分布式哈希表的简单实现

1140310118 张义策

目 录

一、 原理简述

(一)分布式哈希表

分布式哈希表（DHT）是一种分布式存储方法。每个客户端负责一个小范围的路由，并存储一小部分数据，从而实现整个 DHT 网络的寻址和存储。

(二)一致性哈希

一致性哈希通常被认为是 DHT 的一种实现。

一致性哈希算法在 1997 年由麻省理工学院提出。其指出在动态变化的 Cache 环境中，哈希算法应该满足以下 4 个适应条件。

1、平衡性(Balance)

平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都能得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)

单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

简单的哈希算法往往不能满足单调性的要求，如最简单的线性哈希：

$$x \rightarrow (ax + b) \bmod (P)$$

在上式中，P 表示全部缓冲的大小。不难看出，当缓冲大小发生变化时(从 P1 到 P2)，原来所有的哈希结果均会发生变化，从而不满足单调性的要求。

哈希结果的变化意味着当缓冲空间发生变化时，所有的映射关系需要在系统内全部更新。而在 P2P 系统内，缓冲的变化等价于 Peer 加入或退出系统，这一情况在 P2P 系统中会频繁发生，因此会带来极大计算和传输负荷。单调性就是要求哈希算法能够避免这一情况的发生。

3、分散性(Spread)

在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希 过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区 中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法 应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

4、负载(Load)

负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

(三) Chord

1.简介

Chord 作为一致性哈希的一种实现，在 2001 年由麻省理工学院提出。Chord 要解决的核心问题是：如何在 P2P 网络中找到存储特定数据的节点；特别地，Chord 在查询节点时做了优化，使得查询需要的跳数由$O(n)$减少为$O(\log(n))$。

2.hash 表分布规则

a) 节点的 ID(NID)为节点 IP 地址的 hash 值，即$NID_i = hash(IP_i)$。
b) 资源<K,V>的 ID 为关键字 K 的 hash 值，即$KID_i = hash(K_i)$。
c) 节点按 ID 值从小到大顺序排列在一个逻辑环上。
d) Successor(K)为从 K 开始顺时针方向距离 K 最近的节点。<K,V>存储在 Successor(K)上。

3.应用程序接口

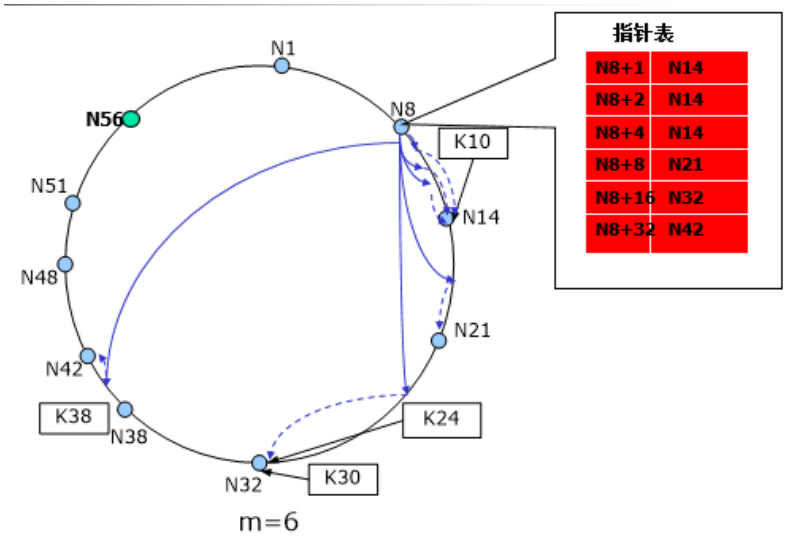
Chord 向外提供 5 个应用程序接口，如下

- Insert (K, V)　　将<K,V>存放到 ID 为 Successor(K)的节点上。
- Lookup (K)　　根据 K 查询相应的 V。
- Update (K, new_V)　　根据 K 更新相应的 V。
- Join (NID)　　节点加入。
- Leave ()　　节点主动退出。

4.查询过程

(1) Finger Table

每个节点维护一个查询表(Finger Table)。对于 ID 为$\mathrm{NID_i}$的节点的 Finger 表$Finger_i$，我们假设它的大小为 n。则此$Finger_i$中的第 j (j∈[0,n-1]) 条数据为$Successor(NID_i + 2^j)$。



(2) 查询过程

查询关键字 k 时，我们首先判断 KID 是否落在当前节点 A 和它的 Successor(A)之间。如果满足，则记录<K, V>存放在 Successor(A)上；否则，转发关键字 K 至当前 Finger 表中 K 的前驱节点。同理，插入和更新，也将按照此步骤完成。

Finger 表中有$O(\log N)$个节点，查询大约经过$O(\log N)$跳。

5.网路波动(Churn)

Churn 由节点的加入、退出和失效引起。

每个节点都周期性运行探测协议来检测新加入节点和退出/失效节点，从而更新自己的指针表和后继节点的指针。

(1) 节点加入

a) 新节点 N 事先知道某个或者某些节点。通过这些节点，节点 N 初始化自己的指针表。

b) 其他节点运行探测协议后，新节点 N 将被反映到相关节点的指针表和后继节点指针中。
c) 新结点 N 的第一个后继结点将其维护的小于 N 节点的 ID 的所有 K 交给该节点维护。

(2) 节点退出
a) 当 Chord 中某个结点 M 退出/失效时，所有在指针表中包含该结点的结点将相应指针指向大于 M 结点 ID 的第一个有效结点即节点 M 的后继节点
b) 为了保证节点 M 的退出/失效不影响系统中正在进行的查询过程，每个 Chord 节点都维护一张包括 r 个最近后继节点的后继列表。如果某个节点注意到它的后继节点失效了，它就用其后继列表中第一个正常节点替换失效节点

二、 实现细节

(一) 节点地址

为了能够在一台机器上实现 DHT，在这里使用不同的端口号表示不同的地址。由于发送消息和等待消息不能使用同一个端口，因此在这里，消息中常附带发送方的地址信息。

(二) 接受消息

对于一个节点，使用一个 socket 接受其他节点发来的消息，并做出响应。称这个 socket 为 recv_socket。recv_socket 绑定一个给定的端口号 port，('127.0.0.1', port)即这个节点的地址。

下表为 recv_socket 接受消息的语法和语义以及相应。

序号	语法	语义
1	'table', source_addr	来自 source_addr 对 finger 表的请求
2	'request kvs', source_addr	来自 source_addr 对记录的请求
3	'kvs'	当前节点将收到一些记录
4	'leave', source_addr, successor_addr	来自 source_addr 的离开消息，source_addr 的后继节点的地址为 successor_addr
5	'insert', k, v	收到关于<K,V>插入的询问
6	'insertkv', k, v	收到在本地直接插入<K,V>的通知
7	'look up', source_addr, k, v	收到关于查找 K 的询问
8	'look up k', source_addr, k, v	收到在本地直接查询<K,V>的通知，并将结果发给 source_addr
9	'kv'	收到接受查询结果<K,V>的通知

(三) 定时更新 Finger 表

节点每经过大小为$t_{interval}$的时间，更新自己的 Finger 表。具体步骤为，向自己 Finger 表中的所有地址，请求它们的 Finger 表，收到它们的 Finger 表之后，判断其中的每一个地址是否应属于自己的 Finger 表，属于则插入到相应的位置。此外，当节点接收到对

Finger 表的请求时，除了发送自己的 Finger 外，还需判断消息中附带的地址判断是否应该属于自己 Finger 表中，属于则插入到相应位置。

```
>>  ‘table’ , my_addr
<<  table.to_string()
```

在这里，有一个问题需要细化，那就是如何判断一个地址是否属于自己的 Finger 表？

首先，计算出此地址(_addr)的_NID，并计算出差值$d = (_NID - my_NID)\%N$，然后计算出此地址在 Finger 表中的位置$index = [log_2 d]$。假设 Finger_table[index]=None，即 index 位置没有元素，则插入_addr ；否则，比较 Finger_table[index]与_NID 的大小，小的应该在 Finger 表中。

(四)　节点的加入与离开

加入 节点 A 想加入网络时，只需将网络某个节点 B 的地址插入到自己的 Finger 表中，根据 Finger 定时更新的规则，一个或多个更新周期($t_{interval}$)之后，节点 A 就成功地加入网络。多个更新周期之后，节点 A 向后继节点 C，请求现存储在 C 中但属于 A 的记录<K,V>。

```
>>  ‘request kv’ , my_addr
<<  k1,v1
<<  k2,v2
 …
<<  END
```

离开 节点 A 离开网络时，首先向自己 Finger 表中的所有地址，发送消息，通知它们自己即将离开，并附上自己和后继节点的地址。相应地，节点 B 接受消息之后，将 Finger 表 A 的地址修改为 A 后继节点的地址。此外，A 将存储在节点 A 上的所有<K,V>记录发送给自己的后继节点。

```
>>  ‘leave’ , my_addr , successor_addr
```

```
>>  ‘kvs’
>>  k1,v1
>>  k2,v2
  …
<<  ‘leave ok’
```

(五)　查询

查询分为两个阶段：

1) 我们首先判断 KID 是否落在当前节点 A 和 Successor(A)的 ID 之间。如果满足，则记录<K, V>存放在 Successor(A)上，向 Successor(A)发送查询请求，结束；否则，进入下一步。在这里，我们需要明确一个问题，如何判断 KID 落在当前节点 A 和 Successor(A)之间：判断 KID 与 Successor(A)的距离是否小于 A 与 Successor(A)的距离，即

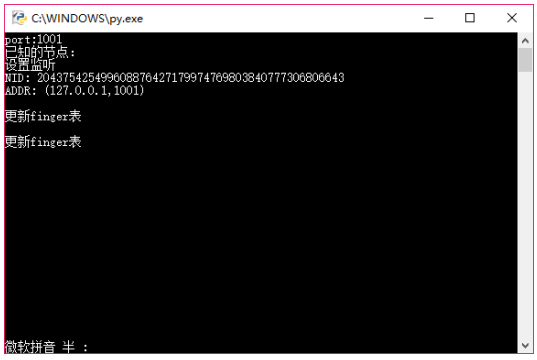
```
if d(Successor(A), K)< d(Successor(A), A):
    >>  'look up k', my_addr, K
```

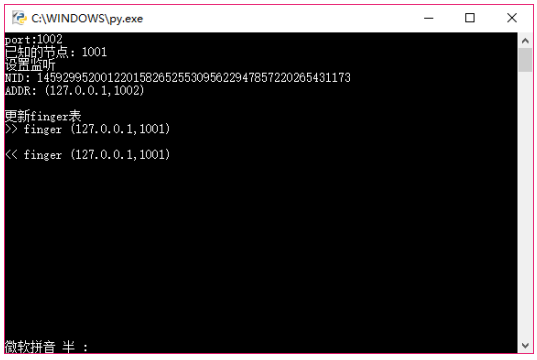
其中，d(A, B)=(hash(A)-hash(B))%N。

2) 转发关键字 K 至当前 Finger 表中 K 的前驱节点。如何找到 K 在 Finger 表中的前驱节点：对于 Finger 表所有元素N_i，计算$d_i=d(K,N_i)$。d_i最小的节点即是所求的前驱节点。

三、 运行结果

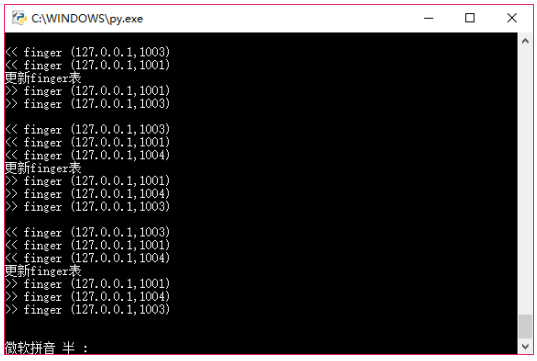
新建节点



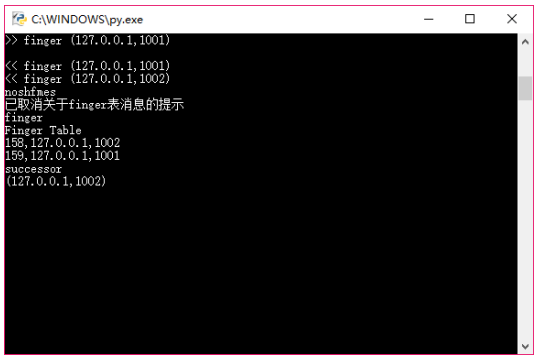


新建了 4 个节点之后。

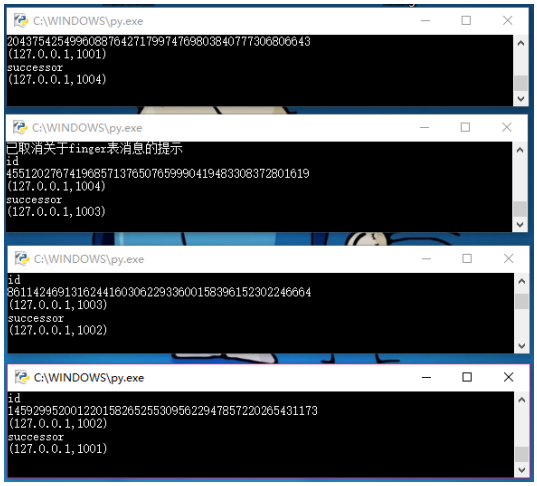
按时更新 Finger 表



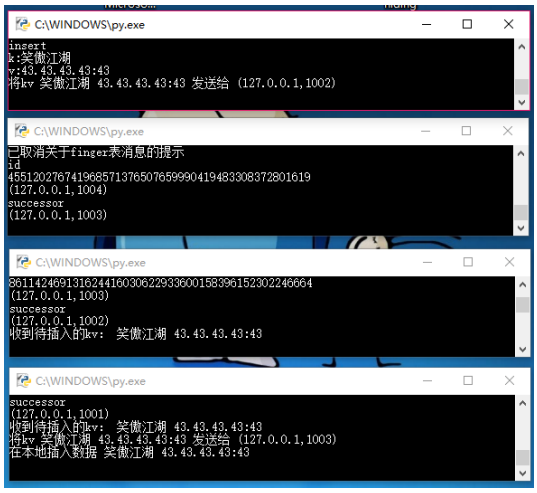
查看 Finger 表和后继节点



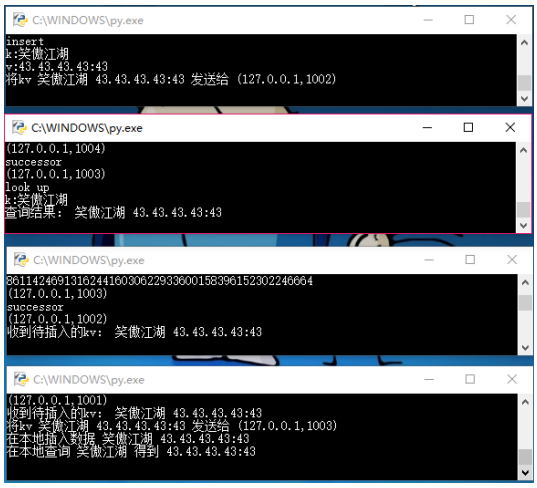
我们可以看出节点之间形成了一个环（1001-1004-1003-1002-1001）。



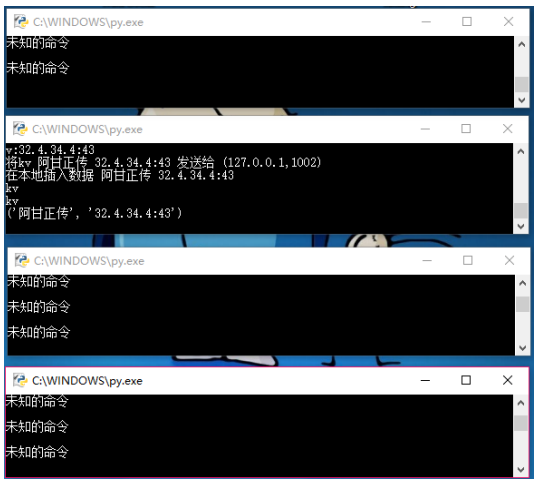
插入数据，在节点 1001 插入数据，经多次转发后，在节点 1002 插入。



在节点 1004 查询，并得到结果



查看节点 1004 的全部记录



节点 1002 离开，将所有纪录交给 1001