# Java进阶面试题

## 说说常见的集合有哪些吧？

Map接口和Collection接口是所有集合框架的父接口：

1. Collection接口的子接口包括：Set接口和List接口
2. Map接口的实现类主要有：HashMap、TreeMap、Hashtable、ConcurrentHashMap以及Properties等
3. Set接口的实现类主要有：HashSet、TreeSet、LinkedHashSet等
4. List接口的实现类主要有：ArrayList、LinkedList、Stack以及Vector等

## ****List、Set、Map 之间的区别是什么？****



## ****2.如何实现数组和 List 之间的转换？****

List转换成为数组：调用ArrayList的toArray方法。  
数组转换成为List：调用Arrays的asList方法。

## 3.ArrayList和Vector的区别（是否有序、是否重复、数据结构、底层实现）

ArrayList和Vector都实现了List接口，他们都是有序集合，并且存放的元素是允许重复的。它们的底层都是通过数组来实现的，因此列表这种数据结构检索数据速度快，但增删改速度慢。

而ArrayList和Vector的区别主要在两个方面：

第一，线程安全。Vector是线程安全的，而ArrayList是线程不安全的。因此在如果集合数据只有单线程访问，那么使用ArrayList可以提高效率。而如果有多线程访问你的集合数据，那么就必须要用Vector，因为要保证数据安全。

第二，数据增长。ArrayList和Vector都有一个初始的容量大小，当存储进它们里面的元素超过了容量时，就需要增加它们的存储容量。ArrayList每次增长原来的0.5倍，而Vector增长原来的一倍。ArrayList和Vector都可以设置初始空间的大小，Vector还可以设置增长的空间大小，而ArrayList没有提供设置增长空间的方法。

## ****HashMap和Hashtable的区别****

HashMap和Hashtable都实现了Map接口，并且都是key-value的数据结构。它们的不同点主要在三个方面：

第一，Hashtable是Java1.1的一个类，它基于陈旧的Dictionary类。而HashMap是Java1.2引进的Map接口的一个实现。

第二，Hashtable是线程安全的，也就是说是线程同步的，而HashMap是线程不安全的。也就是说在单线程环境下应该用HashMap，这样效率更高。

第三，HashMap允许将null值作为key或value，但Hashtable不允许（会抛出NullPointerException）。

## **5.HashMap的put方法的具体流程？**

首先看看源码：

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,

boolean evict) {

HashMap.Node<K,V>[] tab; HashMap.Node<K,V> p; int n, i;

// 1.如果table为空或者长度为0，即没有元素，那么使用resize()方法扩容

if ((tab = table) == null || (n = tab.length) == 0)

n = (tab = resize()).length;

// 2.计算插入存储的数组索引i，此处计算方法同 1.7 中的indexFor()方法

// 如果数组为空，即不存在Hash冲突，则直接插入数组

if ((p = tab[i = (n - 1) & hash]) == null)

tab[i] = newNode(hash, key, value, null);

// 3.插入时，如果发生Hash冲突，则依次往下判断

else {

HashMap.Node<K,V> e; K k;

// a.判断table[i]的元素的key是否与需要插入的key一样，若相同则直接用新的value覆盖掉旧的value

// 判断原则equals() - 所以需要当key的对象重写该方法

if (p.hash == hash &&

((k = p.key) == key || (key != null && key.equals(k))))

e = p;

// b.继续判断：需要插入的数据结构是红黑树还是链表

// 如果是红黑树，则直接在树中插入 or 更新键值对

else if (p instanceof HashMap.TreeNode)

e = ((HashMap.TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

// 如果是链表，则在链表中插入 or 更新键值对

else {

// i .遍历table[i]，判断key是否已存在：采用equals对比当前遍历结点的key与需要插入数据的key

// 如果存在相同的，则直接覆盖

// ii.遍历完毕后任务发现上述情况，则直接在链表尾部插入数据

// 插入完成后判断链表长度是否 > 8：若是，则把链表转换成红黑树

for (int binCount = 0; ; ++binCount) {

if ((e = p.next) == null) {

p.next = newNode(hash, key, value, null);

if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st

treeifyBin(tab, hash);

break;

}

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

break;

p = e;

}

}

// 对于i 情况的后续操作：发现key已存在，直接用新value覆盖旧value&返回旧value

if (e != null) { // existing mapping for key

V oldValue = e.value;

if (!onlyIfAbsent || oldValue == null)

e.value = value;

afterNodeAccess(e);

return oldValue;

}

}

++modCount;

// 插入成功后，判断实际存在的键值对数量size > 最大容量

// 如果大于则进行扩容

if (++size > threshold)

resize();

// 插入成功时会调用的方法（默认实现为空）

afterNodeInsertion(evict);

return null;}

