以太坊的 P2P 网络

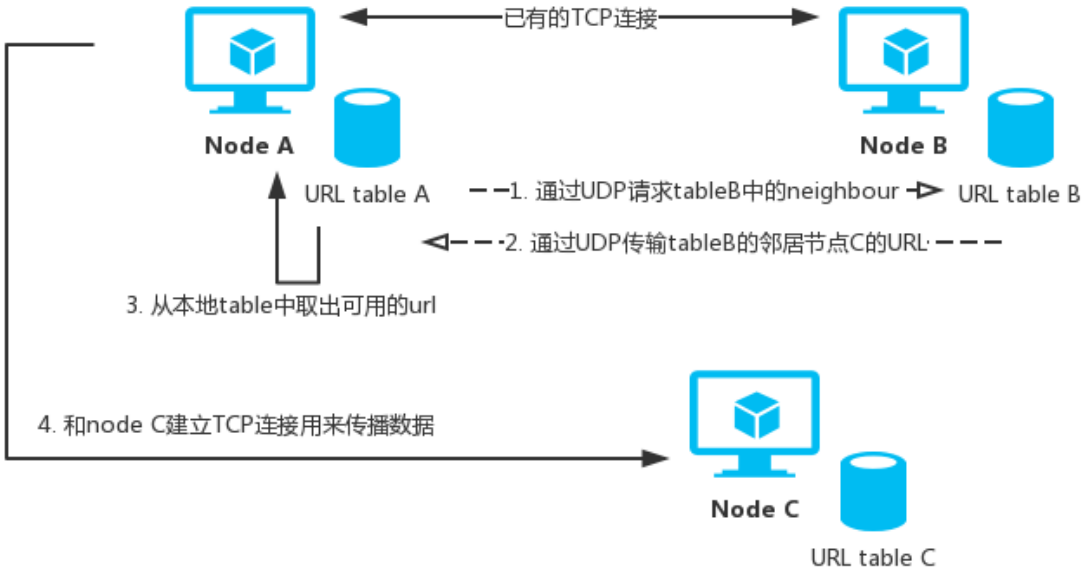
以太坊的p2p网络主要有两部分构成：节点之间互相连接用于传输数据的tcp网络和节点之间互相广播用于节点发现的udp网络。本篇文章将重点介绍用于节点发现的udp网络部分。

p.s. 主要针对以太坊源码的对应实现，相关的算法如kademlia DHT算法可以参考其他文章。

P2P整体结构

在以太坊中，节点之间数据的传输是通过tcp来完成的。但是节点如何找到可以进行数据传输的节点？这就涉及到P2P的核心部分节**点**发现了。每个节点会维护一个table，table中会存储可供连接的其他节点的地址。这个table通过基于udp的节点发现机制来保持更新和可用。当一个节点需要更多的TCP连接用于数据传输时，它就会从这个table中获取待连接的地址并建立TCP连接。

与节点建立连接的流程大致如下图：



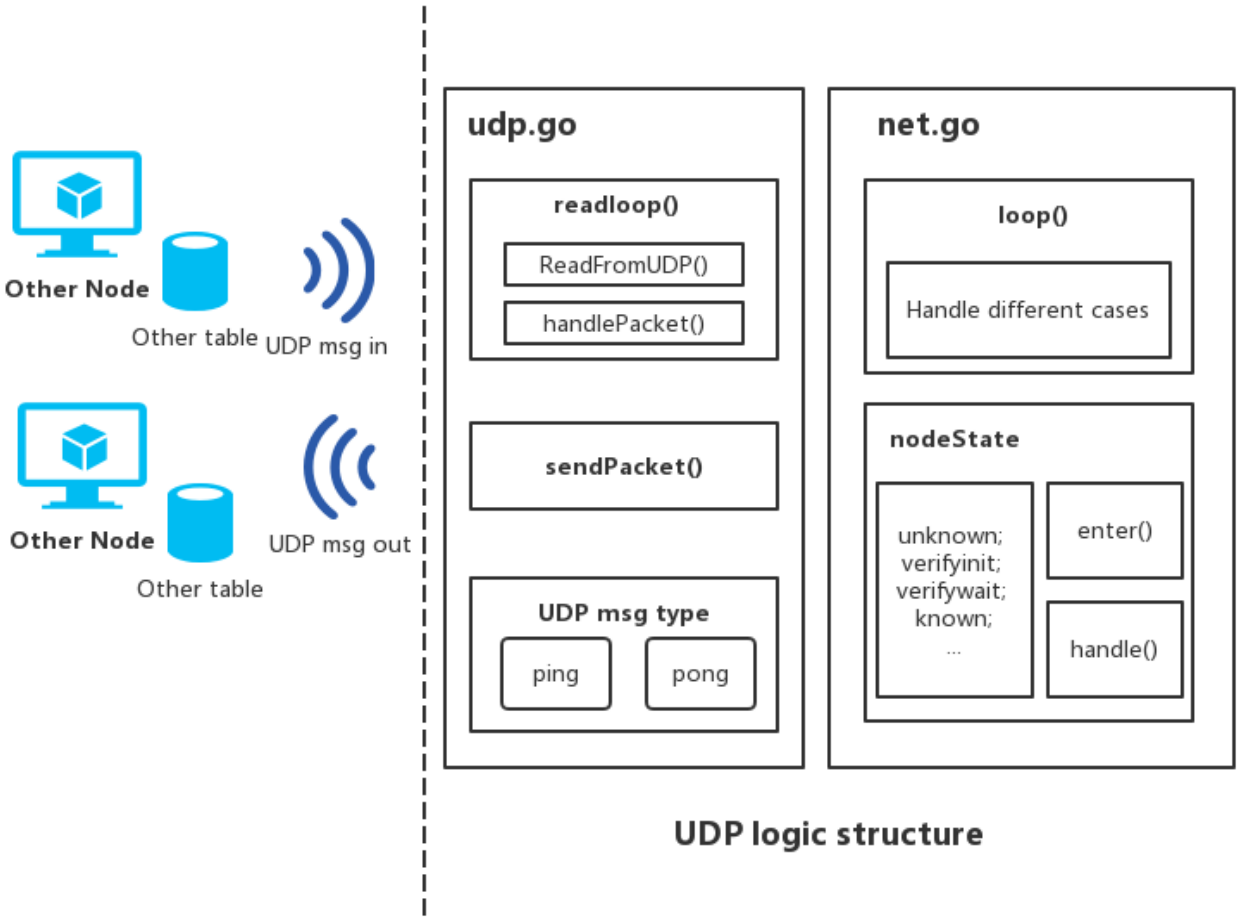
上图中，第一和第二步每隔一段时间就会重复一次。目的是保持当前节点的table所存储的url的可用性。当节点需要寻找新的节点用于tcp数据传输时，就从一直在更新维护的table中取出一定数量的url进行连接即可。

> 为了防止节点之间的 tcp 互联形成信息孤岛，每个以太坊节点间的连接都分为两种类型：bound_in 和 bound_out。bound_in 指本节点接收到别人发来的请求后建立的 tcp 连接，bound_out 指本节点主动发起的 tcp 连接。假设一个节点最多只能与6个节点互联，那么在默认的设置中，节点最多只能主动和 4 个节点连接，剩余的必须留着给其它节点接入用。对于 bound_in 连接的数量则不做限制。

UDP部分

前文提到过，以太坊会维护一个table结构用于存储发现的节点url，当需要连接新的节点时会从table中获取一定数量的url进行tcp连接。在这个过程中，table的更新和维护将会在独立的goroutine中进行。这部分涉及的

代码主要在 `p2p/discvV5` 目录下的 `udp.go`, `net.go`, `table.go` 这三个文件中。下面我们看看udp这部分的主要逻辑结构：



readloop()

`readloop()`方法在 p2p/discvV5/udp.go 文件中，用来监听所有收到的udp消息。通过 `ReadFromUDP()` 接收收到的消息，然后在 `handlePacket()` 方法内将消息序列化后传递给后续的消息处理程序。

sendPacket()

`sendPacket()` 用于将消息通过 udp 发送给目标地址。

UDP msg type

所有通过 udp 发出的消息都分为两种类型：ping 和 pong。ping 作为通信的发起类型，pong 作为通信的应答类型。一个完整的通讯应该是：

1. Node A 向 Node B 发送 ping 类型的 udp 消息。
2. Node B 收到此消息后进行消息处理。
3. Node B 向 Node A 发送 pong 类型的消息作为之前收到的 ping 消息的应答。

loop()

loop() 方法在 p2p/discvV5/net.go 文件中，是整个 p2p 部分的核心方法。它控制了节点发现机制的大部分逻辑内容，由一个大的 select 进行各种 case 的监控。主体代码大致如下：

```
func (net *Network) loop() {
    ...
    for {
        select {
        case pkt := <-net.read:
            // 处理收到的 udp 消息
        case timeout := <-net.timeout:
            // 发送的 udp ping 消息超时未收到 pong 应答
        case q := <-net.queryReq:
            // 从table中查找距离某个target最近的n个节点
        case f := <-net.tableOpReq:
            // 操作table，这个case主要是为了防止table被异步操作
        case <-refreshTimer.C:
            // 计时器到点，刷新table里的url
        ...
        }
    }
    ...
}
```

nodeState

nodeState 表示了一个url对应的状态。在 net.go 的 init() 方法中定义了各种 nodeState：

```
type nodeState struct {
    name     string
    handle   func( ... ) ( ... )
    enter    func( ... )
    canQuery bool
}

var (
    unknown          *nodeState     // 新收到的未知节点
    verifyinit       *nodeState     // 处在初次验证的节点
    verifywait       *nodeState     // 正在等待应答的节点
    remoteverifywait *nodeState
    known            *nodeState     // 已经完成应答的节点，可以加入到table中
    contested        *nodeState
    unresponsive     *nodeState
)

func init() {
    unknown = &nodeState{
        ...
    }
    verifyinit = &nodeState{
        ...
```

```
    }
    ...
}
```

nodeState主要是为了记录新地址的可用状态。当节点接触新的 url 时，此 url 会首先被定义为 unknown 状态，然后进入后续 ping pong 验证阶段。验证流程过完以后 就会被定义为 known 状态然后保存到 table 中。

enter() 和 handle()

nodeState 结构下定义了两个方法 `enter()` 和 `handle()`。

- enter 是 nodeState 的入口方法，当一个 url 从 nodeState1 转到 nodeState2 时，它就会调用 nodeState2 的 enter() 方法。
- handle 是 nodeState 的处理方法，当节点收到某个 url 的 udp 消息时，会调用此 url 当前 nodeState 的 handle() 方法。一般 handle() 方法会针对不同的 udp 消息类型进行不同的逻辑处理：

```
// verifywait 的 handle 方法
func (net *Network, n *Node, ev nodeEvent, pkt *ingressPacket) (*nodeState, error)
{
    switch ev {
        case pingPacket:
            net.handlePing(n, pkt)
            return verifywait, nil
        case pongPacket:
            err := net.handleKnownPong(n, pkt)
            return known, err
        case pongTimeout:
            return unknown, nil
        default:
            return verifywait, errInvalidEvent
    }
}
```

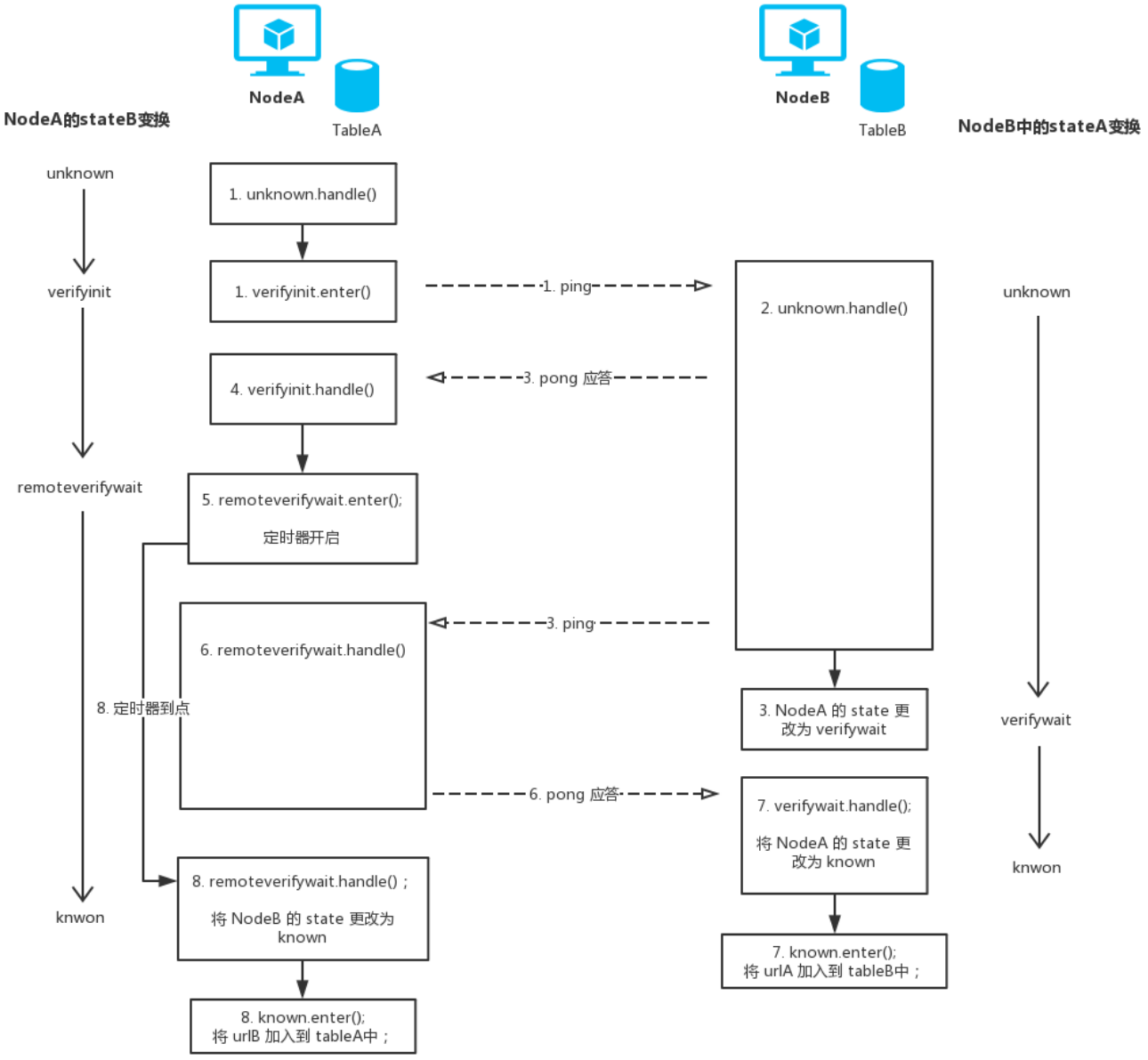
handle() 方法在处理后会返回一个 nodeState，意思是我当前的 nodeState 在处理了这个 udp 消息后下一步将要转换的新的 state。比如在上面的代码中，verifywait 状态下的 url 在收到 pong 类型的 udp 消息后就会转换成 known 状态。

新节点加入流程

新节点的加入分两种：

1. 获取到新的 url 后主动向对方发起 udp 请求，在完成一系列验证后将此 url 加入到 table 中。
2. 收到陌生 url 地址发来的 udp 请求，在互相做完验证后加入到 table 中。

这两种情况是相对应的，NodeA 获取到 NodeB 的 url 后向 NodeB 发起 udp 请求（情况1），NodeB 收到来自 NodeA 的陌生 url 后做应答（情况2）。双方验证完成后互相将对方的 url 加入到table中。整个流程步骤如下：



1. 作为验证发起方，NodeA 在收到 url 后会将这个 url 的 nodeState 设置为 `verifyinit` 并通过 verifyinit 的 enter() 方法向 nodeB 的 url 发起 ping 消息。
2. NodeB 收到此 ping 消息后先获取此 url 的 nodeState，发现未找到于是将其设置为 unknown 然后调用 unknown 的 handle() 方法。
3. 在 handle() 方法中，NodeB 会先向 NodeA 发送一个 pong 消息表示对收到的 ping 的应答。然后再向 NodeA 发送一个新的 ping 消息等待 NodeA的应答。同时将 NodeA 的 state 设置为 verifywait。
4. NodeA 收到 NodeB 发来的 pong 消息后调用 verifyinit 状态的 handle() 然后将 NodeB 设置为 `remoteverifywait` 状态。remoteverifywait 的意思就是 NodeB 在我这个节点（NodeA）这里已经经过 ping pong 验证通过了，但是我还没在 NodeB 那边经过 ping pong 验证。
5. 将 NodeB 的 state 转换为 remoteverifywait 后会调用 remoteverifywait 的 enter() 方法。此方法开启一个定时器，定时器到点后向本节点发送一个 timeout 消息。
6. 处理完 pong 消息后，NodeA 又收到了来自 NodeB的 ping 消息，此时调用 remoteverifywait 的 handle() 向 NodeB 发送一个 pong 应答。
7. NodeB 收到 NodeA 的 pong 应答后调用 verifywait 的 handle() 方法，此方法会将 NodeA 的 state 设置为 known。在 state 转换后调用新 state 也就是 known state 的 enter() 方法。known 的 enter() 中会将 NodeA url 加入到 NodeB 的 table 中。

8. Step5 中 NodeA 的定时器到点并向 NodeA 自己发送 timeout 消息。此消息会调用 remoteverifywait 的 handle() 然后将 NodeB 的状态设置为 known。和 Step7 中一样，更改状态为 known 后 NodeB 的 url 会加入到 NodeA 的 table 中。

p.s. 以上步骤结合源码会更清晰

lookup节点发现

上文中，NodeA 收到了 NodeB 的 url 从而发起两者间的联系。那么 NodeA 是如何得到这个 url 的呢？答案是通过 `lookup()` 方法来发现这个 url。

lookup() 方法源码在 p2p/discvV5/net.go 文件中。此方法在需要获取新的 url 或者需要刷新 table 时被调用。

lookup() 需要输入一个 target 作为目标，然后寻找和 target 最近的 n 个节点。target 是由一个节点的 NodeID 经过哈希计算得来的。在以太坊中，每次生成新的target 都会虚构一个节点 url 然后 hash 计算得到 target。lookup() 方法的基本思路是先从本地 table 获取和 target 最近的一部分邻居节点，然后再获取这些邻居节点的 table 中和 target 最近的部分节点。最后从得到的节点中选距离 target 最近的 n 个节点 url 作为新的 table 内容。如果这些 url 中有部分是之前未接触的，则程序会走上文提到的新节点加入流程来建立关系。

这里提到的距离最近不是指物理距离，而是数理上的最近。程序会计算节点 NodeID 经过 hash 计算后和 target 的差异大小来判断数理上的距离大小，具体可以参考 closest() 方法。

Conclusion

以太坊的P2P大致就是上述的实现方式。总的来说就是通过 udp 发现可供使用的节点 url 并将这些 url 维护在本地的 table 中，通过 tcp 和其他节点进行连接并进行数据传输，当需要新的节点时从 table 中获取未接触过的 url 建立新的 tcp 连接。

（完）