前言
JavaPub说
关于布隆过滤器
1.介绍
1.1.基础介绍
1.1.1.百度百科
1.1.2.原理介绍
1.1.3.布隆过滤器的属性
1.2.数学推导
1.3.哈希
2.基础用法
2.1.Java版
3.进阶
3.1.进阶一(参数定义)
3.1.1.介绍
3.1.2.Java实现
3.2.进阶二(redis版)
3.2.1.介绍
3.2.2.Java代码
扩展阅读

前言

> 声明：参考来源互联网，有任何争议可以留言。站在前人的肩上，我们才能看的更远。

> 本教程纯手打，致力于最实用教程，不需要什么奖励，只希望多多转发支持。
> 欢迎来我公众号，希望可以结识你，也可以催更，微信搜索：JavaPub

> 有任何问题都可以来谈谈，等你！

JavaPub说

布隆大家都知道吧，如果不知道没关系，介绍一下，E技能，坚不可摧

> 坚不可摧　E 消耗法力：30/35/40/45/50冷却时间：18/16/14/12/10　布隆朝一个方向举起盾牌，持续3/3.25/3.5/3.75/4秒，并使来自目标

回忆完以上，下面继续

关于布隆过滤器

- 布隆过滤器主要用来做去重操作。在对准确率要求不高的业务场景使用广泛。
- 布隆过滤器的核心是：**如果计算出有一个元素已存在，那么它可能存在，如果一个元素不存在，那么它一定不存在。**
- 简单来说，宁错杀三千，不放过一个。
- 例如：长城防火墙有100亿个需要屏蔽的网站，计算机的每次请求都要经过防火墙的过滤判断请求URL是否在黑名单中，如果我们使用HashSet来实现过滤的话，我们假设每个URL的大小为64B，那么100亿个就至少需要大约640GB的内存空间，这显然是不符合实际情况的。
- 到目前我使用比较多的是在数据采集中，url去重，邮箱中的垃圾邮件过滤等。

1.介绍

1.1.基础介绍

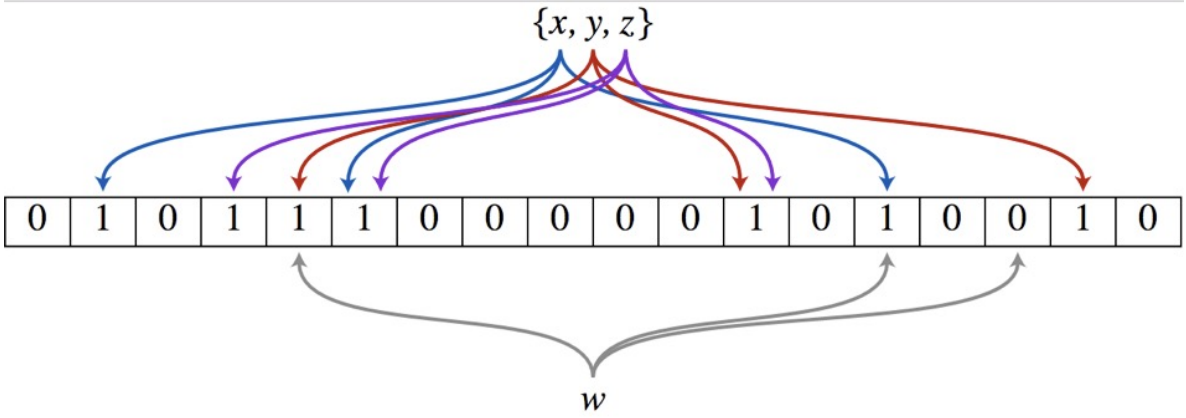
1.1.1.百度百科

> 百度百科：布隆过滤器（Bloom Filter）是1970年由布隆提出的。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比一般的算法要好的多，缺点是有一定的误识别率和删除困难。

1.1.2.原理介绍

布隆过滤器的原理和哈希表的原理有点类似，同样需要使用hash函数，但是在布隆过滤器中，需要使用多个hash函数。布隆过滤器的原理还是比较简单的。

我们有一个位数组bitArray，假设长度为m，就是只存0、1那种。此时有个key，和n个hash函数，可以得到n个key被hash过后的数。我们分别取hash对应bitArray中位置的值，置位1。



> 如图 {x,y,z} 是一个集合，通过三次hash计算，映射对应的值到 位数组 对应位置。当我们要求 **w** 是否存在时，只要对w计算hash，再找对应位置是否为1即可。但是，也有可能正好hash值对应的 位数组 位置都为1，这个概念叫做**误算率**。实际上，这就和哈希表中哈希冲突的情况一样，因为可能会出现两个key值经过k个hash函数之后，取余之后的结果是一样的。

1.1.3.布隆过滤器的属性

1. 如果布隆过滤器判断数据不存在则数据绝对不存在。
2. 这个就是布隆过滤器的特点，数据先经过布隆过滤器，查询数据是否已经存在。如果布隆过滤器判断用户名不存在/或者存在，数据才能够继续向下走。
3. 在前面的判断中，可以判断数据绝对不存在，但是如果判断数据存在，则数据也可能不存在。
4. 布隆过滤器只能插入数据，而不能删除数据。

1.2.数学推导

既然误算率一定存在，当然我们想减小误判到最小（key数量和bitArray长度确定）。

- 数学公式

$$m = -\frac{nlnfpp}{(ln2)^2},\ \mathrm{k} = \frac{m}{n}ln2 = -\frac{lnfpp}{ln2}$$

1.3.哈希

读到这里我们对布隆过滤器有了一定了解，hash函数对 布隆过滤器 的优劣起了决定性作用。

Hash参考百度百科：https://baike.baidu.com/item/hash/390310

目标就是设计一种尽可能少碰撞的hash算法，尽可能让它平均分布到每一位。

2.基础用法

2.1.Java版

- 下面是一篇简单版本的布隆过滤器，使用了 java.util.BitSet

```
package com.javapub.cache;

import java.util.BitSet;

/**
 * @author wangshiyu rodert JavaPub
 * @date 2020/5/26 15:23
 * @description 一篇简单的布隆过滤器
 */
public class BloomFilterSimple {
    private static final int SIZE = 1 << 24;
    private static BitSet bitSet = new BitSet(SIZE);
    private static Hash[] hashes = new Hash[5];
    private static final int seeds[] = new int[]{3, 5, 7, 9, 11};

    static {
        init();
    }

    private static void init() {
        for (int i = 0; i < seeds.length; i++) {
            hashes[i] = new Hash(seeds[i]);
        }
    }

    private boolean add(String data) {
        for (Hash hash : hashes) {
            int hashCode = hash.getHash(data);
            bitSet.set(hashCode, true);
        }
        return true;
    }

    private boolean contains(String data) {
        boolean have = true;
        for (Hash hash : hashes) {
            have &= bitSet.get(hash.getHash(data));
        }
```

```
        return have;
    }

    /**
     * 如果不存在就进行记录并返回false，如果存在了就返回true
     *
     * @param data
     * @return
     */
    private boolean addIfNotExist(String data) {
        boolean contains = contains(data);
        if (contains) {
            return true;
        } else {
            add(data);
            return false;
        }
    }

    public static void main(String[] args) {

        String data = "https://gitee.com/rodert";
        String data2 = "https://gitee.com/rodert/JavaPub";
        BloomFilterSimple bloomFilterSimple = new BloomFilterSimple();
        System.out.println(bloomFilterSimple.add(data));
        System.out.println(bloomFilterSimple.contains(data));
        System.out.println(bloomFilterSimple.addIfNotExist(data2));
        System.out.println(bloomFilterSimple.contains(data2));

        System.out.println(bitSet);

    }

    private static class Hash {
        private int seed = 0;

        private Hash(int seed) {
            this.seed = seed;
        }

        private int getHash(String string) {
            int val = 0;
            int len = string.length();
            for (int i = 0; i < len; i++) {
                val = val * seed + string.charAt(i);
            }
            return val & (SIZE - 1);
        }
    }
}
```

3.进阶

3.1.进阶一(参数定义)

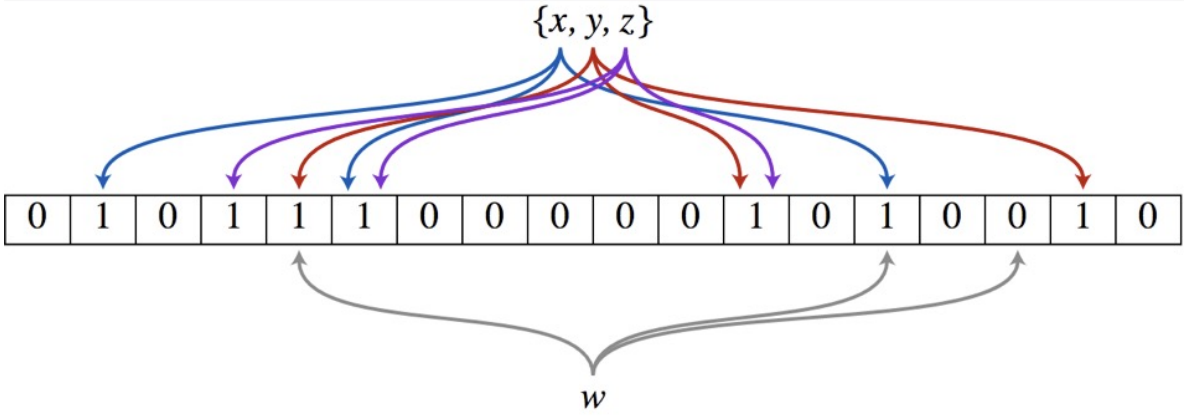
3.1.1.介绍

> 如果你有兴趣了解更多，我们继续往下看

- 前面写了一个简单的DEMO，位数组长度和误差率都是拍脑袋定的，这篇主要讲解如何定义合适的**位数组**长度，计算方式

1.1.2.原理介绍

我们有一个位数组bitArray，假设长度为m，就是只存0、1那种。此时有个key，和k个hash函数，可以得到k个key被hash过后的数。我们分别取hash对应bitArray中位置的值，置位1。



> 如图 {x,y,z} 是一个集合，通过三次hash计算，映射对应的值到 位数组 对应位置。当我们要求 **w** 是否存在时，只要对w计算hash，再找对应位置是否为1即可。但是，也有可能正好hash值对应的 位数组 位置都为1，这个概念叫做**误算率**。实际上，这就和哈希表中哈希冲突的情况一样，因为可能会出现两个key值经过k个hash函数之后，取余之后的结果是一样的。

上面是我们在原理介绍讲到的，综上所述，我们需要多少个哈希函数，创建多长的bit数组比较合适，为了估算出k和m的值，在构造一个布隆过滤器时，需要传入两个参数，即可以接受的误判率fpp和元素总个数n（不一定完全精确）。至于参数估计的方法，有兴趣的同学可以参考维基英文页面，下面直接给出公式：

$$m = -\frac{nlnfpp}{(ln2)^2},\quad \mathrm{k} = \frac{m}{n}ln2 = -\frac{lnfpp}{ln2}$$

1. 哈希函数的要求尽量满足平均分布，这样既降低误判发生的概率，又可以充分利用bit数组的空间；
2. 根据论文《Less Hashing, Same Performance: Building a Better Bloom Filter》提出的一个技巧，可以用2个哈希函数来模拟k个哈希函数，即gi(x) = h1(x) + ih2(x) ，其中0<=i<=k-1；
3. 在吴军博士的《数学之美》一书中展示了不同情况下的误判率，例如，假定一个元素用16位比特，8个哈希函数，那么假阳性的概率是万分之五，这已经相当小了。

3.1.2.Java实现

- 计算 位数组长度

> n是准备存入数据数量，p是误判率。

```
public static long optimalNumOfBits(long n, double p) {
    return (long)((double)(-n) * Math.log(p) / (Math.log(2.0D) * Math.log(2.0D)));
}
```

- 计算hash函数个数

> n是准备存入数据数量，m是bit数组长度。

```
public static int optimalNumOfHashFunctions(long n, long m) {
    return Math.max(1, (int)Math.round((double)m / (double)n * Math.log(2.0D)));
}
```

3.2.进阶二(redis版)

3.2.1.介绍

> 布隆过滤器自提出以后，很多开源工具中都对它进行了实现。如 Google 的 Guava 中。
>
> 对于现在大趋势分布式架构，单机存到缓存肯定是适用场景有限，so，我们借助 redis。
>
> redis 数据类型 bit ，用法和上边一样，这里主要说关于动态扩容。

- 扩容的核心就是在每次插入前判断当前 位数组 为1(jedis.bitcount)的个数比(/) 位数组总长度，超过50%，那么就新建一个 bit，布隆过滤器的核心思想。判断一个元素是否在集合中？**可能在集合中** 和 **绝对不在集合中**

3.2.2.Java代码

> 由于篇幅过长，后面会在公众号单独发出

扩展阅读

布隆过滤器换包含：并行分区的布隆过滤器、稳定的布隆过滤器、可扩展的Bloom过滤器、空间布隆过滤器、衰减的布隆过滤器等。

更多阅读阅读维基百科英文：https://en.wikipedia.org/wiki/Bloom_filter