如何实现网页爬虫中的URL去重功能?

问题:

网页爬虫是搜索引擎中的非常重要的系统，负责爬取几十亿、上百亿的网页。爬虫的工作原理是，通过解析已经爬取页面中的网页链接，然后再爬取这些链接对应的网页。而**同一个网页链接有可能被包含在多个页面中，这就会导致爬虫在爬取的过程中，重复爬取相同的网页。如果你是一名负责爬虫的工程师，你会如何避免这些重复的爬取呢?**

思路:

记录已经爬取的网页链接（也就是 URL），在爬取一个新的网页之前，拿它的链接，在已经爬取的网页链接列表中搜索。如果存在，那就说明这个网页已经被爬取过了；如果不存在，那就说明这个网页还没有被爬取过，可以继续去爬取。等爬取到这个网页之后，我们将这个网页的链接添加到已经爬取的网页链接列表了。

算法解析:

1. 面对上面的问题，我们是否可以数组、链表、树或图等数据结构解决吗?
2. 处理对象：网页链接，也就是 URL
3. 算法支持两个操作：添加一个 URL 和查询一个 URL
4. 要求：除了这两个功能性的要求之外，在非功能性方面，还要求这两个操作的执行效率要尽可能高。除此之外，因为处理的是上亿的网页链接，内存消耗会非常大，所以在存储效率上，要尽可能地高效。
5. 内存计算

   ```
   10亿个网页URL
   每个URL平均长度64字节， 保存10亿个URL，需要60GB内存
   ```

6. 散列表、红黑树、跳表这些动态数据结构，都能支持快速地插入、查找数据，但是对内存消耗方面，是否可以接受呢?
   - 散列表必须维持较小的装载因子，才能保证不会出现过多的散列冲突，导致操作的性能下降
   - 用链表法解决冲突的散列表，还会存储链表指针
   - 如果将这 10 亿个 URL 构建成散列表，那需要的内存空间会远大于 60GB，有可能会超过 100GB
     - 对于一个大型的搜索引擎来说，即便是 100GB 的内存要求，其实也不算太高，我们可以采用分治的思想，用多台机器（比如 20 台内存是 8GB 的机器）来存储这 10 亿网页链接
   - 基于链表实现的散列表存储URL的缺点:
     - 一方面，链表中的结点在内存中不是连续存储的，所以不能一下子加载到 CPU 缓存中，没法很好地利用到 CPU 高速缓存，所以数据访问性能方面会打折扣。
     - 另一方面，链表中的每个数据都是 URL，而 URL 不是简单的数字，是平均长度为 64 字节的字符串。也就是说，我们要让待判重的 URL，跟链表中的每个 URL，做字符串匹配。显然，这样一个字符串匹配操作，比起单纯的数字比对，要慢很多
7. **作为一个有追求的工程师，我们应该考虑，在添加、查询数据的效率以及内存消耗方面，我们是否还有进一步的优化空间呢?**

- 内存方面有明显的节省，那就得换一种解决方案，也就是我们今天要着重讲的这种存储结构，**布隆过滤器**（Bloom Filter）
- 在讲布隆过滤器前，要先讲一下另一种存储结构，**位图**（BitMap）。因为，布隆过滤器本身就是基于位图的，是对位图的一种改进。

一、位图

有 1 千万个整数，整数的范围在 1 到 1 亿之间。如何快速查找某个整数是否在这 1 千万个整数中呢?

- 解决方法
  - 可以使用一种比较“特殊”的散列表（位图），申请一个大小为 1 亿、数据类型为布尔类型（true 或者 false）的数组
  - 将这 1 千万个整数作为数组下标，将对应的数组值设置成 true
  - 整数 5 对应下标为 5 的数组值设置为 true，也就是 array[5]=true
  - 查询某个整数 K 是否在这 1 千万个整数中的时候，我们只需要将对应的数组值 array[K] 取出来，看是否等于 true。如果等于 true，那说明 1 千万整数中包含这个整数 K；相反，就表示不包含这个整数 K。

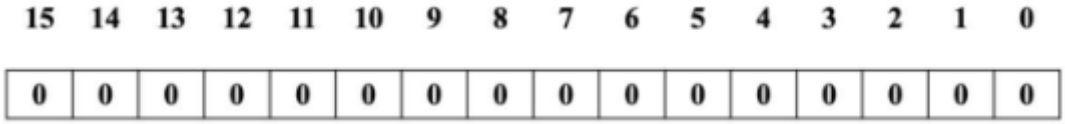
1.1 概念

位图（Bitmap）通常基于数组实现，将数组中的每个元素都看作一系列二进制数，所有元素一起组成更大的二进制集合，这样就可以大大节省空间。

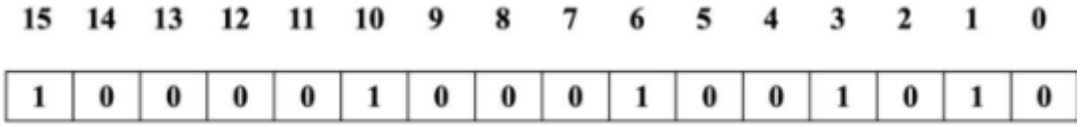
位图通常是用来判断某个数据存不存在的，常用于在Bloom Filter中判断数据是否存在，还可用于无重复整数的排序等，在大数据行业中使用广泛。

1.2 位图的数据结构

位图在内部维护了一个M×N维的数组`char[M][N]`，在这个数组里面每个字节占8位，因此可以存储M×N×8个数据。假如要存储的数据范围为0～15，则只需使用M=1, N=2的数据进行存储，具体的数据结构如图所示。



当要存储的数据为{1,3,6,10,15}时，只需将有数据的位设置为1，表示该位存在数据，将其他位设置为0，具体的数据结构如图所示。



1.3 代码实现

在Java中使用byte[]字节数组来存储bit，`1Byte = 8bit`。对于bit中的第i位，该bit为1则表示true，即数据存在；为0则表示false，即数据不存在。其具体实现分为数据结构的定义、查询方法的实现和修改方法的实现。

1.3.1 数据结构的定义

在如下代码中定义了一个名为Bitmap的类用于位图数据结构的存储，其中byte[]数组用于存储具体的数据，length用于记录数据的长度：

```
// 以bit为存储单位的数据结构，对于给定的第i位， 1表示true, 0表示false
public class Bitmap {
    private byte[] bytes;
    // length 为位图的长度，实际可操作的下标为[0, length)
    private int length;
    public Bitmap(int length) {
        this.length = length;
        bytes = new byte[length%8 == 0 ? length/1 : length/8 + 1];
    }
}
```

1.3.2 查询方法的实现

位图的查询操作为在拿到目标bit所在的Byte后，将其向右位移（并将高位置0），使目标bit在第1位，这样结果值就是目标bit值，方法如下。

1. 通过byte[index >> 3]（等价于byte[index/8]）取到目标bit所在的Byte。
2. 令i = index&7（等价于index%8），得到目标bit在该Byte中的位置。
3. 为了将目标bit前面的高位置0（这样位移后的值才等于目标bit本身），需要构建到目标bit为止的低位掩码，即01111111 >>>(7 - i)，再与原Byte做&运算。
4. 将结果向右位移i位，使目标bit处于第1位，结果值即为所求。具体的查询位图的Java代码实现

```
// 获取指定位的值
public boolean get(int index) {
    int i = index & 7;
    // 构建到index结束的低位掩码并做&运算(为了将高位置0)，然后将结果一直右移，知道目标位(index位)移到第一位，然后根据其值返回结果
    // >>>表示无符号右移，也叫逻辑右移，即若该数为正，则高位补0，而若该数为负数，则右移后高位同样补0
    if((bytes[index >> 3] & (01111111 >>> (7-i))) >> i ==0)
        return false
    else
        return true;
}
```

1.3.3 修改方法的实现

对位图的修改操作根据设定值true或false的不同，分为两种情况。

1. 如果value为true，则表示数据存在，将目标位与1做或运算，需要构建目标位为1、其他位为0的操作数。
2. 如果value为false，则表示数据不存在，将目标位与0做与运算，需要构建目标位为0、其他位为1的操作数。构建目标位为1且其他位为0的操作数的做法为：1 <<(index & 7)。修改位图的Java代码实现如下：

```
// 设置指定位的值
public void set(index, boolean value) {
    if(value) {
        // 通过给定位index，先定位到对应的Byte，并根据value值进行不同位的操作：
        // 1. 如果value为true，则目标位应该做或运算，构建"目标位为1， 其它位为0"的操作数，为了只合理操作目标位，而不影响其它位
        // 2. 如果value为false，则目标位应该做与运算，构建"目标位为0，其它位为1"的操作数
        bytes[index >> 3] |= 1 << (index & 7);
        // 二进制（ob\0B）
        // bytes[index/8] = bytes[index/8] | (ob001 << (index %8))
    } else {
        bytes[index >> 3] &= ~(1 << (index & 7))
    }
}
```

1.4 位图的优缺点

1.4.1 优点

- 从刚刚位图结构的讲解中，可以发现，位图通过数组下标来定位数据，所以，访问效率非常高。而且，每个数字用一个二进制位来表示，在数字范围不大的情况下，所需要的内存空间非常节省。
- 散列表存储这 1 千万的数据，数据是 32 位的整型数，也就是需要 4 个字节的存储空间，那总共至少需要 40MB 的存储空间。
- 通过位图的话，数字范围在 1 到 1 亿之间，只需要 1 亿个二进制位，也就是 12MB 左右的存储空间就够了。

1.4.2 缺点

- 数字的范围很大，比如刚刚那个问题，数字范围不是 1 到 1 亿，而是 1 到 10 亿，那位图的大小就是 10 亿个二进制位，也就是 120MB 的大小，消耗的内存空间，不降反增。

二、 布隆过滤器

- 问题：

  数据个数是 1 千万，数据的范围是 1 到 10 亿。

- 思路
  - 布隆过滤器的做法是，仍然使用一个 1 亿个二进制大小的位图，然后通过哈希函数，对数字进行处理，让它落在这 1 到 1 亿范围内。
  - 比如把哈希函数设计成 f(x)=x%n。其中，x 表示数字，n 表示位图的大小（1 亿），也就是，对数字跟位图的大小进行取模求余。
  - 哈希函数会存在冲突的问题
    - 一亿零一和 1 两个数字，经过你刚刚那个取模求余的哈希函数处理之后，最后的结果都是 1。这样就无法区分，位图存储的是 1 还是一亿零一了
    - 为了降低这种冲突概率
      - 可以设计一个复杂点、随机点的哈希函数
      - 既然一个哈希函数可能会存在冲突，那用多个哈希函数一块儿定位一个数据，是否能降低冲突的概率呢

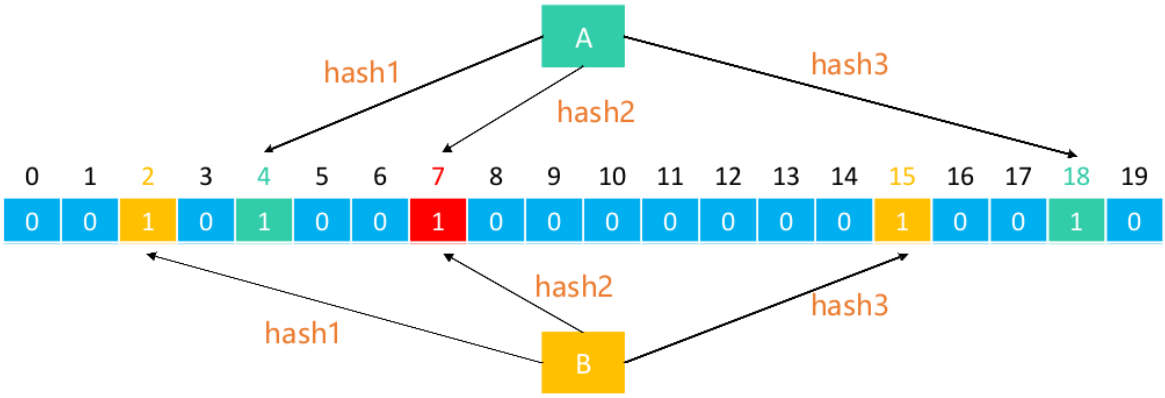
二、 布隆过滤器

2.1 概念

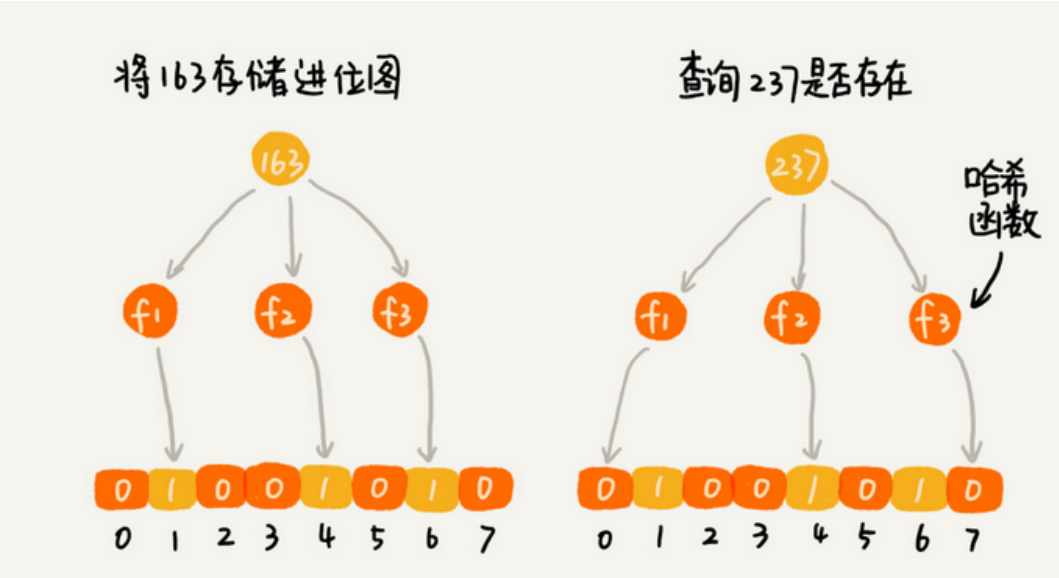
1970年由布隆提出

- 它是一个空间效率高的概率型数据结构，可以用来告诉你：一个元素一定不存在或者可能存在
- 优缺点
  - 优点：空间效率和查询时间都远远超过一般的算法
  - 缺点：有一定的误判率、删除困难
- 它实质上是一个很长的二进制向量和一系列随机映射函数（Hash函数）
- 常见应用
  - 网页黑名单系统、垃圾邮件过滤系统、爬虫的网址判重系统、解决缓存穿透问题

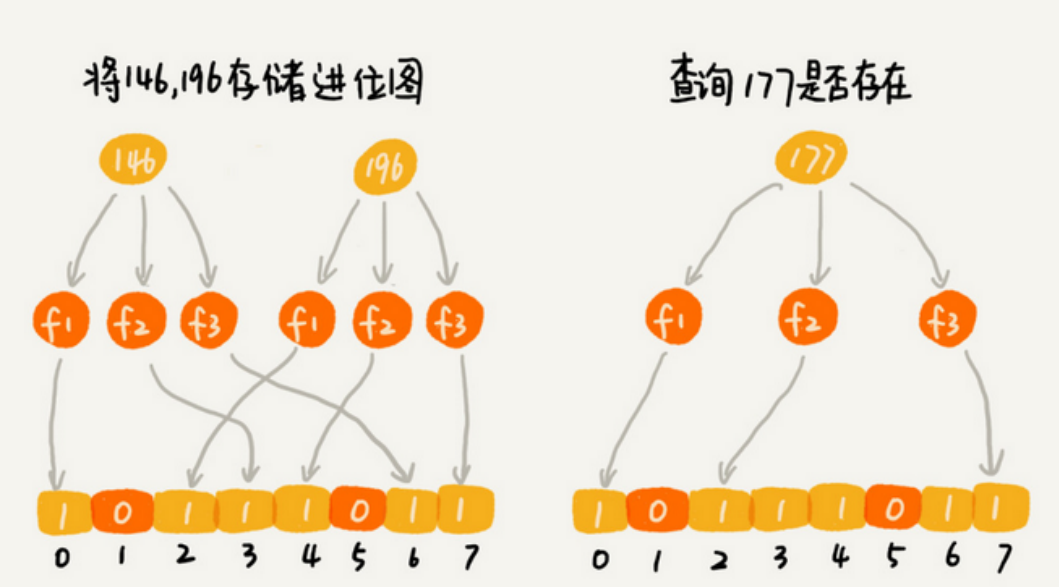
2.2 布隆过滤器的原理



- 假设布隆过滤器由 20位二进制、 3 个哈希函数组成，每个元素经过哈希函数处理都能生成一个索引位置
  - 添加元素：将每一个哈希函数生成的索引位置都设为 1
  - 查询元素是否存在
    ✓ 如果有一个哈希函数生成的索引位置不为 1，就代表不存在（100%准确）



✓ 如果每一个哈希函数生成的索引位置都为 1，就代表存在（存在一定的误判率）



- 添加、查询的时间复杂度都是：O(k)，k 是哈希函数的个数。空间复杂度是：O(m)，m 是二进制位的个数

2.3 布隆过滤器的误判率

- 误判率 p 受 3 个因素影响：二进制位的个数 m、哈希函数的个数 k、数据规模 n

$$\left(1-e^{-\frac{k(n+0.5)}{m-1}}\right)^k \qquad \left(1-e^{-\frac{kn}{m}}\right)^k$$

- 已知误判率 p、数据规模 n，求二进制位的个数 m、哈希函数的个数 k

$$m=-\frac{n\ln p}{(\ln 2)^2} \qquad k=\frac{m}{n}\ln 2$$

$$k=-\frac{\ln p}{\ln 2}=-\log_2 p$$

2.4 布隆过滤器的实现

- 主要接口

```
/**
* 添加元素1
*/
public boolean put(T value);
/**
* 判断一个元素是否存在
*/
public boolean contains(T value);
```

- 具体实现

```
public class BloomFilter<T> {
    /**
     * 二进制向量的长度(一共有多少个二进制位)
     */
    private int bitSize;
    /**
     * 二进制向量
     */
    private long[] bits;
    /**
     * 哈希函数的个数
     */
    private int hashSize;

    /**
     * @param n 数据规模
     * @param p 误判率, 取值范围(0, 1)
     */
    public BloomFilter(int n, double p) {
        if (n <= 0 || p <= 0 || p >= 1) {
            throw new IllegalArgumentException("wrong n or p");
        }

        double ln2 = Math.log(2);
        // 求出二进制向量的长度
        bitSize = (int) (- (n * Math.log(p)) / (ln2 * ln2));
        // 求出哈希函数的个数
        hashSize = (int) (bitSize * ln2 / n);
        // bits数组的长度
        bits = new long[(bitSize + Long.SIZE - 1) / Long.SIZE];
        // 每一页显示100条数据, pageSize
        // 一共有999999条数据, n
        // 请问有多少页 pageCount = (n + pageSize - 1) / pageSize
    }

    /**
     * 添加元素1
     */
    public boolean put(T value) {
        nullCheck(value);

        // 利用value生成2个整数
        int hash1 = value.hashCode();
        int hash2 = hash1 >>> 16;

        boolean result = false;
        for (int i = 1; i <= hashSize; i++) {
            int combinedHash = hash1 + (i * hash2);
            if (combinedHash < 0) {
                combinedHash = ~combinedHash;
            }
            // 生成一个二进位的索引
            int index = combinedHash % bitSize;
            // 设置index位置的二进位为1
            if (set(index)) result = true;
```

```
            //   1010101010010101
            // | 0000000000000100   1 << index
            //   1010101110010101
        }
        return result;
    }

    /**
     * 判断一个元素是否存在
     */
    public boolean contains(T value) {
        nullCheck(value);
        // 利用value生成2个整数
        int hash1 = value.hashCode();
        int hash2 = hash1 >>> 16;

        for (int i = 1; i <= hashSize; i++) {
            int combinedHash = hash1 + (i * hash2);
            if (combinedHash < 0) {
                combinedHash = ~combinedHash;
            }
            // 生成一个二进位的索引
            int index = combinedHash % bitSize;
            // 查询index位置的二进位是否为0
            if (!get(index)) return false;
        }
        return true;
    }

    /**
     * 设置index位置的二进位为1
     */
    private boolean set(int index) {
        long value = bits[index / Long.SIZE];
        int bitValue = 1 << (index % Long.SIZE);
        bits[index / Long.SIZE] = value | bitValue;
        return (value & bitValue) == 0;
    }

    /**
     * 查看index位置的二进位的值
     * @return true代表1, false代表0
     */
    private boolean get(int index) {
        long value = bits[index / Long.SIZE];
        return (value & (1 << (index % Long.SIZE))) != 0;
    }

    private void nullCheck(T value) {
        if (value == null) {
            throw new IllegalArgumentException("Value must not be null.");
        }
    }
}
```

2.5 爬虫网页去重的问题

用布隆过滤器来记录已经爬取过的网页链接，假设需要判重的网页有 10 亿，那我们可以用一个 10 倍大小的位图来存储，也就是 100 亿个二进制位，换算成字节，那就是大约 1.2GB。之前我们用散列表判重，需要至少 100GB 的空间。相比来讲，布隆过滤器在存储空间的消耗上，降低了非常多。、

布隆过滤器用**多个哈希函数**对同一个网页链接进行处理，CPU 只需要将网页链接从内存中读取一次，进行多次哈希计算，理论上讲这组操作是 CPU 密集型的。

而在散列表的处理方式中，需要读取散列冲突拉链的多个网页链接，分别跟待判重的网页链接，进行字符串匹配。这个操作涉及很多内存数据的读取，所以是内存密集型的。

CPU 计算可能是要比内存访问更快速的，所以，理论上讲，布隆过滤器的判重方式，更加快速。

2.6 缓存穿透

缓存穿透指由于缓存系统故障或者用户频繁查询系统中不存在（在系统中不存在，在自然数据库和缓存中都不存在）的数据，而这时请求穿过缓存不断被发送到数据库，导致数据库过载，进而引发一连串并发问题。

比如用户发起一个userName为zhangsan的请求，而在系统中并没有名为zhangsan的用户，这样就导致每次查询时在缓存中都找不到该数据，然后去数据库中再查询一遍。由于zhangsan用户本身在系统中不存在，自然返回空，导致请求穿过缓存频繁查询数据库，在用户频繁发送该请求时将导致数据库系统负载增大，从而可能引发其他问题。常用的解决缓存穿透问题的方法有布隆过滤器和cache null策略。

布隆过滤器：指将所有可能存在的数据都映射到一个足够大的Bitmap中，在用户发起请求时首先经过布隆过滤器的拦截，一个一定不存在的数据会被这个布隆过滤器拦截，从而避免对底层存储系统带来查询上的压力。

跳表：为什么Redis一定要用跳表来实现有序集合

- 问题

  二分查找底层依赖的是数组随机访问的特性，所以只能用数组来实现。

  如果数据存储在链表中，就真的没法用二分查找算法了吗？

- 跳表
  - 只需要对链表稍加改造，就可以支持类似“二分”的查找算法
  - 确实是一种各方面性能都比较优秀的**动态数据结构**，可以支持快速的插入、删除、查找操作，写起来也不复杂，甚至可以替代红黑树（Red-black tree）
- Redis 中的有序集合（Sorted Set）就是用跳表来实现的。如果你有一定基础，应该知道红黑树也可以实现快速的插入、删除和查找操作。那 Redis 为什么会选择用跳表来实现有序集合呢？ 为什么不用红黑树呢？学完今天的内容，你就知道答案了。

三、 跳表

3.1 关于链表的思考

- 一个有序链表搜索、添加、删除的平均时间复杂度是多少？
  - O(n)

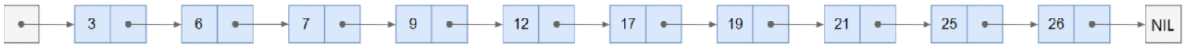
- 能否利用二分搜索优化有序链表，将搜索、添加、删除的平均时间复杂度降低至 O(logn)?
  - 链表没有像数组那样的高效随机访问（O(1) 时间复杂度），所以不能像有序数组那样直接进行二分搜索优化
  - 那有没有其他办法让有序链表搜索、添加、删除的平均时间复杂度降低至 O(logn)?
    - 使用跳表（SkipList）

3.2 跳表（SkipList）

1. 跳表，又叫做跳跃表、跳跃列表，在有序链表的基础上增加了“跳跃”的功能
2. 由William Pugh于1990年发布，设计的初衷是为了取代平衡树（比如红黑树）
3. Redis中 的 SortedSet、LevelDB 中的 MemTable 都用到了跳表
   - Redis、LevelDB 都是著名的 Key-Value 数据库
4. 对比平衡树
   - 跳表的实现和维护会更加简单
   - 跳表的搜索、删除、添加的平均时间复杂度是 O(logn)

3.3 使用跳表优化链表

1. 原始链表



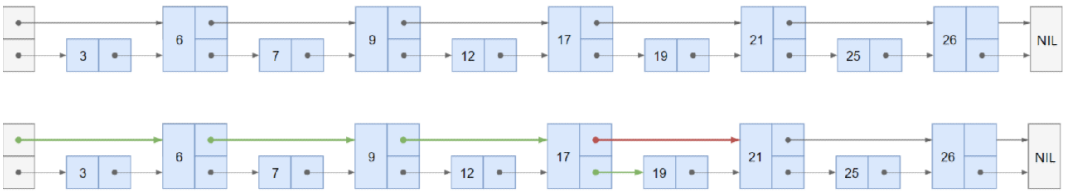
2. 对链表建立一级“索引”
   - 数组结构

     要查找某个结点，比如 20：

     可以先在索引层遍历，当遍历到索引层中值为 17 的结点时，发现下一个结点是 21，那要查找的结点20 肯定就在这两个结点之间。

     然后通过索引层数组的下一个索引对应的值，下降到原始链表这一层，继续遍历。

     这时，继续遍历19和19的下一个节点21, 测试发现19和21之前不存在20, 即20不存在链表中



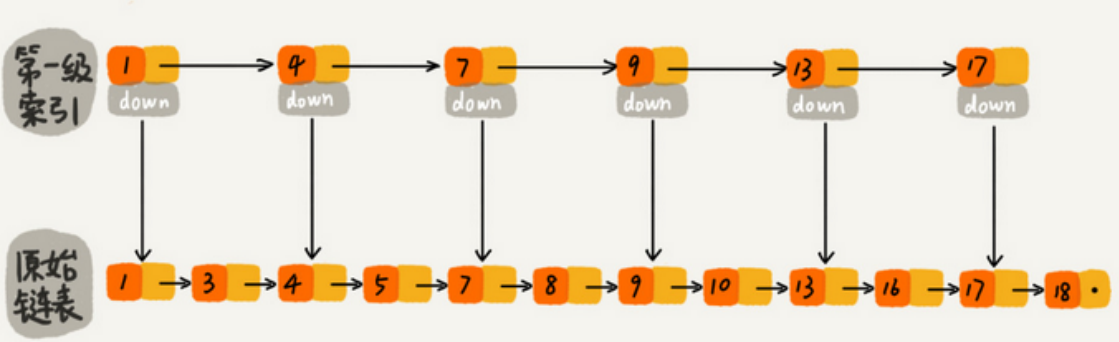
   - 链表结构

     要查找某个结点，比如 16：

     可以先在索引层遍历，当遍历到索引层中值为 13 的结点时，发现下一个结点是 17，那要查找的结点 16 肯定就在这两个结点之间。

     然后通过索引层结点的 down 指针，下降到原始链表这一层，继续遍历。

     这时，只需要再遍历 2 个结点，就可以找到值等于 16 的这个结点了。这样，原来如果要查找 16，需要遍历 10 个结点，现在只需要遍历 7 个结点。

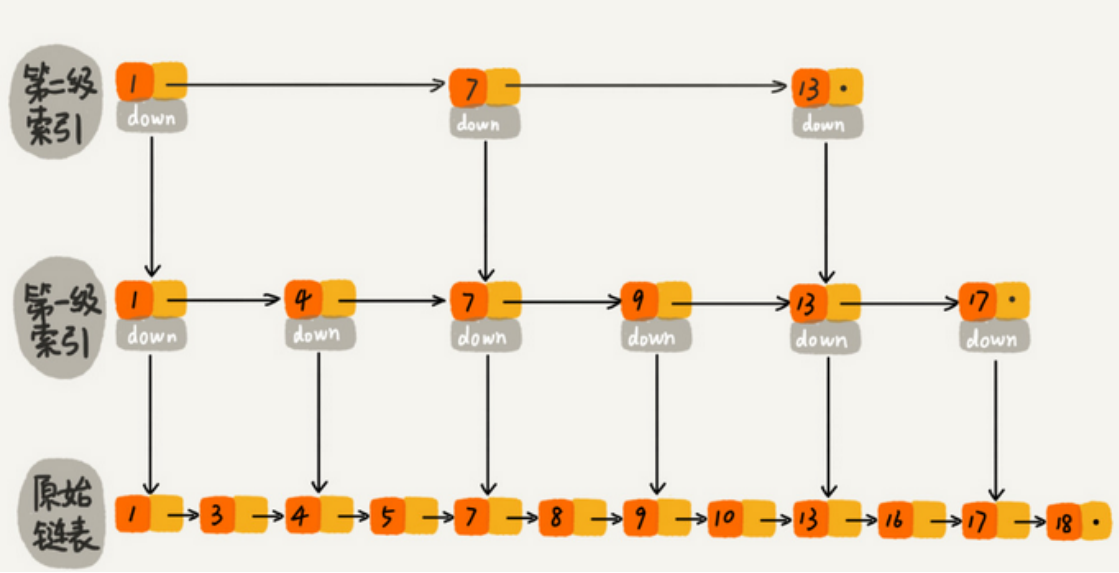


注：**加来一层索引之后，查找一个结点需要遍历的结点个数减少了，也就是说查找效率提高了**

3. 二 级索引

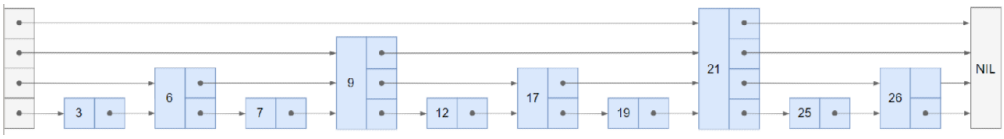
   跟前面建立第一级索引的方式相似，在第一级索引的基础之上，每两个结点就抽出一个结点到第二级索引。

   再来查找 16，只需要遍历 6 个结点了，需要遍历的结点数量又减少了。

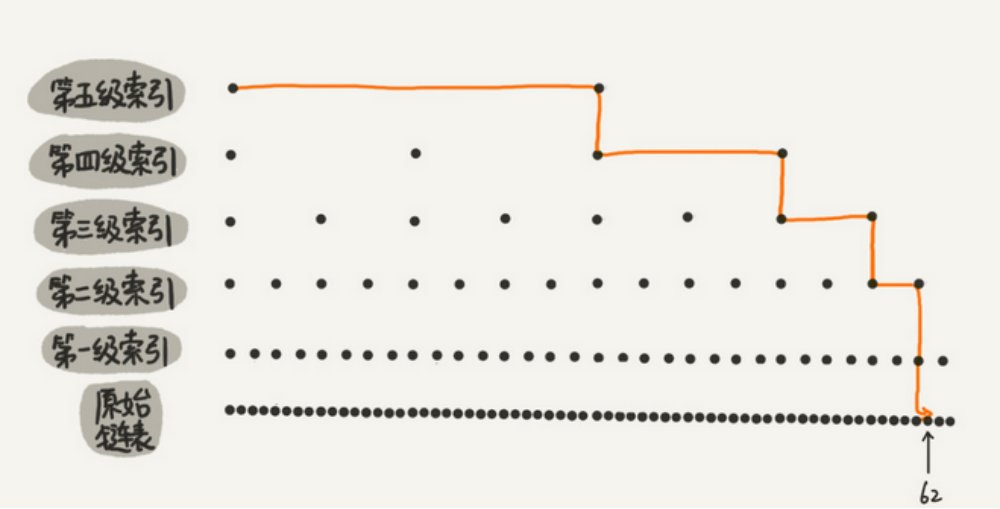


4. 多级索引

   - 数组结构索引（查找20）
     - 头部数组 通过第一个索引找到21，由于比20大
     - 头部数组 通过第二个索引找到9,9所以节点的数组第一个数组指向21， 由于20小于21
     - 开始通过9节点的数组第二个索引找到17， 17节点数组第一个索引指向21，由于20小于21
     - 开始通过17节点的数组第二个索引找到19
     - 19的下一个节点21，定位20不存在



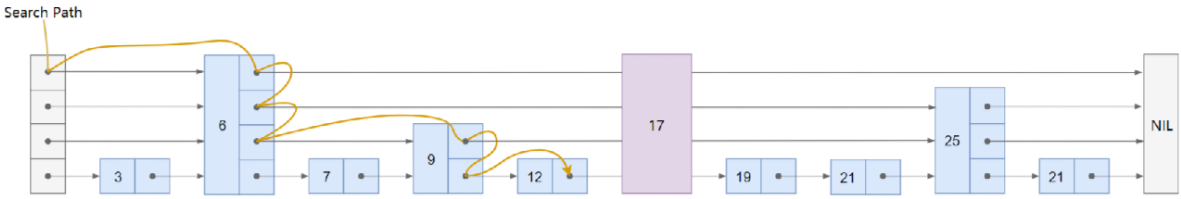
   - 链表结构索引
     - 原来没有索引的时候，查找 62 需要遍历 62 个结点，现在只需要遍历 11 个结点，速度是不是提高了很多？
     - 所以，当链表的长度 n 比较大时，比如 1000、10000 的时候，在构建索引之后，查找效率的提升就会非常明显。



5. 跳表的搜索

- 从顶层链表的首元素开始，从左往右搜索，直至找到一个大于或等于目标的元素，或者到达当前层链表的尾部
- 如果该元素等于目标元素，则表明该元素已被找到
- 如果该元素大于目标元素或已到达链表的尾部，则退回到当前层的前一个元素，然后转入下一层进行搜索

3.4 跳表的添加、删除



添加的细节

随机决定新添加元素的层数

删除的细节

删除一个元素后，整个跳表的层数可能会降低

3.5 跳表的层数

1. 跳表是按层构造的，底层是一个普通的有序链表，高层相当于是低层的“快速通道”
   - 在第 i 层中的元素按某个固定的概率 p（通常为 ½ 或 ¼ ）出现在第 i + 1层中，产生越高的层数，概率越低
     ✓ 元素层数恰好等于 1 的概率为 1 – p
     ✓ 元素层数大于等于 2 的概率为 p，而元素层数恰好等于 2 的概率为 p * (1 – p)
     ✓ 元素层数大于等于 3 的概率为 p^2，而元素层数恰好等于 3 的概率为 p^2 * (1 – p)
     ✓ 元素层数大于等于 4 的概率为 p^3，而元素层数恰好等于 4 的概率为 p^3 * (1 – p)
     ✓ .....
     ✓ 一个元素的平均层数是 1 / (1 – p)

$$1\times(1-p)+2p(1-p)+3p^2(1-p)+4p^3(1-p)+\ldots=(1-p)\sum_{k=1}^{+\infty}kp^{k-1}=(1-p)\cdot\frac{1}{(1-p)^2}=\frac{1}{1-p}$$

2. 当 p = ½ 时，每个元素所包含的平均指针数量是 2
3. 当 p = ¼ 时，每个元素所包含的平均指针数量是 1.33

3.6 跳表的复杂度分析

1. 每一层的元素数量
   □第 1 层链表固定有 n 个元素
   □第 2 层链表平均有 n * p 个元素
   □第 3 层链表平均有 n * p^2 个元素
   □第 k 层链表平均有 n * p^k 个元素
   □...
2. 另外
   □最高层的层数是 log 1/p n ，平均有个 1/p 元素
   □在搜索时，每一层链表的预期查找步数最多是 1/p，所以总的查找步数是 –(log p n/p)，时间复杂度是 O(logn)

3.7 为什么 Redis 要用跳表来实现有序集合，而不是红黑树?

- Redis 中的有序集合是通过跳表来实现的，严格点讲，其实还用到了散列表(本节请忽略)。
- Redis 中的有序集合支持的核心操作主要有下面这几个：
  - 插入一个数据；
  - 删除一个数据；
  - 查找一个数据；
  - 按照区间查找数据（比如查找值在 [100, 356] 之间的数据）；
  - 迭代输出有序序列。
- 其中，插入、删除、查找以及迭代输出有序序列这几个操作，红黑树也可以完成，时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。
- Redis 之所以用跳表来实现有序集合，还有其他原因
  - 跳表更容易代码实现(虽然跳表的实现也不简单，但比起红黑树来说还是好懂、好写多了，而简单就意味着可读性好，不容易出错)
  - 跳表更加灵活，它可以通过改变索引构建策略，有效平衡执行效率和内存消耗

注：跳表也不能完全替代红黑树。因为红黑树比跳表的出现要早一些，很多编程语言中的 Map 类型都是通过红黑树来实现的。我们做业务开发的时候，直接拿来用就可以了，不用费劲自己去实现一个红黑树，但是跳表并没有一个现成的实现，所以在开发中，如果你想使用跳表，必须要自己实现。

3.8 代码实现

```
mport java.util.Comparator;

@SuppressWarnings("unchecked")
public class SkipList<K, V> {
    private static final int MAX_LEVEL = 32;
    private static final double P = 0.25;
    private int size;
    private Comparator<K> comparator;
    /**
     * 有效层数
     */
    private int level;
    /**
     * 不存放任何K-V
     */
    private Node<K, V> first;
```

```
public SkipList(Comparator<K> comparator) {
    this.comparator = comparator;
    first = new Node<>(null, null, MAX_LEVEL);
}

public SkipList() {
    this(null);
}

public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}

public V get(K key) {
    keyCheck(key);

    // first.nexts[3] == 21节点
    // first.nexts[2] == 9节点
    // first.nexts[1] == 6节点
    // first.nexts[0] == 3节点

    // key = 30
    // level = 4

    Node<K, V> node = first;
    for (int i = level - 1; i >= 0; i--) {
        int cmp = -1;
        while (node.nexts[i] != null
                && (cmp = compare(key, node.nexts[i].key)) > 0) {
            node = node.nexts[i];
        }
        // node.nexts[i].key >= key
        if (cmp == 0) return node.nexts[i].value;
    }
    return null;
}

public V put(K key, V value) {
    keyCheck(key);

    Node<K, V> node = first;
    Node<K, V>[] prevs = new Node[level];
    for (int i = level - 1; i >= 0; i--) {
        int cmp = -1;
        while (node.nexts[i] != null
                && (cmp = compare(key, node.nexts[i].key)) > 0) {
            node = node.nexts[i];
        }
        if (cmp == 0) { // 节点是存在的
            V oldV = node.nexts[i].value;
            node.nexts[i].value = value;
            return oldV;
        }
```

```
            prevs[i] = node;
        }

        // 新节点的层数
        int newLevel = randomLevel();
        // 添加新节点
        Node<K, V> newNode = new Node<>(key, value, newLevel);
        // 设置前驱和后继
        for (int i = 0; i < newLevel; i++) {
            if (i >= level) {
                first.nexts[i] = newNode;
            } else {
                newNode.nexts[i] = prevs[i].nexts[i];
                prevs[i].nexts[i] = newNode;
            }
        }

        // 节点数量增加
        size++;

        // 计算跳表的最终层数
        level = Math.max(level, newLevel);

        return null;
    }

    public V remove(K key) {
        keyCheck(key);

        Node<K, V> node = first;
        Node<K, V>[] prevs = new Node[level];
        boolean exist = false;
        for (int i = level - 1; i >= 0; i--) {
            int cmp = -1;
            while (node.nexts[i] != null
                    && (cmp = compare(key, node.nexts[i].key)) > 0) {
                node = node.nexts[i];
            }
            prevs[i] = node;
            if (cmp == 0) exist = true;
        }
        if (!exist) return null;

        // 需要被删除的节点
        Node<K, V> removedNode = node.nexts[0];

        // 数量减少
        size--;

        // 设置后继
        for (int i = 0; i < removedNode.nexts.length; i++) {
            prevs[i].nexts[i] = removedNode.nexts[i];
        }

        // 更新跳表的层数
        int newLevel = level;
        while (--newLevel >= 0 && first.nexts[newLevel] == null) {
            level = newLevel;
```

```
        }

        return removedNode.value;
    }

    private int randomLevel() {
        int level = 1;
        while (Math.random() < P && level < MAX_LEVEL) {
            level++;
        }
        return level;
    }

    private void keyCheck(K key) {
        if (key == null) {
            throw new IllegalArgumentException("key must not be null.");
        }
    }

    private int compare(K k1, K k2) {
        return comparator != null
                ? comparator.compare(k1, k2)
                : ((Comparable<K>)k1).compareTo(k2);
    }

    private static class Node<K, V> {
        K key;
        V value;
        Node<K, V>[] nexts;
//      Node<K, V> right;
//      Node<K, V> down;
//      Node<K, V> top;
//      Node<K, V> left;
        public Node(K key, V value, int level) {
            this.key = key;
            this.value = value;
            nexts = new Node[level];
        }
        @Override
        public String toString() {
            return key + ":" + value + "_" + nexts.length;
        }
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("一共" + level + "层").append("\n");
        for (int i = level - 1; i >= 0; i--) {
            Node<K, V> node = first;
            while (node.nexts[i] != null) {
                sb.append(node.nexts[i]);
                sb.append(" ");
                node = node.nexts[i];
            }
            sb.append("\n");
        }
        return sb.toString();
```

```
    }
}
```