if d.mode != LightSync {
                curFastBlock = d.blockchain.CurrentFastBlock().Number()
                curBlock = d.blockchain.CurrentBlock().Number()
            }
            log.Warn("Rolled back headers", "count", len(hashes),
                "header", fmt.Sprintf("%d->%d", lastHeader, d.lightchain.CurrentHea
der().Number),
                "fast", fmt.Sprintf("%d->%d", lastFastBlock, curFastBlock),
                "block", fmt.Sprintf("%d->%d", lastBlock, curBlock))

            // If we're already past the pivot point, this could be an attack, thre
ad carefully
            if rollback[len(rollback)-1].Number.Uint64() > pivot {
                // If we didn't ever fail, lock in the pivot header (must! not! cha
nge!)
                if atomic.LoadUint32(&d.fsPivotFails) == 0 {
                    for _, header := range rollback {
                        if header.Number.Uint64() == pivot {
                            log.Warn("Fast-sync pivot locked in", "number", pivot,
"hash", header.Hash())
                            d.fsPivotLock = header
                        }
                    }
                }
            }
        }
    }()

    // Wait for batches of headers to process
    gotHeaders := false

    for {
        select {
        case <-d.cancelCh:
            return errCancelHeaderProcessing

        case headers := <-d.headerProcCh:
            // Terminate header processing if we synced up
            if len(headers) == 0 { //处理完成
                // Notify everyone that headers are fully processed
                for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                    select {
                    case ch <- false:
                    case <-d.cancelCh:
                    }
                }
                // If no headers were retrieved at all, the peer violated it's TD p
romise that it had a
```

```
                // better chain compared to ours. The only exception is if it's pro
mised blocks were
                // already imported by other means (e.g. fecher):
                //
                // R <remote peer>, L <local node>: Both at block 10
                // R: Mine block 11, and propagate it to L
                // L: Queue block 11 for import
                // L: Notice that R's head and TD increased compared to ours, start
 sync
                // L: Import of block 11 finishes
                // L: Sync begins, and finds common ancestor at 11
                // L: Request new headers up from 11 (R's TD was higher, it must ha
ve something)
                // R: Nothing to give
                if d.mode != LightSync { // 对方的TD比我们大，但是没有获取到任何东西。
那么认为对方是错误的对方。 会断开和对方的联系
                    if !gotHeaders && td.Cmp(d.blockchain.GetTdByHash(d.blockchain.
CurrentBlock().Hash())) > 0 {
                        return errStallingPeer
                    }
                }
                // If fast or light syncing, ensure promised headers are indeed del
ivered. This is
                // needed to detect scenarios where an attacker feeds a bad pivot a
nd then bails out
                // of delivering the post-pivot blocks that would flag the invalid
content.
                //
                // This check cannot be executed "as is" for full imports, since bl
ocks may still be
                // queued for processing when the header download completes. Howeve
r, as long as the
                // peer gave us something useful, we're already happy/progressed (a
bove check).
                if d.mode == FastSync || d.mode == LightSync {
                    if td.Cmp(d.lightchain.GetTdByHash(d.lightchain.CurrentHeader()
.Hash())) > 0 {
                        return errStallingPeer
                    }
                }
                // Disable any rollback and return
                rollback = nil
                return nil
            }
            // Otherwise split the chunk of headers into batches and process them
            gotHeaders = true

            for len(headers) > 0 {
```

```
                // Terminate if something failed in between processing chunks
                select {
                case <-d.cancelCh:
                    return errCancelHeaderProcessing
                default:
                }
                // Select the next chunk of headers to import
                limit := maxHeadersProcess
                if limit > len(headers) {
                    limit = len(headers)
                }
                chunk := headers[:limit]

                // In case of header only syncing, validate the chunk immediately
                if d.mode == FastSync || d.mode == LightSync { //如果是快速同步模式,
或者是轻量级同步模式(只下载区块头)
                    // Collect the yet unknown headers to mark them as uncertain
                    unknown := make([]*types.Header, 0, len(headers))
                    for _, header := range chunk {
                        if !d.lightchain.HasHeader(header.Hash(), header.Number.Uin
t64()) {
                            unknown = append(unknown, header)
                        }
                    }
                    // If we're importing pure headers, verify based on their recen
tness
                    // 每隔多少个区块验证一次
                    frequency := fsHeaderCheckFrequency
                    if chunk[len(chunk)-1].Number.Uint64()+uint64(fsHeaderForceVeri
fy) > pivot {
                        frequency = 1
                    }
                    // lightchain默认是等于chain的。 插入区块头。如果失败那么需要回滚。
                    if n, err := d.lightchain.InsertHeaderChain(chunk, frequency);
err != nil {
                        // If some headers were inserted, add them too to the rollb
ack list
                        if n > 0 {
                            rollback = append(rollback, chunk[:n]...)
                        }
                        log.Debug("Invalid header encountered", "number", chunk[n].
Number, "hash", chunk[n].Hash(), "err", err)
                        return errInvalidChain
                    }
                    // All verifications passed, store newly found uncertain header
s
                    rollback = append(rollback, unknown...)
                    if len(rollback) > fsHeaderSafetyNet {
```

```
                        rollback = append(rollback[:0], rollback[len(rollback)-fsHe
aderSafetyNet:]...)
                    }
                }
                // If we're fast syncing and just pulled in the pivot, make sure it
's the one locked in
                if d.mode == FastSync && d.fsPivotLock != nil && chunk[0].Number.Ui
nt64() <= pivot && chunk[len(chunk)-1].Number.Uint64() >= pivot { //如果PivotLock,检
查一下Hash是否相同。
                    if pivot := chunk[int(pivot-chunk[0].Number.Uint64())]; pivot.H
ash() != d.fsPivotLock.Hash() {
                        log.Warn("Pivot doesn't match locked in one", "remoteNumber
", pivot.Number, "remoteHash", pivot.Hash(), "localNumber", d.fsPivotLock.Number, "
localHash", d.fsPivotLock.Hash())
                        return errInvalidChain
                    }
                }
                // Unless we're doing light chains, schedule the headers for associ
ated content retrieval
                // 如果我们处理完轻量级链。 调度header来进行相关数据的获取。body，receip
ts
                if d.mode == FullSync || d.mode == FastSync {
                    // If we've reached the allowed number of pending headers, stal
l a bit
                    // 如果当前queue的容量容纳不下了。那么等待。
                    for d.queue.PendingBlocks() >= maxQueuedHeaders || d.queue.Pend
ingReceipts() >= maxQueuedHeaders {
                        select {
                        case <-d.cancelCh:
                            return errCancelHeaderProcessing
                        case <-time.After(time.Second):
                        }
                    }
                    // Otherwise insert the headers for content retrieval
                    // 调用Queue进行调度，下载body和receipts
                    inserts := d.queue.Schedule(chunk, origin)
                    if len(inserts) != len(chunk) {
                        log.Debug("Stale headers")
                        return errBadPeer
                    }
                }
                headers = headers[limit:]
                origin += uint64(limit)
            }
            // Signal the content downloaders of the availablility of new tasks
            // 给通道d.bodyWakeCh，d.receiptWakeCh发送消息，唤醒处理线程。
            for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                select {
```

```
                case ch <- true:
                default:
                }
            }
        }
    }
}
```

bodies处理

fetchBodies函数定义了一些闭包函数，然后调用了fetchParts函数

```
// fetchBodies iteratively downloads the scheduled block bodies, taking any
// available peers, reserving a chunk of blocks for each, waiting for delivery
// and also periodically checking for timeouts.
// fetchBodies 持续的下载区块体，中间会使用到任何可以用的链接，为每一个链接保留一部分的区块体，等待区块被交付，并定期的检查是否超时。
func (d *Downloader) fetchBodies(from uint64) error {
    log.Debug("Downloading block bodies", "origin", from)

    var (
        deliver = func(packet dataPack) (int, error) { //下载完的区块体的交付函数
            pack := packet.(*bodyPack)
            return d.queue.DeliverBodies(pack.peerId, pack.transactions, pack.uncle
s)
        }
        expire   = func() map[string]int { return d.queue.ExpireBodies(d.requestTTL
()) }  //超时
        fetch    = func(p *peerConnection, req *fetchRequest) error { return p.Fetc
hBodies(req) }  // fetch函数
        capacity = func(p *peerConnection) int { return p.BlockCapacity(d.requestRT
T()) } // 对端的吞吐量
        setIdle  = func(p *peerConnection, accepted int) { p.SetBodiesIdle(accepted
) } // 设置peer为idle
    )
    err := d.fetchParts(errCancelBodyFetch, d.bodyCh, deliver, d.bodyWakeCh, expire
,
        d.queue.PendingBlocks, d.queue.InFlightBlocks, d.queue.ShouldThrottleBlocks
, d.queue.ReserveBodies,
        d.bodyFetchHook, fetch, d.queue.CancelBodies, capacity, d.peers.BodyIdlePee
rs, setIdle, "bodies")

    log.Debug("Block body download terminated", "err", err)
    return err
}
```

fetchParts

```
// fetchParts iteratively downloads scheduled block parts, taking any available
// peers, reserving a chunk of fetch requests for each, waiting for delivery and
// also periodically checking for timeouts.
// fetchParts迭代地下载预定的块部分，取得任何可用的对等体，为每个部分预留大量的提取请求，等待交付并且还定期检查超时。
// As the scheduling/timeout logic mostly is the same for all downloaded data
// types, this method is used by each for data gathering and is instrumented with
// various callbacks to handle the slight differences between processing them.
// 由于调度/超时逻辑对于所有下载的数据类型大部分是相同的，所以这个方法被用于不同的区块类型的下载，并且用各种回调函数来处理它们之间的细微差别。
// The instrumentation parameters:
//  - errCancel:   error type to return if the fetch operation is cancelled (mostly makes logging nicer) 如果fetch操作被取消，会在这个通道上发送数据
//  - deliveryCh:  channel from which to retrieve downloaded data packets (merged from all concurrent peers) 数据被下载完成后投递的目的地
//  - deliver:     processing callback to deliver data packets into type specific download queues (usually within `queue`) 处理完成后数据被投递到哪个队列
//  - wakeCh:      notification channel for waking the fetcher when new tasks are available (or sync completed) 用来通知fetcher 新的任务到来，或者是同步完成
//  - expire:      task callback method to abort requests that took too long and return the faulty peers (traffic shaping)  因为超时来终止请求的回调函数。
//  - pending:     task callback for the number of requests still needing download (detect completion/non-completability) 还需要下载的任务的数量。
//  - inFlight:    task callback for the number of in-progress requests (wait for all active downloads to finish) 正在处理过程中的请求数量
//  - throttle:    task callback to check if the processing queue is full and activate throttling (bound memory use) 用来检查处理队列是否满的回调函数。
//  - reserve:     task callback to reserve new download tasks to a particular peer (also signals partial completions)  用来为某个peer来预定任务的回调函数
//  - fetchHook:   tester callback to notify of new tasks being initiated (allows testing the scheduling logic)
//  - fetch:       network callback to actually send a particular download request to a physical remote peer //发送网络请求的回调函数
//  - cancel:      task callback to abort an in-flight download request and allow rescheduling it (in case of lost peer)  用来取消正在处理的任务的回调函数
//  - capacity:    network callback to retrieve the estimated type-specific bandwidth capacity of a peer (traffic shaping) 网络容量或者是带宽。
//  - idle:        network callback to retrieve the currently (type specific) idle peers that can be assigned tasks  peer是否空闲的回调函数
//  - setIdle:     network callback to set a peer back to idle and update its estimated capacity (traffic shaping)  设置peer为空闲的回调函数
//  - kind:        textual label of the type being downloaded to display in log mes ages   下载类型，用于日志
func (d *Downloader) fetchParts(errCancel error, deliveryCh chan dataPack, deliver
func(dataPack) (int, error), wakeCh chan bool,
```

```
    expire func() map[string]int, pending func() int, inFlight func() bool, throttl
e func() bool, reserve func(*peerConnection, int) (*fetchRequest, bool, error),
    fetchHook func([]*types.Header), fetch func(*peerConnection, *fetchRequest) err
or, cancel func(*fetchRequest), capacity func(*peerConnection) int,
    idle func() ([]*peerConnection, int), setIdle func(*peerConnection, int), kind
string) error {

    // Create a ticker to detect expired retrieval tasks
    ticker := time.NewTicker(100 * time.Millisecond)
    defer ticker.Stop()

    update := make(chan struct{}, 1)

    // Prepare the queue and fetch block parts until the block header fetcher's don
e
    finished := false
    for {
        select {
        case <-d.cancelCh:
            return errCancel

        case packet := <-deliveryCh:
            // If the peer was previously banned and failed to deliver it's pack
            // in a reasonable time frame, ignore it's message.
            // 如果peer在之前被禁止而且没有在合适的时间deliver它的数据，那么忽略这个数据
            if peer := d.peers.Peer(packet.PeerId()); peer != nil {
                // Deliver the received chunk of data and check chain validity
                accepted, err := deliver(packet)
                if err == errInvalidChain {
                    return err
                }
                // Unless a peer delivered something completely else than requested
 (usually
                // caused by a timed out request which came through in the end), se
t it to
                // idle. If the delivery's stale, the peer should have already been
 idled.
                if err != errStaleDelivery {
                    setIdle(peer, accepted)
                }
                // Issue a log to the user to see what's going on
                switch {
                case err == nil && packet.Items() == 0:
                    peer.log.Trace("Requested data not delivered", "type", kind)
                case err == nil:
                    peer.log.Trace("Delivered new batch of data", "type", kind, "co
unt", packet.Stats())
                default:
```

```
                    peer.log.Trace("Failed to deliver retrieved data", "type", kind
, "err", err)
                }
            }
            // Blocks assembled, try to update the progress
            select {
            case update <- struct{}{}:
            default:
            }

        case cont := <-wakeCh:
            // The header fetcher sent a continuation flag, check if it's done
            // 当所有的任务完成的时候会写入这个队列。
            if !cont {
                finished = true
            }
            // Headers arrive, try to update the progress
            select {
            case update <- struct{}{}:
            default:
            }

        case <-ticker.C:
            // Sanity check update the progress
            select {
            case update <- struct{}{}:
            default:
            }

        case <-update:
            // Short circuit if we lost all our peers
            if d.peers.Len() == 0 {
                return errNoPeers
            }
            // Check for fetch request timeouts and demote the responsible peers
            for pid, fails := range expire() {
                if peer := d.peers.Peer(pid); peer != nil {
                    // If a lot of retrieval elements expired, we might have overes
timated the remote peer or perhaps
                    // ourselves. Only reset to minimal throughput but don't drop j
ust yet. If even the minimal times
                    // out that sync wise we need to get rid of the peer.
                    //如果很多检索元素过期，我们可能高估了远程对象或者我们自己。 只能重置
为最小的吞吐量，但不要丢弃。 如果即使最小的同步任然超时，我们需要删除peer。
                    // The reason the minimum threshold is 2 is because the downloa
der tries to estimate the bandwidth
                    // and latency of a peer separately, which requires pushing the
 measures capacity a bit and seeing
```

```
                    // how response times reacts, to it always requests one more th
an the minimum (i.e. min 2).
                    // 最小阈值为2的原因是因为下载器试图分别估计对等体的带宽和等待时间,
这需要稍微推动测量容量并且看到响应时间如何反应，总是要求比最小值（即，最小值2）。
                    if fails > 2 {
                        peer.log.Trace("Data delivery timed out", "type", kind)
                        setIdle(peer, 0)
                    } else {
                        peer.log.Debug("Stalling delivery, dropping", "type", kind)
                        d.dropPeer(pid)
                    }
                }
            }
            // If there's nothing more to fetch, wait or terminate
            // 任务全部完成。 那么退出
            if pending() == 0 { //如果没有等待分配的任务， 那么break。不用执行下面的代码
了。
                if !inFlight() && finished {
                    log.Debug("Data fetching completed", "type", kind)
                    return nil
                }
                break
            }
            // Send a download request to all idle peers, until throttled
            progressed, throttled, running := false, false, inFlight()
            idles, total := idle()

            for _, peer := range idles {
                // Short circuit if throttling activated
                if throttle() {
                    throttled = true
                    break
                }
                // Short circuit if there is no more available task.
                if pending() == 0 {
                    break
                }
                // Reserve a chunk of fetches for a peer. A nil can mean either tha
t
                // no more headers are available, or that the peer is known not to
                // have them.
                // 为某个peer请求分配任务。
                request, progress, err := reserve(peer, capacity(peer))
                if err != nil {
                    return err
                }
                if progress {
                    progressed = true
```

```
                }
                if request == nil {
                    continue
                }
                if request.From > 0 {
                    peer.log.Trace("Requesting new batch of data", "type", kind, "f
rom", request.From)
                } else if len(request.Headers) > 0 {
                    peer.log.Trace("Requesting new batch of data", "type", kind, "c
ount", len(request.Headers), "from", request.Headers[0].Number)
                } else {
                    peer.log.Trace("Requesting new batch of data", "type", kind, "c
ount", len(request.Hashes))
                }
                // Fetch the chunk and make sure any errors return the hashes to th
e queue
                if fetchHook != nil {
                    fetchHook(request.Headers)
                }
                if err := fetch(peer, request); err != nil {
                    // Although we could try and make an attempt to fix this, this
error really
                    // means that we've double allocated a fetch task to a peer. If
 that is the
                    // case, the internal state of the downloader and the queue is
very wrong so
                    // better hard crash and note the error instead of silently acc
umulating into
                    // a much bigger issue.
                    panic(fmt.Sprintf("%v: %s fetch assignment failed", peer, kind)
)
                }
                running = true
            }
            // Make sure that we have peers available for fetching. If all peers ha
ve been tried
            // and all failed throw an error
            if !progressed && !throttled && !running && len(idles) == total && pend
ing() > 0 {
                return errPeersUnavailable
            }
        }
    }
}
```

receipt的处理

receipt的处理和body类似。

```
// fetchReceipts iteratively downloads the scheduled block receipts, taking any
// available peers, reserving a chunk of receipts for each, waiting for delivery
// and also periodically checking for timeouts.
func (d *Downloader) fetchReceipts(from uint64) error {
    log.Debug("Downloading transaction receipts", "origin", from)

    var (
        deliver = func(packet dataPack) (int, error) {
            pack := packet.(*receiptPack)
            return d.queue.DeliverReceipts(pack.peerId, pack.receipts)
        }
        expire   = func() map[string]int { return d.queue.ExpireReceipts(d.requestT
TL()) }
        fetch    = func(p *peerConnection, req *fetchRequest) error { return p.Fetc
hReceipts(req) }
        capacity = func(p *peerConnection) int { return p.ReceiptCapacity(d.request
RTT()) }
        setIdle  = func(p *peerConnection, accepted int) { p.SetReceiptsIdle(accept
ed) }
    )
    err := d.fetchParts(errCancelReceiptFetch, d.receiptCh, deliver, d.receiptWakeC
h, expire,
        d.queue.PendingReceipts, d.queue.InFlightReceipts, d.queue.ShouldThrottleRe
ceipts, d.queue.ReserveReceipts,
        d.receiptFetchHook, fetch, d.queue.CancelReceipts, capacity, d.peers.Receip
tIdlePeers, setIdle, "receipts")

    log.Debug("Transaction receipt download terminated", "err", err)
    return err
}
```

processFastSyncContent 和 processFullSyncContent

```
// processFastSyncContent takes fetch results from the queue and writes them to the
// database. It also controls the synchronisation of state nodes of the pivot block
.
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.
    // 启动状态同步
    stateSync := d.syncState(latest.Root)
```

```
    defer stateSync.Cancel()
    go func() {
        if err := stateSync.Wait(); err != nil {
            d.queue.Close() // wake up WaitResults
        }
    }()

    pivot := d.queue.FastSyncPivot()
    for {
        results := d.queue.WaitResults() // 等待队列输出处理完成的区块
        if len(results) == 0 {
            return stateSync.Cancel()
        }
        if d.chainInsertHook != nil {
            d.chainInsertHook(results)
        }
        P, beforeP, afterP := splitAroundPivot(pivot, results)
        // 插入fast sync的数据
        if err := d.commitFastSyncData(beforeP, stateSync); err != nil {
            return err
        }
        if P != nil {
            // 如果已经达到了 pivot point 那么等待状态同步完成,
            stateSync.Cancel()
            if err := d.commitPivotBlock(P); err != nil {
                return err
            }
        }
        // 对于pivot point 之后的所有节点, 都需要按照完全的处理。
        if err := d.importBlockResults(afterP); err != nil {
            return err
        }
    }
}
```

processFullSyncContent,比较简单。 从队列里面获取区块然后插入。

```
// processFullSyncContent takes fetch results from the queue and imports them into
the chain.
func (d *Downloader) processFullSyncContent() error {
    for {
        results := d.queue.WaitResults()
        if len(results) == 0 {
            return nil
        }
        if d.chainInsertHook != nil {
            d.chainInsertHook(results)
```

```
        }
        if err := d.importBlockResults(results); err != nil {
            return err
        }
    }
}
```