四川大学

SICHUAN UNIVERSITY

课程报告

COURSE REPORT

论文题目：Kademlia 协议与其具体 Python 实现的分析

学生姓名：赵雪峰

学生学号：2019141470426

专　　业：工程力学（力学软件实验班）

指导教师：宋万忠

学 院(系)：建筑与环境学院

目 录

第一章　RPC 的 UDP 实现

在介绍 Kademlia 协议的论文中，第 2.3 节有提到实现 Kademlia 协议时，RPC 协议应选用以 UDP 为传输层层协议的 RPC 协议。

在 RPC 的 RFC 中，并没有明确规定应使用 UDP 还是 TCP 来作为 RPC 的底层协议[1]:

> 4. TRANSPORTS AND SEMANTICS
>
> The RPC protocol can be implemented on several different transport protocols. The RPC protocol does not care how a message is passed from one process to another, but only with specification and interpretation of messages. However, the application may wish to obtain information about (and perhaps control over) the transport layer through an interface not specified in this document. For example, the transport protocol may impose a restriction on the maximum size of RPC messages, or it may be stream-oriented like TCP with no size limit. The client and server must agree on their transport protocol choices.
>
> It is important to point out that RPC does not try to implement any kind of reliability and that the application may need to be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP, then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP, it must implement its own time-out, retransmission, and duplicate detection policies as the RPC protocol does not provide these services.

但由于使用 UDP 作为底层协议需要实现超时、重传等可靠传输手段，而 TCP 可以省去这一麻烦，因此现有的网上的 RPC 实现基本上都是使用 TCP 实现的。

因此如果要实现 Kademlia 协议，则需要先实现一个以 UDP 为传输层协议的 RPC 协议。

1.1　RPC 的 UDP 实现的具体分析

该部分的实现位于\rpcudp\protocol.py中。这个协议的实现使用了umsgpack包来对报文进行编码，使用了asyncio来实现异步的 I/O 与 UDP 报文的发送接收。

在这个 RPC 协议中，由于使用了 UDP 作为其底层协议，因此这个 RPC 协议实现了一个超时的机制。

1.1.1 接受处理报文信息部分

```
    def datagram_received(self, data, addr):
        LOG.debug("received datagram from %s", addr)
        asyncio.ensure_future(self._solve_datagram(data, addr))

    async def _solve_datagram(self, datagram, address):
        if len(datagram) < 22:
            LOG.warning("received datagram too small from %s,"
                        " ignoring", address)
            return

        msg_id = datagram[1:21]
        data = umsgpack.unpackb(datagram[21:])

        if datagram[:1] == b'\x00':
            # schedule accepting request and returning the result
            asyncio.ensure_future(self._accept_request(msg_id, data, address))
        elif datagram[:1] == b'\x01':
            self._accept_response(msg_id, data, address)
        else:
            # otherwise, don't know the format, don't do anything
            LOG.debug("Received unknown message from %s, ignoring", address)
```

这一部分代码用于接收处理从传输层传输来的所有 UDP 报文，其中`datagram_received`函数为重写父类`DatagramProtocol`暴露给外界的接口，在其被调用后，会开启一个异步的 Future 来解析报文。`_solve_datagram`为异步解析所有通过`datagram_received`传来的报文所用的函数。这个函数为私有函数，不对外暴露接口。

在这个异步的函数中，首先会检查报文的长度，若报文长度小于 22 字节，则这个报文无效，丢弃这个报文。首先，将这个报文进行拆分，前 2 个字节代表这个报文的类型，第 2 个字节到第 22 个字节为报文的 ID，剩余的字节为报文的数据。首先，会检查这个报文的类型，即通过判断报文前两个字节的信息来判断这个报文为收到请求还是收到回复。若前两个字节为`00`，则表示这个报文为收到请求报文。若为`01`，则表示这个报文为收到回复报文。若是其他数据则会直接被丢弃。判断类型后则会分别交给下面的两个函数来处理数据。

1.1.2 两种不同报文的处理

```
        def _accept_response(self, msg_id, data, address):
        msgargs = (b64encode(msg_id), address)
        if msg_id not in self._outstanding:
            LOG.warning("received unknown message %s "
                        "from %s; ignoring", *msgargs)
            return
        LOG.debug("received response %s for message "
                  "id %s from %s", data, *msgargs)
        future, timeout = self._outstanding[msg_id]
        timeout.cancel()
        future.set_result((True, data))
        del self._outstanding[msg_id]

    async def _accept_request(self, msg_id, data, address):
        if not isinstance(data, list) or len(data) != 2:
            raise MalformedMessage("Could not read packet: %s" % data)
```

```
        funcname, args = data
        func = getattr(self, "rpc_%s" % funcname, None)
        if func is None or not callable(func):
            msgargs = (self.__class__.__name__, funcname)
            LOG.warning("%s has no callable method "
                        "rpc_%s; ignoring request", *msgargs)
            return

        if not asyncio.iscoroutinefunction(func):
            func = asyncio.coroutine(func)
        response = await func(address, *args)
        LOG.debug("sending response %s for msg id %s to %s",
                  response, b64encode(msg_id), address)
        txdata = b'\x01' + msg_id + umsgpack.packb(response)
        self.transport.sendto(txdata, address)
```

对回复的处理在_accept_response函数中，这个函数首先会将数据报的信息（ID与发送方的 IP 地址）解析出来，紧接着会先判断这个回复数据包的信息是否在自己期望的回复数据包中，即通过与自己存储的已发送的数据包的 ID 进行比对，若在自己期望收到的数据报的 ID 中没有这个 ID，则将其视为未知的数据报进行丢弃。若有这个 ID，则会手动取消异步 Future 的进行，并将其返回结果设置为回复数据报中的数据（视为远程调用的返回值）。并将这个数据报的超时取消，从希望获得回复列表中删除这个数据报的 ID。

对请求的处理在_accept_request函数中，收到的数据应由调用过程名与调用过程所需要的参数两部分构成，首先检查这个数据报的数据是否合法，即收到的数据报是否是list且长度为 2，紧接着会从数据中取出调用过程名与参数。接下来进行调用 RPC 的处理。发送过来的请求中包含的过程名实际上并不能直接调用，需要进行过程名的拼接，即在过程名前加上rpc_，以此来形成合法的过程名。然后通过调用self的__getattr__函数来获得一个异步的调用。期间会检查这个调用是否存在，若不存在则丢弃这个请求。若存在则会判断其是否是一个协程，需要将所有的远程调用转化为协程进行执行。紧接着需要异步执行这个协程，获取对远程调用的回复，最终将调用的结果即上述协程返回的结果进行包装发送回请求方。

1.1.3 巧妙的异步函数获取方法

```
        def __getattr__(self, name):
        """
        If name begins with "_" or "rpc_", returns the value of
        the attribute in question as normal.

        Otherwise, returns the value as normal *if* the attribute
        exists, but does *not* raise AttributeError if it doesn't.

        Instead, returns a closure, func, which takes an argument
        "address" and additional arbitrary args (but not kwargs).

        func attempts to call a remote method "rpc_{name}",
        passing those args, on a node reachable at address.
        """
        if name.startswith("_") or name.startswith("rpc_"):
            return getattr(super(), name)

        try:
```

```
            return getattr(super(), name)
        except AttributeError:
            pass

        def func(address, *args):
            msg_id = sha1(os.urandom(32)).digest()
            data = umsgpack.packb([name, args])
            if len(data) > 8192:
                raise MalformedMessage("Total length of function "
                                       "name and arguments cannot exceed 8K")
            txdata = b'\x00' + msg_id + data
            LOG.debug("calling remote function %s on %s (msgid %s)",
                      name, address, b64encode(msg_id))
            self.transport.sendto(txdata, address)

            loop = asyncio.get_event_loop()
            if hasattr(loop, 'create_future'):
                future = loop.create_future()
            else:
                future = asyncio.Future()
            timeout = loop.call_later(self._wait_timeout,
                                      self._timeout, msg_id)
            self._outstanding[msg_id] = (future, timeout)
            return future

        return func
```

这一部分巧妙地处理了 RPC 调用时的写法。当其他节点发送 RPC 给本节点时，处理这些 RPC 的过程必然是不同的，而当本机调用函数给其他节点发送 RPC 时过程都是相似的，即将过程名与参数全部打包发出，因此这里为了避免大量重复代码的使用，使用了高阶函数。若在调用`__getattr__`时名字前有_或`rpc_`，则说明需要调用的是差异定义的变量或者函数，若是其他的情况，不会引起无参数错误，而是会返回一个函数，这个函数会将本机调用其他节点的 RPC 的数据进行打包发送，并将设置一个期望回复加入到期望回复的列表中，并为其设置超时。

1.1.4 超时函数与部分类内成员

在`RPCProtocol`类中有如下成员变量：

```
        self._wait_timeout = wait_timeout
        self._outstanding = {}
        self.transport = None
```

`_wait_timeout`变量为超时时间，`_outstanding`为期望获得回复的请求列表（上文中以“期望请求”出现），在每次发送调用 RPC 请求时加入列表中，`transport`为建立的连接。

```
        def _timeout(self, msg_id):
        args = (b64encode(msg_id), self._wait_timeout)
        LOG.error("Did not receive reply for msg "
                  "id %s within %i seconds", *args)
        self._outstanding[msg_id][0].set_result((False, None))
        del self._outstanding[msg_id]
```

上述为超时函数，若一个请求以超时，则会调用这个函数，这个函数被调用时，会手动停止一个异步过程的进行，并将其结果设置为`(False,None)`，代表这个请求失败，并从

期望列表中删除这个请求的 ID。

1.2 对这个类的使用

在后文中，Kademlia类继承了这个类，并有效利用了这个类的__getattr__函数。

Kademlia 有四个基本的 RPC，即PING、FIND_NODE、FIND_VALUE、STORE，在Kademlia类中，所有的其他节点调用本节点的过程的函数均为rpc_xxx形式，而本节点调用其他节点过程的函数均为call_xxx的形式，并在call_xxx中调用了并未进行定义的一系列基本 RPC（即那些代码重复的发送数据的函数）。

第二章　Kademlia 协议实现代码细节

2.1　部分基本工具的介绍

这部分对应为代码中的`utils.py`文件。这个文件中定义了四个外部工具。

2.1.1　异步过程集执行与过程结果收集工具

这个工具可以将一个异步过程的集合全部执行并将其结果整理为一个字典。

```
async def gather_dict(dic):
    """
    异步执行一个任务字典中的所有任务，并将执行得到的结果与key对应，返回一个（key，任务结果）的字典
    """
    # 将任务字典中的任务全部取出
    cors = list(dic.values())
    # gather 函数可以用来执行一组任务执行传入的任务字典中的所有任务
    results = await asyncio.gather(*cors)
    # zip 可以将 iterable 对象中的元素一一取出，组成元组将任务结果与任务的 key 对应
    return dict(zip(dic.keys(), results))
```

这个工具被用在了 Kademlia 论文中定义的 Node Lookup 的过程上。在寻找节点（或寻找值）时会给 α 个节点发送`FIND_NODE`（或`FIND_VALUE`) 请求。这个过程应是并发且异步的，因此会产生多个协程，并将这些协程加入一个列表中，返回的字典中 key 为询问的节点 ID，而 value 则为`FIND_NODE`(或`FIND_VALUE`) 过程所返回的 k 个最近节点的信息。[2]

2.1.2　Hash 码转换工具

```
def digest(string):
    """
    将一个数据（byte类型）转化为这个字符串的sha1哈希码
    """
    if not isinstance(string, bytes):
        # 若这个字符串不是 byte 类型的字符串，则将这个字符串转化为 utf-8 编码的 byte 类型的字符串
        string = str(string).encode('utf8')
        # 返回这个 byte 类型字符串的 sha1 哈希值的二进制形式
    return hashlib.sha1(string).digest()
```

这个工具在 Kademlia 的实现中经常被使用到，用来根据一个随机数或者已经知道的节点的整形非 bit 型的 ID 来获取其 160bit 长的 hash 过的 ID。这里得到的 ID 为二进制的形式，即为一个 byte 型的串。

2.1.3　寻找最长共同前缀工具

```
def shared_prefix(args):
    """
    Find the shared prefix between the strings.
    找到一组字符串（list类型）中的最长前缀
    For instance:

        sharedPrefix(['blahblah', 'blahwhat'])
```

```
    returns 'blah'.
    """
    i = 0
    # 确定这组字符串中最短的字符串长度
    while i < min(map(len, args)):
        # operator.itemgetter(int) 返回一个函数，用于取出这个函数的参数（一个 iterable 对象）中
的第 int 个量
        # 一个巧妙的比较方法：上面的步骤后可以取出 n 个字母（n 为 args 中字符串数量，这 n 个字母为这
些字符串在
        # 第 i 个位置上的字母），将这些字母放入一个集合中，若集合的长度不为 1，则说明在这个位置上，字符
串
        # 出现了不同。
        if len(set(map(operator.itemgetter(i), args))) != 1:
            break
        i += 1
        # 返回最长前缀
    return args[0][:i]
```

这个函数多用来以寻找多个已转化为二进制形式串的 ID 的最长前缀为方法来计算一个节点与其他节点的最小深度差。因为在实现 Kademlia 协议时，过大的节点 ID 差异会导致 KBucket 所表示的二叉树的极度不平衡，会导致检索效率的下降。[2]

2.1.4 字节串转 bit 串工具

```
def bytes_to_bit_string(bites):
    """
    将bites中所有字符串的二进制形式从第三位开始分割开，
    若得到的二进制形式不满8位则补0右对齐，若满8位则直接返回
    """
    bits = [bin(bite)[2:].rjust(8, '0') for bite in bites]
    return "".join(bits)
```

多用来将一个整数形式的 ID 转化为 bit 串的形式，便于进行核心的异或算法来计算距离。

2.2 基本数据结构——节点与节点堆

这部分对应文件`Node.py`。

2.2.1 节点

```
class Node:
    """
    Simple object to encapsulate the concept of a Node (minimally an ID, but
    also possibly an IP and port if this represents a node on the network).
    This class should generally not be instantiated directly, as it is a low
    level construct mostly used by the router.
    节点的底层实现，仅从数据与基本的操作上表示一个节点，在实际中不会单独进行实例化。
    """

    def __init__(self, node_id, ip=None, port=None):
        """
        Create a Node instance.

        Args:
```

```
            node_id (int): A value between 0 and 2^160
            ip (string): Optional IP address where this Node lives
            port (int): Optional port for this Node (set when IP is set)
        """
        self.id = node_id  # pylint: disable=invalid-name
        self.ip = ip  # pylint: disable=invalid-name
        self.port = port
        self.long_id = int(node_id.hex(), 16)

    def same_home_as(self, node):
        """
        检测一个节点是否跟自己是同一个主机
        """
        return self.ip == node.ip and self.port == node.port

    def distance_to(self, node):
        """
        Get the distance between this node and another.
        获得一个节点与自己的异或距离
        """
        return self.long_id ^ node.long_id

    def __iter__(self):
        """
        Enables use of Node as a tuple - i.e., tuple(node) works.
        在迭代时返回的元组格式
        """
        return iter([self.id, self.ip, self.port])

    def __repr__(self):
        return repr([self.long_id, self.ip, self.port])

    def __str__(self):
        return "%s:%s" % (self.ip, str(self.port))
```

Node类为一个网络中节点的底层实现，仅用于存储节点的信息。这里的节点不仅包括P2P 中的 Peer 节点，还包括数据节点，每个数据都会获得一个 160 位的 hash 值，这些数据也会对应节点，但这些数据对应的节点仅仅会在部分需要计算距离的情况下被临时实例化，不会被加入路由表中。

在类的属性中，id表示一个节点的int型的 ID 值，ip则表示这个节点所代表的 IP 地址，port代表端口地址，long_id代表这个节点的int型的 ID 的十六进制形式。

same_home_as函数用来判断一个节点在网络层上是否属于同一个应用。

distance_to函数用来获得本节点与另一节点的异或距离。

2.2.2 节点堆

```
class NodeHeap:
    """
    A heap of nodes ordered by distance to a given node.
    一个根据距离排列节点的堆
    """

    def __init__(self, node, maxsize):
        """
        Constructor.
```

```
        @param node: The node to measure all distnaces from.拥有这个堆的节点
        @param maxsize: The maximum size that this heap can grow to.堆中节点数最大值
        """
        self.node = node
        self.heap = []
        self.contacted = set()
        self.maxsize = maxsize

    def remove(self, peers):
        """
        Remove a list of peer ids from this heap.  Note that while this
        heap retains a constant visible size (based on the iterator), it's
        actual size may be quite a bit larger than what's exposed.  Therefore,
        removal of nodes may not change the visible size as previously added
        nodes suddenly become visible.
        """
        # 这里的 peers 参数应为一个节点 id 的元组
        peers = set(peers)
        if not peers:
            return
        nheap = []
        for distance, node in self.heap:
            # 选出那些不在需要删除的节点数组中的节点，将它们加入到 nheap 中
            # 再用 nheap 代替 heap，以此达到删除 peers 中节点的目的
            if node.id not in peers:
                heapq.heappush(nheap, (distance, node))
        self.heap = nheap

    def get_node(self, node_id):
        """
        通过节点的id在这个节点堆中获取到一个节点，若在这个堆中无要查询的节点，则返回None
        """
        for _, node in self.heap:
            if node.id == node_id:
                return node
        return None

    def have_contacted_all(self):
        """
        检查这个堆中的节点是否都已联系过，即是否可以连接通
        """
        return len(self.get_uncontacted()) == 0

    def get_ids(self):
        """
        返回这个节点堆中所有的节点的ID
        """
        return [n.id for n in self]

    def mark_contacted(self, node):
        """
        通过id将一个节点标记为已连接过的，即将已连接过的节点加入到已连接过的节点的数组中
        """
        self.contacted.add(node.id)

    def popleft(self):
        """
```

```
        若自己保存的节点数不为空，则返回最早联系过的节点
        """
        return heapq.heappop(self.heap)[1] if self else None

    def push(self, nodes):
        """
        Push nodes onto heap.

        @param nodes: This can be a single item or a C{list}.
        """
        if not isinstance(nodes, list):
            # 如果不是一个 list 则将它转化为 list 存储的形式
            nodes = [nodes]

        for node in nodes:
            # 将在 nodes 中的所有节点存入自己的节点堆中
            if node not in self:
                distance = self.node.distance_to(node)
                # 这里可以保证节点堆有序
                heapq.heappush(self.heap, (distance, node))

    def __len__(self):
        return min(len(self.heap), self.maxsize)

    def __iter__(self):
        """
        迭代函数返回在保存的节点堆中距离自己最近的节点
        """
        nodes = heapq.nsmallest(self.maxsize, self.heap)
        return iter(map(itemgetter(1), nodes))

    def __contains__(self, node):
        """
        检测一个节点是否在自己保存的节点堆中
        """
        for _, other in self.heap:
            if node.id == other.id:
                return True
        return False

    def get_uncontacted(self):
        """
        获得没有连接过的节点，即从所有自己保存的节点中不在已连接过的节点数组中的节点
        """
        return [n for n in self if n.id not in self.contacted]
```

这是一个普通的用于存储节点的数据结构，在插入节点时便会保持节点的有序，即根据距离来对加入这个堆的节点进行排序。

需要注意的是，这个数据结构并不会被用来存放路由表中的节点信息。在路由表中存放节点的顺序应为现在距离最近访问的时间差的大小来决定，越早访问的节点越在列表的头端，越晚访问越往尾端靠近。这个数据结构仅会在进行路由工作时选择异或距离最近的点时使用。

2.3 路由表及其管理

这一部分对应代码中的`routing.py`文件，这一文件实现了协议中必须的路由表结构，用于存储节点的联系信息（ID、IP、Port 等）。

2.3.1 路由表的基本数据结构——KBucket

```
self.range = (rangeLower, rangeUpper)
self.nodes = OrderedDict()
self.replacement_nodes = OrderedDict()
self.touch_last_updated()
self.ksize = ksize
self.max_replacement_nodes = self.ksize * replacementNodeFactor
```

在 KBucket 中，基本参数与成员为：

`kszie` 规定每个 KBucket 的最大的大小。

`range` 规定这个 KBucket 所存放节点的 ID 的范围（一般为从 2^i 2^{i+1}）。

`nodes` 存放节点。`OrderedDict`是一个根据放入先后顺序来对存入内容进行排序的字典。这也就实现了对节点的，根据最近联系时间的排序。

`replacement_nodes` 临时的，用于替换 `nodes` 中的节点的节点列表。

`max_replacement_nodes` 临时替换列表的最大数量。

这一实现的具体优化措施，即加入临时替换列表（在 Kademlia 中被称作缓存）在论文中的 4.1 节被提到。

2.3.2 `KBucket` 中部分基本的函数

```
def get_nodes(self):
    return list(self.nodes.values())

    '''
移除一个节点：
如果节点在临时替换节点表中，则直接从临时表中删除
删除一个节点后从临时替换节点表中得到一个最新的节点加入bucket
'''

def remove_node(self, node):
    if node.id in self.replacement_nodes:
        del self.replacement_nodes[node.id]

    if node.id in self.nodes:
        del self.nodes[node.id]

        if self.replacement_nodes:
            newnode_id, newnode = self.replacement_nodes.popitem()
            self.nodes[newnode_id] = newnode

'''
检查一个节点是否在这个bucket的范围内
'''

def has_in_range(self, node):
    return self.range[0] <= node.long_id <= self.range[1]
```

```
    '''
    判断一个节点是否为新的节点
    通过判断它在不在节点列表中
    '''

    def is_new_node(self, node):
        return node.id not in self.nodes

     '''
    计算搜索深度本身与自己的路由表（一棵二叉树）中节点的最小深度差
    '''

    def depth(self):
        vals = self.nodes.values()
        sprefix = shared_prefix([bytes_to_bit_string(n.id) for n in vals])
        return len(sprefix)

    '''
    返回最早节点的信息
    '''

    def head(self):
        return list(self.nodes.values())[0]
```

get_nodes 获得这个 KBucket 中所有的节点。

remove_node 从这个 KBucket 中删除一个节点，若这个节点仅在替换缓存列表中，则直接从列表中删除，若在节点列表中，则在删除后从替换缓存中取出一个**最新加入**的节点放入节点列表中。

has_in_range 检查一个节点是否在这个 KBucket 所管辖的范围内（即是否应该加入这个 KBucket）。

is_new_node 检查一个节点是否在这个 KBucket 内。

depth 计算这个 KBucket 中，其他节点与本节点的最小深度差，用于实现论文中 4.2 节的检索节点的加速。

head 返回节点列表中最早加入的节点。

2.3.3 两个比较重要的函数

```
    '''
    将自己的KBucket分裂为两个，以KBucket的中间点为分割点，并将自己所记录的节点与临时节点
    串起来一起分类放入两个分裂得到的KBucket中
    返回这两个KBucket
    （数组套数组型二叉树）
    '''

    def split(self):
        midpoint = (self.range[0] + self.range[1]) // 2
        one = KBucket(self.range[0], midpoint, self.ksize)
        two = KBucket(midpoint + 1, self.range[1], self.ksize)
        nodes = chain(self.nodes.values(), self.replacement_nodes.values())
        for node in nodes:
            bucket = one if node.long_id <= midpoint else two
            bucket.add_node(node)
```

```
        return (one, two)

    '''
    添加节点
    '''

    def add_node(self, node):
        """
        Add a C{Node} to the C{KBucket}.  Return True if successful,
        False if the bucket is full.

        If the bucket is full, keep track of node in a replacement list,
        per section 4.1 of the paper.
        """
        # 如果要添加的节点在 kbucket 中，则将这个节点移动至队尾
        if node.id in self.nodes:
            del self.nodes[node.id]
            self.nodes[node.id] = node
            # 如果这个 bucket 没满，则直接加入
        elif len(self) < self.ksize:
            self.nodes[node.id] = node
        else:
            """
            OrderedDict.popitem()
            OrderedDict.popitem()有一个可选参数last（默认为True）
            当last为True时它从OrderedDict中删除最后一个键值对并返回该键值对
            当last为False时它从OrderedDict中删除第一个键值对并返回该键值对。
            不指定last（即为True）
            """
            '''
            如果bucket满或者无要添加的节点，则将新发现的节点加入临时的替换节点中
            总是将  最新  的临时替换节点加入
            如果临时替换节点已满，则将最早加入的临时替换节点弹出
            若放入临时替换节点栏则算作加入失败
            '''
            if node.id in self.replacement_nodes:
                del self.replacement_nodes[node.id]
            self.replacement_nodes[node.id] = node
            while len(self.replacement_nodes) > self.max_replacement_nodes:
                self.replacement_nodes.popitem(last=False)
            return False
        return True
```

split函数为 Kademlia 协议实现中比较核心的函数之一，路由表为一棵二叉树的 Kademlia 协议期望加快检索速度且不会占用太大空间，实现节点信息存储的优化，因此定义了 KBucket 的 k 值与这个分裂函数，在后面的有关节点加入的操作中，这个函数会用来将 KBucket 分裂为两个，以此实现一棵二叉树。

add_node函数主要优化由论文中的 4.1 来完成。普通的 Kademlia 协议在 KBucket 满的情况下收到未知节点的请求时通常会去 Ping 最早联系的节点，检查它是否仍然有效，以此实现一个 LRU 的算法。但这样会明显增加网络中的流量，为了流量控制，Kademlia 使用替换缓存（即replacement_nodes）来减少对节点的 Ping 操作，以此控制流量。

```
    def touch_last_updated(self):
        self.last_updated = time.monotonic()
```

由于 Kademlia 中要求每个 KBucket 需要至少保证每个小时一次的更新，因此需要一个

变量及函数来记录这个 KBucket 最后更新的时间。

2.3.4 路由表的实现与其基本遍历

以下是一些比较基本的函数。

```
    class RoutingTable:
    def __init__(self, protocol, ksize, node):
        """
        @param node: The node that represents this server.  It won't
        be added to the routing table, but will be needed later to
        determine which buckets to split or not.
        """
        self.node = node                                    # 这个代表的是此路由表由哪
个节点掌管
        self.protocol = protocol                            # 路由表所使用的协议
        self.ksize = ksize                                  # 每个 KBucket 的最大长
度
        self.flush()                                        # 初始化一个最初的大
KBucket

    '''
    初始化时仅有一个大的KBucket，没有被分裂过
    '''

    def flush(self):
        self.buckets = [KBucket(0, 2 ** 160, self.ksize)]


    '''
    获取所有一个小时以上都没有更新过的Bucket
    '''

    def lonely_buckets(self):
        """
        Get all of the buckets that haven't been updated in over
        an hour.
        """
        hrago = time.monotonic() - 3600
        return [b for b in self.buckets if b.last_updated < hrago]

    '''
    很明显这个函数是用来移除一个节点的
    '''

    def remove_contact(self, node):
        index = self.get_bucket_for(node)
        self.buckets[index].remove_node(node)

    '''
    很明显是用来判断一个节点是不是新节点用的
    '''

    def is_new_node(self, node):
        index = self.get_bucket_for(node)
        return self.buckets[index].is_new_node(node)

    '''
```

```
    获得一个节点会落到的范围内
    '''

    def get_bucket_for(self, node):
        """
        Get the index of the bucket that the given node would fall into.
        """
        for index, bucket in enumerate(self.buckets):
            if node.long_id < bucket.range[1]:
                return index
        # we should never be here, but make linter happy(sure python is just like
my rigid mother)
        return None
```

下面的添加节点函数实现了一些优化并修改了一些缺陷。

```
    '''
    添加一个节点
    '''

    def add_contact(self, node):
        index = self.get_bucket_for(node)
        bucket = self.buckets[index]

        # this will succeed unless the bucket is full
        if bucket.add_node(node):
            return

        '''
        在论文中的4.2节有提到，为了平衡树，二叉树的实现需要进行一些优化，若新加入的节点
        与本身节点并不在同一个Kbucket中，则只有在特定的情况下才会进行分裂，不会直接分裂，
        直接分裂导致二叉树过度不平衡，会导致算法性能下降，每次检索需要的条数增加
        '''
        # Per section 4.2 of paper, split if the bucket has the node
        # in its range or if the depth is not congruent to 0 mod 5
        if bucket.has_in_range(self.node) or bucket.depth() % 5 != 0:
            self.split_bucket(index)
            self.add_contact(node)
        else:
            # 节点加入失败会直接 ping 最早看到的节点
            asyncio.ensure_future(self.protocol.call_ping(bucket.head()))
```

在优化方面这个添加节点的函数实现了论文中的 4.2 节的检索节点的加速。在 Kademlia 论文的 2.4 节中有提到，为了解决不分裂分本身节点所在的 KBucket，带来的本身节点只能检索到最多 k 个节点的问题，会对那些不包含本身节点的满 KBucket 进行分裂。若每次都会分裂，则最终会导致这张路由表深度过深，每次检索节点都要执行算法复杂度很高的检索算法，为了减少算法复杂度，当这个二叉树的深度到达 5 时便会停止分裂，以此减少每次检索节点的时间。同时这里缓解了 2.4 节中提到的缺陷（没有完全解决）。这是一种拿空间换时间的做法，提高性能时在解决问题上进行了妥协。

```
    def find_neighbors(self, node, k=None, exclude=None):
        k = k or self.ksize
        nodes = []
        for neighbor in TableTraverser(self, node):
            # 除了被排除的节点和与被排除的节点同一主机的节点
            notexcluded = exclude is None or not neighbor.same_home_as(exclude)
            if neighbor.id != node.id and notexcluded:
```

```
                heapq.heappush(nodes, (node.distance_to(neighbor), neighbor))
            # 找到 k 个最近的邻居节点后结束循环
            if len(nodes) == k:
                break
        # 返回 k 个距离最小的节点
        return list(map(operator.itemgetter(1), heapq.nsmallest(k, nodes)))
```

find_neighbors函数是在 Kademlia 协议中另一个比较核心的函数，这个函数用来找到本节点所记录的节点中离自己最近的 k 个节点。这个函数在 Node Lookup 的过程中是核心。

下面这个类作为一个单独的迭代器，用来遍历一张路由表。

```
'''
这个类用来遍历一个路由表
'''


class TableTraverser:
    def __init__(self, table, startNode):
        index = table.get_bucket_for(startNode)
        table.buckets[index].touch_last_updated()
        self.current_nodes = table.buckets[index].get_nodes()
        self.left_buckets = table.buckets[:index]
        self.right_buckets = table.buckets[(index + 1):]
        self.left = True

    def __iter__(self):
        return self

    def __next__(self):
        """
        Pop an item from the left subtree, then right, then left, etc.
        """
        # left 成员变量用来控制先左后右的遍历顺序
        # 先判断一个 Bucket 中节点数是否为 0，若不为 0 则弹出这个 Bucket 中的节点
        # 若为 0，则进行先左 Bucket 再右 Bucket 的遍历顺序
        if self.current_nodes:
            return self.current_nodes.pop()

        if self.left and self.left_buckets:
            self.current_nodes = self.left_buckets.pop().get_nodes()
            self.left = False
            return next(self)

        if self.right_buckets:
            self.current_nodes = self.right_buckets.pop(0).get_nodes()
            self.left = True
            return next(self)

        raise StopIteration
```

2.4 Kademlia 协议的实现

基于以上介绍的 Kademlia 中的数据结构，现可以在其基础上搭建一个 Kademlia 协议。

这个 Kademlia 协议类继承于RPCProtocol, 因此只需要完成对 4 个 RPC 的定义即可，即其他节点调用本节点的 RPC 的定义于本节点调用其他节点 RPC 时的定义，以此来完成协

议的约定，使协议形成闭环。

2.4.1 处理新节点的函数

```
    def welcome_if_new(self, node):
        """
        Given a new node, send it all the keys/values it should be storing,
        then add it to the routing table.

        新节点加入时，被联系的节点需要做什么

        @param node: A new node that just joined (or that we just found out
        about).

        Process:
        For each key in storage, get k closest nodes.  If newnode is closer
        than the furtherst in that list, and the node for this server
        is closer than the closest in that list, then store the key/value
        on the new node (per section 2.5 of the paper)
        """
        if not self.router.is_new_node(node):
            return

        log.info("never seen %s before, adding to router", node)
        for key, value in self.storage:
            keynode = Node(digest(key))
            neighbors = self.router.find_neighbors(keynode)
            if neighbors:
                # 若离 key 的距离最近的节点中，新节点离 key 的距离比自己知道的离 key 比较近的最远
节点
                # 与 key 的距离小，且新节点离自己的距离比自己知道的节点离自己最近的节点与自己的距离
小
                # 则将这对 key,value 保存到新节点上
                #  （同时保证 key,value 应保存在合理范围内的条件与自己应便于查询的条件）
                last = neighbors[-1].distance_to(keynode)
                new_node_close = node.distance_to(keynode) < last
                first = neighbors[0].distance_to(keynode)
                this_closest = self.source_node.distance_to(keynode) < first
            if not neighbors or (new_node_close and this_closest):
                asyncio.ensure_future(self.call_store(node, key, value))
        self.router.add_contact(node)
```

这个函数用来实现新节点加入时的动作。

在 Kademlia 论文中，2.5 节有写道若一个新节点加入，这个新节点必须存储离其最近的 k 个节点中的至少一个节点，并且为了防止冗余的存储，本节点只会将自己存储的那些距离新节点更近的键值对发送到新节点上。同时为了保证查询的效率，键值对只会保存在比自己离那个键值对还要仅得的节点上。

2.4.2 其他节点对自己的调用的定义

```
    """
    下面这四个函数为其他节点远程调用自己的RPC时的函数
    """

    def rpc_ping(self, sender, nodeid):
```

```
        source = Node(nodeid, sender[0], sender[1])
        self.welcome_if_new(source)
        return self.source_node.id

    def rpc_store(self, sender, nodeid, key, value):
        source = Node(nodeid, sender[0], sender[1])
        self.welcome_if_new(source)
        log.debug("got a store request from %s, storing '%s'='%s'",
                  sender, key.hex(), value)
        self.storage[key] = value
        return True

    def rpc_find_node(self, sender, nodeid, key):
        log.info("finding neighbors of %i in local table",
                 int(nodeid.hex(), 16))
        source = Node(nodeid, sender[0], sender[1])
        self.welcome_if_new(source)
        node = Node(key)
        neighbors = self.router.find_neighbors(node, exclude=source)
        return list(map(tuple, neighbors))

    def rpc_find_value(self, sender, nodeid, key):
        source = Node(nodeid, sender[0], sender[1])
        self.welcome_if_new(source)
        value = self.storage.get(key, None)
        if value is None:
            return self.rpc_find_node(sender, nodeid, key)
        return {'value': value}
```

四个函数的共同特征为，都要先对节点的信息进行解析，然后检查那个节点是否在网络中为新加入的节点，紧接着才会处理自己的事情。

rpc_ping 其他节点询问本节点是否在线。

rpc_store 其他节点请求将 key、value 存储至本节点上。

rpc_find_node 其他节点询问本节点所保存的节点中距离 nodeid 最近的 k 个节点。

rpc_find_value 在本节点中寻找值，若没找到值，则调用`rpc_find_node`。

2.4.3 自己向其他节点发起 RPC 的调用

```
    """
    下面这四个函数为自己调用其他节点的RPC时的函数
    下面的这四个函数会调用到rpcudp\backslash protocol.py中的\_\_getattr\_\_函数，由于这个attr是这个类所没有的
    因此会跳转到func（）部分，即进行远程过程调用，远程调用的过程会将一个（调用函数名，参数）的包
    udp报文结构为【\backslash x00表示为request】+【message_id】+【前面打包的包】，这个函数会返回一个Future
    await函数会将这个Future执行，直到Future执行完成得到返回值
    在这个过程中会将Future与一个timeout打包
    这个Future会设置一个超时时间，用call_later进行调用，在一定时间后会调用\_timeout函数来直接将这个
    Future结束掉，设置返回的元组为（False，None），并将\_outstanding中的Future删除
    """

    async def call_find_node(self, node_to_ask, node_to_find):
        address = (node_to_ask.ip, node_to_ask.port)
        result = await self.find_node(address, self.source_node.id,
                                      node_to_find.id)
        return self.handle_call_response(result, node_to_ask)

    async def call_find_value(self, node_to_ask, node_to_find):
```

```
        address = (node_to_ask.ip, node_to_ask.port)
        result = await self.find_value(address, self.source_node.id,
                                       node_to_find.id)
        return self.handle_call_response(result, node_to_ask)

    async def call_ping(self, node_to_ask):
        address = (node_to_ask.ip, node_to_ask.port)
        result = await self.ping(address, self.source_node.id)
        return self.handle_call_response(result, node_to_ask)

    async def call_store(self, node_to_ask, key, value):
        address = (node_to_ask.ip, node_to_ask.port)
        result = await self.store(address, self.source_node.id, key, value)
        return self.handle_call_response(result, node_to_ask)
```

这部分的代码需要结合一RPC 的 UDP 实现中对__getattr__来进行分析。四个函数全部为异步的调用，首先形成一个目的节点地址信息的元组，然后通过__getattr__来实现远程调用过程，即将过程名与参数发送至对方节点，接着异步函数返回的回复进行处理。

2.4.4 对回复的处理

```
    def handle_call_response(self, result, node):
        """
        If we get a response, add the node to the routing table.  If
        we get no response, make sure it's removed from the routing table.
        这个函数用来处理远程调用后目标主机返回的信息，首先判断是否超时，若超时则
        从自己的路由表中删除这个节点，若没超时，则welcome_if_new
        """
        if not result[0]:
            log.warning("no response from %s, removing from router", node)
            self.router.remove_contact(node)
            return result

        log.info("got successful response from %s", node)
        self.welcome_if_new(node)
        return result
```

在这一层的实现中，对回复的处理仅仅在看这个回复是否有效（即是否超时）与判断与添加新节点，解析的工作交给了上面一层。

2.5 核心路由过程实现

2.5.1 基本实现

这部分的功能对应到文件crawling.py。

核心的路由过程主要为 Node Lookup 的过程。由于要尽可能多的找到其他节点的信息，因此不论是FIND_NODE操作还是FIND_VALUE操作，都会将自己收集到的其他节点中存储的节点的信息返回回来，这样不断收集其他节点信息的过程便是 Node Lookup。

对于FIND_NODE与FIND_VALUE来说，这两个不同的 RPC 有着不同的检索算法，基本的过程都是先选出 α 个节点来发送请求信息，然后根据自身的停止循环条件来判断是否停止，若条件不满足，则继续循环地对前面的 RPC 返回的其他节点保存的节点信息发送相同的请求，直到满足条件为止。

可以看出，其初始化与不满足结束条件时的动作是相同的，因此由基类中_find函数来满足。

```
class SpiderCrawl:
    """
    Crawl the network and look for given 160-bit keys.
    """

    def __init__(self, protocol, node, peers, ksize, alpha):
        self.protocol = protocol
        self.ksize = ksize
        self.alpha = alpha  # 论文中的 alpha 值
        self.node = node  # 自己
        self.nearest = NodeHeap(self.node, self.ksize)  # k 个最近的节点
        self.last_ids_crawled = []  # 上次递归询问的节点
        log.info("creating spider with peers: %s", peers)
        self.nearest.push(peers)

    async def _find(self, rpcmethod):
        log.info("crawling network with nearest: %s", str(tuple(self.nearest)))
        # 从 k 个距离自己最近的节点中选出 alpha 个节点来询问，若最近的 k 个节点中
        # 没有比最近的还要近的（即最近一次的询问并没有刷新最近节点），则直接
        # 全部询问
        count = self.alpha
        if self.nearest.get_ids() == self.last_ids_crawled:
            count = len(self.nearest)
        self.last_ids_crawled = self.nearest.get_ids()

        dicts = {}
        # "await func()" 是一个协程对象，这个函数返回的是协程
        for peer in self.nearest.get_uncontacted()[:count]:
            # 将即将被查询的节点与查询这个节点的异步过程放入一个 dict 中
            # 并将其标记为已经看到过的节点
            dicts[peer.id] = rpcmethod(peer, self.node)
            self.nearest.mark_contacted(peer)
        # 使用 gather_dict 函数异步执行对所有节点的询问，并将每个节点的询问
        # 结果对应地放入字典中
        found = await gather_dict(dicts)
        # 处理 find_node 与 find_value 得到的节点
        return await self._nodes_found(found)

    async def _nodes_found(self, responses):
        raise NotImplementedError
```

即基本的算法如下所示:

不同点就在于while跳出的条件不同，对于FIND_NODE,跳出条件为所有 k 个节点全部都已被询问过且均收到了回复，而FIND_VALUE跳出的条件为找到了想要的值。

2.5.2 针对不同 RPC 的实现

下面的代码展示了核心检索节点算法针对两种请求的不同的实现。

```
class ValueSpiderCrawl(SpiderCrawl):
    """
    这个类用来处理find_value调用时所返回的节点
    """

    def __init__(self, protocol, node, peers, ksize, alpha):
```

算法 2–1 Node Lookup 基本框架

Data: ksize,alpha,node,peers

Result: value or new routing table

pick up α nodes from its closest non-empty k-bucket;
send parallel,asynchronous RPCs to the α nodes;
while the stopping condition is still not satisfied **do**
    send RPCs to those nodes learned by previous RPCs;
    if RPCs failed to return a node any closer than the closest already seen **then**
        resend the RPCs to all of the k closest nodes it has not already queried;
    end
end

```
        SpiderCrawl.__init__(self, protocol, node, peers, ksize, alpha)
        # keep track of the single nearest node without value - per
        # section 2.3 so we can set the key there if found
        self.nearest_without_value = NodeHeap(self.node, 1)

    async def find(self):
        """
        Find either the closest nodes or the value requested.
        """
        return await self._find(self.protocol.call_find_value)

    async def _nodes_found(self, responses):
        """
        Handle the result of an iteration in _find.
        """
        toremove = []
        found_values = []
        for peerid, response in responses.items():
            response = RPCFindResponse(response)
            if not response.happened():
                # 若某个节点没有即使应答，则将其加入即将被移除的列表中
                toremove.append(peerid)
            elif response.has_value():
                # 若这个节点返回了一个有被查询的值的应答，则将这个应答的
                # 值加入列表中
                found_values.append(response.get_value())
            else:
                # 若有应答但没有查询到值，则将被查询的节点放入无值的列表中
                # 将返回的所有的节点信息存入最近节点中
                peer = self.nearest.get_node(peerid)
                self.nearest_without_value.push(peer)
                self.nearest.push(response.get_node_list())
        self.nearest.remove(toremove)

        if found_values:
            # 处理已查询到的值
            return await self._handle_found_values(found_values)
        if self.nearest.have_contacted_all():
            # not found!
            return None
        return await self.find()
```

```
    async def _handle_found_values(self, values):
        """
        We got some values!  Exciting.  But let's make sure
        they're all the same or freak out a little bit.  Also,
        make sure we tell the nearest node that *didn't* have
        the value to store it.
        检查一个得到的值是否正确
        告诉没有这个值的节点存储这个值
        """
        value_counts = Counter(values)
        # 若获得了多个值，则在这多个值中获取最普遍存储的
        if len(value_counts) != 1:
            log.warning("Got multiple values for key %i: %s",
                        self.node.long_id, str(values))
        value = value_counts.most_common(1)[0][0]
        # 若在最近的节点中有没存储这个值的节点，则将这个值存入他们的存储中
        peer = self.nearest_without_value.popleft()
        if peer:
            await self.protocol.call_store(peer, self.node.id, value)
        return value
```

这部分为针对FIND_VALUE的，这部分的跳出条件为找到值或对所有节点查询后都没有找到值。其中有两部分需要注意。第一部分为针对找到多个值时的问题，在这个实现的解决方案中，会取被存储次数最多的那个版本为信任版本。第二部分为对无值节点的记录，这个记录可以使查询过程得到一定优化。在查询结束后，本节点会根据情况将键值对存储到没有这对键值的节点上。

```
class NodeSpiderCrawl(SpiderCrawl):
    """
    用于爬取在网络上的节点
    """

    async def find(self):
        """
        Find the closest nodes.
        """
        return await self._find(self.protocol.call_find_node)

    async def _nodes_found(self, responses):
        """
        Handle the result of an iteration in _find.
        """
        toremove = []
        for peerid, response in responses.items():
            response = RPCFindResponse(response)
            # 如果在发送 find_node 请求后没有回复，则说明这个节点已经不在线，需要从最近的节点中删除
            if not response.happened():
                toremove.append(peerid)
            else:
                self.nearest.push(response.get_node_list())
        self.nearest.remove(toremove)

        # 若在最近的节点中所有的节点都被询问过了，则递归结束
        if self.nearest.have_contacted_all():
            return list(self.nearest)
        return await self.find()
```

这部分是针对FIND_NODE的，其跳出条件为已对所有 k 个节点发送了询问请求并接收

到其回复。

2.5.3 一些其他内容

```
class RPCFindResponse:
    """
    处理RPC在收到回复后回复的内容
    """

    def __init__(self, response):
        """
        A wrapper for the result of a RPC find.
        Args:
            response: This will be a tuple of (<response received>, <value>)
                      where <value> will be a list of tuples if not found or
                      a dictionary of {'value': v} where v is the value desired
        """
        self.response = response

    def happened(self):
        """
        Did the other host actually respond?
        检查被查询的节点是否响应
        """
        return self.response[0]

    def has_value(self):
        # 检测回复中是否有值
        return isinstance(self.response[1], dict)

    def get_value(self):
        # 获取回复中的值
        return self.response[1]['value']

    def get_node_list(self):
        """
        Get the node list in the response.  If there's no value, this should
        be set.
        获取在回复中的节点列表
        """
        nodelist = self.response[1] or []
        return [Node(*nodeple) for nodeple in nodelist]
```

这个类主要用于解析收到的回复数据报。

2.6 Kademlia 的顶层应用实现

有前述的各数据结构、基本操作、路由实现的基础,顶层的实现便可轻松定义。只需要调用一个可以通过一个通过 UDP 传输数据报的组件,即可完成本协议。此处对应`network.py`文件。

2.6.1 定义及其中的成员

```
class Server:
    """
```

```
    High level view of a node instance.  This is the object that should be
    created to start listening as an active node on the network.
    高层次的服务器实现，在实际运用中应实例化这个类
    """

    protocol_class = KademliaProtocol

    def __init__(self, ksize=20, alpha=3, node_id=None, storage=None):
        """
        Create a server instance.  This will start listening on the given port.

        Args:
            ksize (int): The k parameter from the paper
            alpha (int): The alpha parameter from the paper
            node_id: The id for this node on the network.
            storage: An instance that implements the interface
                     :class:`~kademlia.storage.IStorage`
        """
        self.ksize = ksize                          # KBucket 的大小，默认为 20
        self.alpha = alpha                          # 在遍历节点时 alpha 的大小，默认为 3
        self.storage = storage or ForgetfulStorage()        # 创建存放文件的存储
        self.node = Node(node_id or digest(random.getrandbits(255)))   # 节点底层实现，
        # 可自己规定节点 id，也可以获取一个随机的节点 id
        self.transport = None                # 在调用 listen 函数时可以调用
        # 一个创建 UDP 传输的对象，该对象即为 transport
        self.protocol = None                        # 使用的协议
        self.refresh_loop = None                    # 刷新路由表使用的 loop
        self.save_state_loop = None                 # 周期性保存状态使用的 loop
```

2.6.2 基本的操作

```
    async def get(self, key):
        """
        Get a key if the network has it.

        Returns:
            :class:`None` if not found, the value otherwise.
        """
        log.info("Looking up key %s", key)
        dkey = digest(key)
        # if this node has it, return it
        if self.storage.get(dkey) is not None:
            return self.storage.get(dkey)
        node = Node(dkey)
        nearest = self.protocol.router.find_neighbors(node)
        if not nearest:
            log.warning("There are no known neighbors to get key %s", key)
            return None
        spider = ValueSpiderCrawl(self.protocol, node, nearest,
                                  self.ksize, self.alpha)
        return await spider.find()

    async def set(self, key, value):
        """
        Set the given string key to the given value in the network.
        """
        if not check_dht_value_type(value):
```

```
            raise TypeError(
                "Value must be of type int, float, bool, str, or bytes"
            )
        log.info("setting '%s' = '%s' on network", key, value)
        dkey = digest(key)
        return await self.set_digest(dkey, value)

    async def set_digest(self, dkey, value):
        """
        Set the given SHA1 digest key (bytes) to the given value in the
        network.
        """
        node = Node(dkey)

        nearest = self.protocol.router.find_neighbors(node)
        if not nearest:
            log.warning("There are no known neighbors to set key %s",
                        dkey.hex())
            return False

        spider = NodeSpiderCrawl(self.protocol, node, nearest,
                                 self.ksize, self.alpha)
        nodes = await spider.find()
        log.info("setting '%s' on %s", dkey.hex(), list(map(str, nodes)))

        # if this node is close too, then store here as well
        biggest = max([n.distance_to(node) for n in nodes])
        if self.node.distance_to(node) < biggest:
            self.storage[dkey] = value
        results = [self.protocol.call_store(n, dkey, value) for n in nodes]
        # return true only if at least one store call succeeded
        return any(await asyncio.gather(*results))

def check_dht_value_type(value):
    """
    Checks to see if the type of the value is a valid type for
    placing in the dht.
    """
    typeset = [
        int,
        float,
        bool,
        str,
        bytes
    ]
    return type(value) in typeset
```

get 用来获取值，首先检查自身的存储中是否存在这个键，若存在则直接取出值，若不存在则使用寻找值得路由实现即`ValueSpiderCrawl`来从其他节点中寻找值。

set 先使用`check_dht_value_type`用来设置网络中这个 key 对应的值，首先对值进行类型的检查，接着启动一个异步的过程（寻找节点的路由实现即`NodeSpiderCrawl`）来获取网络上应该存储这对键值对的节点（初始用的节点为本节点保存的距离目的节点最近的 k 个节点），接着用异步的形式（用`gather`函数将列表中的所有协程启动）将键值对存储到这些节点中。若节点本身距离这个 key 也比较近，则也把 key 存放进本身的存储中。

2.6.3 节点的启动与停止

```
    """
    bootstrap为对节点的初始化，即将一个新的节点加入现有网络的函数
    """
    async def bootstrap(self, addrs):
        """
        Bootstrap the server by connecting to other known nodes in the network.
        Args:
            addrs: A list of (ip, port) tuple pairs.  Note that only IP
                   addresses are acceptable - hostnames will cause an error.
        """
        log.debug("Attempting to bootstrap node with %i initial contacts",
                  len(addrs))
        cos = list(map(self.bootstrap_node, addrs))
        gathered = await asyncio.gather(*cos)
        nodes = [node for node in gathered if node is not None]
        spider = NodeSpiderCrawl(self.protocol, self.node, nodes,
                                 self.ksize, self.alpha)
        return await spider.find()

    async def bootstrap_node(self, addr):
        result = await self.protocol.ping(addr, self.node.id)
        return Node(result[1], addr[0], addr[1]) if result[0] else None

    async def listen(self, port, interface='0.0.0.0'):
        """
        Start listening on the given port.
        Provide interface="::" to accept ipv6 address
        """
        loop = asyncio.get_event_loop()
        listen = loop.create_datagram_endpoint(self._create_protocol,
                                               local_addr=(interface, port))
        log.info("Node %i listening on %s:%i",
                 self.node.long_id, interface, port)
        self.transport, self.protocol = await listen
        # finally, schedule refreshing table
        self.refresh_table()

    def _create_protocol(self):
        """
        实例化一个协议
        """
        return self.protocol_class(self.node, self.storage, self.ksize)

    def stop(self):
        """
        停止节点服务的运行
        """
        if self.transport is not None:
            self.transport.close()

        if self.refresh_loop:
            self.refresh_loop.cancel()

        if self.save_state_loop:
            self.save_state_loop.cancel()
```

启动一个节点主要应分为两步，第一步启动监听，通过实例化一个传输与 Kademlia 协

议的实现来实现对传输与协议的绑定与对端口的监听。同时开始路由表的周期刷新。

第二步为为对节点的初始化：使用bootstrap函数将多个 IP 地址作为输入，向其发送PING的远程调用，检查其是否在线，若在线则加入到后续用于节点查找路由的节点列表中，接着对这些节点进行节点查找路由。这个过程可以为这个节点建立起一个路由表来。

停止一个节点的运行只需要停止传输，取消掉周期的刷新路由表与保存状态。

2.6.4 路由表的刷新

```
    def refresh_table(self):
        log.debug("Refreshing routing table")
        asyncio.ensure_future(self._refresh_table())
        loop = asyncio.get_event_loop()
        # 每 60 分钟刷新一次自己的路由表
        self.refresh_loop = loop.call_later(3600, self.refresh_table)

    async def _refresh_table(self):
        """
        Refresh buckets that haven't had any lookups in the last hour
        (per section 2.3 of the paper).
        """
        results = []
        for node_id in self.protocol.get_refresh_ids():
            node = Node(node_id)
            nearest = self.protocol.router.find_neighbors(node, self.alpha)
            spider = NodeSpiderCrawl(self.protocol, node, nearest,
                                     self.ksize, self.alpha)
            results.append(spider.find())

        # do our crawling
        await asyncio.gather(*results)

        # now republish keys older than one hour
        for dkey, value in self.storage.iter_older_than(3600):
            await self.set_digest(dkey, value)
```

实现 Kademlia 中 2.3 节所提到的对路由表的更新，首先通过get_refresh_ids从路由表的每个超一个小时未被刷新 KBucket 中获取一个随机的节点，接着对这些节点进行节点查找路由，以此实现对路由表的刷新。接着对每个自己的存储中的键值对调用set_digest函数来实现 Kademlia 论文中 2.5 节提到的对数据的重发布，以此解决数据的持久存储问题，避免其他节点下线或更近的节点替换旧节点导致的无法查询到数据的问题。

2.6.5 节点状态的保存

```
    """
    下面的三个函数为周期性地保存节点地状态，即将k值、alpha值、id、邻居信息周期性地保存为文件，
    并提供读取状态的函数
    """
    def save_state(self, fname):
        """
        Save the state of this node (the alpha/ksize/id/immediate neighbors)
        to a cache file with the given fname.
        """
```

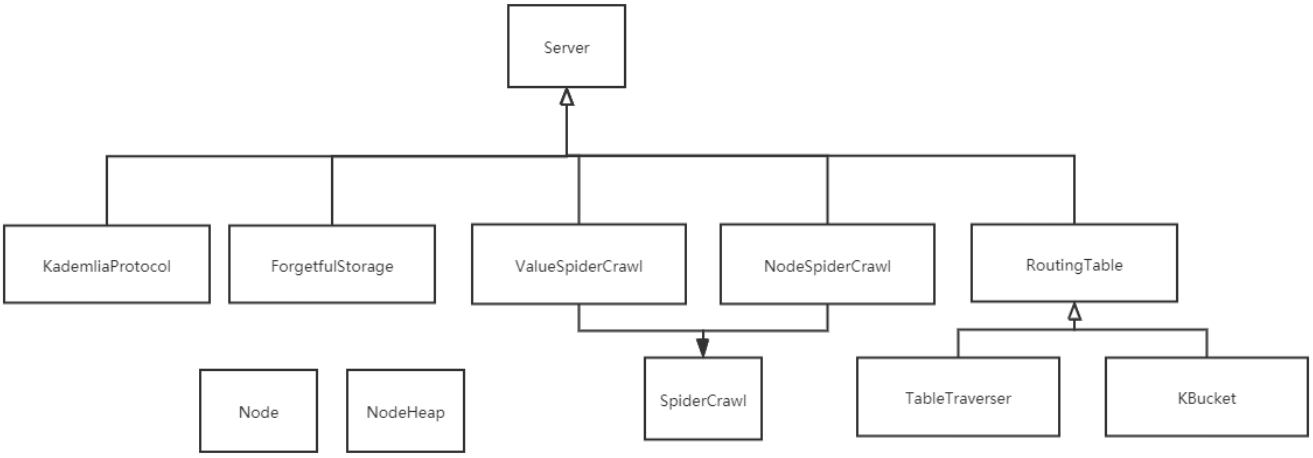
```
        log.info("Saving state to %s", fname)
        data = {
            'ksize': self.ksize,
            'alpha': self.alpha,
            'id': self.node.id,
            'neighbors': self.bootstrappable_neighbors()
        }
        if not data['neighbors']:
            log.warning("No known neighbors, so not writing to cache.")
            return
        with open(fname, 'wb') as file:
            pickle.dump(data, file)

    @classmethod
    async def load_state(cls, fname, port, interface='0.0.0.0'):
        """
        Load the state of this node (the alpha/ksize/id/immediate neighbors)
        from a cache file with the given fname and then bootstrap the node
        (using the given port/interface to start listening/bootstrapping).
        """
        log.info("Loading state from %s", fname)
        with open(fname, 'rb') as file:
            data = pickle.load(file)
        svr = Server(data['ksize'], data['alpha'], data['id'])
        await svr.listen(port, interface)
        if data['neighbors']:
            await svr.bootstrap(data['neighbors'])
        return svr

    def save_state_regularly(self, fname, frequency=600):
        """
        Save the state of node with a given regularity to the given
        filename.

        Args:
            fname: File name to save retularly to
            frequency: Frequency in seconds that the state should be saved.
                        By default, 10 minutes.
        """
        self.save_state(fname)
        loop = asyncio.get_event_loop()
        self.save_state_loop = loop.call_later(frequency,
                                               self.save_state_regularly,
                                               fname,
                                               frequency)

    def bootstrappable_neighbors(self):
        """
        Get a list of (ip, port) tuple pairs suitable for
        use as an argument to the bootstrap method.
        The server should have been bootstrapped
        already - this is just a utility for getting some neighbors and then
        storing them if this server is going down for a while.  When it comes
        back up, the list of nodes can be used to bootstrap.
        """
        neighbors = self.protocol.router.find_neighbors(self.node)
        return [tuple(n)[-2:] for n in neighbors]
```

save_state 保存节点信息的函数。

load_state 从文件中加载节点。
save_state_regularly 开启一个 loop 来周期性地保存节点的信息。
bootstrappable_neighbors 为那些已经被初始化过的节点选取一些用来初始化节点用的节点，取最新联系的 2 个节点来用于停止后的节点的初始化。

第三章　总结

我们可以看出，整个 Kademlia 协议的实现的结构是这样的：



这个协议的实现具有很强的层次性与组件性，首先这个协议的实现调用了`asyncio`库，并充分利用了这个库可以异步接受请求、传输消息与文件 I/O 的特点。其中最底层的 RPC 的 UDP 实现继承了这个库中的`asyncio.DatagramProtocol`类，意在通过继承这个类实现基于 UDP 的应用层的协议。下面的实现 Kademlia 协议的类又继承了实现基于 UDP 的 RPC 的类，这样我们可以分析出，在一层又一层的网络协议中，网络协议之间的关系应为继承关系，高层的、更具体实现的协议继承了底层协议。其次，这个协议的实现偏组件化，代码风格像 C，这个协议把基本的路由算法与具体的数据分离，将一个过程转化为了一个组件来实现各个模块之间的解耦。

参考文献

[1] SRINIVASAN R. RPC : Remote Procedure Call Protocol Specification Version 2[J/OL]. RFC 1078, 1995. https://ci.nii.ac.jp/naid/10018701068/en/.

[2] Maymounkov P, Mazières D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric[C]. in: Druschel P, Kaashoek F, Rowstron A. Peer-to-Peer Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002: 53-65.