**[遍历List集合的三种方法](https://www.cnblogs.com/lzq198754/p/5774593.html)**

方法一：  
超级for循环遍历  
for(String attribute : list) {  
 System.out.println(attribute);  
}  
方法二：  
对于ArrayList来说速度比较快, 用for循环, 以size为条件遍历:  
for(int i = 0 ; i < list.size() ; i++) {  
 system.out.println(list.get(i));  
}  
方法三：  
集合类的通用遍历方式, 从很早的版本就有, 用迭代器迭代  
Iterator it = list.iterator();  
while(it.hasNext()) {  
 System.ou.println(it.next);  
}

## [Java遍历Map对象的四种方式](https://www.cnblogs.com/fqfanqi/p/6187085.html)

方式一 这是最常见的并且在大多数情况下也是最可取的遍历方式。在键值都需要时使用。

|  |  |
| --- | --- |
| 1  2  3  4 | Map<Integer, Integer> map = new HashMap<Integer, Integer>();  for (Map.Entry<Integer, Integer> entry : map.entrySet()) {    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());  } |

方法二 在for-each循环中遍历keys或values。

如果只需要map中的键或者值，你可以通过keySet或values来实现遍历，而不是用entrySet。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Map<Integer, Integer> map = new HashMap<Integer, Integer>();  //遍历map中的键  for (Integer key : map.keySet()) {    System.out.println("Key = " + key);  }  //遍历map中的值  for (Integer value : map.values()) {    System.out.println("Value = " + value);  } |

该方法比entrySet遍历在性能上稍好（快了10%），而且代码更加干净。

方法三使用Iterator遍历

使用泛型：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Map<Integer, Integer> map = new HashMap<Integer, Integer>();  Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator();  while (entries.hasNext()) {    Map.Entry<Integer, Integer> entry = entries.next();    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());  } |

不使用泛型：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Map map = new HashMap();  Iterator entries = map.entrySet().iterator();  while (entries.hasNext()) {    Map.Entry entry = (Map.Entry) entries.next();    Integer key = (Integer)entry.getKey();    Integer value = (Integer)entry.getValue();    System.out.println("Key = " + key + ", Value = " + value);  } |

你也可以在keySet和values上应用同样的方法。

该种方式看起来冗余却有其优点所在。首先，在老版本java中这是惟一遍历map的方式。另一个好处是，你可以在遍历时调用iterator.remove()来删除entries，另两个方法则不能。根据javadoc的说明，如果在for-each遍历中尝试使用此方法，结果是不可预测的。

从性能方面看，该方法类同于for-each遍历（即方法二）的性能。

方法四、通过键找值遍历（效率低）

|  |  |
| --- | --- |
| 1  2  3  4 | Map<Integer, Integer> map = new HashMap<Integer, Integer>();  for (Integer key : map.keySet()) {    Integer value = map.get(key);    System.out.println("Key = " + key + ", Value = " + value); |

作为方法一的替代，这个代码看上去更加干净；但实际上它相当慢且无效率。因为从键取值是耗时的操作（与方法一相比，在不同的Map实现中该方法慢了20%~200%）。如果你安装了FindBugs，它会做出检查并警告你关于哪些是低效率的遍历。所以尽量避免使用。

总结

如果仅需要键(keys)或值(values)使用方法二。如果你使用的语言版本低于java 5，或是打算在遍历时删除entries，必须使用方法三。否则使用方法一(键值都要)。

**ArrayList实现原理要点概括**

1.ArrayList是List接口的可变数组非同步实现，并允许包括null在内的所有元素。

2.底层使用数组实现

3.该集合是可变长度数组，数组扩容时，会将老数组中的元素重新拷贝一份到新的数组中，每次数组容量增长大约是其容量的1.5倍，这种操作的代价很高。

4.采用了Fail-Fast机制，面对并发的修改时，迭代器很快就会完全失败，而不是冒着在将来某个不确定时间发生任意不确定行为的风险

5.remove方法会让下标到数组末尾的元素向前移动一个单位，并把最后一位的值置空，方便GC

实现原理：

1. 底层使用数组实现
2. 构造方法

 ArrayList提供了三种方式的构造器，可以构造一个**默认初始容量为10**的空列表，构造一个指定初始容量的空列表以及构造一个包含指定集合的​​元素的列表，这些元素按照该集合的迭代器返回它们的顺序排列的；

此处代码块

**LinkedList实现原理要点概括**

1.LinkedList是List接口的双向链表非同步实现，并允许包括null在内的所有元素。

2.底层的数据结构是基于双向链表的，该数据结构我们称为节点

3.双向链表节点对应的类Node的实例，Node中包含成员变量：prev，next，item。其中，prev是该节点的上一个节点，next是该节点的下一个节点，item是该节点所包含的值。

4.它的查找是分两半查找，先判断index是在链表的哪一半，然后再去对应区域查找，这样最多只要遍历链表的一半节点即可找到

LinkedList提供了两个构造方法。

第一个构造方法不接受参数，将header实例的previous和next全部指向header实例（注意，这个是一个双向循环链表，如果不是循环链表，空链表的情况应该是header节点的前一节点和后一节点均为null），这样整个链表其实就只有header一个节点，用于表示一个空的链表。

第二个构造方法接收一个Collection参数c，调用第一个构造方法构造一个空的链表，之后通过addAll将c中的元素全部添加到链表中。

1.元素添加add();

2.构造方法中的调用了addAll(Collection<? extends E> c)方法，而在addAll(Collection<? extends E> c)方法中仅仅是将size当做index参数调用了addAll(int index,Collection<? extends E> c)方法。

3. addFirst() 4.addLast()

5. clear()

6.contains() 数据包含

7. indexOf(Object o)

8. remove() 删除数据

9. 数据获取get() ;( 首先判断位置信息是否合法（大于等于0，小于当前LinkedList实例的Size），然后遍历到具体位置，获得节点的业务数据（element）并返回)

10. 数据复制clone()与toArray()

11. 遍历数据：Iterator()

    LinkedList的Iterator

    除了Entry，LinkedList还有一个内部类：ListItr。

    ListItr实现了ListIterator接口，可知它是一个迭代器，通过它可以遍历修改LinkedList。

在LinkedList中提供了获取ListItr对象的方法：listIterator(int index)。

**HashMap实现原理要点概括**

1.HashMap是基于哈希表的Map接口的非同步实现，允许使用null值和null键，但不保证映射的顺序。

2.底层使用数组实现，数组中每一项是个单向链表，即数组和链表的结合体；当链表长度大于一定阈值时，链表转换为红黑树，这样减少链表查询时间。

3.HashMap在底层将key-value当成一个整体进行处理，这个整体就是一个Node对象。HashMap底层采用一个Node[]数组来保存所有的key-value对，当需要存储一个Node对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Node时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Node。

4.HashMap进行数组扩容需要重新计算扩容后每个元素在数组中的位置，很耗性能

5.采用了Fail-Fast机制，通过一个modCount值记录修改次数，对HashMap内容的修改都将增加这个值。迭代器初始化过程中会将这个值赋给迭代器的expectedModCount，在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map，马上抛出异常

   Fail-Fast机制：

   我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

   这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

Fail-Fast机制：**Java代码  [收藏代码](javascript:void())**

1. HashIterator() {
2. expectedModCount = modCount;
3. **if** (size > 0) { // advance to first entry
4. Entry[] t = table;
5. **while** (index < t.length && (next = t[index++]) == **null**)
6. ;
7. }
8. }

   在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map：

   注意到modCount声明为volatile，保证线程之间修改的可见性。

HashMap的存取实现：

1. **public** V put(K key, V value) {
2. // HashMap允许存放null键和null值。
3. // 当key为null时，调用putForNullKey方法，将value放置在数组第一个位置。
4. **if** (key == **null**)
5. **return** putForNullKey(value);
6. // 根据key的keyCode重新计算hash值。
7. **int** hash = hash(key.hashCode());
8. // 搜索指定hash值在对应table中的索引。
9. **int** i = indexFor(hash, table.length);
10. // 如果 i 索引处的 Entry 不为 null，通过循环不断遍历 e 元素的下一个元素。
11. **for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {
12. Object k;
13. **if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
14. V oldValue = e.value;
15. e.value = value;
16. e.recordAccess(**this**);
17. **return** oldValue;
18. }
19. }
20. // 如果i索引处的Entry为null，表明此处还没有Entry。
21. modCount++;
22. // 将key、value添加到i索引处。
23. addEntry(hash, key, value, i);
24. **return** **null**;
25. }

当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

  addEntry(hash, key, value, i)方法根据计算出的hash值，将key-value对放在数组table的i索引处。addEntry 是 HashMap 提供的一个包访问权限的方法，代码如下：

1. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
2. // 获取指定 bucketIndex 索引处的 Entry
3. Entry<K,V> e = table[bucketIndex];
4. // 将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
5. table[bucketIndex] = **new** Entry<K,V>(hash, key, value, e);
6. // 如果 Map 中的 key-value 对的数量超过了极限
7. **if** (size++ >= threshold)
8. // 把 table 对象的长度扩充到原来的2倍。
9. resize(2 \* table.length);
10. }

**Hashtable实现原理要点概括**

1.Hashtable是基于哈希表的Map接口的同步实现，不允许使用null值和null键

2.底层使用数组实现，数组中每一项是个单链表，即数组和链表的结合体

3.Hashtable在底层将key-value当成一个整体进行处理，这个整体就是一个Entry对象。Hashtable底层采用一个Entry[]数组来保存所有的key-value对，当需要存储一个Entry对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

4.synchronized是针对整张Hash表的，即每次锁住整张表让线程独占

**ConcurrentHashMap实现原理要点概括**

1.ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。

2.它使用了多个锁来控制对hash表的不同段进行的修改，每个段其实就是一个小的hashtable，它们有自己的锁。只要多个并发发生在不同的段上，它们就可以并发进行。

3.ConcurrentHashMap在底层将key-value当成一个整体进行处理，这个整体就是一个Entry对象。Hashtable底层采用一个Entry[]数组来保存所有的key-value对，当需要存储一个Entry对象时，会根据key的hash算法来决定其在数组中的存储位置，在根据equals方法决定其在该数组位置上的链表中的存储位置；当需要取出一个Entry时，也会根据key的hash算法找到其在数组中的存储位置，再根据equals方法从该位置上的链表中取出该Entry。

4.与HashMap不同的是，ConcurrentHashMap使用多个子Hash表，也就是段(Segment)

5.ConcurrentHashMap完全允许多个读操作并发进行，读操作并不需要加锁。如果使用传统的技术，如HashMap中的实现，如果允许可以在hash链的中间添加或删除元素，读操作不加锁将得到不一致的数据。ConcurrentHashMap实现技术是保证HashEntry几乎是不可变的。

**HashSet实现原理要点概括**

对于LinkedHashSet而言，它继承与HashSet、又基于LinkedHashMap来实现的。LinkedHashSet底层使用LinkedHashMap来保存所有元素，它继承与HashSet，其所有的方法操作上又与HashSet相同。

1. isEmpty();
2. contains();
3. add();
4. remove();
5. clear();
6. clone();