Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
论文阅读笔记

陈超　201922080611

July 26, 2021

一致性哈希

一致哈希算法应该满足的条件：

- 平衡性：哈希的结果尽可能分布到所有的缓存中去，这样可以使得所有的缓冲空间都得到利用。
- 单调性：指如果已经有一些内容通过哈希分派到了相应的缓存中，又有新的缓存加入到系统中。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存中去，而不会被映射到旧的缓存集合中其它缓冲区。

Chord

基本原理

- 采用环型拓扑
- 应用接口
  - Insert(K,V) (将 <K,V> 对存放到节点 ID 为 Successor(K) 上)
  - Lookup(K) (根据 K 查询对应的值)
  - 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)

Chord: Hash 表分布规则

- Hash(节点的 IP 地址) $\Rightarrow$ m 位节点 ID (表示为 NID)
- Hash(内容的关键字) $\Rightarrow$ m 位 K (表示为 KID)
- 节点按 ID 从小到大顺序排列在一个逻辑环中
- <K,V> 存储在后继节点上
  - Successor(K): 从 K 开始顺时钟方向距离 **K** 最近的节点。

Chrod 的定义 I

- m be the number of bits in the key/node identifiers. (key 标识符).
- Each node, n, maintains a routing table with (at most) m entries(表项), called the finger tables.
- The i^{th} entry in the table at node n contains the identity of the first node, **s, that succeeds n**(在 n 的后代) by at least 2^{i-1} on the identifier circle.
- $s = successor(n + 2^{i-1})$, where $1 \leq i \leq m$。
- call node s the i^{th} finger of node n, and denote by n.finger[i].node。
- A finger table entry includes both the Chrod identifier and the IP address(and port number) of the relevant node.
- Note the first finger of n is its immediate successor(直接后继者) on the circle.

Chrod 的定义 II

Notation	Definition	
finger[k].start	$(n+2^{i-1}) \mod 2^m, 1 \leq k \leq m$	在 hash 圈中, 至少距离 n 节点 2^{i-1} 的最近节点 s
finger[k].interval	$[finger[k].start, finger[k+1].start)$	在 hash 圈中, 距离 n 节点大于等于 2^{i-1} 距离, 并且小于 2^i 距离
.node	first node $\geq$ n.finger[k].start	距离节点 n 至少 2^{i-1} 距离的最近节点
successor	the next node on the identifier ciriclefinger[1].node	
predecessor(前驱)	**the previous node on the identifer circle**	

Table: Definition of variables for node n, using m-bit identifiers

Chrod 的定义 III

finger 表项的计算为:

$$\text{finger}[k].\text{start} = (n + 2^{k-1}) \mod 2^m$$

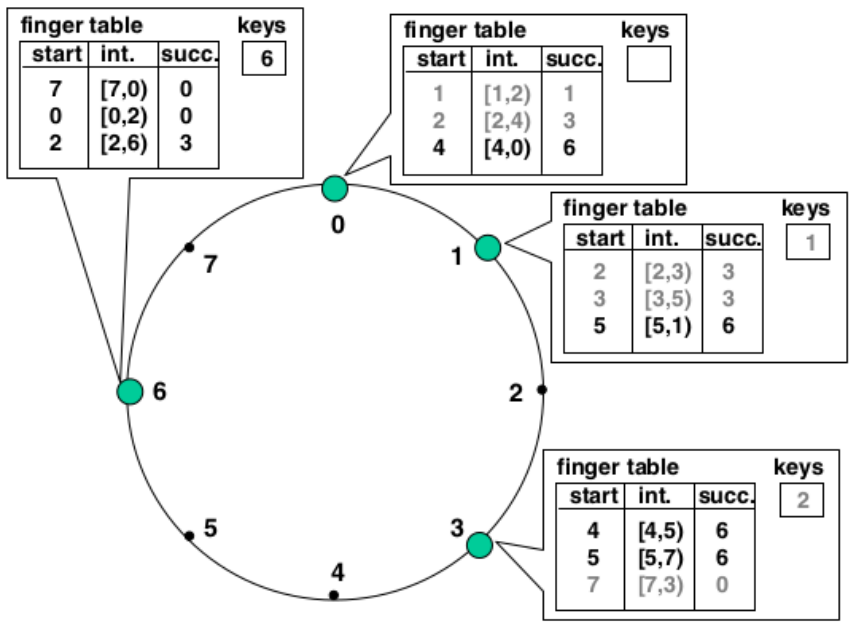


Figure: Finger tables and key locations for a net with node 1,2,and 3(只有三个节点), and keys 1,2,and 6.

Chrod 的定义 IV

finger Scheme

This scheme has two important characteristics.

- each node stores information about only a small number of other nodes, and knows more **about nodes closely** following it on the identifier circle than about nodes farther away.
- a node's finger table generally does not contain enough information to determine the successor(后继值, 实际的节点) of an aribtrary key。

Chrod 的定义 V

FAQ: What happens when a node n does not know the successor of a key k(不知道 key k 的后继节点)?

If n can find a node whose ID is closer than it own to k,(如果 n 能够知道另外一个节点 p，比 n 更靠近 key k)。that node will know more about the identifier circle in the region of k (关于 key k 的环形区域) than n does.

Thus n searches for the node j whose ID **most immediately precedes k(最接近 key k)**, and asks j for the node it knows whose ID is closest to k . By repeating this process, n learns about nodes with IDs closer and closer to k.

- n searches for the node j whose ID **most immediately precedes k**。
- asks j for the node it knows whose ID is closest to k。
- **repeating** this process。

Chrod 的定义 VI

find_successor works by 目的：finding the **immediate predecessor** node of the desire identifier.

- Remote calls and variable references are preceded by the remote node identifier
- Local variance references and procedure calls **omit** the local node (本地调用，不加前缀)

Chrod 的定义 VII

```
1        //ask node n to find id's successor (id可能是key)
2        n.find_successor(id)
3           //调用本地n的方法，find_predecessor
4           n1 = find_predecessor(id);
5           return n1.successor;
6        //ask node n to find id's
7        n.find_predecessor(id)
8            n1 = n;
9            //id 在这个左开右闭内
10           while(id \notin (n1, n1.successor])
11              n1 = n1.closest_preceding_finger(id);
12          return n1;
13       //return closest finger preceding it
14       n.closest_preceding_finger(id)
15           //顺时钟，
16           //为什么，i从m开始？
17           //有点像skiplist查找，总是从最高的level开始查找，
18           // 找到距离id最近的节点。
19          for i = m downto 1
20             if(finger[i].node \in (n,id)) //开区间 (n,id)
21               return finger[i].node;
22          return n;
```

Chrod 的定义 VIII

Node joins 节点加入

The main challenge in implementing these operations is **preserving the ability to locate every key in the network.** To achieve this goal, Chord needs to preserve two invariants:

- Each node's successor is correctly maintained.
- For every key k, node successor(k) is responsible for k.

To simplify the join and leave machanisms, each node in Chord maintains a predecessor pointer(前驱点). A node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node. and can be used to walk counterclockwise around the identifier circle.

To preserve the invariants stated above. Chord must preform three tasks when **a node n joins the network.**

1. Initialize the predecessor and fingers of node n.
2. Update the fingers and predecessors of existing nodes to reflect the addtition of n.(反映 n 节点的加入)

Chrod 的定义 IX

FAQ: 新节点如何加入 chord 环?
分为两个部分,

- Initializing fingers and predecessor.(初始化新加入节点的 fingers 表和它的直接前驱)
- Updating fingers of existing nodes.(更新之前存在的节点的 fingers 表)

Chrod 的定义 X

Initializing fingers and predecessor

两个循环不变式

- Each node's successor is correctly maintained.
- For each key k, node successor(k) is responsible for k.

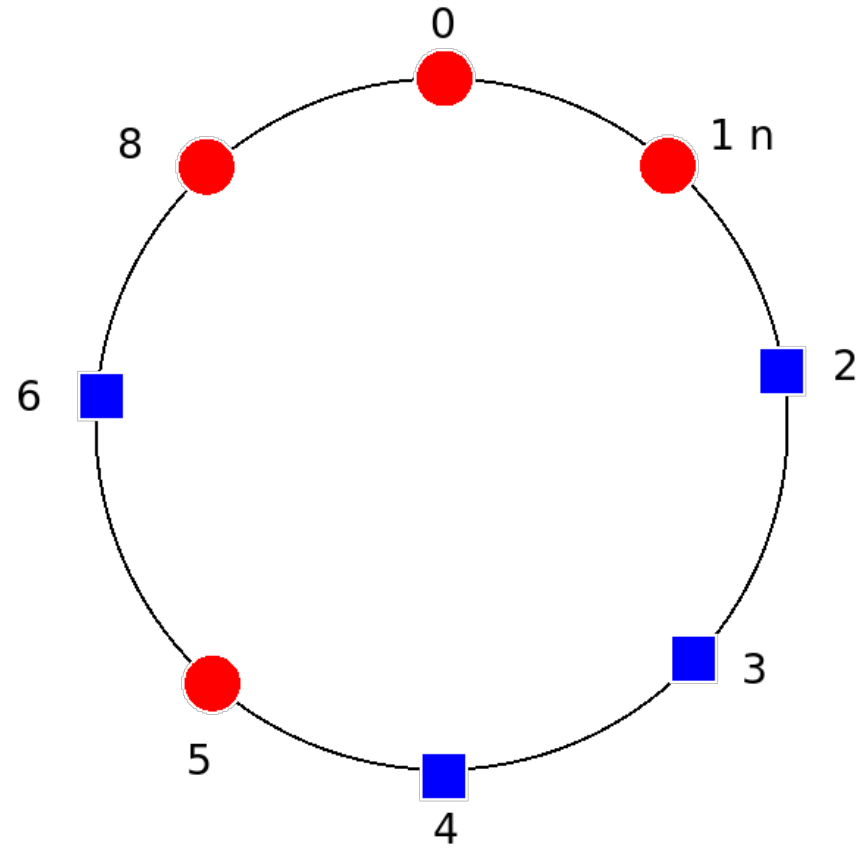
FAQ: 如何找到它的位置呢?
联系任意一个在 chrod 环的节点，进行find_successor操作。而一个 key 负责响应的节点，必须为大于等于 hash(key) 的最小节点。找到这个successor的前驱**precessor** 必为新节点 n 的前驱。

Chrod 的定义 XI

Initializing fingers and predecessor

FAQ: 如何更新 n 的 fingers 表呢?
Naively performing find_successor for each of the m finger entries. 但是, 可能 chrod 中节点是不仅稀疏的, 所有我们可以跳过某些 find_successor. 比如在 i^{th} finger entry 的 node, 可能也是 $i+1^{th}$ finger entry 的节点。(.node first node $\geq$ n.finger[k].start)

Chrod 的定义 XII



n.finger[1].interval = [2,3), n.finger[1].node = 5
n.finger[2].interval = [3,5), n.finger[2].node = 5
n.finger[3].interval = [5,1), n.finger[3].node = 1

Figure: n fingers node 示意图

Chrod 的定义 XIII

代码：

```
1       #define successor finger[1].start
2        //node n 加入chord环
3       // n1 是chord环中任意的节点
4       n.join(n1):
5        if(n1)
6            init_finger_table(n1);
7            update_others();
8       else //n is the only node in the network
9            for i = 1 to m
10               finger[i].node = n;
11           predecessor = n;
12      //初始化n中的fingers表项
13      n.init_finger_table(n1):
14      finger[1].node = n1.find_successor(finger[1].start); //找到finger[1]表项中的节点
15      predecessor = successor.predecessor; //找到节点n的直接前驱
16      //找到n中的fingers表项
17      for i = 1 to m - 1:
18         //chord是稀疏的
19         if finger[i+1].start \in [n, finger[i].node)
20           finger[i+1].node = finger[i].node;
21        //查找
22         else
23          finger[i+1].node = n1.find_successor(finger[i+1].start);
```

Chrod 的定义 XIV

Updating fingers of existing nodes

假设 n 是节点 p 中 finger[i] 的 node，我们需要满足如下两个条件：

- p precedes n by at least 2^{i-1}. (ditance(p, n) $\geq 2^{i-1}$)
- the i^{th} finger of node p successes n.

Chrod 的定义 XV

一个例子，p(当前为 1) 节点，原来的 p.finger[2].node = 5，其中 p.finger[2].interval = [3,5)，现在，新节点 3 加入 chord 环了，更新 p.finger[2].node = 3，满足条件：

- 当前 $i = 2$，新加入的节点为 n == 3。
- p.finger[2].node 是大于等于 p.finger[2].start 的最近编号节点。
- 条件一：distance(p, n) $\geq 2^{i-1}$，同时 p 的 predecessor 也满足，递归下去。
- 条件二：p.finger[2].node == 5 > (n = 3)。

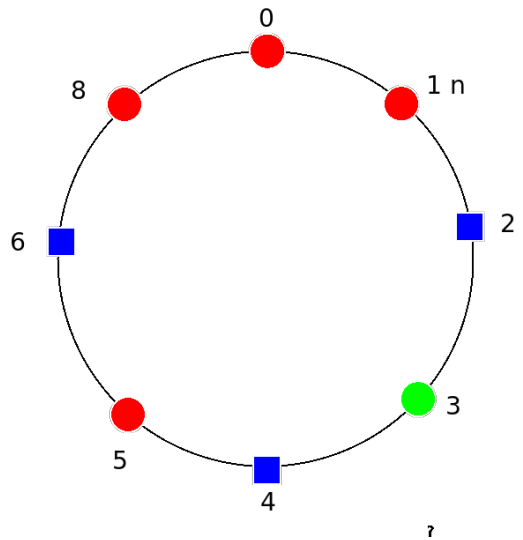


Figure: Updating fingers of existing nodes

Chrod 的定义 XVI

算法代码：

```
1         //如果n节点成为p节点中finger[i].node
2         //必须满足如下两个条件：
3         // 1. n - p >= 2^{i-1}
4         // 变化一下 n - 2^{i-1} >= p
5         // 找到小于等于n - 2^{i-1}的最大节点，也就是(n - 2^{i-1})的predecessor。
6         // 2. p.finger[i].node > n
7
8        //update all nodes whose finger tables should refer to n
9        n.update_others() {
10         for i 1 to m :
11            //find the last node p whose i^{th} finger table might be n
12            //找到满足条件一
13            p = find_predecessor(n - 2 ^{i-1})
14            p.update_finger_table(n, i)
15      }
16      //if s is i^{th} of n, update n's finger table with s
17      n.update_finger_table(s, i) {
18        //满足条件二
19        if( s \in (n, n.finger[i].node)){
20            n.finger[i].node = s;
21            p = prdecessor; //get first node preceding n (n的前驱节点，同样满足条件一，试图更新)
22            p.update_finger_table(s, i);
23        }
24     }
```

Chrod 的定义 XVII

A base "stablilization" protocol is used to keep nodes' **successor pointers** up to date, which is sufficient to guarantee correctness of lookups. Those successor pointers are then used to verify and correct finger table entries, which allows these lookup to be fast as well as correct.
our stablilization scheme guarantees to add nodes to a Chord ring in a way

- reachability of existing nodes, even in the fact of concurrent joins and lost and reordered messages.
- won't correct a Chord system that has split into multiple disjoint cycles, or a single cycle that loops multiple times around the identifier space.

Chrod 的定义 XVIII

FAQ：如何实现 stablilization 呢?

- Every node runs **stabilize** **periodically** (this is how newly joined node are noticed by the network)， when node n runs **stabilize**,
  - it asks n's successor for the successor's predecessor p。(向 n 的后继节点，询问其前驱节点 p)
  - decides whether p should be n's successor instead. （前驱 p 能否成为 n 的新的后继，离 n 更近!）
- stabilze also nitifies node n's successor of n's existence, (通知 n 的后继节点，它的存在性) giving the successor the change to change its predecessor to n.

Chrod 的定义 XIX

一个 stablize 过程的例子：

1. 在节点 3 加入前，节点 1 的 successor 为节点 5，节点 5 的前驱为 1。
2. 当节点 3 加入时，**节点 3 的 successor 为节点 5。**
3. 节点 3 运行 stabilize，节点 3 的 successor 节点 5 运行 notify，**节点 3 认为节点 3 可能是节点 5 的前驱节点。**
4. 节点 5 检查节点 3 是否是它的新前驱，如果节点 3 比原来的前驱节点 5 更近。就认为，节点 3 是当前节点 5 的新前驱。
5. 节点 1 运行 stabilize，询问节点 5 的当前前驱（为节点 3）。检查节点 3 是否比原来的后继节点 5 更近，设置节点 1 的后继节点为 3。节点 1 的新后继节点 3 运行 notify，**节点 1 认为节点 1 可能是节点 3 的前驱节点。**
6. 节点 3 检查节点 1 是否是它的新前驱，如果新前驱更近。就认为，节点 3 是当前节点 1 的前驱。

Chrod 的定义 XX

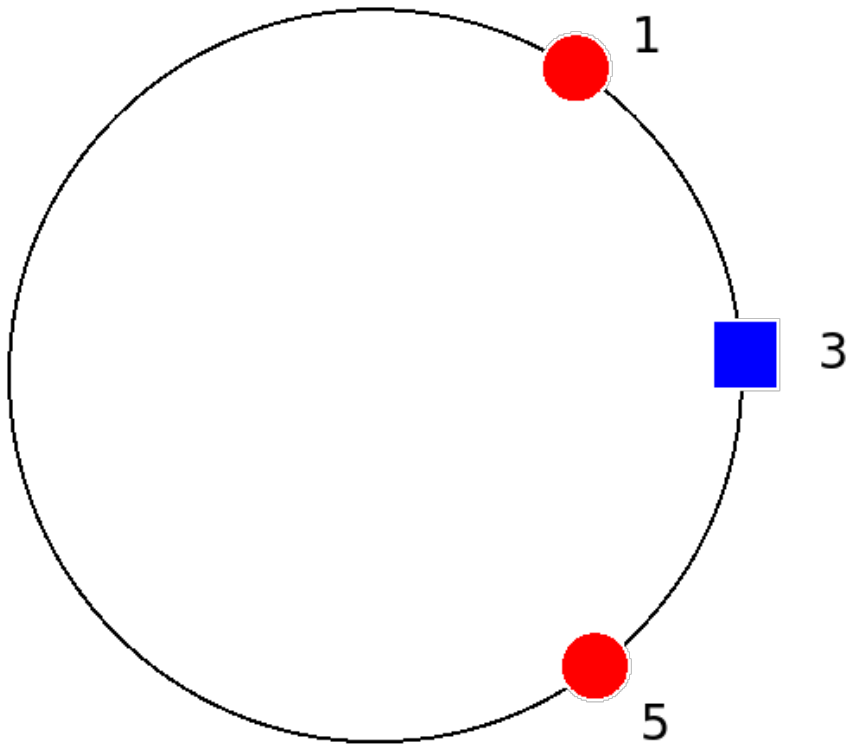


Figure: stabilze 示意图

Chrod 的定义 XXI

代码：

```
1        //n节点 联系在chrod环中的n1节点
2        n.join(n1):
3            predecessor = nil
4            successor = n1.find_successor(n);
5
6        //周期地验证n's的直接后继节点
7        n.stabilize():
8            x = successor.predecessor;
9            //比之前的successor还要离n近
10           if(x \in (n, successor)
11               successor = x; //设置新的successor
12           successor.notify(n); //n 认为可能是successor的新的前驱
13
14       //n1 认为可能是n的前驱
15       n.notify(n1):
16           if(predecessor is nil or n1 \in (predecessor, n)) //比原来的predecessor 更近
17               predecessor = n1;
18
19       //周期地刷新finger table entries
20       n.fix_fingers()
21           i = random index > 1 into finger[];
22           finger[i].node = find_successor(finger[i].start);
```

Chord: 查询表 (Finger Table)

m=6 的环形表，总共有 $2^m = 2^6 = 64$ 个 hash 节点。

指针表

节点 S 的第 i 个指针

$$successor[n + 2^{i-1}], 1 \leq i \leq m$$

$$successor[N8] = \begin{cases} \begin{array}{|c|c|} \hline N8+1 & N14 \\ \hline N8+2 & N14 \\ \hline N8+4 & N14 \\ \hline N8+8 & N21 \\ \hline N8+16 & N32 \\ \hline N8+32 & N42 \\ \hline \end{array} \end{cases}$$

Successor(K): 从 K 开始顺时钟方向距离 **K** 最近的节点。

Chord: 基于查询表 (Finger Table) 的扩展查找过程

查找 **Lookup(K54)** 过程:

$successor[N8] =$

$N8+1$	$N14$
$N8+2$	$N14$
$N8+4$	$N14$
$N8+8$	$N21$
$N8+16$	$N32$
$N8+32$	$N42$

$successor[N42] =$

$N42+1$	$N48$
$N42+2$	$N48$
$N42+4$	$N48$
$N42+8$	$N51$
$N42+16$	$N1$
$N42+32$	$N12$

通过 N8 节点找到 N42 节点，然后读取 N42 节点的查询表。在 N42 节点的查找表中，找到小于 K54 的最大跳跃值，然后跳跃到那个节点。

Chord 节点加入，节点退出

Chord 节点加入

- 新节点 N 事先知道某个或某些节点，并且通过这些节点初始化自己的指针表。(新节点 N 将要求已知的系统中某节点为它查找指针表中的各个表项) $successor[n + 2^{i-1}]\ 1 \leq i \leq m$。
- 在其它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。
- 新节点 N 的第一个后继节点将其维护的小于 N 节点 ID 的所有的 K 交个该节点维护。

Chord 节点加入，节点退出

Chord 节点退出

- 当 Chord 中某个节点 M 退出时，所有在指针表 (Finger table)包含该节点的节点，将相应的指针指向节点 M 的后继节点。
- 每个 Chord 节点都维护一张包括 r 个最近后续节点的后继列表。作为一个 cache。