# 集合类

## List和Set比较，各自的子类比较

#### **Arraylist与LinkedList的比较**

1、ArrayList是实现了基于动态数组的数据结构,因为地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。

2、因为地址连续， ArrayList要移动数据,所以插入和删除操作效率比较低。

3、LinkedList基于链表的数据结构,地址是任意的，所以在开辟内存空间的时候不需要等一个连续的地址，对于新增和删除操作add和remove，LinedList比较占优势。

4、因为LinkedList要移动指针,所以查询操作性能比较低。

**适用场景分析：**

当需要对数据进行对此访问的情况下选用ArrayList，当需要对数据进行多次增加删除修改时采用LinkedList。

#### **ArrayList与Vector的比较**

1 Vector的方法都是同步的，是线程安全的，而ArrayList的方法不是，由于线程的同步必然要影响性能。因此，ArrayList的性能比Vector好。

2、当Vector或ArrayList中的元素超过它的初始大小时，Vector会将它的容量翻倍，而ArrayList只增加50%的大小，这样。ArrayList就有利于节约内存空间。

3、大多数情况不使用Vector，因为性能不好，但是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性。

4、Vector可以设置增长因子，而ArrayList不可以。

**适用场景分析：**

1、Vector是线程同步的，所以它也是线程安全的，而ArrayList是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用ArrayList效率比较高。

2、如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用Vector有一定的优势。

#### **HashSet与TreeSet的比较**

1.TreeSet 是二叉树实现的，Treeset中的数据是自动排好序的，不允许放入null值 。

2.HashSet 是哈希表实现的，HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束 。

3.HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例。

**适用场景分析：**

HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。

#### ****HashMap(线程不安全)****

##### Jdk1.8之前

HashMap 的实例有两个参数影响其性能：“**初始容量**” 和 “**加载因子**”。容量 是哈希表中桶的数量，初始容量 只是哈希表在创建时的容量。加载因子 是哈希表在其容量自动增加之前可以达到多满的一种尺度。当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 rehash 操作（即重建内部数据结构），从而哈希表将具有大约两倍的桶数。

通常，**默认加载因子是 0.75**, 这是在时间和空间成本上寻求一种折衷。加载因子过高虽然减少了空间开销，但同时也增加了查询成本（在大多数 HashMap 类的操作中，包括 get 和 put 操作，都反映了这一点）。在设置初始容量时应该考虑到映射中所需的条目数及其加载因子，以便最大限度地减少 rehash 操作次数。如果初始容量大于最大条目数除以加载因子，则不会发生 rehash 操作

##### Jdk1.8之后

而JDK1.8中，HashMap采用数组+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

在JDK 1.8中，hash方法如下：

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

（1）首先获取对象的hashCode()值，然后将hashCode值右移16位，然后将右移后的值与原来的hashCode做异或运算，返回结果。（其中h>>>16，在JDK1.8中，优化了高位运算的算法，使用了零扩展，无论正数还是负数，都在高位插入0）。

（2）在putVal源码中，我们通过(n-1)&hash获取该对象的键在hashmap中的位置。（其中hash的值就是（1）中获得的值）其中n表示的是hash桶数组的长度，并且该长度为2的n次方，这样(n-1)&hash就等价于hash%n。因为&运算的效率高于%运算。

#### **HashMap的数据存储实现原理**

流程：

1. 根据key计算得到key.hash = (h = k.hashCode()) ^ (h >>> 16)；

2. 根据key.hash计算得到桶数组的索引index = key.hash & (table.length - 1)，这样就找到该key的存放位置了：

① 如果该位置没有数据，用该数据新生成一个节点保存新数据，返回null；

② 如果该位置有数据是一个红黑树，那么执行相应的插入 / 更新操作；

③ 如果该位置有数据是一个链表，分两种情况一是该链表没有这个节点，另一个是该链表上有这个节点，注意这里判断的依据是key.hash是否一样：

如果该链表没有这个节点，那么采用尾插法新增节点保存新数据，返回null；如果该链表已经有这个节点了，那么找到该节点并更新新数据，返回老数据。

#### ****ConcurrentHashMap****

## 锁分段技术

 HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，

#### HashMap和HashTable有什么不同？

虽然都实现了Map、Cloneable、Serializable三个接口。但是HashMap继承自抽象类AbstractMap，而HashTable继承自抽象类Dictionary。其中Dictionary类是一个已经被废弃的类