table.go主要实现了p2p的Kademlia协议。

1. 几类数据结构
   1. Kad路由表

|  |
| --- |
| type Table struct {  mutex sync.Mutex // protects buckets, their content, and nursery  buckets [nBuckets]\*bucket // index of known nodes by distance  nursery []\*Node // bootstrap nodes 引导节点  db \*nodeDB // database of known nodes 数据库  bondmu sync.Mutex  bonding map[NodeID]\*bondproc  bondslots chan struct{} // limits total number of active bonding processes  nodeAddedHook func(\*Node) // for testing  net transport  self \*Node // metadata of the local node  } |

* 1. 路由表的每一行K-桶

|  |
| --- |
| type bucket struct {  lastLookup time.Time //上次lookup刷新时刻  entries []\*Node //k-bucket 的真实队列 (最多放 16 个) 实时条目，按上次联系时间排序  } |

* 1. 存储距离target最近的若干节点

|  |
| --- |
| type nodesByDistance struct {  entries []\*Node  target common.Hash  } |

1. 函数

|  |
| --- |
| func newTable(t transport, ourID NodeID, ourAddr \*net.UDPAddr, nodeDBPath string) \*Table  在本地新建一个Kad路由表 |

|  |
| --- |
| func randUint(max uint32) uint32  具体实现不是很理解 |

1. Table结构体的方法

|  |
| --- |
| func (tab \*Table) Self() \*Node  返回存储该Table路由表的本地节点 |

|  |
| --- |
| func (tab \*Table) ReadRandomNodes(buf []\*Node) (n int)  作用不是很清楚 |

|  |
| --- |
| func (tab \*Table) Close()  关闭Table结构体相关的通信socket和leveldb数据库 |

|  |
| --- |
| func (tab \*Table) Bootstrap(nodes []\*Node)   1. 将table表的nursery切片中存储的引导节点ID求哈希值 2. 调用table的refresh方法更新整个Kad路由表(分为路由表为空和非空两种处理方式，这里将会采用为空的处理方式，也就是利用引导节点来填充空的Kad路由表) |

|  |
| --- |
| func (tab \*Table) Lookup(targetID NodeID) []\*Node 重点  对给定的目标节点进行查询   1. 调用table的closest()方法查询自身路由表中距离目标节点最近的bucketSize个节点 2. 在上述查询结果不为空的情况下，针对于每一个查询获取的最近节点调用table的net接口中的findnode()方法，获取其bucketSize个邻居节点(邻居节点的选取似乎是随机的，而非距离目标最近的节点) 3. 与这些邻居节点建立bond，仅取出成功建立bond的节点 4. 取出所有唯一不重复的邻居节点存储在result切片中。调用result的push方法保证result中只存储距离目标节点最近的bucketSize个不重复的邻居节点 5. 将result.entries(包含新的距离目标节点最近的 bucketSize 个节点)作为返回值   疑问：r, err := tab.net.findnode(n.ID, n.addr(), targetID) 具体的实现在哪？猜测是在udp.go中实现？ |

|  |
| --- |
| func (tab \*Table) refresh()  本方法负责更新整个Kad路由表   1. 需要判断Kad路由表是否为空 2. 非空：生成一个随机的目标ID，对这个目标节点执行tab.Lookup方法(更新自己的路由表，与新的节点建立bond) 3. 为空：在数据库中查询以前获知的若干个种子节点，与这些种子节点建立bond连接，之后再调用tab.Lookup方法与这些种子节点的邻居节点建立bond连接 |

|  |
| --- |
| func (tab \*Table) closest(target common.Hash, nresults int) \*nodesByDistance  查找本地Kad路由表中与目标节点ID的哈希最为接近的若干节点‘’  底层调用了nodesByDistance结构体的push方法获得了存储距离target最近的nresults个节点的close结构体作为返回值 |

|  |
| --- |
| func (tab \*Table) len() (n int)  计算整个Kad路由表保存的远端节点的数目 |

|  |
| --- |
| func (tab \*Table) bond(pinged bool, id NodeID, addr \*net.UDPAddr, tcpPort uint16) (\*Node, error) 重点  在本地节点向指定节点发送findnode请求之前必须先建立bond   1. 在本地数据库中查询该指定id的节点 2. 没有检索到：调用tab.pingpong方法，成功完成ping-pong之后即可建立bond连接 3. 检索到了：调用tab.pingreplace方法，更新对应行的K-桶(不需要重新建立bond) |

|  |
| --- |
| func (tab \*Table) bondall(nodes []\*Node) (result []\*Node)  采用多协程方式调用bond方法，与nodes切片中的若干节点建立bond |

|  |
| --- |
| func (tab \*Table) pingpong(w \*bondproc, pinged bool, id NodeID, addr \*net.UDPAddr, tcpPort uint16)   1. 调用tab.ping方法向指定节点发送ping消息，然后等待pong回复。收到回复之后将对方节点更新入自己的数据库 2. 判断是否收到对方的ping消息。如果没有则调用tab.net.waitping方法等待对方的ping消息；如果有就不必等待了   Ping-pong完成之后还需等待对方向自己发送ping？ |

|  |
| --- |
| func (tab \*Table) pingreplace(new \*Node, b \*bucket)  更新b对应的一行K-桶，确定是否将new对应的远端节点插入该行K-桶中。确保b对应的K-桶总是存储与本节点能正常通信的，而且按照通信时间进行先后排序(队首是最近联系的节点) |

|  |
| --- |
| func (tab \*Table) ping(id NodeID, addr \*net.UDPAddr) error  调用tab.net.ping方法向远端节点发送ping消息，并等待pong回复 |

|  |
| --- |
| func (tab \*Table) add(entries []\*Node)  add方法负责将形参entries切片中的节点放入Kad路由表tab相应距离的的K-桶中(若该K-桶未满) |

|  |
| --- |
| func (tab \*Table) del(node \*Node)  del方法负责将指定entry中的节点从Kad的路由表中删除(用于撤销错误的/无绑定的发现节点) |

1. nodesByDistance结构体的方法

|  |
| --- |
| func (h \*nodesByDistance) push(n \*Node, maxElems int)  负责将节点n插入到h的entries数组中，同时保证h的字段h.entries切片存储的距离h.target最近的maxElems个节点 |

1. bucket结构体的方法

|  |
| --- |
| func (b \*bucket) bump(n \*Node) bool  如果目标id的节点存在于该行K-桶中，将该节点移动到该行K-桶的最前方  移动成功返回true，未移动则返回false |