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 2020-04-03

00:00 11:14
讲述：冯永吉 大小：10.30M

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**，也就是说，运行结果到底是对还是错，要视情况而定。

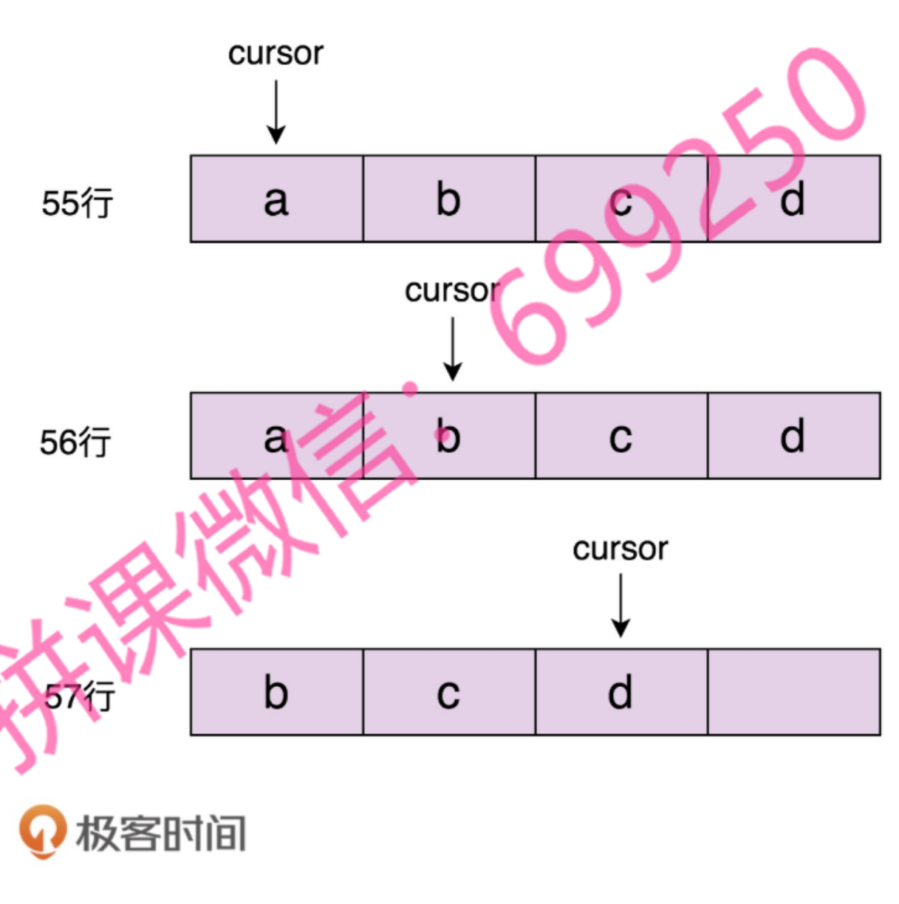
怎么理解呢？我们通过一个例子来解释一下。我们还是延续上一节课实现的 ArrayList 迭代器的例子。为了方便你查看，我把相关的代码都重新拷贝到这里了。

复制代码
```java
public interface Iterator<E> {
  boolean hasNext();
  void next();
  E currentItem();
}

public class ArrayIterator<E> implements Iterator<E> {
  private int cursor;
  private ArrayList<E> arrayList;

  public ArrayIterator(ArrayList<E> arrayList) {
    this.cursor = 0;
    this.arrayList = arrayList;
  }

  @Override
  public boolean hasNext() {
    return cursor < arrayList.size();
  }

  @Override
  public void next() {
    cursor++;
  }

  @Override
  public E currentItem() {
    if (cursor >= arrayList.size()) {
      throw new NoSuchElementException();
    }
    return arrayList.get(cursor);
  }
}

public interface List<E> {
  Iterator iterator();
}

public class ArrayList<E> implements List<E> {
  //...
  public Iterator iterator() {
    return new ArrayIterator(this);
  }
  //...
}

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> iterator = names.iterator();
    iterator.next();
    names.remove("a");
  }
}
```

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。

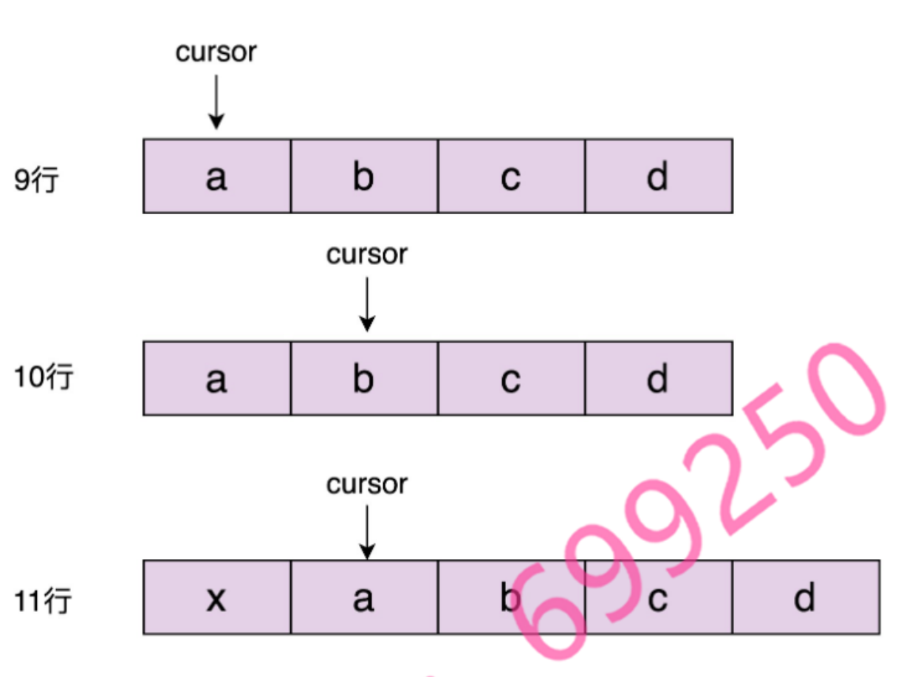
在遍历的过程中删除集合元素，有可能会导致某个元素遍历不到，那在遍历的过程中添加集合元素，会发生什么情况呢？还是结合刚刚那个例子来讲解，我们将上面的代码稍微改造一下，把删除元素改为添加元素。具体的代码如下所示：

复制代码
```java
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> iterator = names.iterator();
    iterator.next();
    names.add(0, "x");
  }
}
```

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
```java
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> iterator = names.iterator();
    while (iterator.hasNext()) {
      String name = iterator.currentItem();
      if (name.equals("b")) {
        break;
      }
    }
  }
}
```

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
```java
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> iterator = names.iterator();
    while (iterator.hasNext()) {
      String name = iterator.currentItem();
      if (name.equals("b")) {
        iterator.finishIteration();//主动告知容器这个迭代器用完了
        break;
      }
    }
  }
}
```

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
```java
public class ArrayIterator implements Iterator {
  private int cursor;
  private ArrayList arrayList;
  private int expectedModCount;

  public ArrayIterator(ArrayList arrayList) {
    this.cursor = 0;
    checkForComodification();
    return cursor < arrayList.size();
  }

  @Override
  public void next() {
    checkForComodification();
    cursor++;
  }

  @Override
  public Object currentItem() {
    checkForComodification();
    return arrayList.get(cursor);
  }

  private void checkForComodification() {
    if (arrayList.modCount != expectedModCount)
        throw new ConcurrentModificationException();
  }
}

//代码示例
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> iterator = names.iterator();
    iterator.next();
    names.remove("a");
    iterator.next();//抛出ConcurrentModificationException异常
  }
}
```

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
```java
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> iterator = names.iterator();
    iterator.next();
    iterator.remove();
    iterator.remove(); //报错，抛出IllegalStateException异常
  }
}
```

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
```java
public class ArrayList<E> {
  transient Object[] elementData;
  private int size;

  public Iterator<E> iterator() {
    return new Itr();
  }

  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;

    Itr() {}

    public boolean hasNext() {
      return cursor != size;
    }

    @SuppressWarnings("unchecked")
    public E next() {
      checkForComodification();
      int i = cursor;
      if (i >= size)
        throw new NoSuchElementException();
      Object[] elementData = ArrayList.this.elementData;
      if (i >= elementData.length)
        throw new ConcurrentModificationException();
      cursor = i + 1;
      return (E) elementData[lastRet = i];
    }

    public void remove() {
      if (lastRet < 0)
        throw new IllegalStateException();
      checkForComodification();

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

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
```java
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();
    iterator2.next(); // 运行结果?
  }
}
```

1. 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。