链滴

Exception in thread "main" java.util.ConcurrentModificationException

作者：yiburuxin
原文链接：https://ld246.com/article/1547867376678
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

遍历List是我们在写代码中经常碰到的，有时候我们会碰到想在遍历的途中删掉某些元素的需求，但不小心可能就会报快速失败错误（FastFail），这其实跟List当中的一些实现有关，下面我选取了两种况，来自知乎大神的回答，加上自己的疑惑和总结。

作者：RednaxelaFX

链接：https://www.zhihu.com/question/56586732/answer/149650876

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

总结下就是两种情况：一种是可以删除末尾的元素，另一种是在倒数第二时可以删除任何一个元素。

我觉得题主的首要问题是被自己写的代码（以及JDK的ArrayList那可恶的API）给坑了。

ArrayList上有两个版本的remove方法：

public E remove(int index)

public boolean remove(Object o)

题主很可能以为自己调用的是第二个版本，但实际上调用的是第一个版本——remove(3) 删除了位于尾的元素，而不是位于倒数第二的元素。

ArrayList.iterator() 返回出来的 Iterator，里面的hasNext()是不关心modification count的，而next(会去检查modification count：

List中的内部类

```
/**
* An optimized version of AbstractList.Itr
*/
private class Itr implements Iterator {
    int cursor;       // 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();
    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;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

原文链接：Exception in thread "main" java.util.ConcurrentModificationException

```
}
```

所以题主的那个程序实际做的事情就是：

- 通过 ArrayList.iterator() 得到了一个新的iterator，开始遍历
- list.remove(3)：删除了位于index 3的元素（Integer.valueOf(4) 得到的对象）
- 然后调用iterator.hasNext()，得到false，于是就退出了循环而没有去执行那个会检查modification ount的next()方法。

ArrayList的JavaDoc说：

The iterators returned by this class's iterator and listIterator methods are fail-fast: if the list is s ructurally modified at any time after the iterator is created, in any way except through the iter tor's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than r sking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking impossible to make any hard guarantees in the presence of unsynchronized concurrent modif cation. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. There ore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

没有特别指定Iterator里的哪些方法一定会根据fail-fast原则而抛异常。但Iterator.hasNext()的JavaD c说：

hasNext

boolean hasNext()

Returns true if the iteration has more elements. (In other words, returns true if next() would re urn an element rather than throwing an exception.)

Returns:true if the iteration has more elements

就这个规定来说，我觉得题主观察到的现象应该算是JDK实现的上的巧合：因为如果在这个位置调用n xt()的话会抛ConcurrentModificationException异常，所以hasNext()也要返回false，于是就好啦。

但JDK这个具体实现看起来还是有bug，应该让hasNext()也做checkForComodification()的不抛异 对应动作才对。

就这样。

还是未能解决下面的疑问：

在判断的是倒数第二个的时候，所有的元素都能删。

```
List list = new ArrayList();

list.add("a");

list.add("b");
```

```
list.add("c");
list.add("d");
list.add("e");
list.add("f");

Iterator it = list.iterator();
while (it.hasNext()) {
    String itt = (String) it.next();
    if (itt.equals("e")) {
        list.remove("a");
    }
}
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

下面是解释：

作者：xRay

链接：https://www.zhihu.com/question/56916067/answer/151995061

来源：知乎

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

为什么没抛异常，跟下代码其实就一清二楚了。

ArralyList有个版本号modCount，每次修改ArrayList时版本号就会往上加。Iterator里面也有个本号expectedModCount它的初始值就是modCount。只有expectedModCount != modCount会抛异常。所以printList里面的绝对不会抛异常。那问题就出在进行remove的那个迭代器上，首先代器是在什么时候比较这两个值呢？ 答案是在remove跟next的时候，也就是说hasNext它不会抛这个异常。

List中的删除方法：

```
/**
 * Removes the element at the specified position in this list.
```

```
 * Shifts any subsequent elements to the left (subtracts one from their
 * indices).
 *
 * @param index the index of the element to be removed
 * @return the element that was removed from the list
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public E remove(int index) {
    rangeCheck(index);
    modCount++;
    E oldValue = elementData(index);
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work
    return oldValue;
}
```

然后，看下题主的这段代码

```
Iterator iterator = list.iterator();
while(iterator.hasNext()) {
    Integer integer = iterator.next();
    if(integer == 2)
        list.remove(integer);
}
```

这里面调用了List的remove方法，所以它不会抛出ConcurrentModificationException。问题是remove之后为什么hasNext会返回false。我们看下hasNext方法

```
int cursor; // index of next element to return
```

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

看注释，cursor是指向一个元素的，也就是当list.remove(integer=2)的时候，cursor实际等于2。而lit.remove(integer)后size=3-1=2，所以hasNext返回false，也就不会执行next方法，所以也就不会出ConcurrentModificationException。

不过，这个问题，更有意思的ArrayList有两个remove方法

public E remove(int index) boolean remove(Object o)

当参数是Integer时会调用哪个？ 我在JLS中找到这么一段，意思是选择重载函数时不会优先考虑装箱拆箱

The first phase (§15.12.2.2) performs overload resolution without permitting boxing or unboxing conversion, or the use of variable arity method invocation. If no applicable method is found during this phase then processing continues to the second phase.