# 一. HashMap

https://blog.csdn.net/jiary5201314/article/details/51439982

## 1. hashMap的原理

hashmap 是数组和链表的结合体，数组每个元素存的是链表的头结点 往 hashmap  
里面放键值对的时候先得到 key 的 hashcode，然后重新计算 hashcode， （让 1 分布均匀因为如果分布不均匀，低位全是 0，则后来计算数组下标的时候会 冲突），然后与 length-1 按位与，计算数组出数组下标 如果该下标对应的链表为空，则直接把键值对作为链表头结点，如果不为空，则 遍历链表看是否有 key 值相同的，有就把 value 替换， 没有就把该对象最为链表的第 一个节点，原有的节点最为他的后续节点

## 2. hashcode的计算

https://www.zhihu.com/question/20733617/answer/111577937  
https://blog.csdn.net/justloveyou\_/article/details/62893086  
Key.hashcode是key的自带的hascode函数是一个int值32位

hashmap jdk1.8 中 hashmap 重计算 hashcode 方法改动： 高 16 位异或低 16 位

return (key == null) ? 0 : h = key.hashCode() ^ (h >>> 16);  
首先确认：当 length 总是 2 的n 次方时， h & (length - 1) 等价于 hash 对length 取模 ， 但是&比%具有更高的效率；  
Jdk1.7 之前：h & (length - 1);//第三步，取模运算

![](data:text/html; charset=utf-8;base64,)

## 3. hashMap参数以及扩容机制

初始容量 16，达到阀值扩容，阀值等于最大容量\*负载因子，扩容每次 2 倍，总是 2 的n 次方  
扩容机制：  
使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有 Entry 数组的元素拷贝到新的 Entry 数组里，Java1.重新计算每个元素在数组中的位置。Java1.8 中不是重新计算，而是用了一种更巧妙的方式。

## 4. get()方法

**整个过程都不需要加锁**  
![fig:](data:text/html; charset=utf-8;base64,)

## 5. put()方法

![fig:](data:text/html; charset=utf-8;base64,)  
这里HashMap里面用到链式数据结构的一个概念。上面我们提到过Entry类里面有一个next属性，作用是指向下一个Entry。打个比方， 第一个键值对A进来，通过计算其key的hash得到的index=0，记做:Entry[0] = A。一会后又进来一个键值对B，通过计算其index也等于0，现在怎么办？HashMap会这样做:B.next = A,Entry[0] = B,如果又进来C,index也等于0,那么C.next = B,Entry[0] = C；这样我们发现index=0的地方其实存取了A,B,C三个键值对,他们通过next这个属性链接在一起。所以疑问不用担心。也就是说数组中存储的是最后插入的元素。到这里为止，HashMap的大致实现

## 6. HashMap问题 jdk1.8优化

（1）HashMap如果有很多相同key，后面的链很长的话，你会怎么优化？或者你会用什么数据结构来存储？针对HashMap中某个Entry链太长，查找的时间复杂度可能达到O(n)，怎么优化？  
Java8 做的改变：

* HashMap 是数组+链表+红黑树（JDK1.8 增加了红黑树部分），当链表长度>=8 时转化为红黑树  
  在 JDK1.8 版本中，对数据结构做了进一步的优化，引入了红黑树。而当链表长度太长（默认超过 8）时，链表就转换为红黑树，利用红黑树快速增删改查的特点提高 HashMap 的性能，其中会用到红黑树的插入、删除、查找等算法。
* java8 中对 hashmap 扩容不是重新计算所有元素在数组的位置，而是我们使用的是 2 次幂的扩展(指长度扩为原来 2 倍)，所以，元素的位置要么是在原位置，要么是在原位置再移动 2 次幂的位置在扩充 HashMap 的时候，不需要像 JDK1.7 的实现那样重新计算 hash， 只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了，是 0 的话索引没变，是 1 的话索引变成“原索引+oldCap”。

## 7. 一些面试题

### 7.1 HashMap 和 TreeMap 的区别

Hashmap 使用的是数组+链表，treemap 是红黑树

### 7.2 hashmap 为什么可以插入空值？

HashMap 中添加 key==null 的 Entry 时会调用 putForNullKey 方法直接去遍历  
table[0]Entry 链表，寻找 e.key==null 的 Entry 或者没有找到遍历结束

如果找到了 e.key==null，就保存 null 值对应的原值 oldValue，然后覆盖原值，并返回oldValue  
如果在 table[0]Entry 链表中没有找到就调用addEntry 方法添加一个 key 为 null 的 Entry

### 7.3 Hashmap 为什么线程不安全：(hash 碰撞和扩容导致)

* HashMap 底层是一个Entry 数组，当发生 hash 冲突的时候，hashmap 是采用链表的方式来解决的，在对应的数组位置存放链表的头结点。对链表而言，新加入的节点会从头结点加入。假如 A 线程和B 线程同时对同一个数组位置调用 addEntry，两个线程会同时得到现在的头结点，然后 A 写入新的头结点之后，B 也写入新的头结点，那 B 的写入操作就会覆盖A 的写入操作造成A 的写入操作丢失
* 删除键值对的代码如上：当多个线程同时操作同一个数组位置的时候，也都会先取得现在状态下该位置存储的头结点，然后各自去进行计算操作，之后再把结果写会到该数组位置去， 其实写回的时候可能其他的线程已经就把这个位置给修改过了，就会覆盖其他线程的修改
* 当多个线程同时检测到总数量超过门限值的时候就会同时调用 resize 操作，各自生成新的数组并 rehash 后赋给该map 底层的数组 table，结果最终只有最后一个线程生成的新数组被赋给 table 变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的 table 作为原始数组，这样也会有问题。

### 7.4 Hashmap 碰撞严重

可以自定义重写 hash——多 hash 函数

### 7.4 HashMap 高并发情况下会出现什么问题？

扩容问题

### 7.5 HashMap 的存放自定义类时，需要实现自定义类的什么方法？

答：hashCode 和equals。通过 hash(hashCode)然后模运算（其实是与的位操作）定位在Entry 数组中的下标，然后遍历这之后的链表，通过 equals 比较有没有相同的 key，如果有直接覆盖 value，如果没有就重新创建一个 Entry。

### 7.6 Hashmap为什么线程不安全

hash碰撞和扩容导致,HashMap扩容的的时候可能会形成环形链表，造成死循环。

### 7.7 Hashmap中的key可以为任意对象或数据类型吗？

可以为null但不能是可变对象,如果是可变对象的话,对象中的属性改变,则对象 HashCode也进行相应的改变,导致下次无法查找到己存在Map中的效据  
如果可变对象在 HashMap中被用作键,时就要小心在改变对象状态的时候,不要改变它的哈希值了。我们只需要保证成员变量的改变能保证该对象的哈希值不变即可.

# 二. CurrentHashMap

http://www.importnew.com/21781.html  
https://blog.csdn.net/dingji\_ping/article/details/51005799  
https://www.cnblogs.com/chengxiao/p/6842045.html  
http://ifeve.com/hashmap-concurrenthashmap-%E7%9B%B8%E4%BF%A1%E7%9C%8B%E5%AE%8C%E8%BF%99%E7%AF%87%E6%B2%A1%E4%BA%BA%E8%83%BD%E9%9A%BE%E4%BD%8F%E4%BD%A0%EF%BC%81/

## 1. 概述

![fig:](data:text/html; charset=utf-8;base64,)  
一个ConcurrentHashMap维护一个Segment数组，一个Segment维护一个HashEntry数组。

## 2. JDK1.7 ConCurrentHashMap原理

其中 Segment 继承于 ReentrantLock  
ConcurrentHashMap 使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问， 能够实现真正的并发访问。

![](data:text/html; charset=utf-8;base64,)

Segment继承了ReentrantLock，表明每个segment都可以当做一个锁。这样对每个segment中的数据需要同步操作的话都是使用每个segment容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的segment。  
![fig:](data:text/html; charset=utf-8;base64,)

## 3. JDK1.7 Get

![](data:text/html; charset=utf-8;base64,)

* CurrentHashMap是否使用了锁？？？

它也没有使用锁来同步，只是判断获取的entry的value是否为null，为null时才使用加锁的方式再次去获取。 这里可以看出并没有使用锁，但是value的值为null时候才是使用了加锁！！！

* Get原理：  
  第一步，先判断一下 count != 0；count变量表示segment中存在entry的个数。如果为0就不用找了。假设这个时候恰好另一个线程put或者remove了这个segment中的一个entry，会不会导致两个线程看到的count值不一致呢？看一下count变量的定义： transient volatile int count;  
  它使用了volatile来修改。我们前文说过，Java5之后，JMM实现了对volatile的保证：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。所以，每次判断count变量的时候，即使恰好其他线程改变了segment也会体现出来  
  ![fig:](data:text/html; charset=utf-8;base64,)

### 3.1 在get代码的①和②之间，另一个线程新增了一个entry

如果另一个线程新增的这个entry又恰好是我们要get的，这事儿就比较微妙了。下图大致描述了put 一个新的entry的过程。

![](data:text/html; charset=utf-8;base64,)

因为每个HashEntry中的next也是final的，没法对链表最后一个元素增加一个后续entry所以新增一个entry的实现方式只能通过头结点来插入了。newEntry对象是通过 new HashEntry(K k , V v, HashEntry next) 来创建的。如果另一个线程刚好new 这个对象时，当前线程来get它。因为没有同步，就可能会出现当前线程得到的newEntry对象是一个没有完全构造好的对象引用。 如果在这个new的对象的后面，则完全不影响，如果刚好是这个new的对象，那么当刚好这个对象没有完全构造好，也就是说这个对象的value值为null,就出现了如下所示的代码，需要重新加锁再次读取这个值！  
![fig:](data:text/html; charset=utf-8;base64,)

### 3.2 在get代码的①和②之间，另一个线程修改了一个entry的value

value是用volitale修饰的，可以保证读取时获取到的是修改后的值。

### 3.3 在get代码的①之后，另一个线程删除了一个entry

假设我们的链表元素是：e1-> e2 -> e3 -> e4 我们要删除 e3这个entry，因为HashEntry中next的不可变，所以我们无法直接把e2的next指向e4，而是将要删除的节点之前的节点复制一份，形成新的链表。它的实现大致如下图所示：  
![fig:](data:text/html; charset=utf-8;base64,)  
如果我们get的也恰巧是e3，可能我们顺着链表刚找到e1，这时另一个线程就执行了删除e3的操作，而我们线程还会继续沿着旧的链表找到e3返回。这里没有办法实时保证了，也就是说没办法看到最新的。  
我们第①处就判断了count变量，它保障了在 ①处能看到其他线程修改后的。①之后到②之间，如果再次发生了其他线程再删除了entry节点，就没法保证看到最新的了，这时候的get的实际上是未更新过的！！！。  
不过这也没什么关系，即使我们返回e3的时候，它被其他线程删除了，暴漏出去的e3也不会对我们新的链表造成影响。

### 4. JDK1.7 PUT

* 1.将当前 Segment 中的 table 通过 key 的 hashcode 定位到 HashEntry。
* 2.遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value。
* 3.不为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容。
* 4.最后会解除在 1 中所获取当前 Segment 的锁。
* 5.可以说是首先找到segment，确定是哪一个segment,然后在这个segment中遍历查找 key值是要查找的key值得entry,如果找到，那么就修改该key,如果没找到，那么就在头部新加一个entry.  
  ![fig:](data:text/html; charset=utf-8;base64,)  
  ![fig:](data:text/html; charset=utf-8;base64,)  
  ![fig:](data:text/html; charset=utf-8;base64,)

### 5. JDK1.7 Remove

![fig:](data:text/html; charset=utf-8;base64,)  
![fig:](data:text/html; charset=utf-8;base64,)  
![fig:](data:text/html; charset=utf-8;base64,)

### 6. JDK1.7 & JDK1.8 size()

![](data:text/html; charset=utf-8;base64,)

public int size() {  
 long n = sumCount();  
 return ((n < 0L) ? 0 :  
 (n > (long)Integer.MAX\_VALUE) ? Integer.MAX\_VALUE :  
 (int)n);  
}

volatile 保证内存可见，最大是65535.

### 5.JDK 1.8 CurrentHashMap概述

![fig:](data:text/html; charset=utf-8;base64,)  
1.其中抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。  
2.大于8的时候才去红黑树链表转红黑树的阀值，当table[i]下面的链表长度大于8时就转化为红黑树结构。

### 6. JDK1.8 put

![](data:text/html; charset=utf-8;base64,)

* 根据 key 计算出 hashcode 。
* 判断是否需要进行初始化。
* f即为当前 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
* 如果当前位置的 hashcode == MOVED == -1,则需要进行扩容。
* 如果都不满足，则利用 synchronized 锁写入数据(分为链表写入和红黑树写入）。
* 如果数量大于 TREEIFY\_THRESHOLD 则要转换为红黑树。

### 7. JDK1.8 get方法

![](data:text/html; charset=utf-8;base64,)

* 根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值。
* 如果是红黑树那就按照树的方式获取值。
* 就不满足那就按照链表的方式遍历获取值。

### 8.rehash过程

* Redis rehash ：dictRehash每次增量rehash n个元素，由于在自动调整大小时已设置好了ht[1]的大小，因此rehash的主要过程就是遍历ht[0]，取得key，然后将该key按ht[1]的 桶的大小重新rehash，并在rehash完后将ht[0]指向ht[1],然后将ht[0]清空。在这个过程中rehashidx非常重要，它表示上次rehash时在ht[0]的下标位置。
* 可以看到，redis对dict的rehash是分批进行的，这样不会阻塞请求，设计的比较优雅。
* 但是在调用dictFind的时候，可能需要对两张dict表做查询。唯一的优化判断是，当key在ht[0]不存在且不在rehashing状态时，可以速度返回空。如果在rehashing状态，当在ht[0]没值的时候，还需要在ht[1]里查找。
* dictAdd的时候，如果状态是rehashing，则把值插入到ht[1]，否则ht[0]

# 三 Hashtable

https://blog.csdn.net/ns\_code/article/details/36191279

## 1.参数

-（1）table是一个Entry[]数组类型，而Entry实际上就是一个单向链表。哈希表的"key-value键值对"都是存储在Entry数组中的。  
-（2）count是Hashtable的大小，它是Hashtable保存的键值对的数量。

* （3）threshold是Hashtable的阈值，用于判断是否需要调整Hashtable的容量。threshold的值="容量\*加载因子"。
* （4）loadFactor就是加载因子。
* （5）modCount是用来实现fail-fast机制的

## 1.put

从下面的代码中我们可以看出，Hashtable中的key和value是不允许为空的，当我们想要想Hashtable中添加元素的时候，首先计算key的hash值，然  
后通过hash值确定在table数组中的索引位置，最后将value值替换或者插入新的元素，如果容器的数量达到阈值，就会进行扩充。  
![fig:](data:text/html; charset=utf-8;base64,)

## 2.get

![](data:text/html; charset=utf-8;base64,)

## 3.Remove

在下面代码中，如果prev为null了，那么说明第一个元素就是要删除的元素，那么就直接指向第一个元素的下一个即可。

![](data:text/html; charset=utf-8;base64,)

## 4.扩容

* 默认初始容量为11
* 线程安全，但是速度慢，不允许key/value为null
* 加载因子为0.75：即当 元素个数 超过 容量长度的0.75倍 时，进行扩容
* 扩容增量：2\*原数组长度+1如 HashTable的容量为11，一次扩容后是容量为23

# 四. 一些面试题

## 4.1 hashtable和hashmap的区别

![fig:](data:text/html; charset=utf-8;base64,)

## 4.2 HashMap和ConCurrentHashMap区别

![](data:text/html; charset=utf-8;base64,)

## 4.3 ConcurrentHashMap和HashTable区别

* ConcurrentHashMap仅仅锁定map的某个部分，而Hashtable则会锁定整个map。
* hashtable(同一把锁):使用synchronized来保证线程安全，但效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用put添加元素，另一个线程不能使用put添加元素，也不能使用get，竞争会越来越激烈效率越低。
* concurrenthashmap(分段锁):(锁分段技术)每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。
* concurrenthashmap是由Segment数组结构和HahEntry数组结构组成。Segment是一种可重入锁ReentrantLock，扮演锁的角色。HashEntry用于存储键值对数据。一个concurrenthashmap里包含一个Segment数组。Segment的结构和Hashmap类似，是一种数组和链表结构，一个Segment包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得对应的Segment。

## 4.4 linkedHashMap

https://blog.csdn.net/justloveyou\_/article/details/71713781  
![fig:](data:text/html; charset=utf-8;base64,)

## 4.5 Linkedhashmap与 hashmap的区别

* 1.LinkedHashMap是HashMap的子类  
  -2.LinkedHashMap中的Entry增加了两个指针 before 和 after，它们分别用于维护双向链接列表。
* 3.在put操作上，虽然LinkedHashMap完全继承了HashMap的put操作，但是在细节上还是做了一定的调整，比如，在LinkedHashMap中向哈希表中插入新Entry的同时，还会通过Entry的addBefore方法将其链入到双向链表中。
* 4.在扩容操作上，虽然LinkedHashMap完全继承了HashMap的resize操作，但是鉴于性能和LinkedHashMap自身特点的考量，LinkedHashMap对其中的重哈希过程(transfer方法)进行了重写
* 5.在读取操作上，LinkedHashMap中重写了HashMap中的get方法，通过HashMap中的getEntry方法获取Entry对象。在此基础上，进一步获取指定键对应的值。

## 4.6 HashSet

对于HashSet而言，它是基于HashMap实现的  
Hashset源码 http://zhangshixi.iteye.com/blog/673143

**Hashset 如何保证集合的没有重复元素？**  
可以看出hashset底层是hashmap但是存储的是一个对象，hashset实际将该元素e作为key放入hashmap,当key值(该元素e)相同时，只是进行更新value，并不会新增加，所以set中的元素不会进行改变。  
![fig:](data:text/html; charset=utf-8;base64,)   
![fig:](data:text/html; charset=utf-8;base64,)

## 4.7 hashmap与hashset区别

![](data:text/html; charset=utf-8;base64,)

## 4.8 Collections.sort 内部原理

![fig:](data:text/html; charset=utf-8;base64,)  
重写 Collections.sort()

import java.util.\*;  
class xd{  
 int a;  
 int b;  
 xd(int a,int b){  
 this.a = a;  
 this.b = b;  
 }  
}  
public class Main {  
 public static void main(String[] arg) {  
 xd a = new xd(2,3);  
 xd b = new xd(4,1);  
 xd c = new xd(1,2);  
 ArrayList<xd> array = new ArrayList<>();  
 array.add(a);  
 array.add(b);  
 array.add(c);  
 Collections.sort(array, new Comparator<xd>() {  
 @Override  
 public int compare(xd o1, xd o2) {  
 if(o1.a > o2.a)  
 return 1;  
 else if(o1.a < o2.a)  
 return -1;  
 return 0;  
 }  
 });  
 for(int i=0;i<array.size();i++)  
 System.out.println(array.get(i).a);  
 for(int i=0;i<array.size();i++)  
 System.out.println(array.get(i).b);  
 }  
}

## 4.9 hash算法

![](data:text/html; charset=utf-8;base64,)

# 五.ArrayList，LinkedList和Vector的区别和实现原理

Vector :   
https://blog.csdn.net/chenssy/article/details/37520981

## 1.ArrayList与LinkedList区别

ArrayList和LinkedList都实现了List接口，他们有以下的不同点：  
ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。  
相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。  
LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

## 2. Vetor arraylist Linkedlist 区别

### ArrayList#

**ArrayList## 就是动态数组**，是Array的复杂版本，动态的增加和减少元素.当更多的元素加入到ArrayList中时,其大小将会动态地增长。它的元素可以通过get/set方法直接访问，因为ArrayList本质上是一个数组。初始容量为10。

* 1.插入元素的时候可能扩容，删除元素时不会缩小容量。
* 2.扩容增长为Arraylist增长原来的0.5倍
  1. 而Arraylist 没有设置增长空间的方法。
* 4.线程不同步

### Vector

**Vector** 和ArrayList类似, 区别在于Vector是同步类(synchronized).因此,开销就比ArrayList要大。初始容量为10。实现了随机访问接口，可以随机访问。Vector是内部是以动态数组的形式来存储数据的。

* 1.Vector还可以设置增长的空间大小，
  1. 及Vector增长原来的1倍3.vector 线程同步

### LinkedList

LinkedList 是一个双链表,在添加和删除元素时具有比ArrayList更好的性能.但在get与set方面弱于ArrayList.当然,这些对比都是指数据量很大或者操作很频繁的情况下的对比。它还实现了 Queue 接口,该接口比List提供了更多的方法,包括 offer(),peek(),poll()等.

ArrayList和LinkedList的使用场景，其中add方法的实现ArrayList,LinkedList的实现以及插入，查找，删除的过程

## 3.使用ArrayList的迭代器会出现什么问题？

单线程和多线程环境下；  
常用的迭代器设计模式，iterator方法返回一个父类实现的迭代器。

* 1、迭代器的hasNext方法的作用是判断当前位置是否是数组最后一个位置，相等为false，否则为true。
* 2、迭代器next方法用于返回当前的元素，并把指针指向下一个元素，值得注意的是，每次使用next方法的时候，都会判断创建迭代器获取的这个容器的计数器modCount是否与此时的不相等，不相等说明集合的大小被修改过，如果是会抛出ConcurrentModificationException异常，如果相等调用get方法返回元素即可。

## 4. 数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？

答：不同点：定义上：Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。容量上：Array大小固定，ArrayList的大小是动态变化的。操作上：ArrayList提供更多的方法和特性，如：addAll()，removeAll()，iterator()等等。使用基本数据类型或者知道数据元素数量的时候可以考虑Array;ArrayList处理固定数量的基本类型数据类型时会自动装箱来减少编码工作量，但是相对较慢。

## 5.ArrayList和Vector有何异同点？

### 相同点：

* （1）两者都是基于索引的，都是基于数组的。
* （2）两者都维护插入顺序，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。

### 不同点：

* （1）Vector是同步，线程安全，而ArrayList非同步，线程不安全。对于ArrayList，如果迭代时改变列表，应该使用CopyOnWriteArrayList。
* （2）但是，ArrayList比Vector要快，它因为有同步，不会过载。
* （3）在使用上，ArrayList更加通用，因为Collections工具类容易获取同步列表和只读列表。

## 6. 快速失败(fail-fast)和安全失败(fail-safe)

### 快速失败（fail—fast）

* 在用迭代器遍历一个集合对象时，如果遍历过程中对集合对象的内容进行了修改（增加、删除、修改），则会抛出Concurrent Modification Exception。
* 原理：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。
* 注意：这里异常的抛出条件是检测到 modCount！=expectedmodCount 这个条件。如果集合发生变化时修改modCount值刚好又设置为了expectedmodCount值，则异常不会抛出。因此，不能依赖于这个异常是否抛出而进行并发操作的编程，这个异常只建议用于检测并发修改的bug。
* 场景：java.util包下的集合类都是快速失败的，不能在多线程下发生并发修改（迭代过程中被修改）。

### 安全失败（fail—safe）

* 采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。
* 原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发Concurrent Modification Exception。
* 缺点：基于拷贝内容的优点是避免了Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。
* 场景：java.util.concurrent包下的容器都是安全失败，可以在多线程下并发使用，并发修改。

快速失败和安全失败是对迭代器而言的。 快速失败：当在迭代一个集合的时候，如果有另外一个线程在修改这个集合，就会抛出ConcurrentModification异常，java.util下都是快速失败。 安全失败：在迭代时候会在集合二层做一个拷贝，所以在修改集合上层元素不会影响下层。在java.util.concurrent下都是安全失败

### Iterator和ListIterator的区别是什么？

答：Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。  
ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

### 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

答：Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败的，而java.util.concurrent包下面的所有的类都是安全失败的。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。

### Enumeration接口和Iterator接口的区别有哪些？

答：Enumeration速度是Iterator的2倍，同时占用更少的内存。但是，Iterator远远比Enumeration安全，因为其他线程不能够修改正在被iterator遍历的集合里面的对象。同时，Iterator允许调用者删除底层集合里面的元素，这对Enumeration来说是不可能的。