**迭代器模式：就是提供一种方法对一个容器对象中的各个元素进行访问，而又不暴露该对象容器的内部细节。**

**概述**

　　Java集合框架的集合类，我们有时候称之为**容器**。容器的种类有很多种，比如ArrayList、LinkedList、HashSet...，每种容器都有自己的特点，ArrayList底层维护的是一个数组；LinkedList是链表结构的；HashSet依赖的是哈希表，每种容器都有自己特有的数据结构。

　　因为容器的内部结构不同，很多时候可能不知道该怎样去遍历一个容器中的元素。所以为了使对容器内元素的操作更为简单，Java引入了迭代器模式！

　　把访问逻辑从不同类型的集合类中抽取出来，从而避免向外部暴露集合的内部结构。

**对于数组我们使用的是下标来进行处理的：**

int array[] = new int[3];

for (int i = 0; i < array.length; i++) {

System.out.println(array[i]);

}

**对ArrayList的处理：**

List<String> list = new ArrayList<String>();

for(int i = 0 ; i < list.size() ; i++){

String string = list.get(i);

}

　　对于这两种方式，我们总是都知道它的内部结构，访问代码和集合本身是紧密耦合的，无法将访问逻辑从集合类和客户端代码中分离出来。不同的集合会对应不同的遍历方法，客户端代码无法复用。在实际应用中如何将上面两个集合整合是相当麻烦的。所以才有***Iterator，它总是用同一种逻辑来遍历集合。使得客户端自身不需要来维护集合的内部结构，所有的内部状态都由Iterator来维护。客户端不用直接和集合进行打交道，而是控制Iterator向它发送向前向后的指令，就可以遍历集合。***

1. **java.util.Iterator**

下面让我们看看Java中的Iterator接口是如何实现的

　　在Java中Iterator为一个接口，它只提供了迭代的基本规则。在JDK中它是这样定义的：对Collection进行迭代的迭代器。迭代器取代了Java Collection Framework中的Enumeration。迭代器与枚举有两点不同:

　　1. 迭代器在迭代期间可以从集合中移除元素。

　　2. 方法名得到了改进，Enumeration的方法名称都比较长。

其接口定义如下:

package java.util;

public interface Iterator<E> {

boolean hasNext(); //判断是否存在下一个对象元素

E next(); //获取下一个元素

void remove(); //移除元素

}

1. **Iterable**

　　Java中还提供了一个Iterable接口，**Iterable接口实现后的功能是“返回”一个迭代器**，我们常用的实现了该接口的子接口有:Collection<E>、List<E>、Set<E>等。该接口的iterator()方法返回一个标准的Iterator实现。**实现Iterable接口允许对象成为Foreach语句的目标。就可以通过foreach语句来遍历你的底层序列**。

　　Iterable接口包含一个能产生Iterator对象的方法，并且Iterable被foreach用来在序列中移动。因此如果创建了实现Iterable接口的类，都可以将它用于foreach中。

Iterable接口的具体实现:

Package java.lang;

import java.util.Iterator;

public interface Iterable<T> {

Iterator<T> iterator();

}

**使用迭代器遍历集合:**

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

list.add("张三1");

list.add("张三2");

list.add("张三3");

list.add("张三4");

List<String> linkList = new LinkedList<String>();

linkList.add("link1");

linkList.add("link2");

linkList.add("link3");

linkList.add("link4");

Set<String> set = new HashSet<String>();

set.add("set1");

set.add("set2");

set.add("set3");

set.add("set4");

//使用迭代器遍历ArrayList集合

Iterator<String> listIt = list.iterator();

while(listIt.hasNext()){

System.out.println(listIt.hasNext());

}

//使用迭代器遍历Set集合

Iterator<String> setIt = set.iterator();

while(setIt.hasNext()){

System.out.println(listIt.hasNext());

}

//使用迭代器遍历LinkedList集合

Iterator<String> linkIt = linkList.iterator();

while(linkIt.hasNext()){

System.out.println(listIt.hasNext());

}

}

**使用foreach遍历集合:**

List<String> list = new ArrayList<String>();

list.add("张三1");

list.add("张三2");

list.add("张三3");

list.add("张三4");

for (String string : list) {

System.out.println(string);

}

　　可以看出使用foreach遍历集合的优势在于代码更加的简洁，更不容易出错，不用关心下标的起始值和终止值。

**3、Iterator遍历时不可以删除集合中的元素问题**

**在使用Iterator的时候禁止对所遍历的容器进行改变其大小结构的操作。**例如: 在使用Iterator进行迭代时，如果对集合进行了**add操作**就会出现ConcurrentModificationException异常。

List<String> list = new ArrayList<String>();

list.add("张三1");

list.add("张三2");

list.add("张三3");

list.add("张三4");

//使用迭代器遍历ArrayList集合

Iterator<String> listIt = list.iterator();

while(listIt.hasNext()){

Object obj = listIt.next();

if(obj.equals("张三3")){

list.remove(obj); //可以删除是因为

}

}

　　因为在你迭代之前，迭代器已经被通过list.itertor()创建出来了，如果在迭代的过程中，又对list进行了改变其容器大小的操作，那么Java就会给出异常。因为此时Iterator对象已经无法主动同步list做出的改变，Java会认为你做出这样的操作是线程不安全的，就会给出善意的提醒（抛出ConcurrentModificationException异常）

Iterator的实现源码:

[IMG_264](https://www.cnblogs.com/zyuze/p/javascript:void(0);)

1 private class Itr implements Iterator<E> { 2 int cursor; // index of next element to return 3 int lastRet = -1; // index of last element returned; -1 if no such 4 int expectedModCount = modCount; 5 6 public boolean hasNext() { 7 return cursor != size; 8 } 9 10 @SuppressWarnings("unchecked")11 public E next() {12 checkForComodification();13 int i = cursor;14 if (i >= size)15 throw new NoSuchElementException();16 Object[] elementData = ArrayList.this.elementData;17 if (i >= elementData.length)18 throw new ConcurrentModificationException();19 cursor = i + 1;20 return (E) elementData[lastRet = i];21 }22 23 public void remove() {24 if (lastRet < 0)25 throw new IllegalStateException();26 checkForComodification();27 28 try {29 ArrayList.this.remove(lastRet);30 cursor = lastRet;31 lastRet = -1;32 expectedModCount = modCount;33 } catch (IndexOutOfBoundsException ex) {34 throw new ConcurrentModificationException();35 }36 }37 38 final void checkForComodification() {39 if (modCount != expectedModCount)40 throw new ConcurrentModificationException();41 }42 }

[IMG_265](https://www.cnblogs.com/zyuze/p/javascript:void(0);)

　　通过查看源码发现原来检查并抛出异常的是checkForComodification()方法。在ArrayList中modCount是当前集合的版本号，每次修改(增、删)集合都会加1；expectedModCount是当前迭代器的版本号，在迭代器实例化时初始化为modCount。我们看到在checkForComodification()方法中就是在验证modCount的值和expectedModCount的值是否相等，所以当你在调用了ArrayList.add()或者ArrayList.remove()时，只更新了modCount的状态，而迭代器中的expectedModCount未同步，因此才会导致再次调用Iterator.next()方法时抛出异常。但是为什么使用Iterator.remove()就没有问题呢？通过源码的第32行发现，在Iterator的remove()中同步了expectedModCount的值，所以当你下次再调用next()的时候，检查不会抛出异常。

　　使用该机制的主要目的是为了实现ArrayList中的快速失败机制（fail-fast），在Java集合中较大一部分集合是存在快速失败机制的。

　　快速失败机制产生的条件:当多个线程对Collection进行操作时，若其中某一个线程通过Iterator遍历集合时，该集合的内容被其他线程所改变，则会抛出ConcurrentModificationException异常。

　　所以要保证在使用Iterator遍历集合的时候不出错误，就应该保证在遍历集合的过程中不会对集合产生结构上的修改。

 使用Foreach时对集合的结构进行修改会出现异常:

　　上面我们说了实现了Iterable接口的类就可以通过Foreach遍历，那是因为foreach要依赖于Iterable接口返回的Iterator对象，所以从本质上来讲，Foreach其实就是在使用迭代器，在使用foreach遍历时对集合的结构进行修改，和在使用Iterator遍历时对集合结构进行修改本质上是一样的。所以同样的也会抛出异常，执行快速失败机制。

　　foreach是JDK1.5新增加的一个循环结构，foreach的出现是为了简化我们遍历集合的行为。

 for循环与迭代器的对比:

　　\* 效率上各有各的优势:

　　　　> ArrayList对随机访问比较快，而for循环中使用的get()方法，采用的即是随机访问的方法，因此在ArrayList里for循环快。

　　　　> LinkedList则是顺序访问比较快，Iterator中的next()方法采用的是顺序访问方法，因此在LinkedList里使用Iterator较快。

　　　　> 主要还是要依据集合的数据结构不同的判断。