链滴

集合字符串整合

作者：Gao-Eason

原文链接：https://ld246.com/article/1649609661291

来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20210515.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">

<h3 id="Java-中操作字符串都有哪些类-它们之间有什么区别-">Java 中操作字符串都有
些类？它们之间有什么区别？</h3>
操作字符串的类有**：<code>String</c
de>、<code>StringBuffer</code>、<code>StringBu
lder</code>。**
String 和 StringBuffer、StringBuilder 的区别在于 Str
ng 声明的是不可变的对象，每次操作都会生成新的 String 对象，
后将指针指向新的 String 对象。
而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，
以在经常改变字符串内容的情况下最好不要使用 String。
StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全
，而 StringBuilder 是非线程安全的**，但 StringBuilder 的性能却高于 String
uffer，**
所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。
/strong>
<h3 id="String-StringBuffer和StringBuilder区别-类似上一题-">String、StringBuffer 和
StringBuilder 区别(类似上一题)</h3>
1、数据可变和不可变

<code>String</code> 底层使用一个不可变的字符数组 <code>private f
nal char value[];</code> 所以它内容不可变。
<code>StringBuffer</code> 和 <code>StringBuilder</code> <stro
g>都继承了 <code>AbstractStringBuilder</code> 底层使用的是可变字符
组：<code>char[] value;</code>

2、线程安全

<code>StringBuilder</code> 是线程不安全的，效率较高；而 <code>S
ringBuffer</code> 是线程安全的，效率较低。

通过他们的 <code>append()</code> 方法来看，
code>StringBuffer</code> 是有同步锁，而 <code>StringBuilder</code>
没有：
<pre><code class="highlight-chroma"><span class="highlight
cl"> @Override
 public syn
hronized StringBuffer append(Object obj) {
    &
bsp;toStringCache = null;
    &
bsp;super.append(String.valueOf(obj));
    &
bsp;return this;
 }
 @Override
 public Stri
gBuilder append(String str) {
    &
bsp;super.append(str);
    &

bsp;return this;
 }
</code></pre>
3、 相同点
<code>StringBuilder</code> 与 <code>StringBuffer</code> 有公共父类 <code>AbstractStringBuilder</code>。
最后，操作可变字符串速度：<code>StringBuilder > StringBuffer > String</code>，这个答案就显得不足为奇了。
<h3 id="String-str--i-与-String-str-new-String--i--一样吗-">String str="i"与 String str=new String("i")一样吗？</h3>
不一样，因为内存的分配方式不一样。String str="i"的方式，Java 虚拟机会将其分配常量池中；而 String str=new String("i") 则会被分到堆内存中。
代码示例：
<pre><code class="highlight-chroma"> String x = "叶痕秋";
 String y = 叶痕秋";
 String z = ew String("叶痕秋");
 System.out println(x == y); // true
 System.out println(x == z); // false
</code></pre>
String x = "叶痕秋" 的方式，Java 虚拟机会将其分配到常量池中，而常量池中没有重的元素，比如当执行“叶痕秋”时，java 虚拟机会先在常量池中检索是否已经有“叶痕秋”，如果有么就将“叶痕秋”的地址赋给变量，如果没有就创建一个，然后在赋给变量；
而 String z = new String(“叶痕秋”) 则会被分到堆内存中，即使内容一样还是会创新的对象。
<h3 id="String-类的常用方法都有那些-">String 类的常用方法都有那些？</h>

indexOf()：返回指定字符的索引。
charAt()：返回指定索引处的字符。
replace()：字符串替换。
trim()：去除字符串两端空白。
split()：分割字符串，返回一个分割后的字符串数组。
getBytes()：返回字符串的 byte 类型数组。
length()：返回字符串长度。
toLowerCase()：将字符串转成小写字母。
toUpperCase()：将字符串转成大写字符。
substring()：截取字符串。
equals()：字符串比较。

<h3 id="String-s---new-String--xyz---创建了几个StringObject-是否可以继承String类-">String s = new String("xyz");创建了几个 StringObject？是否可以继承 String 类?</h>
两个或一个都有可能，” xyz”对应一个对象，这个对象放在字符串常量缓冲区，常量xyz”不管出现多少遍，都是缓冲区中的那一个。NewString 每写一遍，就创建一个新的对象，它使常量”xyz”对象的内容来创建出一个新 String 对象。如果以前就用过’xyz’，那么这里就不会创”xyz”了，直接从缓冲区拿，这时创建了一个 StringObject；但如果以前没有用过"xyz"，那么此时会创建一个对象并放入缓冲区，这种情况它创建两个对象。至于 String 类是否继承，答案是否定的因为 String 默认 final 修饰，是不可继承的。
<h3 id="下面这条语句一共创建了多少个对象-String-s--a---b---c---d-">下面这条语句</h3>

共创建了多少个对象：String s="a"+"b"+"c"+"d"</h3>
对于如下代码：
<pre><code class="highlight-chroma"> String s1 = "a";

 String s2 = s1 + "b";

 String s3 = "a" + "b";

 System.out println(s2 == "ab");

 System.out println(s3 == "ab");
</code></pre>
第一条语句打印的结果为 false，第二条语句打印的结果为 true，这说明 javac 编译以对字符串常量直接相加的表达式进行优化，不必要等到运行期再去进行加法运算处理，而是在编译去掉其中的加号，直接将其编译成一个这些常量相连的结果。
题目中的第一行代码被编译器在编译时优化后，相当于直接定义了一个”abcd”的字串，所以，上面的代码应该只创建了一个 String 对象。写如下两行代码，
<pre><code class="highlight-chroma"> String s ="a" + "b" +"c" + "d";

 System.out println(s== "abcd");
</code></pre>
最终打印的结果应该为 true。
<h3 id="简述Java中的集合">简述 Java 中的集合</h3>

Collection 下：List 系(有序、元素允许重复)和 Set 系(无序、元素不重复)
<blockquote>
set 根据 equals 和 hashcode 判断，一个对象要存储在 Set 中，必须重写 equals 和 ashCode 方法
</blockquote>

Map 下：HashMap 线程不同步；TreeMap 线程同步
Collection 系列和 Map 系列：Map 是对 Collection 的补充，两个没什么关系

<h3 id="List-Map-Set三个接口-存取元素时-各有什么特点-">List、Map、Set 三个接口存取元素时，各有什么特点？</h3>
首先，List 与 Set 具有相似性，它们都是单列元素的集合，所以，它们有一个共同的接口，叫 Collection。
1、Set 里面不允许有重复的元素
即不能有两个相等（注意，不是仅仅是相同）的对象，即假设 Set 集合中有了一个 A 对象，在我要向 Set 集合再存入一个 B 对象，但 B 对象与 A 对象 equals 相等，则 B 对象存储不进去，**所以，Set 集合的 add 方法有一个 boolean 的返回值，当集合中没有某个元素，此时 add 方法可功加入该元素时，则返回 true，当集合含有与某个元素 equals 相等的元素时，此时 add 方法无法加该元素，返回结果为 false。****Set 取元素时，不能细说要取第几个，只能以 Iterator 接取得所有的元素，再逐一遍历各个元素。
2、List 表示有先后顺序的集合
注意，不是那种按年龄、按大小、按价格之类的排序。当我们多次调用 add(Obje)方

时，每次加入的对象就像火车站买票有排队顺序一样，按先来后到的顺序排序。有时候，也可以插队即调用 add(intindex,Obj e)方法，就可以指定当前对象在集合中的存放位置。一个对象可以被反复储进 List 中，每调用一次 add 方法，这个对象就被插入进集合中一次，其实，并不是把这个对象本存储进了集合中，而是在集合中用一个索引变量指向这个对象，当这个对象被 add 多次时，即相当集合中有多个索引指向了这个对象。List 除了可以用 Iterator 接口取得所有的元素，再逐一遍历各个素之外，还可以调用 get(index i)来明确说明取第几个。
3、Map 与 List 和 Set 不同
它是双列的集合，其中有 put 方法，定义如下：put(obj key,obj value)，每次存储时要存储一对 key/value，不能存储重复的 key，这个重复的规则也是按 equals 比较相等。取则可以据 key 获得相应的 value，即 get(Object key)返回值为 key 所对应的 value。另外，也可以获得所的 key 的结合，还可以获得所有的 value 的结合，还可以获得 key 和 value 组合成的 Map.Entry 对的集合。
<h4 id="总结">总结</h4>
List 以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 存 key-value 值，value 可多值。
<h3 id="Set里的元素是不能重复的-那么用什么方法来区分重复与否呢-是用--还是equals---它们有区别-">Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢?是用==还是 equas()?它们有何区别?</h3>
Set 里的元素是不能重复的，元素重复与否是使用 equals()方法进行判断的。</stron>
==和 equal 区别也是考烂了的题，这里再重复说一下：
=操作符专门用来比较两个变量的值是否相等，也就是用于比较变量所对应的内存中存储的数值是否相同，要比较两个基本类型的数据或两个引用变量是否相等，只能用=**作符。**
equals 方法是用于比较两个独立对象的内容是否相同，就好比去比较两个人的长相是相同，它比较的两个对象是独立的。
比如：两条 new 语句创建了两个对象，然后用 a/b 这两个变量分别指向了其中一个象，这是两个不同的对象，它们的首地址是不同的，即 a 和 b 中存储的数值是不相同的，所以，表式 a==b 将返回 false，而这两个对象中的内容是相同的，所以，表达式 a.equals(b)将返回 true。</trong>
<h3 id="ArrayList和LinkedList区别-">ArrayList 和 LinkedList 区别？</h3>

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。</strng>
对于随机访问 get 和 set，ArrayList 觉得优于 LinkedList，因为 LinkedList 要移动指。
对于新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动据。

<h3 id="ArrayList和Vector的区别">ArrayList 和 Vector 的区别</h3>
两个类都实现了 List 接口（List 接口继承了 <code>Collection</code> 接口），他们都是有序集合，即存储在这两个集合中的元素的位置都是有顺序的，相当于一种动态的数组，我们以后可以按位置引号取出某个元素，并且其中的数据是允许重复的，这是与 <code>HashSet</code> 之类的集合的最大不同处，<code>HashSet</code> 之类的集合不可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：.

同步性：
<code>Vector</code> 是线程安全的，也就是说是它的方法之间是线程同步的，而/strong> <code>ArrayList</code> 是线程序不安全的，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 <code>ArrayList</code>，为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用 <code>Vector</code>，因为不需要我们自己再去考虑和编写线程安全的代码。

数据增长：
<code>ArrayList</code> 与 <code>Vector</code> 都有个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加 <code>ArrayList</code> 与 <code>Vector</code> 的存储空间，每次要加存储空间时，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空间利用与程序效率之间要取得一定的平衡。<code>Vector</code> 默认增为原来两倍，而 <code>ArrayList</code> 的增长策略在文档中没有明确规（从源代码看到的是增长为原来的 1.5 倍）。<code>ArrayList</code> 与 <code>Vector</code> 都可以设置初始的空间大小，<code>Vector</code> 还可以设置增长的空间大小，而 <code>ArrayList</code> 没有提供设置增长空间的方法。
总结：即 Vector 增长原来的一倍，<code>ArrayList</code> 增加原来的 0.5 倍。
<h3 id="ArrayList-Vector-LinkedList的存储性能和特性">ArrayList,Vector,LinkedList 存储性能和特性</h3>
<code>ArrayList</code> 和 <code>Vector</code> 都是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，它们都允许直接按序索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，
<code>Vector</code> 由于使用了 <code>synchronized</code> 方法（线程安全），通常性能上较 <code>ArrayList</code> 差。而 <code>LinkedList</code> 使用双向链表实现存储，按序号索引数据需要进行前或后向遍历，索引就变慢了，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。
<code>LinkedList</code> 也是线程不安全的，<code>LinkedList</code> 提供了一些方法，使得 <code>LinkedList</code> 可以被当堆栈和队列来使用。
<h3 id="HashMap和Hashtable的区别">HashMap 和 Hashtable 的区别</h3>
<code>HashMap</code> 是 <code>Hashtable</code> 轻量级实现（非线程安全的实现），他们都完成了 Map 接口，
主要区别在于 <code>HashMap</code> 允许空（null）键值 key）,由于非线程安全，在只有一个线程访问的情况下，效率要高于 <code>Hashtable</code>。
<code>HashMap</code> 允许将 null 作为一个 entry 的 key 或者 value，而 <code>Hashtable</code> 不允许。
<code>HashMap</code> 把 <code>Hashtable</code> <code>contains</code> 方法去掉了，改成 <code>containsvalue</code> 和 <code>containsKey</code>。因为 contains 方容易让人引起误解。
<code>Hashtable</code> 继承自 <code>Dictionary</code> 类，而 <code>HashMap</code> 是 Java1.2 引进的 Map interface 的一实现。
最大的不同是，<code>Hashtable</code> 的方法是 <code>Synchronize</code> 的，而 <code>HashMap</code> 不是，在多个线程访问 <code>Hashtable</code> 时，不需要自己为它的法实现同步，而 <code>HashMap</code> 就必须为之提供同步。
就 <code>HashMap</code> 与 <code>HashTable</code> 主要从三方面来说。

历史原因:<code>Hashtable</code> 是基于陈旧的 <code>Dictionary</code> 类的，<code>HashMap</code> 是 Java 1.2 引进的 Map 接口的一个实现

**同步性: **<code>Hashtable</code> 是线程安全的，也就是说是同步的，而 <code>HashMap</code> 是线程序不安全的，不是同步的
值：只有 <code>HashMap</code> 可以让你将空值作为一表的条目的 key 或 value

<h3 id="Java中的同步集合与并发集合有什么区别">Java 中的同步集合与并发集合有什区别</h3>
同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍系统的扩展性。Java5 介绍了并发集合 <code>ConcurrentHashMap</code>，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。
不管是同步集合还是并发集合他们都支持线程安全，他们之间主要的区别体现在性能可扩展性，还有他们如何实现的线程安全上。
同步 <code>HashMap</code>，<code>Hashtabl</code>，<code>HashSet</code>，<code>Vector</ode>，<code>ArrayList</code> 相比他们并发的实现（<code>ConcurrentHashMap</code>，<code>CopyOnWriteArrayList</cde>，<code>CopyOnWriteHashSet</code>）会慢得多。造成此慢的主要原因是锁， 同步集合会把整个 Map 或 List 锁起来，而并发集合不会。并发集合实现线程全是通过使用先进的和成熟的技术像锁剥离。
比如 <code>ConcurrentHashMap</code> 会把整个 Map 分成几个片段，只对相关的几个片段上锁，同时允许多线程访问其他未上锁的片段。
同样的，<code>CopyOnWriteArrayList</code> 允许多个线以非同步的方式读，当有线程写的时候它会将整个 List 复制一个副本给它。
如果在读多写少这种对并发集合有利的条件下使用并发集合，这会比使用同步集合更有可伸缩性。
<h3 id="Java中的集合及其继承关系">Java 中的集合及其继承关系</h3>
关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原更该如此.这里我们看这张图即可:
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile com/file/2022/04/solo-fetchupload-15364346277393864071-03459f0a.png?imageView2/2/in erlace/1/format/jpg">
<h3 id="poll--方法和remove--方法区别-">poll()方法和 remove()方法区别?</h3>
poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候返回空，但是 remove() 失败的时候会抛出异常。
<h3 id="LinkedHashMap和PriorityQueue的区别">LinkedHashMap 和 PriorityQueue 的区别</h3>
PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。
<h3 id="WeakHashMap与HashMap的区别是什么-">WeakHashMap 与 HashMap 的别是什么?</h3>
WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就当 key 对象没有任何引用时，key/value 将会被回收。
<h3 id="ArrayList和LinkedList的区别-">ArrayList 和 LinkedList 的区别?</h>
最明显的区别是 ArrrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O 1)，而 LinkedList 是 O(n)。
<h3 id="ArrayList和Array有什么区别-">ArrayList 和 Array 有什么区别?</h>
Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象。
ArrayList 是 Java 集合框架类的一员,可以称它为一个动态数组. array 是静态的,所以

个数据一旦创建就无法更改他的大小
<h3 id="ArrayList和HashMap默认大小-">ArrayList 和 HashMap 默认大小?</h3>
在 java 7 中，ArrayList 的默认大小是 10 个元素，HashMap 的默认大小是 16 个元（必须是 2 的幂）。这就是 Java 7 中 ArrayList 和 HashMap 类的代码片段
<pre><code class="highlight-chroma"> private static final int DEFAULT_CAPACITY = 10;

  //from HashMap.java JDK 7
  static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
</code></pre>
<h3 id="Comparator和Comparable的区别-">Comparator 和 Comparable 的区别?</h3>
相同点
都是用于比较两个对象“顺序”的接口
都可以使用 Collections.sort()方法来对对象集合进行排序
不同点
Comparable 位于 java.lang 包下，而 Comparator 则位于 java.util 包下
Comparable 是在集合内部定义的方法实现的排序，Comparator 是在集合外部实现排序
总结
使用 Comparable 接口来实现对象之间的比较时，可以使这个类型（设为 A）实现 Comparable 接口，并可以使用 Collections.sort()方法来对 A 类型的 List 进行排序，之后可以通过 a1.compareTo(a2)来比较两个对象；
当使用 Comparator 接口来实现对象之间的比较时，只需要创建一个实现 Comparator 接口的比较器（设为 AComparator），并将其传给 Collections.sort()方法即可对 A 类型的 List 进行排序，之后也可以通过调用比较器 AComparator.compare(a1, a2)来比较两个对象。

可以说一个是自己完成比较，一个是外部程序实现比较的差别而已。
用 Comparator 是策略模式（strategy design pattern），就是不改变对象自身，而一个策略对象（strategy object）来改变它的行为。
比如：你想对整数采用绝对值大小来排序，Integer 是不符合要求的，你不需要去修改 Integer 类（实际上你也不能这么做）去改变它的排序行为，这时候只要（也只有）使用一个实现了 Comparator 接口的对象来实现控制它的排序就行了。
两种方式，各有各的特点：使用 Comparable 方式比较时，我们将比较的规则写入了比较的类型中，其特点是高内聚。但如果哪天这个规则需要修改，那么我们必须修改这个类型的源代码。如果使用 Comparator 方式比较，那么我们不需要修改比较的类，其特点是易维护，但需要自定义一个比较器，后续比较规则的修改，仅仅是改这个比较器中的代码即可。
<h3 id="如何实现集合排序-">如何实现集合排序?</h3>
你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。
<h3 id="如何打印数组内容">如何打印数组内容</h3>
你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内容，但是 Arrays.toString() 可以打印每个元素。
<h3 id="LinkedList的是单向链表还是双向-">LinkedList 的是单向链表还是双向?</h3>
双向循环列表,具体实现自行查阅源码.
<h3 id="TreeMap是实现原理">TreeMap 是实现原理</h3>
TreeMap 是一个通过红黑树实现有序的 key-value 集合。
TreeMap 继承 AbstractMap，也即实现了 Map，它是一个 Map 集合</

>
TreeMap 实现了 NavigableMap 接口，它支持一系列的导航方法，
TreeMap 实现了 Cloneable 接口，它可以被克隆
TreeMap 本质是 Red-Black Tree，它包含几个重要的成员变量：root、size、comparator。其中 root 是红黑树的根节点。它是 Entry 类型，Entry 是红黑树的节点，它包含了红黑树的 6 个基本组成：key、value、left、right、parent 和 color。Entry 节点根据根据 Key 排序，包含的内是 value。Entry 中 key 比较大小是根据比较器 comparator 来进行判断的。size 是红黑树的节点个。
<h3 id="遍历ArrayList时如何正确移除一个元素">遍历 ArrayList 时如何正确移除一个元</h3>
错误写法示例一：
<pre><code class="highlight-chroma"> public static void remove(ArrayList<String> list) {
     for (int i = 0; i < list.size(); i++) {
       String s = list.get(i);
     if (s.equals("bb")) {
         list.remove(s);
      }
    }
 }
</code></pre>
错误写法示例二：
<pre><code class="highlight-chroma"> public static void remove(ArrayList<String> list) {
     for (String s : list) {
     if (s.equals("bb")) {
         list.remove(s);
      }
    }
 }
</code></pre>
要分析产生上述错误现象的原因唯有翻一翻 jdk 的 ArrayList 源码，先看下 ArrayList 的 remove 方法（注意 ArrayList 中的 remove 有两个同名方法，只是入参不同，这里看的是入参为 bject 的 remove 方法）是怎么实现的：
<pre><code class="highlight-chroma"> public boolean remove(Object o) {
     if (o == null) {
       for (int index = 0; index < size; index++)
         if (elementData[index] == null) {
    &

bsp;   fastRemove(index);
    &
bsp;   return true;
    &
bsp;   }
     } e
se {
    &
bsp;   for (int index = 0; index < size; index++)
    &
bsp;   if (o.equals(elementData[index])) {
    &
bsp;   fastRemove(index);
    &
bsp;   return true;
    &
bsp;   }
     }

    &
bsp;return false;
 }
</code></pre>
按一般执行路径会走到 else 路径下最终调用 faseRemove 方法：
<pre><code class="highlight-chroma"><span class="highlight
cl">private void fastRemove(int index) {
    modCount++;
    int numMoved
 size - index - 1;
    if (numMoved
gt; 0)
        System.array
opy(elementData, index+1, elementData, index,
                    n
mMoved);
    elementData[--s
ze] = null; // Let gc do its work
}
</code></pre>
可以看到会执行 System.arraycopy 方法，导致删除元素时涉及到数组元素的移动。对错误写法一，在遍历第二个元素字符串 bb 时因为符合删除条件，所以将该元素从数组中删除，并将后一个元素移动（也是字符串 bb）至当前位置，导致下一次循环遍历时后一个字符串 bb 并没有历到，所以无法删除。 针对这种情况可以倒序删除的方式来避免：
<pre><code class="highlight-chroma"><span class="highlight
cl">public static void remove(ArrayList<String> list) {
    for (int i = list.si
e() - 1; i >= 0; i--) {
        String s = list
get(i);
        if (s.equals("
b")) {
            list.remove
s);
        }

```
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>因为数组倒序遍历时即使发生元素删除也不影响后序元素遍历。</strong></p>
<p><strong>而错误二产生的原因却是 foreach 写法是对实际的 Iterable、hasNext、next 方法的写，问题同样处在上文的 fastRemove 方法中，可以看到第一行把 modCount 变量的值加一，但在 ArrayList 返回的迭代器（该代码在其父类 AbstractList 中）：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public Iterator&amp;lt;E&amp;gt; iterator() {
</span></span><span class="highlight-line"><span class="highlight-cl">    return new Itr();

</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>这里返回的是 AbstractList 类内部的迭代器实现 private class Itr implements Iterato，看这个类的 next 方法：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public E next() {
</span></span><span class="highlight-line"><span class="highlight-cl">    checkForComodification();
</span></span><span class="highlight-line"><span class="highlight-cl">    try {
</span></span><span class="highlight-line"><span class="highlight-cl">        E next = get(cursor);
</span></span><span class="highlight-line"><span class="highlight-cl">        lastRet = cursor++;
</span></span><span class="highlight-line"><span class="highlight-cl">        return next;
</span></span><span class="highlight-line"><span class="highlight-cl">    } catch (IndexOutOfBoundsException e) {
</span></span><span class="highlight-line"><span class="highlight-cl">        checkForComodification();
</span></span><span class="highlight-line"><span class="highlight-cl">        throw new NoSuchElementException();
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>第一行 checkForComodification 方法：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">final void checkForComodification() {
</span></span><span class="highlight-line"><span class="highlight-cl">    if (modCount != expectedModCount)
</span></span><span class="highlight-line"><span class="highlight-cl">        throw new ConcurrentModificationException();
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>这里会做迭代器内部修改次数检查，因为上面的 remove(Object)方法把修改了 modCunt 的值，所以才会报出并发修改异常。要避免这种情况的出现则在使用迭代器迭代时（显示或 foreah 的隐式）不要使用 ArrayList 的 remove，改为用 Iterator 的 remove 即可。</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public static void remove(ArrayList&amp;lt;String&amp;gt; list) {
</span></span><span class="highlight-line"><span class="highlight-cl">    Iterator&amp;lt;String&amp;gt; it = list.iterator();
</span></span><span class="highlight-line"><span class="highlight-cl">    while (it.hasNext()) {
</span></span><span class="highlight-line"><span class="highlight-cl">        String s = it.ext();
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">        if (s.equals("
b")) {
</span></span><span class="highlight-line"><span class="highlight-cl">            it.remove();

</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h3 id="HashMap的实现原理"><strong>HashMap 的实现原理</strong></h3>
<p><strong>HashMap 是基于哈希表实现的 map，哈希表（也叫关联数组）一种通用的数据结构是 Java 开发者常用的类，常用来存储和获取数据，功能强大使用起来也很方便，是居家旅行...不对，Java 开发需要掌握的基本技能，也是面试必考的知识点，所以，了解 HashMap 是很有必要的。</strong></p>
<p><strong><strong>原理</strong></strong></p>
<p><strong>简单讲解下 HashMap 的原理：HashMap 基于 Hash 算法，我们通过 put(key,value)存储，get(key)来获取。当传入 key 时，HashMap 会根据 key.hashCode()计算出 hash 值，根据 hash 值将 value 保存在 bucket 里。当计算出的 hash 值相同时怎么办呢，我们称之为 Hash 冲突，HashMap 的做法是用链表和红黑树存储相同 hash 值的 value。当 Hash 冲突的个数比较少时，使用链表否则使用红黑树。</strong></p>
<p><strong><strong>内部存储结构</strong></strong></p>
<p><strong>HashMap 类实现了 Map&amp;lt; K, V&amp;gt; 接口，主要包含以下几个方法：</strong></p>
<ul>
<li><strong>V put(K key, V value)</strong></li>
<li><strong>V get(Object key)</strong></li>
<li><strong>V remove(Object key)</strong></li>
<li><strong>Boolean containsKey(Object key)</strong></li>
</ul>
<p><strong>HashMap 使用了一个内部类 Node&amp;lt; K, V&amp;gt; 来存储数据</strong></p>
<blockquote>
<p><strong>我阅读的是 Java 8 的源码，在 Java 8 之前存储数据的内部类是 Entry&amp;lt; K, V&amp;gt; 代码大体都是一样的</strong></p>
</blockquote>
<p><strong>Node 代码：</strong></p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">static class Node&amp;lt;K,V&amp;gt; implements Map.Entry&amp;lt;K,V&amp;gt; {
</span></span><span class="highlight-line"><span class="highlight-cl">    final int hash;
</span></span><span class="highlight-line"><span class="highlight-cl">    final K key;
</span></span><span class="highlight-line"><span class="highlight-cl">    V value;
</span></span><span class="highlight-line"><span class="highlight-cl">    Node&amp;lt;K,V&amp;gt; next;
</span></span><span class="highlight-line"><span class="highlight-cl">    ...
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p><strong>可以看见 Node 类中除了键值对（key-value）以外，还有额外的两个数据：</strong></p>
<ul>
<li><strong>hash : 这个是通过计算得到的散列值</strong></li>
<li><strong>next：指向另一个 Node，这样 HashMap 可以像链表一样存储数据</strong></li>
</ul>
<p><strong>因此可以知道，HashMap 的结构大致如下：</strong></p>
<p><strong>我们可以将每个横向看成一个个的桶，每个桶中存放着具有相同 Hash 值的 Node,通一个 list 来存放每个桶。</strong></p>
<p><strong><strong>内部变量</strong></strong></p>
```

<pre><code class="highlight-chroma">// 默认容量大小
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
// 最大容量
static final int MAXIMUM_CAPACITY = 1 << 30;
// 装载因子
static final float DEFAULT_LOAD_FACTOR = 0.75f;
// 转换为二叉树的值
static final int TREEIFY_THRESHOLD = 8;
// 转换为二叉树的低阀值
static final int UNREEIFY_THRESHOLD = 6;
// 二叉树最小容量
static final int MINTREEIFY_CAPACITY = 64;
// 哈希表
transient Node&ltK,V>[] table;
// 键值对的数量
transient int size;
// 记录HashMap构改变次数，与HashMap的快速失败相关
transient int modount;
// 扩容的阈值
int threshold;
// 装载因子
final float loadFactor;
</code></pre>
常用方法
put 操作
put 函数大致的思路为：

对 key 的 hashCode()做 hash，然后再计算 index;
如果没碰撞直接放到 bucket 里；
如果碰撞了，以链表的形式存在 buckets 后；
如果碰撞导致链表过长(大于等于 TREEIFY_THRESHOLD)，就把链表转换成红黑树；
如果节点已经存在就替换 old value(保证 key 的唯一性)
如果 bucket 满了(超过 load factor*current capacity)，就要 resize。

<pre><code class="highlight-chroma">public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

```
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">final V putVal(int
ash, K key, V value, boolean onlyIfAbsent,
</span></span><span class="highlight-line"><span class="highlight-cl">               boole
n evict) {
</span></span><span class="highlight-line"><span class="highlight-cl">    Node&amp;lt;K,V&amp;gt
[] tab; Node&amp;lt;K,V&amp;gt; p; int n, i;
</span></span><span class="highlight-line"><span class="highlight-cl">    if ((tab = table)
== null || (n = tab.length) == 0)
</span></span><span class="highlight-line"><span class="highlight-cl">        n = (tab = res
ze()).length; // resize()是调整table数组大小的，如果table数组为空或长度为0，重新调整大小
</span></span><span class="highlight-line"><span class="highlight-cl">    if ((p = tab[i = (
 - 1) &amp;amp; hash]) == null) // i = (n - 1) &amp;amp; hash | 这里计算出来的i值就是存放数组的位置，
果当前位置为空，则直接放入其中
</span></span><span class="highlight-line"><span class="highlight-cl">        tab[i] = new
ode(hash, key, value, null);
</span></span><span class="highlight-line"><span class="highlight-cl">    else { // hash
突
</span></span><span class="highlight-line"><span class="highlight-cl">        Node&amp;lt;K,V
gt; e; K k;
</span></span><span class="highlight-line"><span class="highlight-cl">        if (p.hash ==
hash &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">            ((k = p.key)
== key || (key != null &amp;amp;&amp;amp; key.equals(k)))) // 如果hash相同，并且key值也相同，则找
存放位置
</span></span><span class="highlight-line"><span class="highlight-cl">            e = p;
</span></span><span class="highlight-line"><span class="highlight-cl">        else if (p inst
nceof TreeNode) // 如果当前p是二叉树，则放入二叉树中
</span></span><span class="highlight-line"><span class="highlight-cl">            e = ((Tree
ode&amp;lt;K,V&amp;gt;)p).putTreeVal(this, tab, hash, key, value);
</span></span><span class="highlight-line"><span class="highlight-cl">        else { // 存
到链表中
</span></span><span class="highlight-line"><span class="highlight-cl">            for (int bin
ount = 0; ; ++binCount) {
</span></span><span class="highlight-line"><span class="highlight-cl">                if ((e = p
next) == null) { // 遍历链表并将值放到链表最后
</span></span><span class="highlight-line"><span class="highlight-cl">                    p.next
= newNode(hash, key, value, null);
</span></span><span class="highlight-line"><span class="highlight-cl">                    if (bi
Count &amp;gt;= TREEIFY_THRESHOLD - 1) // -1 for 1st
</span></span><span class="highlight-line"><span class="highlight-cl">                        tree
fyBin(tab, hash); // 如果链表中的值大于TREEIFY_THRESHOLD - 1，则将链表转换成二叉树
</span></span><span class="highlight-line"><span class="highlight-cl">                    break;
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">                if (e.has
 == hash &amp;amp;&amp;amp;
</span></span><span class="highlight-line"><span class="highlight-cl">                    ((k = e
key) == key || (key != null &amp;amp;&amp;amp; key.equals(k))))
</span></span><span class="highlight-line"><span class="highlight-cl">                    break;
</span></span><span class="highlight-line"><span class="highlight-cl">                p = e;
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        if (e != null) {
/ 表示对于当前key早已经存在
```

            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null) // 如果onlyIfAbsent为false或则oldValue为空，替换原来的值
                e.value = value;
            afterNodeAccess(e);
            return oldValue; // 返回原来的值
        }
    }
    ++modCount; // HashMap结构修改次数，主要用于判断迭代器中fail-fast
    if (++size > threshold) // 如果++size后的值比阀值大，则重新调整大小
        resize();
    afterNodeInsertion(evict);
    return null;
}
</code></pre>
代码也比较容易看懂，值得注意的就是
<pre><code class="highlight-chroma">else if (p instanceof TreeNode) // 如果当前p是二叉树，则放入二叉树中
    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
</code></pre>
与
<pre><code class="highlight-chroma">if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
    treeifyBin(tab, hash); // 如果链表中的值大于TREEIFY_THRESHOLD - 1，则将链表转换成二叉树
</code></pre>
这是 Java 8 相对于以前版本一个比较大的改变。
在 Java 8 以前，每次产生 hash 冲突，就将记录追加到链表后面，然后通过遍历链表查找。如果某个链表中记录过大，每次遍历的数据就越多，效率也就很低，复杂度为 O(n)；
在 Java 8 中，加入了一个常量 TREEIFY_THRESHOLD=8，如果某个链表中的记录大这个常量的话，HashMap 会动态的使用一个专门的 treemap 实现来替换掉它。这样复杂度是 O(log)，比链表的 O(n)会好很多。
对于前面产生冲突的那些 KEY 对应的记录只是简单的追加到一个链表后面，这些记录能通过遍历来进行查找。但是超过这个阈值后 HashMap 开始将列表升级成一个二叉树，使用哈希值为树的分支变量，如果两个哈希值不等，但指向同一个桶的话，较大的那个会插入到右子树里。如果希值相等，HashMap 希望 key 值最好是实现了 Comparable 接口的，这样它可以按照顺序来进行入。
get 操作
在理解了 put 之后，get 就很简单了。大致思路如下：

bucket 里的第一个节点，直接命中；
如果有冲突，则通过 key.equals(k)去查找对应的 entry
若为树，则在树中通过 key.equals(k)查找，O(logn)；
若为链表，则在链表中通过 key.equals(k)查找，O(n)。


```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k)))) // 如果hash相同并且key值一样则返回当前node
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode) // 如果当前node为二叉树，则在二叉树中查找
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do { // 遍链表
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

HashMap 自动扩容

如果在初始化 HashMap 中没有指定初始容量，那么默认容量为 16，但是如果后来 HashMap 中存放的数量超过了 16，那么便会有大量的 hash 冲突；在 HashMap 中有自动扩容机制，果当前存放的数量大于某个界限，HashMap 便会调用 resize()方法，扩大 HashMap 的容量。

当 hashmap 中的元素个数超过数组大小*loadFactor 时就会进行数组扩容，loadFactor 的默认值为 0.75，也就是说，默认情况下，数组大小为 16，那么当 ashmap 中元素个数超过 16* 0.75=12 的时候，就把数组的大小扩展为 2*16=3，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果们已经预知 hashmap 中元素的个数，那么预设元素的个数能够有效的提高 hashmap 的性能。

HashMap 的 capacity 必须满足是 2 的 N 次方,如果在构造函数内指定的容量 n 不满足 HashMap 会通过下面的算法将其转换为大于 n 的最小的 2 的 N 次方数。
<pre><code class="highlight-chroma">// 减1→移位→按位或运算→加1返回
static final int tablSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
}
</code></pre>
<h3 id="HashMap线程安全吗-">HashMap 线程安全吗？</h3>
HashMap 是非线程安全的，如果在多线程环境下，可以使用 HashTable，HashTable 中所有 CRUD 操作都是线程同步的，同样的，线程同步的代价就是效率变低了。
再 Java 5 以后，有了一个线程安全的 HashMap——ConcurrentHashMap，ConcurrentHashMap 相对于 HashTable 来说，ConcurrentHashMap 将 hash 表分为 16 个桶（默认值）诸如 get,put,remove 等常用操作只锁当前需要用到的桶。试想，原来只能一个线程进入，现在却能时 16 个写线程进入（写线程才需要锁定，而读线程几乎不受限制，并发性的提升是显而易见。
快速失败(fast-fail)
“快速失败”也就是 fail-fast，它是 Java 集合的一种错误检测机制。当多个线程对集进行结构上的改变的操作时，有可能会产生 fail-fast 机制。记住是有可能，而不是一定。例如：假设在两个线程（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 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抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。
在 HashMap 的 forEach 方法中有以下代码：
<pre><code class="highlight-chroma">@Override
public void forEach(BiConsumer<? super K, ? super V> action) {
    Node<K,V>[] tab;
    if (action == null)
        throw new NullPointerException();
    if (size > 0 && (tab = table) != null) {
        int mc = modCount;
        for (int i = 0; i < tab.length; ++i) {
            for (Node<K,V> e = tab[i]; e != null; e = e.next)
</code></pre>

                action.accept(e.key, e.value);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
}
</code></pre>
在上面我们说到，modCount 是记录每次 HashMap 结构修改。 forEach 方法会在进入 for 循环之前，将 modCount 赋值给 mc，如果在 for 循环之后，HashMap 的结构变化了，那导致的结果就是 modCount != mc，则抛出 ConcurrentModificationException()异常。
<h3 id="HashMap总结">HashMap 总结</h3>
1、什么时候会使用 HashMap？他有什么特点？ 是基于 Map 接口的实现，存储键值时，它可以接收 null 的键值，是非同步的，HashMap 存储着 Entry(hash, key, value, next)对象。
2、你知道 HashMap 的工作原理吗？ 通过 hash 的方法，通过 put 和 get 存储和获对象。存储对象时，我们将 K/V 传给 put 方法时，它调用 hashCode 计算 hash 从而得到 bucket 置，进一步存储，HashMap 会根据当前 bucket 的占用情况自动调整容量(超过 Load Facotr 则 resiz 为原来的 2 倍)。获取对象时，我们将 K 传给 get，它调用 hashCode 计算 hash 从而得到 bucket 置，并进一步调用 equals()方法确定键值对。如果发生碰撞的时候，Hashmap 通过链表将产生碰撞突的元素组织起来，在 Java 8 中，如果一个 bucket 中碰撞冲突的元素超过某个限制(默认是 8)，则用红黑树来替换链表，从而提高速度。
3、你知道 get 和 put 的原理吗？equals()和 hashCode()的都有什么作用？ 通过对 ky 的 hashCode()进行 hashing，并计算下标(n-1 & hash)，从而获得 buckets 的位置。如果生碰撞，则利用 key.equals()方法去链表或树中去查找对应的节点
4、你知道 hash 的实现吗？为什么要这样实现？ 在 Java 1.8 的实现中，是通过 hashode()的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度功效、质量来考虑的，这么做可以在 bucket 的 n 比较小的时候，也能保证考虑到高低 bit 都参与到 hsh 的计算中，同时不会有太大的开销。
5、如果 HashMap 的大小超过了负载因子(load factor)定义的容量，怎么办？ 如果过了负载因子(默认 0.75)，则会重新 resize 一个原来长度两倍的 HashMap，并且重新调用 hash 方。
<hr>
<blockquote>
前段时间因为找工作的缘故背了一些关于 HashMap 的面试题，死记硬背，也不是很，最近看了源码，很多知识才变的清晰，而且看源码挺有趣的。再接再厉。
</blockquote>
<h3 id="Java集合框架是什么-说出一些集合框架的优点-">Java 集合框架是什么？说出一集合框架的优点？</h3>
**每种编程语言中都有集合。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。
<h3 id="集合框架中的泛型有什么优点-">集合框架中的泛型有什么优点？</h3>
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。它也给运行时带来处，因为不会产生类型检查的字节码指令。

<h3 id="Java集合框架的基础接口有哪些-">Java 集合框架的基础接口有哪些？</h3>
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多只能映射一个 value。
一些其它的接口有 Queue、Dequeue、SortedSet、SortedMap 和 ListIterator。
<h3 id="为何Collection不从Cloneable和Serializable接口继承-">为何 Collection 不从 Cloneable 和 Serializable 接口继承？</h3>
克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此应该由集合类的具体实现来决定如何被克隆或者是序列化。
<h3 id="为何Map接口不继承Collection接口-">为何 Map 接口不继承 Collection 接口？/strong></h3>
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，它提抽取 key 或 value 列表集合的方法，但是它不适合“一组对象”规范。
<h3 id="Iterator是什么-">Iterator 是什么？</h3>
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过程中移除元素。
<h3 id="Iterator和ListIterator的区别是什么-">Iterator 和 ListIterator 的区别是什么？</h3>
下面列出了他们的区别： Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只用来遍历 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，等等。
<h3 id="Enumeration和Iterator接口的区别-">Enumeration 和 Iterator 接口的区别？</trong></h3>
Enumeration 速度是 Iterator 的 2 倍，同时占用更少的内存。但是，Iterator 远远比numeration 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。
<h3 id="为何没有像Iterator-add--这样的方法-向集合中添加元素-">为何没有像 Iterator.dd()这样的方法，向集合中添加元素？</h3>
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。
<h3 id="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-">为何迭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？</h3>
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，每个继都要去实现它，这没有意义。
<h3 id="Iterater和ListIterator之间有什么区别-">Iterater 和 ListIterator 之间有什么区？</h3>
1、我们可以使用 Iterator 来遍历 Set 和 List 集合，而 ListIterator 只能遍历 List。<trong>
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、替换一个元素、获取前面或后面元素的索引位置。
<h3 id="遍历一个List有哪些不同的方式-">遍历一个 List 有哪些不同的方式？/h3>

<pre><code class="highlight-chroma">List<String> strList = new ArrayList<>();
//使用for-each循环
for(String obj : strL st){
  System.out.print n(obj);
}
//using iterator
Iterator<String gt; it = strList.iterator();
while(it.hasNext())

  String obj = it.ne t();
  System.out.print n(obj);
}
</code></pre>
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，会抛出 ConcurrentModificationException。
<h3 id="通过迭代器fail-fast属性-你明白了什么-">通过迭代器 fail-fast 属性，你明白了么？</h3>
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何改动。如果发现任何改动，它抛出 ConcurrentModificationException。Collection 中所有 Iterato 的实现都是按 fail-fast 来设计的（ConcurrentHashMap 和 CopyOnWriteArrayList 这类并发集合除外）。
<h3 id="fail-fast与fail-safe有什么区别-">fail-fast 与 fail-safe 有什么区别？<h3>
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。ja a.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会抛出这样的异常。
<h3 id="在迭代一个集合的时候-如何避免ConcurrentModificationException-">在迭代个集合的时候，如何避免 ConcurrentModificationException？</h3>
在遍历一个集合的时候，我们可以使用并发集合类来避免 ConcurrentModificationEx eption，比如使用 CopyOnWriteArrayList，而不是 ArrayList。
<h3 id="为何Iterator接口没有具体的实现-">为何 Iterator 接口没有具体的实现？</stro g></h3>
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。
<h3 id="UnsupportedOperationException是什么-">UnsupportedOperationExceptio 是什么？</h3>
UnsupportedOperationException 是用于表明操作不支持的异常。在 JDK 类中已被量运用，在集合框架 java.util.Collections.UnmodifiableCollection 将会在所有 add 和 remove 操中抛出这个异常。
<h3 id="在Java中-HashMap是如何工作的-">在 Java 中，HashMap 是如何工作的？</s rong></h3>
HashMap 在 Map.Entry 静态内部类实现中存储 key-value 对。HashMap 使用哈希法，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 ut 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 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 Ent y，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 默认初始容量是 32，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。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。
<h3 id="hashCode--和equals--方法有何重要性-">hashCode()和 equals()方法有何重性？</h3>
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，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 tru 。
<h3 id="我们能否使用任何类作为Map的key-">我们能否使用任何类作为 Map 的 key？</h3>
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：</trong>
（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(' ankaj'); //assume hashCode=1234 myHashMap.put(key, 'Value'); // 以下的代码会改变 key 的 ha hCode()和 equals()值 key.setName('Amit'); //assume new hashCode=7890 //下面会返回 null，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 myHashM p.get(new MyKey('Pankaj')); 那就是为何 String 和 Integer 被作为 HashMap 的 key 大量使用。 /p>
<h3 id="Map接口提供了哪些不同的集合视图-">Map 接口提供了哪些不同的集合视图？</h3>
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 m p 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 emove、Set.remove、removeAll、retainAll 和 clear 操作进行元素移除，从 map 中移除对应的映 。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 Collecti n 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 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 结果会变为未定义。集合支持通过 Iterator 的 Remove、Set.remove、removeAll、retainAll 和 cle r 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

。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、St.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。
<h3 id="HashMap和HashTable有何不同-">HashMap 和 HashTable 有何不同？</strog></h3>
（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，HshTable 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，它的顺序是不预知的。
（4）HashMap 提供对 key 的 Set 进行遍历，因此它是 fail-fast 的，但 HashTable 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，你应该用 CocurrentHashMap。
<h3 id="如何决定选用HashMap还是TreeMap-">如何决定选用 HashMap 还是 TreeMa？</h3>
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，也向 HashMap 中添加元素会更快，将 map 换为 TreeMap 进行有序 key 的遍历。
<h3 id="ArrayList和Vector有何异同点-">ArrayList 和 Vector 有何异同点？<h3>
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，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。**
<h3 id="Array和ArrayList有何区别-什么时候更适合用Array-">Array 和 ArrayList 有何别？什么时候更适合用 Array？</h3>
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，大部分情况下是存和遍历它们。**
2、** 对于遍历基本数据类型，尽管 Collections 使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。
<h3 id="ArrayList和LinkedList有何区别-">ArrayList 和 LinkedList 有何区别？</h3>
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，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。**
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。**
<h3 id="哪些集合类提供对元素的随机访问-">哪些集合类提供对元素的随机访问？</h3>
ArrayList、HashMap、TreeMap 和 HashTable 类提供对元素的随机访问。
<h3 id="EnumSet是什么-">EnumSet 是什么？</h3>
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(numSet s)。
<h3 id="哪些集合类是线程安全的-">哪些集合类是线程安全的？</h3>
Vector、HashTable、Properties 和 Stack 是同步类，所以它们是线程安全的，可以多线程环境下使用。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隆上，所以它们在多线程环境中是安全的。
<h3 id="并发集合类是什么-">并发集合类是什么？</h3>
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。迭代器被设计为 fail-fast 的，会抛出 ConcurrentModificationException。一部分类为：CopyO WriteArrayList、 ConcurrentHashMap、CopyOnWriteArraySet。
<h3 id="BlockingQueue是什么-">BlockingQueue 是什么？</h3>
Java.util.concurrent.BlockingQueue 是一个队列，在进行检索或移除一个元素的时，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。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 的实类中被处理了。
Java 提供了集中 BlockingQueue 的实现，比如 ArrayBlockingQueue、LinkedBlock ngQueue、PriorityBlockingQueue,、SynchronousQueue 等。
<h3 id="队列和栈是什么-列出它们的区别-">队列和栈是什么，列出它们的区别？</h3>
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。
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。
<h3 id="Collections类是什么-">Collections 类是什么？</h3>
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，比如折半搜索、排序、混编和逆序等。
<h3 id="Comparable和Comparator接口是什么-">Comparable 和 Comparator 接口什么？</h3>
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。/strong>
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

排序。这就是我们需要使用 Comparator 接口的情景，因为 Comparable.compareTo(Object o)方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，回正整数。
<h3 id="Comparable和Comparator接口有何区别-">Comparable 和 Comparator 接有何区别？</h3>
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。
<h3 id="我们如何对一组对象进行排序-">我们如何对一组对象进行排序？</h>
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 Compable）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。</>
<h3 id="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-">当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？</h3>
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 UnsupportedOperationExceptio。
<h3 id="我们如何从给定集合那里创建一个synchronized的集合-">我们如何从给定集合里创建一个 synchronized 的集合？</h3>
我们可以使用 Collections.synchronizedCollection(Collection c)根据指定集合来获一个 synchronized（线程安全的）集合。
<h3 id="集合框架里实现的通用算法有哪些-">集合框架里实现的通用算法有哪些？</strog></h3>
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、最大最小值。
<h3 id="大写的O是什么-举几个例子-">大写的 O 是什么？举几个例子？</h3>

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 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来查找需要的元素。
<h3 id="与Java集合框架相关的有哪些最好的实践-">与 Java 集合框架相关的有哪些最好实践？</h3>
1、** 根据需要选择正确的集合类型。比如，如果指了大小，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 TeeMap。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 CassCastException。**

****5、 ****使用 JDK 提供的不可变类作为 Map 的 key，可以避免自己实现 hashCod
()和 equals()。
6、** 尽可能使用 Collections 工具类，或者获取只
、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
**
<h3 id="TreeMap和TreeSet在排序时如何比较元素-Collections工具类中的sort--方法如何比较元素
">TreeMap 和 TreeSet 在排序时如何比较元素？Collections 工具类中的 sort()方法如何
较元素？</h3>
TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元
的 compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap 要求存放的键值对映
的键必须实现 Comparable 接口从而根据键对元素进行排序。Collections 工具类的 sort 方法有两
重载的形式，第一种要求传入的待排序容器中存放的对象比较实现 Comparable 接口以实现元素的
较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator
接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实
是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java 中对函数式编程的支持）。</str
ng>