for循环和foreach的区别

关于for循环和foreach的区别，你真的知道，用了那么多年使用起来已经很熟悉了，可突然问我讲讲这两的区别，一下还真把我给卡住了一下，下面从源码的角度简单分析一下吧；

for循环的使用

for循环通过下标的方式，对集合中指定位置进行操作，每次遍历会执行判断条件 i<list.size()，满足则继续执行，执行完一次i++；

```
for(int i=0;i<list.size();i++)
{
   System.out.println(i + ":" + list.get(i));
}
```

也就是说，即使在for循环中对list的元素remove和add也是可以的,因为添加或删除后list中元素个数变化，继续循环会再次判断i<list.size()； 也就是说list.size()值也发生了变化，所以是可行的，具体操作如下代码

```
for (int i = 0; i < list.size(); i++) {
   if (i == 3) {
    list.add("中间插入的一个字符串");
   }
   if (i == 5) {
    {
    list.remove(6);
   }
   }
   System.out.println(i + ":" + list.get(i));
  }
```

增强for循环：foreach循环的原理

同样地，使用foreach遍历上述集合，注意foreach是C#中的写法，在Java中写法依然是for （int i : list）

写法for(String str : list)

查看文档可知，foreach除了可以遍历数组，还可以用于遍历所有实现了Iterable<T>接口的对象。

用普通for循环的方式模拟实现一个foreach,由于List实现了Iterable<T>,

过程如下：首先通过iterator()方法获得一个集合的迭代器，然后每次通过游标的形式依次判断是否有下一个元素，如果有通过 next()方法则可以取出。 注意：执行完next()方法，游标向后移一位，只能后移，不能前进。

用传统for循环的方式模拟 增强for循环

```
for (Iterator<String> iterator = list.iterator(); iterator.hasNext();) {
    System.out.println(iterator.next());
}
```


和for循环的区别在于，它对索引的边界值只会计算一次。所以在foreach中对集合进行添加或删掉会导致错误，抛出异常

java.util.ConcurrentModificationException

```
private static void testForeachMethod(ArrayList<String> list) {
  int count = 0; // 记录index
  for (String str : list) {
   System.out.println(str);
   count++;
   if (count == 3) {
    // foreach中修改集合长度会抛出异常
    // list.add("foreach中插入的ABC");
   }
  }
 }
```

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
        at java.util.AbstractList$Itr.next(AbstractList.java:343)
        at com.jichu.ArrayListDemo.testForeachMethod(ArrayListDemo.java:35)
        at com.jichu.ArrayListDemo.main(ArrayListDemo.java:14)
```

具体可以从源码的角度进行理解

1.首先是调用iterator()方法获得一个集合迭代器

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

```
private class Itr implements Iterator<E> {
/**
 * Index of element to be returned by subsequent call to next.
 */
int cursor = 0;

/**
 * Index of element returned by most recent call to next or
 * previous.  Reset to -1 if this element is deleted by a call
 * to remove.
 */
int lastRet = -1;

/**
 * The modCount value that the iterator believes that the backing
 * List should have.  If this expectation is violated, the iterator
 * has detected concurrent modification.
 */
int expectedModCount = modCount;
```

初始化时

expectedModCount记录修改后的个数，当迭代器能检测到expectedModCount是否有过修改

```
 public void add(E e) {
         checkForComodification();

     try {
     AbstractList.this.add(cursor++, e);
     lastRet = -1;
     expectedModCount = modCount;
     } catch (IndexOutOfBoundsException ex) {
     throw new ConcurrentModificationException();
     }
 }
 }
```

```
final void checkForComodification() {
    if (modCount != expectedModCount)
    throw new ConcurrentModificationException();
}
}
```

在创建迭代器之后，除非通过迭代器自身的 remove 或 add 方法从结构上对列表进行修改，否则在任何时间以任何方式对列表进行修改，迭代器都会抛出 ConcurrentModificationException。因此，面对并发的修改，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。

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误的做法：迭代器的快速失败行为应该仅用于检测 bug。

https://blog.csdn.net/lang791534167/article/details/50414586

foreach对集合的遍历是调用他的迭代器

对数组的遍历是调用他的循环

foreach遍历list集合删除某些元素一定会报错吗，来，先上一段代码：

1）报错啦

```
List list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");
        for (String item : list) {
            if (item.equals("3")) {
                System.out.println(item);
                list.remove(item);
            }
        }
        System.out.println(list.size());
```

理所应当，控制台就愉快的报出了java.util.ConcurrentModificationException。
这是怎么回事，然后去看了看这个异常，才发现自己果然还是太年轻啊。
我们都知道增加for循环即foreach循环其实就是根据list对象创建一个iterator迭代对象，用这个迭代对象来遍历list，相当于list对象中元素的遍历托管给了iterator，如果要对list进行增删操作，都必须经过iterator。
每次foreach循环时都有以下两个操作：
1.iterator.hasNext(); //判读是否有下个元素
2.item = iterator.next();//下个元素是什么，并把它赋给item。
首先，我们来看看这个异常信息是什么。

```
public boolean hasNext() {
            return cursor != size;
        }

        @SuppressWarnings("unchecked")
        public E next() {
            checkForComodification();//此处报错
            int i = cursor;
            if (i >= size)
                throw new NoSuchElementException();
            Object[] elementData = ArrayList.this.elementData;
            if (i >= elementData.length)
                throw new ConcurrentModificationException();
            cursor = i + 1;
            return (E) elementData[lastRet = i];
        }
    final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
```

可以看到是进入checkForComodification()方法的时候报错了，也就是说modCount != expectedModCount。具体的原因，是在于foreach方式遍历元素的时候，是生成iterator，然后使用iterator遍历。在生成iterator的时候，会保存一个expectedModCount参数，这个是生成iterator的时候List中修改元素的次数。如果你在遍历过程中删除元素，List中modCount就会变化，如果这个modCount和exceptedModCount不一致，就会抛出异常，这个是为了安全的考虑。看看list的remove源码：

```
public boolean remove(Object o) {
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
```

```
                fastRemove(index);
                return true;
            }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

看，并没有对expectedModCount进行任何修改，导致expectedModCount和modCount不一致，抛出异常。所以，遍历list删除元素一律用Iterator这样不会报错，如下：

```
Iterator it = list.iterator();
    while(it.hasNext()){
        if(it.next().equals("3")){
            it.remove();
        }
    }
```

看看Iterator的remove()方法的源码，是对expectedModCount重新做了赋值处理的，如下：

```
public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
            expectedModCount = modCount;//处理expectedModCount
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
```

这样的话保持expectedModCount = modCount相等，就不会报出错了。

2）是不是foreach所有的list删除操作都会报出这个错呢

其实不一定，有没有发现如果删除的元素是倒数第二个数的话，其实是不会报错的，为什么呢，来一起看看。

之前说了foreach循环会走两个方法hasNext() 和next()。如果不想报错的话，只要不进next()方法就好啦，看看hasNext()的方法。

```
public boolean hasNext() {
    return cursor != size;
}
```

那么就要求hasNext()的方法返回false了，即cursor == size。其中cursor是Itr类（Iterator子类）中的一个字段，用来保存当前iterator的位置信息，从0开始。cursor本身就是游标的意思，在数据库的操作中用的比较多。只要curosr不等于size就认为存在元素。由

于Itr是ArrayList的内部类，因此直接调用了ArrayList的size字段，所以这个字段的值是动态变化的，既然是动态变化的可能就会有问题出现了。
我们以上面的代码为例，当到倒数第二个数据也就是”4”的时候，cursor是4,然后调用删除操作，此时size由5变成了4，当再调用hasNext判断的时候，cursor==size,就会调用后面的操作直接退出循环了。我们可以在上面的代码添加一行代码查看效果：

```
1. for (String item : list) {
2.             System.out.println(item);
3.             if (item.equals("4")) {
4.                 list.remove(item);
5.             }
6. }
```

输出是：1 2 3 4
这样的话就可以看到执行到hasNext（）方法就退出了，也就不会走后面的异常了。
由此可以得出，用foreach删除list元素的时候只有倒数第二个元素删除不会报错，其他都会报错，所以用Iterator啦。
never too late!
参考：http://rongmayisheng.com/post/%E7%A0%B4%E9%99%A4%E8%BF%B7%E4%BF%A1java-util-arraylist%E5%9C%A8foreach%E5%BE%AA%E7%8E%AF%E9%81%8D%E5%8E%86%E6%97%B6%E5%8F%AF%E4%BB%A5%E5%88%A0%E9%99%A4%E5%85%83%E7%B4%
http://blog.csdn.net/Jywangkeep_/article/details/48754189