Java 基础（3）：加强型 for 循环与 Iterator

从 JDK1.5 起，增加了加强型的 for 循环语法，也被称为“for-Each 循环”。加强型循环在操作数组与集合方面增加了很大的方便性。那么，加强型 for 循环是怎么解析的呢？同时，这是不是意味着基本 for 循环就会被取代呢？

语法：

```
for(var item:items){//var 代表各钟类型
    //相关操作
}
```

我们先来看一下数组中的 for-Each 循环的使用；

```
String str[]= new String[]{"1","2","3"};
    //普通 for 循环
    for(int i=0;i<str.length;i++){
        String item = str[i];
        item += "str";
        System.out.println(item);
    }
    //加强型 for 循环
    for(String item:str){
        item += "str";
        System.out.println(item);
    }
```

通过比较上面例子中的两种类型的 for 循环，可以看出，for-Each 循环编写起来更加简单，更加方便程序员。因此，在程序中，应该多使用加强型循环。

回答一下上面提出的两个问题：

1、编译器是怎么处理数组中的 for-Each 循环的？

事实上，在数组中的 for-Each 最终会被编译器处理成一个普通的 for 循环，也就是说 for-Each 循环是完全与普通 for 循环等价的，没有任何特殊的命令。可以通过反编译来验证，很简单，此处不再多说。

2、在数组中，for-Each 循环能否完全替代普通 for 循环？

答案是否定的。虽然 for-Each 写起来方便，但也有以下几个局限性：

- 只能对元素进行顺序的访问；
- 只能访问数组或集合中的所有元素；
- 循环中没有当前的索引，无法对指定的元素操作。如更换当前索引位置的元素。

数组的加强型的 for-Each 循环很简单，我们再来看一下集合中的 for-Each 循环又是怎么样的。我们都知道集合中的遍历都是通过迭代（iterator）完成的。也许有人说，也可以按

照下面的方式来遍历集合，不一定非要使用迭代：

```
List<String> list = new LinkedList<String>();
    list.add("a");
    list.add("b");
    list.add("c");
    for(int i=0;i<list.size();i++){
        String item = list.get(i);
        System.out.println(item);
    }
```

然而，这种方式对于基于链表实现的 List 来说，是比较耗性能的，因为 get(int i)方法包含了一个循环，而且这个循环就是迭代遍历一次 List，直到遇到第 i 个元素，才停止循环，返回第 i 个元素。对于数量小，遍历不频繁的 List 来说，开销可以忽略。否则，开销将不容忽视。所以，正确集合遍历是使用迭代器 Iterator 来遍历的：

```
List<String> list = new LinkedList<String>();
    list.add("a");
    list.add("b");
    list.add("c");
    //获取集合的迭代器
    Iterator<String> itor = list.iterator();
    //集合的普通 for 循环
    for(;itor.hasNext();){//相当于 while(itor.hasNext())
        String item = itor.next();
        System.out.println(item);
    }
```

再看看对应的 for-Each 循环的例子：

```
List<String> list = new LinkedList<String>();
    list.add("a");
    list.add("b");
    list.add("c");
    for(String item:list){//for-Each
        System.out.println(item);
    }
```

可以看出，for-Each 循环比普通 for 循环要简洁很多。我们依旧回答上面的两个问题：

- **编译器是如何处理集合中的 for-Each 循环的？**

```
public static void main(String args[])
    {
        List list = new LinkedList();
        list.add("aa");
```

```
        list.add("bb");
        for(String item:list)
        {
            if("bb".equals(item))
                list.add("cc");
        }
    }
```

我们看一下上面例子的反编译代码：

```
   public static void main(String args[]){
        List list = new LinkedList();
        list.add("aa");
        list.add("bb");
        for(Iterator iterator = list.iterator(); iterator.hasNext();)
        {
            String item = (String)iterator.next();
            if("bb".equals(item))
                list.add("cc");
        }
    }
```

与数组类似，编译器最终也就是将集合中的 for-Each 循环处理成集合的普通 for 循环。而集合的 Collection 接口通过扩展 Iterable 接口来提供 iterator()方法。那么我们换一个角度，是不是只要实现 Iterable 接口，提供 iterator()方法，也可以使用 for-Each 循环呢？来看个例子：

```
class  MyList<T> implements Iterable<T>{
    private ArrayList<T> list = new ArrayList<>();
    public void addId(T id){
        list.add(id);
    }
    public boolean removeId(T id){
        return list.remove(id);
    }
    @Override
    public Iterator<T> iterator() {//扩展自 Iterable 接口
           //为了简单起见，就直接使用已有的迭代器
        return list.iterator();
    }
    public static void main(String[] args) {
        MyList<String> myList = new MyList<>();
        myList.addId("666999");
```

```
        myList.addId("973219");
            //for-Each
        for(String item:myList){
            System.out.println(item);
        }
    }
}
```

上面的例子编译通过，并且运行无误。所以，**只要实现了 Iterable 接口的类，都可以使用 for-Each 循环来遍历。**

集合迭代的陷阱

集合循环遍历时所使用的迭代器 Iterator 有一个要求：**在迭代的过程中，除了使用迭代器 (如:Iterator.remove()方法) 对集合增删元素外，是不允许直接对集合进行增删操作。**否则将会抛出 ConcurrentModificationException 异常。所以，**由于集合的 for-Each 循环本质上使用的还是 Iterator 来迭代，因此也要注意这个陷阱。**for-Each 循环很隐蔽地使用了 Iterator，导致程序员很容易忽略掉这个细节，所以一定要注意。看下面的例子，for-Each 循环中修改了集合。

```
public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("aa");
        list.add("bb");
        for (String item : list) {//for-Each
            if ("bb".equals(item)) {
                list.add("cc"); //直接操作 list
            }
        }
    }
```

运行抛出异常:

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.LinkedList$ListItr.checkForComodification(Unknown
        at java.util.LinkedList$ListItr.next(Unknown Source)
        at concurrentTest.Test_42.main(Test_42.java:12)
```


上面仅仅是 **单线程** 下的情况，如果你有并发编程的基础的话，就会知道：在 多线程 的环境中，线程是交替运行的（时间片轮转调度）。这就意味着，如果有两个线程 A、B，线程 A 对集合使用 Iterator 迭代遍历，线程 B 则对集合进行增删操作。线程 A、B 一旦交替运行，就会出现在迭代的同时对集合增删的效果，也会抛出异常。解决办法就是加锁变成原子操作，多线程在这里不是本文重点，不多说了。

同样也是不能的。集合中的 for-Each 循环的局限性与数组的 for-Each 循环是一样的。集合的 for-Each 循环是不能对集合进行增删操作、也不能获取索引。而集合的普通 for 循环可以使用的迭代器提供了对集合的增删方法 (如：Iterator.remove，ListIterator.add())，获

取索引的方法（如：ListIterator.nextIndex()、ListIterator.previousIndex()）；

我们来分析一下 Iterator 源码，主要看看为什么在集合迭代时，修改集合可能会抛出 ConcurrentModificationException 异常。以 ArrayList 中实现的 Iterator 为例。

先来看一下 ArrayList.iterator()方法，如下：

```
public Iterator<E> iterator() {
        return new Itr();
}
```

iterator()方法直接创建了一个类 Itr 的对象。那就接着看 Itr 类的定义吧！发现 Itr 其实是 ArrayList 的内部类，实现了 Iterator 接口。

```
/**
     * An optimized version of AbstractList.Itr
     */
    private class Itr implements Iterator<E> {
        int cursor;       // 当前的索引值，index of next element to return
        int lastRet = -1; // index of last element returned; -1 if no such
        int expectedModCount = modCount;
        public boolean hasNext() {
            return cursor != size;
        }
        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            //ArrayList 的底层数组
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }
        public void remove() {
            if (lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();
            try {
                ArrayList.this.remove(lastRet);
                cursor = lastRet;
```

```
            lastRet = -1;
            //再次更新 expectedModCount
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
    @Override
    @SuppressWarnings("unchecked")
    public void forEachRemaining(Consumer<? super E> consumer) {
        Objects.requireNonNull(consumer);
        final int size = ArrayList.this.size;
        int i = cursor;
        if (i >= size) {
            return;
        }
        final Object[] elementData = ArrayList.this.elementData;
        if (i >= elementData.length) {
            throw new ConcurrentModificationException();
        }
        while (i != size && modCount == expectedModCount) {
            consumer.accept((E) elementData[i++]);
        }
        // update once at end of iteration to reduce heap write traffic
        cursor = i;
        lastRet = i - 1;
        checkForComodification();
    }
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}
```

ArrayList.this.elementData 是 ArrayList 的底层数组，上面这些方法都很简单，都是对 ArrayList.this.elementData 这个底层数组进行操作。

重点看一下 checkForComodification（）方法，这个方法就是用来抛出 ConcurrentModificationException 异常，这个方法也很简单，就是判断 modCount 与 expectedModCount 是否相等。modCount 存储的 AarryList 中的元素个数。而 expectedModCount 则是对象创建时将 modCount 的值赋给它，也就是说

expectedModCount 存储的是迭代器创建时元素的个数。那么 checkForComodification( ) 方法其实在比较迭代期间，ArrayList 元素的个数 是否发生了改变，如果改变了，就抛出异常。注意一下，expectedModCount 除了在声明时赋值外，也在 remove()方法中更新了一次。

- 无论是在数组中还是在集合中，for-Each 加强型 for 循环都是它们各自的普通 for 循环的一种“简写方式”，即两者意思上是等价的，但前者方便简单，建议多使用。
- for-Each 循环不能完全代替普通 for 循环，因为 for-Each 有一定的局限性。
- for-Each 循环只能用于 数组、Iterable 类型（包括集合）。
- 集合中的 for-Each 循环本质上使用了 Iterator 迭代器，所以要注意 Iterator 迭代陷阱（单线程和多线程都有问题）。