第一步：与远程peer节点完成协议handshake，调取协议Run函数(即handle函数)。协议Run函数负责的功能有：

1. p.Handshake(td, head, genesis) 向对方peer节点发送本地区块链的 td难度累计值/head 头区块hash/genesis 创世区块hash，然后等待对端peer的回复(也包含这三个量)。其中必须保证genesis 创世区块hash是相同的。(最终获取对方区块链td难度累计值/head 头区块hash)
2. 完成双方节点节点交易的同步(调用pm.syncTransactions(p)等待向对方peer节点发送本地交易池中现有的尚未发布到区块中交易)
3. 循环调用pm.handleMsg(p)负责等待接收对方peer节点的各类消息

第二步：调用func (pm \*ProtocolManager) Start() 开始进行双方节点间的同步：

1. pm.txSub = pm.eventMux.Subscribe(core.TxPreEvent{}) 注册core.TxPreEvent类型事件
2. pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{}) 注册core.NewMinedBlockEvent类型事件
3. 调用go pm.txBroadcastLoop()不断检索本地新产生的TxPreEvent事件,将其广播到其他已连接的peer节点(测试时可以通过self.eventMux.Post(TxPreEvent{tx})产生一个交易事件)
4. 调用go pm.minedBroadcastLoop()不断检索本地新产生的NewMinedBlockEvent事件,将其广播给其他已连接的peer节点(可以通过self.eventMux.Post(NewMinedBlockEvent{block})产生一个挖矿事件) ---广播时分两次,一次广播区块实体,一次仅广播区块hash值
5. 调用go pm.syncer() , 进行与远程peer节点的区块同步
6. 调用go pm.txsyncLoop() , 进行与远程peer节点的尚未发布到区块的交易的同步

第三步：调用func (pm \*ProtocolManager) Stop() 结束双方节点的同步

1. pm.txSub.Unsubscribe() 注销TxPreEvent类型事件,停止了txBroadcastLoop()
2. pm.minedBlockSub.Unsubscribe() 注销NewMinedBlockEvent类型事件,停止了blockBroadcastLoop
3. close(pm.quitSync) 结束交易同步函数与区块同步函数
4. pm.wg.Wait() 等待本次处理结束
5. 结束整个同步环节

如何进行交易同步 --- go pm.txsyncLoop() 的完成流程

1. 在协议Run函数(即handle函数)的第二步中会调用pm.syncTransactions(p)从本地交易池中取出挂起(pending)的交易,将这些交易打包到txsync消息中,然后发送到pm.txsyncCh管道
2. 在协程中单独运行的txsyncLoop()会从pm.txsyncCh中不断获取这些txsync消息,然后调用p.SendTransactions(pack.txs)将其发送至远程peer节点(TxMsg消息)
3. 远程peer节点的pm.handleMsg(p)所在协程收到TxMsg消息,调用pm.txpool.AddTransactions(txs)将这些交易加入到自己本地的交易池中
4. 本次交易同步结束

如何进行区块同步 --- go pm.syncer() 的完成流程

1. 开启区块同步的条件有两个,满足其中任何一个即可开始同步：、
2. 完成了与新peer节点的协议handshake(pm.newPeerCh管道得到了信号)，同时确保有足够多的已连接peer可供选择(pm.peers.Len() ≥ minDesiredPeerCount)
3. forceSync计时器到达计时周期，即使没有足够的对等点，也强制同步

满足以上任何一种case,调用go pm.synchronise(pm.peers.BestPeer())执行区块同步，实参pm.peers.BestPeer()表示本地节点会选择已连接peer中具有最大td难度累计值(意味着具有最长区块链)的节点进行同步

1. 进入pm.synchronise(pm.peers.BestPeer())内部,首先将判断对方peer节点的td难度累计值是否大于本地区块链的td值(只有大于才能说明对方peer节点具有更长的区块链,有进行同步的必要)，判断完成后调用pm.downloader.Synchronise(peer.id, peer.Head(), peer.Td())，通过downloader从对方节点下载区块完成同步
2. 进入pm.downloader.Synchronise(peer.id, peer.Head(), peer.Td())内部,调用d.synchronise(id, head, td)进行同步。并根据同步的结果进行日志记录
3. 进入d.synchronise(id, head, td)内部，处理步骤如下：、
   1. 如果设置了模拟测试用同步方法d.synchroniseMock,则调用
   2. atomic.CompareAndSwapInt32(&d.synchronising, 0, 1)确保一次最多只有一个goroutine在进行区块同步(每次只与一个peer节点执行当前synchronise方法),否则退出本次同步
   3. 检测要进行同步的头区块的hash是否是被禁止的(位于d.banned队列---该队列的更新由banBlocks()方法负责),若禁止则退出本次同步
   4. 检查上一轮同步是否存在已下载但尚未加入到本地区块链中的残余区块(即检查blockCache缓存中是否还有剩余的区块)，若存在需结束本次同步
   5. 调用d.syncWithPeer(p, hash, td) 与该peer节点完成区块链同步
4. 进入d.syncWithPeer(p, hash, td)内部。首先产生一个同步开始事件d.mux.Post(StartEvent{})，根据协议handshake时获取的对方peer的子类协议eth版本号选择同步策略：

case eth60：

* 1. 调用d.fetchHashes60(p, hash) 向对端申请获取区块hash值,并申请获取相应区块
  2. 调用d.fetchBlocks60()等待对端回复的区块集合

case eth61:

* 1. 调用number, err := d.findAncestor(p) 获取本地链与p节点链的公共祖先区块的编号number
  2. 协程调用d.fetchHashes(p, td, number+1)
  3. 协程调用d.fetchBlocks(number + 1)

Eth60同步策略

1. d.fetchHashes60(p, hash)
2. d.fetchHasher60(p.hash)将调用active.getRelHashes(from)向目标peer节点详情获取以头区块hash开始的MaxHashFetch个区块hash(发送GetBlockHashesMsg消息),然后for循环等待对端peer的回复
3. 当对端peer节点的eth协议Run函数所在协程的handleMsg()方法收到此GetBlockHashesMsg消息时,就会调用pm.chainman.GetBlockHashesFromHash(request.Hash, request.Amount)从本地区块链中检索对应区块的hash值,然后调用p.SendBlockHashes(hashes)回复BlockHashesMsg消息
4. 每当本地节点的handleMsg()方法收到对方peer回复的BlockHashesMsg消息,就会调用pm.downloader.DeliverHashes(p.id, hashes)将这些收到的区块hash组装为hashPack消息,传输到d.hashCh管道
5. d.fetchHashes60(p, hash)的case hashPack := <-d.hashCh分支被触发，提取出这些区块hash值加入到q.hashQueue队列中，接着需要进一步检验对端peer是否真的存在对应的区块，通过调用active.getBlocks([]common.Hash{origin})向对端peer申请获取这些区块,通过收到的回复包确认对端peer是否真正拥有这些区块(GetBlocksMsg消息)
6. 当对端peer节点的handleMsg()方法收到此GetBlocksMsg消息，调用pm.chainman.GetBlock(hash)在本地链中检索获取,然后调用p.SendBlocks(blocks)完成发送(BlocksMsg)
7. 当本地节点的handleMsg()方法收到对方peer回复的BlockMsg消息，首先调用pm.fetcher.Filter(blocks)对区块进行过滤,满足过滤条件的区块会被直接插入到本地区块链上，同时会调用f.broadcastBlock(block, true)广播给其他peer节点(参数二为true,则分两次发送。一次为NewBlockMsg消息，另一次为NewBlockHashesMsg消息)，对于不满足过滤条件的区块调用pm.downloader.DeliverBlocks(p.id, blocks) 。将这些收到的区块组装为blockPack消息,传输到d.blockCh管道
8. d.fetchHashes60(p, hash)的blockPack := <-d.blockCh分支被触发,将完成检查的区块从d.checks中删除
9. d.fetchHashes60(p, hash)必须保证在crossTicker计时器计时周期内完成本次d.checks中所有的区块检查操作，否则认为此次区块同步出错(疑问：满足filter过滤条件的区块并不会被验证，这是否会导致出错？？？)
10. d.fetchHashes60(p, hash)还必须保证在timeout计时器计时周期内，收到对端peer对active.getRelHashes(from)操作的回复,否则会向新的信誉值最高的空闲节点申请获取这些区块hash

注：在case hashPack := <-d.hashCh分支中getHashes(head)操作会被执行两次,一次是在active.getBlocks([]common.Hash{origin})前，一次是在之后。在分支的最后还会根据active.getBlocks([]common.Hash{origin})的结果重新配置q.blockOffset区块链偏移量

1. d.fetchBlocks60()
2. case <-ticker.C分支：从q.hashQueue中取出待获取的区块hash值，组装成fetchRequest查询包，调用peer.Fetch(request)向空闲节点发送此查询包(底层调用p.getBlocks(hashes)用于向peer节点获取区块)
3. case blockPack := <-d.blockCh分支：接收到对端peer的BlockMsg消息，首先需要保证本次不是过时的区块check消息。将获取的结果区块注入到q.blockCache队列(并根据获取的结果重新评判对端peer的信誉值p.req)。最后调用d.process()将q.blockCache队列中的区块添加到本地区块链

Eth61同步策略

1. number, err := d.findAncestor(p)获取本地与p节点的公共祖先区块的编号
2. from := int64(head) - int64(MaxHashFetch) 计算获取在进行本次区块hash fetch前的上一次最新区块(head block)hash值
3. p.getAbsHashes(uint64(from), MaxHashFetch) 申请获取从区块编号from开始的MaxHashFetch个区块的hash值(发送GetBlockHashesFromNumberMsg消息)
4. case hashPack := <-d.hashCh分支, 得到了对端节点对GetBlockHashesFromNumberMsg消息的回应hash
5. 根据回应的结果中能否在本地上查询到相应区块分为两种处理方式：

情况1：在本地区块链上获取到了hash值对应的区块,此区块即为公共祖先区块,返回其编号(通过这种方式查询到的祖先区块是靠前的)

情况2：如果没能查询到共同祖先区块,需要在整个链上进行二分查找(通过此方法查询到的公共祖先区块在区块链上要靠后)

1. 调用d.fetchHashes(p, td, number+1)。从公共祖先区块的hash值开始,不断向对端peer节点申请获取新的区块hash(不断调用getHashes(from)),直到q.hashQueue到达容量上限
2. 调用d.fetchBlocks(number + 1)方法
   1. 定期检查q.hashQueue中是否还有等待发送的区块hash值,向空闲节点申请这些区块hash(case <-ticker.C分支引发的case <-update分支)—发送GetBlocksMsg消息
   2. 若q.hashQueue到达容量上限，向空闲节点申请这些区块hash(case cont := <-d.processCh分支引发的case <-update分支) —发送GetBlocksMsg消息
   3. case blockPack := <-d.blockCh分支,收到了对方peer回复的区块(BlocksMsg消息)，调用d.queue.Deliver(blockPack.peerId, blockPack.blocks)将获取的区块注入到q.blockCache队列(同时根据本次回复的情况调整该远程peer的信誉值p.rep)。最后调用d.process()将区块从q.blockCache导入到本地区块链