以太坊源码解析：区块同步-downloader

2019-05-09 fatcat22 ethereum 原创 ethereum downloader 区块同步 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

在之前的文章里，我们分析了以太坊区块同步的架框代码，了解了以太坊的区块同步整体上是如何运作的。那么这篇文章里，我们就详细看一下这个框架中一个比较重要的部分：downloader 模块。这个模块的主要功能就是从别的节点同步区块到自己节点，虽然功能简单但实现得却比较复杂。

需要说明的是，light 模式下的区块同步使用的是 les 目录下的代码，因此 eth/downloader 目录下的代码虽然有少部分针对 light 模式的处理，但完全中可以忽略。所以在本文下面的分析中，会忽略那些针对 light 模式特殊处理的代码。

源码目录

downloader 模块的代码位于 eth/downloader 目录下。其主要的功能代码分别在 downloader.go、peer.go、queue.go、statesync.go 中。

downloader.go 文件中实现了区块同步的主要功能和逻辑，而 queue.go 实现了 `queue` 对象（关于 `queue` 对象的介绍请参看这篇文章），你可以理解为这是一个对区块的组装队列。而 peer.go 实际上是对 eth/peer.go 中的对象的封装，增加了节点是否空闲(idle) 的统计，statesync.go 很明显是用来同步 state 对象的。

同步模式

在介绍 downloader 之前，我们需要先介绍一下同步模式。在以太坊中，区块同步有三种同步模式：full、fast、light。你可以在配置文件中的 [Eth] 节的 SyncMode 字段对其进行配置，或者在命令行使用 「--syncmode」 指定。下面我们分别解释一下这三个种模式的区别。

full mode

「full mode」 在数据库中保存所有区块数据，但在同步时，只从远程节点同步 header 和 body 数据，state 和 receipt 数据则是在本地计算出来的。在 full 模式下，downloader 会同步区块的 header 和 body 数据组成一个区块，然后通过 blockchain 模块的 `BlockChain.InsertChain` 向数据库中插入区块。在 `BlockChain.InsertChain` 中，会逐个计算和验证每个块的 state 和 recepit 等数据，如果一切正常就将区块数据以及自己计算得到的 state、recepit 数据一起写入到数据库中。

fast mode

所谓的「快速模式」，是相对于「full mode」来说的。在 full 模式下，state 和 recepit 是根据区块数据中的交易在当前机器上计算 出来的。而在 fast 模式下，recepit 不再由本地计算，而是和区块数据一样，直接由 downloader 从其它节点中同步；state 数据并不会全部计算和下载，而是选一个较新的区块（称之为 pivot，后面会有详细解释）的 state 进行下载，以这个区块为分界，之前的区块是没有 state 数据的，之后的区块会像 full 模式下一样在本地计算 state。因此在 fast 模式下，同步的数据除了 header 和 body，还有 receipt，以及 pivot 区块的 state。

可以看出，因为 fast 模式忽略了大部分 state 数据，并且使用网络直接同步 receipt 数据的方式替换了 full 模式下的本地计算，所以才比较快。

Content

ligth mode

light 模式也叫做轻模式，它只对区块头进行同步，而不同步其它的数据。当使用 light 模式时，同步的代码在 les 目录，而非 eth/downloader 目录。在本篇文章的分析中我们不涉及 light 模式的下载。

deliver

downloader 模块中关于「deliver」的代码虽然比较简单，但为了完整性和后面一些问题的解释的方便，我们不得不首先提一下「deliver」相关的问题。

downloader 虽然名字叫做「下载」，但它实际上不做任何网络流量 IO。我们在这篇文章分析过，所有的数据发送和接收都是通过 eth 目录下的 `peer` 对象和 `ProtocolManager.handleMsg` 实现的。那么 downloader 模块是怎么拿到下载到的数据的呢？

当 `ProtocolManager.handleMsg` 接收到数据时，会调用 `Downloader` 对象中几个「deliver」方法，将接收到的数据传递到 downloader 模块。这几个「deliver」方法分别是：

> DeliverHeaders
> DeliverBodies
> DeliverReceipts
> DeliverNodeData

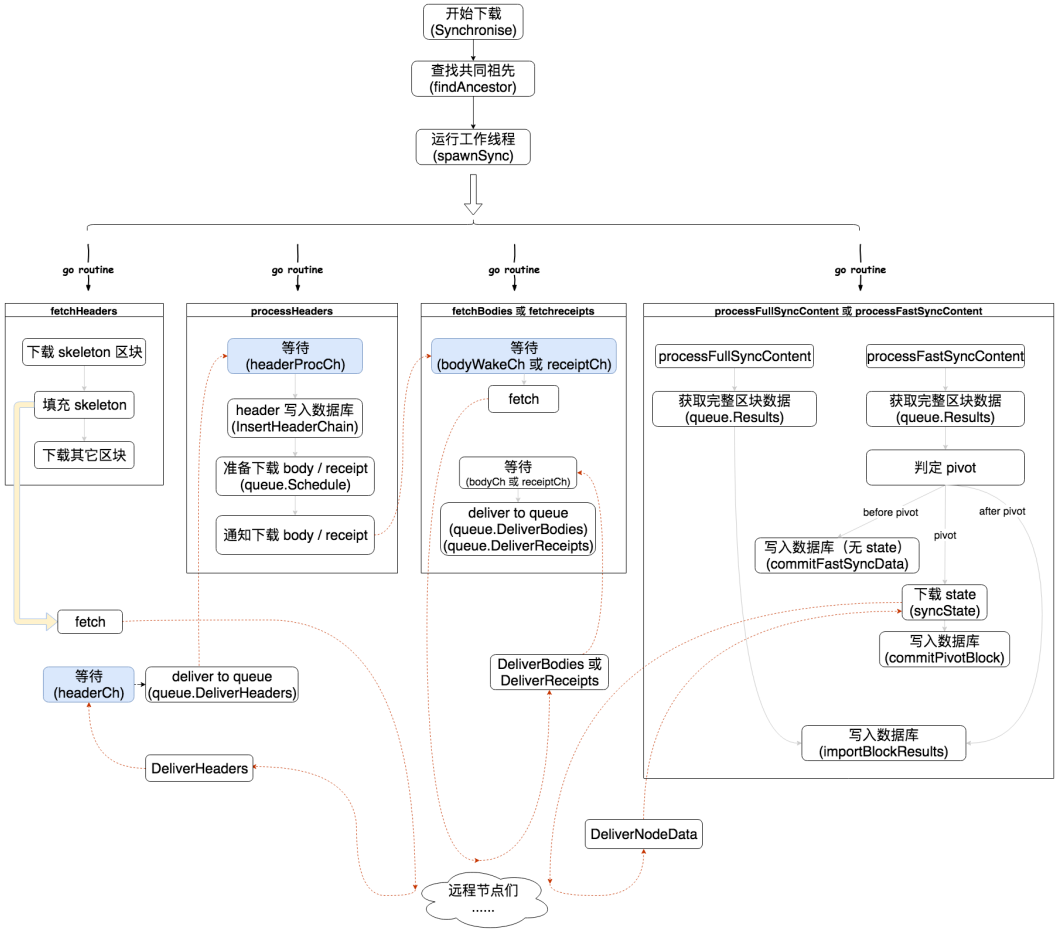
稍微看一下它们的代码就会发现，它们都调用了 `Downloader.deliver` 方法。这个方法通过参数传入的 channel，通知相应的等待数据的线程。因此这几个 channel 就是 downloader 获取数据的通道：

- headerCh : 传送header数据
- bodyCh : 传送body数据
- receiptCh: 传送receipt数据
- stateCh : 传送state数据

后面分析的代码中，有多处向远程节点发送请求数据、然后等待这些 channel 返回数据的代码。读者要牢记这些 channel 的作用。

区块下载流程

我们用一张图来概括一下 downloader 的工作流程：



可以看到，downloader 总是从 `Downloader.Synchronise` 开始执行区块同步任务，它首先使用 `Downloader.findAncestor` 找到与指定节点的共同的祖先块，并从那个块开始同步。在同步时，会同时启动多个 go routine 来同步不同的数据，比如 header、body、receipt 等，但除 header 之外，其它数据总要等到获取到相应的 header 后才开始同步，比如要同步高度为1万的区块，那么需要先同步高度为1万的 header，成功后再通知同步 body 和 receipts 的线程，以便让他们开始同步各自的数据。我们从下面的代码中可以看出这种顺序，注意 `Downloader.bodyWakeCh` 和 `Downloader.receiptWakeCh` 这两个channel：

```
func (d *Downloader) processHeaders(origin uint64, pivot uint64, td *big.Int) error {
    ......
    for {
        select {
        case headers := <-d.headerProcCh:
            if len(headers) == 0 {
                //header全部下载完成，向bodyWakeCh和receiptWakeCh通知这一消息
                for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                    select {
                    case ch <- false:
                    case <-d.cancelCh:
                    }
                }
                ......
            }
            ......

            //有新的header下载完成了，但还没有全部完成。向bodyWakeCh和receiptWakeCh通知这一消息
            for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                select {
                case ch <- true:
                default:
                }
            }
        }
```

```
    }
}

//获取body的线程代码
func (d *Downloader) fetchBodies(from uint64) error {
    ......
    //注意这里的wakeCh为bodyWakeCh
    err := d.fetchParts(..., d.bodyCh, d.bodyWakeCh, ......)
    ......
}

//获取receipt的线程代码
func (d *Downloader) fetchReceipts(from uint64) error {
    ......
    //注意这里的wakeCh为receiptWakeCh
    err := d.fetchParts(..., d.receiptCh, d.receiptWakeCh, ......)
    ......
}

func (d *Downloader) fetchParts(..., deliveryCh chan dataPack, wakeCh chan bool, ......) {
    ......
    update := make(chan struct{}, 1)
    for {
        select {
        case <-d.cancelCh: ......
        case packet := <-deliveryCh: ......
        //等待wakeCh
        case cont := <-wakeCh: ......
        case <-ticker.C: ......
        case <-update:
            ......
            //如果没有需要下载的数据，就继续等待或或如果所有数据处理完成的话就返回
            if pending() == 0 {
                if !inFlight() && finished {
                    return nil
                }
                break
            }
            ......
        }
    }
}
```

在 `Downloader.fetchParts` 中，框架代码就是等待，如果没有消息传来，它是不会做什么事情的（有一个周期性的 ticker 事件，但如果没有发起数据请求，即 `pending()` 返回值为 0，仍然也什么不会做）。`deliveryCh` 的值为 `BodyCh` 或 `receiptCh`，在「deliver」小节里我们说过，这两个 channel 是在数据下载成功时传递数据的，而数据的下载请求是在收到 `update` 时发起的，所以目前也不会收到这个消息。唯一可能从外部接收到的有效的消息就是 `wakeCh`，它的值为 `bodyWakeCh` 或 `receiptWakeCh`，在上面的代码中也很容易看出，只有成功处理了下载的 header 数据，才会激活这两个 channel。因此，我们可以说虽然同步 header 和 body 等其它数据的线程同时开启，但别的线程都会首先进行等待，如果 header 没有下载成功（ wakeCh 没有消息），其它线程是不会有任何动作的。

在同步 header 时，会先从指定的节点中同步一个骨架（skeleton），然后再调用 `Downloader.fillHeaderSkeleton` 填充这个骨架。所谓的「skeleton」，是指每隔一定的区间同步一个区块，我们会在下面的详细进行说明。

在 fast 模式下，downloader 还有一个「pivot」的概念。我们知道 fast 模式不会在本地计算 state 和 receipt 数据。但在同步时，还是会有某个高度以后的区块的 state 会在本地计算，而非从网络同步，这个高度就是 pivot。高度低于 pivot 的区块没有 state 数据；pivot 区块从网络中同步 state；高度大于 pivot 的区块则在本地计算 state 数据。关于 pivot 的具体内容我们在下面还会进行详细的说明。

在 downloader 模块中，还有一个 `queue` 对象。这个对象的功能就是对将要同步的数据进行管理，以及将同步到的数据进行组装。比如在同步之初，`Downloader` 对象会将需要同步的「骨架」写入 `queue` 对象中，并在header同步成功时告诉 `queue` ；而下载 body 和 receipt 的线程也可能询问 `queue` 有哪些数据是它们可以下载的。最后，`Downloader` 还可以询问 `queue` 对象有哪些区块的所有数据已经下载完成了。关于 `queue` 的详细内容，请参看这篇文章。

下面我们就逐一地详细分析一下这些比较重要的内容。

findAncestor

区块同步的第一件重要的事情，就是确定需要同步哪些区块。这里使用一个高度区间来表示这些需要同步的区块，这个区间的顶是远程节点的区块的最高值，这个从 eth 下的 `peer.Head` 方法中可以获取；而区间的底就是两个节点都拥有的相同区块的最高高度。一般情况下，本地节点中的主链上高度最高的区块可以满足这个条件，但并不能保证一定是这样。所以 `Downloader.findAncestor` 就是用来找到这个高度最高的、相同的「ancestor」。

查找共同祖先的方式有两种，一种是通过获取一组某一高度区间内、固定间隔高度的区块，看是否能从这些区块中找到一个本地也拥有的区块；另一种是从某一高度区间内，通过二分法查找共同区块。这两种方法都实现在 `Downloader.findAncestor` 中。我们分别来看一下这两种方法。

固定间隔法

`Downloader.findAncestor` 中首先使用固定间隔的方法查找共同祖先。我们先看看最开始的代码：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    var (
        floor        = int64(-1)
        localHeight  uint64
        remoteHeight = remoteHeader.Number.Uint64()
    )
    switch d.mode {
    case FullSync:
        localHeight = d.blockchain.CurrentBlock().NumberU64()
    case FastSync:
        localHeight = d.blockchain.CurrentFastBlock().NumberU64()
    default:
        localHeight = d.lightchain.CurrentHeader().Number.Uint64()
    }

    if localHeight >= MaxForkAncestry {
        // We're above the max reorg threshold, find the earliest fork point
        floor = int64(localHeight - MaxForkAncestry)
        ......
    }
    from, count, skip, max := calculateRequestSpan(remoteHeight, localHeight)
    go p.peer.RequestHeadersByNumber(uint64(from), count, skip, false)
    ......
}
```

方法最开始首先确定远程节点的高度（`remoteHeight`）和本地高度（`localHeight`），然后调用 `calculateRequestSpan` 计算要获取的高度区间和间隔。这里返回的 `from` 变量代表从哪个高度开始获取区块，`max` 代表高度的上限，因此这两个变量确定了用来查找共同祖先的高度区间。`count` 变量代表一共从远程节点获取多少个区块，`skip` 则代表了间隔，比如 `skip` 值为3，`from` 值为 10，则获取的第一个区块高度为 10，第二个高度为 14。计算出 `from`、`count`、`skip`、`max` 这四个变量以后，就可以调用 `peer.RequestHeadersByNumber` 发送获取header的请求了。

这里有一个 `floor` 变量需要注意下，因为后面的分析会用到。这个变量代表了共同祖先的高度的最小值。它的值为 `localHeight - MaxForkAncestry`（或 -1），也就是说共同祖先的高度最小比本地当前高度小

`MaxForkAncestry` 这么多 ,如果再小，说明本地和远程的分叉分得太夸张了，肯定有问题，不允许再进行同步， `Downloader.findAncestor` 返回错误。

`calculateRequestSpan` 中实现的计算方法比较锁碎，我觉得读者可以不用过于关注。并且我感觉这个函数写得有问题，并不能实现它的目标。根据 `Downloader.findAncestor` 方法的注释中的说明，这种用固定间隔获取共同祖先的方式，应对的是可预见的多数情况：即我们当前节点在正确的链上，并且与其它节点的高度相差不大。我认为这种多数情况就应该是当前节点的最新区块（即上面代码中 `localHeight` 变量所对应的区块）就是共同祖先，所以将要获取的区块区间至少应该包含 `localHeight` 所代表高度。并且，根据 downloader_test.go 中 `TestRemoteHeaderRequestSpan` 函数的实现，我认为作者本身也是想实现这一目标：如果两个节点链的高度差别不大，这个区间就可以包含 `localHeight` 所代表的高度在内。区间为 [1200, 1250] 的这一组测试数据也验证了这个问题。但当我自己填加 [20, 58] 这一组数据到测试代码中时，得到的 `from` 是从 21 开始的，并没有包含高度 20。但这组数据的区块高度差距为 38，比 [1200, 1250] 这一区间的高度差距为 50 还要小。

我们继续看接下来的代码：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......
    for finished := false; !finished; {
        select {
        case <-d.cancelCh:
            return 0, errCancelHeaderFetch
        case packet := <-d.headerCh:
            ......
        case <-timeout:
            return 0, errTimeout
        case <-d.bodyCh:
        case <-d.receiptCh:
        // Out of bounds delivery, ignore
        }
    }
    ......
}
```

这段代码的逻辑很简单，主要功能在 `packet := <-d.headerCh` 这个 case 里，它用来等待返回请求的header数据并进行处理。前面「deliver」小节我们提到过， `headerCh` 就是用来传递接收到的header的。

接下来我们详细看一下读取 `headerCh` 这个channel的case：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......
    for finished := false; !finished; {
        select {
        case packet := <-d.headerCh:
            //验证返回数据的节点是否是我们请求数据的节点
            //验证返回的headers的数量
            //验证返回的headers的高度是我们想要的高度
            ......

            finished = true
            //注意这里是从headers最后一个元素开始查找，也就是高度最高的区块。
            for i := len(headers) - 1; i >= 0; i-- {
                //跳过不在我们请求的高度区间内的区块
                ......

                //检查我们本地是否已经有某个区块了，如果有就算是找到了共同祖先，
                //并将共同祖先的哈希和高度设置在number和hash变量中。
                h := headers[i].Hash()
                n := headers[i].Number.Uint64()

                var known bool
                switch d.mode {
```

```
                case FullSync:
                    known = d.blockchain.HasBlock(h, n)
                case FastSync:
                    known = d.blockchain.HasFastBlock(h, n)
                default:
                    known = d.lightchain.HasHeader(h, n)
                }
                if known {
                    number, hash = n, h
                    break
                }
            }
        }
    }
}
```

这段代码虽然稍长，但逻辑依然很简单。在获取得 headers 数据以后，首先检查是否符合我们的要求。如果符合，则从高度最高的区块开始，检查我们本地是否已经拥有了这个区块。如果有则认为这就是我们和远程节点之间共同的祖先，将这个共同祖先的高度和哈希保存在 `number` 和 `hash` 变量中，这段代码就结束了。如果一直没找到共同祖先，也会跳出最外层的for循环，因为只要headers通过验证符合我们的要求，`finished` 变量就会为 true。

现在已经获取得了远程返回的 header 数据，并且经过了验证和处理。我们继续看后面的代码是怎么做的：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......

    if hash != (common.Hash{}) {
        if int64(number) <= floor {
            return 0, errInvalidAncestor
        }
        return number, nil
    }
}
```

如果 `hash` 变量不是默认值，就认为通过固定间隔法找到了共同祖先。这里还会对其高度与 `floor` 变量进行验证，前面说过 `floor` 变量代表的是共同祖先的高度的最小值，如果找到共同祖先的高度比这个值还小，就认为是两个节点之间分叉太大了，不再允许进行同步。如果一切正常，就返回找到的共同祖先的高度 `number` 变量。

如果 `hash` 变量是默认值，说明一直没有找到共同祖先。所以还需要继续使用二分法查找。

二分法

这里二分法查找共同祖先与我们熟悉的数据结构中的 「二分查找算法」是同样的原理，只不过被查找的数据不是一个已经完整存在的数据数组，而是每需要一个元素，都从远程节点获取。了解了这一点，相信这块代码也就很容易理解了。

我们来看一下二分查找的代码的框架代码。下面的代码是接「固定间隔法」的最后一段代码的：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......

    start, end := uint64(0), remoteHeight
    if floor > 0 {
        start = uint64(floor)
    }

    for start+1 < end {
```

```
        check := (start + end) / 2
        ......
        go p.peer.RequestHeadersByNumber(check, 1, 0, false)

        for arrived := false; !arrived; {
            select {
            case packer := <-d.headerCh:
                ......
                if !known {
                    end = check
                    break
                }
                ......
                start = check
                ......
            }
        }
    }
    ......
}
```

很显然这是一个很标准的二分查找方法。代码首先定义了二分查找的区块高度范围：`start` 和 `end`，然后在for循环中取其中间值 `check`，并在 for 循环内更新 `start` 或 `end` 的值。注意 `start` 的取值仍然考虑了 `floor` 的影响。

select/case 中的其它 case 与「固定间隔法」中的类似并且不重要，因此上面的代码中并没有提及它们。但我们要详细看一下 `headerCh` 这个case：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......

    for start+1 < end {
        check := (start + end) / 2
        ......

        for arrived := false; !arrived; {
            select {
            case packer := <-d.headerCh:
                //验证返回数据的节点是否是我们请求数据的节点
                //验证返回的headers的数量
                arrived = true

                h := headers[0].Hash()
                n := headers[0].Number.Uint64()

                var known bool
                switch d.mode {
                    known = ......
                }

                //区块不本地不存在，说明共同祖先的高度要比check变量小
                if !known {
                    end = check
                    break
                }

                //区块在本地存在
                //这里验证了一个远程返回区块的合法性
                header := d.lightchain.GetHeaderByHash(h)
                if header.Number.Uint64() != check {
                    return 0, errBadPeer
```

```
                }
                //既然区块在本地存在，说明共同祖先的高度要比check变量大
                start = check
                hash = h
            }
        }
    }
}
```

这段代码虽然稍长，但逻辑也依然很简单。在获取到远程返回的 header 后，代码会在本地查找这个 header 是否存在，如果存在，说明共同祖先的高度应该在 [check, end] 这个区间中；如果不存在，说明共同祖先的高度应该在 [start, check] 这个区间。如果存在共同祖先，则当满足 start+1>=end 时，`start` 变量存储的就是共同祖先的高度。

我们继续后二分查找的for循环结束后的代码：

```
func (d *Downloader) findAncestor(p *peerConnection, remoteHeader *types.Header) (uint64, error) {
    ......

    // Ensure valid ancestry and return
    if int64(start) <= floor {
        return 0, errInvalidAncestor
    }
    return start, nil
```

最后就非常简单了，无需进行过多解释。但我感觉有可能存在一个小问题：如果当前的高度小于 `MaxForkAncestry`，即start的值为0时，如果一直没有找到共同祖先，`start` 变量的值会一直不变一直为 0，因此高度为 0 的创世区块就会被返回当作共同祖先，但并没有地方可以保证两个节点的创世区块是相同的，也就是说创世区块也不一定就是共同的祖先。虽然这并不会对后面的逻辑造成破坏（创世块不一样时调用 blockchain 模块存储高度为 1 的区块时会检查失败），但我依然觉得这里是一个失误，应该和「固定间隔法」一样，判断一下 `hash` 变量的值是否是默认值，再认定 `start` 变量是否代表了共同祖先的区块。

header 的下载和 skeleton

在调用 `Downloader.findAncestor` 获取到共同祖先以后，就可以确定同步范围并开始进行区块同步了。以太坊中的区块数据是分开进行同步的，即先同步区块的 header，然后再同步区块的 body 等数据。这样做的原因是有不同的区块同步模式，它们对数据同步的方式不尽相同（参见文章开头对同步模式的介绍）。

在前面我们说过，header 数据是首先被同步的，在同步 header 时，有一个「skeleton」的概念，其想法是每次从某个节点获取的区块的数据是有限制的，大小为 `MaxHeaderFetch`，那么如果将要下载的 header 的数量大于这个值，就得将它们按高度分组，每组最多 `MaxHeaderFetch` 个（尾部可能会剩下不满 `MaxHeaderFetch` 个，它们不计入 skeleton）。从每组中选出最后一个元素，就组成了 skeleton。代码会先将 skeleton 中的所有 header 下载下来，然后为刚才的每一个分组随机选取空闲节点进行下载，也就是谓的「填充skeleton」（`Downloader.fillHeaderSkeleton`）。最后再下载尾部那不到 `MaxHeaderFetch` 个 header。

这种下载方式其实是避免了从同一节点下载过多错误数据这个问题。如果我们连接到了一个恶意节点，它可以创造一个链条很长且TD值也非常高的区块链数据（但这些数据只有它自己有，也就是说这些数据是不被别人承认的）。如果我们的区块从 0 开始全部从它那同步，显然我们察觉不到任意错误，也就下载了一些根本不被别人承认的数据。如果我只从它那同步 `MaxHeaderFetch` 个区块，然后发现这些区块无法正确填充我之前的 skeleton（可能是 skeleton 的数据错了，或者用来填充 skeleton 的数据错了），我就会发现错误并丢掉这些数据。虽然会失败，但不会让自己拿到错误数据而不知。

了解了基本的思路以后，我们就看看代码是如何实现的。header 的下载是在 `Downloader.fetchHeaders` 中：

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64, pivot uint64) error {
    skeleton := true            // Skeleton assembly phase or finishing up

    getHeaders := func(from uint64) {
        if skeleton {
            go p.peer.RequestHeadersByNumber(from+uint64(MaxHeaderFetch)-1, MaxSkeletonSize,
MaxHeaderFetch-1, false)
```

```
        } else {
            go p.peer.RequestHeadersByNumber(from, MaxHeaderFetch, 0, false)
        }
    }
    getHeaders(from)

    for {
        select {
        case <-d.cancelCh: ......
        case packet := <-d.headerCh:......
        case <-timeout.C: ......
    }
}
```

在一开始，代码创建了一个逆名函数用来发起获取 header 的请求。接着就马上调用了它。在这个函数中会判断是否是下载 skeleton,如果是则会从高度 `from+MaxHeaderFetch-1` 开始（包括），每隔 `MaxHeaderFetch-1` 的高度请求一个 header，最多请求 `MaxSkeletonSize` 个。

当前 `skeleton` 变量的值为 true，因此现在已经发起了获取 skeleton 的请求。接下来就是一个 for 循环等待 header 数据的到来并处理，在处理过程中还会调用 `getHeaders` 发起 header 数据的请求。我们继续往下看：

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64, pivot uint64) error {
    ......

    for {
        select {
        case packet := <-d.headerCh:
            ......
            //packet.Items()为0代表获取的header数据为0，也就是没有数据可以获取了
            //进入这个if分支说明skeleton下载完成了
            if packet.Items() == 0 && skeleton {
                skeleton = false
                getHeaders(from)
                continue
            }
            if packet.Items() == 0 {
                //进入这个if说明所有header下载完了，发消息给headerProcCh通知header已全部下载完成
                ......
                select {
                case d.headerProcCh <- nil:
                    return nil
                case <-d.cancelCh:
                    return errCancelHeaderFetch
                }
            }
        }
    }
}
```

在 `headerCh` 有消息时，会首先判断是否有数据。代码先处理没有数据的情况，如果没有数据且当前的状态是在下载 skeleton，说明 skeleton 下载完成了，因此调用 `getHeaders` 获取下载区间尾部的那些不满 `MaxHeaderFetch` 个区块、因而没有被计入 skeleton 小组中的 header。如果当前状态不是在下载 skeleton，说明区间尾部的 header 也已经下载完成了，代码就发送消息给 `headerProcCh` 通知 header 已经全部下载完成。

我们再来看有数据时的处理：

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64, pivot uint64) error {
    ......

    for {
        select {
```

```
        case packet := <-d.headerCh:
            ......

            if skeleton {
                filled, proced, err := d.fillHeaderSkeleton(from, headers)
                headers = filled[proced:]
                from += uint64(proced)
            } else {
                if n := len(headers); n > 0 {
                    head := ... // 本地 current block 的高度

                    // If the head is way older than this batch, delay the last few headers
                    if head+uint64(reorgProtThreshold) < headers[n-1].Number.Uint64() {
                        delay := reorgProtHeaderDelay
                        if delay > n {
                            delay = n
                        }
                        headers = headers[:n-delay]
                    }
                }
            }
        }
    }
}
```

这里首先判断是否是正在处理 skeleton，如果是，说明我们前面请求的 skeleton 数据到了，这里调用 `Downloader.fillHeaderSkeleton` 对其进行填充，这个方法会返回填充的所有headers和已处理的header的索引（所谓「已处理」是指header经由 `Downloader.processHeaders` 处理了），代码利用这两个返回值分别更新 `headers` 和 `from` 变量。

如果当前处理的不是 skeleton，说明我们区块同步得差不多了，只剩下尾部的一些区块还没同步成功了，现在收到的就是这些尾部的 header。这里会先判断本地的主链高度与新收到的 header 的最高高度的高度差是否在 `reorgProtThreshold` 以内，如果不是，就将高度最高的 `reorgProtHeaderDelay` 个 header 丢掉。这是为什么呢？我们先往下看，结合着下面的代码进行说明：

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64, pivot uint64) error {
    ......

    for {
        select {
        case packet := <-d.headerCh:
            ......

            // Insert all the new headers and fetch the next batch
            if len(headers) > 0 {
                select {
                case d.headerProcCh <- headers:
                case <-d.cancelCh: ......
                }
                from += uint64(len(headers))
                getHeaders(from)
            } else {
                // No headers delivered, or all of them being delayed, sleep a bit and retry
                select {
                case <-time.After(fsHeaderContCheck):
                    getHeaders(from)
                    continue
                case <-d.cancelCh: ......
                }
            }
        }
```

```
    }
}
```

在前面的 if 分支中，我们要么填充 skeleton，要么可能丢掉高度最高的 `reorgProtHeaderDelay` 个区块（也可能不丢掉）。那么现在 `headers` 变量里保存的就是经过刚才处理后的区块头了。第一个 if 分支很好理解，如果还有 header 未处理，就发给 `headerProcCh` 进行处理，`Downloader.processHeaders` 会等待这个 channel 的消息并进行处理；如果没有 header 要处理，就再等待 `fsHeaderContCheck` 秒，再次调用 `getHeaders` 请求区块，这是什么意思呢？结合上一段代码，我们一起来看一下这个逻辑。

其实这段代码一开始是不存在、后来才加上的，其 commit 的记录在这里，而 「pull request」 在这里。从 「pull request」 中作者的解释我们可以了解这段代码的逻辑和功能：这个修改主要是为了解决经常出现的 「invalid hash chain」 错误，出现这个错误的原因是因为在我们上一次从远程节点获取到一些区块并将它们加入到本地的主链的过程中，远程节点发生了 reorg 操作（参见这篇文章里关于「主链与侧链」的介绍 ）；当我们再次根据高度请求新的区块时，对方返回给我们的是它的新的主链上的区块，而我们没有这个链上的历史区块，因此在本地写入区块时就会返回 「invalid hash chain」 错误。

要想发生 「reorg」 操作，就需要有新区块加入。在以太坊主网上，新产生一个区块的间隔是 10 秒到 20 秒左右。一般情况下，如果仅仅是区块数据，它的同步速度还是很快的，每次下载也有最大数量的限制。所以在新产生一个区块的这段时间里，足够同步完成一组区块数据而对方节点不会发生 「reorg」 操作。但是注意刚才说的「仅仅是区块数据」的同步较快，state 数据的同步就非常慢了。在 fast 同步模式下，有一个 「pivot」 区块的概念，我们后面会详细说明，简单来说在完成同步之前可能会有多个 「pivot」 区块，这些区块的 state 数据会从网络上下载，这就大大拖慢了整个区块的同步速度，使得本地在同步一组区块的同时对方发生 「reorg」 操作的机率大大增加。

作者认为这种情况下发生的 「reorg」 操作是由新产生的区块的竞争引起的，所以最新的几个区块是「不稳定的」，如果本次同步的区块数量较多（也就是我们同步时消耗的时间比较长）（在这里「本次同步的区数数量较多」的表现是新收到的区块的最高高度与本地数据库中的最高高度的差距大于 `reorgProtThreshold`），那么在同步时可以先避免同步最新区块，这就是 `reorgProtThreshold` 和 `reorgProtHeaderDelay` 这个变量的由来。

关于后面的 sleep `fsHeaderContCheck` 秒，我觉得是一个疏忽，但不会引起错误。从 「pull request」 中 holiman 提问的问题看，作者一开始忽略区块的判断代码为：

```
if head+uint64(reorgProtThreshold) < headers[n-1].Number.Uint64() && n >= reorgProtHeaderDelay
```

后来被作者改成了现在的

```
if head+uint64(reorgProtThreshold) < headers[n-1].Number.Uint64()
```

但无论是改前还是改后，按照现在 `reorgProtThreshold` 和 `reorgProtHeaderDelay` 的取值（ 48 和 2 ），headers 都不可能被全部 delay。所以 sleep `fsHeaderContCheck` 秒的代码应该是执行不到的。

至此，`Downloader.fetchHeaders` 方法就结束了，所有的区块头也就下载完成了。

pivot

前面我们说过，fast 模式的同步方式是从其它节点下载 receipt 和某个较新的区块的 state 数据，这个区块被称为 pivot。在 pivot 区块之后，所有的区块都会按照 full 模式下那样，只使用 header 和 body 在本地计算 state 和 receipt 并验证通过后加入到数据库中，而不再使用网络上下载的 receipt 和 state。这一小节我们详细说说这个机制。

我们首先看一下 pivot 区块的高度是如何被选定的。在同步刚开始确定好了需要同步的区间以后，就计算了 pivot 的高度，具体代码如下：

```
func (d *Downloader) syncWithPeer(p *peerConnection, hash common.Hash, td *big.Int) (err error) {
    ......

    pivot := uint64(0)
    if d.mode == FastSync {
        if height <= uint64(fsMinFullBlocks) {
            origin = 0
```

```
        } else {
            pivot = height - uint64(fsMinFullBlocks)
            if pivot <= origin {
                origin = pivot - 1
            }
        }
    }
    ......
}
```

这里 `height` 变量代表的是对方节点的高度。可以看到，如果是 fast 模式且对方的高度超过了 `fsMinFullBlocks`，则 `pivot` 的值为 `height - fsMinFullBlocks`，否则为 0。这里 `origin` 变量代表的是将要下载的区块高度区间的起始高度（不含），从上面这段代码也可以看到如果在 fast 模式下，无论如何 `origin` 的值要小于或等于 `pivot`，也就是说将要下载的区块区间一定要包含 `pivot` 区块。

这里计算得到 `pivot` 变量有什么用呢？我们从源代码中可以很容易看到，这个变量传给了 `Downloader.fetchHeaders` 和 `Downloader.processHeaders` 两个方法。我们分别看看这两个方法里是怎么使用 `pivot` 这个参数的。不过要先说明的是，pivot 区块的主要逻辑其实在 `Downloader.processFastSyncContent` 中，而不是上面提到的代码段以及 `fetchHeaders` 和 `processHeaders` 中。但我们仍然要先介绍完这两个方法，再去看 `Downloader.processFastSyncContent`。

我们先看 `fetchHeaders`：

```
func (d *Downloader) fetchHeaders(p *peerConnection, from uint64, pivot uint64) error {
    ......
        case packet := <-d.headerCh:
            ......
            if packet.Items() == 0 {
                // Don't abort header fetches while the pivot is downloading
                if atomic.LoadInt32(&d.committed) == 0 && pivot <= from {
                    select {
                    case <-time.After(fsHeaderContCheck):
                        getHeaders(from)
                        continue
                    case <-d.cancelCh:
                        return errCancelHeaderFetch
                    }
                }
                ......
            }
    ......
}
```

`Downloader.committed` 字段代表的意义就是 pivot 区块的所有数据（包括 state）是否已经被提交到本地数据库中，如果是其值为 1，否则为 0。所以这里的代码就比较容易理解了：在接收到的 header 数量为 0（可能对方已经没有数据可以给我们了）的情况下，如果 pivot 区块还没有被提交到本地数据库且当前请求的区块高度大于 pivot 的高度，那么就等一会（`fsHeaderContCheck`）再次请求新的 header，这里也是为了等待 pivot 区块的提交。

我们再看一下 `processHeaders` 中对 `pivot` 参数的使用：

```
func (d *Downloader) processHeaders(origin uint64, pivot uint64, td *big.Int) error {
    ......
    if chunk[len(chunk)-1].Number.Uint64()+uint64(fsHeaderForceVerify) > pivot {
        frequency = 1
    }
    if n, err := d.lightchain.InsertHeaderChain(chunk, frequency); err != nil {
        ......
    }
```

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

`chunk` 变量代表的是一组将要写入本地数据库的 header。这里的 `pivot` 主要用来影响 `frequency` 的值，`frequency` 的值仅仅用来传递给 `lightchain.InsertHeaderChain`，其意义是在将要插入的所有区块头中，每隔多少个高度检查一下 header 的有效性。因此如果 `chunk` 中的区块的最高高度加上 `fsHeaderForceVerify` 大于 `pivot` 参数，那么对 header 的检查就公比较严格。

上面涉及到 pivot 的地方其实都不是逻辑的重点，重点逻辑在 `Downloader.processFastSyncContent` 中。我们来详细看看这个方法。

```
func (d *Downloader) processFastSyncContent(latest *types.Header) error {
    stateSync := d.syncState(latest.Root)
    ......

    pivot := uint64(0)
    if height := latest.Number.Uint64(); height > uint64(fsMinFullBlocks) {
        pivot = height - uint64(fsMinFullBlocks)
    }
    ......
}
```

这里 `latest` 参数就是同步开始时计算的同步区间的最高高度。在这个方法一开始，会首先去同步 `latest` 的 state 数据（但 `latest` 的state数据不重要），然后计算 `pivot` 变量。注意这里的计算方法与前面提到的同步刚开始时的计算是一样的。

我们继续看后面的代码。后面的代码是一个大的 for 循环，但逻辑有点难以理解且代码的关联度比较大，因此我将其精简以后全部贴了上来：

```
func (d *Downloader) processFastSyncContent(latest *types.Header) error {
    ......
    for {
        //等待有可用的完整的区块数据
        results := d.queue.Results(oldPivot == nil)
        ......
        if oldPivot != nil {
            results = append(append([]*fetchResult{oldPivot}, oldTail...), results...)
        }

        //如果满足条件，更新 pivot 区块。
        if atomic.LoadInt32(&d.committed) == 0 {
            latest = results[len(results)-1].Header
            if height := latest.Number.Uint64(); height > pivot+2*uint64(fsMinFullBlocks) {
                pivot = height - uint64(fsMinFullBlocks)
            }
        }

        //以 pivot 为界，切分 results
        P, beforeP, afterP := splitAroundPivot(pivot, results)
        if err := d.commitFastSyncData(beforeP, stateSync); err != nil {
            return err
        }

        //特殊处理 P：下载其 state 数据，并调用 commitPivotBlock 提交到本地数据库。
        if P != nil {
            // If new pivot block found, cancel old state retrieval and restart
            if oldPivot != P {
                stateSync.Cancel()
                stateSync = d.syncState(P.Header.Root)
                ......
                oldPivot = P
```

```
            }
            select {
            case <-stateSync.done:
                d.commitPivotBlock(P) //commitPivotBlock 会将 Downloader.committed 设置为 1
                oldPivot = nil

            case <-time.After(time.Second):
                oldTail = afterP
                continue
            }
        }

        //将 afterP 中的区块写入本地数据库。
        if err := d.importBlockResults(afterP); err != nil {
            return err
        }
    }
}
```

整个 for 循环的工作流程可以总结如下：

1. 获取下载完成的 block，以 pivot 为界将这些 block 切分成三块：P、beforeP、和afterP，其中 `P` 代表 pivot 区块。
2. 将 `beforeP` 写入本地数据库。注意这些区块是没有 state 数据的。
3. 如果当前获取的区块中存在 pivot 区块（即 `P` 不为空），则下载其 state 数据，并在下载成功后将 `P` 写入本地数据库。
4. 如果下载 `P` 的 state 数据超时，则使用 `oldTail` 存储还未写入本地数据库的 `afterP` ，然后重新从第 1 步开始。
5. 如果第 3 步中写入 pivot 成功，则将 `afterP` 写入本地数据库。注意这些区块的 state 是在本地计算出来的。

这里有一点比较绕的是 `oldPivot` 这个变量以及对它的使用。 `oldPivot` 代表的意思是我们发现了一个 pivot 区块，但还未成功写入本地数据库。代码中有两处判断 `oldPivot` 的值，第一处是：

```
if oldPivot != nil {
    results = append(append([]*fetchResult{oldPivot}, oldTail...), results...)
}
```

这里的意思是，如果我们有一个未写入数据库的 pivot 区块，那么 `oldTail` 中一定保存着上一次未加入本地数据库的 `afterP` 中的数据，我们需要将它们入到 `results` 中。而第二处判断 `oldPivot` 的代码是：

```
if oldPivot != P {
    stateSync.Cancel()
    ......
    oldPivot = P
}
```

这里是在判断是否有新的 pivot 区块，如果是则更新相关数据。

还有一个要特别注意的地方是， pivot 区块并非固定的某一个高度的区块，而是可能不断变化的，其更新代码如下：

```
func (d *Downloader) processFastSyncContent(latest *types.Header) error {
    ......
    if atomic.LoadInt32(&d.committed) == 0 {
        latest = results[len(results)-1].Header
        if height := latest.Number.Uint64(); height > pivot+2*uint64(fsMinFullBlocks) {
            pivot = height - uint64(fsMinFullBlocks)
        }
    }
```

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

即如果还没有 pivot 区块被提交过，且当前获取到的区块高度已经比当前的 pivot 高度超出了太多（2 * fsMinFullBlocks），就要利用新区块的高度重新计算 pivot。

为什么要及时更新 pivot 的高度呢？初始时得到一个值并下载它的 state，然后就可以在本地计算这个区块以后的区块的 state 和 receipt 数据，这样做有什么问题吗？

简单来说原因在于区块修剪功能的存在。虽然我们在区块同步的一开始就计算出了 pivot 区块的高度，但我们是在同步的后期才去下载 pivot 区块的 state 数据。在我们下载 pivot 的 state 之前这段时间，对方高度可能会继续增长，且可能远大于 `height`。因此当我们需要下载 pivot 区块的 state 数据时，有可能这个区块在对方节点中已经是一个很旧的区块了。由于旧区块的 state 是可能会被修剪的、不完整的（关于区块修剪的详细内容可能参看这篇文章），所以我们此时去下载这个（pivot）区块的 state 时，可能无法得到完整的数据。

这个问题在这个 pull request中也有提及：

> Since we added in-memory pruning about a year ago, the downloader cannot stick to a single pivot point throughout sync, rather it needs to push it forward if the chain progresses enough (since old state will become unavailable).

fetchParts

`Downloader.fetchParts`这个方法用来获取区块数据，在获取区块的三部分数据的方法中（fillHeaderSkeleton、fetchBodies、fetchReceipts），都是调用 `fetchParts` 实现的，但这个方法又过于复杂，我自己在看的时候反反复复好多次才看明白这个方法。尤其是其巨长无比的参数列表和实现代码，很轻易的让我产生了畏难的心理（可能正是这种畏难心理阻碍了自己，因为看明白以后发现其实实现得也挺简单的）。因此这里专门用一小节来说明一下这个方法。

`fetchParts` 方法中的很多逻辑和功能都和 `queue` 对象密不可分，但这篇文章里我们不会对 `queue` 对象进行详细的介绍（对于 `queue` 的详细分析请参看这篇文章），目前为止读者可将 `queue` 理解成一批区块数据的组装器。区块的下载顺序是先下载 header，再根据 header 去下载 body 和 receipt（如果是 fast 模式），`queue` 提供了这一过程中对这些数据的管理，可以告知调用者哪些数据可以开始下载了、哪些数据正在下载中、哪些数据下载超时等信息。

我们先来看一下它的整体结构代码：

```go
func (d *Downloader) fetchParts(......) error {
    ticker := time.NewTicker(100 * time.Millisecond)
    defer ticker.Stop()

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

    finished := false
    for {
        select {
        case <-d.cancelCh:......
        case packet := <-deliveryCh:......
        case cont := <-wakeCh:......
        case <-ticker.C:......
        case <-update:......
        }
    }
}
```

这样一看，其实这个方法的大逻辑还是比较清晰的。方法首先自己创建了 `ticker` 和 `update` 两个 channel，再加上参数 `deliveryCh` 和 `wakeCh`，`fetchParts` 的主框架就是循环等待这四个 channel 的消息。

其实这里几乎所有的逻辑都在处理 `update` 消息里，其它几个 channel 在收到消息并进行简单的处理后，都会通知 `udpate` 进行下一步的处理。在 `update` 处理代码中，包含了请求超时、下载完成或继续下载的相应处

理。

接下来我们分别分析一下每个 channel 收到消息以后，是如何处理的。

deliveryCh

`deliveryCh` 作用就是传递下载的数据，当有数据被真正下载下来时，就会给这个 channel 发消息将数据传递过来。这个 channel 是与 `Downloader` 对象的 Deliver 方法相关的（DeliverHeaders、DeliverBodies、DeliverReceipts）。从调用 `fetchParts` 的三处代码中可以看到，这个参数的实际值为别是 `Downloader.headerCh`、`Downloader.bodyCh`、`Downloader.receiptCh`。而这三个 channel 分别在以下三个方法中被写入数据：`Downloader.DeliverHeaders`、`Downloader.DeliverBodies`、`Downloader.DeliverReceipts`。我们之前提到过，真正的网络数据处理是在 `ProtocolManager.handleMsg` 中，正是在这个方法中收到数据后，就会调用相应的 Downloader 对象的 deliver 方法，继而将数据传递到 `fetchParts` 中。

我们先看一下 `deliveryCh` 消息是如何处理的：

```
func (d *Downloader) fetchParts(......) error {
    ......
    for {
        select {
        case packet := <- deliveryCh:
            if peer := d.peers.Peer(packet.PeerId()); peer != nil {
                accepted, err := deliver(packet)
                if err == errInvalidChain {
                    return err
                }
                if err != errStaleDelivery {
                    setIdle(peer, accepted)
                }
            }

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

在收到下载数据以后，会首先判断发送此数据的节点是否还有效，因为在有些情况下比如我们认为对方超时了，会将这个节点移除，但可能在真正断开连接之前，我们已收到了这个节点返回的数据。如果是这种情况，我们就直接忽略这些数据了。

如果节点是有效的，我们会调用 `deliver` 参数传递数据，并检查其返回值，直接返回错误或调用 `setIdle` 将节点设置为空闲状态。如果一切没有问题，就发消息给 `update` 进行下一步的处理。

`deliver` 参数是一个回调函数，其意义很明显，就是收到新下载成功的数据以后，将其传递到某个地方。从调用 `fetchParts` 处的代码看，这个参数所代表的函数都调用了 queue 对象的 Deliver 方法：`queue.DeliverHeaders`、`queue.DeliverBodies`、`queue.DeliverReceipts`。也就是说，收到下载成功的数据以后，都给了 queue 进行处理。

`setIdle` 是一个回调函数，它将某个节点（peer）的对某类数据的下载状态设置为空间的（idle）。这个函数的实际实现都是调用了 `peerConnection` 对象的相关方法：`SetHeadersIdle`、`SetBodiesIdle`、`SetReceiptsIdle`。这样当再次需要下载此类数据时，就可以从这些空闲的节点中选取一个进行下载。注意这里一个节点的空闲状态是针对某类数据而言的，比如节点没有在下载 header，它是一个 header 下载的空闲节点，但它仍可能正在下载 body 等数据。

wakeCh

对于 `wakeCh`， 从名字就可以知道，这个参数用来唤醒 `fetchParts`，收到这个消息代表有新的情况可以处理了（有新的数据可以下载，或下载已完成了）。我们再来看一下 `wakeCh` 消息的处理：

```
func (d *Downloader) fetchParts(......) error {
    ......
    for {
        select {
        case cont := <-wakeCh:
            if !cont {
                finished = true
            }

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

`wakeCh` 的消息处理也非常简单。代码会根据消息中的值设置 `finished`。这个 channel 传递的值的意义是，是否还有数据需要下载。这么说您可能还会觉得有些不明白，我们分别看一下不同数据的下载时 `wakeCh` 的发送情况就清晰了。

对于 header 的下载（其实只包含填充 skeleton 时的 heaer）， `wakeCh` 的值其实是 `queue.headerContCh`。只有在 `queue.DeliverHeaders` 中发现所有需要下戴的 header 都下载完成了时，才会发送 false 给这个 channel。 `fetchParts` 在收到这个消息时，就知道没有 header 需要下载了。发送消息给 `queue.headerContCh` 的代码如下：

```
func (q *queue) DeliverHeaders(......) (int, error) {
    ......
    if len(q.headerTaskPool) == 0 {
        q.headerContCh <- false
    }
    ......
}
```

在下载 body 或 receipt 时， `wakeCh` 的值其实是 `Downloader.bodyWakeCh` 或 `Downloader.receiptWakeCh`。在 `Downloader.processHeaders` 中，如果发现所有 header 已经下载完成了，那么发送 false 给这两个 channel，通知它们没有新的 header 了。注意这里仅仅是没有新的 header 了，而非没有新的 body 或 receipt 了，因为 body 和 receipt 的下载依赖于 header、需要 header 先下载完成才能下载，所以对于下戴 body 或 receipt 的 `fetchParts` 来说，收到这个 `wakeCh` 就代表不会再有通知让自己下载数据了、这是最后一次了。

如果在 `Downloader.processHeaders` 中正常处理了一批 header，就会发送 true 给这两个 channel，通知它们有新的处理可以处理了，并且后面可能还会有数据需要处理，你还不能结束（不能设置 `finished` 为 true）。

这两处发送消息的代码如下：

```
func (d *Downloader) processHeaders(origin uint64, pivot uint64, td *big.Int) error {
    for {
        select {
        case headers := <-d.headerProcCh:
            if len(headers) == 0 {
                for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
                    select {
                    case ch <- false:
                    case <-d.cancelCh:
```

```
                }
            }
            ......
            return nil
        }

        ......

        for _, ch := range []chan bool{d.bodyWakeCh, d.receiptWakeCh} {
            select {
            case ch <- true:
            default:
            }
        }
    }
}
```

ticker

`ticker.C` 的消息处理是非常简单的，这个 channel 只是为了周期性的激活 `update` 的消息处理，并没有其它作用。为了完整性这里仍然作为一小节将它列了出来。它的消息处理代码如下：

```
func (d *Downloader) fetchParts(......) error {
    ......
    for {
        select {
        case <-ticker.C:
            select {
            case update <- struct{}{}:
            default:
            }
        }
    }
}
```

update

`update` 是由其它三个 channel 触发的，它的消息处理实现了 `fetchParts` 的所有主要逻辑，其中涉及了比较多的概念。由于代码稍长，我们将其分成两部分分析。

在第一部分的代码中，主要是一些有效性判断，代码如下：

```
func (d *Downloader) fetchParts(......) error {
    ......
    for {
        select {
        case <-update:
            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 fails > 2 {
                        setIdle(peer, 0)
                    } else {
```

```
                        ......
                        d.dropPeer(pid)
                    }
                }
            }

            // If there's nothing more to fetch, wait or terminate
            if pending() == 0 {
                if !inFlight() && finished {
                    return nil
                }
                break
            }
        }
    }
}
```

代码首先会判断是否有有效节点，然后获取所有超时数据的信息，包括下载数据的节点 id 和数据的数量。如果某个节点下载的下载超时数据量太多（大于 2 个），有可能是因为我们过于高估了对方或自己的网络吞吐能力，因此调用 `setIdle` 将这个节点设置成空闲状态；如果超时的数据量小于 2 个，则直接断开与这个节点间的连接（这里其实我是有疑问的，如果一个节点在某次下载中应该下载 100 条数据，有一条超时了，这种情况也会导致断开连接。但我感觉这种情况断开连接是不对的）。

`setIdle` 在介绍 `deliveryCh` 时简单介绍过，因此这里我们只介绍一下 `expire`。`expire` 是一个回调函数，会返回当前所有的超时数据信息。这个函数的实际实现都是调用了 `queue` 对象的 Expire 方法：`queue.ExpireHeaders`、`queue.ExpireBodies`、`queue.ExpireReceipts`，这三个方法的参数就是超时阈值，这个值是 `downloader.requestTTL` 方法的返回值（关于 TTL 请参看 「RTT 与 TTL」小节）。在各个 Expire 方法中，会统计当前正在下载的数据中，起始时间与当前时间的差距超过给定阈值的数据，并将其返回。

在处理完超时数据以后，会调用 `pending` 来判断是否还有需要下载的数据。如果没有，还需要进一步调用 `inFlight` 和 使用 `finished` 变量进行判断。`finished` 是一个局部变量，在处理 `wakeCh` 的消息时有可能被设置为 true（参见「wakeCh」小节）。

`pending` 参数代表了一个回调函数，刚才我们提到，其目的是判断是否还有需要下载的数据。它的实际值是 `queue` 对象的 Pending 方法：`queue.PendingHeaders`、`queue.PendingBlocks`、`queue.PendingReceipts`。这些 Pending 方法会返回 `queue` 对象内部记录的各自待下载的任务数量。

`inFlight` 参数也是一个回调函数，从名字就可以看出，它的目的是判断是否还有正在下载的、但还没下载完成的数据（所谓「漂在空中」）。它的实际值是 `queue` 对象的 InFlight 方法：`queue.InFlightHeaders`、`queue.InFlightBlocks`、`queue.InFlightReceipts`。这些 InFlight 方法的实现与 Pending 方法类似，会返回 `queue` 对象内部记录的正在下载的数据数量。

（关于 `queue` 对象的详细分析，请参看这篇文章）

在有效性判断完成以后，我们再来看看后续的代码：

```
func (d *Downloader) fetchParts(......) error {
    ......
    for {
        select {
        case <-update:
            ......
            progressed, throttled, running := false, false, inFlight()
            idles, total := idle()

            for _, peer := range idles {
                if throttle() {
                    throttled = true
                    break
                }
                if pending() == 0 {
```

```
                    break
                }
                // Reserve a chunk of fetches for a peer. A nil can mean either that
                // no more headers are available, or that the peer is known not to
                // have them.
                request, progress, err := reserve(peer, capacity(peer))
                if err != nil {
                    return err
                }
                if progress {
                    progressed = true
                }
                if request == nil {
                    continue
                }
                if err := fetch(peer, request); err != nil {
                    panic(fmt.Sprintf("%v: %s fetch assignment failed", peer, kind))
                }
                running = true
            }
            // Make sure that we have peers available for fetching. If all peers have been tried
            // and all failed throw an error
            if !progressed && !throttled && !running && len(idles) == total && pending() > 0 {
                return errPeersUnavailable
            }
        }
    }
}
```

这一段代码虽然稍长，但主要逻辑还是很简单的，就是使用所有的空闲节点，调用 `fetch` 函数发送数据请求。在这个过程中，会有一些条件的判断，并用到了较多的参数。要理解这些条件的判断，就要先理解这些参数。所以我们先来分析一下这些参数的意义。

`inFlight` 参数我们刚才介绍过了，它用来返回正在下载的数据的数量。

`idle` 参数是一个回调函数，用来返回所有当前空闲的节点。它的实际值是 `peerSet` 对象的 IdlePeers 方法：`peerSet.HeaderIdlePeers`、`peerSet.BodyIdlePeers`、`peerSet.ReceiptIdlePeers`。在 downloader 模块中的 `peerConnection` 对象中，实现了对 eth 模块中的 `peer` 对象的封装，增加了一些状态信息，其中就包含是否空闲。而 `peerSet` 是一个对 `peerConnection` 的管理对象。关于 `peerConnection` 与 `peerSet` 的详细信息，请参看 「peerConnection 与 peerSet」小节。

`throttle` 参数也是一个回调函数，从名字可以看出，这个函数的功能是返回当前的下载是否需要暂停一下。对于 header 的下载，它的实际值是一个总是返回 false 的函数（可见下载 header 时是从来不会暂停一下的）；而对于 body 和 receipt，它的实际值是 `queue` 对象的 ShouldThrottle 方法：`queue.ShouldThrottleBlocks`、`queue.ShouldThrottleReceipts`。为什么在下载过程中需要暂停一下呢？其实主要是为了防止占用本地太多的内存。从 `queue` 对象的 ShouldThrottle 方法的实现来看，如果下载中的数据个数加上下载完成但仍在缓存中的数据个数，超过了 `limit`，就需要暂停一下；而 `limit` 最大值是 `blockCacheMemory` 这么大小的内存（64 M）所以存放的数据个数。（关于 `queue` 的详细实现，请参看这篇文章）

`reserve` 参数是一个回调函数，它的目的是从下载任务中选取一些可以下载的任务，并构造一个 `fetchRequest` 结构。它的实际实现都是调用 `queue` 对象的 Reserve 方法：`queue.ReserveHeaders`、`queue.ReserveBodies`、`queue.ReserveReceipts`。除了返回 `fetchRequest` 对象，它还返回一个 `process` 变量，标记着是否有空的数据正在被处理。比如有可能某区块中未包含任何一条交易，因此它的 body 和 receipt 都是空的，这种数据其实是不需要下载的。在 `queue` 对象的 Reserve 方法中，会对这种情况进行识别。如果遇到空的数据，这些数据会被直接标记为下载成功。在方法返回时，就将是否发生过「直接标记为下载成功」的情况返回。

`capacity` 参数也是一个回调函数，它的功能是用来决定此次下载需要请求的数据的个数。它的实际实现都是调用 `peerConnection` 的 Capacity 方法：`peerConnection.HeaderCapacity`、`peerConnection.BlockCapacity`、`peerConnection.ReceiptCapacity`。这些 Capacity 方法会根据自身所代表的节点在之前的下载过程中的吞吐量，来决定本次下载的数量。`capacity` 参与要与 `setIdle` 参数合起来

看。因为在调用 `setIlde` 时，会将节点本次接收到的数据个数进行记录。而 `peerConnection` 的 Capacity 方法正是通过上次 `setIdle` 记录的数据进行计算的。关于具体计算方法，可以参看「peerConnection 与 peerSet」小节。

`fetch` 参数也是一个回调函数，它的功能很明显，就是发送获取数据的请求。这个参数应该是最容易理解的，这里就不再赘言了。

在明白了这些参数的意义后，刚才的这段代码就比较好理解了。代码会使用所有的空闲节点下载数据，下载前会判断是否需要临时暂停一下、是否所有数据已经下载完成了；如果判断通过，则选取一定数据的下载任务，进行下载。在最后，如果没有遇到空区块需要下载、且没有暂停下载、且所有有效节点都空闲、且确实有数据需要下载，但下载没有运行起来，就返回 `errPeersUnavailable` 错误。

至此，我们把整个 `update` 消息的处理过程分析完成了。可以看到 `update` 消息的处理主要目的就是为了判断是否还有数据需要下载，如果有需要则发送获取数据的请求。在这个过程中由于和 `queue` 对象结合得太紧密，以及以太坊加了一些特性，导致代码看上去比较复杂。这些特性包括检查数据是否正在下载，限制下载过程中对内存的使用，统计节点空闲状态和吐吞量等。

RTT 与 TTL

如果我们自己写一个简单的网络 IO 程序，肯定会有对超时状况的处理。那么多长时间算超时呢？一般我们会估计一个较大的值，并将其在代码中写成一个常量。但在以太坊的 downloader 模块中不是这么做的，downloader 模块使用 「RTT」 和 「TTL」 这两个数据控制超时时间，并根据通信情况动态更新。下面我们就来看看这是如何实现的。

「RTT」 代表的是从发起请求到获取数据所需要的总时间（Request round trip time）。关于这个概念的解释有两点需要说明，下面我们分别说一下。

一是我们只是说「获取数据所需要的总时间」，并没有说有多少数据（网络 IO 的时间肯定跟数据量有关系的）。实际上这个总时间是指请求和获取 `MaxHeaderFetch` 条数据（header 或 body 或 receipt）的总时间。因为这个值只有在填充 skeleton 时才会被使用和更新，而 skeleton 中每次请求几乎都是以「组」进行的，每组 `MaxHeaderFetch` 条数据。何以确定只有在填充 skeleton 时才会用到 「RTT」 呢？阅读代码你就会发现，在 `Downloader.findAncestor` 和 `Downloader.fetchHeaders` 中，都是直接使用的 `peerConnection.peer` 直接发起数据请求，而不是使用的 `peerConnection` 对象，更没有调用 `peerConnection` 对象的 setIdle 相关方法；而在填充 skeleton 过程中的 `Downlaoder.fetchParts` 方法，就调用了 `peerConntection` 的 Fetch 方法（FetchHeaders、FetchBodies、FetchReceipts等）和 SetIdle 方法（SetHeadersIdle、SetBodiesIdle、SetReceiptsIdle等）。而「RTT」的计算和更新正是在 Fetch 方法和 SetIdle 方法中完成的。

二是我们刚才只说到「数据」和「`MaxHeaderFetch` 条数据」，并没有说单条数据是什么样子的。实际上这里的数据是指区块的 header、body、receipt 中的任意一类，因此 「RTT」 也就代表请求并获取某类 `MaxHeaderFetch` 条数据的最长时间。这是因为单个节点的 RTT 是在 `peerConnection.setIdle` 方法中更新的，而这个方法被 `peerConnection.SetHeadersIdle`、`peerConnection.SetBodiesIdle`、`peerConnection.SetReceiptsIdle` 三个方法调用。

「TTL」 代表的是单个请求所允许的最长时间。超过这个时间没有返回数据，就认为对方节点超时了。这个概念比较简单，就不再多说了。

清楚了这两个概念，下面我们看看它们是如何运作的。首先对于每个节点来说，都会有一个 RTT 值，这个值记录在 `peerConnection.rtt` 中。当有一个新节点接入时，会给它一个初始值，这个值就是当前所有已连接的节点的 RTT 的中位数：

```
func (ps *peerSet) Register(p *peerConnection) error {
    p.rtt = ps.medianRTT()
    ......
}
```

关于 `peerSet.medianRTT` 的实现分析请参看 「peerConnection 与 peerSet」 节中的 「peerSet」 小节。

这个初始值毕竟只是其它节点的中位数，跟新接入节点的实际情况可能相差很远，因此在后续的数据通信过程中，还会对这个节点的 RTT 不断更新，使其接近于此节点的真实情况。更新的方法是在每个 Fetch 方法中（`peerConnection.FetchHeaders`、`peerConnection.FetchBodies`等）记录 fetch 的起始时间，在每个 SetIdle 方法中（`peerConnection.SetHeadersIdle`、`peerConnection.SetBodiesIdle`等）计算 fetch 的起

始时间到当前时间的差距，就是 RTT 的值。下面是相关代码实现（由于各类数据的实现比较类型，因此这里只使用 `peerConnection.FetchHeaders` 和 `peerConnection.SetHeadersIdle` 举例）：

```
func (p *peerConnection) FetchHeaders(from uint64, count int) error {
    ......

    //在 headerStarted 中记录 fetch 的起始时间
    p.headerStarted = time.Now()
    ......
}

func (p *peerConnection) SetHeadersIdle(delivered int) {
    p.setIdle(p.headerStarted, delivered, &p.headerThroughput, &p.headerIdle)
}

func (p *peerConnection) setIdle(started time.Time, delivered int, throughput *float64, idle
*int32) {
    ......

    //利用 time.Since 计算时间差
    elapsed := time.Since(started) + 1 // +1 (ns) to ensure non-zero divisor
    ......
    //更新 rtt
    p.rtt = time.Duration((1-measurementImpact)*float64(p.rtt) +
measurementImpact*float64(elapsed))
    ......
}
```

注意这里更新 RTT 的方法，新的 RTT 值由两部分组成：旧的 RTT 值和刚得到的时间差。其中旧的 RTT 值占比 `(1 - measurementImpact)`，也就是 0.9，刚得到的时间差占比 `measurementImpact`，即 0.1。

现在每个节点的 RTT 有了，在实际下载时怎么使用呢？在 `Downloader` 对象中，有一个叫做 `Downloader.qosTuner` 的 go routine，它会每隔一段时间计算一个属于 `Downloader` 对象的 RTT 值：

```
func (d *Downloader) qosTuner() {
    for {
        // 使用旧值和当前所有节点 RTT 中位数更新 rtt 的值
        rtt := time.Duration(
            (1-qosTuningImpact)*float64(atomic.LoadUint64(&d.rttEstimate)) +
            qosTuningImpact*float64(d.peers.medianRTT())
        )
        atomic.StoreUint64(&d.rttEstimate, uint64(rtt))

        // rtt 更新了，信心指数（rttConfidence）也需要跟着更新
        conf := atomic.LoadUint64(&d.rttConfidence)
        conf = conf + (1000000-conf)/2
        atomic.StoreUint64(&d.rttConfidence, conf)

        select {
        case <-d.quitCh:
            return
        case <-time.After(rtt):
        }
    }
}
```

这个方法中每隔一段时间就更新一下 `Downloader.rttEstimate` 的值。更新方法与单个节点的 RTT 的更新类似，也是使用两部分数据进行更新：旧值占比 `(1-qosTuningImpact)`，即0.75；当前的所有节点中位数占比 `qosTuningImpact`，即 0.25。

除了 `Downloader.rttEstimate`，这个方法还会更新 `Downloader.rttConfidence` 字段，我把它叫做「信心指数」，即表示「在多大程序上相信 `Downloader.rttEstimate` 接近真实情况」，它的值总是小于或等于 1000000，在 `Downloader.qosTuner` 的实现中，根据之前 `Downloader.rttEstimate` 的值，新计算的值是等于或无限接近于 1000000 的。

刚才我们说过，每当有新节点接入时，新节点的 RTT 是一个估算值，可能与实际情况相差很大。因此当有新节点接入时，RTT 的值相对来说是不可靠的，只有当进行了一段时间的数据通信以后，才会重新变得可靠一些。这个「信心指数」就是为了应对这种情况。 `Downloader` 对象有一个 `Downloader.qosReduceConfidence` 方法，这个方法就是用来减小「信心指数」的值的，而这个方法只有在有新的节点接入（`Downloader.RegisterPeer`）时才会被调用。也就是说，当有新节点加入时，由于其 RTT 是估算的，所以 `Downloader.rttEstimate` 也变得不那么可靠了，因此得减少一下「信心指数」即 `Downloader.rttConfidence` 的值。

那么 downloader 模块是如何利用这个「信心指数」的呢？实际上它是用来帮助计算 TTL，即请求超时时间的：

```
func (d *Downloader) requestTTL() time.Duration {
    var (
        rtt  = time.Duration(atomic.LoadUint64(&d.rttEstimate))
        conf = float64(atomic.LoadUint64(&d.rttConfidence)) / 1000000.0
    )
    ttl := time.Duration(ttlScaling) * time.Duration(float64(rtt)/conf)
    if ttl > ttlLimit {
        ttl = ttlLimit
    }
    return ttl
}
```

可以看到，RTT 和「信心指数」一起用来计算 TTL。由于 `Downloader.rttConfidence` 等于或无限接近于 1000000，因此这个方法里 `conf` 变量的值是 1 或小于 1。所以 `rtt / conf` 的值应该大于或等于 `rtt`。也就是说，「信心指数」越小，得到的 TTL 值越大。即越没有信心，越要将超时间设置得宽松一些。

RTT 除了用来计算 TTL，还会和 `peerConnection` 的 Capacity 方法（`peerConnection.HeaderCapacity` 等）一起计算某次下载的最大请求的条数。具体细节我们在「peerConnection 与 peerSet」小节中说明。

peerConnection 与 peerSet

在介绍区块同步协议的文件中我们提到过，代表远程节点的是 `peer` 对象，管理这些对象的是 `peerSet`，它们实现在 eth/peer.go 文件中。而在 downloader 模式中，代表远程节点的是 `peerConnection` 对象，管理这些对象的也叫 `peerSet`，但它们实现在 eth/downloader/peer.go 文件中。这一小节我们就来看看 downloader 模块中的 `peerConnection` 与 `peerSet`。

peerConnection

首先可以很明确地说， `peerConnection` 是对 `eth/peer` 对象的封装，发起数据请求的功能仍然直接调用的是 `eth/peer` 对象，但增加了其它一些统计功能。而 `downloader/peerSet` 虽然与 `eth/peerSet` 代码上没什么关系，但它们的功能是类似的，都是对各自的 peer 对象的管理。

我们刚才提到过， `peerConnection` 用来发起数据请求的功能仍然使用的是 `eth/peer` 的方法，这在 `peerConnection` 对象的 Fetch 方法（`peerConnection.FetchHeaders`、`peerConnection.FetchBodies` 等）的实现中可以看出来，非常简单不再赘述。我们这里重点关注 `peerConnection` 都提供了哪些统计功能：

- idle status: 标记自己是否正在下载某类数据
- capacity: 计算自己在一个 RTT 周期内最多可以下载多少条数据
- lack: 标记自己下载失败的数据

我们分别看一下这三个统计功能。

idle status

`peerConnection` 对象有四个字段，用来记录此对象是否正在下载某类数据，它们分别是：

- headerIdle
- blockIdle
- receiptIdle
- stateIdle

根据字面意义就可以知道它们分别代表了哪类数据的下载状态。它们都是 `int32` 类型，在 fetch 数据之前，先将相应的字段设置为1，比如在 fetch header 前：

```
func (p *peerConnection) FetchHeaders(from uint64, count int) error {
    ......
    if !atomic.CompareAndSwapInt32(&p.headerIdle, 0, 1) {
        return errAlreadyFetching
    }
    ......
}
```

fetch 其它类型的数据的代码与这个是类似的。

当数据下载完成时，会调用 SetIdle 方法（`peerConnection.SetHeadersIdle`、`peerConnection.SetBodiesIdle` 等），将相应的字段的值设置为 0。还是以 header 举例：

```
func (p *peerConnection) SetHeadersIdle(delivered int) {
    p.setIdle(p.headerStarted, delivered, &p.headerThroughput, &p.headerIdle)
}

func (p *peerConnection) setIdle(started time.Time, delivered int, throughput *float64, idle
*int32) {
    defer atomic.StoreInt32(idle, 0)

    ......
}
```

标记 idle 状态的目的是为了有新的数据请求时，可以选取空闲的节点用来发起请求，从而避免某一节点过于忙碌：

```
func (d *Downloader) fetchParts(...) {
    ......
    idles, total := idle()

    for _, peer := range idles {
        ......
        if err := fetch(peer, request); err != nil {
            ......
        }
    }

    ......
}
```

可以看到在 `Downloader.fetchParts` 中使用所有的 idle 节点 fetch 数据。而 `idle` 参数就是 `peerSet` 的 IdlePeers 方法（如 `peerSet.HeaderIdlePeers`、`peerSet.BodyIdlePeers` 等），比如在 `peerSet.HeaderIdlePeers` 方法中，就是使用了 `peerConnection.headerIdle` 来判断节点是否空闲。（关于 `peerSet.HeaderIdlePeers` 的分析请参看后面的「peerSet」小节）

capacity

`peerConnection` 提供了四个 Capacity 方法，用来计算自己在一个 RTT 周期内大约可以同步多少条数据，这四个方法是：

- HeaderCapacity

- BlockCapacity
- ReceiptCapacity
- NodeDataCapacity

它们的实现都是类似的，我们以 header 数据为例看一下 `peerConnection.HeaderCapacity` 的实现：

```
func (p *peerConnection) HeaderCapacity(targetRTT time.Duration) int {
    p.lock.RLock()
    defer p.lock.RUnlock()

    return int(math.Min(
        1 + math.Max(
            1,
            p.headerThroughput * float64(targetRTT)/float64(time.Second)
        ),
        float64(MaxHeaderFetch))
    )
}
```

这个计算看上去复杂，其实很好理解。除了 `math.Min` 和 `math.Max`，最重要的就是 `p.headerThroughput * float64(targetRTT)/float64(time.Second)` 这个式子。`targetRTT` 参数是一个 RTT 值，我们一会再来看它的实际值。所以要看懂这个式子，关键是理解 `headerThroughput` 这个字段的意义。其实 `headerThroughput` 代表的是当前节点一秒钟传输的数据的条数。因此这个式子先把 RTT 换算成秒数，然后乘以 `headerThroughput` 得到一个 RTT 周期内可以传输多少条数据。

如何知道 `peerConnection.headerThroughput` 代表的是一秒钟传输的数据的条数呢？这要看 SetIdle 方法。我们还是以 header 数据举例：

```
func (p *peerConnection) SetHeadersIdle(delivered int) {
    p.setIdle(p.headerStarted, delivered, &p.headerThroughput, &p.headerIdle)
}

func (p *peerConnection) setIdle(started time.Time, delivered int, throughput *float64, idle
*int32) {
    ......

    elapsed := time.Since(started) + 1 // +1 (ns) to ensure non-zero divisor
    //measured 代表 1 秒时间内下载的数据量
    measured := float64(delivered) / (float64(elapsed) / float64(time.Second))

    *throughput = (1-measurementImpact)*(*throughput) + measurementImpact*measured

    ......
}
```

`peerConnection.SetHeadersIdle` 在一轮下载完成后被调用。在此方法中，直接调用 `peerConnection.setIdle` 方法，第三个参数就是 `peerConnection.headerThroughput` 的指针，而 `delivered` 参数代表本轮成功下载的数据的数量。在 `peerConnection.setIdle` 中，首先得到本轮下载消耗的时间 `elapsed`，然后将本次下载的数据量转换成 1 秒时间内下载的数据量 `measured`，再用 `measured` 动态更新 `throughput` 参数，也即 `peerConnection.headerThroughput`。

lack

`peerConnection` 还提供了记录和查询当前节点下载失败的数据的信息，这是通过两个方法来实现的：`peerConnection.MarkLacking` 和 `peerConnection.Lacks`。当下载某数据失败时，就调用 `peerConnection.MarkLacking` 将这条数据对应的 header 的哈希标记为「缺失的」。`peerConnection` 内部使用 `lacking` 字段记录着这个信息。

当 `queue` 对象从任务队列中取出需要下载的任务时，会调用 `peerConnection.Lacks` 来确认此节点没有「缺失」这条数据，如果缺失了就暂时跳过这条数据：

```
func (q *queue) reserveHeaders(...) {
    ......
    for proc := 0; proc < space && len(send) < count && !taskQueue.Empty(); proc++ {
        ......
        if p.Lacks(hash) {
            skip = append(skip, header)
        } else {
            send = append(send, header)
        }
    }
    ......
}
```

peerSet

`peerSet` 对象提供了对 `peerConnection` 的管理功能。除了常规的功能（如使用 id 查询 `peerConnection` 对象、获取当前接入的节点个数、获取所有当前接入的节点等）之外，`peerSet` 主要还有两个功能：返回空闲的节点，和计算所有节点的 RTT 的中位数值。

返回空闲节点的实现很简单，我们以 header 数据为例看一下它的实现：

```
func (ps *peerSet) HeaderIdlePeers() ([]*peerConnection, int) {
    idle := func(p *peerConnection) bool {
        //使用 peerConnection.headerIdle 判断是否空闲
        return atomic.LoadInt32(&p.headerIdle) == 0
    }
    throughput := func(p *peerConnection) float64 {
        p.lock.RLock()
        defer p.lock.RUnlock()
        return p.headerThroughput
    }
    return ps.idlePeers(62, 64, idle, throughput)
}

func (ps *peerSet) idlePeers(minProtocol, maxProtocol int, idleCheck func(*peerConnection) bool,
throughput func(*peerConnection) float64) ([]*peerConnection, int) {
    ps.lock.RLock()
    defer ps.lock.RUnlock()

    //收集所有空闲节点
    idle, total := make([]*peerConnection, 0, len(ps.peers)), 0
    for _, p := range ps.peers {
        if p.version >= minProtocol && p.version <= maxProtocol {
            if idleCheck(p) {
                idle = append(idle, p)
            }
            total++
        }
    }

    //将所有空闲节点按下载速度由高到低排序
    for i := 0; i < len(idle); i++ {
        for j := i + 1; j < len(idle); j++ {
            if throughput(idle[i]) < throughput(idle[j]) {
                idle[i], idle[j] = idle[j], idle[i]
            }
        }
    }
```

```
    return idle, total
}
```

可以看到 `peerSet` 正是使用 `peerConnection.headerIdle` 来判断某个节点是否是 header 空闲的。在获取所有 header 空闲的节点以后，还会根据节点每秒的数据下载数量（`peerConnection.headerThroughput`）对其进行排序。

计算所有节点 RTT 中位数的功能在 `peerSet.medianRTT` 中实现：

```
func (ps *peerSet) medianRTT() time.Duration {
    ps.lock.RLock()
    defer ps.lock.RUnlock()

    //收集所有节点的 rtt 并排序
    rtts := make([]float64, 0, len(ps.peers))
    for _, p := range ps.peers {
        p.lock.RLock()
        rtts = append(rtts, float64(p.rtt))
        p.lock.RUnlock()
    }
    sort.Float64s(rtts)

    //获取中位数。如果连接的节点数多于 qosTuningPeers 个（5 个），
    //就用 rtt 值最高的 5 个节点的中位数（网络最好的 5 个节点的中位数）
    median := rttMaxEstimate
    if qosTuningPeers <= len(rtts) {
        median = time.Duration(rtts[qosTuningPeers/2])
    } else if len(rtts) > 0 {
        median = time.Duration(rtts[len(rtts)/2])
    }

    //限制最大最小值
    if median < rttMinEstimate {
        median = rttMinEstimate
    }
    if median > rttMaxEstimate {
        median = rttMaxEstimate
    }
    return median
}
```

可以看到这个方法使用 `peerConnection.rtt` 字段收集每个节点的 RTT 值，并尽可能取值最大的 5 个节点的中位数。

stateSync

在前面介绍 「pivot」 的概念的时候我们提到过，pivot 区块的 state 数据需要从其它节点下载。那么这一小节里，我们就来看看 downloader 模块是如何同步 pivot 区块的 state 数据的。

state 数据的同步代码在 eth/downloader/statesync.go 中，但整个同步过程需要 trie 模块中的 `trie.Sync` 对象的配合。因为所谓的 state 数据，实际上就是一棵 trie 树，而同步过的过程，就是在同步这棵树上的每一个节点。在最开始的时候，downloader 只有 trie 树的根哈希（Root），使用这个哈希同步到的就是树的根节点。那么接下来需要同步的节点的哈希，就需要 `trie.Sync` 帮忙从这个根结点解析出来（trie 的每个节点会记录子节点的哈希），然后 downloader 才可以使用这些哈希去同步相应的节点。依此不断进行，直到最底层的叶子节点同完成，整个 trie 树就同步完成了。

下面我们按照同步流程中的先后顺序，依次看看 state 的同步是如何实现的。当 `Donwloader` 对象想要同步 state 数据时，它首先调用 `Downloader.syncState` 获取一个 `stateSync` 对象，因此 `Downloader.syncState` 方法也是 state 同步流程的起点。我们先看看这个方法的实现：

```
func (d *Downloader) syncState(root common.Hash) *stateSync {
    s := newStateSync(d, root)
    select {
    case d.stateSyncStart <- s:
    case <-d.quitCh:
        s.err = errCancelStateFetch
        close(s.done)
    }
    return s
}
```

这个方法很简单，它首先调用 `newStateSync` 生成一个新的 `stateSync` 对象，参数为将要同步的 state 的 root 哈希。然后将这个新的对象发送给 `Downloader.stateSyncStart`。最后返回这个新的对象。

那么谁在等待 `Downloader.stateSyncStart` 这个 channel 的消息呢？查看代码可以发现，这个 channel 在 `Downloader.stateFetcher` 方法中被等待。而这个方法实际上是一个运行中的 goroutine，因为在 `Downloader` 对象生成时，会使用 `Downloader.stateFetcher` 方法创建一个 goroutine：

```
func New(...) *Downloader {
    ......
    go dl.stateFetcher()
    ......
}
```

现在我们看一下 `Downloader.stateFetcher` 的实现：

```
func (d *Downloader) stateFetcher() {
    for {
        select {
        case s := <-d.stateSyncStart:
            for next := s; next != nil; {
                next = d.runStateSync(next)
            }
        case <-d.stateCh:
            // Ignore state responses while no sync is running.
        case <-d.quitCh:
            return
        }
    }
}
```

这个方法的实现也很简单，在一个 for 循环里，不断的等待 `Downloader.stateSyncStart` 的消息，然后调用 `Downloader.runStateSync` 处理消息中的数据。这里比较令人疑惑的是处理 `Downloader.stateSyncStart` 消息的 for 循环。乍看上去，还真弄不明白这么写的意义。仅从这段代码上看，好像 `Downloader.runStateSync` 有可能会返回一个新的 `stateSync` 对象（`next` 变量），然后又调用 `Downloader.runStateSync` 进行处理。因此我们先看看 `Downloader.runStateSync` 中关于返回值的代码吧：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......

    go s.run()
    //方法退出的之前，先前之前的同步动作中止掉
    defer s.Cancel()

    for {
        ......
        select {
        // The stateSync lifecycle:
        case next := <-d.stateSyncStart:
```

```
            return next
        case <-s.done:
            return nil

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

在这个方法中，只有收到 `stateSync.done` 或 `Downloader.stateSyncStart` 消息时才会返回。`stateSync.done` 从字面意思上就能看出来，代表区块数据下载完成了，因此返回 nil；而 `Downloader.stateSyncStart` 这个 channel，正是刚才在 `Downloader.stateFetcher` 方法中提到的那个 channel。这到底是什么意思呢？

其实代码这么写是为了在同步某个 state 数据的同时，能够及时发现新的 state 同步请求并进行处理，而处理方式就是停掉之前的同步进程（注意 `Downloader.runStateSync` 中 `defer s.Cancel()` 这条语句），然后返回新的请求并重新调用 `Downloader.runStateSync` 进行处理。

如此实现 state 同步的逻辑是因为，在整个区块同步过程中，同一时间只有一个区块的 state 数据被同步；如果要同步新的 state，需要先停掉旧的同步过程。这从 state 同步的发起处的代码可以看出来：

```
func (d *Downloader) processFastSyncContent(latest *types.Header) error {
    stateSync := d.syncState(latest.Root)
    ......
    for {
        ......
        if P != nil {
            if oldPivot != P {
                stateSync.Cancel()

                stateSync = d.syncState(P.Header.Root)
                ......
            }
        }
        ......
    }
    ......
}
```

`Downloader.processFastSyncContent` 是一个 gotoutine，它一开始就调用 `Downloader.syncState` 发起了对 state 数据的请求。而在后续的代码中，如果发现 pivot 区块发生了改变（`oldPivot != P`），那么就会中终之前对 state 数据的请求（`stateSync.Cannel()`），并重新调用 `Downloader.syncState` 并使用新的 pivot 区块中的 Root 哈希发起 state 同步。

（注意在 `Downloader.processFastSyncContent` 中发起新的 state 同步之前，已经调用了 `stateSync.Cannel` 方法，并且这个方法会一直等待同步过程真的退出了才会返回。因此我认为代码实际运行时，`Downloader.runStateSync` 方法中应该接收到 `stateSync.done` 的消息的概率远大于 收到 `Downloader.stateSyncStart` 消息的概率，甚至没有必要在 `Downloader.stateFetcher` 中弄一个 for 循环和在 `Downloader.runStateSync` 中接收处理 `Downloader.stateSyncStart` 消息。）

（无论如何，我都觉得 `Downloader.stateFetcher` 中的这个 for 循环设计得相当另类，完全有其它更清晰的方法实现同步的功能和逻辑。）

在解释完 `Downloader.stateFetcher` 后，我们再来看看它调用的方法 `Downloader.runStateSync`。这个方法有些长，我们分开来看，先看看第一部分：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    var (
        active   = make(map[string]*stateReq) // Currently in-flight requests
        finished []*stateReq                  // Completed or failed requests
```

```
        timeout  = make(chan *stateReq)       // Timed out active requests
    )
    defer func() {
        // 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))
        }
    }()
    // Run the state sync.
    go s.run()
    defer s.Cancel()

    // Listen for peer departure events to cancel assigned tasks
    peerDrop := make(chan *peerConnection, 1024)
    peerSub := s.d.peers.SubscribePeerDrops(peerDrop)
    defer peerSub.Unsubscribe()

    ......
}
```

方法的一开始定义了三个变量，接下来创建了一个 goroutine 调用 `stateSync.run` 开始对 state 进行同步，并在方法退出时调用 `stateSync.Cannel`。这里要稍微解释一下开头定义的三个变量的意义，因为后面的代码要用到它们：

- active: 正在处理中的请求集合
- finished: 已经完成的请求集合（不管成功或失败）
- timeout: 如果正在处理的请求发生超时，使用这个 channel 进行通知

然后我们继续看后面的代码。之后的代码是一个大的 for 循环，在循环内部等待和处理各种 channel 消息。整个代码乍看上去是比较复杂、难以理解了。因此我调整了一下 select/case 语言中的一些 channel 的位置，先我们先看一看一些比较容易理解的 channel 的处理：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......
    for {
        ......
        select {
        case next := <-d.stateSyncStart:
            return next
        case <-s.done:
            return nil
        case p := <-peerDrop: ......

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

上面代码中摘出来的这三个 channel 是目前为止比较好理解的。`Downloader.stateSyncStart` 这个 channel 我们前面已经说过了，它是为了可以及时发现新的 state 同步请求并处理。`stateSync.done` 从名字就可以看出来，代表 state 同步完成了。而 `peerDrop` 这个变量也从名字上就能看出来，代表有节点断开连接了，因此要作一些相应的处理。

除去这三个 channel 之外，for 循环中还会处理这些 channel： `Downloader.trackStateReq` 、 `timeout` 、 `Downloader.stateCh` 、 `deliverReqCh` 。它们的处理代码乍看上去比较复杂，好像不知道要做什么事。其实只要理解了处理这些 channel 的时间顺序，整个逻辑就很好理解了。因此下面我们先给出这些 channel 的时间顺序，再按这个时间顺序逐个介绍它们的处理代码。它们的时间顺序是这样的：

如上图所示，首先接收到的消息是 `Downloader.trackStateReq`，它实际上代表 `stateSync` 对象发起了一次请求。我们先看看什么情况下会给这个 channel 发送消息：

```
func (s *stateSync) assignTasks() {
    ......
        req := &stateReq{peer: p, timeout: s.d.requestTTL()}
        s.fillTasks(cap, req)
        ......
        select {
        case s.d.trackStateReq <- req:
            req.peer.FetchNodeData(req.items)
        .....
        }
    ......
}
```

从这段代码可以看出，当 `stateSync` 准备好一个请求对象时（`req`），把这个请求对象发送给 `Downloader.trackStateReq` 后，立即调用 `peerConnection.FetchNodeData` 发起数据请求。

下面我们再看看从 `Downloader.trackStateReq` 中收到数据后是如何处理的：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......
    for {
        ......
        select {
        case req := <-d.trackStateReq:
            if old := active[req.peer.id]; old != nil {
                old.timer.Stop()
                old.dropped = true

                finished = append(finished, old)
            }
            // 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:
                }
            })
            active[req.peer.id] = req
        }
    }
}
```

收到请求数据 `req` 后，首先通过 `active` 判断这个节点是否有正在处理的请求，如果是则将旧的请求中断并加入 `finished` 中（完成但失败了）。然后为新的 `req` 设置一个 timer 后，将 `req` 加入到 `active` 中。（还记得 `active` 代表正在处理中的请求的集合吗）

处理过 `Downloader.trackStateReq` 代表一个请求已经发出了，那么接下来会发生两种情况：成功收到数据，或超时。我们先看看超时的情况。刚才处处理 `Downloader.trackStateReq` 消息的代码中，给 `req` 变量设置了一个 timer，它的超时后的处理函数中将 `req` 这个请求发给了 timeout 变量：

```
    req.timer = time.AfterFunc(req.timeout, func() {
        select {
        case timeout <- req:
        case <-s.done:
        }
    })
```

所以当 `timeout` 收到消息时，代表某个请求超时了，我们看代码是怎么处理的：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......
    for {
        ......
        select {
        case req := <-timeout:
            if active[req.peer.id] != req {
                continue
            }
            // Move the timed out data back into the download queue
            finished = append(finished, req)
            delete(active, req.peer.id)
        }
    }
}
```

很简单，如果超时的请求在 `active` 中，则将其从 `active` 中删除，并将其加入到 `finished` 中（完成但失败了）。

另一种情况是成功接收到数据， `Downloader.stateCh` 代表了这种情况。当 eth 模块接收到「NodeDataMsg」消息时会调用 `Downloader.DeliverNodeData` 方法，而 `Downloader.stateCh` 正在在这个方法中被触发的。我们来看看这个消息的处理代码：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......
    for {
        ......
        select {
        case pack := <-d.stateCh:
            req := active[pack.PeerId()]
            if req == nil {
                continue
            }
            // Finalize the request and queue up for processing
            req.timer.Stop()
            req.response = pack.(*statePack).states

            finished = append(finished, req)
            delete(active, pack.PeerId())
        }
    }
}
```

接收到远程节点返回的数据后，代码首先判断是否在 `active` 中。如果在则将返回的数据放到 `req.response` 中，并将 `req` 写入 `finished` 中，然后从 `active` 中删除。

不管是正常接收到数据还是超时，都会将结果写入 `finished` 变量中，然后就该 `deliverReqCh` 发挥作用了。它是一个 for 循环内部的局部变量，我们不得不结合 for 循环中的变量声明来看这个 channel 的处理代码：

```
func (d *Downloader) runStateSync(s *stateSync) *stateSync {
    ......
```

```
    for {
        var (
            deliverReq   *stateReq
            deliverReqCh chan *stateReq
        )
        if len(finished) > 0 {
            deliverReq = finished[0]
            deliverReqCh = s.deliver
        }
        select {
        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]
        }
    }
}
```

因为 `deliverReqCh` 是 for 循环的内部变量，因此循环一遍， `deliverReqCh` 都会被重新定义。如果有完成的请求（`finished` 的长度大于 0），则 `deliverReqCh` 的值为 `stateSync.deliver`，而另一个局部变量 `deliverReq` 的值为 `finished` 的第一个元素；否则 `deliverReqCh` 和 `deliverReq` 都为默认值 nil。接下来在 select/case 语句中，如果 `deliverReqCh` 和 `deliverReq` 两个变量都是有效值，那么 `deliverReq` 中的值就会发送给 `deliverReqCh`，也就是说 `finished` 集合中的第一个已完成的请求就会发送给 `stateSync.deliver`。而消息处理中则将 `finished` 中的第一个元素抹掉（因为这个元素已经通知给 `stateSync.deliver` 了）。

可以看到， `deliverReqCh` 这个 channel 实际上是为了将已经完成的请求发送给 `stateSync.deliver`。

至此，整个 `Downloader.runStateSync` 方法已经分析完了。整体来看，这个方法是连接接收返回的 state 数据和 `stateSync` 对象的中枢，它记录 `stateSync` 对象发出的请求，并将对方返回的数据传送给 `stateSync` 对象。

接下来我们简单了解下 `stateSync` 对象。这个对象的功能很简单，就是根据一个 Root 哈希，下载整个 state 数据，并将其写入 `Downloader.stateDB` 这个数据库中。前面我们说过，所谓 state 数据就是一棵 trie 树，下载 trie 树的方法就是先根据 Root 哈希同步根节点，然后通过 `trie.Sync` 的帮助从根节点中解析出它的子节点的哈希，再使用这些哈希同步相应的节点。依此不断进行，直到最底层的叶子结点也被同步成功，整棵树就算同步成功了。理解了这一下载过程， `stateSync` 对象中的代码就非常容易理解了。关于 `trie.Sync` 的介绍，可以参看这里。

总结

在这篇文章里，我们分析了 downloader 模块的主要逻辑。 downloader 模块对区块的下载分成两种模式，一种是 full 模式，另一种是 fast 模式。在 full 模式下只下载区块的 header 和 body； fast 模式下除了 header 和 body，还会下载所有区块的 receipt 数据和 pivot 区块的 state 数据，pivot 区块以后的区块，就和 full 模式一样了。

在下载前，eth 模块会选择一个 「best peer」 给 downloader 模块，所谓的 「best peer」 其实就是「TotalDifficulty」值最大的那个节点（关于 TotalDifficulty 概念的详细解释可以参看这篇文章）。在 downloader 中，首先通过 `findAncestor` 方法确定自己与 「best peer」 之间的共同组先，然后从这个共同组先开始，至对方的高度最高的那个区块，组成需要下载的区块高度区间进行下载。在下载的过程中， downloader 模块会先将这个区间中的 header 分组，每组的最后一个区块组成 「skeleton」，首先从 「best peer」 中下载这些 skeleton，然后选择其它节点对每一组 header 进行下载（就是所谓的「填充 skeleton」）。每当有 header 下载成功后，就会接着下载它所对应的 body 和 receipt（full 模式下只下载 body）。

在 fast 模式下，downloader 模块在下载区块的过程中，还会从高度较高的区块中选择一个区块作为 「pivot」 区块。 pivot 区块的特殊点在于，在它之前的区块是没有 state 数据的；而 pivot 区块的 state 对数是从其它节点中下载的；pivot 之后的区块以 pivot 区块的 state 为基础，在本地通过 body 计算出自己的 state 数据并存储起来（也就是说 pivot 之后的区块和 full 模式下的区块的处理方式是一样的）。

在整个下载过程中，需要 `Downloader` 对象和 `queue` 对象密切配合，共同完成区块下载的逻辑。本篇文章里我们只分析了 `Downloader` 对象的一些实现， `queue` 对象有的详细分析请参看这里。

本篇文章中只分析了 downloader 模块的一些关键部分，并没有分析到每一处细节。此外限于我的水平，文章中难免有错误。所以如果您有什么问题，欢迎留言或发邮件讨论。

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：区块同步-Protocol

下一篇 以太坊源码解析：downloader/queue

Comments

Comment (1)

Login

Sort by: **Date** Rating Last Activity

's avat Atlas Chiew · *35 weeks ago* 0

你好, 很赞的文章！

我对throttle产生疑问，也把问题提交到 https://ethereum.stackexchange.com/questions/1002...

长话多说，我发现源码里在遇到throttled后，好像没有地方un-throttle,这是为何?

Reply

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name	Email	Website (optional)
Displayed next to your comments.	*Not displayed publicly.*	*If you have a website, link to it here.*

Subscribe to None

Submit Comment

Contact me at:

Site powered by Jekyll & Github Pages. Theme designed by HyG.