8-HashTable-哈希表/散列表

----------比特科技整理----------

HashTable-散列表/哈希表，是根据关键字（key）而直接访问在内存存储位置的数据结构。
它通过一个关键值的函数将所需的数据映射到表中的位置来访问数据，这个映射函数叫做散列函数，存放记录的数组叫做散列表。

构造哈希表的几种方法

1. 直接定址法--取关键字的某个线性函数为散列地址，Hash（Key）= Key 或 Hash（Key）= A*Key + B，A、B为常数。
2. 除留余数法--取关键值被某个不大于散列表长m的数p除后的所得的余数为散列地址。Hash（Key）= Key % P。
3. 平方取中法
4. 折叠法
5. 随机数法
6. 数学分析法

散列表.zip

- 哈希冲突/哈希碰撞

不同的Key值经过哈希函数Hash(Key)处理以后可能产生相同的值哈希地址，我们称这种情况为哈希冲突。任意的散列函数都不能避免产生冲突。

> 散列表的载荷因子定义为：α = 填入表中的元素个数 / 散列表的长度
>
> α是散列表装满程度的标志因子。由于表长是定值，α与“填入表中的元素个数”成正比，所以，α越大，表明填入表中的元素越多，产生冲突的可能性就越大；反之，α越小，标明填入表中的元素越少，产生冲突的可能性就越小。实际上，散列表的平均查找长度是载荷因子α的函数，只是不同处理冲突的方法有不同的函数。
>
> 对于开放定址法，荷载因子是特别重要因素，应严格限制在0.7-0.8以下。超过0.8，查表时的CPU缓存不命中（cache missing）按照指数曲线上升。因此，一些采用开放定址法的hash库，如Java的系统库限制了荷载因子为0.75，超过此值将resize散列表。

- 处理哈希冲突的闭散列方法

1. 线性探测
2. 二次探测

【线性探测】

89 18 49 58 9

Hash(key)+0, Hash(key)+1, . . . , Hash(key)+i

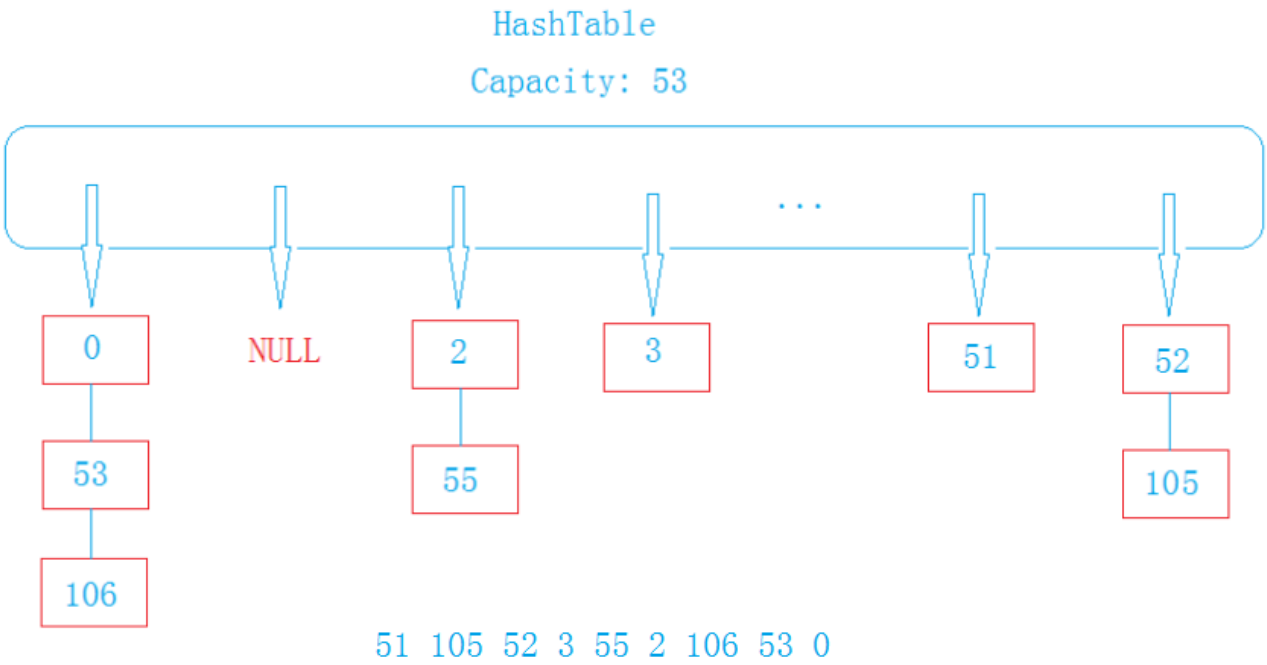
【二次探测】

89 18 49 58 9

Hash(key)+0, Hash(key)+1^2, . . . , Hash(key)+i^2

```
Hash(i) = Hash(0) + i^2
Hash(i-1) = Hash(0) + (i-1)^2
Hash(i) = Hash(i-1) + (2i-1)
```

- 处理哈希冲突的开链法(哈希桶)



- 素数表(使用素数做除数可以减少哈希冲突)

```
//使用素数表对齐做哈希表的容量，降低哈希冲突
const int _PrimeSize= 28;
static const unsigned long _PrimeList[_PrimeSize] =
{
    53ul,         97ul,         193ul,       389ul,       769ul,
    1543ul,       3079ul,       6151ul,      12289ul,     24593ul,
    49157ul,      98317ul,      196613ul,    393241ul,
786433ul,
    1572869ul,    3145739ul,    6291469ul,   12582917ul,
25165843ul,
    50331653ul,   100663319ul,  201326611ul, 402653189ul,
805306457ul,
    1610612741ul, 3221225473ul, 4294967291ul
};
```

- 字符串哈希算法

http://www.cnblogs.com/-clq/archive/2012/05/31/2528153.html

字符串哈希算法.zip

```
static size_t BKDRHash(const char *str)
{
    unsigned int seed= 131;// 31 131 1313 13131 131313
    unsigned int hash= 0;
    while (*str)
    {
        hash=hash*seed+ (*str++);
    }
    return (hash& 0x7FFFFFFF);
}
```

```
/// @brief BKDR Hash Function
/// @detail 本 算法由于在Brian Kernighan与Dennis Ritchie的
《The C Programming Language》一书被展示而得 名，是一种简单快捷的hash算法，也
是Java目前采用的字符串的Hash算法（累乘因子为31）。
template<class T>
size_t BKDRHash(const T *str)
{
    register size_t hash = 0;
    while (size_t ch = (size_t)*str++)
    {
        hash = hash * 131 + ch;   // 也可以乘以31、131、1313、13131、
131313..
        // 有人说将乘法分解为位运算及加减法可以提高效率，如将上式表达为：
hash = hash << 7 + hash << 1 + hash + ch;
        // 但其实在Intel平台上，CPU内部对二者的处理效率都是差不多的，
        // 我分别进行了100亿次的上述两种运算，发现二者时间差距基本为0（如果是
Debug版，分解成位运算后的耗时还要高1/3）；
        // 在ARM这类RISC系统上没有测试过，由于ARM内部使用Booth's Algorithm来
模拟32位整数乘法运算，它的效率与乘数有关：
        // 当乘数8-31位都为1或0时，需要1个时钟周期
        // 当乘数16-31位都为1或0时，需要2个时钟周期
        // 当乘数24-31位都为1或0时，需要3个时钟周期
        // 否则，需要4个时钟周期
        // 因此，虽然我没有实际测试，但是我依然认为二者效率上差别不大
    }
    return hash;
}
```

- 哈希扩展学习--布隆过滤器

https://segmentfault.com/a/1190000002729689
http://www.cnblogs.com/haippy/archive/2012/07/13/2590351.html

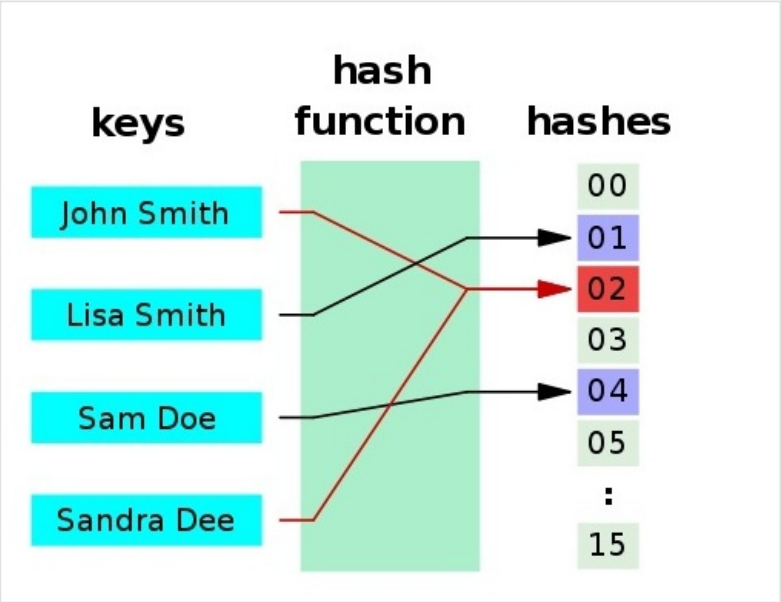
BloomFilter.zip

在日常生活中，包括在设计计算机软件时，我们经常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是要判断 它是否在已知的字典中）；在 FBI，一个嫌疑人的名字是否已经在嫌疑名单上；在网络爬虫里，一个网址是否被访问过等等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新 元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表（hash table）来存储的。它的好处是快速准确，缺点是费存储空间。当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来 了。比如说，一个象 Yahoo,Hotmail 和 Gmai 那样的公众电子邮件（email）提供商，总是需要过滤来自发送垃圾邮件的人（spamer）的垃圾邮件。一个办法就是记录下那些发垃圾邮件的 email 地址。由于那些发送者不停地在注册新的地址，全世界少说也有几十亿个发垃圾邮件的地址，将他们都存起来则需要大量的网络服务器。如果用哈希表，每存储一亿 个 email 地址， 就需要 1.6GB 的内存（用哈希表实现的具体办法是将每一个 email 地址对应成一个八字节的信息指纹（详见：googlechinablog.com/2006/08/blog-post.html），然后将这些信息指纹存入哈希表，由于哈希表的存储效率一般只有 50%，因此一个 email 地址需要占用十六个字节。一亿个地址大约要 1.6GB，即十六亿字节的内存）。因此存贮几十亿个邮件地址可能需要上百 GB 的内存。除非是超级计算机，一般服务器是无法存储的［2］。（该段引用谷歌数学之美：http://www.google.com.hk/ggblog/googlechinablog/2007/07/bloom-filter_7469.html）

基本概念

如果想判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表，树等等数据结构都是这种思路. 但是随着集合中元素的增加，我们需要的存储空间越来越大，检索速度也越来越慢。不过世界上还有一种叫作散列表（又叫哈希表，Hash table）的数据结构。它可以通过一个Hash函数将一个元素映射成一个位阵列（Bit Array）中的一个点。这样一来，我们只要看看这个点是不是 1 就知道可以集合中有没有它了。这就是布隆过滤器的基本思想。

Hash面临的问题就是冲突。假设 Hash 函数是良好的，如果我们的位阵列长度为 m 个点，那么如果我们想将冲突率降低到例如 1%, 这个散列表就只能容纳 m/100 个元素。显然这就不叫空间有效了（Space-efficient）。解决方法也简单，就是使用多个 Hash，如果它们有一个说元素不在集合中，那肯定就不在。如果它们都说在，虽然也有一定可能性它们在说谎，不过直觉上判断这种事情的概率是比较低的。



Bloom Filter 用例

Google 著名的分布式数据库 Bigtable 使用了布隆过滤器来查找不存在的行或列，以减少磁盘查找的IO次数［3］。

Squid 网页代理缓存服务器在 cache digests 中使用了也布隆过滤器［4］。

Venti 文档存储系统也采用布隆过滤器来检测先前存储的数据［5］。

SPIN 模型检测器也使用布隆过滤器在大规模验证问题时跟踪可达状态空间［6］。

Google Chrome浏览器使用了布隆过滤器加速安全浏览服务［7］。

在很多Key-Value系统中也使用了布隆过滤器来加快查询过程，如 Hbase，Accumulo，Leveldb，一般而言，Value 保存在磁盘中，访问磁盘需要花费大量时间，然而使用布隆过滤器可以快速判断某个Key对应的Value是否存在，因此可以避免很多不必要的磁盘IO操作，只是引入布隆过滤器会带来一定的内存消耗，下图是在Key-Value系统中布隆过滤器的典型使用：