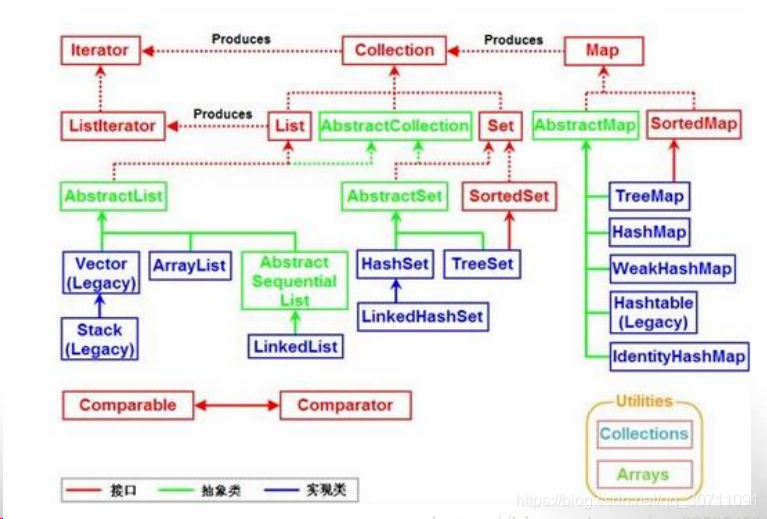
java集合类框架及常问到的问题（FAQ）



# 概述

集合类型主要有3种：set(集）、list(列表）和map(映射)。

集合接口分为：Collection和Map，list、set实现了Collection接口

# 接口

## List

可以重复，通过索引取出加入数据，顺序与插入顺序一致，可以含有null元素

### ArrayList

底层数据结构使数组结构array，查询速度快，增删改慢，因为是一种类似数组的形式进行存储，因此它的随机访问速度极快；

### Vector

底层是数组结构array，与ArrayList相同，查询速度快，增删改慢；

### LinkedList

底层使用链表结构，增删速度快，查询稍慢；

### ArrayList与Vector的区别

1.如果集合中的元素数量大于当前集合数组的长度时，Vector的增长率是目前数组长度的100%，而ArryaList增长率为目前数组长度的50%。所以，如果集合中使用数据量比较大的数据，用Vector有一定优势

2.线程同步ArrayList是线程不同步，所以Vector线程安全，但是因为每个方法都加上了synchronized，所以在效率上小于ArrayList

## Set

数据无序且唯一。

实现类都不是线程安全的类，解决方案：

Set set = Collections.sysnchronizedSet(Set对象);

### HashSet

HashSet是Set接口（Set接口是继承了Collection接口的）最常用的实现类，顾名思义，底层是用了哈希表（散列/hash）算法。其底层其实也是一个数组，存在的意义是提供查询速度，插入的速度也是比较快，但是适用于少量数据的插入操作，判断两个对象是否相等的规则：1、equals比较为true；2、hashCode值相同。要求：要求存在在哈希表中的对象元素都得覆盖equals和hashCode方法。

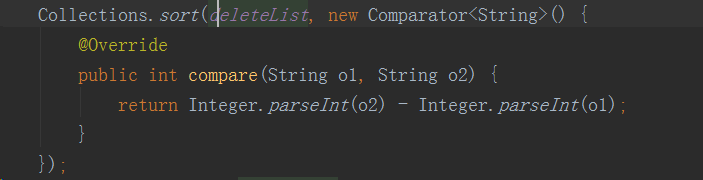
### LinkedHashSet

LinkedHashSet继承了HashSet类，所以它的底层用的也是哈希表的数据结构，但因为保持数据的先后添加顺序，所以又加了链表结构，但因为多加了一种数据结构，所以效率较低，不建议使用，如果要求一个集合急要保证元素不重复，也需要记录元素的先后添加顺序，才选择使用LinkedHashSet

### TreeSet

TreeSet是Set接口的实现类，也拥有set接口的一般特性，但是不同的是他也实现了SortSet接口，它底层采用的是红黑树算法（红黑树就是满足一下红黑性质的二叉搜索树：①每个节点是黑色或者红色②根节点是黑色的③每个叶子结点是黑色的④如果一个节点是红色的，那么他的两个子节点是黑色的⑤对每个节点，从该节点到其所有的后代叶子结点的简单路径上，仅包含相同数目的黑色结点，红黑树是许多“平衡”搜索树的一种，可以保证在最坏情况下的基本操作集合的时间复杂度为O(lgn)。普及：二叉搜索树的性质：它或者是一棵空树；或者是具有下列性质的二叉树：若左子树不空，则左子树上所有结点的值均小于它的根结点的值；若右子树不空，则右子树上所有结点的值均大于它的根结点的值；左、右子树也分别为二叉排序树。若子树为空，查找不成功。），要注意的是在TreeSet集合中只能存储相同类型对象的引用。

Tree最重要的就是它的两种排序方式：自然排序和客户端排序

自然排序；实现了Comparable接口，所以TreeSet可以调用对象的ComparableTo()方法来比较集合的大小，然后进行升序排序，这种排序方式叫做自然排序。其中实现了Comparable接口的还有BigDecimal、BigInteger、Byte、Double、Float、Integer、Long、Short（按照数字大小排序）、Character（按照Unicode值的数字大小进行排序）String（按照字符串中字符的Unicode值进行排序）类等。  
客户化排序：其实就是实现java.util.Comparator<Type>接口提供的具体的排序方式，<Type> 是具体要比较对象的类型，他有个compare的方法，如compare(x,y)返回值大于0表示x大于y，以此类推，当我们希望按照自己的想法排序的时候可以重写compare方法。  


## Map

java的Map(映射)是一种把键对象和值对象进行映射的集合，其中每一个元素都包含了键对象和值对象，其中值对象也可以是Map类型的数据，因此，Map支持多级映射，Map中的键是唯一的，但值可以不唯一，Map集合有两种实现，一种是利用哈希表来完成的叫做HashMap，它和HashSet都是利用哈希表来完成的，区别其实就是在哈希表的每个桶中，HashSet只有key，而HashMap在每个key上挂了一个value；另一种就是TreeMap，它实现了SortMap接口，也就是使用了红黑树的数据结构，和TreeSet一样也能实现自然排序和客户化排序两种排序方式，而哈希表不提供排序。

### HashMap

哈希表的实现原理中，先采用一个数组表示位桶，每个位桶的实现在1.8之前都是使用链表，但当每个位桶的数据较多的时候，链表查询的效率就会不高，因此在1.8之后，当位桶的数据超过阈值（8）的时候，就会采用红黑树来存储该位桶的数据（在阈值之前还是使用链表来进行存储），所以，哈希表的实现包括数组+链表+红黑树，在使用哈希表的集合中我们都认为他们的增删改查操作的时间复杂度都是O(1)的，不过常数项很大，因为哈希函数在进行计算的代价比较高,HashMap和Hashtable类似，不同之处在于HashMap是非同步的，并且允许null，即null value和null key。，但是将HashMap视为Collection时（values()方法可返回Collection），其迭代子操作时间开销和HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将HashMap的初始化容量设得过高，或者load factor过低。

### TreeMap

TreeMap 是一个有序的key-value集合，它是通过红黑树实现的。TreeMap 继承于AbstractMap，所以它是一个Map，即一个key-value集合。TreeMap 实现了NavigableMap接口，意味着它支持一系列的导航方法。比如返回有序的key集合。TreeMap 实现了Cloneable接口，意味着它能被克隆。TreeMap 实现了java.io.Serializable接口，意味着它支持序列化。

TreeMap基于红黑树（Red-Black tree）实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。TreeMap的基本操作 containsKey、get、put 和 remove 的时间复杂度是 log(n) 。另外，TreeMap是非同步的。 它的iterator 方法返回的迭代器是fail-fastl的。

### Hashtable

Hashtable继承Map接口，实现一个key-value映射的哈希表。任何非空（non-null）的对象都可作为key或者value，线程安全。

## 对比图



# FAQ

## 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；

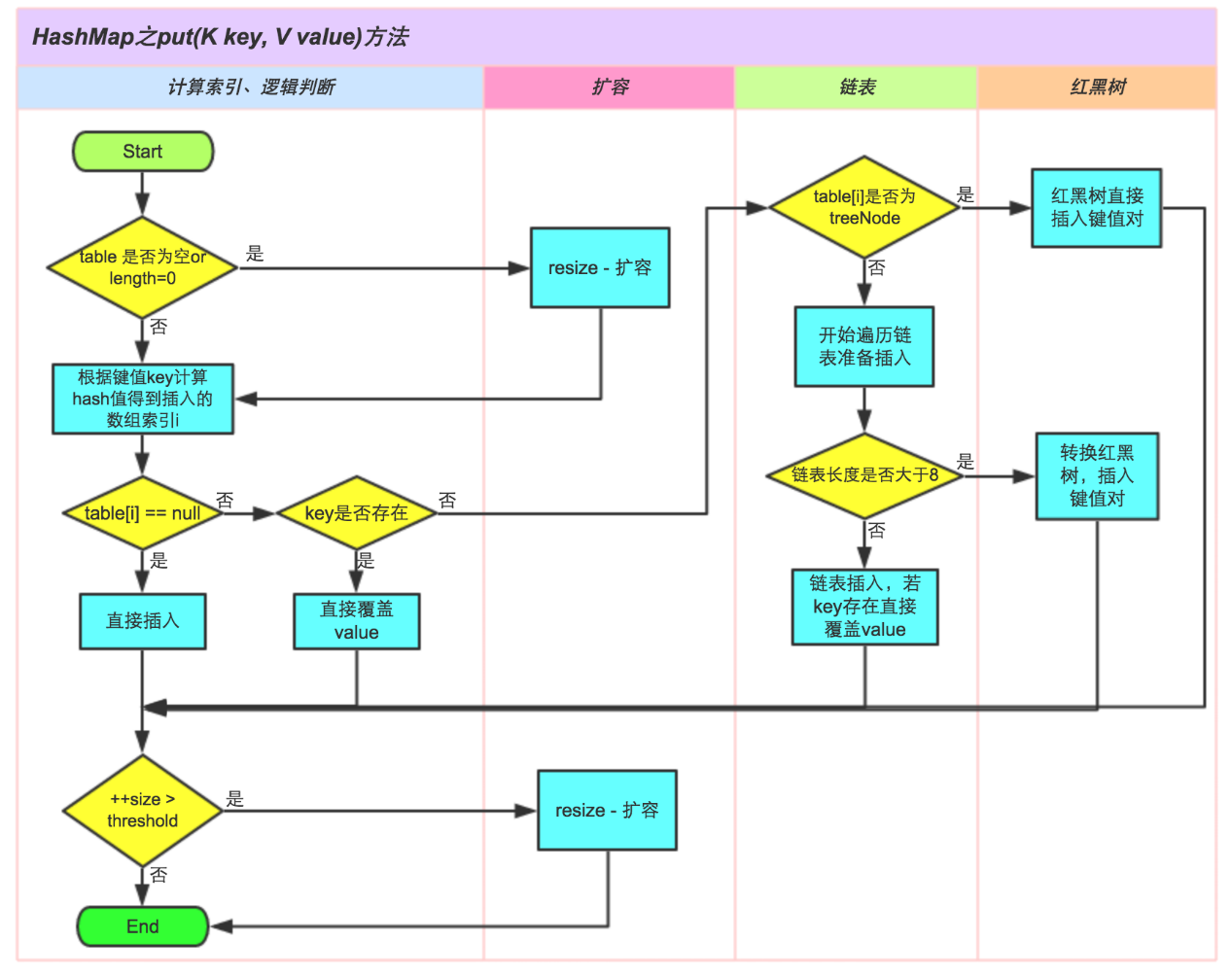
## 3）HashMap的put方法的具体流程？

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;  } |

用图片简单总结：



## 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。