海量数据处理以及缓存穿透这两个场景让我认识了布隆过滤器。

#### 1、什么是布隆过滤器？

布隆过滤器（Bloom Filter）是一个叫做 Bloom 的老哥于1970年提出的。我们可以把它看作由二进制向量（或者说位数组）和一系列随机映射函数（哈希函数）两部分组成的数据结构。

优点：相比于我们平时常用的的 List、Map 、Set 等数据结构，它占用空间更少并且效率更高；

缺点：是其返回的结果是概率性的，而不是非常准确的。理论情况下添加到集合中的元素越多，误报的可能性就越大。并且，存放在布隆过滤器的数据不容易删除。



bit数组中的每个元素都只占用1bit，并且每个元素只能是0或1。这样申请一个100w个元素的位数只占用1000000Bit/8=125000Byte=125000/1024kb≈122kb的空间。

总结：一个名为Bloom的人提出了一种来检索元素是否在给定大集合中的数据结构，这种数据结构是高效且性能很好的，但缺点是具有一定的错误识别率和删除难度。并且，理论情况下，添加到集合中的元素越多，误报的可能性就越大。

#### 2、布隆过滤器的原理介绍

当一个元素加入布隆过滤器中的时候，会进行如下操作：

1. 使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。
2. 根据得到的哈希值，在bit数组中把对应下标的值置为1。

当我们需要判断一个元素是否存在于布隆过滤器的时候，会进行如下操作：

1. 对给定元素再次进行相同的哈希计算；
2. 得到值之后判断bit数组中的每个元素是否为1，如果值都为1，那么说明这个值在布隆过滤器中，如果存在一个值不为1，说明该元素不在布隆过滤器中。

简单例子：



如图所示，当字符串存储要加入布隆过滤器中时，该字符串首先由多个哈希函数生成不同的哈希值，然后在对应的bit数组的下标的元素设置为1（当bit数组初始化时，所有位置均为0）。当第二次存储相同字符串时，因为先前的对应位置已设置为1，所有很容易知道此值已经存在（去重非常方便）。

如果我们需要判断某个字符串是否在布隆过滤器中时，只需要对字符串再次进行相同的哈希计算，得到值之后判断bit数组中的每个元素是否都为1，如果值都为1，那么说明这个值在布隆过滤器中，如果存在一个值不为1，说明该元素不在布隆过滤器中。

不同的字符串可能哈希出来的位置相同，这种情况我们可以适当增加bit数组大小或者调整我们的哈希函数。

总结：布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不存在，那么这个元素一定不在。

#### 3、布隆过滤器使用场景

1. 判断给定数据是否存在：比如判断一个数字是否在于包含大量数字的数字集中（数字集很大，5亿以上！）、防止缓存穿透（判断请求的数据是否有效避免直接绕过缓存请求数据库）等等、邮箱的垃圾邮件过滤、黑名单功能等等。
2. 去重：比如爬给定网址的时候对已经爬取过的URL去重。

#### 4、通过Java编程手动实现布隆过滤器

知道了布隆过滤器的原理之后就可以自己手动实现一个了。

如果手动实现一个，需要：

1. 一个合适大小的bit数组保存数据。
2. 几个不同的哈希函数。
3. 添加元素到bit数组（布隆过滤器）的方法实现。
4. 判断给定元素是否存在于bit数组（布隆过滤器）的方法实现。

package com.chance.basis.filter;

import java.util.BitSet;  
  
*/\*\*  
 \* <p>  
 \* 布隆过滤器实现  
 \* <p>  
 \*  
 \** ***@author*** *chance  
 \** ***@since*** *2020-04-07  
 \*/*public class BloomFilter {  
  
 */\*\*  
 \* 位数组的大小  
 \*/* private static final int *DEFAULT\_SIZE* = 2 << 24;  
 */\*\*  
 \* 通过这个数组可以创建 6 个不同的哈希函数  
 \*/* private static final int[] *SEEDS* = new int[]{3, 13, 46, 71, 91, 134};  
  
 */\*\*  
 \* 位数组。数组中的元素只能是 0 或者 1  
 \*/* private BitSet bits = new BitSet(*DEFAULT\_SIZE*);  
  
 */\*\*  
 \* 存放包含 hash 函数的类的数组  
 \*/* private SimpleHash[] func = new SimpleHash[*SEEDS*.length];  
  
 */\*\*  
 \* 初始化多个包含 hash 函数的类的数组，每个类中的 hash 函数都不一样  
 \*/* public BloomFilter() {  
 // 初始化多个不同的 Hash 函数  
 for (int i = 0; i < *SEEDS*.length; i++) {  
 func[i] = new SimpleHash(*DEFAULT\_SIZE*, *SEEDS*[i]);  
 }  
 }  
  
 */\*\*  
 \* 添加元素到位数组  
 \*/* public void add(Object value) {  
 for (SimpleHash f : func) {  
 bits.set(f.hash(value), true);  
 }  
 }  
  
 */\*\*  
 \* 判断指定元素是否存在于位数组  
 \*/* public boolean contains(Object value) {  
 boolean ret = true;  
 for (SimpleHash f : func) {  
 ret = ret && bits.get(f.hash(value));  
 }  
 return ret;  
 }  
  
 */\*\*  
 \* 静态内部类。用于 hash 操作！  
 \*/* public static class SimpleHash {  
  
 private int cap;  
 private int seed;  
  
 public SimpleHash(int cap, int seed) {  
 this.cap = cap;  
 this.seed = seed;  
 }  
  
 */\*\*  
 \* 计算 hash 值  
 \*/* public int hash(Object value) {  
 int h;  
 return (value == null) ? 0 : Math.*abs*(seed \* (cap - 1) & ((h = value.hashCode()) ^ (h >>> 16)));  
 }  
  
 }  
}

#### 5、利用Google开源的Guava中自带的布隆过滤器

Guava中布隆过滤器的实现比较权威，所有实际项目中不需要手动实现布隆过滤器。

首先在项目中引入Guava的依赖：

<dependency>

<groupId>com.google.guava</groupId>

<artifactId>guava</artifactId>

<version>28.0-jre</version>

</dependency>

实际使用如下：创建一个最多1500个整数的布隆过滤器，并且我们可以容忍误判的概率为百分之（0.01）

//创建布隆过滤器对象

BloomFilter<Integer> filter = BloomFilter.*create*(  
 Funnels.*integerFunnel*(),  
 1500,  
 0.01  
);  
  
//判断指定元素是否存在  
System.*out*.println(filter.mightContain(1));  
System.*out*.println(filter.mightContain(2));  
  
//将元素添加进布隆过滤器  
filter.put(1);  
filter.put(2);  
System.*out*.println(filter.mightContain(1));  
System.*out*.println(filter.mightContain(2));

当mightContain()方法返回true时，我们可以99%确定该元素在过滤器中，当过滤器返回false时，我们可以100%确定该元素不存在于过滤器中。

Guava提供的布隆过滤器的实现还是很不错的，但是它有一个重大的缺陷就是只能单机使用（另外，容量扩展也不容易），而现在互联网一般都是分布式的场景。为了解决这个问题，我们就需要用到 Redis 中的布隆过滤器了。

#### 6、Redis中的布隆过滤器

（1）介绍

RedisBloom模块提供了四种数据类型，一个可伸缩的Bloom过滤器和Cuckoo过滤器，一个最小计数草图和一个前K个数据。Bloom和Cuckoo过滤器用于确定（以给定的确定性）集合中是否存在某项。虽然使用Count-Min Sketch估算子线性空间中的项目数，并且Top-K维护K个最频繁项目的列表。地址：<https://github.com/RedisBloom/RedisBloom>。

（2）使用Docker安装

docker run -p 6379:6379 --name redis-redisbloom redislabs/rebloom:latest

（3）将RedisBloom与redis-cli一起使用

docker exec -it redis-redisbloom bash

# redis-cli

# 127.0.0.1:6379>

通过添加新项目来启动新的Bloom过滤器

# 127.0.0.1:6379> **BF.ADD** newFilter foo

(integer) 1

检查过滤器中是否存在项目

# 127.0.0.1:6379> **BF.EXISTS** newFilter foo

(integer) 1

（4）构建和加载RedisBloom

为了使用此模块，请使用make进行构建并将其加载到Redis中。

加载：在加载模块的情况下调用redis

$ redis-server --loadmodule /path/to/redisbloom.so

（5）客户端类库

|  |  |  |
| --- | --- | --- |
| Project | Language | License |
| JReBloom | Java | BSD |

（6）文档

文档和完整的命令参考位于redisbloom.io。