以太坊源码解析：区块同步-fetcher

2019-05-15 fatcat22 ethereum 原创 ethereum downloader fetcher 源码解析

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

引言

fetcher 模块位于 eth 目录下，是区块同步功能的一部分。它的代码很少，主要代码只有 eth/fetcher/fetcher.go 一个源文件。乍看上去，会感觉它和 downloader 模块有些功能冲突：都是对区块进行同步，为什么要分成两块不同的代码呢？

实际上，fetcher 模块和 downloader 模块所承担的任务是不同的。downloader 功能比较重，用来保证自己的区块链和其它节点之间不会有太多差距；而 fetcher 功能较轻，只会对矿工新产生的区块进行同步和广播。本篇文章里，我们就来看看 fetcher 的具体实现。

调用时机

刚才我们已经提到过， fetcher 只会矿工新产生的区块进行同步和广播。下面我们看看这在代码中是如何体现的。

`Fetcher` 对象对外导出的方法中，能导致同步功能的只有两个方法：`Fetcher.Notify` 和 `Fetcher.Enqueue`。而这两个方法分别在 `ProtocolManager.handleMsg` 中处理 `NewBlockHashesMsg` 和 `NewBlockMsg` 两个消息时被调用。这两个消息又是分别由 `peer.AsyncSendNewBlockHash` 和 `peer.AsyncSendNewBlock` 两个方法发出的，这两个方法只有在矿工挖到新的区块时才会被调用：

```
// 订阅本地挖到新的区块的消息
func (pm *ProtocolManager) minedBroadcastLoop() {
    for obj := range pm.minedBlockSub.Chan() {
        if ev, ok := obj.Data.(core.NewMinedBlockEvent); ok {
            pm.BroadcastBlock(ev.Block, true)  // First propagate block to peers
            pm.BroadcastBlock(ev.Block, false) // Only then announce to the rest
        }
    }
}

func (pm *ProtocolManager) BroadcastBlock(block *types.Block, propagate bool) {
    ......

    if propagate {
        ......

        for _, peer := range transfer {
            peer.AsyncSendNewBlock(block, td) //发送区块数据
        }
    }

    if pm.blockchain.HasBlock(hash, block.NumberU64()) {
        for _, peer := range peers {
            peer.AsyncSendNewBlockHash(block) //发送区块哈希
        }
    }
}
```

Content

所以，当某个矿工产生了新的区块、并将这个新区块广播给其它节点，而其它节点收到广播的消息时，才会用到 fetcher 模块去同步这些区块。

字段与状态

知道了 `Fetcher` 对象的功能和目的，我们现在来看看它的内部实现。在 `Fetcher` 内部对新产生的区块进行同步时，将这一同步过程分成了几个阶段，并且每一个阶段都有一个字段与之对应，用来记录这个阶段的数据。以下我们按下载流程的先后顺序介绍一下这些阶段和字段：

- 通知：`Fetcher.announced`
  此阶段代表有节点宣称自己有了新产生的区块（注意这个新产生的区块不一定是自己产生的，也可能是同步了其它节点新产生的区块），`Fetcher` 对象将相关信息放到 `Fetcher.announced` 中，等待下载。
- 下载中：`Fetcher.fetching`
  此阶段代表之前「宣告」的区块正在被下载。
- header 已下载：`Fetcher.fetched`
  此阶段代表区块的 header 已下载成功，现在等待下载 body。
- 等待完成：`Fetcher.completing`
  此阶段代表 body 已经发起了下载，正在等待 body 下载成功。
- 等待入库：`Fetcher.queue`
  此阶段代表 body 已经下载成功。因此一个区块的数据：header 和 body 都已下载完成，此区块正在等待写入本地数据库。

另外在 `Fetcher` 对象中还有一些 channel，用来对消息进行分流。 `Fetcher` 的实现思想是一个中心化的事件处理循环 `Fetcher.loop`，其它方法几乎所有的事情都通过 channel 将数据发给 `Fetcher.loop` 处理。这里不再对这些 channel 进行的详细说明了。

一个区块的同步流程

在了解了 `Fetcher` 对象的功能，以及它的一些字段代表的意义和阶段以后，在这一小节里，我们通过某个区块从发现到写入本地数据库的整个流程，来详细看一下 `Fetcher` 实现。

发现区块

前面我们提到过，fetcher 模块是用来对新产生的区块进行同步的。而新产生区块时，会使用消息 `NewBlockHashesMsg` 和 `NewBlockMsg` 对其进行传播（细节请参看「调用时机」小节）。因此 `Fetcher` 对象也是从这两个消息处发现新的区块信息的。

我们先看一下 `NewBlockHashesMsg` 消息的处理代码中，与 `Fetcher` 有关的代码：

```
func (pm *ProtocolManager) handleMsg(p *peer) error {
    ......

    switch {
    case msg.Code == NewBlockHashesMsg:
        ......
        for _, block := range unknown {
            pm.fetcher.Notify(p.id, block.Hash, block.Number, time.Now(), p.RequestOneHeader,
p.RequestBodies)
        }
    }
}
```

`unknown` 变量代表的是本次消息收到的区块哈希中，在本地数据库查询不到数据的哈希。这里将这些本地未知的哈希全部通过 `Fetcher.Notify` 通知 `Fetcher` 对象，以便后续安排下载。

接下来我们再看一下 `NewBlockMsg` 消息的处理代码中，与 `Fetcher` 有关的代码：

```
func (pm *ProtocolManager) handleMsg(p *peer) error {
    ......
```

```
    switch {
    case msg.Code == NewBlockMsg:
        ......
        pm.fetcher.Enqueue(p.id, request.Block)
        ......
    }
}
```

这里将新收到的区块通过 `Fetcher.Enqueue` 通知 `Fetcher` 对象。

需要说明一点的是，在接下来的说明步骤中，是按照刚才调用 `Fetcher.Notify` 后的处理步骤进行说明的。如果调用 `Fetcher.Enqueue`，则处理步骤应该直接到了「排队入库（Fetcher.queue）」这一小节。

通知（Fetcher.announced）

在 `NewBlockHashesMsg` 消息处理中，`Fetcher` 对象通过 `Fetcher.Notify` 方法得到了新的区块的哈希。现在就让我们看看 `Fetcher.Notify` 是如何处理的：

```
func (f *Fetcher) Notify(peer string, hash common.Hash, number uint64, time time.Time,
        headerFetcher headerRequesterFn, bodyFetcher bodyRequesterFn) error {
    block := &announce{
        hash:        hash,
        number:      number,
        time:        time,
        origin:      peer,
        fetchHeader: headerFetcher,
        fetchBodies: bodyFetcher,
    }
    select {
    case f.notify <- block:
        return nil
    case <-f.quit:
        return errTerminated
    }
}
```

这个方法很简单，它构造了一个 `announce` 结构，并将其发送给了 `Fetcher.notify` 这个 channel。注意 `announce` 这个结构里带着下载 header 和 body 的方法：`fetchHeader` 和 `fetchBodies`。

那么谁在等待 `Fetcher.notify` 这个消息呢？答案是 `Fetcher.loop` 中。我们分两部分看一下这个方法中关于 `Fetcher.notify` 的消息处理。先看第一部分代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case notification := <-f.notify:
            propAnnounceInMeter.Mark(1)

            //判断这个节点已经通知的、但是还未下载成功的哈希的数量。
            count := f.announces[notification.origin] + 1
            if count > hashLimit {
                propAnnounceDOSMeter.Mark(1)
                break
            }
            //确保当前通知的这个区块不会太旧（比本地区块高度小 maxUncleDist）
            //或太新（比本地区块高度大 maxQueueDist）
            if notification.number > 0 {
```

```
                if dist := int64(notification.number) - int64(f.chainHeight()); dist < -
maxUncleDist || dist > maxQueueDist {
                    propAnnounceDropMeter.Mark(1)
                    break
                }
            }
            // 确保当前通知的区块还未开始下载
            if _, ok := f.fetching[notification.hash]; ok {
                break
            }
            if _, ok := f.completing[notification.hash]; ok {
                break
            }

            ......
        }
    }
}
```

这一部分的代码是一些有效性判断。收到通知后，第一个要判断的是当前这个节点通知的区块中，还有多少是未下载的。如果数量太多，就先不去处理最新的通知了。这样即可以避免进程缓存了太多待下载的区块信息而占用过多内存，也可以防止对方节点不断地恶意发送无效区块。

在第二个 if 中，代码保证新通知的区块不会太旧或太新。如果太旧的话，这个区块是没有意义的。因为如果这个区块是主链上的区块，那么本地肯定已经有这个区块了；而如果这个区块不在主链上，即它是可能是某个区块的 uncle 区块，那么高度差超过 7（即 `maxUncleDist` 的值） 也是没意义的（关于 uncle 区块的高度差为什么超过 7 没有意义，可以参看这篇文章 中的「叔块」小节）。如果太新的话，我的理解是高度差太大，暂时不用去关心。

后面的两个 if 判断很简单，是为了保证同一个区块没会被重复下载。

接下来我们再看看 `Fetcher.notify` 消息处理的第二部分代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case notification := <-f.notify:
            ......

            f.announces[notification.origin] = count
            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)
            }
            if len(f.announced) == 1 {
                f.rescheduleFetch(fetchTimer)
            }
        }
    }
}
```

这里首先更新的了节点的待下载数量（刚才说过这个值用来保证不会缓存太多某个节点的未下载区块），然后将新通知的区块信息加入到 `Fetcher.announced` 中。最后如果 `Fetcher.announced` 中只有刚才新加入的这一个区块信息，那么调用 `Fetcher.rescheduleFetch` 重新设置变量 `fetchTimer` 的周期（至于重置 `fetchTimer` 的意义，我们后面会提到）。

下载中（Fetcher.fetching）

通知以后的区块信息，其状态从「通知」变成了「下载中」是在 `fetchTimer` 这个消息的处理代码中完成的。我们先来看看这段代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case <-fetchTimer.C:
            request := make(map[string][]common.Hash)

            // 选择要下载的区块，从 announced 转移到 fetching 中，
            // 并将要下载的哈希填充到 request 中
            for hash, announces := range f.announced {
                if time.Since(announces[0].time) > arriveTimeout-gatherSlack {
                    // 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 fetching
                    if f.getBlock(hash) == nil {
                        request[announce.origin] = append(request[announce.origin], hash)
                        f.fetching[hash] = announce
                    }
                }
            }

            // 发送下载 header 的请求
            for peer, hashes := range request {
                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
                    }
                }()
            }

            //重新设置下次的下载发起时间
            f.rescheduleFetch(fetchTimer)
        }
    }
}
```

这段代码虽然长，但不算复杂，我将其一下子全部放了上来。`fetchTimer` 的处理代码中主要有两个 for 循环，第一个用来从 `Fetcher.announced` 中获取「可以下载的」区块信息（注意「可以下载的」条件是区块通知的时间已经过去了 `arriveTimeout-gatherSlack` 这么长时间），并将相关信息转移到 `Fetcher.fetching` 中。也就是在这里，区块的状态从「通知」变成了「下载中」（即区块信息从 `Fetcher.announced` 字段转移到了 `Fetcher.fetching` 字段中）。

第二个 for 循环就是发起下载请求。

最后又调用了 `Fetcher.rescheduleFetch`，再次重新设置 `fetchTimer` 的周期。

这里我们顺便解释一下 `Fetcher.rescheduleFetch` 这个方法，它是为了重新设置 `fetchTimer` 这个变量的周期。我们先看看它是怎么实现的：

```go
func (f *Fetcher) rescheduleFetch(fetch *time.Timer) {
    // Short circuit if no blocks are announced
    if len(f.announced) == 0 {
        return
    }
    // Otherwise find the earliest expiring announcement
    earliest := time.Now()
    for _, announces := range f.announced {
        if earliest.After(announces[0].time) {
            earliest = announces[0].time
        }
    }
    fetch.Reset(arriveTimeout - time.Since(earliest))
}
```

其实还是比较简单的，首先从 `Fetcher.announced` 中找出通知的区块中，通知时间距当前最近的时间（也即最晚通知的时间），然后利用这个时间重置 `fetch` 这个参数。这里的意思是：在最近一个次通知过去 `arriveTimeout` 这么长时间以后，再触发 `fetch` 这个 timer。而查遍代码，这个 timer 只能是 `fetchTimer` 这个变量。但代码写成这种逻辑的原因是什么呢？

刚才我们已经看到了， `fetchTimer` 的功能就是定期发起请求获取区块的 header。而 `Fetcher.rescheduleFetch` 中设置的时间，就是要在区块通知过去 `arriveTimeout - time.Since(earliest)` 这么长时间以后，再去发起下载请求。我猜这样是因为刚产生的区块并不稳定，有可能过了一会它变成了一个废块，也有可能变成了别人的叔块，稍等片刻再去处理时可能这些变化已经完成，从而避免自己对这些变化进行处理。（不过我对这个猜测并没有百分百的信心）

header 下载完成（Fetcher.fetched）

前面获取 header 的请求已发出，接下来就等待 header 的到来啦。有 header 到来时是通过 `Fetcher.FilterHeaders` 通知的。需要提一下的是，虽然 `Fetcher.FilterHeaders` 方法从名字看其功能是「过滤」，但如果参数传入的数据中存在我们正在下载的 header，它也会将其保留下来，从而完成下载 header 过程。下面我们会详细分析这一点。

还记得所有通信消息都是在 `ProtocolManager.handleMsg` 中处理的吗？当请求的 header 到来时，会通过 `Fetcher.FilterHeaders` 通知 `Fetcher` 对象：

```go
func (pm *ProtocolManager) handleMsg(p *peer) error {
    ......
    switch {
    case msg.Code == BlockHeadersMsg:
        ......
        filter := len(headers) == 1
        if filter {
            ......
            headers = pm.fetcher.FilterHeaders(p.id, headers, time.Now())
        }
        ......
    }
}
```

而 `Fetcher.FilterHeaders` 又是如何实现的呢：

```go
func (f *Fetcher) FilterHeaders(peer string, headers []*types.Header, time time.Time)
[]*types.Header {
    filter := make(chan *headerFilterTask)

    // 先发一个通信用的 channel 给 headerFilter
    select {
    case f.headerFilter <- filter:
```

```
    }
    // 将要过滤的 header 发送给 filter
    select {
    case filter <- &headerFilterTask{peer: peer, headers: headers, time: time}:
    }
    // 再从 filter 中获取过滤结果
    select {
    case task := <-filter:
        return task.headers
}
}
```

这里 `Fetcher.FilterHeaders` 仍然是将信息发给 `Fetcher.loop` 处理，因此我们直接看 `Fetcher.loop` 中对 `Fetcher.headerFilter` 的处理：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case filter := <-f.headerFilter:
            var task *headerFilterTask
            select {
            case task = <-filter:
            case <-f.quit:
                return
            }

            ......
        }
    }
}
```

结合 `Fetcher.FilterHeaders` 的实现，这里从 `filter` 变量中取得要过滤的信息 `task`。我们继续往下看：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case filter := <-f.headerFilter:
            ......
            unknown, incomplete, complete := []*types.Header{}, []*announce{}, []*types.Block{}
            for _, header := range task.headers {
                hash := header.Hash()

                // 判断是否是我们正在下载的 header
                if announce := f.fetching[hash]; announce != nil && announce.origin == task.peer
&& f.fetched[hash] == nil && f.completing[hash] == nil && f.queued[hash] == nil {
                    if header.Number.Uint64() != announce.number {
                        f.dropPeer(announce.origin)
                        f.forgetHash(hash)
                        continue
                    }
                    // 判断此区块在本地是否已存在
                    if f.getBlock(hash) == nil {
                        announce.header = header
                        announce.time = task.time

                        // 判断是否是空区块。
                        // 对于空区块，直接加入到 Fetcher.completing 中
                        if header.TxHash == types.DeriveSha(types.Transactions{}) &&
```

```
header.UncleHash == types.CalcUncleHash([]*types.Header{}) {
                        block := types.NewBlockWithHeader(header)
                        block.ReceivedAt = task.time

                        complete = append(complete, block)
                        f.completing[hash] = announce
                        continue
                    }

                    // 非空区块，保存在 incomplete 中
                    incomplete = append(incomplete, announce)
                } else {
                    // 本地忆存在这个区块
                    f.forgetHash(hash)
                }
            } else {
                // 不是我们想要下载的区块
                unknown = append(unknown, header)
            }
        }
        ......
    }
}
}
```

这块代码就是一个大的 for 循环，对所有需要过滤的 header 进行过滤。从开始的三个变量定义可以看出来，这段代码将需要过滤的 header 分成三类：unknown，incomplete，complete。unknown 代表「未知」，这些区块根本不是 `Fetcher` 对象发起下载的；incomplete 代表是 `Fetcher` 发起下载的区块，但这里只有 header 数据，还需少 body 数据；complete 也代表是 `Fetcher` 发起的区块，并且这个区块已经不缺数据可以直接导入本地数据库了。complete 状态的数据都是空块，因为空区块的 body 为空，不需要下载。

分类 header 的方法也比较简单，在 for 循环中，首先判断 header 是否是 `Fetcher` 对象发起下载的：如果不是，就将其加入到 unknown 中。

如果 header 是我们发起下载的，则还会判断本地是否已经存在这个区块了，因为在 `Fetcher` 发起下载的过程中，downloader 模块可能已经将其下载完成了。如果本地没有这个区块，则根据区块是否为空，将其记录到 incomplete 或 complete 中。

在这个过程中需要提一下的空区块的状态转变，这里直接将空区块的信息放到了 `Fetcher.completing` 中，也就是可以可以等待入库了。

我们继续看这个 for 循环后面的代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case filter := <-f.headerFilter:
            ......

            // 将 unknown 通知 Fetcher.FilterHeaders
            select {
            case filter <- &headerFilterTask{headers: unknown, time: task.time}:
            case <-f.quit:
                return
            }

            // 将 incomplete 加入到 Fetcher.fetched 中
            for _, announce := range incomplete {
                hash := announce.header.Hash()
                if _, ok := f.completing[hash]; ok {
```

```
                continue
            }
            f.fetched[hash] = append(f.fetched[hash], announce)
            if len(f.fetched) == 1 {
                f.rescheduleComplete(completeTimer)
            }
        }
        // 将 complete 加入等待入库的队列中
        for _, block := range complete {
            if announce := f.completing[block.Hash()]; announce != nil {
                f.enqueue(announce.origin, block)
            }
        }
    }
  }
}
```

这段代码逻辑比较简单了，就是处理前面分出来的三类数据：unknown，incomplete，complete。对于 unknown 数据，我们退回给 `Fetcher.FilterHeaders`；对于 incomplete 数据，我们将其加放到 `Fetcher.fetched` 中；而对于 `complete` 数据，我们调用 `Fetcher.enqueue` 将其加入到待入库队列 `Fetcher.queue` 中。也就是在这里，数据的状态随着在字段之间的转移也发生了变化。

注意在处理 incomplete 时，会调用 `Fetcher.rescheduleComplete` 重置 `completeTimer`。这一方法与 `Fetcher.rescheduleFetch` 类似，我们不再详细说明。

body 下载中（Fetcher.completing）

现在 header 已经下载完成了，只要再把 body 下载下来，那么整个区块数据就有了。发起 body 下载请求的代码在 `completeTimer` 消息的处理过程中，我们看一下这段代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case <-completeTimer.C:
            request := make(map[string][]common.Hash)

            // 从 Fetcher.fetched 中选取将要下载 body 的是信息放入 request 中
            for hash, announces := range f.fetched {
                // 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)
                    f.completing[hash] = announce
                }
            }

            // 发送请求
            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)))
```

```
                go f.completing[hashes[0]].fetchBodies(hashes)
            }
            // Schedule the next fetch if blocks are still pending
            f.rescheduleComplete(completeTimer)
        }
    }
}
```

这段代码与 `fetchTimer` 消息的处理代码类似。代码首先检查 `Fetcher.fetched` 中的信息，如果其哈希在本地不存在，则将信息转移到 `Fetcher.completing` 中，并填充 `request` 变量。注意这一过程中，下载状态从「header 下载完成」变成了「body 下载中」。

接下来的 for 循环则是对 `request` 中的每一条信息，发起获取 body 的请求。

body 下载完成（Fetcher.queue）

发起 body 请求后，接下来就是等待 body 数据的到达啦。与 header 类似，当有 body 数据到达时是通过 `Fetcher.FilterBodies` 通知的，并且虽然名字也是「过滤」，在处理时也会将自己正在下载的 body 数据保留下来。

`Fetcher.FilterBodies` 的实现与 `Fetcher.FilterHeaders` 类似，因此我们这里直接看 `Fetcher.loop` 中对 `Fetcher.bodyFilter` 的处理：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case filter := <-f.bodyFilter:
            var task *bodyFilterTask
            select {
            case task = <-filter:
            }

            blocks := []*types.Block{}
            // 对于每一组数据，查看其 tx 哈希 和 uncle 哈希是否在 Fetcher.queued 中的 header 中
            for i := 0; i < len(task.transactions) && i < len(task.uncles); i++ {
                // Match up a body to any possible completion request
                matched := false

                for hash, announce := range f.completing {
                    if f.queued[hash] == nil {
                        txnHash := types.DeriveSha(types.Transactions(task.transactions[i]))
                        uncleHash := types.CalcUncleHash(task.uncles[i])

                        if txnHash == announce.header.TxHash && uncleHash ==
announce.header.UncleHash && announce.origin == task.peer {
                            // 这是我们发起下载的数据
                            matched = true

                            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 {
                                f.forgetHash(hash)
                            }
                        }
```

```
                }
            }
            if matched {
                task.transactions = append(task.transactions[:i], task.transactions[i+1:]...)
                task.uncles = append(task.uncles[:i], task.uncles[i+1:]...)
                i--
                continue
            }
        }
        ......
    }
  }
}
```

这段代码虽然长，但并不复杂。上面这段代码中，首先从 `filter` 中获取要过滤的数据 `task`。然后对于 `task` 中的每一组数据，计算它们的 transaction 哈希和 uncle 哈希，然后去 `Fetcher.completing` 中记录的 header 中查找，看是否能匹配到相应的 header。如果匹配到，说明当前的数据正是我们发起下载的数据，代码中将这些匹配和 header 一起，组成一个 `Block` 结构放到 `blocks` 变量中。如果不匹配，则数据会留在 `task` 中。

我们继续看后续的处理代码：

```
func (f *Fetcher) loop() {
    for {
        ......
        select {
        case filter := <-f.bodyFilter:
            ......

            select {
            case filter <- task:
            }
            // Schedule the retrieved blocks for ordered import
            for _, block := range blocks {
                if announce := f.completing[block.Hash()]; announce != nil {
                    f.enqueue(announce.origin, block)
                }
            }
        }
    }
}
```

这里首先将留在 `task` 中的数据返回给 `Fetcher.FilterBodies`，然后调用 `Fetcher.enqueue` 将 `blocks` 中的所有区块加入到待入库的队列 `Fetcher.queue` 中。

我们顺便看一下 `Fetcher.enqueue` 的代码：

```
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 {
        f.forgetHash(hash)
        return
    }
    // Discard any past or too distant blocks
    if dist := int64(block.NumberU64()) - int64(f.chainHeight()); dist < -maxUncleDist || dist >
maxQueueDist {
        f.forgetHash(hash)
        return
```

```
    }
    // 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)
        }
    }
}
```

代码也是比较简单的，首先确保不会有太多来自同一节点的区块等待入库。然后确保这个区块不会太旧也不会太新（这一点在「通知（Fetcher.announced）」这一小节中介绍过）。最后将区块数据放到了 `Fetcher.queue` 中。注意 `Fetcher.queue` 是一个优先级队列，它的优先级为高度的负值，所以在 Pop 操作中，高度越小的元素优先级越高，越会被首先弹出。

正式入库

前面经过一系列的步骤，一个完整的区块信息已经存放到 `Fetcher.queue` 中了。那在它们是在何处被正式的写入到本地数据库中的呢？

将区块写入数据库的代码位于 `Fetcher.loop` 中的 for 循环的开始部分的代码：

```
func (f *Fetcher) loop() {
    for {
        // 去掉下载超时的区块
        for hash, announce := range f.fetching {
            if time.Since(announce.time) > fetchTimeout {
                f.forgetHash(hash)
            }
        }

        // Import any queued blocks that could potentially fit
        height := f.chainHeight()
        for !f.queue.Empty() {
            op := f.queue.PopItem().(*inject)
            hash := op.block.Hash()
            if f.queueChangeHook != nil {
                f.queueChangeHook(hash, false)
            }

            //如果高度大于本地最高高度 + 1，则将信息放回队列并结束入库
            number := op.block.NumberU64()
            if number > height+1 {
                f.queue.Push(op, -int64(number))
                if f.queueChangeHook != nil {
                    f.queueChangeHook(hash, true)
                }
                break
            }

            // 确保区块不会太旧并且在本地未存储
            if number+maxUncleDist < height || f.getBlock(hash) != nil {
                f.forgetBlock(hash)
```

```
                continue
            }
            // 存入本地数据库
            f.insert(op.origin, op.block)
        }

        ......
    }
}
```

这段代码就在 `Fetcher.loop` 中的 for 循环的开头，因此每执行一次 for 循环，这段代码就会执行一次。代码首先去掉超时的区块，然后对于每一个区块，检查它的高度是否超过了本地区块最高高度的下一个高度（即 `number > height+1`），如果超过了就将信息放回队列。因为 `Fetcher.queue` 是一个优先级队列，每次都是弹出高度最小的区块，如果某次高度判断失败，那么后面弹出来区块肯定也会判断失败，因此这里直接 break 掉了这个循环。

如果高度上限符合要求，还要判断一下下限，即高度差不能低于 `maxUncleDist`（这一点在「通知（Fetcher.announced）」这一小节中介绍过），并且本地仍然没有保存这个区块，那么就调用 `Fetcher.insert` 将这个区块写入到本地数据库中。

我们顺便简单看一下 `Fetcher.insert` 的实现：

```
func (f *Fetcher) insert(peer string, block *types.Block) {
    hash := block.Hash()

    // Run the import on a new thread
    go func() {
        defer func() { f.done <- hash }()

        // If the parent's unknown, abort insertion
        parent := f.getBlock(block.ParentHash())
        if parent == nil {
            return
        }
        // Quickly validate the header and propagate the block if it passes
        switch err := f.verifyHeader(block.Header()); err {
        case nil:
            // All ok, quickly propagate to our peers
            propBroadcastOutTimer.UpdateSince(block.ReceivedAt)
            go f.broadcastBlock(block, true)

        case consensus.ErrFutureBlock:
            // Weird future block, don't fail, but neither propagate
        default:
            f.dropPeer(peer)
            return
        }

        // Run the actual import and log any issues
        if _, err := f.insertChain(types.Blocks{block}); err != nil {
            return
        }
        // If import succeeded, broadcast the block
        go f.broadcastBlock(block, false)

        // Invoke the testing hook if needed
        if f.importedHook != nil {
            f.importedHook(block)
        }
    }()
}
```

可以看到，这个方法里使用一个 goroutine 将区块插入数据库。需要说明的是，在校验区块成功和写入数据库成功以后，都会立刻调用 `Fetcher.broadcastBlock` 广播区块。由于 `Fetcher` 对象只用来同步矿工新产生抠块，此处的广播进一步加快了新产生区块的传播速度。

总结

这篇文章里我们分析了 fetcher 模块的作用和实现。fetcher 模块主要是用来及时的同步矿工新产生的区块的，这一功能与 downloader 相互补充，共同组成了以太坊区块同步的逻辑。在 `Fetcher` 对象同步区块时，很清晰的将同步过程分成了几个不同的状态，并使用不同的字段代表这些状态。被同步的数据就在这些字段中流转，最终写入本地数据库。

以上就是对 fetcher 模块的分析。水平有限，如果有错误还请留言或邮件指出，非常感谢。

Similar Posts

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

上一篇 以太坊源码解析：downloader/queue

下一篇 以太坊源码解析：state

Comments

Comments

Login

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

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name

Displayed next to your comments.

Email

Not displayed publicly.

Website (optional)

If you have a website, link to it here.

Subscribe to None

Submit Comment
