以太坊源码之P2P网络及节点发现机制

1 分布式网络介绍

以太坊底层分布式网络即P2P网络，使用了经典的Kademlia网络，简称kad。

1.1 Kad网介绍

Kademlia在2002年由美国纽约大学的PetarP.Manmounkov和DavidMazieres提出，是一种分布式散列表(DHT)技术，以异或运算为距离度量基础，已经在BitTorrent、BitComet、Emule等软件中得到应用。

1.2 Kad网络节点距离

以太坊网络节点距离计算方法：

Node1：节点1 NodeId Node2：节点2 NodeId

1.3 K桶

Kad的路由表是通过称为K桶的数据构造而成，K桶记录了节点NodeId，distance，endpoint，ip等信息。以太坊K桶按照与target节点距离进行排序，共256个K桶，每个K桶包含16个节点。

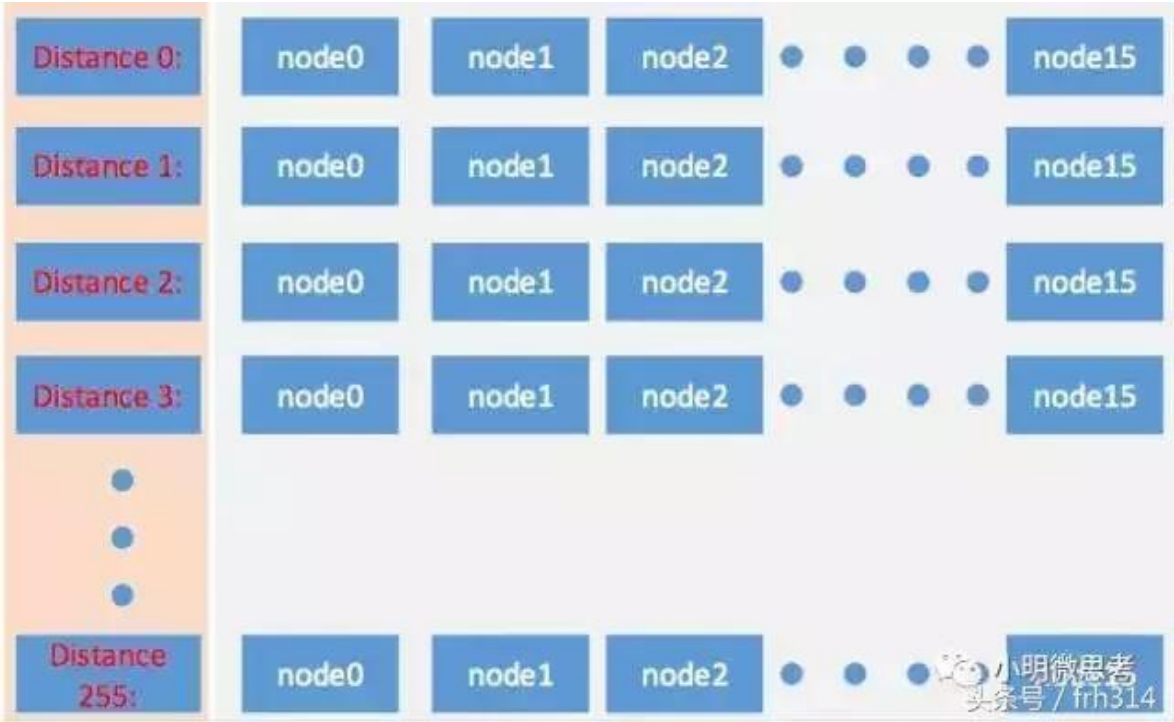

图1.1

1.4 Kad通信协议

以太坊Kad网络中节点间通信基于UDP，主要由以下几个命令构成，若两个节点间PING-PONG握手通过，则认为相应节点在线。

2 邻居节点

2.1 NodeTable类主要成员

C++版本以太坊源码中，NodeTable是以太坊 P2P网络的关键类，所有与邻居节点相关的数据和方法均由NodeTable类实现。

2.2 邻居节点发现方法

邻居节点是指加入到K桶，并通过PING-PONG握手的节点。

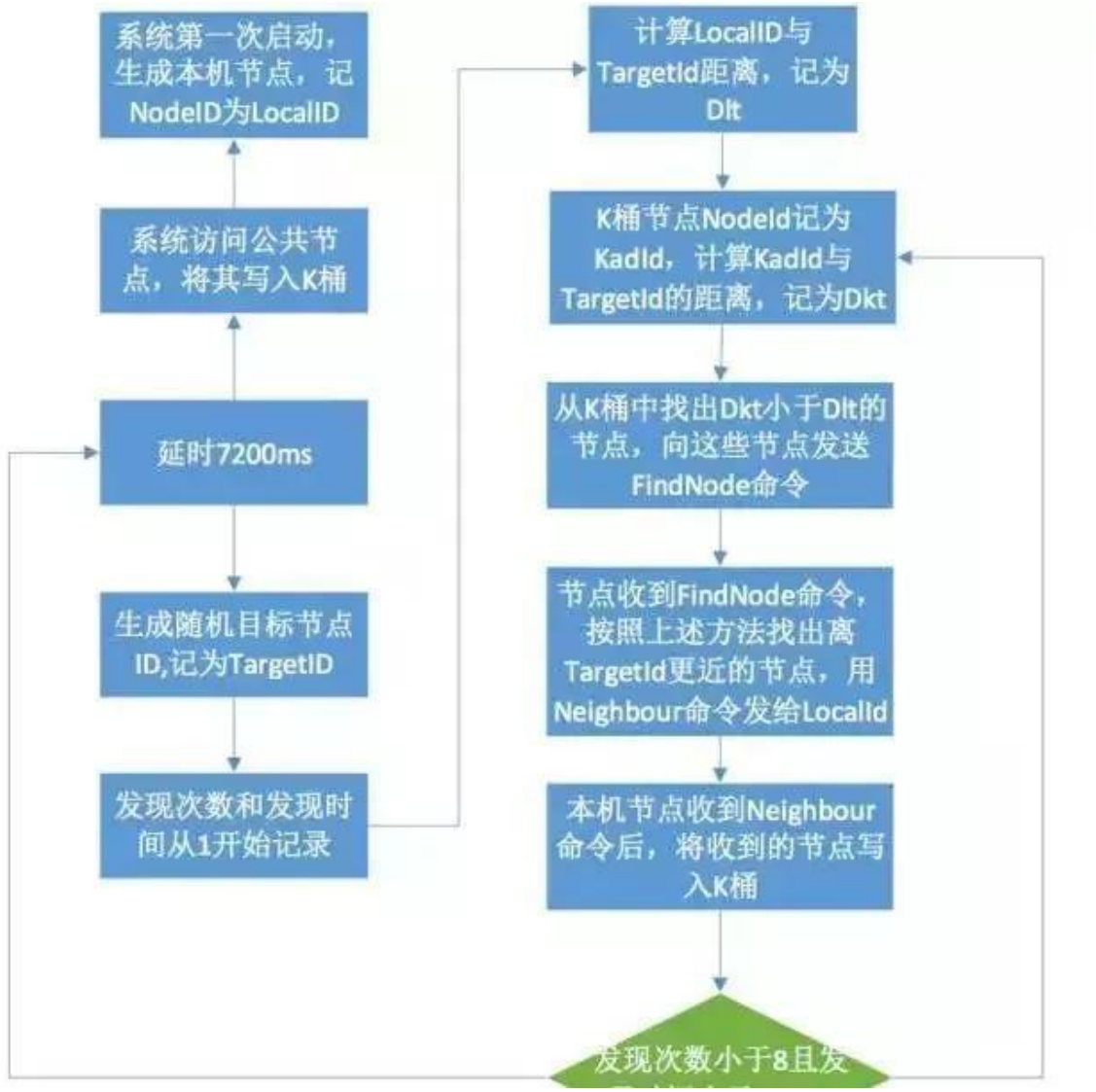

图2.1

邻居节点发现流程说明：

系统第一次启动随机生成本机节点NodeId，记为LocalId,生成后将固定不变，本地节点记为local-eth。 系统读取公共节点信息，ping-pong握手完成后，将其写入K桶。 系统每隔7200ms刷新一次K桶。 刷新K桶流程如下：

a. 随机生成目标节点Id，记为TargetId，从1开始记录发现次数和刷新时间。

b. 计算TargetId与LocalId的距离，记为Dlt

c. K桶中节点的NodeId记为KadId，计算KadId与TargetId的距离，记为Dkt

d. 找出K桶中Dlt大于Dkt的节点，记为k桶节点，向k桶节点发送

FindNODE命令，FindNODE命令包含TargetId
e. K桶节点收到FindNODE命令后，同样执行b-d的过程，将从K桶中找到的节点使用Neighbours命令发回给本机节点。
f. 本机节点收到Neighbours后，将收到的节点写入到K桶中。
g. 若搜索次数不超过8次，刷新时间不超过600ms，则返回到b步骤循环执行。
2.3 邻居节点网络拓扑及刷新机制。

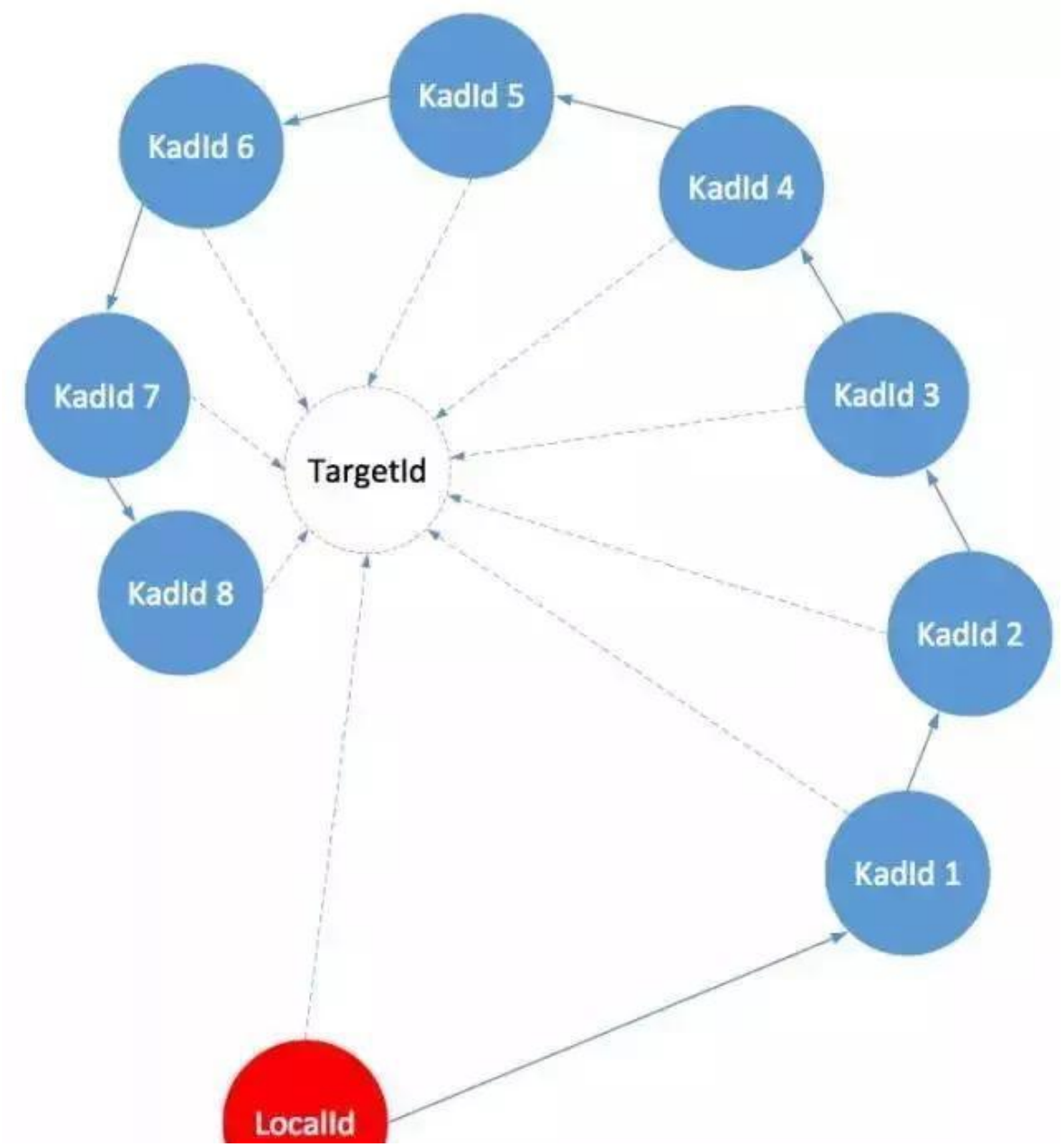

图2.2
1 TargetId为随机生成的虚拟节点ID。
2 以太坊Kad网络与传统Kad网络的区别：

以太坊节点在发现邻居节点的8次循环中，所查找的节点均在距离上向随机生成的TargetId收敛。 传统Kad网络发现节点时，在距离上向节点本身收敛。

以太坊p2p原理与实现

区块链技术的去中心依赖于底层组网技术，以太坊的底层实现了p2pServer，大约可以分为这样三层。

- 底层路由表。封装了kad路由，节点的数据结构以及计算记录，节点搜索，验证等功能。
- 中层peer抽象，message开放发送接口，server对外提供peer检测，初始化，事件订阅，peer状态查询，启动，停止等功能
- 以太坊最上层peer，peerset再封装，通过协议的Run函数，在中层启动peer时，获取peer，最终通过一个循环截取稳定peer，包装在peerset中使用。

底层路由表

这里简化问题仅讨论Node Discovery Protocol。 这一层维护了一个buckets桶，总共有17个桶，每个桶有16个节点和10个替换节点。 Node放入时先要计算hash和localNode的距离。再按距离选择一个桶放进去，取的时候逐个计算target和每个桶中对象的举例，详细参考closest函数，后面会贴出来。
距离公式满足：f(x,y)=256−8*n−map(x[n+1]^y[n+1]) 注：n为相同节点数量 map为一个负相关的映射关系。
简单来说就是相似越多，值越小。细节参考Node.go的logdist函

数。 这里需要了解算法Kademlia，

```
.
├── database.go          //封装node数据库相关操作
├── node.go              //节点数据结构
├── ntp.go               //同步时间
├── table.go             //路由表
├── udp.go               //网络相关操作
```

其中最重要的就是table对象，table公共方法有：

- newTable 实例创建
- Self local节点获取
- ReadRandomNodes 随机读取几个节点
- Close 关闭
- Resolve 在周边查找某个节点
- Lookup 查找某个节点的邻近节点

逐个来分析这些方法：

newTable

- 1：生成对象实例（获取数据库客户端，LocalNode etc）

```
    // If no node database was given, use an in-memory
one
    db, err := newNodeDB(nodeDBPath, Version, ourID)
    if err != nil {
        return nil, err
    }
    tab := &Table{
        net:        t,
        db:         db,
        self:       NewNode(ourID, ourAddr.IP,
uint16(ourAddr.Port), uint16(ourAddr.Port)),
        bonding:    make(map[NodeID]*bondproc),
        bondslots:  make(chan struct{},
maxBondingPingPongs),
        refreshReq: make(chan chan struct{}),
        initDone:   make(chan struct{}),
        closeReq:   make(chan struct{}),
        closed:     make(chan struct{}),
        rand:       mrand.New(mrand.NewSource(0)),
        ips:        netutil.DistinctNetSet{Subnet:
tableSubnet, Limit: tableIPLimit},
    }
```

- 2：载入引导节点，初始化k桶。

```
    if err := tab.setFallbackNodes(bootnodes); err != nil
{
        return nil, err
    }
    for i := 0; i < cap(tab.bondslots); i++ {
        tab.bondslots <- struct{}{}
    }
    for i := range tab.buckets {
        tab.buckets[i] = &bucket{
            ips: netutil.DistinctNetSet{Subnet:
bucketSubnet, Limit: bucketIPLimit},
        }
    }
```

- 3：将节点放入到桶里，生成一条协程用于刷新，验证节点。

```
    tab.seedRand()
    tab.loadSeedNodes(false)  //载入种子节点
    // Start the background expiration goroutine after
loading seeds so that the search for
    // seed nodes also considers older nodes that would
otherwise be removed by the
    // expiration.
    tab.db.ensureExpirer()
    go tab.loop()
```

载入种子节点

```
    func (tab *Table) loadSeedNodes(bond bool) {
        seeds := tab.db.querySeeds(seedCount, seedMaxAge)
        //数据库中的种子节点和引导节点合并
        seeds = append(seeds, tab.nursery...)
        if bond {
            seeds = tab.bondall(seeds)   //节点验证
        }
        for i := range seeds {
            seed := seeds[i]
            age := log.Lazy{Fn: func() interface{}
{ return time.Since(tab.db.bondTime(seed.ID)) }}
            log.Debug("Found seed node in database",
"id", seed.ID, "addr", seed.addr(), "age", age)
            tab.add(seed)                //节点入桶
        }
    }
```

节点入桶，同时也要检查ip等限制。

```
    func (tab *Table) add(new *Node) {
        tab.mutex.Lock()
```

```
        defer tab.mutex.Unlock()

        b := tab.bucket(new.sha)   //获取当前节点对应的桶
        if !tab.bumpOrAdd(b, new) {
            // Node is not in table. Add it to the
replacement list.
            tab.addReplacement(b, new)
        }
    }
```

桶的选择

```
    func (tab *Table) bucket(sha common.Hash) *bucket {
        d := logdist(tab.self.sha, sha)  //计算hash举例
        if d <= bucketMinDistance {
            //这里按算法来看，只要hash前三位相等就会到第一个
buckets
            return tab.buckets[0]
        }
        return tab.buckets[d-bucketMinDistance-1]
    }
```

Resolve

根据Node的Id查找Node，先在当前的桶里面查找，查找一遍之后没找到就在周边的节点里面搜索一遍再找。

```
    // Resolve searches for a specific node with the
given ID.
    // It returns nil if the node could not be found.
    func (tab *Table) Resolve(targetID NodeID) *Node {
        // If the node is present in the local table, no
        // network interaction is required.
        hash := crypto.Keccak256Hash(targetID[:])
        tab.mutex.Lock()
        //查找最近节点
        cl := tab.closest(hash, 1)
        tab.mutex.Unlock()
        if len(cl.entries) > 0 && cl.entries[0].ID ==
targetID {
            return cl.entries[0]
        }
        // Otherwise, do a network lookup.
        //不存在 搜索邻居节点
        result := tab.Lookup(targetID)
        for _, n := range result {
            if n.ID == targetID {
                return n
            }
```

```
        }
        return nil
    }
```

这里需要理解的函数是 closest，遍历所有桶的所有节点，查找最近的一个

```
    // closest returns the n nodes in the table that are
closest to the
    // given id. The caller must hold tab.mutex.
    func (tab *Table) closest(target common.Hash,
nresults int) *nodesByDistance {
        // This is a very wasteful way to find the
closest nodes but
        // obviously correct. I believe that tree-based
buckets would make
        // this easier to implement efficiently.
        close := &nodesByDistance{target: target}
        for _, b := range tab.buckets {
            for _, n := range b.entries {
                close.push(n, nresults)
            }
        }
        return close
    }

    func (h *nodesByDistance) push(n *Node, maxElems int)
{
        ix := sort.Search(len(h.entries), func(i int)
bool {
            return distcmp(h.target, h.entries[i].sha,
n.sha) > 0
        })
        if len(h.entries) < maxElems {
            h.entries = append(h.entries, n)
        }
        if ix == len(h.entries) {
            // farther away than all nodes we already
have.
            // if there was room for it, the node is now
the last element.
        } else {
            // slide existing entries down to make room
            // this will overwrite the entry we just
appended.
            //近的靠前边
            copy(h.entries[ix+1:], h.entries[ix:])
            h.entries[ix] = n
```

```
        }
    }
```

ReadRandomNodes

整体思路是先拷贝出来，再逐个桶的抽最上面的一个，剩下空桶移除，剩下的桶合并后，下一轮再抽桶的第一个节点，直到填满给定数据或者桶全部空掉。最后返回填到数组里面的数量。

```
    // ReadRandomNodes fills the given slice with random
nodes from the
    // table. It will not write the same node more than
once. The nodes in
    // the slice are copies and can be modified by the
caller.
    func (tab *Table) ReadRandomNodes(buf []*Node) (n
int) {
        if !tab.isInitDone() {
            return 0
        }
        tab.mutex.Lock()
        defer tab.mutex.Unlock()

        // Find all non-empty buckets and get a fresh
slice of their entries.
        var buckets [][]*Node
        //拷贝节点
        for _, b := range tab.buckets {
            if len(b.entries) > 0 {
                buckets = append(buckets, b.entries[:])
            }
        }
        if len(buckets) == 0 {
            return 0
        }
        // Shuffle the buckets.
        for i := len(buckets) - 1; i > 0; i-- {
            j := tab.rand.Intn(len(buckets))
            buckets[i], buckets[j] = buckets[j],
buckets[i]
        }
        // Move head of each bucket into buf, removing
buckets that become empty.
        var i, j int
        for ; i < len(buf); i, j = i+1,
(j+1)%len(buckets) {
            b := buckets[j]
```

```
            buf[i] = &(*b[0])  //取第一个节点
            buckets[j] = b[1:] //移除第一个
            if len(b) == 1 {
                //空桶移除
                buckets = append(buckets[:j],
buckets[j+1:]...)
            }
            if len(buckets) == 0 {
                break
            }
        }
        return i + 1
    }
```

Lookup

lookup会要求已知节点查找邻居节点，查找的邻居节点又递归的找它周边的节点

```
    for {
        // ask the alpha closest nodes that we haven't
asked yet
        for i := 0; i < len(result.entries) &&
pendingQueries < alpha; i++ {
            n := result.entries[i]
            if !asked[n.ID] {
                asked[n.ID] = true
                pendingQueries++
                go func() {
                    // Find potential neighbors to bond
with
                    r, err := tab.net.findnode(n.ID,
n.addr(), targetID)
                    if err != nil {
                        // Bump the failure counter to
detect and evacuate non-bonded entries
                        fails := tab.db.findFails(n.ID) +
1
                        tab.db.updateFindFails(n.ID,
fails)
                        log.Trace("Bumping findnode
failure counter", "id", n.ID, "failcount", fails)

                        if fails >= maxFindnodeFailures {
                            log.Trace("Too many findnode
failures, dropping", "id", n.ID, "failcount", fails)
                            tab.delete(n)
                        }
```

```
                    }
                    reply <- tab.bondall(r)
                }()
            }
        }
        if pendingQueries == 0 {
            // we have asked all closest nodes, stop the
search
            break
        }
        // wait for the next reply
        for _, n := range <-reply {    //此处会阻塞请求
            if n != nil && !seen[n.ID] {
                seen[n.ID] = true
                result.push(n, bucketSize)
            }
        }
        pendingQueries--
    }
```

桶的维护

桶初始化完成后会进入一个循环逻辑，其中通过三个timer控制调整周期。

- 验证timer 间隔 10s左右
- 刷新timer 间隔 30 min
- 持久化timer 间隔 30s

```
    revalidate     =
time.NewTimer(tab.nextRevalidateTime())
    refresh        = time.NewTicker(refreshInterval)
    copyNodes      = time.NewTicker(copyNodesInterval)
```

刷新逻辑:重新加载种子节点，查找周边节点，随机三个节点，并查找这三个节点的周围节点。

```
    func (tab *Table) doRefresh(done chan struct{}) {
        defer close(done)

        tab.loadSeedNodes(true)

        tab.lookup(tab.self.ID, false)

        for i := 0; i < 3; i++ {
            var target NodeID
            crand.Read(target[:])
            tab.lookup(target, false)
        }
```

```
    }
```

验证逻辑:验证每个桶的最末尾节点，如果该节点通过验证则放到队首（验证过程是本地节点向它发送ping请求，如果回应pong则通过）

```
    last, bi := tab.nodeToRevalidate()  //取最后一个节点
    if last == nil {
        // No non-empty bucket found.
        return
    }

    // Ping the selected node and wait for a pong.
    err := tab.ping(last.ID, last.addr())   //通信验证

    tab.mutex.Lock()
    defer tab.mutex.Unlock()
    b := tab.buckets[bi]
    if err == nil {
        // The node responded, move it to the front.
        log.Debug("Revalidated node", "b", bi, "id",
last.ID)
        b.bump(last)    //提到队首
        return
    }
```

Peer/Server

相关文件

```
.
├── dial.go           //封装一个任务生成处理结构以及三种任务结构中（此处命名不太精确）
├── message.go        //定义一些数据的读写接口，以及对外的Send/SendItem函数
├── peer.go           //封装了Peer 包括消息读取
├── rlpx.go           //内部的握手协议
├── server.go         //初始化，维护Peer网络，还有一些对外的接口
```

这一层会不断的从路由中提取节点，提取出来的节点要经过身份验证，协议检查之后加入到peer里面，紧接着如果没有人使用这个peer，这个peer就会被删除，再重新选择一些节点出来继续这个流程，peer再其中是随生随销，这样做是为了平均的使用所有的节点，而不是仅仅依赖于特定的几个节点。因而这里从Server开始入手分析整个流程

```
    Peers()                                  //peer对象
    PeerCount()                              //peer数量
    AddPeer(node *discover.Node)             //添加节点
    RemovePeer(node *discover.Node)          //删除节点
    SubscribeEvents(ch chan *PeerEvent) //订阅内部的事件（节点的增加，删除）
    //以上四个属于对外的接口，不影响内部逻辑
    Start()                                  //server开始工作
    SetupConn(fd net.Conn, flags connFlag, dialDest *discover.Node)  //启动一个连接，经过两次验证之后，如果通过则加入到peer之中。
```

Start初始化

Start做了三件事，生成路由表于建立底层网络。生成DialState用于驱动维护本地peer的更新与死亡，监听本地接口用于信息应答。这里主要分析peer的维护过程。函数是run函数。

```
    func (srv *Server) Start() (err error) {

        //**************初始化代码省略
        if !srv.NoDiscovery && srv.DiscoveryV5 {
            unhandled = make(chan discover.ReadPacket, 100)
            sconn = &sharedUDPConn{conn, unhandled}
        }

        // node table
        if !srv.NoDiscovery {
            //路由表生成
            cfg := discover.Config{
                PrivateKey:   srv.PrivateKey,
                AnnounceAddr: realaddr,
                NodeDBPath:   srv.NodeDatabase,
                NetRestrict:  srv.NetRestrict,
                Bootnodes:    srv.BootstrapNodes,
                Unhandled:    unhandled,
            }
            ntab, err := discover.ListenUDP(conn, cfg)
            if err != nil {
                return err
            }
            srv.ntab = ntab
        }
```

```
        if srv.DiscoveryV5 {
            //路由表生成
            var (
                ntab *discv5.Network
                err  error
            )
            if sconn != nil {
                ntab, err =
discv5.ListenUDP(srv.PrivateKey, sconn, realaddr, "",
srv.NetRestrict) //srv.NodeDatabase)
            } else {
                ntab, err =
discv5.ListenUDP(srv.PrivateKey, conn, realaddr, "",
srv.NetRestrict) //srv.NodeDatabase)
            }
            if err != nil {
                return err
            }
            if err :=
ntab.SetFallbackNodes(srv.BootstrapNodesV5); err != nil {
                return err
            }
            srv.DiscV5 = ntab
        }

        dynPeers := srv.maxDialedConns()
        //newDialState 对象生成，这个对象包含Peer的实际维护代码
        dialer := newDialState(srv.StaticNodes,
srv.BootstrapNodes, srv.ntab, dynPeers, srv.NetRestrict)

        // handshake  协议加载
        srv.ourHandshake = &protoHandshake{Version:
baseProtocolVersion, Name: srv.Name, ID:
discover.PubkeyID(&srv.PrivateKey.PublicKey)}
        for _, p := range srv.Protocols {
            srv.ourHandshake.Caps =
append(srv.ourHandshake.Caps, p.cap())
        }
        // listen/dial
        //监听本地端口
        if srv.ListenAddr != "" {
            if err := srv.startListening(); err != nil {
                return err
            }
        }
        if srv.NoDial && srv.ListenAddr == "" {
```

```
            srv.log.Warn("P2P server will be useless,
neither dialing nor listening")
        }

        srv.loopWG.Add(1)
        //重要的一句，开个协程，在其中做peer的维护
        go srv.run(dialer)
        srv.running = true
        return nil
    }
```

run 开始peer的生成

该函数中定义了两个队列

```
    runningTasks []task //正在执行的任务
    queuedTasks  []task //尚未执行的任务
```

定义了三个匿名函数

```
    //从正在执行任务中删除任务
    delTask := func(t task) {
        for i := range runningTasks {
            if runningTasks[i] == t {
                runningTasks = append(runningTasks[:i],
runningTasks[i+1:]...)
                break
            }
        }
    }
    //开始一批任务
    startTasks := func(ts []task) (rest []task) {
        i := 0
        for ; len(runningTasks) < maxActiveDialTasks &&
i < len(ts); i++ {
            t := ts[i]
            srv.log.Trace("New dial task", "task", t)
            go func() {
                t.Do(srv); taskdone <- t
            }()
            runningTasks = append(runningTasks, t)
        }
        return ts[i:]
    }
    //启动开始一批任务再调用dialstate的newTasks函数生成一批任
务，加载到任务队列里面
    scheduleTasks := func() {
        // Start from queue first.
```

```
            queuedTasks = append(queuedTasks[:0],
startTasks(queuedTasks)...)
            // Query dialer for new tasks and start as many
as possible now.
            if len(runningTasks) < maxActiveDialTasks {
                nt := dialstate.newTasks(len(runningTasks)
+len(queuedTasks), peers, time.Now())
                queuedTasks = append(queuedTasks,
startTasks(nt)...)
            }
        }
```

定义了一个循环，分不同的chanel执行对应的逻辑

```
        for {
            //调度开始找生成任务
            scheduleTasks()

            select {
            case <-srv.quit://退出
                break running
            case n := <-srv.addstatic:
                //增加一个节点  该节点最终会生成一个dialTask
                //并在newTasks的时候加入到读列
                srv.log.Debug("Adding static node", "node",
n)
                dialstate.addStatic(n)
            case n := <-srv.removestatic:
                //直接删除该节点 节点不再参与维护，很快就会死掉了
                dialstate.removeStatic(n)
                if p, ok := peers[n.ID]; ok {
                    p.Disconnect(DiscRequested)
                }
            case op := <-srv.peerOp:
                //  Peers 和 PeerCount 两个外部接口，只是读取
peer信息
                op(peers)
                srv.peerOpDone <- struct{}{}
            case t := <-taskdone:
                //task完成后会根据不同的任务类型进行相应的处理
                srv.log.Trace("Dial task done", "task", t)
                dialstate.taskDone(t, time.Now())
                delTask(t)
            case c := <-srv.posthandshake:
                //身份验证通过
                if trusted[c.id] {
                    // Ensure that the trusted flag is set
```

```
before checking against MaxPeers.
                    c.flags |= trustedConn
                }
                select {
                case c.cont <-
srv.encHandshakeChecks(peers, inboundCount, c):
                case <-srv.quit:
                    break running
                }
            case c := <-srv.addpeer:
                //身份协议验证通过 加入队列
                err := srv.protoHandshakeChecks(peers,
inboundCount, c)
                if err == nil {
                    // The handshakes are done and it
passed all checks.
                    p := newPeer(c, srv.Protocols)
                    // If message events are enabled, pass
the peerFeed
                    // to the peer
                    if srv.EnableMsgEvents {
                        p.events = &srv.peerFeed
                    }
                    name := truncateName(c.name)
                    srv.log.Debug("Adding p2p peer",
"name", name, "addr", c.fd.RemoteAddr(), "peers",
len(peers)+1)
                    go srv.runPeer(p)   //触发事件 此处是最上
层截取peer的位置，如果此物没有外部影响，那么这个peer很快就被销毁了
                    peerAdd++
                    fmt.Printf("--count %d--- add %d-- del
%d--\n",len(peers),peerAdd,peerDel)

                    peers[c.id] = p
                    if p.Inbound() {
                        inboundCount++
                    }
                }
                // The dialer logic relies on the
assumption that
                // dial tasks complete after the peer has
been added or
                // discarded. Unblock the task last.
                select {
                case c.cont <- err:
                case <-srv.quit:
```

```
                    break running
                }
            case pd := <-srv.delpeer:
                //移除peer
                d := common.PrettyDuration(mclock.Now() -
pd.created)
                pd.log.Debug("Removing p2p peer",
"duration", d, "peers", len(peers)-1, "req",
pd.requested, "err", pd.err)
                delete(peers, pd.ID())
                peerDel++
                fmt.Printf("--count %d--- add %d-- del %d--
\n",len(peers),peerAdd,peerDel)
                if pd.Inbound() {
                    inboundCount--
                }
            }
        }
```

记住上面的代码，再来逐个的看：

scheduleTasks

scheduleTasks调度生成任务，生成的任务中有一种dialTask的任务，该任务结构如下

```
    type dialTask struct {
        flags        connFlag
        dest         *discover.Node
        lastResolved time.Time
        resolveDelay time.Duration
    }

    func (t *dialTask) Do(srv *Server) {
        if t.dest.Incomplete() {
            if !t.resolve(srv) {
                return
            }
        }
        err := t.dial(srv, t.dest)  //此处会调用到setupConn
函数
        if err != nil {
            log.Trace("Dial error", "task", t, "err",
err)
            // Try resolving the ID of static nodes if
dialing failed.
            if _, ok := err.(*dialError); ok &&
t.flags&staticDialedConn != 0 {
```

```
                if t.resolve(srv) {
                    t.dial(srv, t.dest)
                }
            }
        }
    }
```

dial最终回调用到setupConn函数，函数只保留重点的几句，篇幅有点长了

```
    func (srv *Server) setupConn(c *conn, flags connFlag,
dialDest *discover.Node) error {

        //身份验证码 获取设备，标识等信息
        if c.id, err = c.doEncHandshake(srv.PrivateKey,
dialDest); err !=
        //此处会往chanel中添加连接对象，最终触发循环中的
posthandshake分支
        err = srv.checkpoint(c, srv.posthandshake)
        //协议验证
        phs, err := c.doProtoHandshake(srv.ourHandshake)
        c.caps, c.name = phs.Caps, phs.Name
        //此处会往chanel中添加连接对象 最终触发循环中的addpeer分
支
        err = srv.checkpoint(c, srv.addpeer)
    }
```

posthandshake 分支仅仅做了验证，addpeer做的事情就比较多了，重要的就是执行runPeer函数

```
    func (srv *Server) runPeer(p *Peer) {
        // 广播 peer add
        srv.peerFeed.Send(&PeerEvent{
            Type: PeerEventTypeAdd,
            Peer: p.ID(),
        })

        // run the protocol
        remoteRequested, err := p.run() //

        // 广播 peer drop
        srv.peerFeed.Send(&PeerEvent{
            Type:  PeerEventTypeDrop,
            Peer:  p.ID(),
            Error: err.Error(),
        })
        //移除peer
```

```
        srv.delpeer <- peerDrop{p, err, remoteRequested}
    }

    func (p *Peer) run() (remoteRequested bool, err
error) {
        //*************
        writeStart <- struct{}{}
        p.startProtocols(writeStart, writeErr)
        //*************
        //这一句阻塞性确保了peer的存活
        p.wg.Wait()
    }

    func (p *Peer) startProtocols(writeStart <-chan
struct{}, writeErr chan<- error) {
        p.wg.Add(len(p.running))
        for _, proto := range p.running {
            proto := proto
            proto.closed = p.closed
            proto.wstart = writeStart
            proto.werr = writeErr
            var rw MsgReadWriter = proto
            if p.events != nil {
                rw = newMsgEventer(rw, p.events, p.ID(),
proto.Name)
            }
            p.log.Trace(fmt.Sprintf("Starting protocol
%s/%d", proto.Name, proto.Version))
            go func() {
                //其他的都是为这一句做准备的，在以太坊中p2p就是靠
这一句对上层暴露peer对象
                err := proto.Run(p, rw)
                if err == nil {
                    p.log.Trace(fmt.Sprintf("Protocol %s/
%d returned", proto.Name, proto.Version))
                    err = errProtocolReturned
                } else if err != io.EOF {
                    p.log.Trace(fmt.Sprintf("Protocol %s/
%d failed", proto.Name, proto.Version), "err", err)
                }
                p.protoErr <- err
                p.wg.Done()
            }()
        }
    }
```

这样就可以可理出一条思路 scheduleTasks执行生成dialTask任

务 dialTask任务执行过程中逐个填充posthandshake，addPeer这两个chanel。 addPeer执行时对上层暴露了Peer对象，完成后填充了delpeer，最后删除了Peer。

任务的生成

具体看代码中的注释

```
    func (s *dialstate) newTasks(nRunning int, peers
map[discover.NodeID]*Peer, now time.Time) []task {
        if s.start.IsZero() {
            s.start = now
        }

        var newtasks []task
        //这里声明了一个添加任务的函数
        addDial := func(flag connFlag, n *discover.Node)
bool {
            if err := s.checkDial(n, peers); err != nil {
                log.Trace("Skipping dial candidate",
"id", n.ID, "addr", &net.TCPAddr{IP: n.IP, Port:
int(n.TCP)}, "err", err)
                return false
            }
            s.dialing[n.ID] = flag  //排除掉已经再测试的
            newtasks = append(newtasks, &dialTask{flags:
flag, dest: n})
            return true
        }

        // Compute number of dynamic dials necessary at
this point.
        needDynDials := s.maxDynDials     //当前系统中最大连
接数目
        for _, p := range peers {        //扣除已建立链接的
peer
            if p.rw.is(dynDialedConn) {
                needDynDials--
            }
        }
        for _, flag := range s.dialing {  //扣除已建立链接的
peer
            if flag&dynDialedConn != 0 {
                needDynDials--
            }
        }
```

```
        //外部命令添加的节点 这种节点不占用needDynDials数目,
        //是为了保证手动加的节点能够起效
        for id, t := range s.static {
            err := s.checkDial(t.dest, peers)
            switch err {
            case errNotWhitelisted, errSelf:
                log.Warn("Removing static dial
candidate", "id", t.dest.ID, "addr", &net.TCPAddr{IP:
t.dest.IP, Port: int(t.dest.TCP)}, "err", err)
                delete(s.static, t.dest.ID)
            case nil:
                s.dialing[id] = t.flags
                newtasks = append(newtasks, t)
            }
        }
        // If we don't have any peers whatsoever, try to
dial a random bootnode. This
        // scenario is useful for the testnet (and
private networks) where the discovery
        // table might be full of mostly bad peers,
making it hard to find good ones.
        if len(peers) == 0 && len(s.bootnodes) > 0 &&
needDynDials > 0 &&
        //检查引导节点  因为引导节点比搜索到的节点更大概率靠谱 因
而比较靠前
        now.Sub(s.start) > fallbackInterval {
            bootnode := s.bootnodes[0]
            s.bootnodes = append(s.bootnodes[:0],
s.bootnodes[1:]...)
            s.bootnodes = append(s.bootnodes, bootnode)

            if addDial(dynDialedConn, bootnode) {
                needDynDials--
            }
        }
        //随机的从路由中抽取最大节点的二分之一
        randomCandidates := needDynDials / 2
        if randomCandidates > 0 {
            n := s.ntab.ReadRandomNodes(s.randomNodes)
            for i := 0; i < randomCandidates && i < n; i+
+ {
                if addDial(dynDialedConn,
s.randomNodes[i]) {
                    needDynDials--
                }
```

```
            }
        }
        // 从lookupbuf中抽取
        i := 0
        for ; i < len(s.lookupBuf) && needDynDials > 0;
i++ {
            if addDial(dynDialedConn, s.lookupBuf[i]) {
                needDynDials--
            }
        }
        s.lookupBuf = s.lookupBuf[:copy(s.lookupBuf,
s.lookupBuf[i:])]
        // 如果还是不够，路由再去搜索节点
        if len(s.lookupBuf) < needDynDials && !
s.lookupRunning {
            s.lookupRunning = true
            newtasks = append(newtasks, &discoverTask{})
        }

        // wait
        if nRunning == 0 && len(newtasks) == 0 &&
s.hist.Len() > 0 {
            t :=
&waitExpireTask{s.hist.min().exp.Sub(now)}
            newtasks = append(newtasks, t)
        }
        return newtasks
    }
```

消息发送

另一个是message中的Send，SendItem函数 实现了MsgWriter的对象都可以调用这个函数写入，觉得这里没什么必要，完全可以封装到peer里面去，不过它上层做广播的时候确实是调用的这两个函数。

```
    func Send(w MsgWriter, msgcode uint64, data
interface{}) error {
        size, r, err := rlp.EncodeToReader(data)
        if err != nil {
            return err
        }
        return w.WriteMsg(Msg{Code: msgcode, Size:
uint32(size), Payload: r})
    }

    func SendItems(w MsgWriter, msgcode uint64,
```

```
elems ...interface{}) error {
        return Send(w, msgcode, elems)
    }
```

以太坊上层调用

Peer/PeerSet

文件：go-ethereum/eth/peer.go
定义了两个struct，Peer和PeerSet。Peer封装了底层的p2p.Peer,集成了一些和业务相关的方法，比如SendTransactions，SendNewBlock等。PeerSet是Peer的集合

```
    type peer struct {
        id string

        *p2p.Peer
        rw p2p.MsgReadWriter

        version  int         // Protocol version negotiated
        forkDrop *time.Timer // Timed connection dropper if forks aren't validated in time

        head common.Hash
        td   *big.Int
        lock sync.RWMutex

        knownTxs    *set.Set // Set of transaction hashes known to be known by this peer
        knownBlocks *set.Set // Set of block hashes known to be known by this peer
    }

    type peerSet struct {
        peers  map[string]*peer
        lock   sync.RWMutex
        closed bool
    }
```

Peer注册/注销

文件：go-ethereum/eth/handler.go manager.handle在检查了peer后会把这个peer注册到peerset中，表示此peer可用，发生错误后peerset注销该peer，返回错误，最后再Server中销毁。

```
    manager.SubProtocols = make([]p2p.Protocol, 0, len(ProtocolVersions))
```

```
    for i, version := range ProtocolVersions {
        // Skip protocol version if incompatible with
the mode of operation
        if mode == downloader.FastSync && version <
eth63 {
            continue
        }
        // Compatible; initialise the sub-protocol
        version := version // Closure for the run
        manager.SubProtocols =
append(manager.SubProtocols, p2p.Protocol{
            Name:    ProtocolName,
            Version: version,
            Length:  ProtocolLengths[i],
            Run: func(p *p2p.Peer, rw
p2p.MsgReadWriter) error {
                peer := manager.newPeer(int(version),
p, rw)
                select {
                case manager.newPeerCh <- peer:
                    manager.wg.Add(1)
                    defer manager.wg.Done()
                  //此处如果顺利会进入for循环 如果失败返回错误
我会销毁掉这个peer
                    return manager.handle(peer)
                case <-manager.quitSync:
                    return p2p.DiscQuitting
                }
            },
            NodeInfo: func() interface{} {
                return manager.NodeInfo()
            },
            PeerInfo: func(id discover.NodeID)
interface{} {
                if p :=
manager.peers.Peer(fmt.Sprintf("%x", id[:8])); p != nil {
                    return p.Info()
                }
                return nil
            },
        })
    }
```