fetcher源码解析

1、简介

Fetcher是广播的同步工具，它和ProtocolManager的handlerMsg()方法息息相关，Fetcher是一个累积块通知的模块，然后对这些通知进行管理和调度，取得完整的block加入本地。

2、源码分析

先介绍下Fetcher的数据结构，如下：

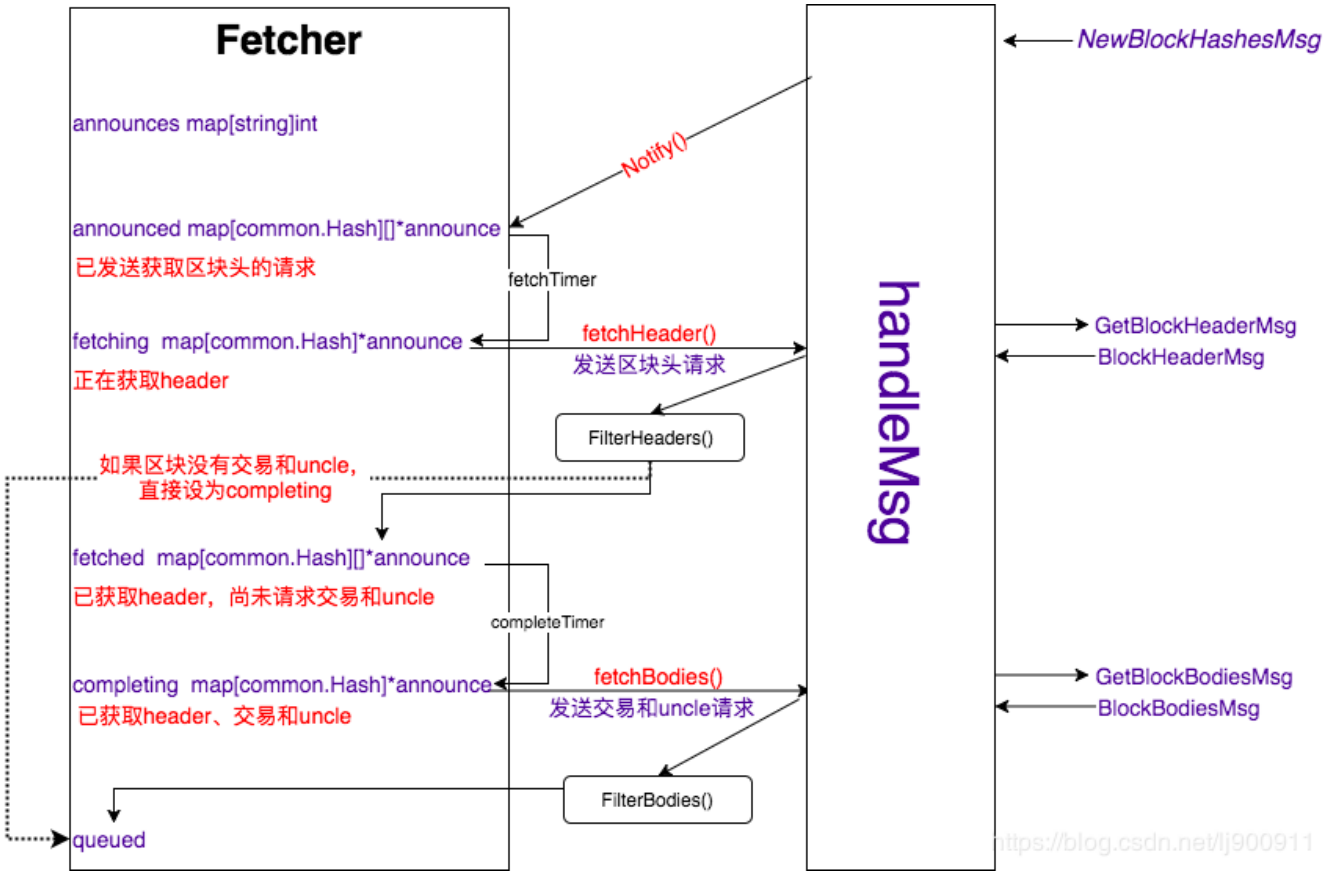
```
type Fetcher struct {

    notify chan *announce   // 收到区块hash值的通道
    inject chan *inject    // 收到完整区块的通道

    blockFilter chan chan []*types.Block
    headerFilter chan chan *headerFilterTask    // 过滤header的通道的通道
    bodyFilter chan chan *bodyFilterTask    // 过滤body的通道的通道

    done chan common.Hash
    quit chan struct{}

    // Announce states
    announces map[string]int    // Peer已经给了本节点多少区块头通知
    announced map[common.Hash][]*announce    // 已经announced的区块列表
    fetching map[common.Hash]*announce     // 正在fetching区块头的请求
    fetched map[common.Hash][]*announce     // 已经fetch到区块头，还差body的请求，用来获取body
    completing map[common.Hash]*announce    // 已经得到区块头的

    // Block cache
    queue  *prque.Prque               // queue，优先级队列，高度做优先级
    queues map[string]int              // queues，统计peer通告了多少块
    queued map[common.Hash]*inject     // queued，代表这个块如队列了

    // Callbacks
    getBlock       blockRetrievalFn   // Retrieves a block from the local chain
    verifyHeader   headerVerifierFn   // 验证区块头，包含了Pow验证
    broadcastBlock blockBroadcasterFn // 广播给peer
    chainHeight    chainHeightFn      // Retrieves the current chain's height
    insertChain    chainInsertFn      // 插入区块到链的函数
    dropPeer       peerDropFn         // Drops a peer for misbehaving

    // Testing hooks
    announceChangeHook func(common.Hash, bool)
    queueChangeHook    func(common.Hash, bool)
    fetchingHook       func([]common.Hash)
    completingHook     func([]common.Hash)
    importedHook       func(*types.Block)
```

```
}
```

下面先看下有哪些消息处理时候用到了Fetcher：

```
case msg.Code == BlockHeadersMsg:
        ...................
        // Irrelevant of the fork checks, send the header to the fetcher just in case
        headers = pm.fetcher.FilterHeaders(p.id, headers, time.Now())
        if len(headers) > 0 || !filter {
            err := pm.downloader.DeliverHeaders(p.id, headers)
            if err != nil {
                log.Debug("Failed to deliver headers", "err", err)
            }
        }
case msg.Code == BlockBodiesMsg:
        ....................
        if filter {
            transactions, uncles = pm.fetcher.FilterBodies(p.id, transactions, uncles, time.Now())
        }
        if len(transactions) > 0 || len(uncles) > 0 || !filter {
            err := pm.downloader.DeliverBodies(p.id, transactions, uncles)
            if err != nil {
                log.Debug("Failed to deliver bodies", "err", err)
            }
        }
case msg.Code == NewBlockHashesMsg:
        ...............
        for _, block := range unknown {
            pm.fetcher.Notify(p.id, block.Hash, block.Number, time.Now(), p.RequestOneHeader,
p.RequestBodies)
        }
case msg.Code == NewBlockMsg:
        ........................
        pm.fetcher.Enqueue(p.id, request.Block)
         ........................
    }
```

上面就是使用到Fetcher的主要消息，下面来使用一张图来总览下Fetcher的流程：



先从区块消息来讲解，当一个新的区块产生时候，在ProtocolManager的区块广播执行时候会对一部分节点只发送区块的hash和number，此时一个以太坊节点收到的消息就是NewBlockHashesMsg，从上面的case语句可知，此时会调用`pm.fetcher.Notify(p.id, block.Hash, block.Number, time.Now(), p.RequestOneHeader, p.RequestBodies)`，方法传入了peer的id信息、block的hash和number、接收到消息时候的时间戳、用来请求header和blockBody的两个函数。p.RequestOneHeader, p.RequestBodies方法实现如下：

```
func (p *peer) RequestOneHeader(hash common.Hash) error {
    p.Log().Debug("Fetching single header", "hash", hash)
    return p2p.Send(p.rw,GetBlockHeadersMsg,&getBlockHeadersData{Origin:hashOrNumber{Hash: hash},
Amount: uint64(1), Skip: uint64(0), Reverse: false})
}
func (p *peer) RequestBodies(hashes []common.Hash) error {
    p.Log().Debug("Fetching batch of block bodies", "count", len(hashes))
    return p2p.Send(p.rw, GetBlockBodiesMsg, hashes)
}
```

下面介绍下Fetcher的Notify方法和Fetcher的主循环处理流程:

```
func (f *Fetcher) Notify(peer string, hash common.Hash, number uint64, time time.Time,
    headerFetcher headerRequesterFn, bodyFetcher bodyRequesterFn) error {
    // 将Notify传来的参数封装为一个announce结构体
    block := &announce{
        hash:        hash,
        number:      number,
        time:        time,
        origin:      peer,
        fetchHeader: headerFetcher,
        fetchBodies: bodyFetcher,
    }
```

```
    select {
     // 将announce传入f.notify通道，交由fetcher主循环处理
    case f.notify <- block:
        return nil
    case <-f.quit:
        return errTerminated
    }
}
case notification := <-f.notify:
     // A block was announced, make sure the peer isn't DOSing us
     propAnnounceInMeter.Mark(1)
     // announces  map[string]int  Per peer announce counts to prevent memory exhaustion 每个peer宣布数量计数，以防止内存耗尽
     count := f.announces[notification.origin] + 1
     // hashLimit=256
     if count > hashLimit {
         log.Debug("Peer exceeded outstanding announces", "peer", notification.origin, "limit", hashLimit)
         propAnnounceDOSMeter.Mark(1)
         break
     }
     // 如果是一个可验证区块，检查是否在可用范围内
     // If we have a valid block number, check that it's potentially useful
     if notification.number > 0 {
         // maxUbcleDist = 7  MaxQueueDist = 32
        if dist := int64(notification.number) - int64(f.chainHeight()); dist < -maxUncleDist || dist > maxQueueDist {
            log.Debug("Peer discarded announcement", "peer", notification.origin, "number", notification.number, "hash", notification.hash, "distance", dist)
            propAnnounceDropMeter.Mark(1)
            break
        }
    }
    // 若区块已经在f.fetching或者f.completing中存在，那么跳出循环
    // All is well, schedule the announce if block's not yet downloading
    if _, ok := f.fetching[notification.hash]; ok {
        break
    }
    if _, ok := f.completing[notification.hash]; ok {
        break
    }
    // 记录此peer广播次数
    f.announces[notification.origin] = count
    // announced  map[common.Hash][]*announce // Announced blocks, scheduled for fetching
    // 将此区块的广播信息存在f.announced中
    f.announced[notification.hash] = append(f.announced[notification.hash], notification)
    if f.announceChangeHook != nil && len(f.announced[notification.hash]) == 1 {
        f.announceChangeHook(notification.hash, true)
    }
    // 如果有任务，则重设处理任务计时器
    if len(f.announced) == 1 {
        f.rescheduleFetch(fetchTimer)
    }
```

从上面可以看出，主循环的case里面主要做了如下几件事：

1、防Dos攻击

2、检查区块高度是否在能处理的范围内

3、检查区块是否已经处于处理状态

4、将任务添加到announced中供其他case处理

下面开始看如何处理announced中的消息：

```
case <-fetchTimer.C:
    // At least one block's timer ran out, check for needing retrieval
    request := make(map[string][]common.Hash)
    // 遍历announced列表
    for hash, announces := range f.announced {
    // 由于切片中第一个元素是最早添加的，所以时间也是最早，计算现在时间距离最早添加到announced中的区块时间差大于arriveTimeout-gatherSlack的情况下
    if time.Since(announces[0].time) > arriveTimeout-gatherSlack {
    // 此处从announces随机拿出来一个announce进行处理，为什么拿出来一个是因为announced是map[hash][]announce的结构，意思存的是一个区块hash为key,所有peer广播的这个hash的区块信息为value,所以value中不同值代表不同的peer广播同一个区块，故随机拿出来一个即可
        // Pick a random peer to retrieve from, reset all others
            announce := announces[rand.Intn(len(announces))]
            f.forgetHash(hash)
        // 如果本地当前还没有获取到此区块，那么就创建获取请求，并将此hash放入fetching里面，方便其他处理逻辑查询
        // If the block still didn't arrive, queue for fetching
            if f.getBlock(hash) == nil {
                request[announce.origin] = append(request[announce.origin], hash)
                f.fetching[hash] = announce
            }
        }
    }
    // 遍历request，将每个请求发送出去
    // Send out all block header requests
    for peer, hashes := range request {
        log.Trace("Fetching scheduled headers", "peer", peer, "list", hashes)
        // Create a closure of the fetch and schedule in on a new thread
        fetchHeader, hashes := f.fetching[hashes[0]].fetchHeader, hashes
        go func() {
            if f.fetchingHook != nil {
                f.fetchingHook(hashes)
            }
            for _, hash := range hashes {
                headerFetchMeter.Mark(1)
                fetchHeader(hash) // Suboptimal, but protocol doesn't allow batch header retrievals 非最优，但是协议不予许批量拉取header
            }
        }()
    }
    // Schedule the next fetch if blocks are still pending
    f.rescheduleFetch(fetchTimer)
```

以上就是调度器的处理逻辑，主要做了如下几件事：

1、遍历announced，创建符合条件的request

2、遍历request，将所有请求发送出去

在远程节点收到GetBlockHeaderMsg的时候会给我们返回相应的BlockHeaderMsg，下面开始分析收到区块头消息后的处理方式

```
case msg.Code == BlockHeadersMsg:
    ...................
    // Irrelevant of the fork checks, send the header to the fetcher just in case
    headers = pm.fetcher.FilterHeaders(p.id, headers, time.Now())
    if len(headers) > 0 || !filter {
        err := pm.downloader.DeliverHeaders(p.id, headers)
        if err != nil {
            log.Debug("Failed to deliver headers", "err", err)
        }
}
```

上面是收到blockHeaderMsg消息时候的处理方式，主要先调用了fetcher的filterHeaders()方法，再将此方法返回的结果传递给Downloader处理，这里暂时先不介绍Downloader中方法的具体实现，先来看下FilterHeaders方法的实现

```
func (f *Fetcher) FilterHeaders(peer string, headers []*types.Header, time time.Time)
[]*types.Header {
    log.Trace("Filtering headers", "peer", peer, "headers", len(headers))
    // 创建一个通道，里面传送headerFiterTask结构体
    // Send the filter channel to the fetcher
    filter := make(chan *headerFilterTask)
    // 先将此通道传入fetcher主循环
    select {
    case f.headerFilter <- filter:
    case <-f.quit:
        return nil
    }
    // 向filter通道发送headerFilterTask任务，此时需要转到主循环逻辑讲解
    // Request the filtering of the header list
    select {
    case filter <- &headerFilterTask{peer: peer, headers: headers, time: time}:
    case <-f.quit:
        return nil
    }
    // 再从filter通道取出数据作为函数结果返回
    // Retrieve the headers remaining after filtering
    select {
    case task := <-filter:
        return task.headers
    case <-f.quit:
        return nil
    }
}
```

上面是FilterHeaders方法的主要逻辑，主要做了如下几件事：

1、创建一个传送hederFilterTask类型数据的通道

2、将此通道和主循环通道连接

3、将headerFilterTask任务放入通道让主循环进行处理，因为传入的是指针，所以主循环对通道中的数据修改对此方法可见

4、从filter通道再读取主逻辑处理完（filter完）任务数据，将相关数据作为函数结果返回

下面我们来看下主循环如何处理FilterHeaderTask的：

```
case filter := <-f.headerFilter:
    // Headers arrived from a remote peer. Extract those that were explicitly
    // requested by the fetcher, and return everything else so it's delivered
    // to other parts of the system.
    var task *headerFilterTask
    select {
    // 从filter通道读取filterHeaderTask对象
    case task = <-filter:
    case <-f.quit:
        return
    }
    headerFilterInMeter.Mark(int64(len(task.headers)))

    // Split the batch of headers into unknown ones (to return to the caller),
    // known incomplete ones (requiring body retrievals) and completed blocks.
    unknown, incomplete, complete := []*types.Header{}, []*announce{}, []*types.Block{}
    // 遍历task中的headers
    for _, header := range task.headers {
        hash := header.Hash() // 取出header中的hash
        // 如下是个重要逻辑判断，先从fetching中根据此次循环中的hash获取此次请求的，若存在则说明是我们发出去的请求，那么紧接着判断task.peer跟当时广播此header的peer是否同一个，且fetched,completing,queued中不存在这个hash记录，那么进入if逻辑
        // Filter fetcher-requested headers from other synchronisation algorithms
        if announce := f.fetching[hash]; announce != nil && announce.origin == task.peer &&
f.fetched[hash] == nil && f.completing[hash] == nil && f.queued[hash] == nil {
            // If the delivered header does not match the promised number, drop the announcer
如果上面if的条件符合了，还需满足如下if条件，就是当时请求的区块号和当前返回的区块号是不是一样，若不一样则用fetcher drop掉相关peer和忘记此hash，跳过此次循环
            if header.Number.Uint64() != announce.number {
                log.Trace("Invalid block number fetched", "peer", announce.origin, "hash",
header.Hash(), "announced", announce.number, "provided", header.Number)
                f.dropPeer(announce.origin)
                f.forgetHash(hash)
                continue
            }
             // 如果从本地链中获取不到此hash的block，那么才做处理，若已经存在此hash对应的block，那么简单的忘记此hash的处理
            // Only keep if not imported by other means
            if f.getBlock(hash) == nil {
                announce.header = header
                announce.time = task.time

                // If the block is empty (header only), short circuit into the final import queue
如果此block中不包含交易和叔块数据，那么相当于一个不含叔块的空快，那么跳过获取其body，直接将此block添加到complete中，要不然添加到incomplete中等待主循环进一步获取其body
                if header.TxHash == types.DeriveSha(types.Transactions{}) && header.UncleHash ==
types.CalcUncleHash([]*types.Header{}) {
                    log.Trace("Block empty, skipping body retrieval", "peer", announce.origin,
"number", header.Number, "hash", header.Hash())
```

```
                    block := types.NewBlockWithHeader(header)
                    block.ReceivedAt = task.time

                    complete = append(complete, block)
                    f.completing[hash] = announce
                    continue
                }
                // Otherwise add to the list of blocks needing completion
                incomplete = append(incomplete, announce)
            } else {
                log.Trace("Block already imported, discarding header", "peer", announce.origin,
"number", header.Number, "hash", header.Hash())
                f.forgetHash(hash)
            }
        } else {
             // 如果上面for循环进来后第一个if语句不满足，那么就将此header放入unknown中
            // Fetcher doesn't know about it, add to the return list
            unknown = append(unknown, header)
        }
    }
    headerFilterOutMeter.Mark(int64(len(unknown)))
    select {
     // 最后将过滤完的headerFilterTask结构体重新返回给FilterHeaders()方法
    case filter <- &headerFilterTask{headers: unknown, time: task.time}:
    case <-f.quit:
        return
    }
    // 开始遍历incomplete列表
    // Schedule the retrieved headers for body completion
    for _, announce := range incomplete {
        hash := announce.header.Hash()
        // 如果已经存在于f.completing中，那么跳过此次循环
         if _, ok := f.completing[hash]; ok {
            continue
        }
         // 将此hash的数据加入f.fetched中，待其他主循环完成其后续操作
        f.fetched[hash] = append(f.fetched[hash], announce)
        if len(f.fetched) == 1 {
             // 重设定时器，将调用Complete.C
            f.rescheduleComplete(completeTimer)
        }
    }
    //遍历complete中数据，符合条件的调用f.enqueue进行处理
    // Schedule the header-only blocks for import
    for _, block := range complete {
        if announce := f.completing[block.Hash()]; announce != nil {
            f.enqueue(announce.origin, block)
        }
    }
```

上面是主循环中filterHeader通道的主要处理逻辑，主要做了以下几件事情：

1、从filter通道读取相应task

2、遍历task中的headers，根据条件将其放入unknow、incomplete、complete三个切片中

3、将unknow、incomplete、complete三个切片中的数据分类处理

下面分析下Complete.C通道和f.enqueue方法的处理逻辑

```
case <-completeTimer.C:
    // At least one header's timer ran out, retrieve everything
    request := make(map[string][]common.Hash)
    // 遍历f.fetched
    for hash, announces := range f.fetched {
         // 这部分逻辑类似fetchTimer.C中讲的，在此不再赘述
        // Pick a random peer to retrieve from, reset all others
        announce := announces[rand.Intn(len(announces))]
        f.forgetHash(hash)
        // If the block still didn't arrive, queue for completion
        if f.getBlock(hash) == nil {
            request[announce.origin] = append(request[announce.origin], hash)
            // 将创建请求的区块hash添加到f.completing中
             f.completing[hash] = announce
        }
    }
    // 遍历请求，讲每个请求发送出去
    // Send out all block body requests
    for peer, hashes := range request {
        log.Trace("Fetching scheduled bodies", "peer", peer, "list", hashes)

        // Create a closure of the fetch and schedule in on a new thread
        if f.completingHook != nil {
            f.completingHook(hashes)
        }
        bodyFetchMeter.Mark(int64(len(hashes)))
         // 此段是发送拉去区块body的代码
        go f.completing[hashes[0]].fetchBodies(hashes)
    }
    // Schedule the next fetch if blocks are still pending
    f.rescheduleComplete(completeTimer)
```

上面主要是completeTimer.C的处理逻辑，主要是遍历f.fetched，创建区块body请求，将请求发送出去，下面来看下当收到区块body时候如何处理

```
case msg.Code == BlockBodiesMsg:
    ..................
    if filter {
    transactions, uncles = pm.fetcher.FilterBodies(p.id, transactions, uncles, time.Now())
}
    if len(transactions) > 0 || len(uncles) > 0 || !filter {
        err := pm.downloader.DeliverBodies(p.id, transactions, uncles)
        if err != nil {
            log.Debug("Failed to deliver bodies", "err", err)
        }
}
// 先看 FilterBodies的实现
func (f *Fetcher) FilterBodies(peer string, transactions [][]*types.Transaction, uncles [][]*types.Header, time time.Time) ([][]*types.Transaction, [][]*types.Header) {
    log.Trace("Filtering bodies", "peer", peer, "txs", len(transactions), "uncles", len(uncles))
    // Send the filter channel to the fetcher
```

```
    filter := make(chan *bodyFilterTask)

    select {
    case f.bodyFilter <- filter:
    case <-f.quit:
        return nil, nil
    }
    // Request the filtering of the body list
    select {
    case filter <- &bodyFilterTask{peer: peer, transactions: transactions, uncles: uncles, time:
time}:
    case <-f.quit:
        return nil, nil
    }
    // Retrieve the bodies remaining after filtering
    select {
    case task := <-filter:
        return task.transactions, task.uncles
    case <-f.quit:
        return nil, nil
    }
}
// 上面路基和headerFilter的逻辑类似，不在此赘述，直接看fetcher主循环如何处理
case filter := <-f.bodyFilter:
    // Block bodies arrived, extract any explicitly requested blocks, return the rest
    var task *bodyFilterTask
    select {
    // 从filter通道读取bodyFilterTask
    case task = <-filter:
    case <-f.quit:
        return
    }
    bodyFilterInMeter.Mark(int64(len(task.transactions)))

    blocks := []*types.Block{}
    // 遍历请求结果返回的交易和叔块，进行数据匹配
    for i := 0; i < len(task.transactions) && i < len(task.uncles); i++ {
        // Match up a body to any possible completion request
        matched := false
        // 遍历f.completing中的数据
        for hash, announce := range f.completing {
            // 如果queued中不包含key为hash的数据
            if f.queued[hash] == nil {
                txnHash := types.DeriveSha(types.Transactions(task.transactions[i]))
                uncleHash := types.CalcUncleHash(task.uncles[i])
                //若task中拿出来的txhash、uncleHash、peer和我们发送请求时候的相等，那就说明匹配成功
                if txnHash == announce.header.TxHash && uncleHash == announce.header.UncleHash &&
announce.origin == task.peer {
                    // Mark the body matched, reassemble if still unknown
                    matched = true
                    // 如果本地不包含此区块，那么构建此block，并将block添加到blocks切片中随后做处理
                    if f.getBlock(hash) == nil {
                        block :=
types.NewBlockWithHeader(announce.header).WithBody(task.transactions[i], task.uncles[i])
                        block.ReceivedAt = task.time

                        blocks = append(blocks, block)
```

```
                    } else {
                            // 如果此区块存在区块链，那么从fetcher中忘记此hash
                        f.forgetHash(hash)
                    }
                }
            }
        }
        // 如果成功匹配，则从task中删除相应位置的数据，跳过此次循环直到for循环条件不满足
        if matched {
            task.transactions = append(task.transactions[:i], task.transactions[i+1:]...)
            task.uncles = append(task.uncles[:i], task.uncles[i+1:]...)
            i--
            continue
        }
    }

    bodyFilterOutMeter.Mark(int64(len(task.transactions)))
    select {
     // 将过滤完的task在返回给FilterBodies方法
    case filter <- task:
    case <-f.quit:
        return
    }
    // 遍历已经完成请求的区块blocks，将其中符合条件的传递给f.enqueue处理
    // Schedule the retrieved blocks for ordered import
    for _, block := range blocks {
        if announce := f.completing[block.Hash()]; announce != nil {
            f.enqueue(announce.origin, block)
        }
    }
}
// 下面开始讲解f.enqueue方法
func (f *Fetcher) enqueue(peer string, block *types.Block) {
    hash := block.Hash()

    // Ensure the peer isn't DOSing us
    count := f.queues[peer] + 1
    if count > blockLimit {
        log.Debug("Discarded propagated block, exceeded allowance", "peer", peer, "number",
block.Number(), "hash", hash, "limit", blockLimit)
        propBroadcastDOSMeter.Mark(1)
        f.forgetHash(hash)
        return
    }
    // Discard any past or too distant blocks
    if dist := int64(block.NumberU64()) - int64(f.chainHeight()); dist < -maxUncleDist || dist >
maxQueueDist {
        log.Debug("Discarded propagated block, too far away", "peer", peer, "number",
block.Number(), "hash", hash, "distance", dist)
        propBroadcastDropMeter.Mark(1)
        f.forgetHash(hash)
        return
    }
    // 判断此hash在queued中不存在后将其相关数据组装成一个inject结构体加入f.queue这个优先级队列里面，以区
块号负数为权值，那么区块号越小的就越排在优先级队列前面
    // Schedule the block for future importing
    if _, ok := f.queued[hash]; !ok {
```

```
        op := &inject{
            origin: peer,
            block:  block,
        }
        f.queues[peer] = count
        f.queued[hash] = op
        f.queue.Push(op, -int64(block.NumberU64()))
        if f.queueChangeHook != nil {
            f.queueChangeHook(op.block.Hash(), true)
        }
        log.Debug("Queued propagated block", "peer", peer, "number", block.Number(), "hash", hash,
"queued", f.queue.Size())
    }
}

// 下面是fetcher主循环开始时候对f.queue中数据的处理
for !f.queue.Empty() {
        // 从优先级队列中取出区块号最小的一个区块
        op := f.queue.PopItem().(*inject)
        hash := op.block.Hash()
        if f.queueChangeHook != nil {
            f.queueChangeHook(hash, false)
        }
        // 判断区块号是否大于当前区块高度+1，那么此时将此区块放入f.queue中随后再处理，并跳出此循环，要不
然继续往下执行
        // If too high up the chain or phase, continue later
        number := op.block.NumberU64()
        if number > height+1 {
            f.queue.Push(op, -int64(number))
            if f.queueChangeHook != nil {
                f.queueChangeHook(hash, true)
            }
            break
        }
        // 如果此区块的区块号距离本地区块高度为-7以后或者已经存在本地链中，那么fetcher忘记此区块
        // Otherwise if fresh and still unknown, try and import
        if number+maxUncleDist < height || f.getBlock(hash) != nil {
            f.forgetBlock(hash)
            continue
        }
        // 若区块经过以上检查后，还能执行到此部逻辑，那么调用insert将区块插入本地链
        f.insert(op.origin, op.block)
}
```

小结：

以上就是整个fetcher所能支撑以太坊的所有逻辑，可以看出来它是一个累积块通知的模块，然后对这些通知进行管理和调度，取得完整的block，最后插入本地链中。