什么是缓存击穿，如何解决缓存击穿

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：缓存、BF

题目描述

什么是缓存击穿，如何解决缓存击穿

题目解决

缓存击穿

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。

解决缓存击穿

解决方案一

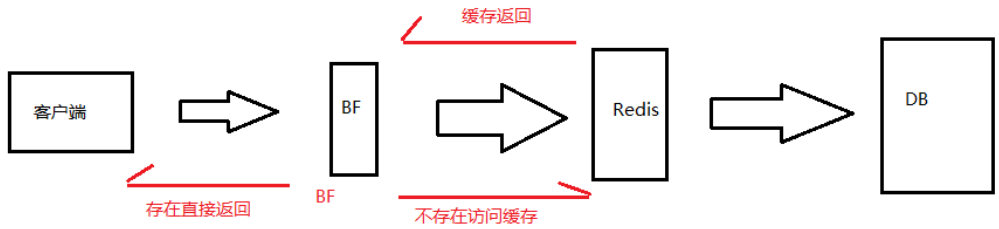
> 当数据库和redis中都不存在key，在数据库返回null时，在redis中插入<key,null,expireTime>当key再次请求时,redis直接返回null，而不用再次请求数据库。

解决方案二

可以设置一些过滤规则, 如布隆过滤器

将数据库中所有的查询条件，放入布隆过滤器中，

当一个查询请求过来时，先经过布隆过滤器进行查，如果判断请求查询值存在，则继续查；如果判断请求查询不存在，直接丢弃。



布隆过滤器的方式解决缓存穿透问题（重点）

简介

布隆过滤器（Bloom Filter，下文简称BF）由Burton Howard Bloom在1970年提出，是一种空间效率高的概率型数据结构。**它专门用来检测集合中是否存在特定的元素。**听起来是很稀松平常的需求，为什么要使用BF这种数据结构呢?

优缺点

优点：

- 不需要存储数据本身，只用比特表示，因此空间占用相对于传统方式有巨大的优势，并且能够保密数据；
- 时间效率也较高，插入和查询的时间复杂度均为O(k)；
- 哈希函数之间相互独立，可以在硬件指令层面并行计算。

缺点：

- 存在假阳性的概率，不适用于任何要求100%准确率的情境；
- 只能插入和查询元素，不能删除元素，这与产生假阳性的原因是相同的。我们可以简单地想到通过计数（即将一个比特扩展为计数值）来记录元素数，但仍然无法保证删除的元素一定在集合中。

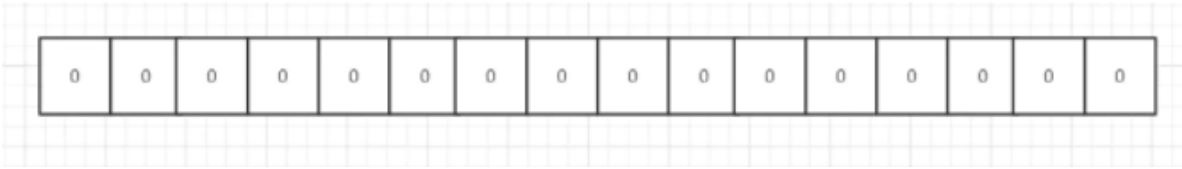
所以，BF在对查准度要求没有那么苛刻，而对时间、空间效率要求较高的场合非常合适，另外，由于它不存在假阴性问题，所以用作"不存在"逻辑的处理时有奇效，比如可以用来作为缓存系统（如Redis）的缓冲，防止缓存穿透。

设计思想

BF是由一个**长度为m比特的位数组（bit array）**与**k个哈希函数（hash function）**组成的数据结构。位数组均初始化为0，所有哈希函数都可以分别把输入数据尽量均匀地散列。

它本身是一个很长的二进制向量，既然是二进制的向量，那么显而易见的，存放的不是0，就是1。

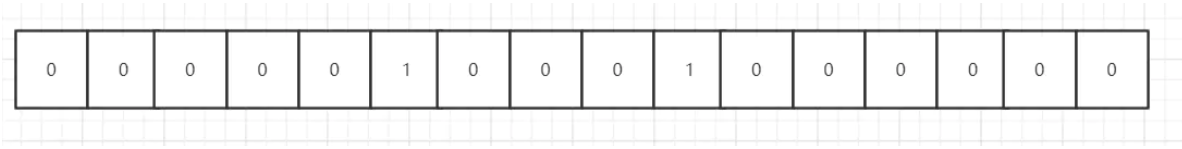
现在我们新建一个长度为16的布隆过滤器，默认值都是0，就像下面这样：



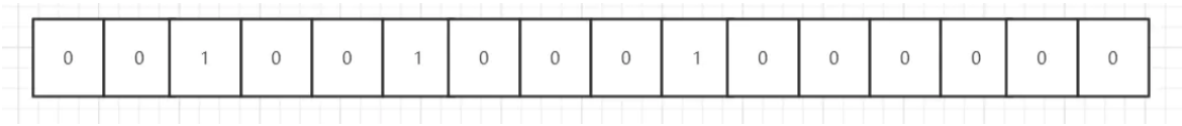
现在需要添加一个数据：

我们通过某种计算方式，比如Hash1，计算出了Hash1(数据)=5，我们就把下标为5的格子改成1，就像下面这样：

我们又通过某种计算方式，比如Hash2，计算出了Hash2(数据)=9，我们就把下标为9的格子改成1，就像下面这样：



还是通过某种计算方式，比如Hash3，计算出了Hash3(数据)=2，我们就把下标为2的格子改成1，就像下面这样：



这样，刚才添加的数据就占据了布隆过滤器“5”，“9”，“2”三个格子。

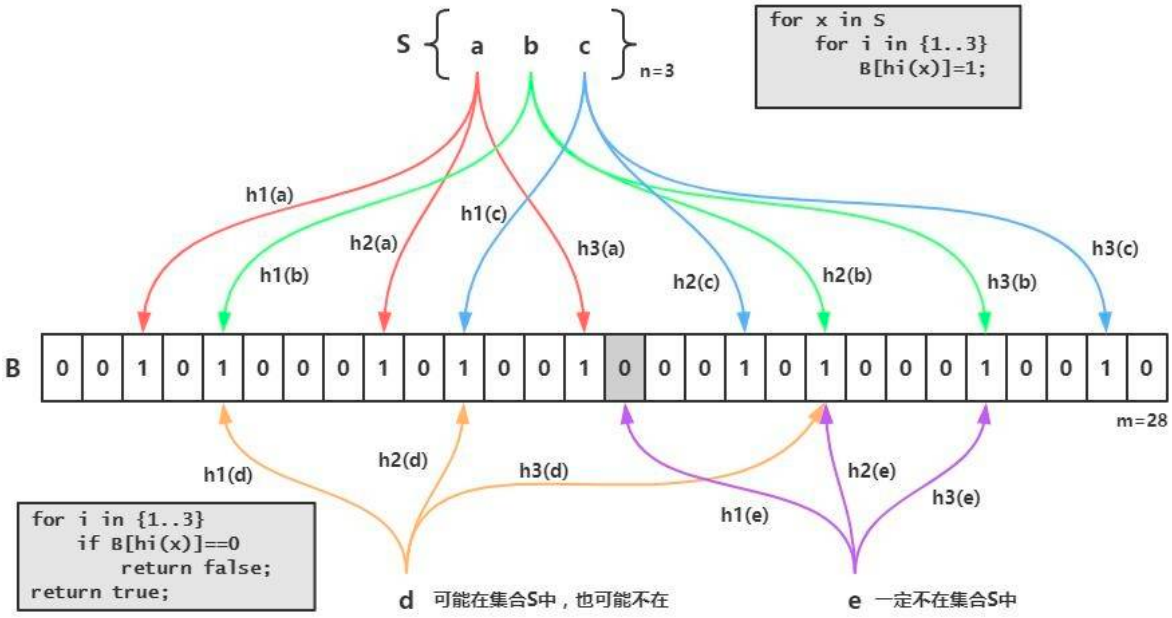
可以看出，仅仅从布隆过滤器本身而言，根本没有存放完整的数据，只是运用一系列随机映射函数计算出位置，然后填充二进制向量。

这有什么用呢？比如现在再给你一个数据，你要判断这个数据是否重复，你怎么做？

你只需利用上面的三种固定的计算方式，计算出这个数据占据哪些格子，然后看看这些格子里面放置的是否都是1，如果有一个格子不为1，那么就代表这个数字不在其中。这很好理解吧，比如现在又给你了刚才你添加进去的数据，你通过三种固定的计算方式，算出的结果肯定和上面的是一模一样的，也是占据了布隆过滤器“5”，“9”，“2”三个格子。

但是有一个问题需要注意，如果这些格子里面放置的都是1，不一定代表给定的数据一定重复，也许其他数据经过三种固定的计算方式算出来的结果也是相同的。这也很好理解吧，比如我们需要判断对象是否相等，是不可以仅仅判断他们的哈希值是否相等的。

也就是说布隆过滤器只能判断数据是否一定不存在，而无法判断数据是否一定存在。如图：



按理来说，介绍完了新增、查询的流程，就要介绍删除的流程了，但是很遗憾的是布隆过滤器是很难做到删除数据的，为什么？你想想，比如你要删除刚才给你的数据，你把“5”，“9”，“2”三个格子都改成了0，但是可能其他的数据也映射到了“5”，“9”，“2”三个格子啊，这不就乱套了吗？

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

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：$$m = -\frac{nlnfpp}{(ln2)^2}$$

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：$$k = \frac{m}{n}ln2$$

- 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。

看看Guava中BloomFilter中对于m和k值计算的实现，在com.google.common.hash.BloomFilter类中：

```
/**
 * 计算 Bloom Filter的bit位数m
 *
 * <p>See
http://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives for the
 * formula.
 *
 * @param n 预期数据量
 * @param p 误判率 (must be 0 < p < 1)
 */
@VisibleForTesting
static long optimalNumOfBits(long n, double p) {
  if (p == 0) {
    p = Double.MIN_VALUE;
  }
  return (long) (-n * Math.log(p) / (Math.log(2) * Math.log(2)));
}




/**
 * 计算最佳k值，即在Bloom过滤器中插入的每个元素的哈希数
 *
 * <p>See http://en.wikipedia.org/wiki/File:Bloom_filter_fp_probability.svg for
the formula.
 *
 * @param n 预期数据量
 * @param m bloom filter中总的bit位数 (must be positive)
 */
@VisibleForTesting
static int optimalNumOfHashFunctions(long n, long m) {
  // (m / n) * log(2), but avoid truncation due to division!
  return Math.max(1, (int) Math.round((double) m / n * Math.log(2)));
}
```

BloomFilter实现的另一个重点就是怎么利用hash函数把数据映射到bit数组中。Guava的实现是对元素通过MurmurHash3计算hash值，将得到的hash值取高8个字节以及低8个字节进行计算，以得当前元素在bit数组中对应的多个位置。MurmurHash3算法详见:Murmur哈希，于2008年被发明。这个算法hbase,redis,kafka都在使用。

这个过程的实现在两个地方：

- 将数据放入bloom filter中
- 判断数据是否已在bloom filter中

这两个地方的实现大同小异，区别只是，前者是put数据，后者是查数据。

这里看一下put的过程，hash策略以MURMUR128_MITZ_64为例：

```
public <T> boolean put(
    T object, Funnel<? super T> funnel, int numHashFunctions, LockFreeBitArray
bits) {
  long bitSize = bits.bitSize();

  //利用MurmurHash3得到数据的hash值对应的字节数组
  byte[] bytes = Hashing.murmur3_128().hashObject(object,
funnel).getBytesInternal();


  //取低8个字节、高8个字节，转成long类型
  long hash1 = lowerEight(bytes);
  long hash2 = upperEight(bytes);

  boolean bitsChanged = false;


  //这里的combinedHash = hash1 + i * hash2
  long combinedHash = hash1;


  //根据combinedHash，得到放入的元素在bit数组中的k个位置，将其置1
  for (int i = 0; i < numHashFunctions; i++) {
    bitsChanged |= bits.set((combinedHash & Long.MAX_VALUE) % bitSize);
    combinedHash += hash2;
  }
  return bitsChanged;
}
```

判断元素是否在bloom filter中的方法mightContain与上面的实现基本一致，不再赘述。

Bloom Filter的使用

简单写个demo，用法很简单

```
package com.qunar.sage.wang.common.bloom.filter;

import com.google.common.base.Charsets;
```

```
import com.google.common.hash.BloomFilter;
import com.google.common.hash.Funnel;
import com.google.common.hash.Funnels;
import com.google.common.hash.PrimitiveSink;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.ToString;

/**
 * BloomFilterTest
 *
 */
public class BloomFilterTest {

    public static void main(String[] args) {
        long expectedInsertions = 10000000;
        double fpp = 0.00001;

        BloomFilter<CharSequence> bloomFilter =
BloomFilter.create(Funnels.stringFunnel(Charsets.UTF_8), expectedInsertions,
fpp);

        bloomFilter.put("aaa");
        bloomFilter.put("bbb");
        boolean containsString = bloomFilter.mightContain("aaa");
        System.out.println(containsString);

        BloomFilter<Email> emailBloomFilter = BloomFilter
                .create((Funnel<Email>) (from, into) ->
into.putString(from.getDomain(), Charsets.UTF_8),
                        expectedInsertions, fpp);

        emailBloomFilter.put(new Email("sage.wang", "quanr.com"));
        boolean containsEmail = emailBloomFilter.mightContain(new
Email("sage.wangaaa", "quanr.com"));
        System.out.println(containsEmail);
    }

    @Data
    @Builder
    @ToString
    @AllArgsConstructor
    public static class Email {
        private String userName;
        private String domain;
    }

}
```

练习题

1.基于Spring boot 模拟并解决缓存穿透