# Java | 如何正确地在遍历 List 时删除元素

最近在一个 Android 项目里遇到一个偶现的 java.util.ConcurrentModificationException 异常导致的崩溃，经过排查，导致异常的代码大概是这样的：

```
private List<XxxListener> listeners;

public void foo() {
    for (XxxListener listener : listeners) {
        listener.doSomething();
    }
}

public class XxxListener {
    public void doSomething() {
        // some code here
        if (...) {
            listeners.remove(this);
        }
    }
}
```

把函数调用展开一下就等效于：

```
for (XxxListener listener : listeners) {
    // some code here
    if (...) {
        listeners.remove(listener);
    }
}
```

这个异常之所以不是必现，是因为 `listeners.remove` 不是总被执行到。

我先直接说一下正确的写法吧，就是使用迭代器的写法：

```
Iterator<XxxListener> iterator = listeners.iterator();
while (iterator.hasNext()) {
    XxxListener listener = iterator.next();
    // some code here
    if (...) {
        iterator.remove();
    }
}
```

然后再进一步分析。

## 源码分析

先来从源码层面分析下上述 java.util.ConcurrentModificationException 异常是如何抛出的。

写一段简单的测试源码：

```
List<String> list = new ArrayList<>();
list.add(”Hello”);
list.add(”World”);
list.add(”Hi”);

for (String str : list) {
    list.remove(str);
}
```

执行抛出异常：

```
Exception in thread ”main” java.util.ConcurrentModificationException
at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:911)
at java.util.ArrayList$Itr.next(ArrayList.java:861)
```

由此可以推测，`for (String str : list)` 这种写法实际只是一个语法糖，编译器会将其转换为迭代器的写法：

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String str = iterator.next();
    // do something
}
```

这可以从反编译后的字节码得到验证：

```
36: invokeinterface #8,  1            // InterfaceMethod
java/util/List.iterator:()Ljava/util/Iterator;
41: astore_2
42: aload_2
43: invokeinterface #9,  1            // InterfaceMethod
java/util/Iterator.hasNext:()Z
48: ifeq          72
51: aload_2
52: invokeinterface #10,  1           // InterfaceMethod
java/util/Iterator.next:()Ljava/lang/Object;
57: checkcast     #11                 // class java/lang/String
60: astore_3
61: aload_1
62: aload_3
63: invokeinterface #12,  2           // InterfaceMethod
java/util/List.remove:(Ljava/lang/Object;)Z
```

```

那么，`iterator.next()` 里发生了什么导致了异常的抛出呢？ArrayList$Itr 类的源码如下：

```
 private class Itr implements Iterator<E> {
    int cursor;       // index of next element to return
    int lastRet = -1; // index of last element returned; -1 if no such
    int expectedModCount = modCount;

    public E next() {
        checkForComodification();
        // ...
    }

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

    // ...
 }
```

其中 `modCount` 是 ArrayList 类的成员，表示对 ArrayList 进行增删改的次数。`expectedModCount` 是 ArrayList$Itr 类的成员，初始值是迭代器创建时 ArrayList 的 `modCount` 的值。在每次调用 `next()` 时，都会检查 `modCount` 是否等于 `expectedModCount`，如果不等则抛出异常。

那为什么 `list.remove` 会导致 `modCount` 的值不等于 `expectedModCount`，而 `iterator.remove` 不会呢？

```
// ArrayList 的 remove 方法
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;
}

// ArrayList$Itr 的 remove 方法
public void remove() {
    if (lastRet < 0)
        throw new IllegalStateException();
    checkForComodification();

    try {
        ArrayList.this.remove(lastRet);
        // 注意这三行
        cursor = lastRet;
        lastRet = -1;
        expectedModCount = modCount;
    } catch (IndexOutOfBoundsException ex) {
        throw new ConcurrentModificationException();
    }
}
```

可以看到 `ArrayList#remove` 里 `modCount++`，但并不会修改到 Itr 的 `expectedModCount`——它们当然就不相等了。而 `ArrayList$Itr#remove` 在先调用了 `ArrayList#remove` 后，又将 `modCount` 的最新值赋给了 `modCount`，这样就保证了 `modCount` 和 `expectedModCount` 的一致性。

同时，`ArrayList$Itr#remove` 里还有一个 `cursor = lastRet`，实际上是将迭代器的游标做了修正，前移一位，以实现后续调用 `next()` 的行为正确。

## 小结

> 源码面前，了无秘密。

* 如果需要在遍历 List 时删除元素，应使用迭代器的写法，即 `iterator.remove()`；
* 在非遍历场景下，使用 `ArrayList#remove` 也没什么问题——同理，即使是遍历场景下，使用 `ArrayList#remove` 后马上 break 也 OK；
* 如果遍历时做的事情不多，`Collection#removeIf` 方法也是一个不错的选择（实际也是上述迭代器写法的封装）。

---

如果读完文章有收获，可以我的公众号「闷骚的程序员」并设为星标，随时阅读更多内容。

原文链接: https://juejin.cn/post/7362833213960994852