避免缓存击穿的利器之BloomFilter

Bloom Filter 概念

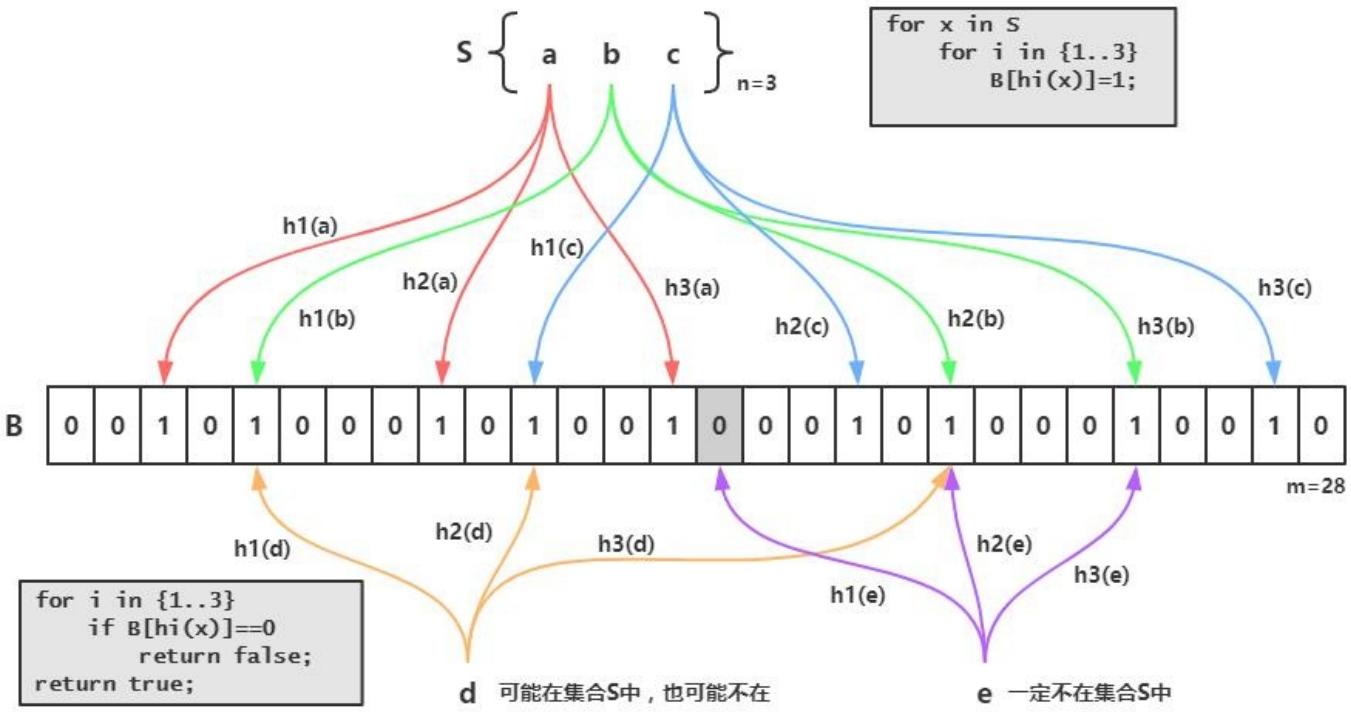
布隆过滤器（英语：Bloom Filter）是1970年由一个叫布隆的小伙子提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

面试关联：一般都会在回答缓存穿透，或者海量数据去重这个时候引出来，加分项哟

Bloom Filter 原理

布隆过滤器的原理是，当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点，把它们置为1。检索时，我们只要看看这些点是不是都是1就（大约）知道集合中有没有它了：如果这些点有任何一个0，则被检元素一定不在；如果都是1，则被检元素很可能在。这就是布隆过滤器的基本思想。

Bloom Filter跟单哈希函数Bit-Map不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。



简单的说一下就是我们先把我们数据库的数据都加载到我们的过滤器中，比如数据库的id现在有：1、2、3

那就用id：1 为例子他在上图中经过三次hash之后，把三次原本值0的地方改为1

下次我进来查询如果id也是1 那我就把1拿去三次hash 发现跟上面的三个位置完全一样，那就能证明过滤器中有1的

反之如果不一样就说明不存在了

那应用的场景在哪里呢？一般我们都会用来防止缓存击穿（如果不知道缓存击穿是啥的小伙伴不要着急，我已经帮你准备好了，传送门↓）

简单来说就是你数据库的id都是1开始然后自增的，那我知道你接口是通过id查询的，我就拿负数去查询，这个时候，会发现缓存里面没这个数据，我又去数据库查也没有，一个请求这样，100个，1000个，10000个呢？你的DB基本上就扛不住了，如果在缓存里面加上这个，是不是就不存在了，你判断没这个数据就不去查了，直接return一个数据为空不就好了嘛。

这玩意这么好使那有啥缺点么？有的，我们接着往下看

Bloom Filter的缺点

bloom filter之所以能做到在时间和空间上的效率比较高，是因为牺牲了判断的准确率、删除的便利性

- 存在误判，可能要查到的元素并没有在容器中，但是hash之后得到的k个位置上值都是1。如果bloom filter中存储的是黑名单，那么可以通过建立一个白名单来存储可能会误判的元素。
- 删除困难。一个放入容器的元素映射到bit数组的k个位置上是1，删除的时候不能简单的直接置为0，可能会影响其他元素的判断。可以采用Counting Bloom Filter

Bloom Filter 实现

布隆过滤器有许多实现与优化，Guava中就提供了一种Bloom Filter的实现。

在使用bloom filter时，绕不过的两点是预估数据量n以及期望的误判率fpp，

在实现bloom filter时，绕不过的两点就是hash函数的选取以及bit数组的大小。

对于一个确定的场景，我们预估要存的数据量为n，期望的误判率为fpp，然后需要计算我们需要的Bit数组的大小m，以及hash函数的个数k，并选择hash函数

(1)Bit数组大小选择

根据预估数据量n以及误判率fpp，bit数组大小的m的计算方式：$$m = -\frac{nlnfpp}{(ln2)^2}$$

(2)哈希函数选择

由预估数据量n以及bit数组长度m，可以得到一个hash函数的个数k：$$\mathrm{k} = \frac{m}{n} ln2$$

哈希函数的选择对性能的影响应该是很大的，一个好的哈希函数要能近似等概率的将字符串映射到各个Bit。选择k个不同的哈希函数比较麻烦，一种简单的方法是选择一个哈希函数，然后送入k个不同的参数。

哈希函数个数k、位数组大小m、加入的字符串数量n的关系可以参考Bloom Filters - the math，Bloom_filter-wikipedia

要使用BloomFilter，需要引入guava包：

```
 <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>23.0</version>
 </dependency>
```

测试分两步：

1、往过滤器中放一百万个数，然后去验证这一百万个数是否能通过过滤器

2、另外找一万个数，去检验漏网之鱼的数量

```
/**
 * 测试布隆过滤器(可用于redis缓存穿透)
 *
 * @author 敖丙
 */
public class TestBloomFilter {

    private static int total = 1000000;
    private static BloomFilter<Integer> bf =
BloomFilter.create(Funnels.integerFunnel(), total);
//    private static BloomFilter<Integer> bf =
BloomFilter.create(Funnels.integerFunnel(), total, 0.001);

    public static void main(String[] args) {
        // 初始化1000000条数据到过滤器中
        for (int i = 0; i < total; i++) {
            bf.put(i);
        }

        // 匹配已在过滤器中的值，是否有匹配不上的
        for (int i = 0; i < total; i++) {
            if (!bf.mightContain(i)) {
                System.out.println("有坏人逃脱了~~~");
            }
        }

        // 匹配不在过滤器中的10000个值，有多少匹配出来
        int count = 0;
        for (int i = total; i < total + 10000; i++) {
            if (bf.mightContain(i)) {
                count++;
            }
        }
        System.out.println("误伤的数量: " + count);
    }

}
```

运行结果：

```
<terminated> TestBloomFilter [Java Application] C:\Program Files\Java\jdk1.8.0_172\b
误伤的数量：320
```

运行结果表示，遍历这一百万个在过滤器中的数时，都被识别出来了。一万个不在过滤器中的数，误伤了320个，错误率是0.03左右。

看下BloomFilter的源码：

```
public static <T> BloomFilter<T> create(Funnel<? super T> funnel, int
expectedInsertions) {
        return create(funnel, (long) expectedInsertions);
    }

    public static <T> BloomFilter<T> create(Funnel<? super T> funnel, long
expectedInsertions) {
        return create(funnel, expectedInsertions, 0.03); // FYI, for 3%, we always
get 5 hash functions
    }

    public static <T> BloomFilter<T> create(
          Funnel<? super T> funnel, long expectedInsertions, double fpp) {
        return create(funnel, expectedInsertions, fpp,
BloomFilterStrategies.MURMUR128_MITZ_64);
    }

    static <T> BloomFilter<T> create(
      Funnel<? super T> funnel, long expectedInsertions, double fpp, Strategy
strategy) {
     ......
    }
```

BloomFilter一共四个create方法，不过最终都是走向第四个。看一下每个参数的含义：

funnel：数据类型(一般是调用Funnels工具类中的)

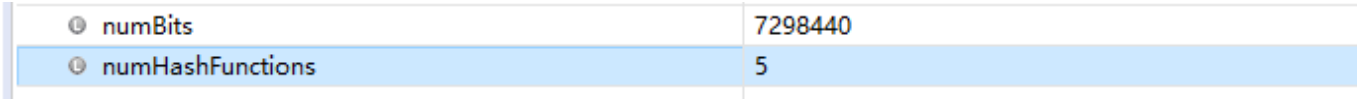
expectedInsertions：期望插入的值的个数

fpp 错误率(默认值为0.03)

strategy 哈希算法(我也不懂啥意思)Bloom Filter的应用

在最后一个create方法中，设置一个断点：

```
  long numBits = optimalNumOfBits(expectedInsertions, fpp);
  int numHashFunctions = optimalNumOfHashFunctions(expectedInsertions, numBits);
  try {
    return new BloomFilter<T>(new LockFreeBitArray(numBits), numHashFunctions, funnel, strategy);
  } catch (IllegalArgumentException e) {
    throw new IllegalArgumentException("Could not create BloomFilter of " + numBits + " bits", e);
  }
}
```



numBits	7298440
numHashFunctions	5

上面的numBits，表示存一百万个int类型数字，需要的位数为7298440，700多万位。理论上存一百万个数，一个int是4字节32位，需要4*8*1000000=3200万位。如果使用HashMap去存，按HashMap50%的存储效率，需要6400万位。可以看出BloomFilter的存储空间很小，只有HashMap的1/10左右

上面的numHashFunctions，表示需要5个函数去存这些数字

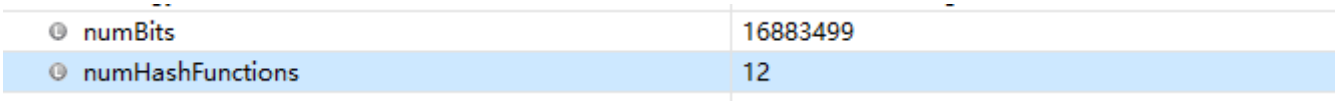
使用第三个create方法，我们设置下错误率：

```
private static BloomFilter<Integer> bf =
BloomFilter.create(Funnels.integerFunnel(), total, 0.0003);
```

再运行看看：

```
<terminated> TestBloomFilter [Java Application] C:\Program Files\Java\jdk1.8.0_172\bin\javaw.exe (2018年9月
误伤的数量：4
```

此时误伤的数量为4，错误率为0.04%左右。



numBits	16883499
numHashFunctions	12

当错误率设为0.0003时，所需要的位数为16883499，1600万位，需要12个函数

和上面对比可以看出，错误率越大，所需空间和时间越小，错误率越小，所需空间和时间约大

常见的几个应用场景：

- cerberus在收集监控数据的时候, 有的系统的监控项量会很大, 需要检查一个监控项的名字是否已经被记录到db过了, 如果没有的话就需要写入db.
- 爬虫过滤已抓到的url就不再抓，可用bloom filter过滤
- 垃圾邮件过滤。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，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。而Bloom Filter只需要哈希表 1/8到 1/4 的大小就能解决同样的问题。

觉得有用的话欢迎 关注 点赞 分享 【敖丙】| 文