Java中的容器

Collection是整个集合框架的基础，其中有分为collection和map。

Collection：set和list

List是有序存储并且可以重复，而set是无序存储不可以重复。

List：arraylist底层是数组实现的，在增加和删除的时候涉及到数组的扩容等问题所以比较慢，但是其查询由于存在索引所以比较快。

其底层维护了一个长度为10的object[]数组，每次扩容0.5倍+1。

Linkedlist底层是链表进行实现的，增加时候只需要让前一个元素记住自己就行，所以在增删的时候比较快，但是查询速度慢。线程是不安全的。

Vector：底层原理同arraylist相同并且是线程安全的，但是执行的效率低。每次扩容一倍。

Set:

Hashset:线程不是安全的，底层是使用hash表来实现的，存取速度快。

它的底层是通过hashmap来实现的，由于hashmap的key值不能重复（这个是用equals和hashcode（）来进行比较的），而Value时有一个统一的值private static final Object PRESENT = new Object();，(定义一个虚拟的Object对象作为HashMap的value，将此对象定义为static final。)

默认初始的容量为16，加载因子为0.75，每次扩容一倍。

Treeset：

底层是使用红黑树进行实现的，红黑树是一种自平衡排序二叉树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点。

由于插入元素的时候需要循环找到插入元素的位置所以比较慢，但是查询的速度快！

LinkedHashSet

Map：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及Properties等

1. hashmap 和 hashtable的区别？

Hashmap没有考虑到同步的问题是线程不安全的，hashtable考虑到了线程安全的问题；hashmap允许k和v为null，后者不允许；hashmap是继承于abstractmap类；而hashtable是继承于dictnory类。

1. hashmap是怎么解决hash冲突的？

Hash是把任意长度的输入通过散列变换得到固定长度的输出，该输出就是散列值。所有的散列函数都是有一个特征：如果根据散列变换得到的结果不同他们的输入肯定是不同的，但是反之则不一定成立。

两个不同的输入通过hash的散列变换得到了相同的值就叫做hash碰撞。

<https://www.jianshu.com/p/939b8a672070>

解决hash冲突的方法：（任何函数都无法彻底避免碰撞）

开放定址法：一旦发生冲突就去寻找下一个空的散列地址。只要散列地址足够大，空的散列地址总能找到，并将数据记入。

再哈希法：当哈希地址冲突的时候用其它函数计算出另一个哈希地址，直到不冲突为止。

链表地址法：将哈希表的每个单元作为链表的头结点，所有哈希地址为i的元素构成一个同义词链表。即发生冲突时就把该关键字链在以该单元为头结点的链表的尾部。

1. Hashmap的数据结构？

在java中保存数据有两种简单的数据结构：数组和链表。数据是查询快，插入慢。链表是刚好相反，所以在jdk1.8之前，hashmap的底层是采用数组和链表这样的一种形式。并采用链地址法解决hash的冲突。

由于hashmap的初始容量为16，我们如果简单采用hashcode来取余（因为hashcode位数较高，16对应2进制是4位，所以hashcode（）的高位是没有参与运算的）会大大提高hash碰撞的概率。所以我们的思想就是让hashcode的高位也参与运算，进一步降低hash碰撞的概率，使得数据分布的平均，这种操作称为扰动。（高低位异或运算）

Jdk1.7中是4次位运算，5次异或运算。而jdk1.8中是一次位运算一次异或运算。

在jdk1.8下某一个bucket下对应的链表长度为n，所以时间复杂度为O（n），当n的长度大于8时就采用红黑树的结构此时的时间复杂度为O（logn）

1. 使用链地址法（使用散列表）来链接拥有相同hash值的数据；

2. 使用2次扰动函数（hash函数）来降低哈希冲突的概率，使得数据分布更平均；

3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快；

HashMap：

底层维护一个Entry<V,V>的数组，数组元素Node<K,V>实现了Entry接口。首先有一个每个元素都是链表（可能表述不准确）的数组，当添加一个元素（key-value）时，就首先计算元素key的hash值，以此确定插入数组中的位置，但是可能存在同一hash值的元素已经被放在数组同一位置了，这时就添加到同一hash值的元素的后面，他们在数组的同一位置，但是形成了链表，同一各链表上的Hash值是相同的，所以说数组存放的是链表。而当链表长度太长时，链表就转换为红黑树，这样大大提高了查找的效率。

加载因子（默认0.75）：为什么需要使用加载因子，为什么需要扩容呢？因为如果填充比很大，说明利用的空间很多，如果一直不进行扩容的话，链表就会越来越长，这样查找的效率很低，因为链表的长度很大（当然最新版本使用了红黑树后会改进很多），扩容之后，将原来链表数组的每一个链表分成奇偶两个子链表分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率。

get(key)方法时获取key的hash值，计算hash&(n-1)得到在链表数组中的位置first=tab[hash&(n-1)],先判断first的key是否与参数key相等，不等就遍历后面的链表找到相同的key值返回对应的Value值即可

下面简单说下添加键值对put(key,value)的过程：

1，判断键值对数组tab[]是否为空或为null，否则以默认大小resize()；

2，根据键值key计算hash值得到插入的数组索引i，如果tab[i]==null，直接新建节点添加，否则转入3

3，判断当前数组中处理hash冲突的方式为链表还是红黑树(check第一个节点类型即可),分别处理

构造hash表时，如果不指明初始大小，默认大小为16（即Node数组大小16），如果Node[]数组中的元素达到（填充比\*Node.length）重新调整HashMap大小 变为原来2倍大小,扩容很耗时。

5.HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标？

1）hashCode()方法返回的是int整数类型，其范围为-(2 ^ 31)~(2 ^ 31 - 1)，约有40亿个映射空间，而HashMap的容量范围是在16（初始化默认值）~2 ^ 30，HashMap通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，从而导致通过hashCode()计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置；

答：

HashMap自己实现了自己的hash()方法，通过两次扰动使得它自己的哈希值高低位自行进行异或运算，降低哈希碰撞概率也使得数据分布更平均；

在保证数组长度为2的幂次方的时候，使用hash()运算之后的值与运算（&）（数组长度 - 1）来获取数组下标的方式进行存储，这样一来是比取余操作更加有效率，二来也是因为只有当数组长度为2的幂次方时，h&(length-1)才等价于h%length，三来解决了“哈希值与数组大小范围不匹配”的问题；

1. hashmap为什么长度要保证是2的幂次方？
2. 只有当为2的幂次方的时候，h&（length-1）才等价于h%length，即实现了key的定位，2的幂次方也减少了冲突的次数，提高了hashmap的查询效率。
3. 如果hashmap的长度为2的幂次方的时候，则length-1必定是1111。。的形式，如果不是2的幂次方的时候可能就是导致中间的某些位数出现了0的情况，这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

为什么在jdk1.8中是两次扰动？

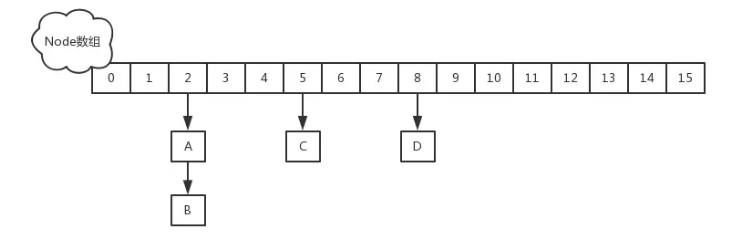
因为两次已经足够让高低位参与运算，减少hash冲突。

1. concurrenthashmap 和 hashtable的区别？

concurrenthashmap 结合了hashmap 和 hashtable的优点。HashMap 没有考虑同步，HashTable 考虑了同步的问题。但是 HashTable 在每次同步执行时都要锁住整个结构。 ConcurrentHashMap 锁的方式是稍微细粒度的。

在jdk1.7中，concurrenthashmap 采用segment + hashentry来实现的，segment是一种可重入的锁，每个 Segment 守护一个HashEntry 数组里得元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment 锁。

在JDK1.8中，放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现，结构如下：



1. 如果相应的node没有发生变化，则采用CAS插入相关的数据。
2. 如果相应的node不为空，且当前该节点不处于移动状态，则对该节点加synchronized锁，
3. 如果该节点是TreeBin类型的节点，说明是红黑树结构，则通过putTreeVal方法往红黑树中插入节点；如果binCount不为0，说明put操作对数据产生了影响，如果当前链表的个数达到8个，则通过treeifyBin方法转化为红黑树，如果oldVal不为空，说明是一次更新操作，没有对元素个数产生影响，则直接返回旧值；
4. 如果插入的是一个新节点，则执行addCount()方法尝试更新元素个数baseCount；