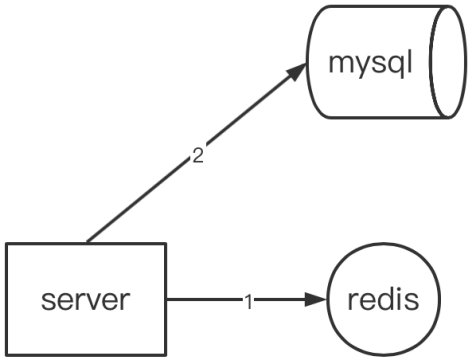
零声学院出品 Mark老师QQ：2548898954

源码地址：http://gitlab.0voice.com/0voice/bloomfilter/tree/master

背景

- 在使用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？允许有误差
- 垃圾邮件（短信）过滤算法如何设计？允许有误差
- 公安办案时，如何判断某嫌疑人是否在网逃名单中？控制误差 假阳率
- 缓存穿透问题如何解决？允许有误差



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；

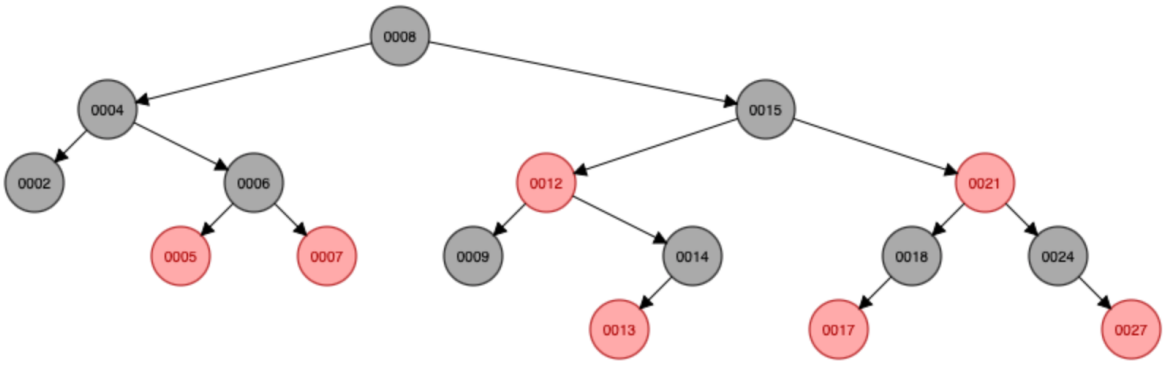
  - 描述缓存场景，为了减轻落盘数据库（mysql）的访问压力，在server端与mysql之间加入一层缓冲数据层（用来存放热点数据）；
  - 缓存穿透发生的场景是server端向数据库**请求数据时**，缓存数据库（redis）和落盘数据库（mysql）**都不包含该数据**，数据请求压力全部涌向落盘数据库（mysql）。
  - 数据请求步骤：如上图 2 的描述；
  - 发生原因：黑客利用漏洞伪造数据攻击或者内部业务bug重复大量请求不存在的数据；
  - 解决方法：如上图 3 的描述；

需求

- 从海量数据中查询某字符串是否存在。

set和map

- c++标准库（STL）中的set和map结构都是采用红黑树实现的，它增删改查的时间复杂度是 $o(\log_2 n)$；
- 图结构示例：



- 对于严格平衡二叉搜索树(AVL)，100w条数据组成的红黑树，只需要**比较20次**就能找到该值；对于10亿条数据只需要**比较30次**就能找到该数据；也就是查找次数跟树的高度是一致的；
- 对于红黑树来说平衡的是黑节点高度，所以研究比较次数需要考虑树的高度差，最好情况某条树链路全是黑节点，假设此时高度为h1，最差情况某条树链路全是黑红节点间隔，那么此时树高度为2*h1;
- 在红黑树中每一个节点都存储key和val字段，key是用来做比较的字段；红黑树并没有要求key字段唯一，在set和map实现过程中限制了key字段唯一。我们来看nginx的红黑树实现：

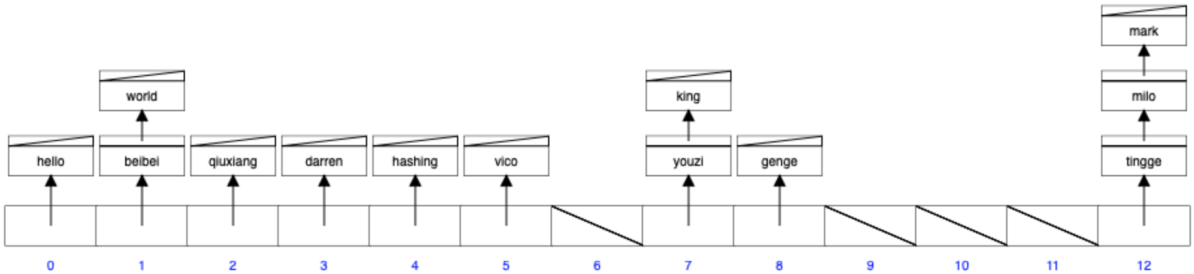
```
// 这个是截取 nginx 的红黑树的实现，这段代码是 insert 操作中的一部分，执行完这个函数还需要检测插入节点后是否平衡（主要是看他的父节点是否也是红色节点）
// 调用 ngx_rbtree_insert_value 时，temp传的参数为 红黑树的根节点，node传的参数为待插入的节点
void ngx_rbtree_insert_value(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
    ngx_rbtree_node_t *sentinel)
{
  ngx_rbtree_node_t  **p;
  for ( ;; ) {
    p = (node->key < temp->key) ? &temp->left : &temp->right;// 这行很重要
    if (*p == sentinel) {
      break;
    }
    temp = *p;
  }
  *p = node;
  node->parent = temp;
  node->left = sentinel;
  node->right = sentinel;
  ngx_rbt_red(node);
}
// 不插入相同节点    如果插入相同 让它变成修改操作    此时 红黑树当中就不会有相同的key了 定时器 key 时间戳
// 如果我们插入key = 12，如上图红黑树，12号节点应该在哪个位置？ 如果我们要实现插入存在的节点变成修改操作，该怎么改上面的函数
void ngx_rbtree_insert_value_ex(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
    ngx_rbtree_node_t *sentinel)
{
```

```
    ngx_rbtree_node_t  **p;
    for ( ;; ) {
//  {-------------add-------------
        if (node->key == temp->key) {
            temp->value = node->value;
            return;
        }
//  }-------------add-------------
        p = (node->key < temp->key) ? &temp->left : &temp->right;// 这行很重要
        if (*p == sentinel) {
            break;
        }
        temp = *p;
    }
    *p = node;
    node->parent = temp;
    node->left = sentinel;
    node->right = sentinel;
    ngx_rbt_red(node);
}
```

- 另外set和map的关键区别是set不存储val字段；
- 优点：存储效率高，访问速度高效；
- 缺点：对于数据量大且查询字符串比较长且查询字符串相似时将会是噩梦；

unordered_map

- c++标准库（STL）中的unordered_map<string, bool>是采用hashtable实现的；
- 构成：数组+hash函数；
- 它是将字符串通过hash函数生成一个整数再映射到数组当中；它增删改查的时间复杂度是o(1);
- 图结构示例：



- hash函数的作用：**避免插入的时候字符串的比较**；hash函数计算出来的值通过对数组长度的取模能**随机分布**在数组当中；
- hash函数一般返回的是64位整数，将多个大数映射到一个小数组中，必然会产生冲突；
- 如何选取hash函数?
    1. 选取计算速度快；
    2. 哈希相似字符串能保持**强随机分布性**（防碰撞）；

- 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

- 负载因子：数组存储元素的个数/数组长度；负载因子越小，冲突越小；负载因子越大，冲突越大；
- hash冲突解决方案：
  - 链表法

    引入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；

    2. 具体原理：https://www.cnblogs.com/organic/p/6283476.html

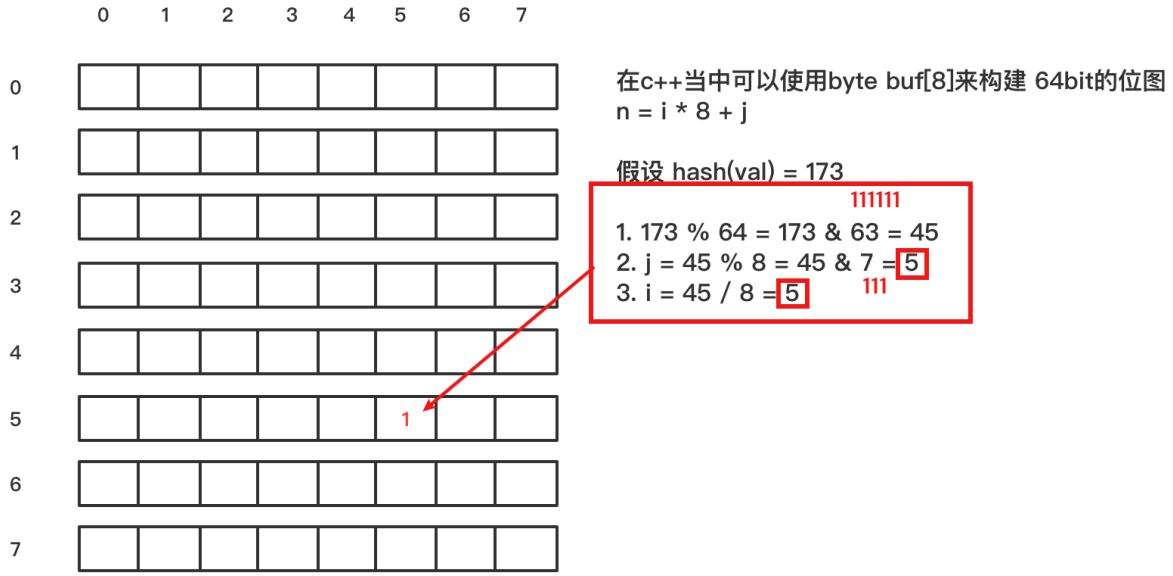
- 同样的hashtable中节点存储了key和val，hashtable并没有要求key的大小顺序，我们同样可以修改代码让插入存在的数据变成修改操作；
- 优点：访问速度更快；**不需要进行字符串比较**；
- 缺点：需要引入策略避免冲突，存储效率不高；空间换时间；

总结

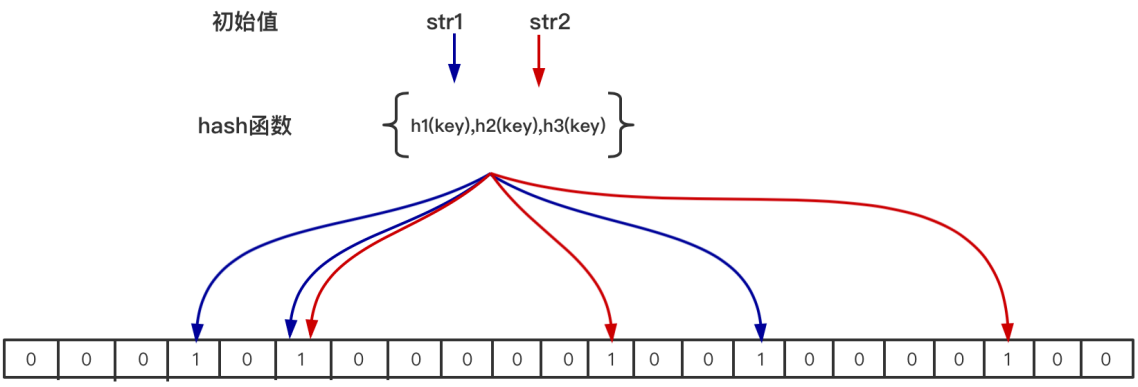
- 红黑树和hashtable都不能解决海量数据问题，它们都需要存储具体字符串，如果数据量大，提供不了几百G的内存；所以需要尝试探寻不存储key的方案，并且拥有hashtable的优点（不需要比较字符串）；

布隆过滤器

- 定义：布隆过滤器是一种**概率型**数据结构，它的特点是高效的**插入和查询**，能明确告知某个字符串**一定不存在**或者**可能存在**；
- 布隆过滤器相比传统的查询结构（例如：hash，set，map等数据结构）**更加高效，占用空间更小**；但是**其缺点**是它返回的结果是概率性的，也就是说**结果存在误差**的，虽然这个**误差是可控的**；同时它**不支持删除操作**；
- 组成：位图（bit数组）+ n个hash函数



- 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，那么认为不存在；如果全部为1，则可能存在（存在误差）；



- 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，但不明确它被设置了多少次；也就是不知道**被多少个str1哈希映射**以及是**被哪个hash函数**映射过来的；所以不支持删除操作；
- 在实际应用过程中，布隆过滤器该如何使用？要选择多少个hash函数，要分配多少空间的位图，存储多少元素？另外如何控制假阳率（布隆过滤器能明确一定不存在，不能明确一定存在，那么存在的判断是有误差的，假阳率就是错误判断存在的概率）？

```
n        -- 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));
```

- 假定我们选取这四个值为：

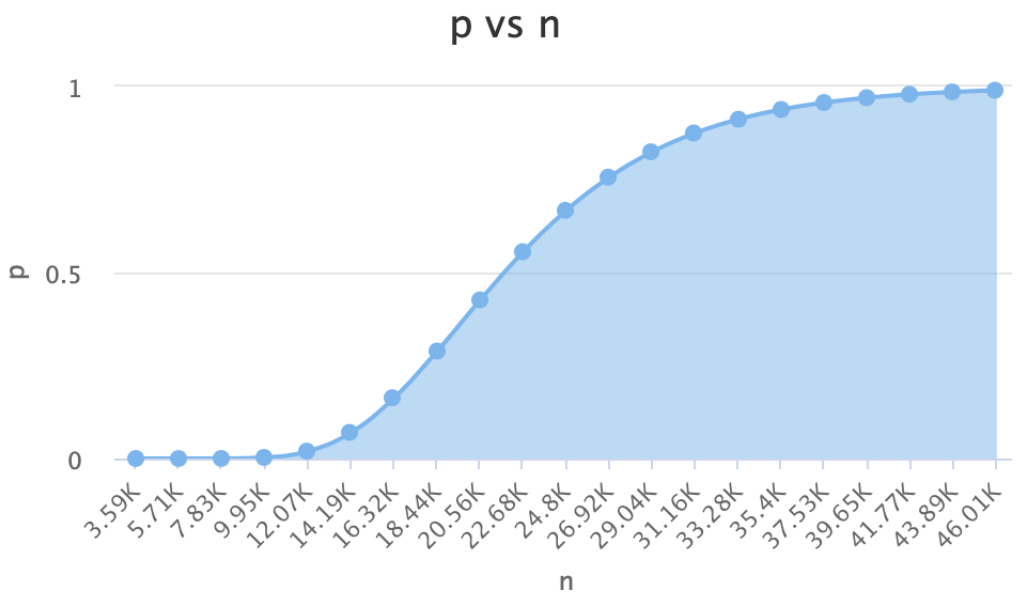
  n = 4000

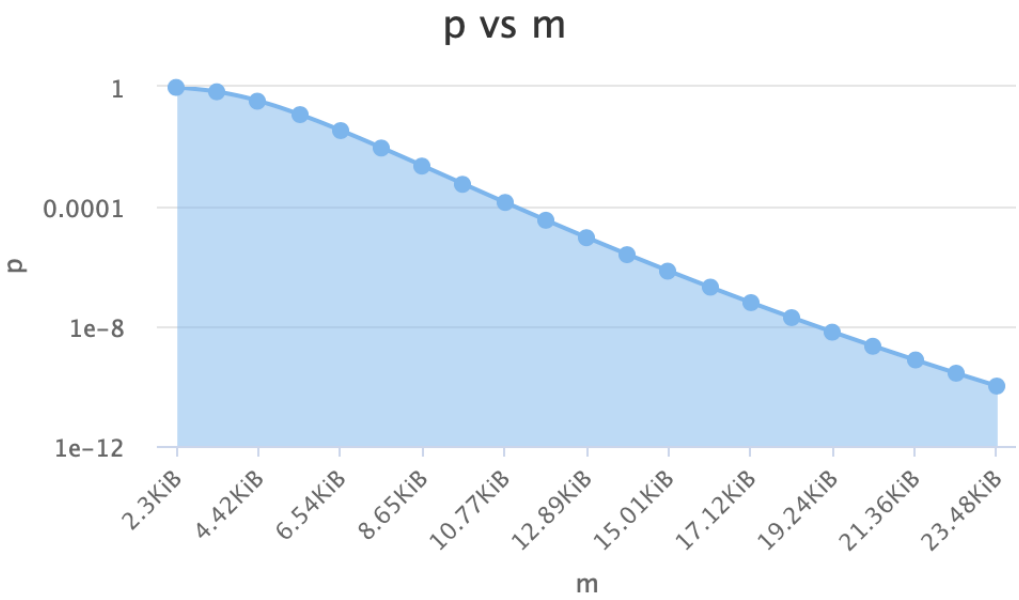
  p = 0.000000001

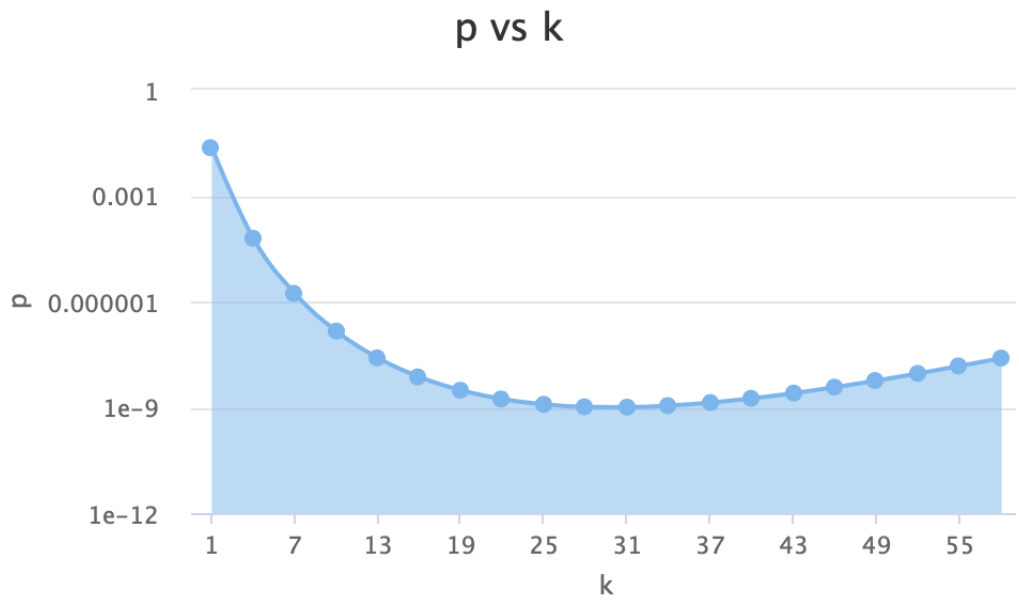
  m = 172532

  k = 30

- 四个值的关系：







- 在实际应用中，我们确定n和p，通过上面的计算算出m和k；也可以在网站上选取合适的值：https://hur.st/bloomfilter
- 已知k，如何选择k个hash函数?

```
// 采用一个hash函数，给hash传不同的种子偏移值
// #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
}
// 通过这种方式来模拟 k 个hash函数 跟我们前面开放寻址法 双重hash是一样的思路
```

- 应用源码：http://gitlab.0voice.com/0voice/bloomfilter/tree/master