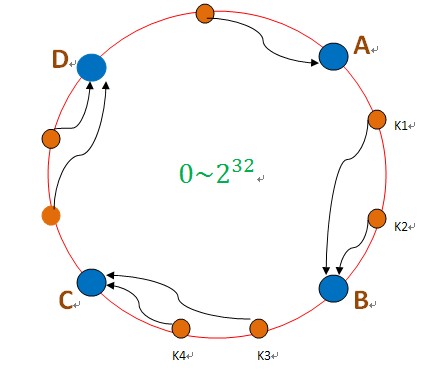
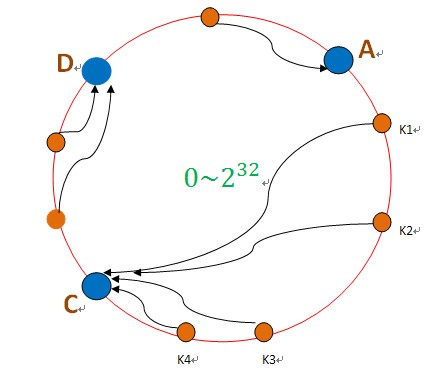
一致性哈希算法是分布式系统中常用的算法。比如，一个分布式的存储系统，要将数据存储到具体的节点上，如果采用普通的hash方法，将数据映射到具体的节点上，如key%N，key是数据的key，N是机器节点数，如果有一个机器加入或退出这个集群，则所有的数据映射都无效了，如果是持久化存储则要做数据迁移，如果是分布式缓存，则其他缓存就失效了。

    因此，引入了一致性哈希算法：



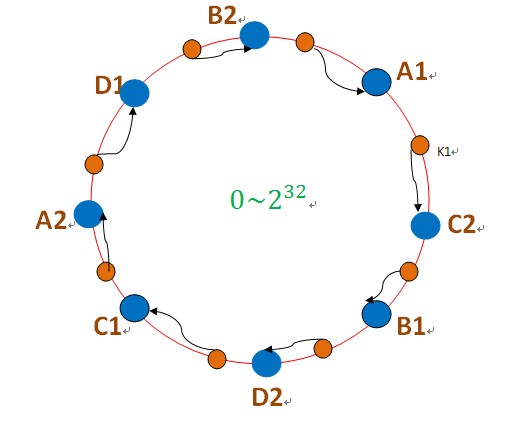
把数据用hash函数（如MD5），映射到一个很大的空间里，如图所示。数据的存储时，先得到一个hash值，对应到这个环中的每个位置，如k1对应到了图中所示的位置，然后沿顺时针找到一个机器节点B，将k1存储到B这个节点中。

如果B节点宕机了，则B上的数据就会落到C节点上，如下图所示：



这样，只会影响C节点，对其他的节点A，D的数据不会造成影响。然而，这又会造成一个“雪崩”的情况，即C节点由于承担了B节点的数据，所以C节点的负载会变高，C节点很容易也宕机，这样依次下去，这样造成整个集群都挂了。

       为此，引入了“虚拟节点”的概念：即把想象在这个环上有很多“虚拟节点”，数据的存储是沿着环的顺时针方向找一个虚拟节点，每个虚拟节点都会关联到一个真实节点，如下图所使用：



图中的A1、A2、B1、B2、C1、C2、D1、D2都是虚拟节点，机器A负载存储A1、A2的数据，机器B负载存储B1、B2的数据，机器C负载存储C1、C2的数据。由于这些虚拟节点数量很多，均匀分布，因此不会造成“雪崩”现象。

**Java实现：**

**[java]** [view plaincopyprint?](http://blog.csdn.net/wuhuan_wp/article/details/7010071)

1. public class Shard<S> { // S类封装了机器节点的信息 ，如name、password、ip、port等
3. private TreeMap<Long, S> nodes; // 虚拟节点
4. private List<S> shards; // 真实机器节点
5. private final int NODE\_NUM = 100; // 每个机器节点关联的虚拟节点个数
7. public Shard(List<S> shards) {
8. super();
9. this.shards = shards;
10. init();
11. }
13. private void init() { // 初始化一致性hash环
14. nodes = new TreeMap<Long, S>();
15. for (int i = 0; i != shards.size(); ++i) { // 每个真实机器节点都需要关联虚拟节点
16. final S shardInfo = shards.get(i);
18. for (int n = 0; n < NODE\_NUM; n++)
19. // 一个真实机器节点关联NODE\_NUM个虚拟节点
20. nodes.put(hash("SHARD-" + i + "-NODE-" + n), shardInfo);
22. }
23. }
25. public S getShardInfo(String key) {
26. SortedMap<Long, S> tail = nodes.tailMap(hash(key)); // 沿环的顺时针找到一个虚拟节点
27. if (tail.size() == 0) {
28. return nodes.get(nodes.firstKey());
29. }
30. return tail.get(tail.firstKey()); // 返回该虚拟节点对应的真实机器节点的信息
31. }
33. /\*\*
34. \*  MurMurHash算法，是非加密HASH算法，性能很高，
35. \*  比传统的CRC32,MD5，SHA-1（这两个算法都是加密HASH算法，复杂度本身就很高，带来的性能上的损害也不可避免）
36. \*  等HASH算法要快很多，而且据说这个算法的碰撞率很低.
37. \*  http://murmurhash.googlepages.com/
38. \*/
39. private Long hash(String key) {
41. ByteBuffer buf = ByteBuffer.wrap(key.getBytes());
42. int seed = 0x1234ABCD;
44. ByteOrder byteOrder = buf.order();
45. buf.order(ByteOrder.LITTLE\_ENDIAN);
47. long m = 0xc6a4a7935bd1e995L;
48. int r = 47;
50. long h = seed ^ (buf.remaining() \* m);
52. long k;
53. while (buf.remaining() >= 8) {
54. k = buf.getLong();
56. k \*= m;
57. k ^= k >>> r;
58. k \*= m;
60. h ^= k;
61. h \*= m;
62. }
64. if (buf.remaining() > 0) {
65. ByteBuffer finish = ByteBuffer.allocate(8).order(
66. ByteOrder.LITTLE\_ENDIAN);
67. // for big-endian version, do this first:
68. // finish.position(8-buf.remaining());
69. finish.put(buf).rewind();
70. h ^= finish.getLong();
71. h \*= m;
72. }
74. h ^= h >>> r;
75. h \*= m;
76. h ^= h >>> r;
78. buf.order(byteOrder);
79. return h;
80. }
82. }