Java集合必会14问（精选面试题整理）

转自：https://www.jianshu.com/p/939b8a672070

> 前言：把这段时间复习的关于集合类的东西整理出来，特别是HashMap相关的一些东西，之前都没有很注意1.7 ->> 1.8的变化问题，但后来发现这其实变化挺大的，而且很多整理的面试资料都没有更新（包括我之前整理的...）

1）说说常见的集合有哪些吧?

答：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

2）HashMap与HashTable的区别?

答：

1. HashMap没有考虑同步，是线程不安全的；Hashtable使用了synchronized关键字，是线程安全的；
2. HashMap允许K/V都为null；后者K/V都不允许为null；
3. HashMap继承自AbstractMap类；而Hashtable继承自Dictionary类；

3）HashMap的put方法的具体流程?

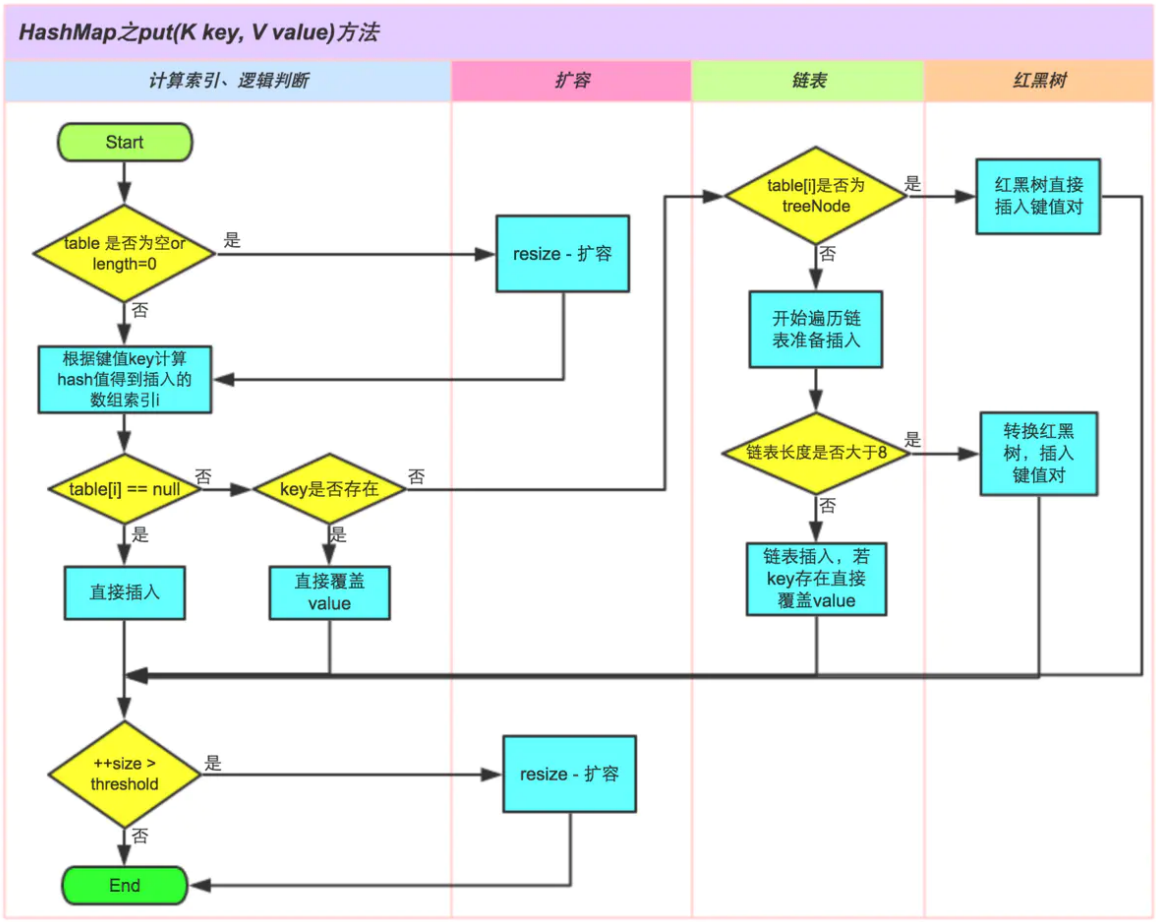
> 图引用自：https://blog.csdn.net/u011240877/article/details/53358305

答：下面先来分析一下源码

```
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;
}
```

图片简单总结为：



4）HashMap的扩容操作是怎么实现的?

答：通过分析源码我们知道了HashMap通过`resize()`方法进行扩容或者初始化的操作，下面是对源码进行的一些简单分析：

```
/**
 * 该函数有2中使用情况：1.初始化哈希表；2.当前数组容量过小，需要扩容
 */
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;// 扩容前的数组（当前数组）
    int oldCap = (oldTab == null) ? 0 : oldTab.length;// 扩容前的数组容量（数组长度）
    int oldThr = threshold;// 扩容前数组的阈值
    int newCap, newThr = 0;

    if (oldCap > 0) {
        // 针对情况2：若扩容前的数组容量超过最大值，则不再扩容
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        // 针对情况2：若没有超过最大值，就扩容为原来的2倍（左移1位）
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }

    // 针对情况1：初始化哈希表（采用指定或者使用默认值的方式）
```

```
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }

    // 计算新的resize上限
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes","unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        // 把每一个bucket都移动到新的bucket中去
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
```

```
            }
        }
        return newTab;
    }
```

5）HashMap是怎么解决哈希冲突的?

> 参考资料：https://juejin.im/post/5ab99afff265da23a2291dee

答：在解决这个问题之前，我们首先需要知道**什么是哈希冲突**，而在了解哈希冲突之前我们还要知道**什么是哈希**才行；

什么是哈希?

Hash，一般翻译为“散列”，也有直接音译为“哈希”的，这就是把任意长度的输入通过散列算法，变换成固定长度的输出，该输出就是散列值（哈希值）；这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来唯一的确定输入值。**简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。**

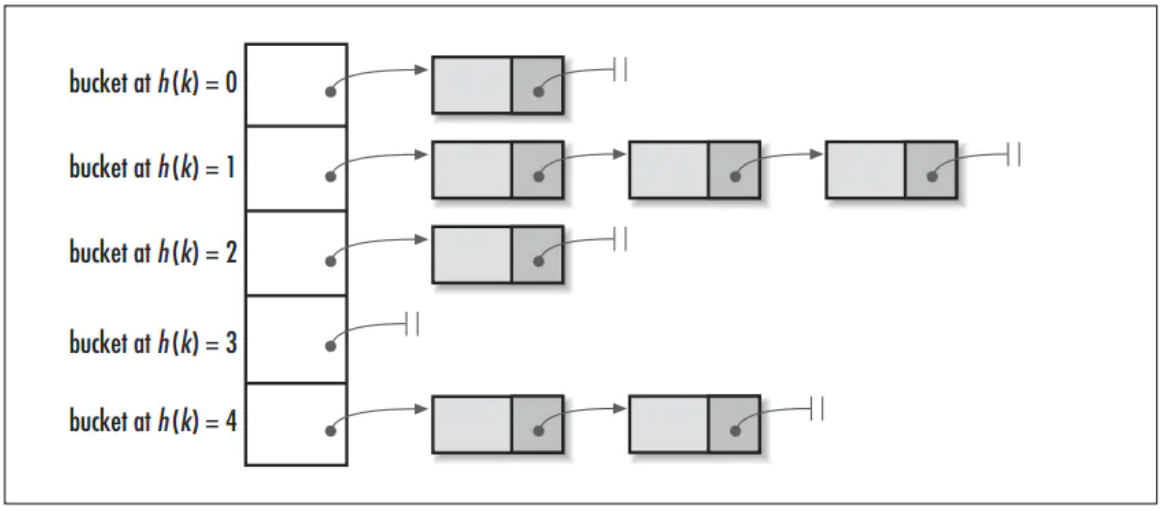
所有散列函数都有如下一个基本特性：**根据同一散列函数计算出的散列值如果不同，那么输入值肯定也不同。但是，根据同一散列函数计算出的散列值如果相同，输入值不一定相同。**

什么是哈希冲突?

当两个不同的输入值，根据同一散列函数计算出相同的散列值的现象，我们就把它叫做碰撞（哈希碰撞）。

HashMap的数据结构

在Java中，保存数据有两种比较简单的数据结构：数组和链表。**数组的特点是：寻址容易，插入和删除困难；链表的特点是：寻址困难，但插入和删除容易；**所以我们将数组和链表结合在一起，发挥两者各自的优势，使用一种叫做**链地址法**的方式可以解决哈希冲突：



这样我们就可以将拥有相同哈希值的对象组织成一个链表放在hash值所对应的bucket下，**但相比于hashCode返回的int类型，我们HashMap初始的容量大小**`DEFAULT_INITIAL_CAPACITY = 1 << 4`**（即2的四次方16）要远小于int类型的范围，所以我们如果只是单纯的用hashCode取余来获取对应的bucket这将会大大增加哈希碰撞的概率，并且最坏情况下还会将HashMap变成一个单链表，**所以我们还需要对hashCode作一定的优化

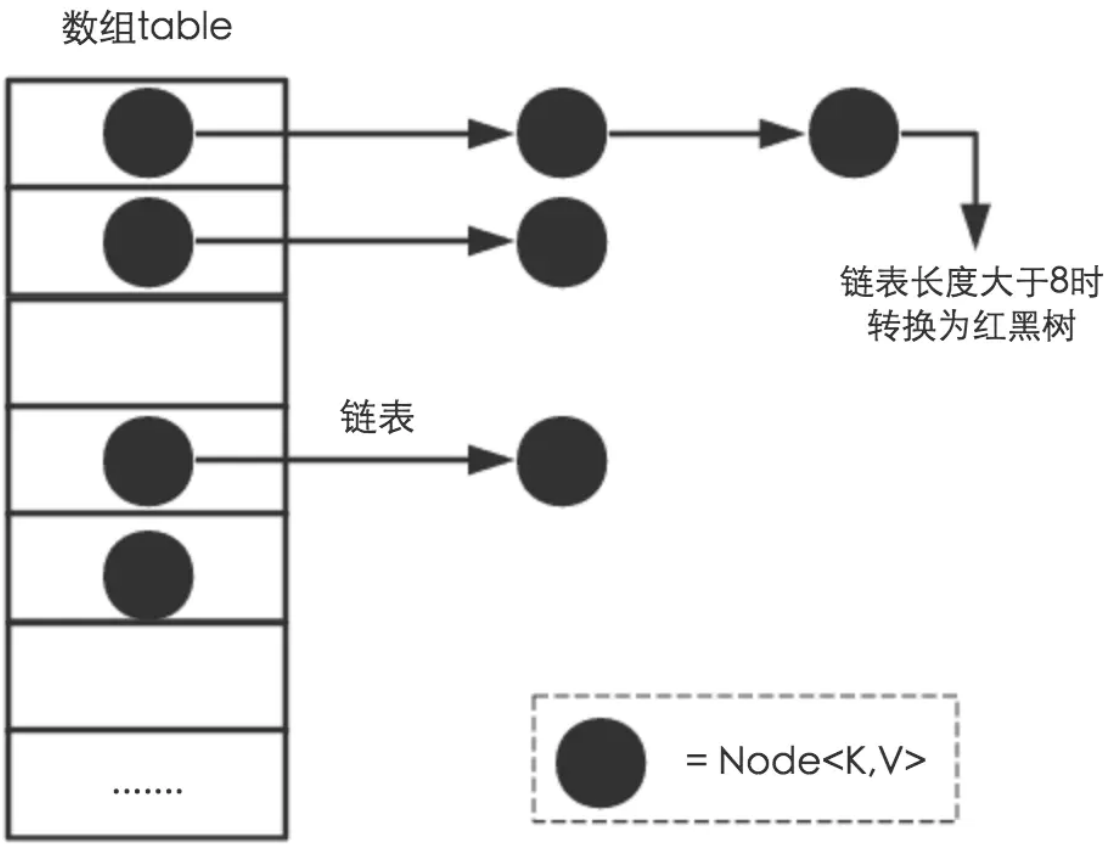
hash()函数

上面提到的问题，主要是因为如果使用hashCode取余，那么相当于**参与运算的只有hashCode的低位**，高位是没有起到任何作用的，所以我们的思路就是让hashCode取值出的高位也参与运算，进一步降低hash碰撞的概率，使得数据分布更平均，我们把这样的操作称为**扰动**，在**JDK 1.8**中的hash()函数如下：

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);// 与自己右移16位进行异或运算（高低位异或）
}
```

这比在**JDK 1.7**中，更为简洁，**相比在1.7中的4次位运算，5次异或运算（9次扰动），在1.8中，只进行了1次位运算和1次异或运算（2次扰动）；**

JDK1.8新增红黑树



通过上面的**链地址法（使用散列表）**和**扰动函数**我们成功让我们的数据分布更平均，哈希碰撞减少，但是当我们的HashMap中存在大量数据时，加入我们某个bucket下对应的链表有n个元素，那么遍历时间复杂度就为O(n)，为了针对这个问题，JDK1.8在HashMap中新增了红黑树的数据结构，进一步使得遍历复杂度降低至O(logn)；

总结

简单总结一下HashMap是使用了哪些方法来有效解决哈希冲突的：

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；

6）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?

答：`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；

面试官：那怎么解决呢?

答：

1. 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；
2. 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；

面试官：为什么数组长度要保证为2的幂次方呢?

答：

1. 只有当数组长度为2的幂次方时，h&(length-1)才等价于h%length，即实现了key的定位，2的幂次方也可以减少冲突次数，提高HashMap的查询效率；
2. 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111……的形式，在于 h 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，在于 h 与操作，最后一位都为 0 ，而 0001，0011，0101，1001，1011，0111，1101 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。

面试官：那为什么是两次扰动呢?

答：这样就是加大哈希值低位的随机性，使得分布更均匀，从而提高对应数组存储下标位置的随机性&均匀性，最终减少Hash冲突，两次就够了，已经达到了高位低位同时参与运算的目的；

7）HashMap在JDK1.7和JDK1.8中有哪些不同?

答：

不同	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数：`inflateTable()`	直接集成到了扩容函数`resize()`中
hash值计算方式	扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算	扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算
存放数据的规则	无冲突时，存放数组；冲突时，存放链表	无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8：树化并存放红黑树
插入数据方式	头插法（先讲原位置的数据移到后1位，再插入数据到该位置）	尾插法（直接插入到链表尾部/红黑树）
扩容后存储位置的计算方式	全部按照原来方法进行计算（即hashCode ->> 扰动函数 ->> (h&length-1)）	按照扩容后的规律计算（即扩容后的位置=原位置 or 原位置 + 旧容量）

8）为什么HashMap中String、Integer这样的包装类适合作为K?

答：String、Integer等包装类的特性能够保证Hash值的不可更改性和计算准确性，能够有效的减少Hash碰撞的几率

1. 都是final类型，即不可变性，保证key的不可更改性，不会存在获取hash值不同的情况
2. 内部已重写了`equals()`、`hashCode()`等方法，遵守了HashMap内部的规范（不清楚可以去上面看看putValue的过程），不容易出现Hash值计算错误的情况；

面试官：如果我想要让自己的Object作为K应该怎么办呢?

答：重写`hashCode()`和`equals()`方法

1. **重写`hashCode()`是因为需要计算存储数据的存储位置**，需要注意不要试图从散列码计算中排除掉一个对象的关键部分来提高性能，这样虽然能更快但可能会导致更多的Hash碰撞；
2. **重写`equals()`方法**，需要遵守自反性、对称性、传递性、一致性以及对于任何非null的引用值x，x.equals(null)必须返回false的这几个特性，**目的是为了保证key在哈希表中的唯一性**；

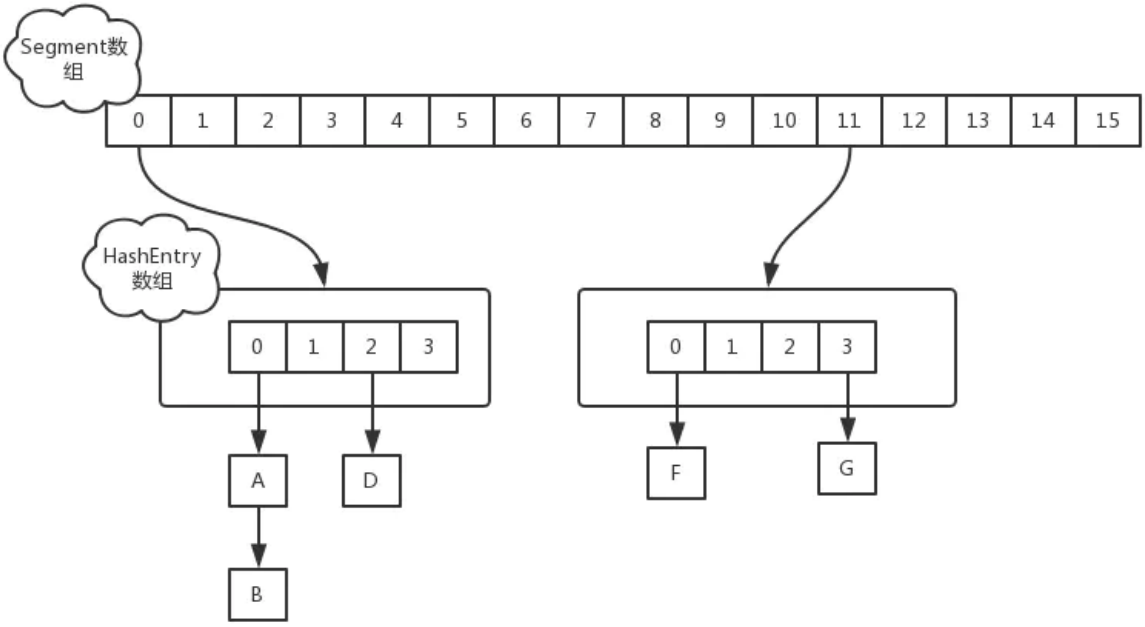
9）ConcurrentHashMap和Hashtable的区别?

答：ConcurrentHashMap 结合了 HashMap 和 HashTable 二者的优势。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。

面试官：ConcurrentHashMap的具体实现知道吗?

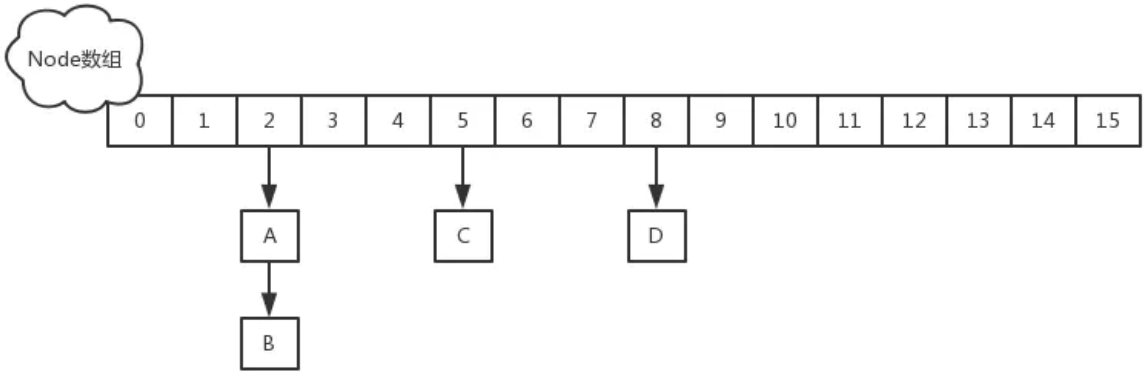
> 参考资料：http://www.importnew.com/23610.html

答：**在JDK1.7中，ConcurrentHashMap采用Segment + HashEntry的方式进行实现**，结构如下：



1. 该类包含两个静态内部类 HashEntry 和 Segment；前者用来封装映射表的键值对，后者用来充当锁的角色；
2. Segment 是一种可重入的锁 ReentrantLock，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：



插入元素过程（建议去看看源码）：

1. 如果相应位置的Node还没有初始化，则调用CAS插入相应的数据；

```
else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
    if (casTabAt(tab, i, null, new Node<K,V>(hash, key, value, null)))
        break;                   // no lock when adding to empty bin
}
```

1. 如果相应位置的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，如果该节点的hash不小于0，则遍历链表更新节点或插入新节点；

```
if (fh >= 0) {
    binCount = 1;
    for (Node<K,V> e = f;; ++binCount) {
```

```
            K ek;
            if (e.hash == hash &&
                ((ek = e.key) == key ||
                 (ek != null && key.equals(ek)))) {
                oldVal = e.val;
                if (!onlyIfAbsent)
                    e.val = value;
                break;
            }
            Node<K,V> pred = e;
            if ((e = e.next) == null) {
                pred.next = new Node<K,V>(hash, key, value, null);
                break;
            }
        }
    }
```

1. 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；
2. 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；

10）Java集合的快速失败机制 “fail-fast”?

答：

是java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素 的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生fail-fast机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 modCount 变量。集合在被遍历期间如果内容发生变化，就会改变modCount的值。每当迭代器使用hashNext()/next()遍历下一个元素之前，都会检测modCount变量是否为expectedmodCount值，是的话就返回遍历；否则抛出异常，终止遍历。

解决办法：

1. 在遍历过程中，所有涉及到改变modCount值得地方全部加上synchronized。

2. 使用CopyOnWriteArrayList来替换ArrayList

11）ArrayList 和 Vector 的区别?

答：

这两个类都实现了 List 接口（List 接口继承了 Collection 接口），他们都是有序集合，即存储在这两个集合中的元素位置都是有顺序的，相当于一种动态的数组，我们以后可以按位置索引来取出某个元素，并且其中的数据是允许重复的，这是与 HashSet 之类的集合的最大不同处，HashSet 之类的集合不可以按索引号去检索其中的元素，也不允许有重复的元素。

ArrayList 与 Vector 的区别主要包括两个方面：

1. 同步性：
   Vector 是线程安全的，也就是说它的方法之间是线程同步（加了synchronized 关键字）的，而 ArrayList 是线程不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用 ArrayList，因为它不考虑线程安全的问题，所以效率会高一些；如果有多个线程会访问到集合，那最好是使用 Vector，因为不需要我们自己再去考虑和编写线程安全的代码。
2. 数据增长：
   ArrayList 与 Vector 都有一个初始的容量大小，当存储进它们里面的元素的个人超过了容量时，就需要增加 ArrayList 和 Vector 的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要去的一定的平衡。Vector 在数据满时（加载因子1）增长为原来的两倍（扩容增量：原容量的 2 倍），而 ArrayList 在数据量达到容量的一半时（加载因子 0.5）增长为原容量的 (0.5 倍 + 1) 个空间。

12）ArrayList和LinkedList的区别?

答：

1. LinkedList 实现了 List 和 Deque 接口，一般称为双向链表；ArrayList 实现了 List 接口，动态数组；
2. LinkedList 在插入和删除数据时效率更高，ArrayList 在查找某个 index 的数据时效率更高；
3. LinkedList 比 ArrayList 需要更多的内存；

面试官：Array 和 ArrayList 有什么区别？什么时候该应 Array 而不是 ArrayList 呢?

答：它们的区别是：

1. Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。
2. Array 大小是固定的，ArrayList 的大小是动态变化的。
3. ArrayList 提供了更多的方法和特性，比如：addAll()，removeAll()，iterator() 等等。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

13）HashSet是如何保证数据不可重复的?

答：HashSet的底层其实就是HashMap，只不过我们**HashSet是实现了Set接口并且把数据作为K值，而V值一直使用一个相同的虚值来保存**，我们可以看到源码：

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;// 调用HashMap的put方法,PRESENT是一个至始至终都相同的虚值
}
```

由于HashMap的K值本身就不允许重复，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，那么在HashSet中执行这一句话始终会返回一个false，导致插入失败，这样就保证了数据的不可重复性；

14）BlockingQueue是什么?

答：

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。Java提供了集中BlockingQueue的实现，比如ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue等。

java集合详解和集合面试题目

转载自：https://blog.csdn.net/u010775025/article/details/79315361

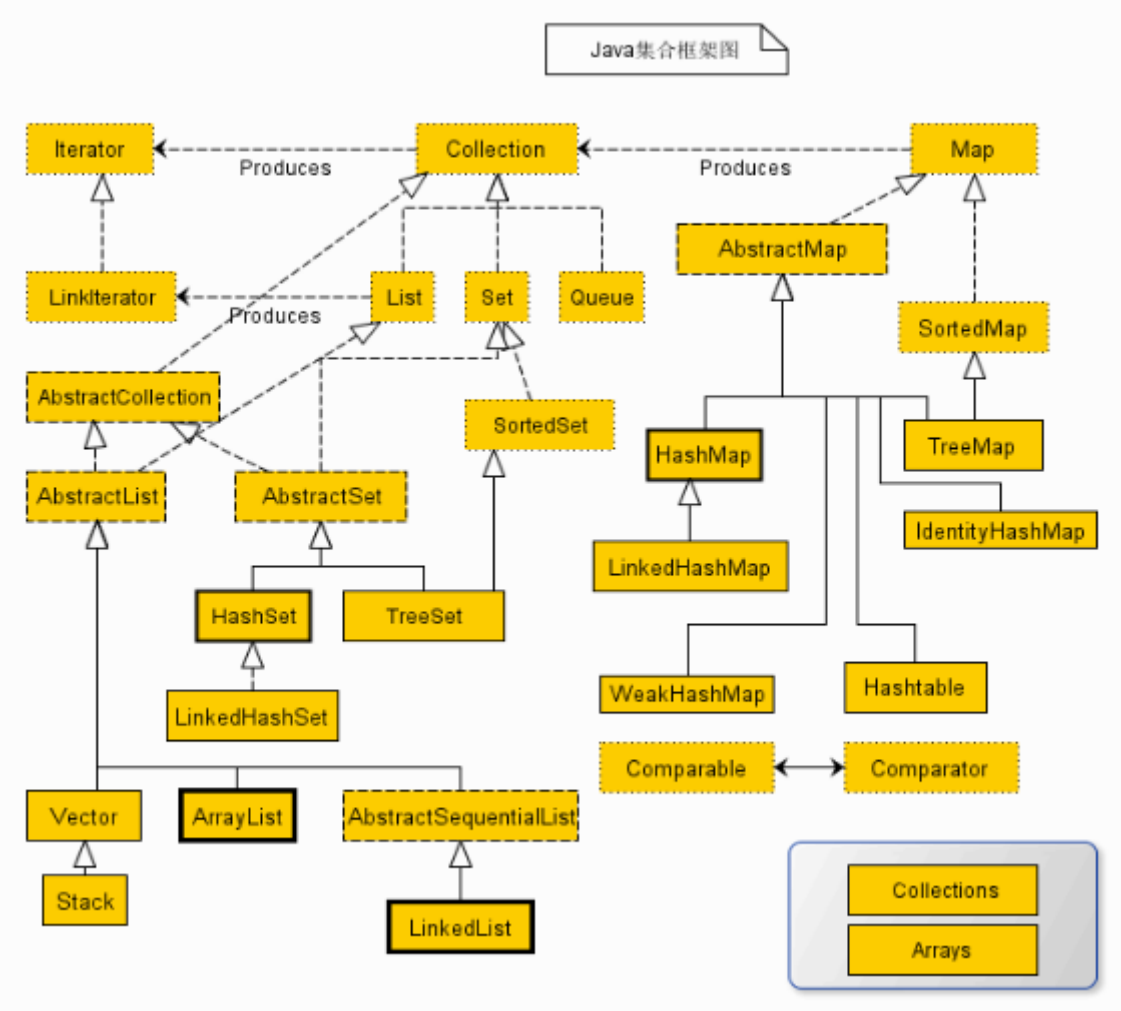
一、集合与数组

数组（可以存储基本数据类型）是用来存现对象的一种容器，但是数组的长度固定，不适合在对象数量未知的情况下使用。

集合（只能存储对象，对象类型可以不一样）的长度可变，可在多数情况下使用。

二、层次关系

如图所示：图中，实线边框的是实现类，折线边框的是抽象类，而点线边框的是接口



Collection接口是集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是Set和List。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

Map是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：
1.hasNext()是否还有下一个元素。
2.next()返回下一个元素。
3.remove()删除当前元素。

三、几种重要的接口和类简介

1、List（有序、可重复）
List里存放的对象是有序的，同时也是可以重复的，List关注的是索引，拥有一系列和索引相关的方法，查询速度快。因为往list集合里插入或删除数据时，会伴随着后面数据的移动，所有插入删除数据速度慢。

2、Set（无序、不能重复）
Set里存放的对象是无序，不能重复的，集合中的对象不按特定的方式排序，只是简单地把对象加入集合中。

3、Map（键值对、键唯一、值不唯一）
Map集合中存储的是键值对，键不能重复，值可以重复。根据键得到值，对map集合遍历时先得到键的set集合，对set集合进行遍历，得到相应的值。

对比如下：

		是否有序	**是否允许元素重复**
Collection			
List	是	是	
Set	AbstractSet	否	否
	HashSet		
	TreeSet	是（用二叉排序树）	
Map	AbstractMap	否	使用key-value来映射和存储数据，key必须唯一，value可以重复
	HashMap		
	TreeMap	是（用二叉排序树）	

四、遍历

在类集中提供了以下四种的常见输出方式：

1）Iterator：迭代输出，是使用最多的输出方式。

2）ListIterator：是Iterator的子接口，专门用于输出List中的内容。

3）foreach输出：JDK1.5之后提供的新功能，可以输出数组或集合。

4）for循环

代码示例如下：

for的形式：for（int i=0;i<arr.size();i++）{...}

foreach的形式： for（int　i：arr）{...}

iterator的形式：
Iterator it = arr.iterator();
while(it.hasNext()){ object o =it.next(); ...}

五、ArrayList和LinkedList

ArrayList和LinkedList在用法上没有区别，但是在功能上还是有区别的。LinkedList经常用在增删操作较多而查询操作很少的情况下，ArrayList则相反。

六、Map集合

实现类：HashMap、Hashtable、LinkedHashMap和TreeMap

HashMap

HashMap是最常用的Map，它根据键的HashCode值存储数据，根据键可以直接获取它的值，具有很快的访问速度，遍历时，取得数据的顺序是完全随机的。因为键对象不可以重复，所以HashMap最多只允许一条记录的键为Null，允许多条记录的值为Null，是非同步的

Hashtable

Hashtable与HashMap类似，是HashMap的线程安全版，它支持线程的同步，即任一时刻只有一个线程能写Hashtable，因此也导致了Hashtale在写入时会比较慢，它继承自Dictionary类，不同的是它不允许记录的键或者值为null，同时效率较低。

ConcurrentHashMap

线程安全，并且锁分离。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

LinkedHashMap

LinkedHashMap保存了记录的插入顺序，在用Iteraor遍历LinkedHashMap时，先得到的记录肯定是先插入的，在遍历的时候会比HashMap慢，有HashMap的全部特性。

TreeMap

TreeMap实现SortMap接口，能够把它保存的记录根据键排序，默认是按键值的升序排序（自然顺序），也可以指定排序的比较器，当用Iterator遍历TreeMap时，得到的记录是排过序的。不允许key值为空，非同步的；

map的遍历

第一种：KeySet()
将Map中所有的键存入到set集合中。因为set具备迭代器。所有可以迭代方式取出所有的键，再根据get方法。获取每一个键对应的值。 keySet():迭代后只能通过get()取key 。
取到的结果会乱序，是因为取得数据行主键的时候，使用了HashMap.keySet()方法，而这个方法返回的Set结果，里面的数据是乱序排放的。
典型用法如下：
Map map = new HashMap();

```
map.put("key1","lisi1");
map.put("key2","lisi2");
map.put("key3","lisi3");
map.put("key4","lisi4");
//先获取map集合的所有键的set集合，keyset（）
Iterator it = map.keySet().iterator();
 //获取迭代器
while(it.hasNext()){
Object key = it.next();
System.out.println(map.get(key));
}
```

第二种：entrySet（）
Set<Map.Entry<K,V>> entrySet() //返回此映射中包含的映射关系的 Set 视图。（一个关系就是一个键-值对），就是把(key-value)作为一个整体一对一对地存放到Set集合当中的。Map.Entry表示映射关系。
entrySet()：迭代后可以e.getKey()，e.getValue()两种方法来取key和value。返回的是Entry接口。
典型用法如下：

```
Map map = new HashMap();
map.put("key1","lisi1");
map.put("key2","lisi2");
map.put("key3","lisi3");
map.put("key4","lisi4");
//将map集合中的映射关系取出，存入到set集合
Iterator it = map.entrySet().iterator();
while(it.hasNext()){
Entry e =(Entry) it.next();
System.out.println("键"+e.getKey () + "的值为" + e.getValue());
}
```

推荐使用第二种方式，即entrySet()方法，效率较高。
对于keySet其实是遍历了2次，一次是转为iterator，一次就是从HashMap中取出key所对于的value。而entryset只是遍历了第一次，它把key和value都放到了entry中，所以快了。两种遍历的遍历时间相差还是很明显的。

七、主要实现类区别小结

Vector和ArrayList
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，vector增长率为目前数组长度的100%，而arraylist增长率为目前数组长度的50%。如果在集合中使用数据量比较大的数据，用vector有一定的优势。
3，如果查找一个指定位置的数据，vector和arraylist使用的时间是相同的，如果频繁的访问数据，这个时候使用vector和arraylist都可以。而如果移动一个指定位置会导致后面的元素都发生移动，这个时候就应该考虑到使用linklist,因为它移动一个指定位置的数据时其它元素不移动。
ArrayList 和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要涉及到数组元素移动等内存操作，所以索引数据快，插入数据慢，Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快。

arraylist和linkedlist
1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。 这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

HashMap与TreeMap
1、 HashMap通过hashcode对其内容进行快速查找，而TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap（HashMap中元素的排列顺序是不固定的）。
2、在Map 中插入、删除和定位元素，HashMap是最好的选择。但如果您要按自然顺序或自定义顺序遍历键，那么TreeMap会更好。使用HashMap要求添加的键类明确定义了hashCode()和 equals()的实现。
两个map中的元素一样，但顺序不一样，导致hashCode()不一样。
同样做测试：
在HashMap中，同样的值的map,顺序不同，equals时，false;
而在treeMap中，同样的值的map,顺序不同,equals时，true，说明，treeMap在equals()时是整理了顺序了的。

HashTable与HashMap
1、同步性:Hashtable是线程安全的，也就是说是同步的，而HashMap是线程序不安全的，不是同步的。
2、HashMap允许存在一个为null的key，多个为null的value 。

3、hashtable的key和value都不允许为null。

Java集合框架为Java编程语言的基础，也是Java面试中很重要的一个知识点。这里，我列出了一些关于Java集合的重要问题和答案。

1.Java集合框架是什么？说出一些集合框架的优点?

每种编程语言中都有集合，最初的Java版本包含几种集合类：Vector、Stack、HashTable和Array。随着集合的广泛使用，Java1.2提出了囊括所有集合接口、实现和算法的集合框架。在保证线程安全的情况下使用泛型和并发集合类，Java已经经历了很久。它还包括在Java并发包中，阻塞接口以及它们的实现。集合框架的部分优点如下：

（1）使用核心集合类降低开发成本，而非实现我们自己的集合类。

（2）随着使用经过严格测试的集合框架类，代码质量会得到提高。

（3）通过使用JDK附带的集合类，可以降低代码维护成本。

（4）复用性和可操作性。

2.集合框架中的泛型有什么优点?

Java1.5引入了泛型，所有的集合接口和实现都大量地使用它。泛型允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在编译时报错。这避免了在运行时出现ClassCastException，因为你将会在编译时得到报错信息。泛型也使得代码整洁，我们不需要使用显式转换和instanceOf操作符。它也给运行时带来好处，因为不会产生类型检查的字节码指令。

3.Java集合框架的基础接口有哪些?

Collection为集合层级的根接口。一个集合代表一组对象，这些对象即为它的元素。Java平台不提供这个接口任何直接的实现。

Set是一个不能包含重复元素的集合。这个接口对数学集合抽象进行建模，被用来代表集合，就如一副牌。

List是一个有序集合，可以包含重复元素。你可以通过它的索引来访问任何元素。List更像长度动态变换的数组。

Map是一个将key映射到value的对象.一个Map不能包含重复的key：每个key最多只能映射一个value。

一些其它的接口有Queue、Dequeue、SortedSet、SortedMap和ListIterator。

4.为何Collection不从Cloneable和Serializable接口继承?

Collection接口指定一组对象，对象即为它的元素。如何维护这些元素由Collection的具体实现决定。例如，一些如List的Collection实现允许重复的元素，而其它的如Set就不允许。很多Collection实现有一个公有的clone方法。然而，把它放到集合的所有实现中也是没有意义的。这是因为Collection是一个抽象表现。重要的是实现。

当与具体实现打交道的时候，克隆或序列化的语义和含义才发挥作用。所以，具体实现应该决定如何对它进行克隆或序列化，或它是否可以被克隆或序列化。

在所有的实现中授权克隆和序列化，最终导致更少的灵活性和更多的限制。特定的实现应该决定它是否可以被克隆和序列化。

5.为何Map接口不继承Collection接口?

尽管Map接口和它的实现也是集合框架的一部分，但Map不是集合，集合也不是Map。因此，Map继承Collection毫无意义，反之亦然。

如果Map继承Collection接口，那么元素去哪儿？Map包含key-value对，它提供抽取key或value列表集合的方法，但是它不适合“一组对象”规范。

6.Iterator是什么?

Iterator接口提供遍历任何Collection的接口。我们可以从一个Collection中使用迭代器方法来获取迭代器实例。迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者在迭代过程中移除元素。

7.Enumeration和Iterator接口的区别?

Enumeration的速度是Iterator的两倍，也使用更少的内存。Enumeration是非常基础的，也满足了基础的需要。但是，与Enumeration相比，Iterator更加安全，因为当一个集合正在被遍历的时候，它会阻止其它线程去修改集合。

迭代器取代了Java集合框架中的Enumeration。迭代器允许调用者从集合中移除元素，而Enumeration不能做到。为了使它的功能更加清晰，迭代器方法名已经经过改善。

8.为何没有像Iterator.add()这样的方法，向集合中添加元素?

语义不明，已知的是，Iterator的协议不能确保迭代的次序。然而要注意，ListIterator没有提供一个add操作，它要确保迭代的顺序。

9.为何迭代器没有一个方法可以直接获取下一个元素，而不需要移动游标?

它可以在当前Iterator的顶层实现，但是它用得很少，如果将它加到接口中，每个继承都要去实现它，这没有意义。

10.Iterater和ListIterator之间有什么区别?

（1）我们可以使用Iterator来遍历Set和List集合，而ListIterator只能遍历List。

（2）Iterator只可以向前遍历，而LIstIterator可以双向遍历。

（3）ListIterator从Iterator接口继承，然后添加了一些额外的功能，比如添加一个元素、替换一个元素、获取前面或后面元素的索引位置。

11.遍历一个List有哪些不同的方式?

```
List<String> strList = new ArrayList<>();


//使用for-each循环


for(String obj : strList){


  System.out.println(obj);


}


//using iterator


Iterator<String> it = strList.iterator();


while(it.hasNext()){


  String obj = it.next();


  System.out.println(obj);


}
```

使用迭代器更加线程安全，因为它可以确保，在当前遍历的集合元素被更改的时候，它会抛出ConcurrentModificationException。

12.通过迭代器fail-fast属性，你明白了什么?

每次我们尝试获取下一个元素的时候，Iterator fail-fast属性检查当前集合结构里的任何改动。如果发现任何改动，它抛出ConcurrentModificationException。Collection中所有Iterator的实现都是按fail-fast来设计的（ConcurrentHashMap和CopyOnWriteArrayList这类并发集合类除外）。

13.fail-fast与fail-safe有什么区别?

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail-fast的，而java.util.concurrent中的集合类都为fail-safe的。Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

14.在迭代一个集合的时候，如何避免ConcurrentModificationException?

在遍历一个集合的时候，我们可以使用并发集合类来避免ConcurrentModificationException，比如使用CopyOnWriteArrayList，而不是ArrayList。

15.为何Iterator接口没有具体的实现?

Iterator接口定义了遍历集合的方法，但它的实现则是集合实现类的责任。每个能够返回用于遍历的Iterator的集合类都有它自己的Iterator实现内部类。

这就允许集合类去选择迭代器是fail-fast还是fail-safe的。比如，ArrayList迭代器是fail-fast的，而CopyOnWriteArrayList迭代器是fail-safe的。

16.UnsupportedOperationException是什么?

UnsupportedOperationException是用于表明操作不支持的异常。在JDK类中已被大量运用，在集合框架java.util.Collections.UnmodifiableCollection将会在所有add和remove操作中抛出这个异常。

17.在Java中，HashMap是如何工作的?

HashMap在Map.Entry静态内部类实现中存储key-value对。HashMap使用哈希算法，在put和get方法中，它使用hashCode()和equals()方法。当我们通过传递key-value对调用put方法的时候，HashMap使用Key hashCode()和哈希算法来找出存储key-value对的索引。Entry存储在LinkedList中，所以如果存在entry，它使用equals()方法来检查传递的key是否已经存在，如果存在，它会覆盖value，如果不存在，它会创建一个新的entry然后保存。当我们通过传递key调用get方法时，它再次使用hashCode()来找到数组中的索引，然后使用equals()方法找出正确的Entry，然后返回它的值。下面的图片解释了详细内容。

其它关于HashMap比较重要的问题是容量、负荷系数和阀值调整。HashMap默认的初始容量是16，负荷系数是0.75。阀值是为负荷系数乘以容量，无论何时我们尝试添加一个entry，如果map的大小比阀值大的时候，HashMap会对map的内容进行重新哈希，且使用更大的容量。容量总是2的幂，所以如果你知道你需要存储大量的key-value对，比如缓存从数据库里面拉取的数据，使用正确的容量和负荷系数对HashMap进行初始化是个不错的做法。

18.hashCode()和equals()方法有何重要性?

HashMap使用Key对象的hashCode()和equals()方法去决定key-value对的索引。当我们试着从HashMap中获取值的时候，这些方法也会被用到。如果这些方法没有被正确地实现，在这种情况下，两个不同Key也许会产生相同的hashCode()和equals()输出，HashMap将会认为它们是相同的，然后覆盖它们，而非把它们存储到不同的地方。同样的，所有不允许存储重复数据的集合类都使用hashCode()和equals()去查找重复，所以正确实现它们非常重要。equals()和hashCode()的实现应该遵循以下规则：

（1）如果o1.equals(o2)，那么o1.hashCode() == o2.hashCode()总是为true的。

（2）如果o1.hashCode() == o2.hashCode()，并不意味着o1.equals(o2)会为true。

19.我们能否使用任何类作为Map的key?

我们可以使用任何类作为Map的key，然而在使用它们之前，需要考虑以下几点：

（1）如果类重写了equals()方法，它也应该重写hashCode()方法。

（2）类的所有实例需要遵循与equals()和hashCode()相关的规则。请参考之前提到的这些规则。

（3）如果一个类没有使用equals()，你不应该在hashCode()中使用它。

（4）用户自定义key类的最佳实践是使之为不可变的，这样，hashCode()值可以被缓存起来，拥有更好的性能。不可变的类也可以确保hashCode()和equals()在未来不会改变，这样就会解决与可变相关的问题了。

比如，我有一个类MyKey，在HashMap中使用它。

```
//传递给MyKey的name参数被用于equals()和hashCode()中


MyKey key = new MyKey('Pankaj'); //assume hashCode=1234


myHashMap.put(key, 'Value');


// 以下的代码会改变key的hashCode()和equals()值


key.setName('Amit'); //assume new hashCode=7890


//下面会返回null，因为HashMap会尝试查找存储同样索引的key，而key已被改变了，匹配失败，返回
null


myHashMap.get(new MyKey('Pankaj'));
```

那就是为何String和Integer被作为HashMap的key大量使用。

20.Map接口提供了哪些不同的集合视图?

Map接口提供三个集合视图：

（1）Set keyset()：返回map中包含的所有key的一个Set视图。集合是受map支持的，map的变化会在集合中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

（2）Collection values()：返回一个map中包含的所有value的一个Collection视图。这个collection受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个collection时，若map被修改了（除迭代器自身的移除操作以外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

（3）Set<Map.Entry<K,V>> entrySet()：返回一个map钟包含的所有映射的一个集合视图。这个集合受map支持的，map的变化会在collection中反映出来，反之亦然。当一个迭代器正在遍历一个集合时，若map被修改了（除迭代器自身的移除操作，以及对迭代器返回的entry进行setValue外），迭代器的结果会变为未定义。集合支持通过Iterator的Remove、Set.remove、removeAll、retainAll和clear操作进行元素移除，从map中移除对应的映射。它不支持add和addAll操作。

21.HashMap和HashTable有何不同?

（1）HashMap允许key和value为null，而HashTable不允许。

（2）HashTable是同步的，而HashMap不是。所以HashMap适合单线程环境，HashTable适合多线程环境。

（3）在Java1.4中引入了LinkedHashMap，HashMap的一个子类，假如你想要遍历顺序，你很容易从HashMap转向LinkedHashMap，但是HashTable不是这样的，它的顺序是不可预知的。

（4）HashMap提供对key的Set进行遍历，因此它是fail-fast的，但HashTable提供对key的Enumeration进行遍历，它不支持fail-fast。

（5）HashTable被认为是个遗留的类，如果你寻求在迭代的时候修改Map，你应该使用CocurrentHashMap。

22.如何决定选用HashMap还是TreeMap?

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

23.ArrayList和Vector有何异同点?

ArrayList和Vector在很多时候都很类似。

（1）两者都是基于索引的，内部由一个数组支持。

（2）两者维护插入的顺序，我们可以根据插入顺序来获取元素。

（3）ArrayList和Vector的迭代器实现都是fail-fast的。

（4）ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

（1）Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。

（2）ArrayList比Vector快，它因为有同步，不会过载。

（3）ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

24.Array和ArrayList有何区别？什么时候更适合用Array?

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是固定的。

Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。尽管ArrayList明显是更好的选择，但也有些时候Array比较好用。

（1）如果列表的大小已经指定，大部分情况下是存储和遍历它们。

（2）对于遍历基本数据类型，尽管Collections使用自动装箱来减轻编码任务，在指定大小的基本类型的列表上工作也会变得很慢。

（3）如果你要使用多维数组，使用[][]比List<List<>>更容易。

25.ArrayList和LinkedList有何区别?

ArrayList和LinkedList两者都实现了List接口，但是它们之间有些不同。

（1）ArrayList是由Array所支持的基于一个索引的数据结构，所以它提供对元素的随机访问，复杂度为O(1)，但LinkedList存储一系列的节点数据，每个节点都与前一个和下一个节点相连接。所以，尽管有使用索引获取元素的方法，内部实现是从起始点开始遍历，遍历到索引的节点然后返回元素，时间复杂度为O(n)，比ArrayList要慢。

（2）与ArrayList相比，在LinkedList中插入、添加和删除一个元素会更快，因为在一个元素被插入到中间的时候，不会涉及改变数组的大小，或更新索引。

（3）LinkedList比ArrayList消耗更多的内存，因为LinkedList中的每个节点存储了前后节点的引用。

26.哪些集合类提供对元素的随机访问?

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机访问。

27.EnumSet是什么?

java.util.EnumSet是使用枚举类型的集合实现。当集合创建时，枚举集合中的所有元素必须来自单个指定的枚举类型，可以是显示的或隐示的。EnumSet是不同步的，不允许值为null的元素。它也提供了一些有用的方法，比如copyOf(Collection c)、of(E first,E...rest)和complementOf(EnumSet s)。

28.哪些集合类是线程安全的?

Vector、HashTable、Properties和Stack是同步类，所以它们是线程安全的，可以在多线程环境下使用。Java1.5并发API包括一些集合类，允许迭代时修改，因为它们都工作在集合的克隆上，所以它们在多线程环境中是安全的。

29.并发集合类是什么?

Java1.5并发包（java.util.concurrent）包含线程安全集合类，允许在迭代时修改集合。迭代器被设计为fail-fast的，会抛出ConcurrentModificationException。一部分类为：CopyOnWriteArrayList、ConcurrentHashMap、CopyOnWriteArraySet。

30.BlockingQueue是什么?

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在BlockingQueue的实现类中被处理了。Java提供了集中BlockingQueue的实现，比如ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue等。

31.队列和栈是什么，列出它们的区别?

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque接口允许从两端检索元素。

栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口。

32.Collections类是什么?

Java.util.Collections是一个工具类仅包含静态方法，它们操作或返回集合。它包含操作集合的多态算法，返回一个由指定集合支持的新集合和其它一些内容。这个类包含集合框架算法的方法，比如折半搜索、排序、混编和逆序等。

33.Comparable和Comparator接口是什么?

如果我们想使用Array或Collection的排序方法时，需要在自定义类里实现Java提供Comparable接口。Comparable接口有compareTo(T OBJ)方法，它被排序方法所使用。我们应该重写这个方法，如果“this”对象比传递的对象参数更小、相等或更大时，它返回一个负整数、0或正整数。但是，在大多数实际情况下，我们想根据不同参数进行排序。比如，作为一个CEO，我想对雇员基于薪资进行排序，一个HR想基于年龄对他们进行排序。这就是我们需要使用Comparator接口的情景，因为Comparable.compareTo(Object o)方法实现只能基于一个字段进行排序，我们不能根据对象排序的需要选择字段。Comparator接口的compare(Object o1, Object o2)方法的实现需要传递两个对象参数，若第一个参数比第二个小，返回负整数；若第一个等于第二个，返回0；若第一个比第二个大，返回正整数。

34.Comparable和Comparator接口有何区别?

Comparable和Comparator接口被用来对对象集合或者数组进行排序。Comparable接口被用来提供对象的自然排序，我们可以使用它来提供基于单个逻辑的排序。

Comparator接口被用来提供不同的排序算法，我们可以选择需要使用的Comparator来对给定的对象集合进行排序。

35.我们如何对一组对象进行排序?

如果我们需要对一个对象数组进行排序，我们可以使用Arrays.sort()方法。如果我们需要排序一个对象列表，我们可以使用Collection.sort()方法。两个类都有用于自然排序（使用Comparable）或基于标准的排序（使用Comparator）的重载方法sort()。Collections内部使用数组排序方法，所有它们两者都有相同的性能，只是Collections需要花时间将列表转换为数组。

36.当一个集合被作为参数传递给一个函数时，如何才可以确保函数不能修改它?

在作为参数传递之前，我们可以使用Collections.unmodifiableCollection(Collection c)方法创建一个只读集合，这将确保改变集合的任何操作都会抛出UnsupportedOperationException。

37.我们如何从给定集合那里创建一个synchronized的集合?

我们可以使用Collections.synchronizedCollection(Collection c)根据指定集合来获取一个synchronized（线程安全的）集合。

38.集合框架里实现的通用算法有哪些?

Java集合框架提供常用的算法实现，比如排序和搜索。Collections类包含这些方法实现。大部分算法是操作List的，但一部分对所有类型的集合都是可用的。部分算法有排序、搜索、混编、最大最小值。

39.大写的O是什么？举几个例子?

大写的O描述的是，就数据结构中的一系列元素而言，一个算法的性能。Collection类就是实际的数据结构，我们通常基于时间、内存和性能，使用大写的O来选择集合实现。比如：例子1：ArrayList的get(index i)是一个常量时间操作，它不依赖list中元素的数量。所以它的性能是O(1)。例子2：一个对于数组或列表的线性搜索的性能是O(n)，因为我们需要遍历所有的元素来查找需要的元素。

40.与Java集合框架相关的有哪些最好的实践?

（1）根据需要选择正确的集合类型。比如，如果指定了大小，我们会选用Array而非ArrayList。如果我们想根据插入顺序遍历一个Map，我们需要使用TreeMap。如果我们不想重复，我们应该使用Set。

（2）一些集合类允许指定初始容量，所以如果我们能够估计到存储元素的数量，我们可以使用它，就避免了重新哈希或大小调整。

（3）基于接口编程，而非基于实现编程，它允许我们后来轻易地改变实现。

（4）总是使用类型安全的泛型，避免在运行时出现ClassCastException。

（5）使用JDK提供的不可变类作为Map的key，可以避免自己实现hashCode()和equals()。

（6）尽可能使用Collections工具类，或者获取只读、同步或空的集合，而非编写自己的实现。它将会提供代码重用性，它有着更好的稳定性和可维护性。

HashMap常见面试题