ConcurrentModificationException异常出现的原因

　先看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Test {      public static void main(String[] args)  {          ArrayList<Integer> list = new ArrayList<Integer>();          list.add(2);          Iterator<Integer> iterator = list.iterator();          while(iterator.hasNext()){              Integer integer = iterator.next();              if(integer==2)                  list.remove(integer);          }      }  } |

 　　运行结果：



　　从异常信息可以发现，异常出现在checkForComodification()方法中。

　　我们不忙看checkForComodification()方法的具体实现，我们先根据程序的代码一步一步看ArrayList源码的实现：

　　首先看ArrayList的iterator()方法的具体实现，查看源码发现在ArrayList的源码中并没有iterator()这个方法，那么很显然这个方法应该是其父类或者实现的接口中的方法，我们在其父类AbstractList中找到了iterator()方法的具体实现，下面是其实现代码：

|  |  |
| --- | --- |
| 1  2  3 | public Iterator<E> iterator() {      return new Itr();  } |

 　　从这段代码可以看出返回的是一个指向Itr类型对象的引用，我们接着看Itr的具体实现，在AbstractList类中找到了Itr类的具体实现，它是AbstractList的一个成员内部类，下面这段代码是Itr类的所有实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | private class Itr implements Iterator<E> {      int cursor = 0;      int lastRet = -1;      int expectedModCount = modCount;      public boolean hasNext() {             return cursor != size();      }      public E next() {             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();      }  } |

 　　首先我们看一下它的几个成员变量：

　　cursor：表示下一个要访问的元素的索引，从next()方法的具体实现就可看出

　　lastRet：表示上一个访问的元素的索引

expectedModCount：表示对ArrayList修改次数的期望值，它的初始值为modCount。

modCount是ArrayList的一个属性，继承自抽象类AbstractList，用于表示ArrayList对象被修改次数。modCount是AbstractList类中的一个成员变量

|  |  |
| --- | --- |
| 1 | protected transient int modCount = 0; |

 　　该值表示对List的修改次数，查看ArrayList的add()和remove()方法就可以发现，每次调用add()方法或者remove()方法就会对modCount进行加1操作。整个ArrayList中修改modCount的方法比较多，有add、remove、clear、ensureCapacityInternal等，凡是设计到ArrayList对象修改的都会自增modCount属性。

　　当调用list.iterator()返回一个Iterator之后，通过Iterator的hashNext()方法判断是否还有元素未被访问，我们看一下hasNext()方法，hashNext()方法的实现很简单：

|  |  |
| --- | --- |
| 1  2  3 | public boolean hasNext() {      return cursor != size();  } |

 　　如果下一个访问的元素下标不等于ArrayList的大小，就表示有元素需要访问，这个很容易理解，如果下一个访问元素的下标等于ArrayList的大小，则肯定到达末尾了。

　　然后通过Iterator的next()方法获取到下标为0的元素，我们看一下next()方法的具体实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public E next() {      checkForComodification();   try {      E next = get(cursor);      lastRet = cursor++;      return next;   } catch (IndexOutOfBoundsException e) {      checkForComodification();      throw new NoSuchElementException();   }  } |

 　　这里是非常关键的地方：首先在next()方法中会调用checkForComodification()方法，然后根据cursor的值获取到元素，接着将cursor的值赋给lastRet，并对cursor的值进行加1操作。初始时，cursor为0，lastRet为-1，那么调用一次之后，cursor的值为1，lastRet的值为0。注意此时，modCount为0，expectedModCount也为0。

　　接着往下看，程序中判断当前元素的值是否为2，若为2，则调用list.remove()方法来删除该元素。

　　我们看一下在ArrayList中的remove()方法做了什么：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | 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;  }      private void fastRemove(int index) {      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  } |

 　　通过remove方法删除元素最终是调用的fastRemove()方法，在fastRemove()方法中，首先对modCount进行加1操作（因为对集合修改了一次），然后接下来就是删除元素的操作，最后将size进行减1操作，并将引用置为null以方便垃圾收集器进行回收工作。

　　那么注意此时各个变量的值：对于iterator，其expectedModCount为0，cursor的值为1，lastRet的值为0。

　　对于list，其modCount为1，size为0。

　　接着看程序代码，执行完删除操作后，继续while循环，调用hasNext方法()判断，由于此时cursor为1，而size为0，那么返回true，所以继续执行while循环，然后继续调用iterator的next()方法：

　　注意，此时要注意next()方法中的第一句：checkForComodification()。

　　在checkForComodification方法中进行的操作是：

|  |  |
| --- | --- |
| 1  2  3  4 | final void checkForComodification() {      if (modCount != expectedModCount)      throw new ConcurrentModificationException();  } |

 　　如果modCount不等于expectedModCount，则抛出ConcurrentModificationException异常。

　　很显然，此时modCount为1，而expectedModCount为0，因此程序就抛出了ConcurrentModificationException异常。

　　到这里，想必大家应该明白为何上述代码会抛出ConcurrentModificationException异常了。

***关键点就在于：调用list.remove()方法导致modCount和expectedModCount的值不一致。***

注意，像使用for-each进行迭代实际上也会出现这种问题。

补充（Iterator 是工作在一个独立的线程中，并且拥有一个 mutex 锁。

Iterator 被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变。

当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出 java.util.ConcurrentModificationException 异常。

所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。但你可以使用 Iterator 本身的方法 remove() 来删除对象， Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。）

**在单线程环境下的解决办法**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 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();      }  } |

 　　在这个方法中，删除元素实际上调用的就是list.remove()方法，但是它多了一个操作：

|  |  |
| --- | --- |
| 1 | expectedModCount = modCount; |

 　　因此，在迭代器中如果要删除元素的话，需要调用Itr类的remove方法。

　　将上述代码改为下面这样就不会报错了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Test {      public static void main(String[] args)  {          ArrayList<Integer> list = new ArrayList<Integer>();          list.add(2);          Iterator<Integer> iterator = list.iterator();          while(iterator.hasNext()){              Integer integer = iterator.next();              if(integer==2)                  iterator.remove();   //注意这个地方          }      }  } |

## 在多线程环境下的解决方法

1）在使用iterator迭代的时候使用synchronized或者Lock进行同步；

2）使用并发容器CopyOnWriteArrayList代替ArrayList和Vector。