布隆过滤器

2022年3月23日 13:44

布隆过滤器，这一篇给你讲的明明白白

2020-09-2450800

简介：浅谈布隆过滤器

什么是 BloomFilter

布隆过滤器（英语：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。主要用于判断一个元素是否在一个集合中。

通常我们会遇到很多要判断一个元素是否在某个集合中的业务场景，一般想到的是将集合中所有元素保存起来，然后通过比较确定。链表、树、散列表（又叫哈希表，Hash table）等等数据结构都是这种思路。但是随着集合中元素的增加，我们需要的存储空间也会呈现线性增长，最终达到瓶颈。同时检索速度也越来越慢，上述三种结构的检索时间复杂度分别为 $O(n)$，$O(logn)$，$O(1)$。

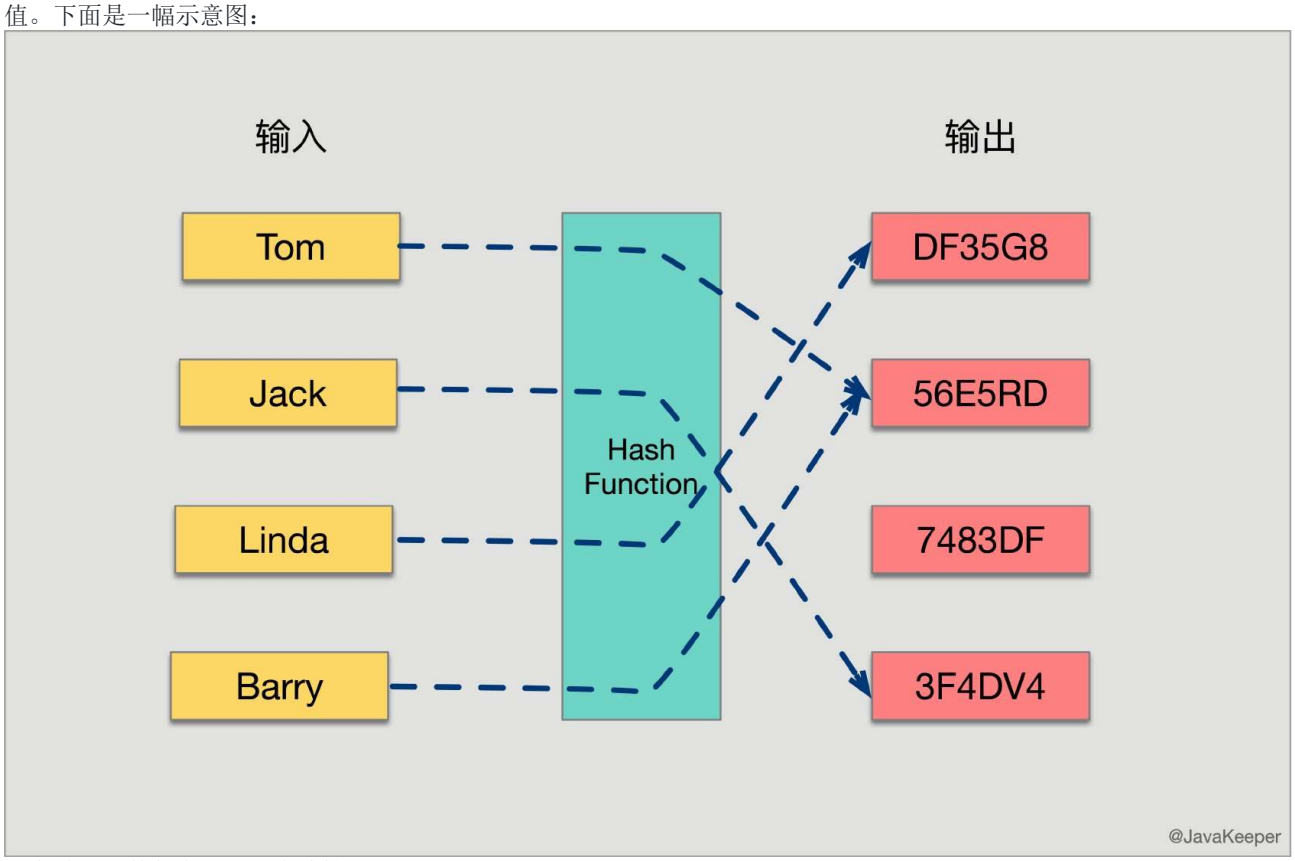
这个时候，布隆过滤器（Bloom Filter）就应运而生。

布隆过滤器原理

了解布隆过滤器原理之前，先回顾下 Hash 函数原理。

哈希函数

哈希函数的概念是：将任意大小的输入数据转换成特定大小的输出数据的函数，转换后的数据称为哈希值或哈希编码，也叫散列值。下面是一幅示意图：



所有散列函数都有如下基本特性：

- 如果两个散列值是不相同的（根据同一函数），那么这两个散列值的原始输入也是不相同的。这个特性是散列函数具有确定性的结果，具有这种性质的散列函数称为单向散列函数。
- 散列函数的输入和输出不是唯一对应关系的，如果两个散列值相同，两个输入值很可能是相同的，但也可能不同，这种情况称为“散列碰撞（collision）”。

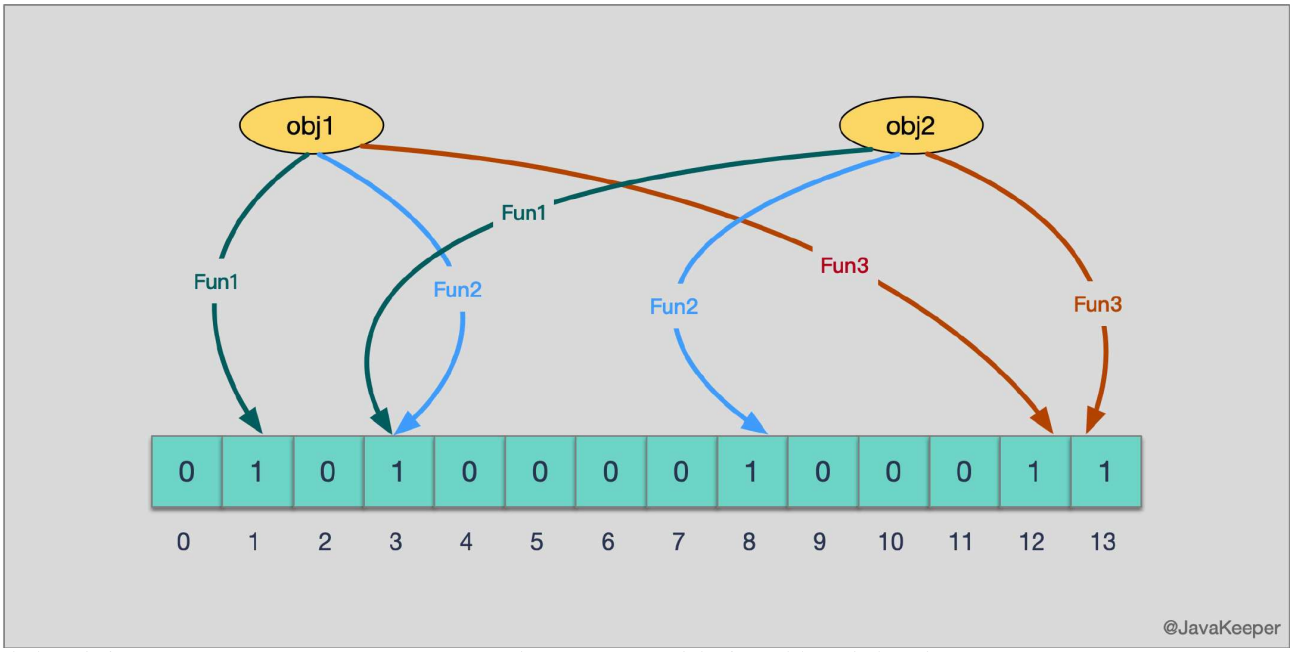
但是用 hash表存储大数据量时，空间效率还是很低，当只有一个 hash 函数时，还很容易发生哈希碰撞。

布隆过滤器数据结构

BloomFilter 是由一个固定大小的二进制向量或者位图（bitmap）和一系列映射函数组成的。

在初始状态时，对于长度为 m 的位数组，它的所有位都被置为0，如下图所示：

当有变量被加入集合时，通过 K 个映射函数将这个变量映射成位图中的 K 个点，把它们置为 1（假定有两个变量都通过 3 个映射函数）。



查询某个变量的时候我们只要看看这些点是不是都是 1 就可以大概率知道集合中有没有它了

- 如果这些点有任何一个 0，则被查询变量一定不在；
- 如果都是 1，则被查询变量很可能存在

为什么说是可能存在，而不是一定存在呢？那是因为映射函数本身就是散列函数，散列函数是会有碰撞的。

误判率

布隆过滤器的误判是指多个输入经过哈希之后在相同的bit位置1了，这样就无法判断究竟是哪个输入产生的，因此误判的根源在于相同的 bit 位被多次映射且置 1。

这种情况也造成了布隆过滤器的删除问题，因为布隆过滤器的每一个 bit 并不是独占的，很有可能多个元素共享了某一位。如果我们直接删除这一位的话，会影响其他的元素。(比如上图中的第 3 位)

特性

- 一个元素如果判断结果为存在的时候元素不一定存在，但是判断结果为不存在的时候则一定不存在。
- 布隆过滤器可以添加元素，但是不能删除元素。因为删掉元素会导致误判率增加。

添加与查询元素步骤

添加元素

1. 将要添加的元素给 k 个哈希函数
2. 得到对应于位数组上的 k 个位置
3. 将这k个位置设为 1

查询元素

1. 将要查询的元素给k个哈希函数
2. 得到对应于位数组上的k个位置

3. 如果k个位置有一个为 0，则肯定不在集合中
4. 如果k个位置全部为 1，则可能在集合中

优点

相比于其它的数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数 $O(K)$，另外，散列函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。
布隆过滤器可以表示全集，其它任何数据结构都不能；

缺点

但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。
另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位数组变成整数数组，每插入一个元素相应的计数器加 1，这样删除元素时将计数器减掉就可以了。然而要保证安全地删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面。这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。
在降低误算率方面，有不少工作，使得出现了很多布隆过滤器的变种。

布隆过滤器使用场景和实例

在程序的世界中，布隆过滤器是程序员的一把利器，利用它可以快速地解决项目中一些比较棘手的问题。
如网页 URL 去重、垃圾邮件识别、大集合中重复元素的判断和缓存穿透等问题。
布隆过滤器的典型应用有：

- 数据库防止穿库。 Google Bigtable，HBase 和 Cassandra 以及 Postgresql 使用BloomFilter来减少不存在的行或列的磁盘查找。避免代价高昂的磁盘查找会大大提高数据库查询操作的性能。
- 业务场景中判断用户是否阅读过某视频或文章，比如抖音或头条，当然会导致一定的误判，但不会让用户看到重复的内容。
- 缓存宕机、缓存击穿场景，一般判断用户是否在缓存中，如果在则直接返回结果，不在则查询db，如果来一波冷数据，会导致缓存大量击穿，造成雪崩效应，这时候可以用布隆过滤器当缓存的索引，只有在布隆过滤器中，才去查询缓存，如果没查询到，则穿透到db。如果不在布隆器中，则直接返回。
- WEB拦截器，如果相同请求则拦截，防止重复被攻击。用户第一次请求，将请求参数放入布隆过滤器中，当第二次请求时，先判断请求参数是否被布隆过滤器命中。可以提高缓存命中率。Squid 网页代理缓存服务器在 cache digests 中就使用了布隆过滤器。Google Chrome浏览器使用了布隆过滤器加速安全浏览服务
- Venti 文档存储系统也采用布隆过滤器来检测先前存储的数据。
- SPIN 模型检测器也使用布隆过滤器在大规模验证问题时跟踪可达状态空间。

Coding~

知道了布隆过滤去的原理和使用场景，我们可以自己实现一个简单的布隆过滤器

自定义的 BloomFilter

```
public class MyBloomFilter {
/**
     * 一个长度为10 亿的比特位
     */
    private static final int DEFAULT_SIZE = 256 << 22;
/**
     * 为了降低错误率，使用加法hash算法，所以定义一个8个元素的质数数组
     */
    private static final int[] seeds = {3, 5, 7, 11, 13, 31, 37, 61};
/**
     * 相当于构建 8 个不同的hash算法
     */
    private static HashFunction[] functions = new HashFunction[seeds.length];
/**
     * 初始化布隆过滤器的 bitmap
     */
    private static BitSet bitset = new BitSet(DEFAULT_SIZE);
/**
     * 添加数据
     *
     * @param value 需要加入的值
     */
    public static void add(String value) {
        if (value != null) {
            for (HashFunction f : functions) {
                //计算 hash 值并修改 bitmap 中相应位置为 true
                bitset.set(f.hash(value), true);
            }
        }
    }
```

```
/**
     * 判断相应元素是否存在
     * @param value 需要判断的元素
     * @return 结果
     */
    public static boolean contains(String value) {
        if (value == null) {
            return false;
        }
        boolean ret = true;
        for (HashFunction f : functions) {
            ret = bitset.get(f.hash(value));
            //一个 hash 函数返回 false 则跳出循环
            if (!ret) {
                break;
            }
        }
        return ret;
    }
/**
     * 模拟用户是不是会员，或用户在不在线。。。
     */
    public static void main(String[] args) {
for (int i = 0; i < seeds.length; i++) {
            functions[i] = new HashFunction(DEFAULT_SIZE, seeds[i]);
        }
// 添加1亿数据
        for (int i = 0; i < 100000000; i++) {
            add(String.valueOf(i));
        }
        String id = "123456789";
        add(id);
System.out.println(contains(id));   // true
        System.out.println("" + contains("234567890"));  //false
    }
}
class HashFunction {
private int size;
    private int seed;
public HashFunction(int size, int seed) {
        this.size = size;
        this.seed = seed;
    }
public int hash(String value) {
        int result = 0;
        int len = value.length();
        for (int i = 0; i < len; i++) {
            result = seed * result + value.charAt(i);
        }
        int r = (size - 1) & result;
        return (size - 1) & result;
    }
}
```

What？我们写的这些早有大牛帮我们实现，还造轮子，真是浪费时间，No，No，No，我们学习过程中是可以造轮子的，造轮子本身就是我们自己对设计和实现的具体落地过程，不仅能提高我们的编程能力，在造轮子的过程中肯定会遇到很多我们没有思考过的问题，成长看的见~~

实际项目使用的时候，领导和我说项目一定要稳定运行，没自信的我放弃了自己的轮子。

Guava 中的 BloomFilter

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>23.0</version>
</dependency>
public class GuavaBloomFilterDemo {
public static void main(String[] args) {
        //后边两个参数：预计包含的数据量，和允许的误差值
```

```
        BloomFilter<Integer> bloomFilter = BloomFilter.create(Funnels.integerFunnel(), 100000, 0.01);
        for (int i = 0; i < 100000; i++) {
            bloomFilter.put(i);
        }
        System.out.println(bloomFilter.mightContain(1));
        System.out.println(bloomFilter.mightContain(2));
        System.out.println(bloomFilter.mightContain(3));
        System.out.println(bloomFilter.mightContain(100001));
//bloomFilter.writeTo();
    }
}
```

分布式环境中，布隆过滤器肯定还需要考虑是可以共享的资源，这时候我们会想到 Redis，是的，Redis 也实现了布隆过滤器。当然我们也可以把布隆过滤器通过 bloomFilter.writeTo()写入一个文件，放入OSS、S3这类对象存储中。

Redis 中的 BloomFilter

Redis 提供的 bitMap 可以实现布隆过滤器，但是需要自己设计映射函数和一些细节，这和我们自定义没啥区别。

Redis 官方提供的布隆过滤器到了 Redis 4.0 提供了插件功能之后才正式登场。布隆过滤器作为一个插件加载到 Redis Server 中，给 Redis 提供了强大的布隆去重功能。

在已安装 Redis 的前提下，安装 RedisBloom，有两种方式

直接编译进行安装

```
git clone https://github.com/RedisBloom/RedisBloom.git
cd RedisBloom
make     #编译 会生成一个rebloom.so文件
redis-server --loadmodule /path/to/rebloom.so   #运行redis时加载布隆过滤器模块
redis-cli    # 启动连接容器中的 redis 客户端验证
```

使用Docker进行安装

```
docker pull redislabs/rebloom:latest # 拉取镜像
docker run -p 6379:6379 --name redis-redisbloom redislabs/rebloom:latest #运行容器
docker exec -it redis-redisbloom bash
redis-cli
```

使用

布隆过滤器基本指令：

- bf.add 添加元素到布隆过滤器
- bf.exists 判断元素是否在布隆过滤器
- bf.madd 添加多个元素到布隆过滤器，bf.add 只能添加一个
- bf.mexists 判断多个元素是否在布隆过滤器

```
127.0.0.1:6379> bf.add user Tom
(integer) 1
127.0.0.1:6379> bf.add user John
(integer) 1
127.0.0.1:6379> bf.exists user Tom
(integer) 1
127.0.0.1:6379> bf.exists user Linda
(integer) 0
127.0.0.1:6379> bf.madd user Barry Jerry Mars
1) (integer) 1
2) (integer) 1
3) (integer) 1
127.0.0.1:6379> bf.mexists user Barry Linda
1) (integer) 1
2) (integer) 0
```

我们只有这几个参数，肯定不会有误判，当元素逐渐增多时，就会有一定的误判了，这里就不做这个实验了。

上面使用的布隆过滤器只是默认参数的布隆过滤器，它在我们第一次 add 的时候自动创建。

Redis 还提供了自定义参数的布隆过滤器，bf.reserve 过滤器名 error_rate initial_size

- error_rate：允许布隆过滤器的错误率，这个值越低过滤器的位数组的大小越大，占用空间也就越大
- initial_size：布隆过滤器可以储存的元素个数，当实际存储的元素个数超过这个值之后，过滤器的准确率会下降

但是这个操作需要在 add 之前显式创建。如果对应的 key 已经存在，bf.reserve 会报错

```
127.0.0.1:6379> bf.reserve user 0.01 100
(error) ERR item exists
127.0.0.1:6379> bf.reserve topic 0.01 1000
OK
```

我是一名 Javaer，肯定还要用 Java 来实现的，Java 的 Redis 客户端比较多，有些还没有提供指令扩展机制，笔者已知的 Redisson 和 lettuce 是可以使用布隆过滤器的，我们这里用 Redisson

```
public class RedissonBloomFilterDemo {
public static void main(String[] args) {
Config config = new Config();
```

```
        config.useSingleServer().setAddress("redis://127.0.0.1:6379");
        RedissonClient redisson = Redisson.create(config);
RBloomFilter<String> bloomFilter = redisson.getBloomFilter("user");
        // 初始化布隆过滤器，预计统计元素数量为55000000，期望误差率为0.03
        bloomFilter.tryInit(55000000L, 0.03);
        bloomFilter.add("Tom");
        bloomFilter.add("Jack");
        System.out.println(bloomFilter.count());   //2
        System.out.println(bloomFilter.contains("Tom"));  //true
        System.out.println(bloomFilter.contains("Linda"));  //false
    }
}
```

扩展

为了解决布隆过滤器不能删除元素的问题，布谷鸟过滤器横空出世。论文《Cuckoo Filter：Better Than Bloom》作者将布谷鸟过滤器和布隆过滤器进行了深入的对比。相比布谷鸟过滤器而言布隆过滤器有以下不足：查询性能弱、空间利用效率低、不支持反向操作（删除）以及不支持计数。

由于使用较少，暂不深入。

参考与感谢

https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf
http://www.justdojava.com/2019/10/22/bloomfilter/
https://www.cnblogs.com/cpselvis/p/6265825.html
https://juejin.im/post/5cc5aa7ce51d456e431adac5