

[首页](#) [资讯](#) [精华](#) [论坛](#) [问答](#) [博客](#) [专栏](#) [群组](#) [更多](#) ▼
[您还未登录！](#) [登录](#) [注册](#)

追随大师的脚步,,,

- [博客](#)
- [微博](#)
- [相册](#)
- [收藏](#)
- [留言](#)
- [关于我](#)



BloomFilter - 大规模数据处理利器

博客分类:

- [java](#)

BloomFilter-大规模数据处理利器

Bloom Filter是由Bloom在1970年提出的一种多哈希函数映射的快速查找算法。通常应用在一些需要快速判断某个元素是否属于集合，但是并不严格要求100%正确的场合。

一. 实例

为了说明Bloom Filter存在的重要意义，举一个实例：

假设要你写一个网络爬虫程序（web crawler）。由于网络间的链接错综复杂，爬虫在网络间爬行很可能会形成“环”。为了避免形成“环”，就需要知道爬虫程序已经访问过那些URL。给一个URL，怎样知道爬虫程序是否已经访问过呢？稍微想想，就会有如下几种方案：

1. 将访问过的URL保存到数据库。
2. 用HashSet将访问过的URL保存起来。那只需接近O(1)的代价就可以查到一个URL是否被访问过了。
3. URL经过MD5或SHA-1等单向哈希后再保存到HashSet或数据库。
4. Bit-Map方法。建立一个BitSet，将每个URL经过一个哈希函数映射到某一位。

方法1~3都是将访问过的URL完整保存，方法4则只标记URL的一个映射位。

以上方法在数据量较小的情况下都能完美解决问题，但是当数据量变得非常庞大时问题就来了。

方法1的缺点：数据量变得非常庞大后关系型数据库查询的效率会变得很低。而且每来一个URL就启动一次数据库查询是不是太小题大做了？

方法2的缺点：太消耗内存。随着URL的增多，占用的内存会越来越多。就算只有1亿个URL，每个URL只算50个字符，就需要5GB内存。

方法3：由于字符串经过MD5处理后的信息摘要长度只有128Bit，SHA-1处理后也只有160Bit，因此方法3比方法2节省了好几倍的内存。

方法4消耗内存是相对较少的，但缺点是单一哈希函数发生冲突的概率太高。还记得数据结构课上学过的Hash表冲突的各种解决方法么？若要降低冲突发生的概率到1%，就要将BitSet的长度设置为URL个数的100倍。

二. Bloom Filter的算法

废话说到这里，下面引入本篇的主角-Bloom Filter。其实上面方法4的思想已经很接近Bloom Filter了。方法四的致命缺点是冲突概率高，为了降低冲突的概念，Bloom Filter使用了多个哈希函数，而不是一个。

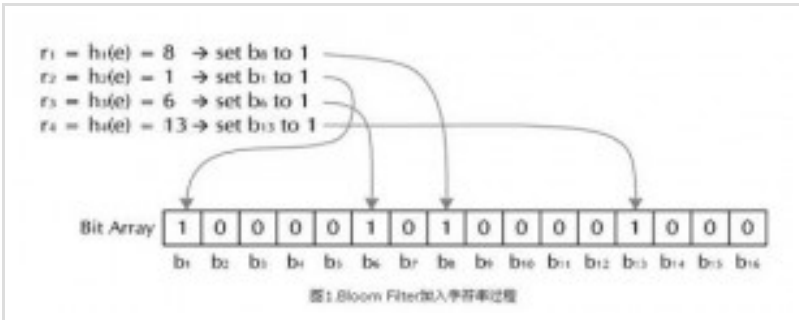
Bloom Filter算法如下：

创建一个m位BitSet，先将所有位初始化为0，然后选择k个不同的哈希函数。第i个哈希函数对字符串str哈希的结果记为h (i, str)，且h (i, str) 的范围是0到m-1。

(1) 加入字符串过程

下面是每个字符串处理的过程，首先是将字符串str“记录”到BitSet中的过程：

对于字符串str，分别计算h (1, str)，h (2, str) h (k, str)。然后将BitSet的第h (1, str)、h (2, str) h (k, str) 位设为1。



很简单吧？这样就将字符串str映射到BitSet中的k个二进制位了。

(2) 检查字符串是否存在过程

下面是检查字符串str是否被BitSet记录过的过程：

对于字符串str，分别计算h (1, str)，h (2, str) h (k, str)。然后检查BitSet的第h (1, str)、h (2, str) h (k, str) 位是否为1，若其中任何一位不为1则可以判定str一定没有被记录过。若全部位都是1，则“认为”字符串str存在。

若一个字符串对应的Bit不全为1，则可以肯定该字符串一定没有被Bloom Filter记录过。（这是显然的，因为字符串被记录过，其对应的二进制位肯定全部被设为1了）

但是若一个字符串对应的Bit全为1，实际上是不能100%的肯定该字符串被Bloom Filter记录过的。（因为有可能该字符串的所有位都刚好是被其他字符串所对应）这种将该字符串划分错的情况，称为false positive。

(3) 删除字符串过程

字符串加入了就被不能删除了，因为删除会影响到其他字符串。实在需要删除字符串的可以使用Counting bloomfilter(CBF)，这是一种基本Bloom Filter的变体，CBF将基本Bloom Filter每一个Bit改为一个计数器，这样就可以实现删

除字符串的功能了。

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。

三. Bloom Filter参数选择

(1) 哈希函数选择

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。

(2) m,n,k值，我们如何取值

我们定义：

可能把不属于这个集合的元素误认为属于这个集合（False Positive）

不会把属于这个集合的元素误认为不属于这个集合（False Negative）。

哈希函数的个数k、位数组大小m、加入的字符串数量n的关系。哈希函数个数k取10，位数组大小m设为字符串个数n的20倍时，false positive发生的概率是0.0000889，即10万次的判断中，会存在9次误判，对于一天1亿次的查询，误判的次数为9000次。

算法分析：

我们假设 $kn < m$ 且各个哈希函数是完全随机的。当集合 $S=\{x_1, x_2, \dots, x_n\}$ 的所有元素都被k个哈希函数映射到m位的位数组中时，这个位数组中某一位还是0的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

False Positive的概率是：

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

p' 表示1的概率,k次方表示k次hash都为1的概率。

当 $k = \ln 2 * m/n$ 时，右边的等式值最小，此时等式转变成：

$$\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}.$$

四. Bloom Filter实现代码（简易版）

下面给出一个简单的Bloom Filter的Java实现代码：

```
package org.magnus.utils;
import java.util.BitSet;
//传统的Bloom filter 不支持从集合中删除成员。
//Counting Bloom filter由于采用了计数，因此支持remove操作。
//基于BitSet来实现，性能上可能存在问题
public class SimpleBloomFilter {
    //DEFAULT_SIZE为2的25次方
    private static final int DEFAULT_SIZE = 2 << 24;
    /* 不同哈希函数的种子，一般应取质数，seeds数据共有7个值，则代表采用7种不同的HASH算法 */
    private static final int[] seeds = new int[] { 5, 7, 11, 13, 31, 37, 61 };
    //BitSet实际是由“二进制位”构成的一个Vector。假如希望高效率地保存大量“开-关”信息，就应使用BitSet。
    //BitSet的最小长度是一个长整数（Long）的长度：64位
    private BitSet bits = new BitSet(DEFAULT_SIZE);
    /* 哈希函数对象 */
    private SimpleHash[] func = new SimpleHash[seeds.length];

    public static void main(String[] args) {
        String value = "stone2083@yahoo.cn";
        //定义一个filter，定义的时候会调用构造函数，即初始化七个hash函数对象所需要的信息。
        SimpleBloomFilter filter = new SimpleBloomFilter();
        //判断是否包含在里面。因为没有调用add方法，所以肯定是返回false
        System.out.println(filter.contains(value));
        filter.add(value);
        System.out.println(filter.contains(value));
    }
    //构造函数
    public SimpleBloomFilter() {
        for (int i = 0; i < seeds.length; i++) {
            //给出所有的hash值，共计seeds.length个hash值。共7位。
            //通过调用SimpleHash.hash(),可以得到根据7种hash函数计算得出的hash值。
            //传入DEFAULT_SIZE(最终字符串的长度)，seeds[i](一个指定的质数)即可得到需要的那个hash值的位置。
            func[i] = new SimpleHash(DEFAULT_SIZE, seeds[i]);
        }
    }

    // 将字符串标记到bits中，即设置字符串的7个hash值函数为1
    public void add(String value) {
        for (SimpleHash f : func) {
            bits.set(f.hash(value), true);
        }
    }

    //判断字符串是否已经被bits标记
    public boolean contains(String value) {
        //确保传入的不是空值
        if (value == null) {
            return false;
        }
        boolean ret = true;
        //计算7种hash算法下各自对应的hash值，并判断
        for (SimpleHash f : func) {
            //&&是boolean运算符，只要有一个为0，则为0。即需要所有的位都为1，才代表包含在里面。
            //f.hash(value)返回hash对应的位数
            //bits.get函数返回bitset中对应position的值。即返回hash值是否为0或1。
        }
    }
}
```

```

        ret = ret && bits.get(f.hash(value));
    }
    return ret;
}
/* 哈希函数类 */
public static class SimpleHash {
    //cap为DEFAULT_SIZE的值, 即用于结果的最大的字符串长度。
    //seed为计算hash值的一个给定key, 具体对应上面定义的seeds数组
    private int cap;
    private int seed;

    public SimpleHash(int cap, int seed) {
        this.cap = cap;
        this.seed = seed;
    }

    //计算hash值的具体算法, hash函数, 采用简单的加权和hash
    public int hash(String value) {
        //int的范围最大是2的31次方减1, 或超过值则用负数来表示
        int result = 0;
        int len = value.length();
        for (int i = 0; i < len; i++) {
            //数字和字符串相加, 字符串转换成为ASCII码
            result = seed * result + value.charAt(i);
            //System.out.println(result+"--"+seed+"*"+result+"-"+value.charAt(i));
        }
        // System.out.println("result="+result+";"+((cap - 1) & result));
        // System.out.println(414356308*61+'h'); 执行此运算结果为负数, 为什么?
        //&是java中的位逻辑运算, 用于过滤负数(负数与进算转换成反码进行)。
        return (cap - 1) & result;
    }
}
}

```

五: Bloom Filter的优点及应用。

1.2 优缺点分析

1.2.1 优点:

节约缓存空间(空值的映射), 不再需要空值映射。

减少数据库或缓存的请求次数。

提升业务的处理效率以及业务隔离性。

1.2.2 缺点:

存在误判的概率。

传统的Bloom Filter不能作删除操作。

1.3 使用场景

适用于特定场景, 能够有效的解决数据库空查问题。

以公司的某小表查询为例，该表每天查询量20亿次左右，且数据库中存在大量的下面的空查：

目前表中的记录为8w,即n的值为8w, $m=20*n=160w$ ，占用空间大小195KB。以type||CONTENT复合键作为key值，假设HASH次数k取值为6,误判率为:0.0303%(10000次中存在3次误判)。HASH次数的最优解为14，当k=14时，误判率为：0.014%(10000次中存在1-2次误判)。

测试过程及结果如下（源代码见附件）：

测试场景1： m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:6;对1000w数据进行判定：



```
<terminated> ZhoucangBloomTest [Java Application] D:\Program Files\myeclipse
m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:6
数据装载开始
现在时间:2011年03月16日星期三 20时23分04秒
Wed Mar 16 20:23:04 CST 2011: 成功注册JDBC驱动程序\oracle.jdbc
Wed Mar 16 20:23:04 CST 2011: 成功创建连接池oracle1
Wed Mar 16 20:23:04 CST 2011: 连接池oracle1创建一个新的连接
数据装载结束
现在时间:2011年03月16日星期三 20时23分12秒
判断开始, 判断的数据量为:10000000
现在时间:2011年03月16日星期三 20时23分12秒
判断结束
现在时间:2011年03月16日星期三 20时23分37秒
Positive=3035;count=10000000;Positive%=0.03035
```

测试结果：

2000w数据误判的记录为：3035，误判率约为0.03035%（和理论值0.0303%相差不大）。判断2000万数据的时间为25秒。平均一次判断时间为:2.5微秒。平均一次hash时间为0.417微秒。

测试场景2： m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:6;对2000w数据进行判定：



```
<terminated> ZhoucangBloomTest [Java Application] D:\Program Files\myeclipse
m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:6
数据装载开始
现在时间:2011年03月16日星期三 20时19分57秒
Wed Mar 16 20:19:57 CST 2011: 成功注册JDBC驱动程序\oracle.jdbc
Wed Mar 16 20:19:57 CST 2011: 成功创建连接池oracle1
Wed Mar 16 20:19:57 CST 2011: 连接池oracle1创建一个新的连接
数据装载结束
现在时间:2011年03月16日星期三 20时20分05秒
判断开始, 判断的数据量为:20000000
现在时间:2011年03月16日星期三 20时20分05秒
判断结束
现在时间:2011年03月16日星期三 20时20分56秒
Positive=5839;count=20000000;Positive%=0.029195
```

测试结果：2000w数据误判的记录为：5839，误判率约为0.029%（理论值为0.0303%）。判断1000万数据的时间为51秒。平均一次判断时间为:2.55微秒。平均一次hash时间为0.425微秒。

测试场景3: m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:14;对1000w数据进行判定:

```
Console X
<terminated> ZhoucangBloomTest [Java Application] D:\Program Files\myeclipse
m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:14
数据装载开始
现在时间:2011年03月16日星期三 20时30分07秒
Wed Mar 16 20:30:07 CST 2011: 成功注册JDBC驱动程序\oracle.jdbc
Wed Mar 16 20:30:07 CST 2011: 成功创建连接池oracle1
Wed Mar 16 20:30:07 CST 2011: 连接池oracle1创建一个新的连接
数据装载结束
现在时间:2011年03月16日星期三 20时30分17秒
判断开始,判断的数据量为:10000000
现在时间:2011年03月16日星期三 20时30分17秒
判断结束
现在时间:2011年03月16日星期三 20时30分54秒
Positive=605;count=10000000;Positive%=0.00605
```

测试结果: 1000w数据误判的记录为: 605, 误判率约为0.00605% (和理论值0.014%相差不大)。判断1000万数据的时间为37秒。平均一次判断时间为:3.7微秒。平均一次hash时间为0.265微秒。

测试场景4: m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:14;对2000w数据进行判定:

```
Console X
<terminated> ZhoucangBloomTest [Java Application] D:\Program Files\myeclipse
m=1600000;n=80000;最优解k=14;m/n=20;k的次数为:14
数据装载开始
现在时间:2011年03月16日星期三 20时35分28秒
Wed Mar 16 20:35:28 CST 2011: 成功注册JDBC驱动程序\oracle.jdbc
Wed Mar 16 20:35:28 CST 2011: 成功创建连接池oracle1
Wed Mar 16 20:35:28 CST 2011: 连接池oracle1创建一个新的连接
数据装载结束
现在时间:2011年03月16日星期三 20时35分37秒
判断开始,判断的数据量为:20000000
现在时间:2011年03月16日星期三 20时35分37秒
判断结束
现在时间:2011年03月16日星期三 20时36分51秒
Positive=1224;count=20000000;Positive%=0.00612
```

测试结果: 2000w数据误判的记录为: 1224, 误判率约为0.00612% (理论值为0.014%)。判断1000万数据的时间为84秒。平均一次判断时间为:4.2微秒。平均一次hash时间为0.3微秒。

其它测试略。

结论:

测	m/n	K (括	数据基	误判数	误判率	理论值	用时	一次	一次Hash
---	-----	------	-----	-----	-----	-----	----	----	--------

试		号内为 最优 解)	数				(单 位: 秒)	判定 时间 (单 位: 微秒)	时间(单 位: 微 秒.估参 考)
1	20	6(14)	1000W	3035	0.03035%	0.0303%	25	2.5	0.417
2	20	6(14)	2000W	5839	0.029%	0.0303%	51	2.55	0.425
3	20	14(14)	1000W	605	0.00605%	0.014%	37	3.7	0.265
4	20	14(14)	2000W	1224	0.00612%	0.014%	84	4.2	0.3
5	20	20(14)	1000W	914	0.00914%	不计算	48	4.8	0.24
6	20	20(14)	2000W	1881	0.00941%	不计算	99	4.95	0.2475
7	10	7 (7)	1000w	517854	0.786%	0.819%	41	4.1	0.59
8	5	3 (3)	1000w	901411	9.014%	9.2%	31	3.1	1.033
9	2	1(1)	1000w	3910726	39.107%	39.3%	29	2.9	2.9
10	2	2(1)	1000w	3961065	39.61%	40%	30	3.0	3.0
11	2	5(1)	1000w	6436696	64.37%	不计算	76	7.6	1.52

一次判断时间计算方式为：总时间/总次数

一次HASH所需时间计算方式为：一次判定时间/每次判断需要的hash数。

一次HASH所需时间，当执行hash次数越少，基数越小，误差越大。当一次判断所需的hash次数越大时，一次hash时间越精确。

结论：

m/n的比值越大越好，比较越大，误判率会越代，但同时会使用更多的空间成本。

Hash次数增加带来的收益并不大。需要在条件允许的情况下，尽可能的扩大m/n的值。



六：实施方案思考

适用于一些黑名单,垃圾邮件等的过滤。

当位数组较小时，可以作本地jvm缓存。

当位数组较大时，可以做基于tair的缓存，此时可能需要开辟单独的应用来提供查询支持。

此方案，适用的应用场景需要能够容忍，位数组和的延时。

分享到： 

[特定网站爬虫---原理篇](#) | [bloom filter算法 的Java 实现](#)

- 2012-11-07 11:29
- 浏览 2370
- [评论\(0\)](#)
- 分类:[编程语言](#)