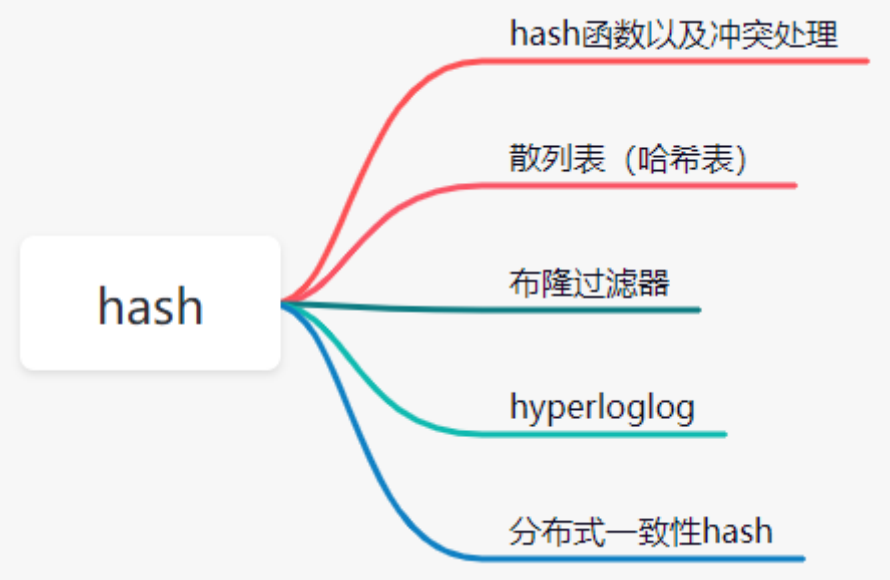
零声学院出品 Mark 老师 QQ: 2548898954

总体知识脉络



背景

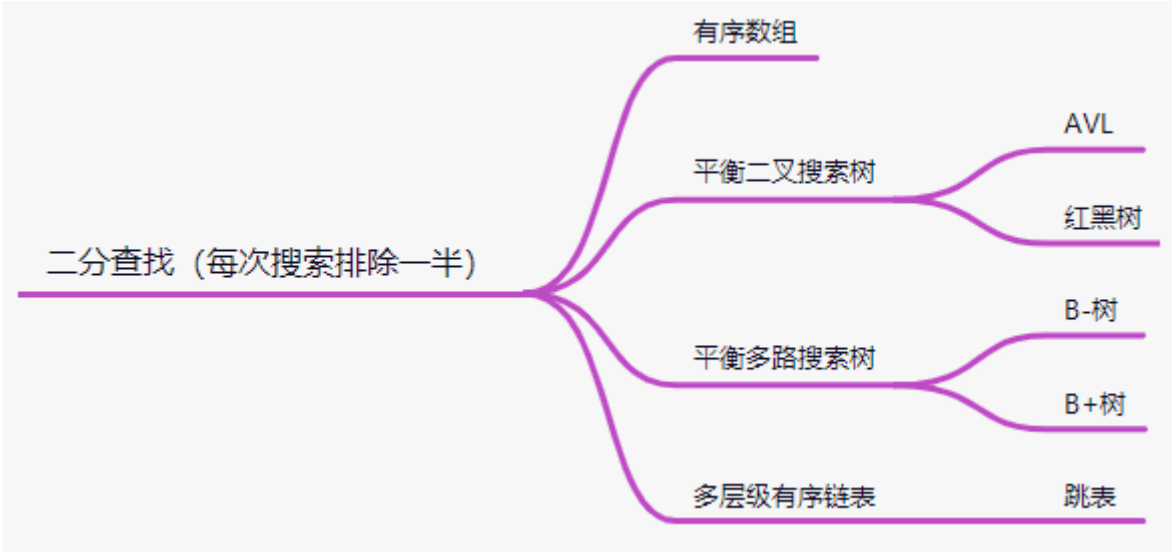
- 使用 word 文档时，word 如何判断某个单词是否拼写正确?
- 网络爬虫程序，怎么让它不去爬相同的 url 页面?
- 垃圾邮件过滤算法如何设计?
- 公安办案时，如何判断某嫌疑人是否在网逃名单中?
- 缓存穿透问题如何解决?

需求

> 从海量数据中查询某个字符串是否存在?

平衡二叉树

> 增删改查时间复杂度为 $O(log_2n)$；
>
> **平衡**的目的是增删改后，保证下次搜索能稳定排除一半的数据；
>
> $O(log_2n)$ 的直观理解：100万个节点，最多比较 20 次；10 亿个节点，最多比较 30 次；
>
> 总结：通过**比较**保证有序，通过每次**排除一半**的元素达到快速索引的目的；



散列表

> 根据 key 计算 key 在表中的位置的数据结构；是 key 和其所在存储地址的映射关系；
>
> 注意：散列表的节点中 kv 是存储在一起的；

```
struct node {
  void *key;
  void *val;
  struct node *next;
};
```

hash 函数

> 映射函数 `Hash(key)=addr`；hash 函数可能会把两个或两个以上的不同 key 映射到同一地址，这种情况称之为**冲突**（或者 hash 碰撞）；

选择 hash

- 计算速度快
- 强随机分布（等概率、均匀地分布在整个地址空间）
- murmurhash1，**murmurhash2**，murmurhash3，**siphash**（redis6.0当中使用，rust等大多数语言选用的hash算法来实现hashmap），**cityhash** 都具备强随机分布性；测试地址如下：

  https://github.com/aappleby/smhasher
- siphash 主要解决字符串接近的强随机分布性；

负载因子

- 数组存储元素的个数 / 数据长度；用来形容散列表的存储密度；负载因子越小，冲突越小，负载因子越大，冲突越大；

冲突处理

- 链表法:

> 引用链表来处理哈希冲突；也就是将冲突元素用链表链接起来；这也是常用的处理冲突的方式；但是可能出现一种极端情况，冲突元素比较多，该冲突链表过长，这个时候可以将这个链表转换为**红黑树**；由原来链表时间复杂度 $O(n)$ 转换为红黑树时间复杂度 $O(log_2 n)$；那么判断该链表过长的依据是多少？可以采用超过 256（经验值）个节点的时候将链表结构转换为红黑树结构；

- 开放寻址法：

> 将所有的元素都存放在哈希表的数组中，不使用额外的数据结构；一般使用线性探查的思路解决；
>
> 1. 当插入新元素的时，使用哈希函数在哈希表中定位元素位置；
> 2. 检查数组中该槽位索引是否存在元素。如果该槽位为空，则插入，否则3；
> 3. 在 2 检测的槽位索引上加一定步长接着检查2； 加一定步长分为以下几种：
>   1. i+1,i+2,i+3,i+4, ... ,i+n
>   2. i-1^2 ,i+2^2 ,i- 3^2,1+4^2, ... 这两种都会导致同类 hash 聚集；也就是近似值它的hash值也近似，那么它的数组槽位也靠近，形成 hash 聚集；第一种同类聚集冲突在前，第二种只是将聚集冲突延后； 另外还可以使用**双重哈希**来解决上面出现hash聚集现象：
>
> ```
> 在.net HashTable类的hash函数Hk定义如下：
> Hk(key) = [GetHash(key) + k * (1 + (((GetHash(key) >> 5) + 1) %
> (hashsize - 1)))] % hashsize
> 在此 (1 + (((GetHash(key) >> 5) + 1) % (hashsize - 1))) 与 hashsize
> 互为素数（两数互为素数表示两者没有共同的质因子）；
> 执行了 hashsize 次探查后，哈希表中的每一个位置都有且只有一次被访问到，也就是
> 说，对于给定的 key，对哈希表中的同一位置不会同时使用 Hi 和 Hj；
> ```
>
>   3. 具体原理：https://www.cnblogs.com/organic/p/6283476.htm

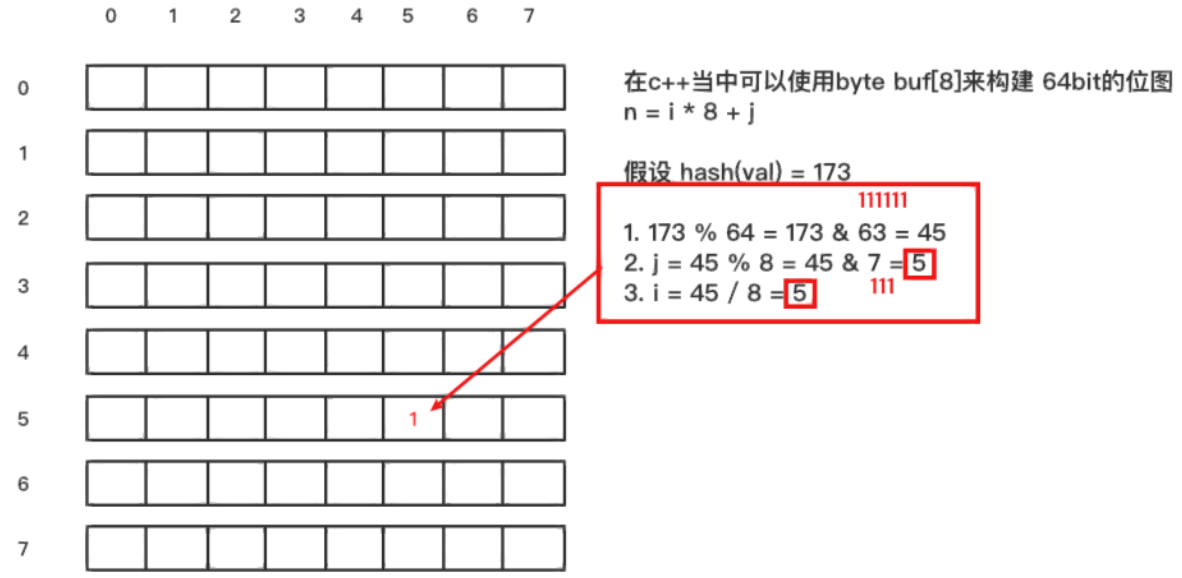
布隆过滤器

背景

> 布隆过滤器是一种**概率型**数据结构，它的特点是高效地插入和查询，能确定某个字符串**一定不存在**或者**可能存在**；
>
> 布隆过滤器不存储具体数据，所以**占用空间小**，查询结果**存在误差**，但是**误差可控**，同时**不支持删除操作**；

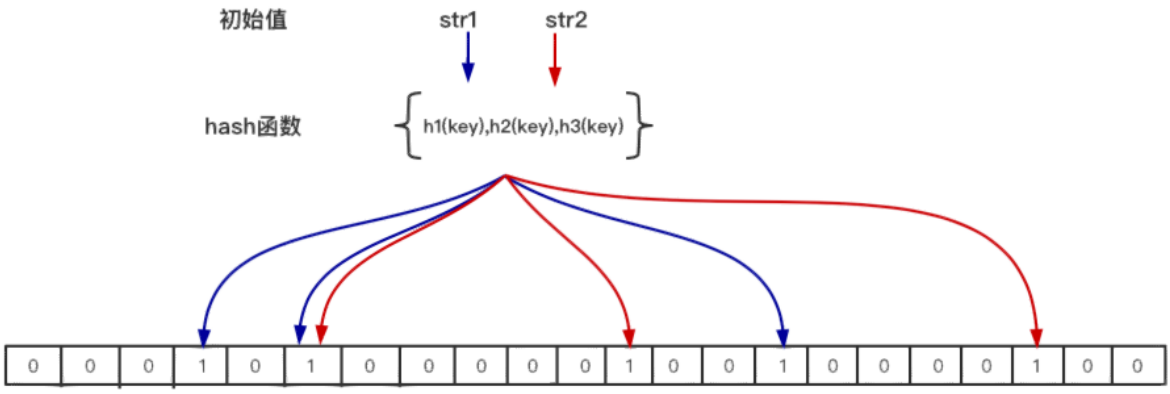
构成

> 位图（BIT 数组）+ n 个 hash 函数
>
> m % 2^n = m & $(2^n - 1)$



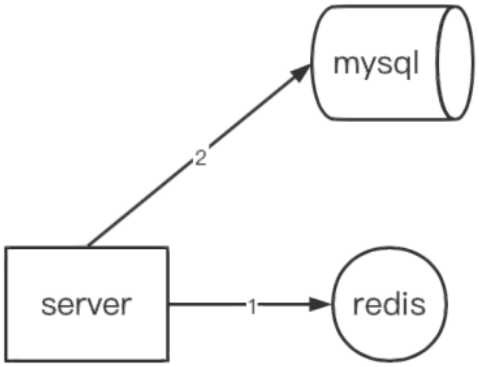
原理

> 当一个元素加入位图时，通过 k 个 hash 函数将这个元素映射到位图的 k 个点，并把它们置为 1；当检索时，再通过 k 个 hash 函数运算检测位图的 k 个点是否都为 1；如果有不为 1 的点，那么认为该 key 不存在；如果全部为 1，则可能存在；
>
> 为什么不支持删除操作?
>
> - 在位图中每个槽位只有两种状态（0 或者 1），一个槽位被设置为 1 状态，但不确定它被设置了多少次；也就是不知道被多少个 key 哈希映射而来以及是被具体哪个 hash 函数映射而来；



应用场景

> 布隆过滤器通常用于判断某个 key 一定不存在的场景，同时允许判断存在时有误差的情况；
>
> 常见处理场景：① 缓存穿透的解决；② 热 key 限流；



1.缓存穿透：
redis，mysql都没有数据，黑客可以利用此漏洞导mysql压力过大，如此以来整个系统将陷入瘫痪。

2.读取步骤：
1> 先访问redis，如存在，直接返回；如不存在走2；
2> 访问mysql，如不存在，直接返回；如存在走3；
3> 将mysql存在的key写回redis；

3.解决方案：
1> 在redis端设置<key, null>键值对，以此避免访mysql；
缺点是<key,null>过多的话，占用过多内存；
* 可以给key设置过期 expire key 600ms，停止攻击后最终由redis自动清除这些无用的key；
2> 在server端存储一个布隆过滤器，将mysql包含的key放入布隆过滤器中；布隆过滤器能过滤一定不存在的数据；

- 描述缓存场景，为了减轻数据库（mysql）的访问压力，在 server 端与数据库（mysql）之间加入缓存用来存储**热点数据**；
- 描述缓存穿透，server端请求数据时，**缓存和数据库都不包含该数据**，最终请求压力全部涌向数据库；
- 数据请求步骤，如图中 2 所示；
- 发生原因：黑客利用漏洞伪造数据攻击或者内部业务 bug 造成大量重复请求不存在的数据；
- 解决方案：如图中 3 所示；

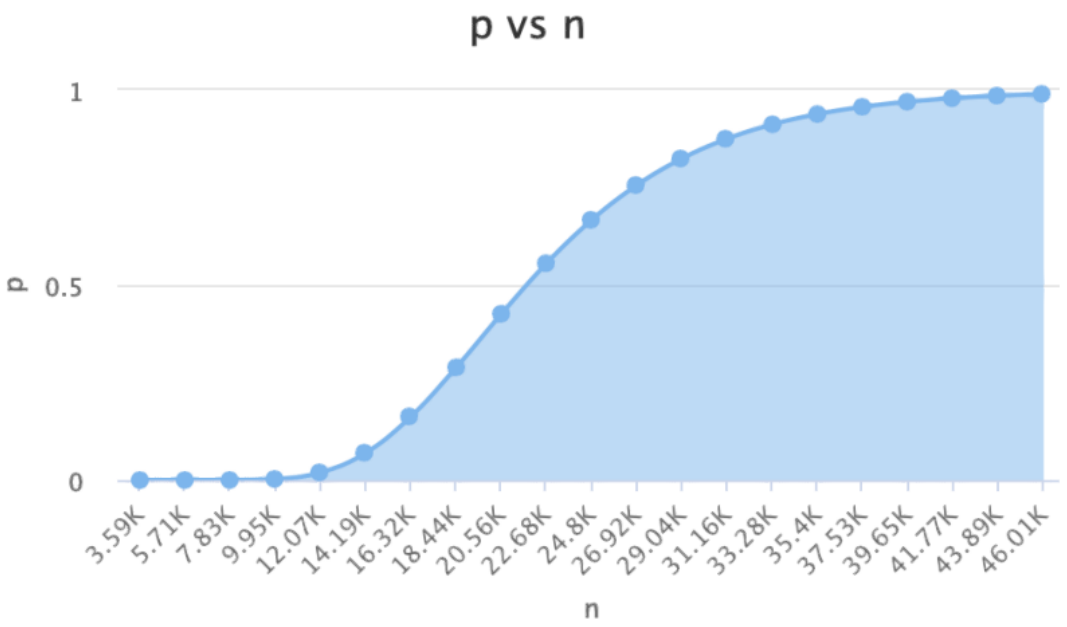
应用分析

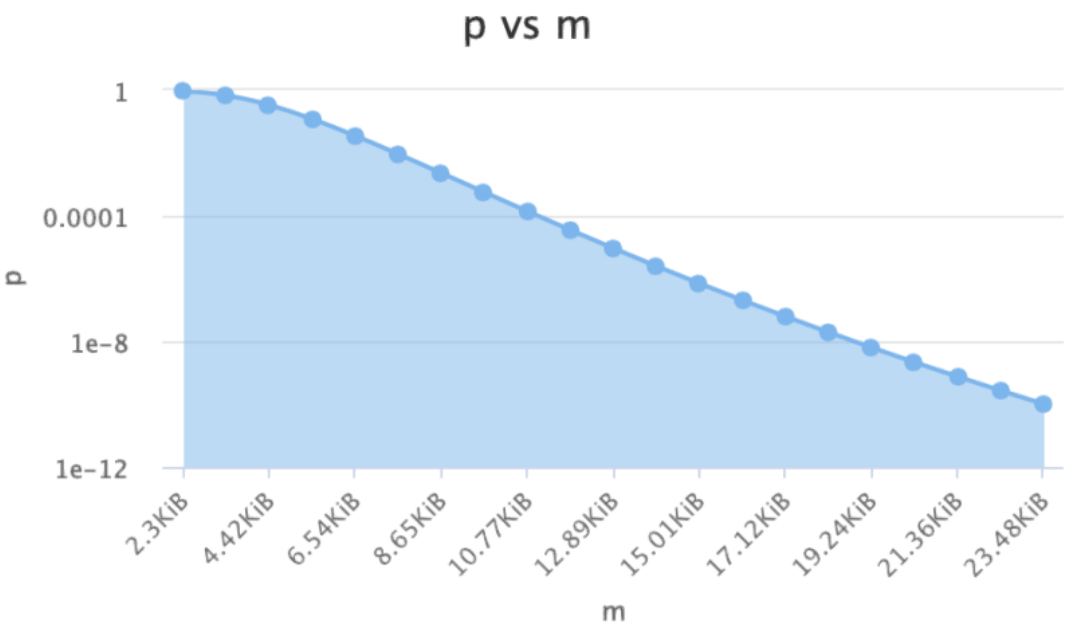
> 在实际应用中，该选择多少个 hash 函数？要分配多少空间的位图？预期存储多少元素？如何控制误差？
>
> 公式如下：

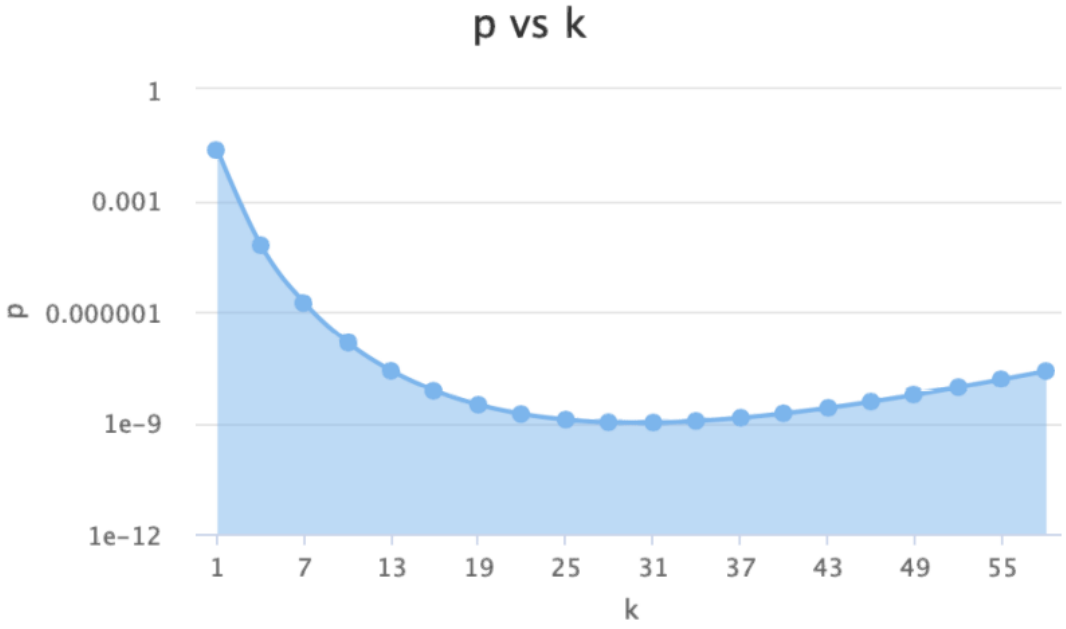
```
n -- 预期布隆过滤器中元素的个数，如上图 只有str1和str2 两个元素 那么 n=2
p -- 假阳率，在0-1之间 0.000000
m -- 位图所占空间
k -- hash函数的个数
公式如下：
n = ceil(m / (-k / log(1 - exp(log(p) / k))))
p = pow(1 - exp(-k / (m / n)), k)
m = ceil((n * log(p)) / log(1 / pow(2, log(2))));
k = round((m / n) * log(2));
```

变量关系

> 假定4个初始值：
>
> `n = 4000`
>
> `p = 0.000000001`
>
> `m = 172532`
>
> `k = 30`







> 面试百度 hash 函数实现过程当中 为什么 会出现 i*31?
>
> - i * 31 = i * (32-1) = i * (1<<5 -1) = i << 5 - i;

> - 31 质数，hash 随机分布性是最好的；

确定 n 和 p

> 在实际使用布隆过滤器时，首先需要确定 n 和 p，通过上面的运算得出 m 和 k；通常可以在下面这个网站上选出合适的值；
>
> https://hur.st/bloomfilter

选择 hash 函数

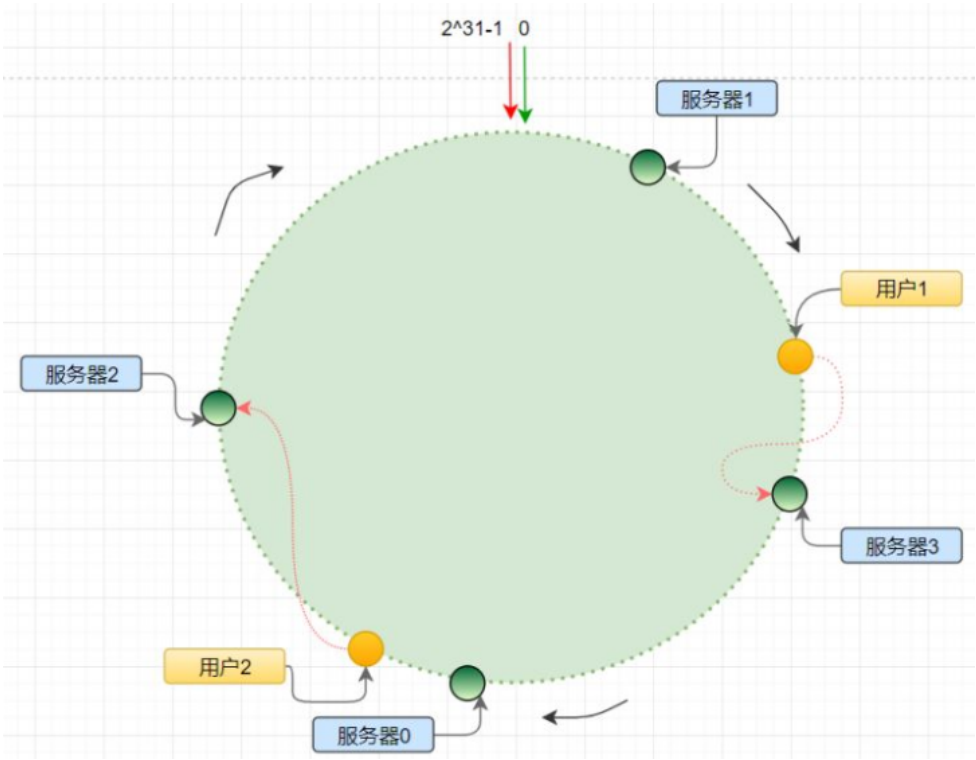
> 选择一个 hash 函数，通过给 hash 传递不同的种子偏移值，采用线性探寻的方式构造多个 hash 函数；

```
#define MIX_UINT64(v) ((uint32_t)((v>>32)^(v)))
uint64_t hash1 = MurmurHash2_x64(key, len, Seed);
uint64_t hash2 = MurmurHash2_x64(key, len, MIX_UINT64(hash1));
for (i = 0; i < k; i++) // k 是hash函数的个数
{
     Pos[i] = (hash1 + i*hash2) % m; // m 是位图的大小
}
```

分布式一致性 hash

背景

> 分布式一致性 hash 算法将哈希空间组织成一个虚拟的圆环，圆环的大小是 2^{32}；
>
> 算法为：`hash(ip)` $\%2^{32}$，最终会得到一个 $[0, 2^{32}-1]$ 之间的一个无符号整型，这个整数代表服务器的编号；多个服务器都通过这种方式在 hash 环上映射一个点来标识该服务器的位置；当用户操作某个 key，通过同样的算法生成一个值，沿环顺时针定位某个服务器，那么该 key 就在该服务器中；图片来源于互联网；

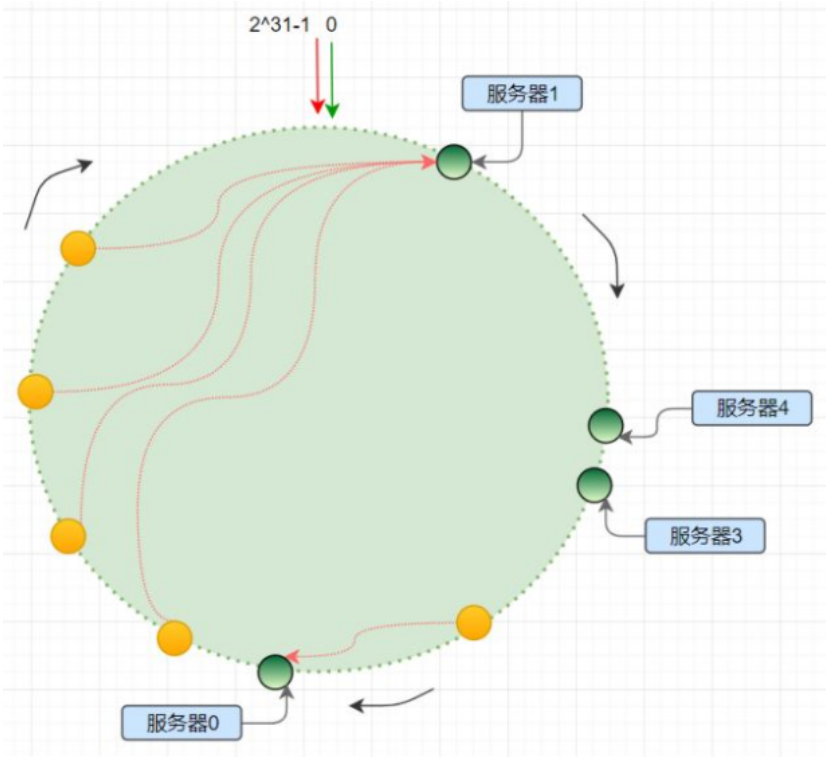


应用场景

> 分布式缓存；将数据均衡地分散在不同的服务器当中，用来分摊缓存服务器的压力；
>
> 解决缓存服务器数量变化尽量不影响缓存失效；

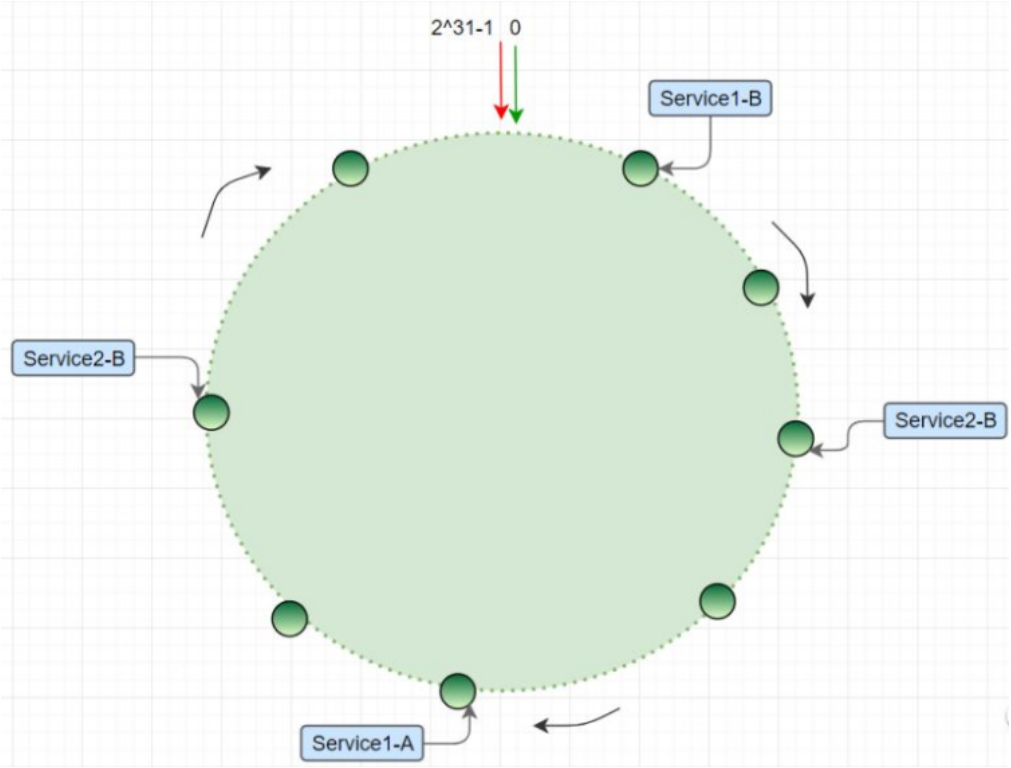
hash 偏移

> hash 算法得到的结果是随机的，不能保证服务器节点均匀分布在哈希环上；分布不均匀造成请求访问不均匀，服务器承受的压力不均匀；图片来源于互联网；



虚拟节点

> 为了解决哈希偏移的问题，增加了虚拟节点的概念；理论上，哈希环上节点数越多，数据分布越均衡；
>
> 为每个服务节点计算多个哈希节点（虚拟节点）；通常做法是，`hash("IP:PORT:seqno")` $\%2^{32}$；
>
> 图片来源于互联网；



思考

- 分布式一致性 hash 增加或者删除节点怎么进行数据迁移?

  参考：https://github.com/metang326/consistent_hashing_cpp