ethereum-p2p(2)节点发现机制代码分析(v1.8.24)

1、引导

此部分主要分析以太坊节点发现机制源码，以太坊节点发现部分主要借助了一种分布式哈希表的结构(DHT)，Kademlia协议是以太坊节点发现机制的基础，它是一种以节点id异或后的结果作为两节点逻辑距离的一种协议，详细介绍在另一部分。下面主要介绍一下以太坊对此协议的具体实现。

先介绍下主要的数据结构：

```
// udp主要由四种互相交互的包，如下
// RPC packet types
const (
    pingPacket = iota + 1 // zero is 'reserved'
    pongPacket
    findnodePacket
    neighborsPacket
)

// RPC request structures
type (
    // ping包的定义
    ping struct {
        senderKey *ecdsa.PublicKey // filled in by preverify

        Version    uint
        From, To   rpcEndpoint
        Expiration uint64
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }
    // pong包的定义
    // pong is the reply to ping.
    pong struct {
        // This field should mirror the UDP envelope address
        // of the ping packet, which provides a way to discover the
        // the external address (after NAT).
        To rpcEndpoint
        //包含ping包的hash值   方便做验证
        ReplyTok   []byte // This contains the hash of the ping packet.
        Expiration uint64 // Absolute timestamp at which the packet becomes invalid.
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }
    // findnode包定义，就是向邻居查找节点的包
    // findnode is a query for nodes close to the given target.
    findnode struct {
        Target     encPubkey
        Expiration uint64
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }
    // 这个包是findnode的回复包
```

```
    // reply to findnode
    neighbors struct {
        Nodes      []rpcNode
        Expiration uint64
        // Ignore additional fields (for forward compatibility).
        Rest []rlp.RawValue `rlp:"tail"`
    }

    rpcNode struct {
        IP  net.IP // len 4 for IPv4 or 16 for IPv6
        UDP uint16 // for discovery protocol
        TCP uint16 // for RLPx protocol
        ID  encPubkey
    }
    ....................
)
```

下面主要涉及两个代码源文件：udp.go和table.go

1.1 go-ethereum/p2p/discover/udp.go

```
// 代码分析从server.Start()中开始
------------------------------------------------------------------------
    f err := srv.setupDiscovery(); err != nil {
        return err
    }
------------------------------------------------------------------------
func (srv *Server) setupDiscovery() error {
    // 如果配置参数指定不开启节点发现机制  则直接返回
    if srv.NoDiscovery && !srv.DiscoveryV5 {
        return nil
    }
    // 解析地址
    addr, err := net.ResolveUDPAddr("udp", srv.ListenAddr)
    if err != nil {
        return err
    }
    //借用net包开启实际的udp监听
    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        return err
    }
    realaddr := conn.LocalAddr().(*net.UDPAddr)
    srv.log.Debug("UDP listener up", "addr", realaddr)
    if srv.NAT != nil {
        if !realaddr.IP.IsLoopback() {
            go nat.Map(srv.NAT, srv.quit, "udp", realaddr.Port, realaddr.Port, "ethereum
discovery")
        }
    }
    srv.localnode.SetFallbackUDP(realaddr.Port)
    // 发现协议v4 一般全节点使用这个
    // Discovery V4
    var unhandled chan discover.ReadPacket
    var sconn *sharedUDPConn
    if !srv.NoDiscovery {
```

```
        if srv.DiscoveryV5 {
            unhandled = make(chan discover.ReadPacket, 100)
            sconn = &sharedUDPConn{conn, unhandled}
        }
        cfg := discover.Config{
            PrivateKey:  srv.PrivateKey,
            NetRestrict: srv.NetRestrict,
            Bootnodes:   srv.BootstrapNodes,
            Unhandled:   unhandled,
        }
        // 此处实际开启监听
        ntab, err := discover.ListenUDP(conn, srv.localnode, cfg)
        if err != nil {
            return err
        }
        srv.ntab = ntab
    }
    // 以下部分忽略
.............................
}
// ListenUDP returns a new table that listens for UDP packets on laddr.
func ListenUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, error) {
    // 调用newUDP方法开启udp服务
    tab, _, err := newUDP(c, ln, cfg)
    if err != nil {
        return nil, err
    }
    return tab, nil
}
unc newUDP(c conn, ln *enode.LocalNode, cfg Config) (*Table, *udp, error) {
    udp := &udp{
        conn:            c,
        priv:            cfg.PrivateKey,
        netrestrict:     cfg.NetRestrict,
        localNode:       ln,
        db:              ln.Database(),
        closing:         make(chan struct{}),
        gotreply:        make(chan reply),
        addReplyMatcher: make(chan *replyMatcher),
    }
    //这里的newTabl方法主要去实现了kad协议，这里暂时不分析，放在后面第二部分，这里主要先分析udp部分做的事
情
    tab, err := newTable(udp, ln.Database(), cfg.Bootnodes)
    if err != nil {
        return nil, nil, err
    }
    udp.tab = tab

    udp.wg.Add(2)
     // 下面是udp部分的主要两个循环，使用两个协程去做
    //-----------------------------------------------------------
    //  udp.loop()主要用来丢掉过期的包 和从各种数据通道读取相关数据做处理
    //  udp.readLoop()主要读取udp端口的数据，将数据解码为相应的四个包 并分发到相应的处理器进行处理 相应处
理器再把结果推给数据通道
    //
    //-----------------------------------------------------------
    //对各种通道和包是否过期进行处理
```

```
    go udp.loop()
    //  读取udp的数据包
    go udp.readLoop(cfg.Unhandled)
    return udp.tab, udp, nil
}
```

下面主要分析udp的两个循环结构loop()和readLoop()

```
// loop runs in its own goroutine. it keeps track of
// the refresh timer and the pending reply queue.
func (t *udp) loop() {
    defer t.wg.Done()

    var (
        plist        = list.New()
        timeout      = time.NewTimer(0)
        nextTimeout  *replyMatcher // head of plist when timeout was last reset
        // 持续过期包计数器
        contTimeouts = 0            // number of continuous timeouts to do NTP checks
        ntpWarnTime  = time.Unix(0, 0)
    )
    <-timeout.C // ignore first timeout
    defer timeout.Stop()
    // 定义重设过期函数
    resetTimeout := func() {
        // 若list头为空或者下一个过期的包是plist里面的头，那么直接返回
        if plist.Front() == nil || nextTimeout == plist.Front().Value {
            return
        }
        // Start the timer so it fires when the next pending reply has expired.
        // 记录当前时间
        now := time.Now()
         // 从头遍历plist
        for el := plist.Front(); el != nil; el = el.Next() {
            nextTimeout = el.Value.(*replyMatcher)
             // 如果一个包的deadline减去当前时间小于2倍respTimeout （相当于1s）
            if dist := nextTimeout.deadline.Sub(now); dist < 2*respTimeout {
                 //那么重设倒计时计时器的时间为dist时间后到期
                timeout.Reset(dist)
                return
            }
             // 要不然发出一个时钟错误的消息，并且将当前的包移除
             // 因为此时的deadline时间太长
            // Remove pending replies whose deadline is too far in the
            // future. These can occur if the system clock jumped
            // backwards after the deadline was assigned.
            nextTimeout.errc <- errClockWarp
            plist.Remove(el)
        }
        nextTimeout = nil
        timeout.Stop()
    }
    // 此处是udp的主循环 每次循环进来先调用一次resetTimeout()函数
    for {
        resetTimeout()
        // 下面主要接收各种通道发来的消息并进行处理
```

```
        select {
         // 若收到关闭消息，那么遍历plist，逐个发送errClosed消息通知关闭
        case <-t.closing:
            for el := plist.Front(); el != nil; el = el.Next() {
                el.Value.(*replyMatcher).errc <- errClosed
            }
            return
 // ------------------------以下两个通道比较重要--------------------------------
        // 将一个待匹配的匹配器添加到plist里面等待消息匹配
        case p := <-t.addReplyMatcher:
            p.deadline = time.Now().Add(respTimeout)
            plist.PushBack(p)
        //将收到的消息回复推送到这里
        case r := <-t.gotreply:
            var matched bool // whether any replyMatcher considered the reply acceptable.
            // 从头遍历plist 看是否和远方的回复包匹配，如匹配则调用相应的callback并传入回复的数据
            for el := plist.Front(); el != nil; el = el.Next() {
                p := el.Value.(*replyMatcher)
                  // 若包的发送方，包类型，ip地址相等，那么认为有一个replyMatcher得到了回复，调用其
callback方法
                if p.from == r.from && p.ptype == r.ptype && p.ip.Equal(r.ip) {
                    ok, requestDone := p.callback(r.data)
                    matched = matched || ok
                    // Remove the matcher if callback indicates that all replies have been
received.
                    if requestDone {
                        p.errc <- nil
                        plist.Remove(el)
                    }
                    // Reset the continuous timeout counter (time drift detection)
                    contTimeouts = 0
                }
            }
            r.matched <- matched
 // ----------------------------------------------------------------------------
        // 定时进行过期检查
        case now := <-timeout.C:
            nextTimeout = nil
             // 遍历plist如果一个包的deadline小于等于当前时间，那么将其移除
            // Notify and remove callbacks whose deadline is in the past.
            for el := plist.Front(); el != nil; el = el.Next() {
                p := el.Value.(*replyMatcher)
                if now.After(p.deadline) || now.Equal(p.deadline) {
                    p.errc <- errTimeout
                    plist.Remove(el)
                    contTimeouts++
                }
            }
            // 若等待回复的列表出现连续32个过期的包，开始使用ntp进行时钟同步
            // If we've accumulated too many timeouts, do an NTP time sync check
            if contTimeouts > ntpFailureThreshold {
                if time.Since(ntpWarnTime) >= ntpWarningCooldown {
                    ntpWarnTime = time.Now()
                    go checkClockDrift()
                }
                contTimeouts = 0
            }
```

```
        }
    }
}
```

上面是udp的loop循环，主要做了如下几件事：

- 首先处理超时信息
- 接收各种通道来的消息，对消息进行相应处理

下面开看第二个循环readLoop()的分析

```
// readLoop runs in its own goroutine. it handles incoming UDP packets.
func (t *udp) readLoop(unhandled chan<- ReadPacket) {
    defer t.wg.Done()
    if unhandled != nil {
        defer close(unhandled)
    }
    // discover 包最大不超过1280字节超过以后就非法
    // Discovery packets are defined to be no larger than 1280 bytes.
    // Packets larger than this size will be cut at the end and treated
    // as invalid because their hash won't match.
    buf := make([]byte, 1280)
    // for无限循环从udp读取数据
    for {
        //
        nbytes, from, err := t.conn.ReadFromUDP(buf)
        if netutil.IsTemporaryError(err) {
            // Ignore temporary read errors.
            log.Debug("Temporary UDP read error", "err", err)
            continue
        } else if err != nil {
            // Shut down the loop for permament errors.
            log.Debug("UDP read error", "err", err)
            return
        }
        // 将读取的数据交给handlePacket处理
        if t.handlePacket(from, buf[:nbytes]) != nil && unhandled != nil {
            select {
            case unhandled <- ReadPacket{buf[:nbytes], from}:
            default:
            }
        }
    }
}
// 这个函数主要用来处理从udp端口收到的数据
func (t *udp) handlePacket(from *net.UDPAddr, buf []byte) error {
    // 先交给decodePacket函数进行捷报，将包解包为四种包的一种
    packet, fromKey, hash, err := decodePacket(buf)
    if err != nil {
        log.Debug("Bad discv4 packet", "addr", from, "err", err)
        return err
    }
    fromID := fromKey.id()
    if err == nil {
        // 在此调用了一下对应包的预验证
        err = packet.preverify(t, from, fromID, fromKey)
```

```
    }
    log.Trace("<< "+packet.name(), "id", fromID, "addr", from, "err", err)
    if err == nil {
        // 无误的时候调用相应包的handle处理相应消息
        packet.handle(t, from, fromID, hash)
    }
    return err
}
// decodePacket方法
func decodePacket(buf []byte) (packet, encPubkey, []byte, error) {
    if len(buf) < headSize+1 {
        return nil, encPubkey{}, nil, errPacketTooSmall
    }
    // 将数据包的数据分开
    hash, sig, sigdata := buf[:macSize], buf[macSize:headSize], buf[headSize:]
    shouldhash := crypto.Keccak256(buf[macSize:])
    if !bytes.Equal(hash, shouldhash) {
        return nil, encPubkey{}, nil, errBadHash
    }
    fromKey, err := recoverNodeKey(crypto.Keccak256(buf[headSize:]), sig)
    if err != nil {
        return nil, fromKey, hash, err
    }
    var req packet
    //判断包的类型，根据类型创建相应的包
    switch ptype := sigdata[0]; ptype {
    case pingPacket:
        req = new(ping)
    case pongPacket:
        req = new(pong)
    case findnodePacket:
        req = new(findnode)
    case neighborsPacket:
        req = new(neighbors)
    default:
        return nil, fromKey, hash, fmt.Errorf("unknown type: %d", ptype)
    }
    s := rlp.NewStream(bytes.NewReader(sigdata[1:]), 0)
    err = s.Decode(req)
    return req, fromKey, hash, err
}
```

上面就是udp.readLoop()的相应代码分析，主要任务就是从udp端口读取数据，将数据解成四种包的一种，然后调用其preverify方法进行预验证，验证通过后交给相应包的handle函数处理。

下面分析下四种包的预验证和处理函数

ping包：

```
// Packet Handlers
func (req *ping) preverify(t *udp, from *net.UDPAddr, fromID enode.ID, fromKey encPubkey) error {
    // 首先检查是否过期
    if expired(req.Expiration) {
        return errExpired
    }
    // 根据发送方加密公钥解出公钥，确保可以正确解出
```

```
    key, err := decodePubkey(fromKey)
    if err != nil {
        return errors.New("invalid public key")
    }
    req.senderKey = key
    return nil
}
//如果收到一个 ping 请求 判断是否收到上一次此节点的pong请求是否过期，一般是24小时 ，如果过期则回发ping包并等待到pong回复时候 将对方节点加入k桶
func (req *ping) handle(t *udp, from *net.UDPAddr, fromID enode.ID, mac []byte) {
    // Reply.
    // 当收到Ping包时候，先向对方发送pong包
    t.send(from, fromID, pongPacket, &pong{
        To:         makeEndpoint(from, req.From.TCP),
        ReplyTok:   mac,
        Expiration: uint64(time.Now().Add(expiration).Unix()),
    })

    // Ping back if our last pong on file is too far in the past.
    // 根据对方请求携带的消息，将相应消息包装为一个node结构
    n := wrapNode(enode.NewV4(req.senderKey, from.IP, int(req.From.TCP), from.Port))
     // 查看数据库上次收到对方pong回应的时间是否大于24小时，若是则向对方发送ping包，并将对方加入k桶，k桶概念下节分析。若小于24小时，直接将对方加入到k桶
    if time.Since(t.db.LastPongReceived(n.ID(), from.IP)) > bondExpiration {
        t.sendPing(fromID, from, func() {
            t.tab.addVerifiedNode(n)
        })
    } else {
        t.tab.addVerifiedNode(n)
    }

    // Update node database and endpoint predictor.
    t.db.UpdateLastPingReceived(n.ID(), from.IP, time.Now())
    t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int(req.To.UDP)})
}
```

pong包：

```
// pong包的预验证逻辑
func (req *pong) preverify(t *udp, from *net.UDPAddr, fromID enode.ID, fromKey encPubkey) error {
    // 判断是否过期
    if expired(req.Expiration) {
        return errExpired
    }
    // 将pong包交给t.gotreply通道，看是否有一个匹配到，若没有匹配结果则返回错误
    if !t.handleReply(fromID, from.IP, pongPacket, req) {
        return errUnsolicitedReply
    }
    return nil
}
// pong包的handle函数
func (req *pong) handle(t *udp, from *net.UDPAddr, fromID enode.ID, mac []byte) {
    t.localNode.UDPEndpointStatement(from, &net.UDPAddr{IP: req.To.IP, Port: int(req.To.UDP)})
     // 简单更新下数据库中收到from节点pong包的时间
    t.db.UpdateLastPongReceived(fromID, from.IP, time.Now())
}
```

findnode包：

```
func (req *findnode) preverify(t *udp, from *net.UDPAddr, fromID enode.ID, fromKey encPubkey)
error {
    // 判断是否过期
    if expired(req.Expiration) {
        return errExpired
    }
    // 如果发送查找节点包的节点上次回复我们pong消息大于24小时，那么直接返回错误
    if time.Since(t.db.LastPongReceived(fromID, from.IP)) > bondExpiration {
        // No endpoint proof pong exists, we don't process the packet. This prevents an
        // attack vector where the discovery protocol could be used to amplify traffic in a
        // DDOS attack. A malicious actor would send a findnode request with the IP address
        // and UDP port of the target as the source address. The recipient of the findnode
        // packet would then send a neighbors packet (which is a much bigger packet than
        // findnode) to the victim.
        return errUnknownNode
    }
    return nil
}

func (req *findnode) handle(t *udp, from *net.UDPAddr, fromID enode.ID, mac []byte) {
    // Determine closest nodes.
    target := enode.ID(crypto.Keccak256Hash(req.Target[:]))
    t.tab.mutex.Lock()
     // 从自己的bucket里面查找bucketSize个离目标节点最近的
    closest := t.tab.closest(target, bucketSize).entries
    t.tab.mutex.Unlock()

    // Send neighbors in chunks with at most maxNeighbors per packet
    // to stay below the 1280 byte limit.
    p := neighbors{Expiration: uint64(time.Now().Add(expiration).Unix())}
    var sent bool
    for _, n := range closest {
        if netutil.CheckRelayIP(from.IP, n.IP()) == nil {
            p.Nodes = append(p.Nodes, nodeToRPC(n))
        }
        // 当添加的节点超过1280个字节时候开始发送回包
        if len(p.Nodes) == maxNeighbors {
                t.send(from, fromID, neighborsPacket, &p)
            p.Nodes = p.Nodes[:0]
            sent = true
        }
    }

    // 如果上面查找到的结果没超过neighbors包的限制，那么使用下面的逻辑再发送
    if len(p.Nodes) > 0 || !sent {
        t.send(from, fromID, neighborsPacket, &p)
    }
}
```

neighbors包：

```go
func (req *neighbors) preverify(t *udp, from *net.UDPAddr, fromID enode.ID, fromKey encPubkey) 
error {
    if expired(req.Expiration) {
        return errExpired
    }
    // 直接交给匹配通道进行匹配
    if !t.handleReply(fromID, from.IP, neighborsPacket, req) {
        return errUnsolicitedReply
    }
    return nil
}

func (req *neighbors) handle(t *udp, from *net.UDPAddr, fromID enode.ID, mac []byte) {
}
```

再来分析一个借助udp包发送ping请求的整个流程来分析它的匹配回调机制

```go
// sendPing sends a ping message to the given node and invokes the callback
// when the reply arrives.
func (t *udp) sendPing(toid enode.ID, toaddr *net.UDPAddr, callback func()) <-chan error {
    // 先封装一个ping请求
    req := &ping{
        // 版本号
        Version:    4,
        // 发送方
        From:       t.ourEndpoint(),
        // 接收方
        To:         makeEndpoint(toaddr, 0), // TODO: maybe use known TCP port from DB
        Expiration: uint64(time.Now().Add(expiration).Unix()),
    }
    // 对包进行编码
    packet, hash, err := encodePacket(t.priv, pingPacket, req)
    if err != nil {
        errc := make(chan error, 1)
        errc <- err
        return errc
    }
    // Add a matcher for the reply to the pending reply queue. Pongs are matched if they
    // reference the ping we're about to send.
    // 开始将包组装成一个replyMatcher放入plist
    errc := t.pending(toid, toaddr.IP, pongPacket, func(p interface{}) (matched bool, requestDone 
bool) {
        matched = bytes.Equal(p.(*pong).ReplyTok, hash)
        if matched && callback != nil {
            callback()
        }
        return matched, matched
    })
    // Send the packet.
    t.localNode.UDPContact(toaddr)
     // 最后发送包
    t.write(toaddr, toid, req.name(), packet)
    return errc
}
```

```go
// pending adds a reply matcher to the pending reply queue.
// see the documentation of type replyMatcher for a detailed explanation.
func (t *udp) pending(id enode.ID, ip net.IP, ptype byte, callback replyMatchFunc) <-chan error {
    ch := make(chan error, 1)
     // 根据包的信息组装为一个replyMatcher并放入addReplyMatcher通道，最后等待t.gotreply有消息收到后进行匹配，看是不是此消息的回复
    p := &replyMatcher{from: id, ip: ip, ptype: ptype, callback: callback, errc: ch}
    select {
    case t.addReplyMatcher <- p:
        // loop will handle it
    case <-t.closing:
        ch <- errClosed
    }
    return ch
}
```

以上就是udp里面的主要循环，主要有loop()主要做的工作是将过期的包删除，处理各种udp相关通道的消息readLoop()主要循环从udp端口读取消息，将消息解码成相应包，调用相应包的处理函数，并将消息推送给相关通道。最后分析了下节点如何利用udp发送包，并把回应包放入plist,若正确收到回复，则调用相应包的callback()函数。

1.2 go-ethereum/p2p/discover/table.go

先介绍下table下的相关数据结构

```go
const (
    alpha           = 3  // Kademlia concurrency factor
    bucketSize      = 16 // Kademlia bucket size bucket单个桶大小
    maxReplacements = 10 // Size of per-bucket replacement list //最大替换数

    // We keep buckets for the upper 1/15 of distances because
    // it's very unlikely we'll ever encounter a node that's closer.
    hashBits          = len(common.Hash{}) * 8 //256
    nBuckets          = hashBits / 15       // Number of buckets 17 bucket个数
    bucketMinDistance = hashBits - nBuckets // Log distance of closest bucket bucket 0层和1层之间的距离

    // IP address limits.
    bucketIPLimit, bucketSubnet = 2, 24 // at most 2 addresses from the same /24
    tableIPLimit, tableSubnet   = 10, 24

    maxFindnodeFailures = 5 // Nodes exceeding this limit are dropped 发现节点最大失败次数
    refreshInterval     = 30 * time.Minute // 刷新table间隙
    revalidateInterval  = 10 * time.Second //做最后节点验证间隙
    copyNodesInterval   = 30 * time.Second // 将liveness大于1的节点拷贝到数据nodes目录的间隙
    seedMinTableTime    = 5 * time.Minute
    seedCount           = 30 // 种子节点数量
    seedMaxAge          = 5 * 24 * time.Hour
)
// table数据结构定义
type Table struct {
    mutex   sync.Mutex        // protects buckets, bucket content, nursery, rand
    buckets [nBuckets]*bucket // index of known nodes by distance bucket数组定义
    nursery []*node           // bootstrap nodes 引导节点切片
    rand    *mrand.Rand       // source of randomness, periodically reseeded
    ips     netutil.DistinctNetSet
```

```
    db         *enode.DB // database of known nodes 使用db将已知节点存入nodes
    net        transport
    refreshReq chan chan struct{}
    initDone   chan struct{}

    closeOnce sync.Once
    closeReq  chan struct{}
    closed    chan struct{}

    nodeAddedHook func(*node) // for testing
}
```

下面开始分析newTable()方法的逻辑

```
func newTable(t transport, db *enode.DB, bootnodes []*enode.Node) (*Table, error) {
    tab := &Table{
        net:        t,// udp实现了transport接口的方法，table借用udp进行数据传输
        db:         db,
        refreshReq: make(chan chan struct{}), // table刷新通道
        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}, // 24  , 10
    }
    if err := tab.setFallbackNodes(bootnodes); err != nil {
        return nil, err
    }
    // 开始将17个bucket进行变量初始化
    for i := range tab.buckets {
        tab.buckets[i] = &bucket{
            ips: netutil.DistinctNetSet{Subnet: bucketSubnet, Limit: bucketIPLimit},
        }
    }
    // 读取一个随机数种子
    tab.seedRand()
    tab.loadSeedNodes()
    //进行table循环  主要做doRefresh   revalidate  copyNodes
    go tab.loop()
    return tab, nil
}
// 下面是table主循环代码
// loop schedules refresh, revalidate runs and coordinates shutdown.
func (tab *Table) loop() {
    var (
        // 首先创建三个定时器，主要功能下面讲解
        revalidate     = time.NewTimer(tab.nextRevalidateTime())
        refresh        = time.NewTicker(refreshInterval)
        copyNodes      = time.NewTicker(copyNodesInterval)
        refreshDone    = make(chan struct{})           // where doRefresh reports completion
        revalidateDone chan struct{}                   // where doRevalidate reports completion
        waiting        = []chan struct{}{tab.initDone} // holds waiting callers while doRefresh
runs
    )
    defer refresh.Stop()
    defer revalidate.Stop()
```

```go
    defer copyNodes.Stop()

    // Start initial refresh.
    go tab.doRefresh(refreshDone)

loop:
    for {
        select {
        // 如果到了刷新时间，则进行table刷新 30分钟一次
        case <-refresh.C:
            tab.seedRand()
            if refreshDone == nil {
                refreshDone = make(chan struct{})
                go tab.doRefresh(refreshDone)
            }
        case req := <-tab.refreshReq:
            waiting = append(waiting, req)
            if refreshDone == nil {
                refreshDone = make(chan struct{})
                go tab.doRefresh(refreshDone)
            }
        case <-refreshDone:
            for _, ch := range waiting {
                close(ch)
            }
            waiting, refreshDone = nil, nil
        //到了验证随机桶里最后一个节点是否存活  主要保持桶鲜活 10秒一次
        case <-revalidate.C:
            revalidateDone = make(chan struct{})
            go tab.doRevalidate(revalidateDone)
        case <-revalidateDone:
            revalidate.Reset(tab.nextRevalidateTime())
            revalidateDone = nil
        // 30 秒一次  将所有 桶里盼活计数大于0 的节点存到数据库
        case <-copyNodes.C:
            go tab.copyLiveNodes()
        case <-tab.closeReq:
            break loop
        }
    }

    if refreshDone != nil {
        <-refreshDone
    }
    for _, ch := range waiting {
        close(ch)
    }
    if revalidateDone != nil {
        <-revalidateDone
    }
    close(tab.closed)
}


// doRefresh performs a lookup for a random target to keep buckets
// full. seed nodes are inserted if the table is empty (initial
// bootstrap or discarded faulty peers).
```

```go
func (tab *Table) doRefresh(done chan struct{}) {
    defer close(done)
    //从数据库加载那些被存在数据库中且仍然存活的节点
    // Load nodes from the database and insert
    // them. This should yield a few previously seen nodes that are
    // (hopefully) still alive.
    tab.loadSeedNodes()
    //先进行一次自我查找 为了让别人尽可能把自己加到bucket里面
    // Run self lookup to discover new neighbor nodes.
    // We can only do this if we have a secp256k1 identity.
    var key ecdsa.PublicKey
    if err := tab.self().Load((*enode.Secp256k1)(&key)); err == nil {
        tab.lookup(encodePubkey(&key), false)
    }

    // The Kademlia paper specifies that the bucket refresh should
    // perform a lookup in the least recently used bucket. We cannot
    // adhere to this because the findnode target is a 512bit value
    // (not hash-sized) and it is not easily possible to generate a
    // sha3 preimage that falls into a chosen bucket.
    // We perform a few lookups with a random target instead.
    // 查询了三个随机节点 ，这三个随机节点不一定存在
    for i := 0; i < 3; i++ {
        var target encPubkey
        crand.Read(target[:])
        tab.lookup(target, false)
    }
}

// lookup performs a network search for nodes close to the given target. It approaches the
// target by querying nodes that are closer to it on each iteration. The given target does
// not need to be an actual node identifier.
func (tab *Table) lookup(targetKey encPubkey, refreshIfEmpty bool) []*node {
    var (
        // 目标节点
        target         = enode.ID(crypto.Keccak256Hash(targetKey[:]))
        // 本次lookup询问过的节点
        asked          = make(map[enode.ID]bool)
        seen           = make(map[enode.ID]bool)
        // 返回结果
        reply          = make(chan []*node, alpha)
        // 处于pending的查询
        pendingQueries = 0
        // 本次lookup的结果
        result         *nodesByDistance
    )
    // 先设置自己为已经询问过的节点
    // don't query further if we hit ourself.
    // unlikely to happen often in practice.
    asked[tab.self().ID()] = true

    for {
        tab.mutex.Lock()
        //  从本地桶里拿出来16个最近的节点返回，根据nodeid进行异或的逻辑距离
        // generate initial result set
        result = tab.closest(target, bucketSize)
        tab.mutex.Unlock()
```

```go
        //
        if len(result.entries) > 0 || !refreshIfEmpty {
            break
        }
        // The result set is empty, all nodes were dropped, refresh.
        // We actually wait for the refresh to complete here. The very
        // first query will hit this case and run the bootstrapping
        // logic.
        <-tab.refresh()
        refreshIfEmpty = false
    }
    // 以下for循环按照逻辑来看，会将本地离target最近的16个节点一一发送findnode包，将返回的结果（若全部返回正常符合条件的结果来看是16*16个）里面再拿出来离target最近的16个放入result里面
    for {
        // 遍历本地查找到离target距离近的节点
        // 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]
             // 如果节点未询问
            if !asked[n.ID()] {
                // 将此节点标位已经询问
                  asked[n.ID()] = true
                 // pending计数递增
                pendingQueries++
                  // 开启协程借助udp发送向n发送findnode包
                go tab.findnode(n, targetKey, reply)
            }
        }
        if pendingQueries == 0 {
            // we have asked all closest nodes, stop the search
            break
        }
        select {
        case nodes := <-reply:
            // 遍历findnode查找到的结果放入result中，并将放入result中的每一个节点标记为seen
            for _, n := range nodes {
                if n != nil && !seen[n.ID()] {
                    seen[n.ID()] = true
                    result.push(n, bucketSize)
                }
            }
        case <-tab.closeReq:
            return nil // shutdown, no need to continue.
        }
        // pending计数递减
        pendingQueries--
    }
    return result.entries
}
// 下面是findnode的逻辑
func (tab *Table) findnode(n *node, targetKey encPubkey, reply chan<- []*node) {
    //先从数据库里面读出此节点的失败次数
    fails := tab.db.FindFails(n.ID(), n.IP())
    //使用udp 向n发送findnode包
    r, err := tab.net.findnode(n.ID(), n.addr(), targetKey)
    if err == errClosed {
        // Avoid recording failures on shutdown.
```

```
        reply <- nil
        return
    } else if len(r) == 0 {
        // 如果向n发送findnode包后返回的结果数为0，那么将此节点的查找失败次数加1
        fails++
        tab.db.UpdateFindFails(n.ID(), n.IP(), fails)
        log.Trace("Findnode failed", "id", n.ID(), "failcount", fails, "err", err)
        //  如果让一个节点查找一个节点失败次数过多 比如大于5此  那么从本地buckets 里面删除此节点
        if fails >= maxFindnodeFailures {
            log.Trace("Too many findnode failures, dropping", "id", n.ID(), "failcount", fails)
            tab.delete(n)
        }
    } else if fails > 0 {
        //  如果本次查找成功 且此节点以前查找失败次数不为0  那么减少一次本节点的失败次数   防止好的节点被删除
        tab.db.UpdateFindFails(n.ID(), n.IP(), fails-1)
    }

    // Grab as many nodes as possible. Some of them might not be alive anymore, but we'll
    // just remove those again during revalidation.
    //将查找到的结果添加到自己的Bucket
    for _, n := range r {
        tab.addSeenNode(n)
    }
    //  将结果返回应答通道
    reply <- r
}
// 下面再分析两个方法findnode和addSeenNode
// findnode 发送一个findnode请求到给定的地址然后一直等待到节点发送k个邻居节点回来
// findnode sends a findnode request to the given node and waits until
// the node has sent up to k neighbors.
func (t *udp) findnode(toid enode.ID, toaddr *net.UDPAddr, target encPubkey) ([]*node, error) {
    // If we haven't seen a ping from the destination node for a while, it won't remember
    // our endpoint proof and reject findnode. Solicit a ping first.
    //24小时过期 计算上次收到to地址的ping包时间是否大于过期时间，如果大于  则ping对方  等待500ms 对方返回结果 ping对方后，对方会将本地节点加入对方的bucket里面
    if time.Since(t.db.LastPingReceived(toid, toaddr.IP)) > bondExpiration {
        t.ping(toid, toaddr)
        // Wait for them to ping back and process our pong.
        time.Sleep(respTimeout) //500毫秒
    }

    // Add a matcher for 'neighbours' replies to the pending reply queue. The matcher is
    // active until enough nodes have been received.
    nodes := make([]*node, 0, bucketSize)
    nreceived := 0
    // 下面将findnode的相关数据包装为一个replyMatcher放入plist等待回复
    errc := t.pending(toid, toaddr.IP, neighborsPacket, func(r interface{}) (matched bool, requestDone bool) {
        reply := r.(*neighbors)
        for _, rn := range reply.Nodes {
            nreceived++
            n, err := t.nodeFromRPC(toaddr, rn)
            if err != nil {
                log.Trace("Invalid neighbor node received", "ip", rn.IP, "addr", toaddr, "err", err)
                continue
```

```
            }
            nodes = append(nodes, n)
        }
        return true, nreceived >= bucketSize
    })
    t.send(toaddr, toid, findnodePacket, &findnode{
        Target:     target,
        Expiration: uint64(time.Now().Add(expiration).Unix()),
    })
    // 将findnode返回的结果放入nodes里面返回
    return nodes, <-errc
}
// 下面是将节点加入table下面bucket里面的逻辑
// addSeenNode adds a node which may or may not be live to the end of a bucket. If the
// bucket has space available, adding the node succeeds immediately. Otherwise, the node is
// added to the replacements list.
// 如果bucket[x]有位置则添加   否则 添加到replacement
// The caller must not hold tab.mutex.
func (tab *Table) addSeenNode(n *node) {
    // 若nodeId是本身则直接返回
    if n.ID() == tab.self().ID() {
        return
    }
    // 加锁
    tab.mutex.Lock()
    defer tab.mutex.Unlock()
    // 计算节点所处位置，若n和当前节点间的距离小于239（因为这样的节点很少）那么都放在buckets[0]位置
    b := tab.bucket(n.ID())
    //  判断是否已经包含
    if contains(b.entries, n.ID()) {
        // Already in bucket, don't add.
        return
    }
    // 判断桶里元素大小
    if len(b.entries) >= bucketSize {
        //如果通满了，则添加节点到替补位置
        // Bucket full, maybe add as replacement.
        tab.addReplacement(b, n)
        return
    }
    if !tab.addIP(b, n.IP()) {
        // Can't add: IP limit reached.
        return
    }
    // 如果桶没满，则追加到桶的最后一个位置
    // Add to end of bucket:
    b.entries = append(b.entries, n)
    //  并且尝试从replacements里面删除n
    b.replacements = deleteNode(b.replacements, n)
    //  将node节点添加进桶的时间更新为现在
    n.addedAt = time.Now()
    if tab.nodeAddedHook != nil {
        tab.nodeAddedHook(n)
    }
}

// addReplacement方法分析如下
```

```go
func (tab *Table) addReplacement(b *bucket, n *node) {
    // 遍历所有替补   看是不是已经有节点n
    for _, e := range b.replacements {
        if e.ID() == n.ID() {
            return // already in list
        }
    }
    //
    if !tab.addIP(b, n.IP()) {
        return
    }
    var removed *node
    //开始将替补节点插入 替补buckets
    b.replacements, removed = pushNode(b.replacements, n, maxReplacements)
    if removed != nil {
        tab.removeIP(b, removed.IP())
    }
}
// pushNode adds n to the front of list, keeping at most max items.
func pushNode(list []*node, n *node, max int) ([]*node, *node) {
    //  首先判断 替补列表小于最大限制
    if len(list) < max {
        //那么给替补列表最后添加一个空元素
        list = append(list, nil)
    }
    // 将替补列表最后一个元素删除 上面和下面步骤合起来看    给列表添加一个空元素是为了返回值方便
    removed := list[len(list)-1]
    // 删除掉最后一个元素以后  将所有元素往后移动一位空出第一个位置   此时要删除的元素已经删除  存放在
removed中
    copy(list[1:], list)
    //将第一个位置放入要添加的节点
    list[0] = n
    return list, removed
}
```

以上就是table下面doRefresh()相关逻辑，下面紧接着分析一下doRevalidate()和copyLiveNodes()的逻辑

```go
// doRevalidate checks that the last node in a random bucket is still live
// and replaces or deletes the node if it isn't.
// 检查随机桶的最后一个节点是否存活，若不是则替代或者删除它
func (tab *Table) doRevalidate(done chan<- struct{}) {
    defer func() { done <- struct{}{} }()
    // 从随机桶buckets[random]中拿出最后一个节点，和桶索引
    last, bi := tab.nodeToRevalidate()
    if last == nil {
        // No non-empty bucket found.
        return
    }
    //对last节点进行一次ping操作
    // Ping the selected node and wait for a pong.
    err := tab.net.ping(last.ID(), last.addr())

    tab.mutex.Lock()
    defer tab.mutex.Unlock()
     // 获取bi位置的桶
    b := tab.buckets[bi]
```

```
        //如果ping通  则将bucket的最后一个元素移到最前面
        if err == nil {
             // 将last节点的判活检查计数自增，这个计数主要用于copyNodes
            // The node responded, move it to the front.
            last.livenessChecks++
            log.Debug("Revalidated node", "b", bi, "id", last.ID(), "checks", last.livenessChecks)
             // 将last移动到bucket[bi]的最前方
            tab.bumpInBucket(b, last)
            return
        }
        // 没有pong包收到时候，那么从replacement里面随机拿一个节点替换last ，如果buckets[bi]的替换者为空，那么将last直接删除
        // No reply received, pick a replacement or delete the node if there aren't
        // any replacements.
        if r := tab.replace(b, last); r != nil {
            log.Debug("Replaced dead node", "b", bi, "id", last.ID(), "ip", last.IP(), "checks", last.livenessChecks, "r", r.ID(), "rip", r.IP())
        } else {
            log.Debug("Removed dead node", "b", bi, "id", last.ID(), "ip", last.IP(), "checks", last.livenessChecks)
        }
}
//------------------------以上主要为revalidate的逻辑-----------------------------------
//下面分析copyLiveNodes()逻辑
// copyLiveNodes adds nodes from the table to the database if they have been in the table
// longer then minTableTime.
//将盼活检查计数大于0的存入数据库
func (tab *Table) copyLiveNodes() {
    tab.mutex.Lock()
    defer tab.mutex.Unlock()

    now := time.Now()
    // 遍历table的所有bucket的所有节点
    for _, b := range &tab.buckets {
        for _, n := range b.entries {
            //  将判断盼活检查次数大于0且存在桶里的时间大于5分钟的节点写入数据库（目录为nodes）
            if n.livenessChecks > 0 && now.Sub(n.addedAt) >= seedMinTableTime {
                tab.db.UpdateNode(unwrapNode(n))
            }
        }
    }
}
```

2、总结

以上就是udp和table的主要逻辑分析，主要是以太坊节点发现机制的实现，结合p2p第一部分代码的分析，就可以很好的了解以太坊p2p底层，上层逻辑等待下次分享protocolManager逻辑时候就可以将p2p部分与以太坊上层应用如交易广播等等逻辑联系起来。