### 1.1 我该访问谁？

对于分布式缓存来说，当一个节点接收到请求，如果该节点并没有存储缓存值，那么它面临的难题是，从谁那获取数据？自己，还是节点1, 2, 3, 4… 。

那有什么办法，对于给定的 key，每一次都选择同一个节点呢？使用 hash 算法也能够做到这一点。假设包括自己在内一共有 10 个节点。那把 key 的每一个字节的 数值加起来，再除以 10 取余数可以吗？当然可以，这可以认为是自定义的 hash 算法。

### 1.2 节点数量变化了怎么办？

假设，移除了其中一台节点，只剩下 9 个，那么之前 hash(key) % 10 变成了 hash(key) % 9，也就意味着几乎缓存值对应的节点都发生了改变。即几乎所有的缓存值都失效了。节点在接收到对应的请求时，均需要重新去数据源获取数据，容易引起 缓存雪崩。

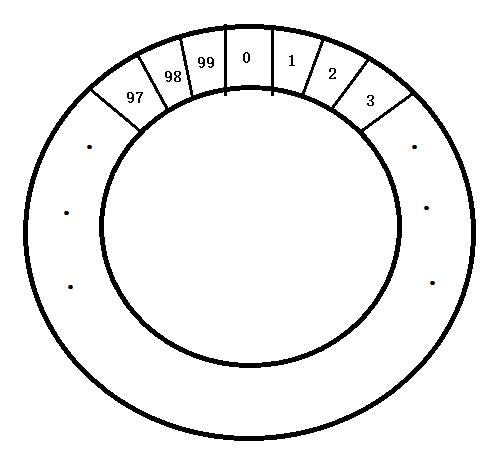
缓存雪崩：缓存在同一时刻全部失效，造成瞬时DB请求量大、压力骤增，引起雪崩。常因为缓存服务器宕机，或缓存设置了相同的过期时间引起。

那如何解决这个问题呢？一致性哈希算法可以。

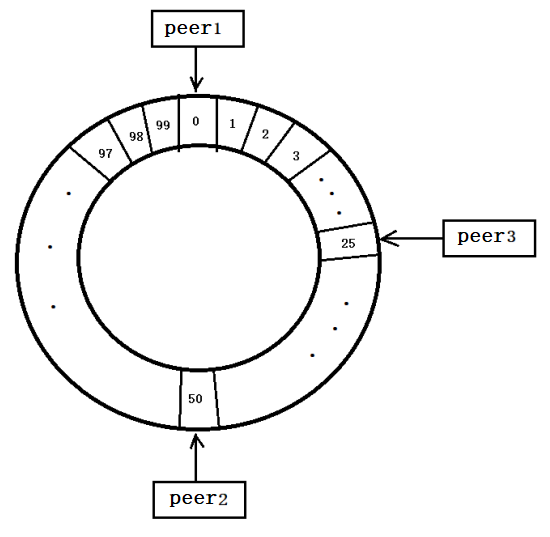
## 2 算法原理

### 2.1 步骤

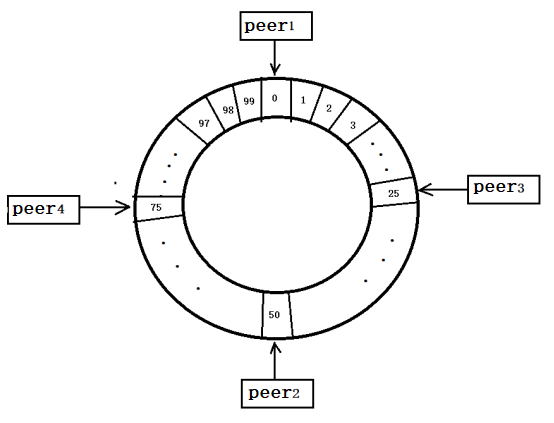
如果把所有key映射到100个hash值上，将这个数字首尾相连，形成一个环。



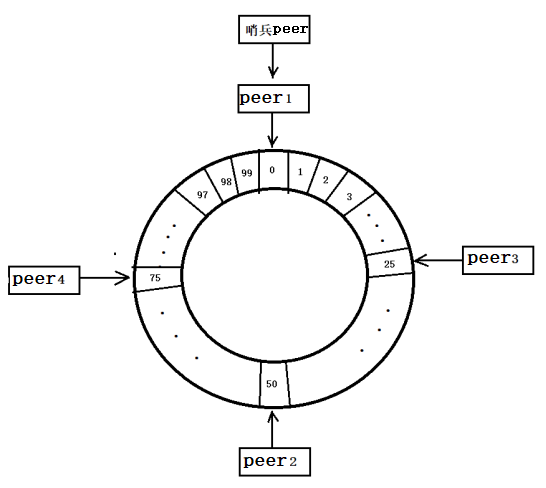
* 添加node，一个node指向一个hash值。



为了hash值能比较均衡得分布在各个Node，所以如果新加入一个Node，会寻找已有得Node中，两个Node之间间隔最大的，插入这两个Node的中间。比如上图的情况，如果要插入一个新Node4，那么Node2到Node1的间隔最大（50），那么Node4就插入Node2到Node1的中间（75）



并且为了便于计算新Node的位置，不用对首尾交界处做特殊处理，于是加一个虚拟的哨兵Node，指向的hash值为100（当前设置并不存在100的hash值）。



* 计算 key 的哈希值，顺时针寻找到的第一个节点，就是应选取的节点/机器。

如果新增节点Node4，假设它新增位置如图所示，那么只有 50-75从Node1 调整到 Node4，其余hash值的归属均没有发生改变。

也就是说，一致性哈希算法，在新增/删除节点时，只需要重新定位该节点附近的一小部分数据，而不需要重新定位所有的节点，这就解决了上述的问题。