

66 | 迭代器模式（中）：遍历集合的同时，为什么不能增删集合元素？

王争 · 设计模式之美



上一节课中，我们通过给 ArrayList、LinkedList 容器实现迭代器，学习了迭代器模式的原理、实现和设计意图。迭代器模式主要作用是解耦容器代码和遍历代码，这也印证了我们前面多次讲过的应用设计模式的主要目的是解耦。

上一节课中讲解的内容都比较基础，今天，我们来深挖一下，如果在使用迭代器遍历集合的同时增加、删除集合中的元素，会发生什么情况？应该如何应对？如何在遍历的同时安全地删除集合元素？


话不多说，让我们正式开始今天的内容吧！

在遍历的同时增删集合元素会发生什么？

在通过迭代器来遍历集合元素的同时，增加或者删除集合中的元素，有可能会造成某个元素被重复遍历或遍历不到。不过，并不是所有情况下都会遍历出错，有的时候也可以正常遍历，所

以，这种行为称为**结果不可预期行为**或者**未决行为**，也就是说，运行结果到底是对还是错，要视情况而定。

怎么理解呢？我们通过一个例子来解释一下。我们还是延续上一节课实现的 ArrayList 迭代器的例子。为了方便你查看，我把相关的代码都重新拷贝到这里了。

 复制代码

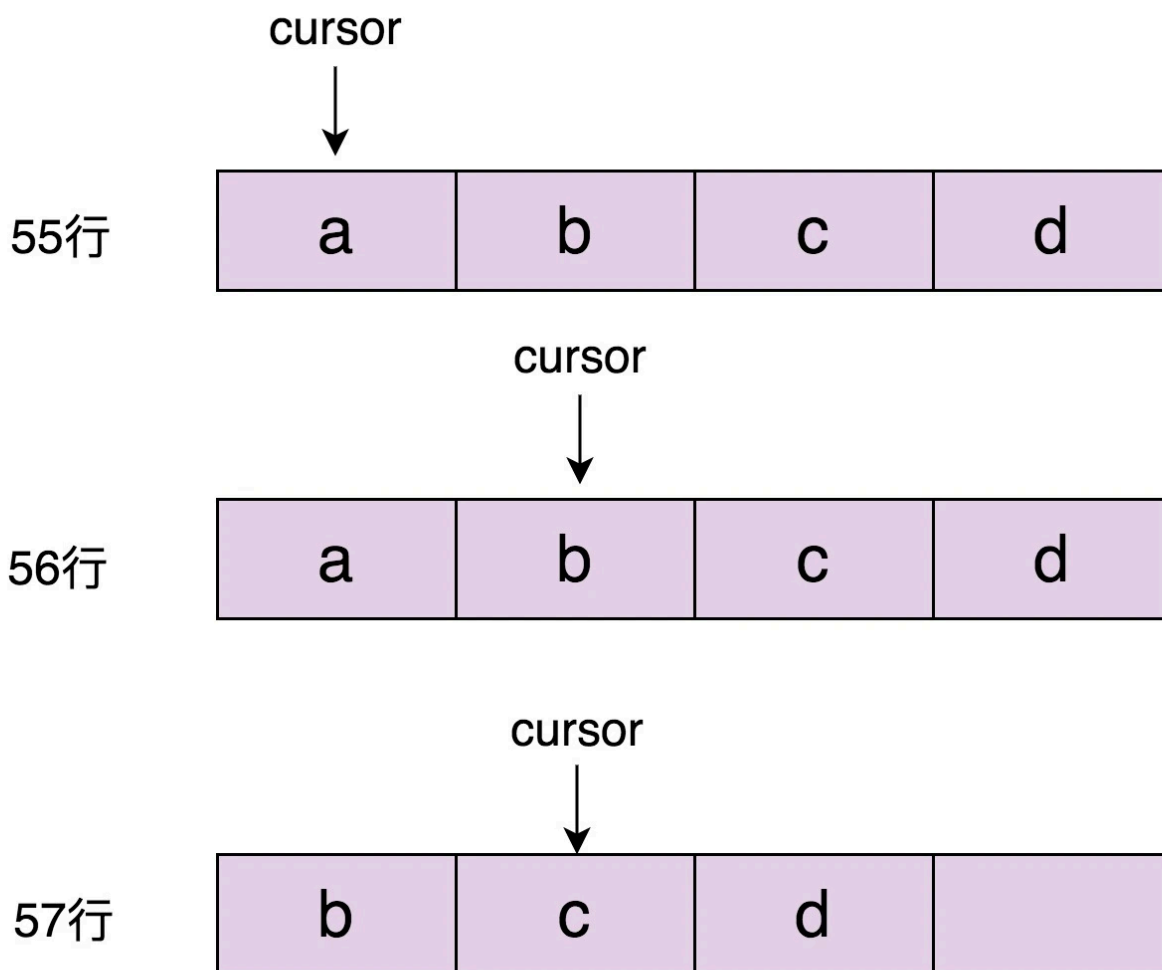
```
1 public interface Iterator<E> {
2     boolean hasNext();
3     void next();
4     E currentItem();
5 }
6
7 public class ArrayIterator<E> implements Iterator<E> {
8     private int cursor;
9     private ArrayList<E> arrayList;
10
11     public ArrayIterator(ArrayList<E> arrayList) {
12         this.cursor = 0;
13         this.arrayList = arrayList;
14     }
15
16     @Override
17     public boolean hasNext() {
18         return cursor < arrayList.size();
19     }
20
21     @Override
22     public void next() {
23         cursor++;
24     }
25
26     @Override
27     public E currentItem() {
28         if (cursor >= arrayList.size()) {
29             throw new NoSuchElementException();
30         }
31         return arrayList.get(cursor);
32     }
33 }
34
35 public interface List<E> {
36     Iterator iterator();
37 }
38
```

```
39 public class ArrayList<E> implements List<E> {
40     //...
41     public Iterator iterator() {
42         return new ArrayIterator(this);
43     }
44     //...
45 }
46
47 public class Demo {
48     public static void main(String[] args) {
49         List<String> names = new ArrayList<>();
50         names.add("a");
51         names.add("b");
52         names.add("c");
53         names.add("d");
54
55         Iterator<String> iterator = names.iterator();
56         iterator.next();
57         names.remove("a");
58     }
59 }
```

我们知道，ArrayList 底层对应的是数组这种数据结构，在执行完第 55 行代码的时候，数组中存储的是 a、b、c、d 四个元素，迭代器的游标 cursor 指向元素 a。当执行完第 56 行代码的时候，游标指向元素 b，到这里都没有问题。

为了保持数组存储数据的连续性，数组的删除操作会涉及元素的搬移（详细的讲解你可以去看我的另一个专栏《数据结构与算法之美》）。当执行到第 57 行代码的时候，我们从数组中将元素 a 删除掉，b、c、d 三个元素会依次往前搬移一位，这就会导致游标本来指向元素 b，现在变成了指向元素 c。原本在执行完第 56 行代码之后，我们还可以遍历到 b、c、d 三个元素，但在执行完第 57 行代码之后，我们只能遍历到 c、d 两个元素，b 遍历不到了。


对于上面的描述，我画了一张图，你可以对照着理解。



不过，如果第 57 行代码删除的不是游标前面的元素（元素 a）以及游标所在位置的元素（元素 b），而是游标后面的元素（元素 c 和 d），这样就不会存在任何问题了，不会存在某个元素遍历不到的情况了。

所以，我们前面说，在遍历的过程中删除集合元素，结果是不可预期的，有时候没问题（删除元素 c 或 d），有时候就有问题（删除元素 a 或 b），这个要视情况而定（到底删除的是哪个位置的元素），就是这个意思。

在遍历的过程中删除集合元素，有可能会造成导致某个元素遍历不到，那在遍历的过程中添加集合元素，会发生什么情况呢？还是结合刚刚那个例子来讲解，我们将上面的代码稍微改造一下，把删除元素改为添加元素。具体的代码如下所示：

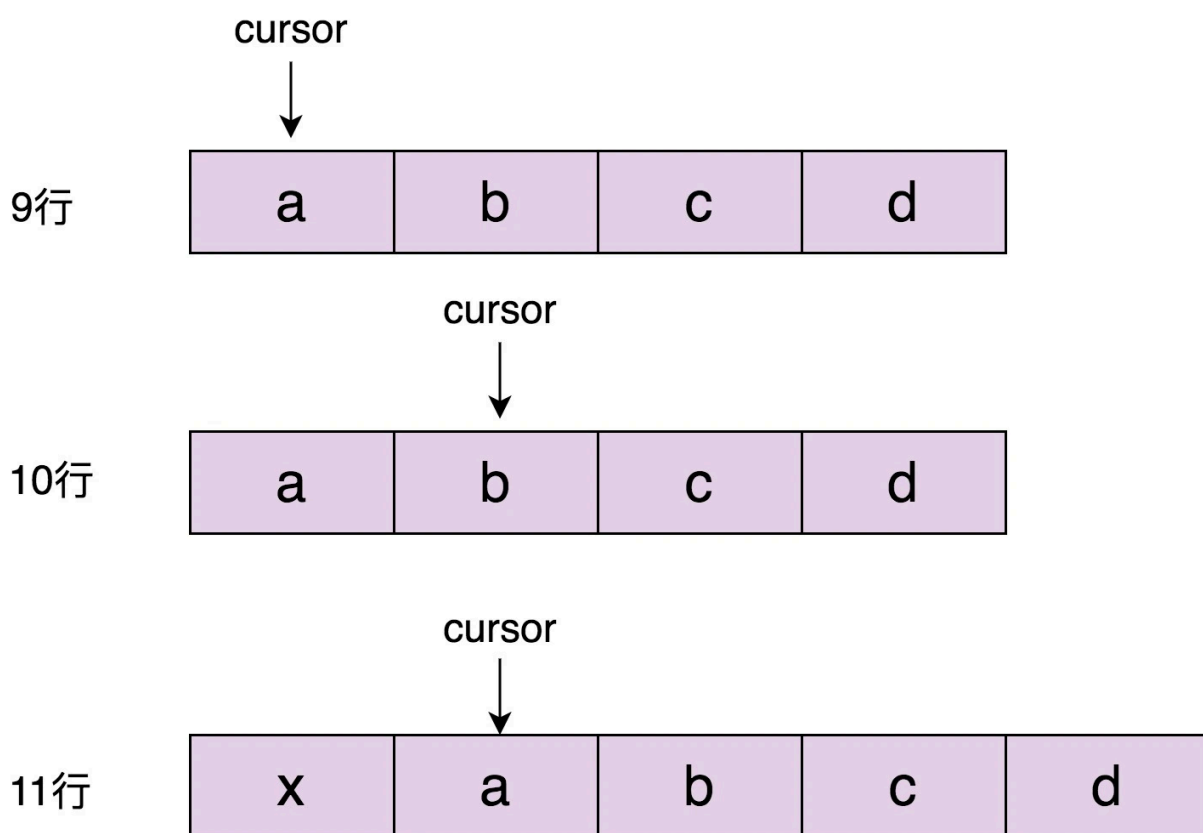
 复制代码

```
1 public class Demo {  
2     public static void main(String[] args) {  
3         List<String> names = new ArrayList<>();  
4         names.add("a");  
5         names.add("b");  
6         names.add("c");  
7         names.add("d");  
8  
9         Iterator<String> iterator = names.iterator();  
10        iterator.next();  
11        names.add(0, "x");  
12    }  
13 }
```

在执行完第 10 行代码之后，数组中包含 a、b、c、d 四个元素，游标指向 b 这个元素，已经跳过了元素 a。在执行完第 11 行代码之后，我们将 x 插入到下标为 0 的位置，a、b、c、d 四个元素依次往后移动一位。这个时候，游标又重新指向了元素 a。元素 a 被游标重复指向两次，也就是说，元素 a 存在被重复遍历的情况。

跟删除情况类似，如果我们在游标的后面添加元素，就不会存在任何问题。所以，在遍历的同时添加集合元素也是一种不可预期行为。

同样，对于上面的添加元素的情况，我们也画了一张图，如下所示，你可以对照着理解。



如何应对遍历时改变集合导致的未决行为？


当通过迭代器来遍历集合的时候，增加、删除集合元素会导致不可预期的遍历结果。实际上，“不可预期”比直接出错更加可怕，有的时候运行正确，有的时候运行错误，一些隐藏很深、很难 debug 的 bug 就是这么产生的。那我们如何才能避免出现这种不可预期的运行结果呢？

有两种比较干脆利索的解决方案：一种是遍历的时候不允许增删元素，另一种是增删元素之后让遍历报错。

实际上，第一种解决方案比较难实现，我们要确定遍历开始和结束的时间点。遍历开始的时间节点我们很容易获得。我们可以把创建迭代器的时间点作为遍历开始的时间点。但是，遍历结


束的时间点该如何来确定呢？

你可能会说，遍历到最后一个元素的时候就算结束呗。但是，在实际的软件开发中，每次使用迭代器来遍历元素，并不一定非要把所有元素都遍历一遍。如下所示，我们找到一个值为 b 的元素就提前结束了遍历。

 复制代码

```
1 public class Demo {
2     public static void main(String[] args) {
3         List<String> names = new ArrayList<>();
4         names.add("a");
5         names.add("b");
6         names.add("c");
7         names.add("d");
8
9         Iterator<String> iterator = names.iterator();
10        while (iterator.hasNext()) {
11            String name = iterator.currentItem();
12            if (name.equals("b")) {
13                break;
14            }
15        }
16    }
17 }
```

你可能还会说，那我们可以在迭代器类中定义一个新的接口 finishIteration(), 主动告知容器迭代器使用完了，你可以增删元素了，示例代码如下所示。但是，这就要求程序员在使用完迭代器之后要主动调用这个函数，也增加了开发成本，还很容易漏掉。

 复制代码

```
1 public class Demo {
2     public static void main(String[] args) {
3         List<String> names = new ArrayList<>();
4         names.add("a");
5         names.add("b");
6         names.add("c");
7         names.add("d");
8
9         Iterator<String> iterator = names.iterator();
10        while (iterator.hasNext()) {
```


```
11     String name = iterator.currentItem();
12     if (name.equals("b")) {
13         iterator.finishIteration();//主动告知容器这个迭代器用完了
14         break;
15     }
16 }
17 }
18 }
```

实际上，第二种解决方法更加合理。Java 语言就是采用的这种解决方案，增删元素之后，让遍历报错。接下来，我们具体来看一下如何实现。

怎么确定在遍历时候，集合有没有增删元素呢？我们在 ArrayList 中定义一个成员变量 modCount，记录集合被修改的次数，集合每调用一次增加或删除元素的函数，就会给 modCount 加 1。当通过调用集合上的 iterator() 函数来创建迭代器的时候，我们把 modCount 值传递给迭代器的 expectedModCount 成员变量，之后每次调用迭代器上的 hasNext()、next()、currentItem() 函数，我们都会检查集合上的 modCount 是否等于 expectedModCount，也就是看，在创建完迭代器之后，modCount 是否改变过。

如果两个值不相同，那就说明集合存储的元素已经改变了，要么增加了元素，要么删除了元素，之前创建的迭代器已经不能正确运行了，再继续使用就会产生不可预期的结果，所以我们选择 fail-fast 解决方式，抛出运行时异常，结束掉程序，让程序员尽快修复这个因为不正确使用迭代器而产生的 bug。

上面的描述翻译成代码就是下面这样子。你可以结合着代码一起理解我刚才的讲解。

 复制代码

```
1 public class ArrayIterator implements Iterator {
2     private int cursor;
3     private ArrayList arrayList;
4     private int expectedModCount;
5
6     public ArrayIterator(ArrayList arrayList) {
7         this.cursor = 0;
8         this.arrayList = arrayList;
9         this.expectedModCount = arrayList.modCount;
10    }
11 }
```



```


12  @Override
13  public boolean hasNext() {
14      checkForComodification();
15      return cursor < arrayList.size();
16  }
17
18  @Override
19  public void next() {
20      checkForComodification();
21      cursor++;
22  }
23
24  @Override
25  public Object currentItem() {
26      checkForComodification();
27      return arrayList.get(cursor);
28  }
29
30  private void checkForComodification() {
31      if (arrayList.modCount != expectedModCount)
32          throw new ConcurrentModificationException();
33  }
34 }
35
36 //代码示例
37 public class Demo {
38     public static void main(String[] args) {
39         List<String> names = new ArrayList<>();
40         names.add("a");
41         names.add("b");
42         names.add("c");
43         names.add("d");
44
45         Iterator<String> iterator = names.iterator();
46         iterator.next();
47         names.remove("a");
48         iterator.next();//抛出ConcurrentModificationException异常
49     }
50 }

```

如何在遍历的同时安全地删除集合元素？


像 Java 语言，迭代器类中除了前面提到的几个最基本的方法之外，还定义了一个 `remove()` 方法，能够在遍历集合的同时，安全地删除集合中的元素。不过，需要说明的是，它并没有提供添加元素的方法。毕竟迭代器的主要作用是遍历，添加元素放到迭代器里本身就不合适。

我个人觉得，Java 迭代器中提供的 remove() 方法还是比较鸡肋的，作用有限。它只能删除游标指向的前一个元素，而且一个 next() 函数之后，只能跟着最多一个 remove() 操作，多次调用 remove() 操作会报错。我还是通过一个例子来解释一下。

 复制代码

```
1 public class Demo {
2     public static void main(String[] args) {
3         List<String> names = new ArrayList<>();
4         names.add("a");
5         names.add("b");
6         names.add("c");
7         names.add("d");
8
9         Iterator<String> iterator = names.iterator();
10        iterator.next();
11        iterator.remove();
12        iterator.remove(); //报错，抛出IllegalStateException异常
13    }
14 }
```

现在，我们一块来看下，为什么通过迭代器就能安全的删除集合中的元素呢？源码之下无秘密。我们来看下 remove() 函数是如何实现的，代码如下所示。稍微提醒一下，在 Java 实现中，迭代器类是容器类的内部类，并且 next() 函数不仅将游标后移一位，还会返回当前的元素。

 复制代码

```
1 public class ArrayList<E> {
2     transient Object[] elementData;
3     private int size;
4
5     public Iterator<E> iterator() {
6         return new Itr();
7     }
8
9     private class Itr implements Iterator<E> {
10        int cursor; // index of next element to return
11        int lastRet = -1; // index of last element returned; -1 if no such
12        int expectedModCount = modCount;
13
14        Itr() {}
15    }
```

```

16     public boolean hasNext() {
17         return cursor != size;
18     }
19
20     @SuppressWarnings("unchecked")
21     public E next() {
22         checkForComodification();
23         int i = cursor;
24         if (i >= size)
25             throw new NoSuchElementException();
26         Object[] elementData = ArrayList.this.elementData;
27         if (i >= elementData.length)
28             throw new ConcurrentModificationException();
29         cursor = i + 1;
30         return (E) elementData[lastRet = i];
31     }
32
33     public void remove() {
34         if (lastRet < 0)
35             throw new IllegalStateException();
36         checkForComodification();
37
38         try {
39             ArrayList.this.remove(lastRet);
40             cursor = lastRet;
41             lastRet = -1;
42             expectedModCount = modCount;
43         } catch (IndexOutOfBoundsException ex) {
44             throw new ConcurrentModificationException();
45         }
46     }
47 }
48 }

```

在上面的代码实现中，迭代器类新增了一个 lastRet 成员变量，用来记录游标指向的前一个元素。通过迭代器去删除这个元素的时候，我们可以更新迭代器中的游标和 lastRet 值，来保证不会因为删除元素而导致某个元素遍历不到。如果通过容器来删除元素，并且希望更新迭代器中的游标值来保证遍历不出错，我们就要维护这个容器都创建了哪些迭代器，每个迭代器是否还在使用等信息，代码实现就变得比较复杂了。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。


在通过迭代器来遍历集合元素的同时，增加或者删除集合中的元素，有可能会产生导致某个元素被重复遍历或遍历不到。不过，并不是所有情况下都会遍历出错，有的时候也可以正常遍历，所以，这种行为称为结果不可预期行为或者未决行为。实际上，“不可预期”比直接出错更加可怕，有的时候运行正确，有的时候运行错误，一些隐藏很深、很难 debug 的 bug 就是这么产生的。

有两种比较干脆利索的解决方案，来避免出现这种不可预期的运行结果。一种是遍历的时候不允许增删元素，另一种是增删元素之后让遍历报错。第一种解决方案比较难实现，因为很难确定迭代器使用结束的时间点。第二种解决方案更加合理。Java 语言就是采用的这种解决方案。增删元素之后，我们选择 fail-fast 解决方式，让遍历操作直接抛出运行时异常。

像 Java 语言，迭代器类中除了前面提到的几个最基本的方法之外，还定义了一个 `remove()` 方法，能够在遍历集合的同时，安全地删除集合中的元素。

课堂讨论

1、基于文章中给出的 Java 迭代器的实现代码，如果一个容器对象同时创建了两个迭代器，一个迭代器调用了 `remove()` 方法删除了集合中的一个元素，那另一个迭代器是否还可用？或者，我换个问法，下面代码中的第 13 行的运行结果是什么？

 复制代码

```
1 public class Demo {
2     public static void main(String[] args) {
3         List<String> names = new ArrayList<>();
4         names.add("a");
5         names.add("b");
6         names.add("c");
7         names.add("d");
8
9         Iterator<String> iterator1 = names.iterator();
10        Iterator<String> iterator2 = names.iterator();
11        iterator1.next();
12        iterator1.remove();
13        iterator2.next(); // 运行结果?
14    }
15 }
```

2、LinkedList 底层基于链表，如果在遍历的同时，增加删除元素，会出现哪些不可预期的行为呢？

欢迎留言和我分享你的想法。如果有收获，欢迎你把这篇文章分享给你的朋友。

AI智能总结

本文深入探讨了在遍历集合时增删元素可能带来的不可预期问题，并提出了解决方案。在遍历过程中增删元素可能导致某些元素被重复遍历或遍历不到，这种行为称为未决行为。文章指出Java语言采用fail-fast解决方式，即在遍历时检查集合的修改次数，如果发生了修改则抛出ConcurrentModificationException异常。此外，文章还详细解释了迭代器的remove()方法如何安全地删除集合元素，并通过代码示例展示了其实现原理。最后，文章提出了课堂讨论问题，引发读者思考。整体而言，本文对于理解迭代器模式的原理和设计意图具有很好的指导意义。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (35)

最新 精选



小晏子

2020-04-03

思考题：

1. iterator1 和 iterator2是两个不同的迭代器对象，修改一个不会影响另外一个，所以执行iterator1.remove()后，再执行iterator2.next时，会执行checkForComodification();检查，可是检查条件“arrayList.modCount != expectedModCount”中arrayList的modCount已经变成了5，而此时iterator2的expectedModCount还是4，所以触发ConcurrentModificationException异常。
2. LinkedList和ArrayList不同是LinkedList底层基于链表实现，增加删除元素不需要移动元素的位置，所以不会出现跟ArrayList不同的情况，比如增加元素时，不论增加的元素时在迭代器前还是后，都能通过指针寻址到下一个元素。

共 13 条评论 >

👍 81



kyle

2020-04-03

迭代器中删除元素那一段，执行完第57行(删除a以后)，游标应该指向c，图中指向d了

共 6 条评论 >

👍 21



Ken张云忠

2020-04-03

1.基于文章中给出的 Java 迭代器的实现代码, 如果一个容器对象同时创建了两个迭代器, 一个迭代器调用了 remove() 方法删除了集合中的一个元素, 那另一个迭代器是否还可用? 或者, 我换个问法, 下面代码中的第 13 行的运行结果是什么?

Exception in thread "main" java.util.ConcurrentModificationException

因为iterator2.expectedModCount的值与names.modCount的值不相等,expectedModCount比modCount小1.

2.LinkedList 底层基于链表, 如果在遍历的同时, 增加删除元素, 会出现哪些不可预期的行为呢?

当在游标及游标之前增删元素时会使得有的元素遍历不到;当在游标之后增删元素时无问题.

LinkedList与ArrayList一样,因为都是集成抽象类java.util.AbstractList,

在遍历的同时调用两次remove()都会抛出异常,都会抛出的是java.lang.IllegalStateException异常.

两个迭代器遍历的同时,其中一个迭代器删除元素都会使另一个迭代器抛出java.util.ConcurrentModificationException异常.

都不支持迭代器里添加元素.

共 4 条评论 >

👍 18



,

2020-04-03

1.ConcurrentModificationException是在调用迭代器的next方法时产生,因为迭代器2并没有使用,所以不会报错,如果在第13行调用的是iterator2.next()则会报错(原因:expectedModCount在新建迭代器的时候初始化,调用iterator1.remove()只修改iterator1的expectedModCount,不会修改iterator2的,所以在调用iterator2.next()时会报错)

2.使用迭代器遍历的同时,使用容器的方法进行增删操作也会触发ConcurrentModificationException,行为和ArrayList是一样的

我有一个问题想问老师,我是培训班出身,而且学历不好,自觉基础不行,所以从工作以来,基本每天都坚持学习,如今已经工作一年多了.可是我每天学习两三个钟头就觉得很累了,脑子像浆糊一样,没办法继续学新东西了,有时学习一整天,从上班开始学,一直学到下班,下班的时候感觉脑子都要扭曲了,好长时间缓解不过来,前几天听说去哪网的前端架构师去世了,年龄才30岁出头,我感觉我保持当下这个状态的话,到不了他的水平就得猝死,我想知道老师是怎么平衡日常生活的?真的有人能坚持每天学习十几个小时吗?这让我觉得特别累,喘不过气来

共 13 条评论 >

👍 15



忆水寒

2020-04-05

第一个问题，由于modcount不一样了，所以会出现异常。
第二个问题，LinkedList和ArrayList行为一致。



👍 6



Jackie

2020-04-03

终于明白报ConcurrentModificationException的真正原因了



👍 5



Monday

2020-04-03

```
public class Demo {  
    public static void main(String[] args) {  
        List<String> names = new ArrayList<>();  
        names.add("a");  
        names.add("b");  
        names.add("c");  
        names.add("d");  
        Iterator<String> iterator1 = names.iterator();  
        Iterator<String> iterator2 = names.iterator();  
        iterator1.next();  
        iterator1.remove();  
        iterator1.next(); // 运行结果?  
    }  
}
```

哈哈老师的题目笔误了吧。
运行结果那行应该是iterator2.next()。

然后结果应该是会抛异常，因为modifyCount不一致了。



👍 4



Liam

2020-04-03

1 第二个迭代器会报错，modCount发生变化

2 链表增删不影响游标，不会出现意外



4



xzy

2020-04-14

看了ArrayList和LinkedList的源码，来回答思考题：

1，会抛出 ConcurrentModificationException 异常

2，添加元素分为两种情况：添加元素在游标前、添加元素在游标后

游标前：nextIndex = next元素实际的下标 - 1，所以hasNext()函数在next()返回链表的最后一个元素后仍然成立

游标后：无影响

删除元素也分为三种情况：删除元素在游标前、删除元素在游标后、删除游标元素

游标前：nextIndex = next元素的实际下标 + 1，所有hasNext()函数在next()返回链表末尾第二个元素后便不在成立

游标后：无影响

删除游标元素：访问当前游标元素后面的元素时会报空指针异常

```
/**
```

```
* LinkedList 迭代器的部分源码
```

```
* /
```

```
private class ListItr implements ListIterator<E> {
```

```
    private Node<E> lastReturned;
```

```
    private Node<E> next;
```

```
    private int nextIndex;
```

```
    private int expectedModCount = modCount;
```

```
    ListItr(int index) {
```

```
        // assert isPositionIndex(index);
```

```
        next = (index == size) ? null : node(index);
```

```
        nextIndex = index;
```

```
    }
```

```
    public boolean hasNext() {
```

```
        return nextIndex < size;
```

```
    }
```

```
    public E next() {
```



```
        checkForComodification();
        if (!hasNext())
            throw new NoSuchElementException();

        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.item;
    }

}
```



👍 3



东征

2020-04-09

问题一，因为每个iterator都会保留自己的expectModCount，而modCount是全局的，所以会抛错。

问题二，因为底层实现是链表，所以若在游标前新增删除，会导致整体遍历漏处理或多处理。在游标上删除，导致后面的内容无法遍历。在游标后新增删除无影响



👍 3