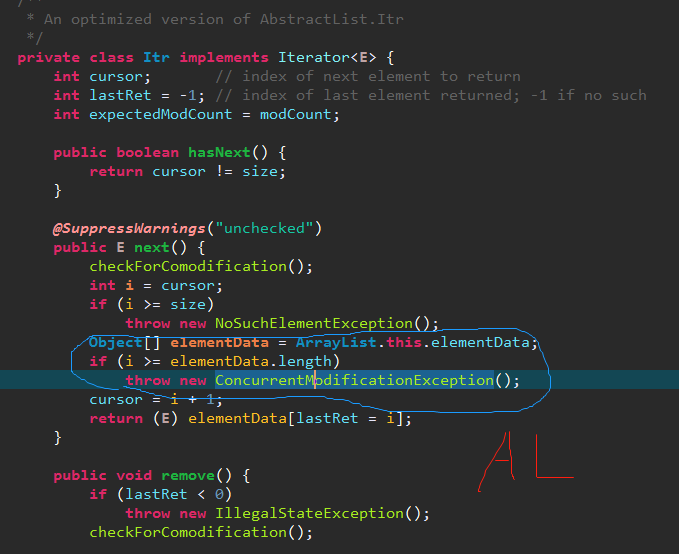
1 fail-fast简介

**fail-fast 机制是java集合(Collection)中的一种错误机制。**当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。  
例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程A访问集合时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。



在详细介绍fail-fast机制的原理之前，先通过一个示例来认识fail-fast。

2 fail-fast示例

示例代码：(**FastFailTest.java**)

1 import java.util.\*;

2 import java.util.concurrent.\*;

3

4 /\*

5 \* @desc java集合中Fast-Fail的测试程序。

6 \*

7 \* fast-fail事件产生的条件：当多个线程对Collection进行操作时，若其中某一个线程通过iterator去遍历集合时，该集合的内容被其他线程所改变；则会抛出ConcurrentModificationException异常。

8 \* fast-fail解决办法：通过util.concurrent集合包下的相应类去处理，则不会产生fast-fail事件。

9 \*

10 \* 本例中，分别测试ArrayList和CopyOnWriteArrayList这两种情况。ArrayList会产生fast-fail事件，而CopyOnWriteArrayList不会产生fast-fail事件。

11 \* (01) 使用ArrayList时，会产生fast-fail事件，抛出ConcurrentModificationException异常；定义如下：

12 \* private static List<String> list = new ArrayList<String>();

13 \* (02) 使用时CopyOnWriteArrayList，不会产生fast-fail事件；定义如下：

14 \* private static List<String> list = new CopyOnWriteArrayList<String>();

15 \*

16 \* @author skywang

17 \*/

18 public class FastFailTest {

19

20 private static List<String> list = new ArrayList<String>();

21 //private static List<String> list = new CopyOnWriteArrayList<String>();

22 public static void main(String[] args) {

23

24 // 同时启动两个线程对list进行操作！

25 new ThreadOne().start();

26 new ThreadTwo().start();

27 }

28

29 private static void printAll() {

30 System.out.println("");

31

32 String value = null;

33 Iterator iter = list.iterator();

34 while(iter.hasNext()) {

35 value = (String)iter.next();

36 System.out.print(value+", ");

37 }

38 }

39

40 /\*\*

41 \* 向list中依次添加0,1,2,3,4,5，每添加一个数之后，就通过printAll()遍历整个list

42 \*/

43 private static class ThreadOne extends Thread {

44 public void run() {

45 int i = 0;

46 while (i<6) {

47 list.add(String.valueOf(i));

48 printAll();

49 i++;

50 }

51 }

52 }

53

54 /\*\*

55 \* 向list中依次添加10,11,12,13,14,15，每添加一个数之后，就通过printAll()遍历整个list

56 \*/

57 private static class ThreadTwo extends Thread {

58 public void run() {

59 int i = 10;

60 while (i<16) {

61 list.add(String.valueOf(i));

62 printAll();

63 i++;

64 }

65 }

66 }

67

68 }

**运行结果**：  
运行该代码，抛出异常java.util.ConcurrentModificationException！即，产生fail-fast事件！

**结果说明**：  
(01) FastFailTest中通过 new ThreadOne().start() 和 new ThreadTwo().start() 同时启动两个线程去操作list。  
    **ThreadOne线程**：向list中依次添加0,1,2,3,4,5。每添加一个数之后，就通过printAll()遍历整个list。  
    **ThreadTwo线程**：向list中依次添加10,11,12,13,14,15。每添加一个数之后，就通过printAll()遍历整个list。  
(02) 当某一个线程遍历list的过程中，list的内容被另外一个线程所改变了；就会抛出ConcurrentModificationException异常，产生fail-fast事件。

3 fail-fast解决办法

fail-fast机制，是一种错误检测机制。**它只能被用来检测错误，因为JDK并不保证fail-fast机制一定会发生。**若在多线程环境下使用fail-fast机制的集合，建议使用“java.util.concurrent包下的类”去取代“java.util包下的类”。  
所以，本例中只需要将ArrayList替换成java.util.concurrent包下对应的类即可。  
即，将代码

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

替换为

private static List<String> list = new CopyOnWriteArrayList<String>();

则可以解决该办法。

4 fail-fast原理

产生fail-fast事件，是通过抛出ConcurrentModificationException异常来触发的。  
那么，ArrayList是如何抛出ConcurrentModificationException异常的呢?

我们知道，ConcurrentModificationException是在操作Iterator时抛出的异常。我们先看看Iterator的源码。ArrayList的Iterator是在父类AbstractList.java中实现的。代码如下：

1 package java.util;

2

3 public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {

4

5 ...

6

7 // AbstractList中唯一的属性

8 // 用来记录List修改的次数：每修改一次(添加/删除等操作)，将modCount+1

9 protected transient int modCount = 0;

10

11 // 返回List对应迭代器。实际上，是返回Itr对象。

12 public Iterator<E> iterator() {

13 return new Itr();

14 }

15

16 // Itr是Iterator(迭代器)的实现类

17 private class Itr implements Iterator<E> {

18 int cursor = 0;

19

20 int lastRet = -1;

21

22 // 修改数的记录值。

23 // 每次新建Itr()对象时，都会保存新建该对象时对应的modCount；

24 // 以后每次遍历List中的元素的时候，都会比较expectedModCount和modCount是否相等；

25 // 若不相等，则抛出ConcurrentModificationException异常，产生fail-fast事件。

26 int expectedModCount = modCount;

27

28 public boolean hasNext() {

29 return cursor != size();

30 }

31

32 public E next() {

33 // 获取下一个元素之前，都会判断“新建Itr对象时保存的modCount”和“当前的modCount”是否相等；

34 // 若不相等，则抛出ConcurrentModificationException异常，产生fail-fast事件。

35 checkForComodification();

36 try {

37 E next = get(cursor);

38 lastRet = cursor++;

39 return next;

40 } catch (IndexOutOfBoundsException e) {

41 checkForComodification();

42 throw new NoSuchElementException();

43 }

44 }

45

46 public void remove() {

47 if (lastRet == -1)

48 throw new IllegalStateException();

49 checkForComodification();

50

51 try {

52 AbstractList.this.remove(lastRet);

53 if (lastRet < cursor)

54 cursor--;

55 lastRet = -1;

56 expectedModCount = modCount;

57 } catch (IndexOutOfBoundsException e) {

58 throw new ConcurrentModificationException();

59 }

60 }

61

62 final void checkForComodification() {

63 if (modCount != expectedModCount)

64 throw new ConcurrentModificationException();

65 }

66 }

67

68 ...

69 }

复制代码

从中，我们可以发现在调用 next() 和 remove()时，都会执行 checkForComodification()。若 “**modCount 不等于 expectedModCount**”，则抛出ConcurrentModificationException异常，产生fail-fast事件。

要搞明白 fail-fast机制，我们就要需要理解什么时候“modCount 不等于 expectedModCount”！  
从Itr类中，我们知道 expectedModCount 在创建Itr对象时，被赋值为 modCount。通过Itr，我们知道：expectedModCount不可能被修改为不等于 modCount。所以，需要考证的就是modCount何时会被修改。

接下来，我们查看ArrayList的源码，来看看modCount是如何被修改的。

从中，我们发现：无论是add()、remove()，还是clear()，只要涉及到修改集合中的元素个数时，都会改变modCount的值。

接下来，我们再系统的梳理一下fail-fast是怎么产生的。步骤如下：  
(01) *新建了一个ArrayList，名称为arrayList。*  
(02) *向arrayList中添加内容。*  
(03)*新建一个“****线程a****”，并在“线程a”中****通过Iterator反复的读取arrayList的值****。*  
(04) *新建一个“****线程b****”，在“线程b”中****删除arrayList中的一个“节点A****”。*  
(05) 这时，就会产生有趣的事件了。  
       在某一时刻，“线程a”创建了arrayList的Iterator。此时“节点A”仍然存在于arrayList中，**创建arrayList时，expectedModCount = modCount**(假设它们此时的值为N)。  
       在“线程a”在遍历arrayList过程中的某一时刻，“线程b”执行了，并且“线程b”删除了arrayList中的“节点A”。“线程b”执行remove()进行删除操作时，在remove()中执行了“modCount++”，此时**modCount变成了N+1**！  
“线程a”接着遍历，当它执行到next()函数时，调用checkForComodification()比较“expectedModCount”和“modCount”的大小；而“expectedModCount=N”，“modCount=N+1”,这样，便抛出ConcurrentModificationException异常，产生fail-fast事件。

至此，**我们就完全了解了fail-fast是如何产生的！**  
即，当***多个线程对同一个集合进行操作的时候，某线程访问集合的过程中，该集合的内容被其他线程所改变(即其它线程通过add、remove、clear等方法，改变了modCount的值)；这时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。***

5 解决fail-fast的原理

上面，说明了“解决fail-fast机制的办法”，也知道了“fail-fast产生的根本原因”。接下来，我们再进一步谈谈java.util.concurrent包中是如何解决fail-fast事件的。  
还是以和ArrayList对应的CopyOnWriteArrayList进行说明。

从中，我们可以看出:

(01) 和ArrayList继承于AbstractList不同，CopyOnWriteArrayList没有继承于AbstractList，它仅仅只是实现了List接口。  
(02) ArrayList的iterator()函数返回的Iterator是在AbstractList中实现的；而CopyOnWriteArrayList是自己实现Iterator。  
(03) ArrayList的Iterator实现类中调用next()时，会“调用checkForComodification()比较‘expectedModCount’和‘modCount’的大小”；但是，CopyOnWriteArrayList的Iterator实现类中，没有所谓的checkForComodification()，更不会抛出ConcurrentModificationException异常！

为什么?

答案:新建COWIterator时，将集合中的元素保存到一个新的拷贝数组中。这样，当原始集合的数据改变，拷贝数据中的值也不会变化