Java 集合系列（3）： fail-fast 总结(通过 ArrayList 来说明 fail-fast 的原理、解决办法)

概要

前面，我们已经学习了 ArrayList。接下来，我们以 ArrayList 为例，对 Iterator 的 fail-fast 机制进行了解。

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)

```
import java.util.*;
import java.util.concurrent.*;
/*
 * @desc java 集合中 Fast-Fail 的测试程序。
 *
 *   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。
 *   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。
 *
 *   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。
 *   (01) 使用 ArrayList 时，会产生 fast-fail 事件，抛出 ConcurrentModificationException 异常；定义如下：
 *           private static List<String> list = new ArrayList<String>();
 *   (02) 使用时 CopyOnWriteArrayList，不会产生 fast-fail 事件；定义如下：
```

```
 *           private static List<String> list = new CopyOnWriteArrayList<String>();
  * @author skywang
 */
public class FastFailTest {
    private static List<String> list = new ArrayList<String>();
    //private static List<String> list = new CopyOnWriteArrayList<String>();
    public static void main(String[] args) {
        // 同时启动两个线程对 list 进行操作！
        new ThreadOne().start();
        new ThreadTwo().start();
    }
    private static void printAll() {
        System.out.println("");
        String value = null;
        Iterator iter = list.iterator();
        while(iter.hasNext()) {
            value = (String)iter.next();
            System.out.print(value+", ");
        }
    }
    /**
     * 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
     */
    private static class ThreadOne extends Thread {
        public void run() {
            int i = 0;
            while (i<6) {
                list.add(String.valueOf(i));
                printAll();
                i++;
            }
        }
    }
    /**
     * 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
     */
    private static class ThreadTwo extends Thread {
        public void run() {
            int i = 10;
            while (i<16) {
```

```
                list.add(String.valueOf(i));
                printAll();
                i++;
            }
        }
    }
}
```

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：

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

```
...
// AbstractList 中唯一的属性
// 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
protected transient int modCount = 0;
// 返回 List 对应迭代器。实际上，是返回 Itr 对象。
public Iterator<E> iterator() {
    return new Itr();
}
// Itr 是 Iterator(迭代器)的实现类
private class Itr implements Iterator<E> {
    int cursor = 0;
    int lastRet = -1;
    // 修改数的记录值。
    // 每次新建 Itr()对象时，都会保存新建该对象时对应的 modCount；
    // 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；
    // 若不相等，则抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。
    int expectedModCount = modCount;
    public boolean hasNext() {
        return cursor != size();
    }
    public E next() {
        // 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；
        // 若不相等，则抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。
        checkForComodification();
        try {
            E next = get(cursor);
            lastRet = cursor++;
            return next;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    public void remove() {
        if (lastRet == -1)
            throw new IllegalStateException();
        checkForComodification();
        try {
            AbstractList.this.remove(lastRet);
```

```
                if (lastRet < cursor)
                    cursor--;
                lastRet = -1;
                expectedModCount = modCount;
            } catch (IndexOutOfBoundsException e) {
                throw new ConcurrentModificationException();
            }
        }
        final void checkForComodification() {
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
    ...
}
```

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。

```
package java.util;
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    ...
    // list 中容量变化时，对应的同步函数
    public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (oldCapacity * 3)/2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            // minCapacity is usually close to size, so this is a win:
```

```
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
    // 添加元素到队列最后
    public boolean add(E e) {
        // 修改 modCount
        ensureCapacity(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }
    // 添加元素到指定的位置
    public void add(int index, E element) {
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);
        // 修改 modCount
        ensureCapacity(size+1);  // Increments modCount!!
        System.arraycopy(elementData, index, elementData, index + 1,
             size - index);
        elementData[index] = element;
        size++;
    }
    // 添加集合
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        // 修改 modCount
        ensureCapacity(size + numNew);  // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }
    // 删除指定位置的元素
    public E remove(int index) {
        RangeCheck(index);
        // 修改 modCount
        modCount++;
        E oldValue = (E) elementData[index];
        int numMoved = size - index - 1;
        if (numMoved > 0)
```

```
            System.arraycopy(elementData, index+1, elementData, index, numMoved);
        elementData[--size] = null; // Let gc do its work
        return oldValue;
    }
    // 快速删除指定位置的元素
    private void fastRemove(int index) {
        // 修改 modCount
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // Let gc do its work
    }
    // 清空集合
    public void clear() {
        // 修改 modCount
        modCount++;
        // Let gc do its work
        for (int i = 0; i < size; i++)
            elementData[i] = null;
        size = 0;
    }
    ...
}
```

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。我们先看看 CopyOnWriteArrayList 的源码：

```
package java.util.concurrent;
import java.util.*;
import java.util.concurrent.locks.*;
import sun.misc.Unsafe;
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    ...
    // 返回集合对应的迭代器
    public Iterator<E> iterator() {
        return new COWIterator<E>(getArray(), 0);
    }
    ...
    private static class COWIterator<E> implements ListIterator<E> {
        private final Object[] snapshot;
        private int cursor;
        private COWIterator(Object[] elements, int initialCursor) {
            cursor = initialCursor;
            // 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。
            snapshot = elements;
        }
        public boolean hasNext() {
            return cursor < snapshot.length;
        }
        public boolean hasPrevious() {
```

```
            return cursor > 0;
        }
        public E next() {
            if (! hasNext())
                throw new NoSuchElementException();
            return (E) snapshot[cursor++];
        }
        public E previous() {
            if (! hasPrevious())
                throw new NoSuchElementException();
            return (E) snapshot[--cursor];
        }
        public int nextIndex() {
            return cursor;
        }
        public int previousIndex() {
            return cursor-1;
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public void set(E e) {
            throw new UnsupportedOperationException();
        }
        public void add(E e) {
            throw new UnsupportedOperationException();
        }
    }
    ...
}
```

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！