开发过程文档-DHT 嗅探器

一． 运行及测试

参数设置：

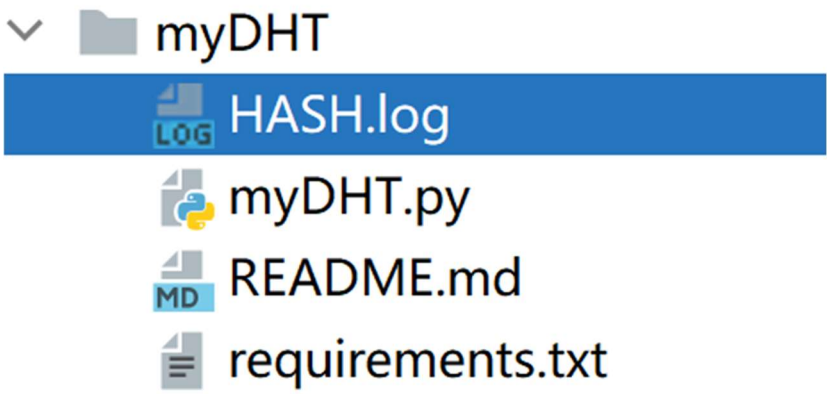
#线程数

THREAD_NUMBER = 3

#线程持续时间

SLEEP_TIME2 = 60*10

```
(venv) D:\工作学习项目\python\dht_for_test\myDHT>python myDHT.py
2021-06-23 23:51:20,776 - std_log - DEBUG - start thread 0 with port 9090
2021-06-23 23:51:21,795 - std_log - DEBUG - start thread 1 with port 9091
2021-06-23 23:51:22,811 - std_log - DEBUG - start thread 2 with port 9092
2021-06-23 23:51:37,285 - std_log - DEBUG - magnet:?xt=urn:btih:242637514ab9cb664224ce20d60a7479b1f1372c from 136.243.245.170:45139
2021-06-23 23:51:37,594 - std_log - DEBUG - magnet:?xt=urn:btih:8f15cd690be3b1f6b7c897d98891577595a6b873 from 49.51.133.101:60020
2021-06-23 23:51:39,447 - std_log - DEBUG - magnet:?xt=urn:btih:b882ead651d5a627d285fa78ca35fa75eb9b4cb8 from 223.230.166.148:6862
2021-06-23 23:51:42,690 - std_log - DEBUG - magnet:?xt=urn:btih:b7b4afb599c7a369a9570636801c1557b384450c from 91.128.181.53:64105
2021-06-23 23:51:42,986 - std_log - DEBUG - magnet:?xt=urn:btih:0a0580a32c495bb76df933572f70480fdd84053c from 192.111.154.186:59607
2021-06-23 23:51:43,066 - std_log - DEBUG - magnet:?xt=urn:btih:c6f4440007cf2240dc009a95e3235629dec6f769 from 176.9.43.177:60660
2021-06-23 23:51:45,169 - std_log - DEBUG - magnet:?xt=urn:btih:a73d6bdfef100ded23b6485b6d863eef2c28e5c9 from 185.162.184.21:54034
2021-06-23 23:51:46,597 - std_log - DEBUG - magnet:?xt=urn:btih:69e23eaf17659f7562cbc76d18fa3d0f963382e2 from 185.207.165.199:62299
2021-06-23 23:51:46,693 - std_log - DEBUG - magnet:?xt=urn:btih:f199a036498e0d607ca7967a8420a01f2538f295 from 101.32.177.105:60020
2021-06-23 23:51:46,709 - std_log - DEBUG - magnet:?xt=urn:btih:8ea873d615d9bd47f5ba47c47ab70dc1c1d8637d from 121.145.238.82:8202
2021-06-23 23:51:46,930 - std_log - DEBUG - magnet:?xt=urn:btih:0d6868cd48489a91bd0bfab79dbce9bf565b457b from 5.39.88.210:8999
```



myDHT.py HASH.log

```
1  2021-06-23 23:51:37,285 - file_log - DEBUG - magnet:?xt=urn:btih:242637514ab9cb664224ce20d60a7479b1f1372c from 136.243.
2  2021-06-23 23:51:37,595 - file_log - DEBUG - magnet:?xt=urn:btih:8f15cd690be3b1f6b7c897d98891577595a6b873 from 49.51.133
3  2021-06-23 23:51:39,448 - file_log - DEBUG - magnet:?xt=urn:btih:b882ead651d5a627d285fa78ca35fa75eb9b4cb8 from 223.230.1
4  2021-06-23 23:51:42,691 - file_log - DEBUG - magnet:?xt=urn:btih:b7b4afb599c7a369a9570636801c1557b384450c from 91.128.18
5  2021-06-23 23:51:42,987 - file_log - DEBUG - magnet:?xt=urn:btih:0a0580a32c495bb76df933572f70480fdd84053c from 192.111.1
6  2021-06-23 23:51:43,067 - file_log - DEBUG - magnet:?xt=urn:btih:c6f4440007cf2240dc009a95e3235629dec6f769 from 176.9.43.
7  2021-06-23 23:51:45,170 - file_log - DEBUG - magnet:?xt=urn:btih:a73d6bdfef100ded23b6485b6d863eef2c28e5c9 from 185.162.1
8  2021-06-23 23:51:46,598 - file_log - DEBUG - magnet:?xt=urn:btih:69e23eaf17659f7562cbc76d18fa3d0f963382e2 from 185.207.1
9  2021-06-23 23:51:46,694 - file_log - DEBUG - magnet:?xt=urn:btih:f199a036498e0d607ca7967a8420a01f2538f295 from 101.32.17
10 2021-06-23 23:51:46,709 - file_log - DEBUG - magnet:?xt=urn:btih:8ea873d615d9bd47f5ba47c47ab70dc1c1d8637d from 121.145.2
11 2021-06-23 23:51:46,932 - file_log - DEBUG - magnet:?xt=urn:btih:0d6868cd48489a91bd0bfab79dbce9bf565b457b from 5.39.88.2
12 2021-06-23 23:51:47,070 - file_log - DEBUG - magnet:?xt=urn:btih:e22381a38396a3a16d7c108fdab284c637476206 from 76.102.11
```

可正常运行，并生成相应 HASH.log 文件。

二． 问题与解决

2.1 问题-1：参考的代码为数据库方式保存数据，不论是 MySQL 还是 Redis 等数据库都需要额外配置与安装，不方便。

解决：采用 log 模块，考虑到实际上长期保存 node 并不合适，同时数据库也不便于老师检阅，采用简单方便易管理的 log 更符合需求，具体实现如下：

```
stdger = logging.getLogger("std_log")
fileger = logging.getLogger("file_log")

def initialLog():
    stdLogLevel = logging.DEBUG
    fileLogLevel = logging.DEBUG
    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
    stdout_handler = logging.StreamHandler()
    stdout_handler.setFormatter(formatter)
    file_handler = logging.FileHandler("HASH.log")
    file_handler.setFormatter(formatter)
    logging.getLogger("file_log").setLevel(fileLogLevel)
    logging.getLogger("file_log").addHandler(file_handler)
    logging.getLogger("std_log").setLevel(stdLogLevel)
    logging.getLogger("std_log").addHandler(stdout_handler)
```

2.2 问题-2：Becoding 编码的库选取 from bencode import bencode,bdecode,BTL，报错

解决：采用 import bencoder

2.3 问题-3：导入 bencoder 库后，bencode 与 bdecode 仍提示报错

解决：导包时要导 pip install bencoder.pyx 下的 bencoder

2.4 问题-4：单独定义路由表

```
class KTable():
    def __init__(self):
        self.nodes = deque(maxlen=MAX_NODE_QSIZE)
```

在 DHT 里引用时报错

解决：将其改为 DHT 里的 self.nodes

2.5 问题-5：进程开始后，嗅探一直没有结果，尝试加减线程调整网络环境无果

解决：发现问题在于

```
def receive_response_forever(self):
    self.bootstrap()
```

```
        while self.isWorking:
            try:
                data, address = self.udp.recvfrom(UDP_RECV_BUFFSIZE)
                msg = bencoder.bdecode(data)
                self.on_message(msg, address)
                time.sleep(SLEEP_TIME)
            except Exception:
                pass
```

中 time.sleep(SLEEP_TIME)处在前文参数设置中为 SLEEP_TIME = 60*10 与后面的线程持续时间重名，遂改为 SLEEP_TIME1 = 1e-5 即 0.00001 秒加以区分。

2.6 问题-6：当线程持续时间 SLEEP_TIME2 = 20，无结果

```
2021-06-24 00:24:23,020 - std_log - DEBUG - start thread 0 with port 9090
2021-06-24 00:24:24,034 - std_log - DEBUG - start thread 1 with port 9091
2021-06-24 00:24:25,044 - std_log - DEBUG - start thread 2 with port 9092
2021-06-24 00:24:46,061 - std_log - DEBUG - stop thread 0
2021-06-24 00:24:46,061 - std_log - DEBUG - stop thread 1
2021-06-24 00:24:46,061 - std_log - DEBUG - stop thread 2
```

解决：线程持续时间太短，经尝试 1min 以上可以保证获取 infohash。

2.7 问题-7：如何保证速度，是否可以提升速度?

解决：采用了多线程的方式，更快更高效，具体实现如下

```
class DHT(Thread):
    def __init__(self, bind_ip, bind_port, process_id, master):
        Thread.__init__(self)
        self.isWorking = True
        self.sfnf_thread = Thread(target=self.send_find_node_forever)
        self.rrf_thread = Thread(target=self.receive_response_forever)
        self.bst_thread = Thread(target=self.bs_timer)
        self.master = master
    def start(self):
        self.sfnf_thread.start()
        self.rrf_thread.start()
        self.bst_thread.start()
        Thread.start(self)
        return self
    def stop(self):
        self.isWorking = False

if __name__ == "__main__":
    initialLog()
```

```
    threads = []
    for i in range(THREAD_NUMBER):
        port = i + SERVER_PORT
        stdger.debug("start thread %d with port %d" % (i, port))
        dht=DHT(SERVER_HOST, port, "thread-%d" % i, Master())
        dht.start()
        threads.append(dht)
        sleep(1)
    sleep(SLEEP_TIME2)
    k = 0
    for i in threads:
        stdger.debug("stop thread %d" % k)
        i.stop()
        i.join()
        k=k+1
```

2.8 其他-1：完整的 DHT 实现思路?

答：https://github.com/bmuller/kademlia 是 DHT 完整规范的实现，在诸如 IPFS 协议的 Python 实现 https://github.com/libp2p/py-libp2p 也是采用的 bmuller 的 kademlia。同时针对 DHT 嗅探器来说，https://github.com/wuzhenda/simDHT 是比较完整的实现，它具体包含了 kdht.py/ktable.py/krpc.py，基本实现了 DHT 网络的要素，遗憾的是我并没有完全掌握它也并不能成功复现。其中对 ktable.py 的定义如下，注释由我个人添加

```
#整个 DHT 是一个哈希表,它把自己的数据化整为零分散在不同的节点里。KTable 里有 KBucket,KBucket 里有 KNode,每个 KBucket 有 K 个 KNode,在 Bittorrent(Mainline DHT)中 K=8。KTable 可以容纳 2^160 个节点,最多有 2^160/K 个 KBucket

#KNode
class KNode(object):
    def __init__(self, nid, ip, port):
        self.nid = nid
        self.ip = ip
        self.port = port
    def __eq__(self, other): #==
        return self.nid == other.nid
    def __ne__(self, other): #!=
        return self.nid != other.nid

```

```
#KBucket: K 桶可以看作结点的路由表,每个桶包含一部分的 Node ID 空间.空的路由表只有一个桶,它的 Node ID 范围从 min=0 到 max=2^160。当 Node ID 为 N 的节点插入到表中时,它将被放到 Node ID 范围在 min<=N<max 的 K 桶中.每个桶最多只能保存 K 个节点,K=8。用 KBucket 来保存与当前节点距离在某个范围内的所有节点列表比如 bucket0,bucket1,bucket2...bucketN 分别记录[1, 2), [2, 4), [4, 8), ... [2^i, 2^(i+1))范围内的节点列表。如果初始 KBucket 数量不够,则需要分裂,实现基于 BEP5 的 DHT 协议。KBucket 会进行自我组织和维护,满了将基于结点活跃性和 ping 去替换不新鲜的结点,以 15min 为 1 周期。
class KBucket(object):
    def __init__(self, min, max):
        self.min = min
        self.max = max
        self.nodes = [] #node 列表
        self.lastAccessed = time() #最后访问时间,即协议中的 lastchange
    def append(self, node):#添加 node,参数 node 是 KNode 实例
        # 如果新插入的 node 的 nid 属性长度不等于 20byte,终止
        if len(node.nid) != 20:
            return
        if node in self:#如果已在该 bucket 里,替换掉
            self.remove(node)
            self.nodes.append(node)
        else: #不在该 bucket 并且未满,插入
            if len(self) < K:
                self.nodes.append(node)
            else: #满了,抛出异常,通知上层进行拆表
                raise BucketFull
        #替换/添加 node 都要更改 bucket 最后访问时间,更新新鲜程度
        self.touch()
    def remove(self, node):#删除节点
        self.nodes.remove(node)
    def touch(self):#更新 bucket 最后访问时间
        self.lastAccessed = time()
    def random(self):#随机选择一个 node
        if len(self.nodes) == 0:
            return None
        return self.nodes[randint(0, len(self.nodes) - 1)]
    def isFresh(self):#bucket 是否新鲜
        return (time() - self.lastAccessed) > BUCKET_LIFETIME
    def inRange(self, target):#目标 node ID 是否在该范围里
        return self.min <= intify(target) < self.max
    def __len__(self):
        return len(self.nodes)
    def __contains__(self, node):
        return node in self.nodes
    def __iter__(self):#return 返回值,记住位置,下次迭代从此开始
        for node in self.nodes:
            yield node
    def __lt__(self, target):#帮助 bisect 有序序列插入,快速定位 bucket 的所在索引
        return self.max <= target
```

```
56.
57.  #K 桶满,要拆表
58.  class BucketFull(Exception):
59.      pass
60.
61.  #路由表
62.  class KTable(object):
63.      def __init__(self, nid):
64.          self.nid = nid #自身 node ID
65.          self.buckets = [KBucket(0, 2 ** 160)] #存储 Kbucket 的表,协议规定第一个 bucket 的 min=0max=2^160
66.      def bucketIndex(self, target): #定位指定 node ID 所在的 bucket 的索引
67.          return bisect_left(self.buckets, intify(target))
68.      def touchBucket(self, target):#更新指定 node 所在 bucket 最后访问时间
69.          try:
70.              self.buckets[self.bucketIndex(target)].touch()
71.          except IndexError:
72.              pass
73.      def append(self, node):#插入 node
74.          if self.nid == node.nid: return  #不存储自己
75.          #定位待插入的 node 属于哪个 bucket,若所归属的 bucket 满了又都是活跃的结点,需要拆表(拆表即算法中的拆子树)
76.          index = self.bucketIndex(node.nid)
77.          try:
78.              bucket = self.buckets[index]
79.              bucket.append(node)
80.          except IndexError:
81.              return
82.          except BucketFull:
83.              #拆表前,先确认自身 node ID 是否也在该 bucket 里,如果不在则终止
84.              if not bucket.inRange(self.nid): return
85.              #如果在 bucket 里,把 bucket 拆分成两个 bucket,平分结点
86.              self.splitBucket(index)
87.              self.append(node)
88.      def findCloseNodes(self, target, n=K):#找出离目标 node ID 最近的 K 个 node
89.          nodes = []
90.          #定位出目标 node ID 所在的 bucket
91.          if len(self.buckets) == 0: return nodes
92.          index = self.bucketIndex(target)
93.          try:
94.              nodes = self.buckets[index].nodes
95.              min = index - 1
96.              max = index + 1
97.              while len(nodes) < n and ((min >= 0) or (max < len(self.buckets))):
98.                  #如果还能往前走
99.                  if min >= 0:
100.                     nodes.extend(self.buckets[min].nodes)
```

```
                #如果还能往后走
                if max < len(self.buckets):
                    nodes.extend(self.buckets[max].nodes)
                min = min-1
                max = max+1
            num = intify(target)
            nodes.sort(lambda a, b, num=num: cmp_to_key(num ^ intify(a.nid), num ^ intify(b.nid)))#按异或值从小到大排序
            return nodes[:n] #n=K=8
        except IndexError:
            return nodes
#拆桶,桶将被分裂为 2 个新的桶,每个新桶的范围都是原来旧桶的一半,原来旧桶中的节点将被重新分配到这两个新的桶中。如果一个新表只有一个桶,这个包含整个范围的桶将总被分裂为 2 个新的桶,每个桶的覆盖范围从 0..2^159 和 2^159..2^160。index 是待拆分的 bucket 即旧桶的索引值。假设 old bucket 的 min:0,max:8.拆分该 old bucket,分界点是 4,改 old bucket 的 max=4,min 仍为 0。创建一个新的 bucket,new bucket 的 min=4max=8。根据的 old bucket 中的各个 node 的 nid,判断在哪个 bucket 的范围里,装到对应 bucket 里。new bucket 的索引值在 old bucket 后面即 index+1,把新的 bucket 插入到路由表。
    def splitBucket(self, index):
        old = self.buckets[index]
        point = old.max - (old.max - old.min) / 2 #分界点为一半
        new = KBucket(point, old.max)
        old.max = point
        self.buckets.insert(index + 1, new) #index+1
        for node in old.nodes[:]:
            if new.inRange(node.nid): #在新桶的结点移过去
                new.append(node)
                old.remove(node)
    def __iter__(self):
        for bucket in self.buckets:
            yield bucket
    def __len__(self):
        length = 0
        for bucket in self:
            length += len(bucket)
        return length
    def __contains__(self, node):
        try:
            index = self.bucketIndex(node.nid)
            return node in self.buckets[index]
        except IndexError:
            return False
```

2.9 其他-2：在优化上有没有更好的思路?

答：如果说最原始的嗅探器是单线程的，那么多线程与异步可以对其进行优化。异步和多线程两者都可以达到避免调用线程阻塞的目的，从而提高软件的可响应

性。在我的作业中我采用了多线程并成功实现了提速，但可惜的是在尝试使用异步时出现了错误，完全仿照他人的代码也无法正常获取 infohash。对于 Python 来说不像 JS 天然带有异步，需要额外的模块辅助，如 asyncio、Twisted、Gevent。在 https://github.com/whtsky/maga 采用的就是 asyncio，在 https://github.com/wuzhenda/simDHT 则采用了 twisted。但是，由于 Twisted、Gevent 实际上是 Python 2 时代的产物，且太重回调太多，基于的协程的 asyncio 在 Python 3.4 被引入到标准库后已经不太适用 Python3 时代了。在我的个人尝试中可能也是由于 Python3 与 Gevent 和 Twisted 不兼容所导致的失败。

2.10 其他-3：获取的 infohash 并非是可用的，原因?

答: 在说明文档中提到，get_peers 和 announce_peer 请求包含了所需的 infohash 信息，我们也是通过 get_peers 和 announce_peer 来获取 infohash。但是，get_peers 中包含的 infohash 对应的种子可能已经失效或者难连接上，事实上此时对方也是在查找对应 infohash 的种子文件，这里获取的 infohash 质量较差。当收到 announce_peer 消息时，这里携带的 infohash 的质量更高，因为它表示控制该节点的下载者开始下载资源了，当前对方正在指定端口下载该种子文件的 metadata 信息。

2.11 其他-4：如何利用 infohash 真正意义上的下载资源?

答：只要找到一个种子的下载者(peer)就可以使用 BEP-9: Extension for Peers to Send Metadata Files 拓展协议从该下载者(peer)处下载种子元信息(info)，同样可以使用元信息(metadata)的 infohash 来验证该信息的真实性，当然也可以从它那下载资源。