哈希函数

Hash，一般翻译做"散列"，也有直接音译为"哈希"的，就是把任意长度的输入（又叫做预映射， pre-image），通过散列算法，变换成固定长度的输出，该输出就是散列值。

性质：

- 输入可以是无穷大，输出是有限范围内
- 同样的输入，同样的输出
- 不同输入可能会有同样的输出(哈希碰撞)
- 输出均匀的，离散的，分散在哈希表
- 将输出取模 m，得到的数据均匀的分布在 0~m-1

问题： 给一个范围`0~2^32-1`的无符号整数，一共有 40 亿个，给定 1G 内存，返回出现次数最多的数。

思路：

- 对所有整数通过哈希函数(f)进行映射，并对 100 取模；
- 此时所有数将均匀分布在 0~99 上，将所有整数按照取模后的结果分发到 0~99 号小文件中；
- 用 100 个哈希表分别记录小文件中整数和出现的次数(key,count)；
- 在所有的哈希表中返回 count 最大的 key；

哈希表

哈希表在使用层面上可以理解为一种集合结构

哈希表的时间复杂度是**常数级别**的，能实现增删改查功能

哈希表的基础实现原理：

- 假设初始哈希表中只有 16(size)个内存地址，对插入哈希表的 key 采用哈希函数进行映射并对 16 取模；
- 此时，取模后的结果将均匀分布在 0~15 上，将 key 按照取模后的结果放置在对应的内存中，同个内存中的记录用链表串联；
- 查询和删除时，将 key 按照 size 取模的结果找到对应内存地址，然后对链表遍历寻找；
- 所以当链表长度过长时影响效率，需要进行扩容操作；
- 假设当链表长度大于 8 时进行扩容。先申请 32 个内存地址，并对之前所有的记录重新用 32 进行哈希函数映射和取模并放入对应的新内存中，此时链表长度将会近似等于 4；

时间复杂度：

- 单个值采用哈希函数进行映射并对 16 取模 O(1)
- 遍历小长度链表 O(1)
- 插入 N 个值需要扩容 logN 次，每次扩容代价为 N O(N*logN)
- 单次插入的平均时间复杂度 O(logN)
- 查找和删除 O(1)

JS 中哈希表的实现

HashSet <key>: 只有 key，没有伴随数据 value。

```
const HashSet = new Set();
// 增
HashSet.add(1);
// 删
HashSet.delete(1);
// 查
HashSet.has(1);
// 清空
HashSet.clear();
```

HashMap <key,value>: 既有 key，又有伴随数据 value，以 key 来排序组织。key 可以是基础类型也可以是**引用类型。**

```
const HashMap = new Map();
// 增
HashMap.set("1", 1);
// 删
HashMap.delete("1");
// 查
HashMap.has("1");
// 改
HashMap.get("1");
// 清空
HashMap.clear();
```

设计 RandomPool 结构

【题目】

设计一种结构，在该结构中有如下三个功能:

insert(key):将某个 key 加入到该结构，做到不重复加入

delete(key):将原本在结构中的某个 key 移除

getRandom(): 等概率随机返回结构中的任何一个 key。

【要求】

Insert、delete 和 getRandom 方法的时间复杂度都是 O(1)

思路:

- 结构中包含 KeyMap{key:index}、IndexMap{index:key}、size。其中 index 是 key 的插入顺序，size 是 Map 的长度;
- 插入时，分别在 KeyMap 和 IndexMap 中插入对应记录，size 加 1;
- 删除 key1 时，将 index 等于 size - 1 的 key 在 KeyMap 和 IndexMap 中更改 index 为要删除 key1 的 index，然后删除 key1;
- 返回随机 key，随机生成 0~size-1 的整数，返回对应 index 的 key;

```
class RandomPool {
  constructor() {
    this.keyMap = new Map();
    this.indexMap = new Map();
    this.size = 0;
```

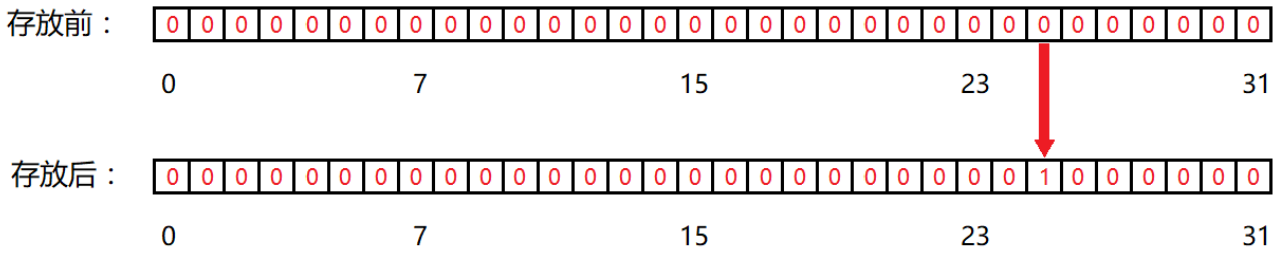
```
  }
  insert(key) {
    if (!this.keyMap.has(key)) {
      this.keyMap.set(key, this.size);
      this.indexMap.set(this.size++, key);
    }
  }
  delete(key) {
    if (this.keyMap.has(key)) {
      // 获取key对应的index
      const index = this.keyMap.get(key);
      // 获取最后的index
      const lastIndex = --this.size;
      // 获取最后一个key
      const keyLast = this.indexMap.get(lastIndex);
      // 将keyLast的index设置成key的index
      this.keyMap.set(keyLast, index);
      // 将index对应的key换成keyLast
      this.indexMap.set(index, keyLast);
      // 删除key
      this.keyMap.delete(key);
      // 删除最后的index
      this.indexMap.delete(lastIndex);
    }
  }
  getRandom() {
    if (this.size === 0) {
      return null;
    }
    const index = parseInt(Math.random() * this.size);
    return this.indexMap.get(index);
  }
}
```

布隆过滤器

布隆过滤器就是一个大的**位图** 设 m 个 bit 范围是 0 ~ m-1，实际占用 m/8 字节。

位图

位图是用数组实现的，数组的每一个元素的每一个二进制位都可以表示一个数据在或者不在，0 表示数据存在，1 表示数据不存在。所以位图其实就是一种直接定址法的哈希，只不过位图只能表示这个值在或者不在。



例：如果我们要表示136这个数据存在
（1）136/32=4，因此136存在第四个区间里
（2）136%32=25，因此136存在第四个区间的第25个比特位上
（3）将该比特位置为1

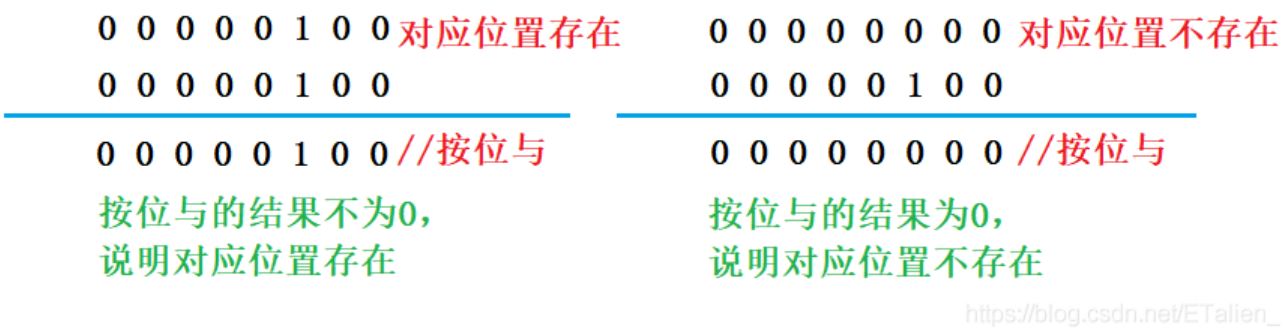
将对应的比特位置变成 1


```
const set = (i) => {
  const index = i / 32; // 计算是数组的第几段
  const num = i % 32; // 计算在段中的第几位

  this.bit[index] = this.bit[index] | (1 << num);
};
```

将对应的比特位置变成 0

```
const reset = (i) => {
  const index = i / 32; // 计算是数组的第几段
  const num = i % 32; // 计算在段中的第几位

  this.bit[index] = this.bit[index] & ~(1 << num);
};
```

判断对应的比特位是 0（不存在）还是 1（存在）




```
const test = (i) => {
  const index = i / 32; // 计算是数组的第几段
  const num = i % 32; // 计算在段中的第几位

  this.bit[index] = this.bit[index] & ~(1 << num);
};
```

使用场景

布隆过滤器使用场景：

- 如黑名单系统
- 无删除行为
- 允许有失误率

例如：公司有长度为 40 亿的黑名单 URL 且**无删除行为**，当用户访问网址时需要校验是否在黑名单中，**允许有失误率**。问该如何查找当前 URL 是否在黑名单中?

使用布隆过滤器来进行查找：

1. 将 40 亿 URL 依次经过k个不同的哈希函数映射并模上m，在长度为m的位图上将得到的结果位置设置为 1；
2. 查询时，将要查询的 URL 同样经过k个哈希函数映射并模上m，如过得到的k个位置在位图上都为 1，则 URL 在黑名单中，如果有任意一个位置为 0，则不在；

性质

- 会将白名单 URL 误判为黑名单(模上m时，可能会导致两个不同的 URL 落在一起)，但不会将黑名单误判为白名单(同一个输入哈希函数得出的结果一致，因此如果已经被加入黑名单中一定不会误判)。
- 失误率与m成反比
- 当给定m时，失误率随k先减后增

m和k的计算公式

$m = -(n * lnp) / (ln2)^2$

$k = ln2 * m / n ≈ 0.7 * m / n$ 向上取整

$p{real} = (1 - e^{-\frac{n*k}{real}}}{m{real}}})^{k}{real}}}$

$m = 预期位图的大小 \qquad k = 预期哈希函数的个数\qquad n = 样本量\qquad p = 预期失误率\qquad p{real} = 真实失误率\qquad k{real} = 真实k值\qquad m_{real} = 真实m值$

一致性哈希

https://zhuanlan.zhihu.com/p/129049724

对于分布式存储，不同机器上存储不同对象的数据，我们使用哈希函数建立从数据到服务器之间的映射关系。

一、使用简单的哈希函数

```
m = hash(o) mod n
```

其中，o 为对象名称，n 为机器的数量，m 为机器编号。

考虑以下例子：

3 个机器节点，10 个数据 的哈希值分别为 1,2,3,4,…,10。使用的哈希函数为：(`m=hash(o) mod 3`)

机器 0 上保存的数据有：3，6，9

机器 1 上保存的数据有：1，4，7，10

机器 2 上保存的数据有：2，5，8

当增加一台机器后，此时 n = 4，各个机器上存储的数据分别为：

机器 0 上保存的数据有：4，8

机器 1 上保存的数据有：1，5，9

机器 2 上保存的数据有：2，6，10

机器 3 上保存的数据有：3，7

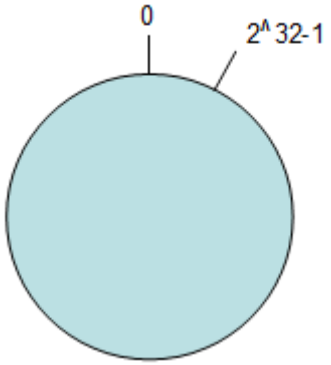
只有数据 1 和数据 2 没有移动，所以当集群中数据量很大时，采用一般的哈希函数，在节点数量动态变化的情况下会造成大量的数据迁移，导致网络通信压力的剧增，严重情况，还可能导致数据库宕机。

二、一致性哈希

一致性 hash 算法正是为了解决此类问题的方法，它可以保证当机器增加或者减少时，节点之间的数据迁移只限于两个节点之间，不会造成全局的网络问题。

1. 环形 Hash 空间

按照常用的 hash 算法来将对应的 key 哈希到一个具有 2^32 次方个桶的空间中，即 0~(2^32)-1 的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形。如下图：



2. 将数据通过 hash 算法映射到环上

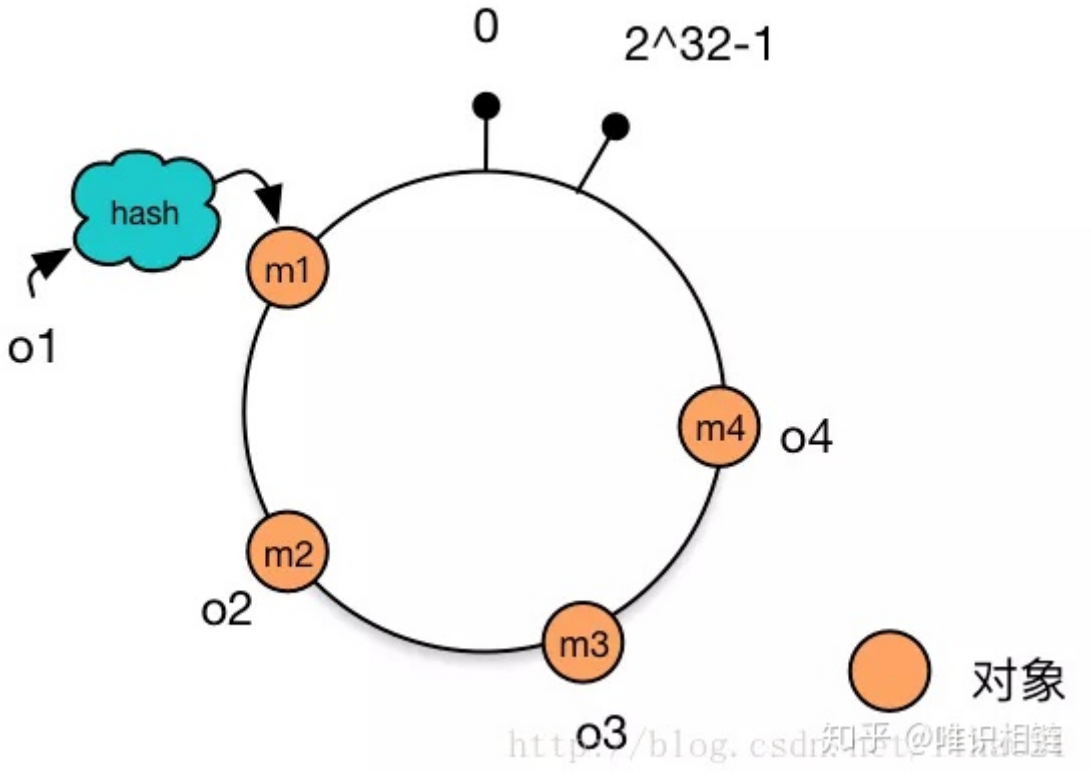
将 object1、object2、object3、object4 四个对象通过特定的 Hash 函数计算出对应的 key 值，然后散列到 Hash 环上。如下图：

Hash(object1) = key1；

Hash(object2) = key2；

Hash(object3) = key3；

Hash(object4) = key4；



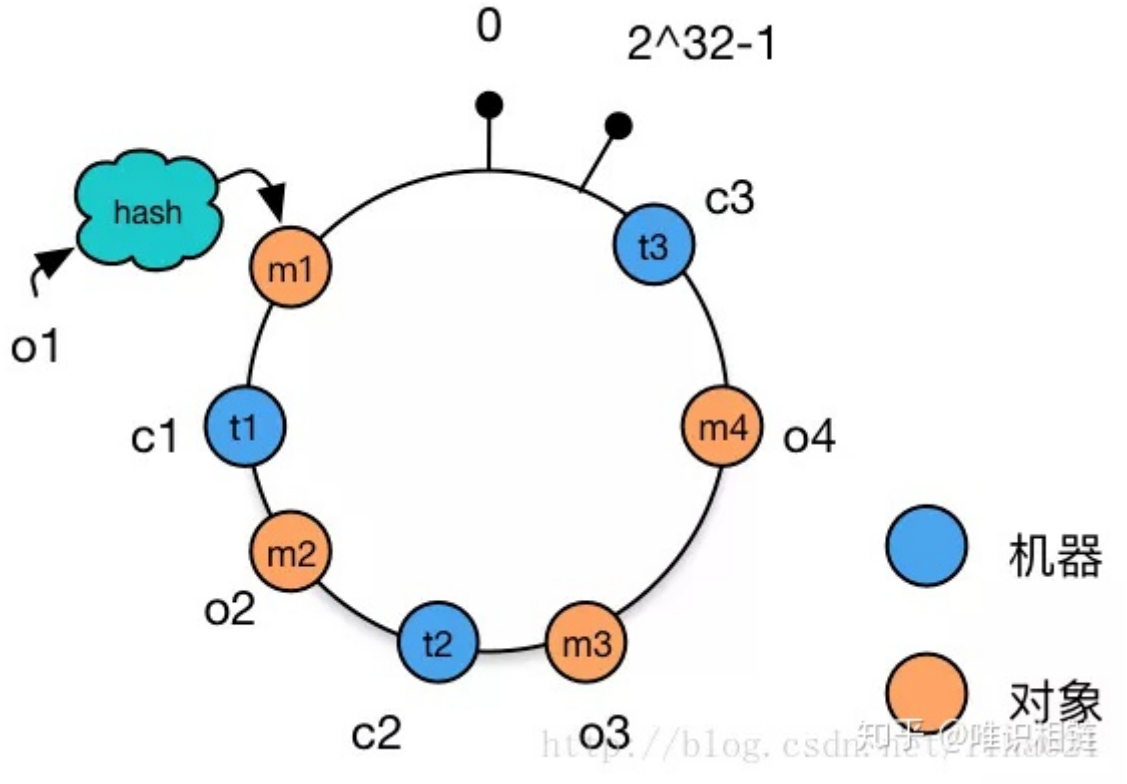
3. 将机器通过 hash 算法映射到环上

假设现在有 NODE1，NODE2，NODE3 三台机器，通过 Hash 算法（机器 IP 或机器的唯一的名称作为输入）得到对应的 KEY 值，映射到环中，其示意图如下：

Hash(NODE1) = KEY1;

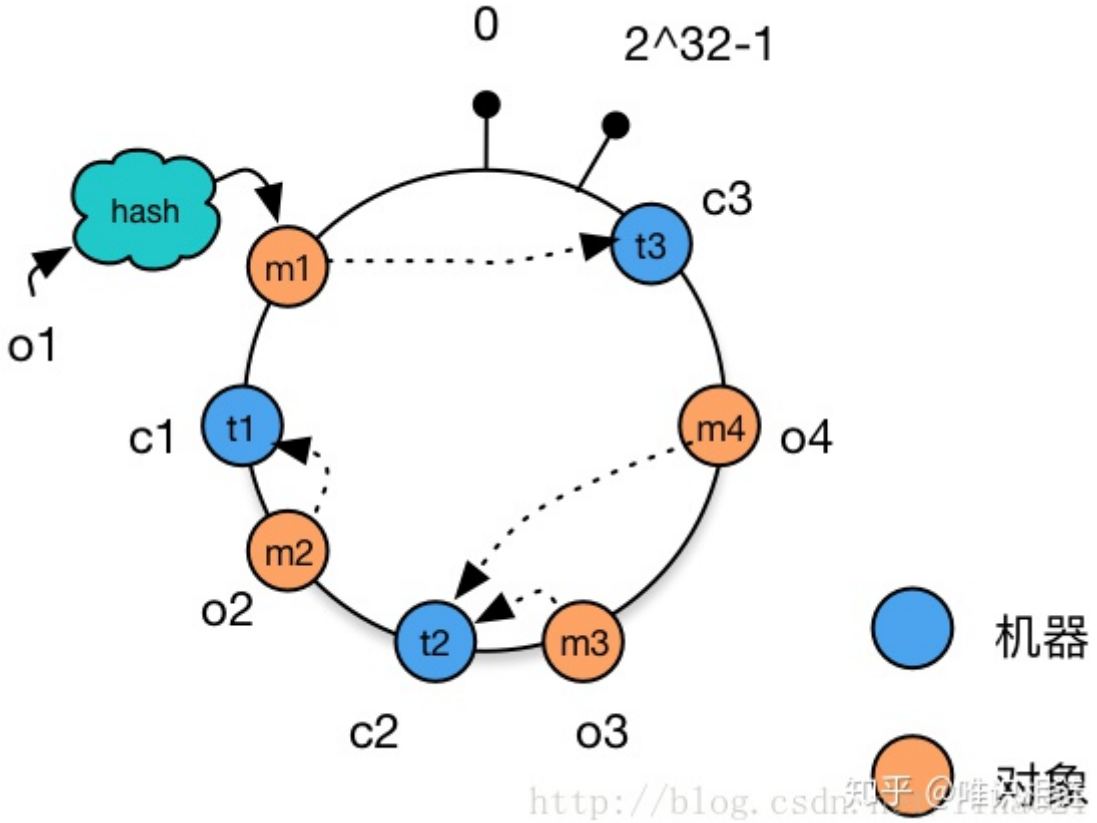
Hash(NODE2) = KEY2;

Hash(NODE3) = KEY3;



4. 将数据存储到机器上

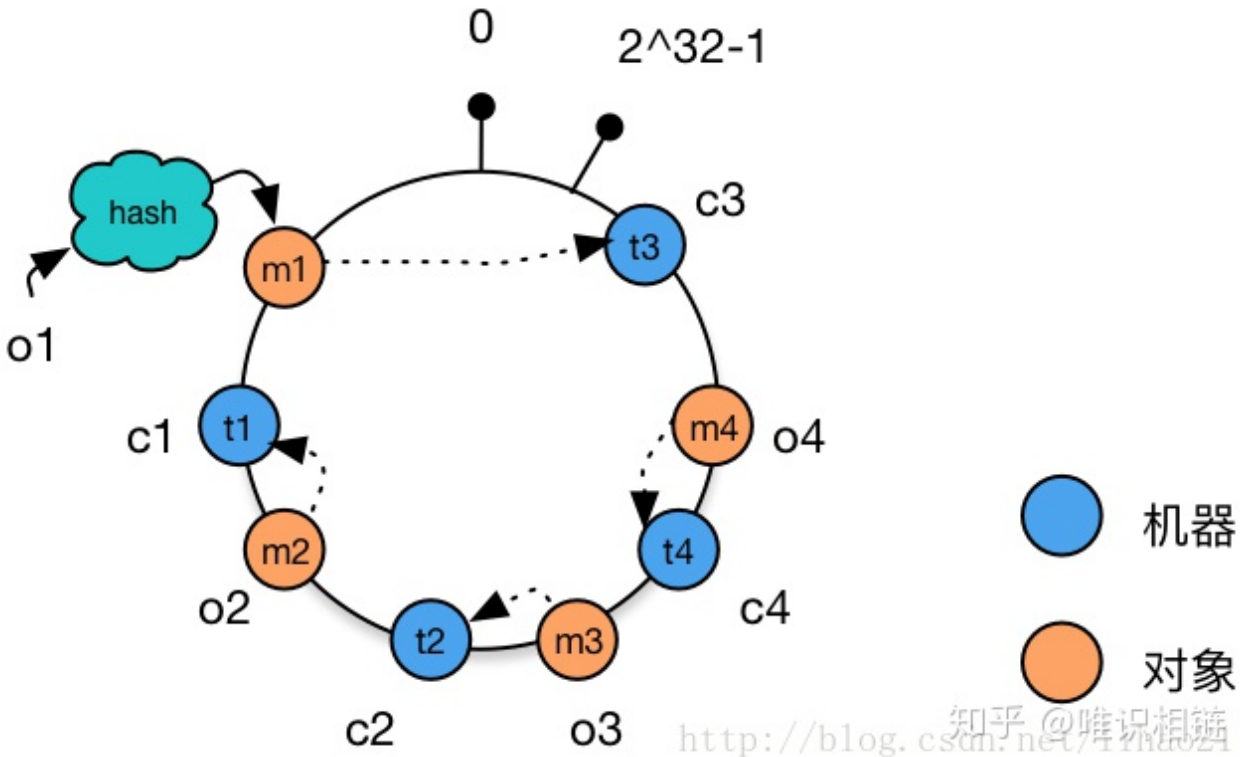
通过上图可以看出对象与机器处于同一哈希空间中，这样按顺时针转动 object1 存储到了 NODE1 中，object3 存储到了 NODE2 中，object2、object4 存储到了 NODE3 中。



5. 机器的添加与删除

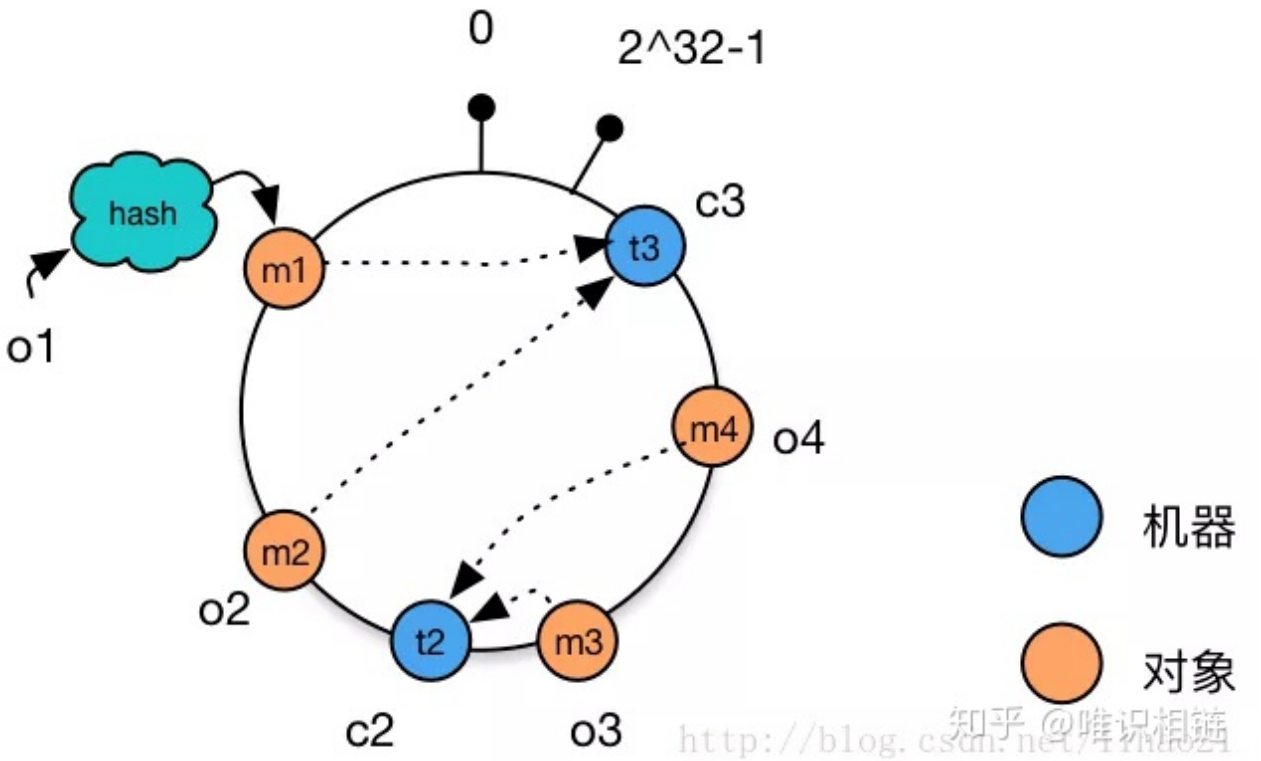
1. 向集群中添加一台新机器

   向集群中增加机器 c4，c4 经过 hash 函数后映射到机器 c2 和 c3 之间。这时根据顺时针存储的规则，数据 m4 从机器 c2 迁移到机器 c4。数据的移动仅发生在 c2 和 c4 之间，其他机器上的数据并未受到影响。



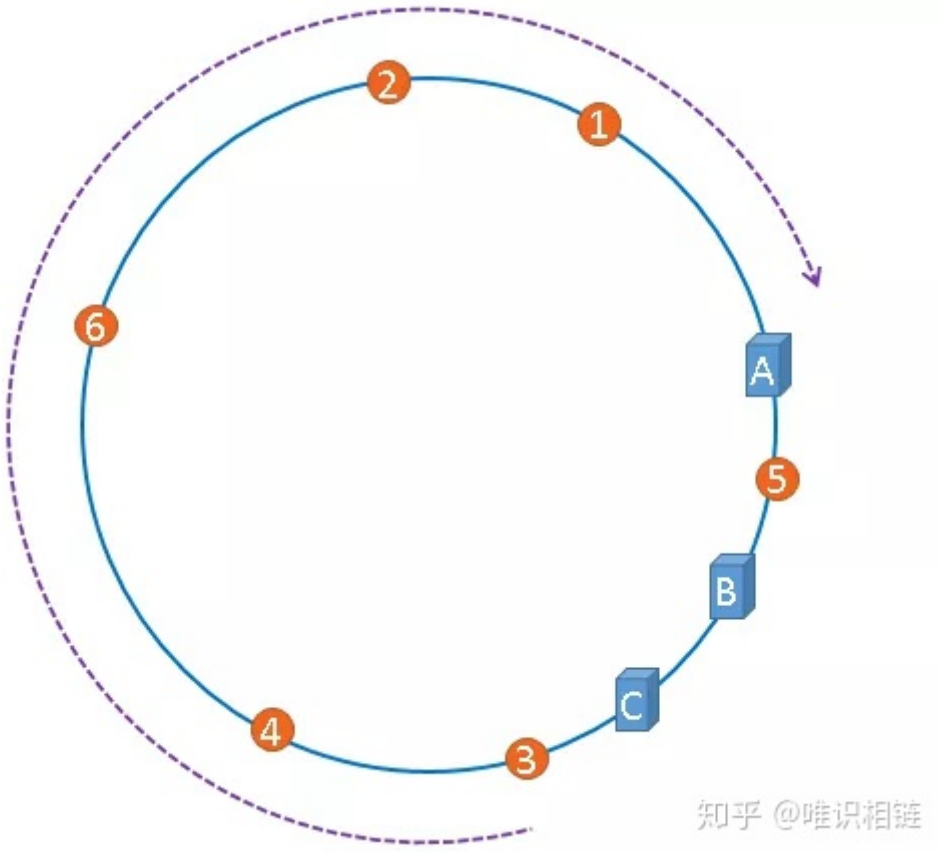
2. 从集群中删除一台机器

   从集群中删除机器 c1，这时只有 c1 原有的数据需要迁移到机器 c3，其他数据并未受到影响。



- 相比于之前的简单取模方法中动态增删集群中机器的数量时，造成全局的数据迁移，使用一致性哈希算法将大大改善这种情况，减轻了网络通信的压力。

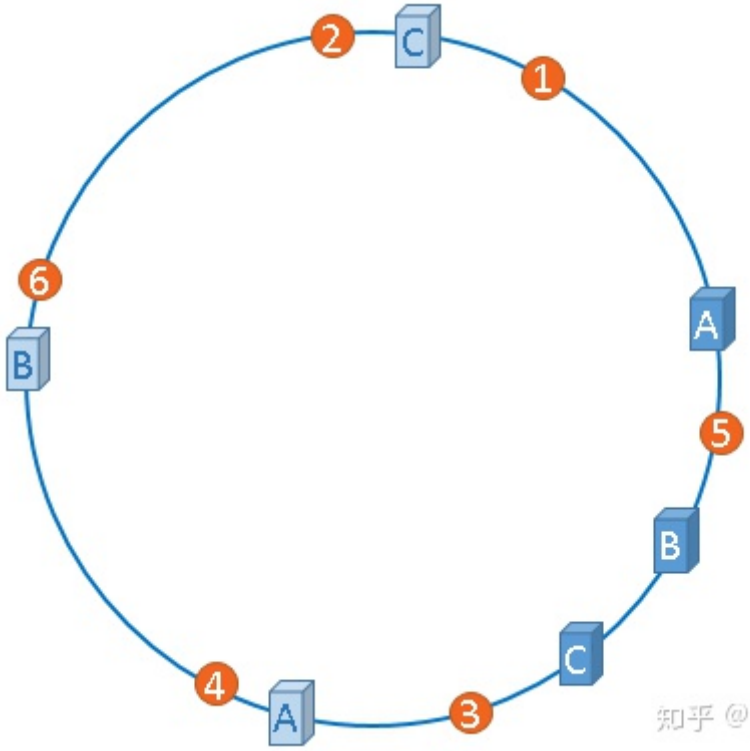
存在的问题：
当集群中的节点数量较少时，可能会出现节点在哈希空间中分布不平衡的问题。如下图所示，图中节点 A、B、C 分布较为集中，造成 hash 环的倾斜。数据 1、2、3、4、6 全部被存储到了节点 A 上，节点 B 上只存储了数据 5，而节点 C 上什么数据都没有存储。**A、B、C 三台机器的负载极其不均衡。**



在极端情况下，假如 A 节点出现故障，存储在 A 上的数据要全部转移到 B 上，大量的数据导可能会导致节点 B 的崩溃，之后 A 和 B 上所有的数据向节点 C 迁移，导致节点 C 也崩溃，由此导致整个集群宕机。这种情况被称为**雪崩效应。**

解决方法——虚拟节点

解决哈希环偏斜问题的方法就是，让集群中的节点尽可能的多，从而让各个节点均匀的分布在哈希空间中。在现实情境下，机器的数量一般都是固定的，所以我们只能将现有的物理节通过虚拟的方法复制多个出来，这些由实际节点虚拟复制而来的节点被称为**虚拟节点。**加入虚拟节点后的情况如下图所示：



从上图可得：加入虚拟节点后，节点 A 存储数据 1、3；节点 B 存储 5、4；节点 C 存储 2、6。节点的负载很均衡。