

10 | 集合类：坑满地的List列表操作

2020-03-31 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)




讲述：王少泽

时长 18:49 大小 17.24M



你好，我是朱晔。今天，我来和你说说 List 列表操作有哪些坑。

Pascal 之父尼克劳斯·维尔特（Niklaus Wirth），曾提出一个著名公式“程序 = 数据结构 + 算法”。由此可见，数据结构的重要性。常见的数据结构包括 List、Set、Map、Queue、Tree、Graph、Stack 等，其中 List、Set、Map、Queue 可以从广义上统称为集合类数据结构。

现代编程语言一般都会提供各种数据结构的实现，供我们开箱即用。Java 也是一样， 提供了集合类的各种实现。Java 的集合类包括 Map 和 Collection 两大类。Collection 包括 List、Set 和 Queue 三个小类，其中 List 列表集合是最重要也是所有业务代码都会用到

的。所以，今天我会重点介绍 List 的内容，而不会集中介绍 Map 以及 Collection 中其他小类的坑。


今天，我们就从把数组转换为 List 集合、对 List 进行切片操作、List 搜索的性能问题等几个方面着手，来聊聊其中最可能遇到的一些坑。

使用 Arrays.asList 把数据转换为 List 的三个坑

Java 8 中 Stream 流式处理的各种功能，大大减少了集合类各种操作（投影、过滤、转换）的代码量。所以，在业务开发中，我们常常会把原始的数组转换为 List 类数据结构，来继续展开各种 Stream 操作。


你可能也想到了，使用 Arrays.asList 方法可以把数组一键转换为 List，但其实没那么简单。接下来，就让我们看看其中的缘由，以及使用 Arrays.asList 把数组转换为 List 的几个坑。

在如下代码中，我们初始化三个数字的 int[] 数组，然后使用 Arrays.asList 把数组转换为 List：

 复制代码

```
1 int[] arr = {1, 2, 3};
2 List list = Arrays.asList(arr);
3 log.info("list:{} size:{} class:{}", list, list.size(), list.get(0).getClass());
```

但，这样初始化的 List 并不是我们期望的包含 3 个数字的 List。通过日志可以发现，这个 List 包含的其实是一个 int 数组，整个 List 的元素个数是 1，元素类型是整数数组。

 复制代码

```
1 12:50:39.445 [main] INFO org.geekbang.time.commonmistakes.collection.asList.As
```

其原因是，只能是把 int 装箱为 Integer，不可能把 int 数组装箱为 Integer 数组。我们知道，Arrays.asList 方法传入的是一个泛型 T 类型可变参数，最终 int 数组整体作为了一个对象成为了泛型类型 T：

```
1 public static <T> List<T> asList(T... a) {  
2     return new ArrayList<>(a);  
3 }
```

[复制代码](#)

直接遍历这样的 List 必然会出现 Bug，修复方式有两种，如果使用 Java8 以上版本可以使用 Arrays.stream 方法来转换，否则可以把 int 数组声明为包装类型 Integer 数组：

```
1 int[] arr1 = {1, 2, 3};  
2 List list1 = Arrays.stream(arr1).boxed().collect(Collectors.toList());  
3 log.info("list:{} size:{} class:{}", list1, list1.size(), list1.get(0).getClass());  
4  
5  
6 Integer[] arr2 = {1, 2, 3};  
7 List list2 = Arrays.asList(arr2);  
8 log.info("list:{} size:{} class:{}", list2, list2.size(), list2.get(0).getClass());
```

[复制代码](#)

修复后的代码得到如下日志，可以看到 List 具有三个元素，元素类型是 Integer：

```
1 13:10:57.373 [main] INFO org.geekbang.time.commonmistakes.collection.asList.As
```

[复制代码](#)

可以看到第一个坑是，**不能直接使用 Arrays.asList 来转换基本类型数组**。那么，我们获得了正确的 List，是不是就可以像普通的 List 那样使用了呢？我们继续往下看。


把三个字符串 1、2、3 构成的字符串数组，使用 Arrays.asList 转换为 List 后，将原始字符串数组的第二个字符修改为 4，然后为 List 增加一个字符串 5，最后数组和 List 会是什么呢？

```
1 String[] arr = {"1", "2", "3"};  
2 List list = Arrays.asList(arr);  
3 arr[1] = "4";  
4 try {  
5     list.add("5");  
6 } catch (Exception ex) {  
7     ex.printStackTrace();  
8 }
```

[复制代码](#)

```
9 log.info("arr:{} list:{}", Arrays.toString(arr), list);
```


可以看到，日志里有一个 `UnsupportedOperationException`，为 `List` 新增字符串 5 的操作失败了，而且把原始数组的第二个元素从 2 修改为 4 后，`asList` 获得的 `List` 中的第二个元素也被修改为 4 了：

 复制代码

```
1 java.lang.UnsupportedOperationException
2   at java.util.AbstractList.add(AbstractList.java:148)
3   at java.util.AbstractList.add(AbstractList.java:108)
4   at org.geekbang.time.commonmistakes.collection.aslist.AsListApplication.wrong
5   at org.geekbang.time.commonmistakes.collection.aslist.AsListApplication.main
6 13:15:34.699 [main] INFO org.geekbang.time.commonmistakes.collection.aslist.As
```

这里，又引出了两个坑。

第二个坑，**`Arrays.asList` 返回的 `List` 不支持增删操作**。`Arrays.asList` 返回的 `List` 并不是我们期望的 `java.util.ArrayList`，而是 `Arrays` 的内部类 `ArrayList`。`ArrayList` 内部类继承自 `AbstractList` 类，并没有覆写父类的 `add` 方法，而父类中 `add` 方法的实现，就是抛出 `UnsupportedOperationException`。相关源码如下所示：


 复制代码

```
1 public static <T> List<T> asList(T... a) {
2     return new ArrayList<>(a);
3 }
4
5 private static class ArrayList<E> extends AbstractList<E>
6     implements RandomAccess, java.io.Serializable
7 {
8     private final E[] a;
9
10
11     ArrayList(E[] array) {
12         a = Objects.requireNonNull(array);
13     }
14 ...
15
16     @Override
17     public E set(int index, E element) {
18         E oldValue = a[index];
19         a[index] = element;
20         return oldValue;
21     }
22 }
```

```
21     }
22     ...
23 }
24
25 public abstract class AbstractList<E> extends AbstractCollection<E> implements
26 ...
27 public void add(int index, E element) {
28     throw new UnsupportedOperationException();
29 }
30 }
```

第三个坑，**对原始数组的修改会影响到我们获得的那个 List**。看一下 ArrayList 的实现，可以发现 ArrayList 其实是直接使用了原始的数组。所以，我们要特别小心，把通过 Arrays.asList 获得的 List 交给其他方法处理，很容易因为共享了数组，相互修改产生 Bug。

修复方式比较简单，重新 new 一个 ArrayList 初始化 Arrays.asList 返回的 List 即可：

 复制代码

```
1 String[] arr = {"1", "2", "3"};
2 List list = new ArrayList(Arrays.asList(arr));
3 arr[1] = "4";
4 try {
5     list.add("5");
6 } catch (Exception ex) {
7     ex.printStackTrace();
8 }
9 log.info("arr:{} list:{}", Arrays.toString(arr), list);
```

修改后的代码实现了原始数组和 List 的“解耦”，不再相互影响。同时，因为操作的是真正的 ArrayList，add 也不再出错：

 复制代码

```
1 13:34:50.829 [main] INFO org.geekbang.time.commonmistakes.collection.asList.As
```

使用 List.subList 进行切片操作居然会导致 OOM?

业务开发时常常要对 List 做切片处理，即取出其中部分元素构成一个新的 List，我们通常会想到使用 List.subList 方法。但，和 Arrays.asList 的问题类似，List.subList 返回的子

List 不是一个普通的 ArrayList。这个子 List 可以认为是原始 List 的视图，会和原始 List 相互影响。如果不注意，很可能会因此产生 OOM 问题。接下来，我们就一起分析下其中的坑。

如下代码所示，定义一个名为 data 的静态 List 来存放 Integer 的 List，也就是说 data 的成员本身是包含了多个数字的 List。循环 1000 次，每次都从一个具有 10 万个 Integer 的 List 中，使用 subList 方法获得一个只包含一个数字的子 List，并把这个子 List 加入 data 变量：

 复制代码

```
1 private static List<List<Integer>> data = new ArrayList<>();
2
3 private static void oom() {
4     for (int i = 0; i < 1000; i++) {
5         List<Integer> rawList = IntStream.rangeClosed(1, 100000).boxed().collect(Collectors.toList());
6         data.add(rawList.subList(0, 1));
7     }
8 }
```


你可能会觉得，这个 data 变量里面最终保存的只是 1000 个具有 1 个元素的 List，不会占用很大空间，但程序运行不久就出现了 OOM：

 复制代码

```
1 Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
2   at java.util.Arrays.copyOf(Arrays.java:3181)
3   at java.util.ArrayList.grow(ArrayList.java:265)
```

出现 OOM 的原因是，循环中的 1000 个具有 10 万个元素的 List 始终得不到回收，因为它始终被 subList 方法返回的 List 强引用。那么，返回的子 List 为什么会强引用原始的 List，它们又有什么关系呢？我们再继续做实验观察一下这个子 List 的特性。

首先初始化一个包含数字 1 到 10 的 ArrayList，然后通过调用 subList 方法取出 2、3、4；随后删除这个 SubList 中的元素数字 3，并打印原始的 ArrayList；最后为原始的 ArrayList 增加一个元素数字 0，遍历 SubList 输出所有元素：

 复制代码

```
1 List<Integer> list = IntStream.rangeClosed(1, 10).boxed().collect(Collectors.toList());
```



```
2 List<Integer> subList = list.subList(1, 4);
3 System.out.println(subList);
4 subList.remove(1);
5 System.out.println(list);
6 list.add(0);
7 try {
8     subList.forEach(System.out::println);
9 } catch (Exception ex) {
10     ex.printStackTrace();
11 }
```

代码运行后得到如下输出：

 复制代码


```
1 [2, 3, 4]
2 [1, 2, 4, 5, 6, 7, 8, 9, 10]
3 java.util.ConcurrentModificationException
4   at java.util.ArrayList$SubList.checkForComodification(ArrayList.java:1239)
5   at java.util.ArrayList$SubList.listIterator(ArrayList.java:1099)
6   at java.util.AbstractList.listIterator(AbstractList.java:299)
7   at java.util.ArrayList$SubList.iterator(ArrayList.java:1095)
8   at java.lang.Iterable.forEach(Iterable.java:74)
```

可以看到两个现象：

原始 List 中数字 3 被删除了，说明删除子 List 中的元素影响到了原始 List；

尝试为原始 List 增加数字 0 之后再遍历子 List，会出现 ConcurrentModificationException。

我们分析下 ArrayList 的源码，看看为什么会是这样。

 复制代码

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
3 {
4     protected transient int modCount = 0;
5     private void ensureExplicitCapacity(int minCapacity) {
6         modCount++;
7         // overflow-conscious code
8         if (minCapacity - elementData.length > 0)
9             grow(minCapacity);
10    }
```

```

11  public void add(int index, E element) {
12      rangeCheckForAdd(index);
13
14      ensureCapacityInternal(size + 1); // Increments modCount!!
15      System.arraycopy(elementData, index, elementData, index + 1,
16                          size - index);
17      elementData[index] = element;
18      size++;
19  }
20
21  public List<E> subList(int fromIndex, int toIndex) {
22      subListRangeCheck(fromIndex, toIndex, size);
23      return new SubList(this, offset, fromIndex, toIndex);
24  }
25
26  private class SubList extends AbstractList<E> implements RandomAccess {
27      private final AbstractList<E> parent;
28      private final int parentOffset;
29      private final int offset;
30      int size;
31
32      SubList(AbstractList<E> parent,
33              int offset, int fromIndex, int toIndex) {
34          this.parent = parent;
35          this.parentOffset = fromIndex;
36          this.offset = offset + fromIndex;
37          this.size = toIndex - fromIndex;
38          this.modCount = ArrayList.this.modCount;
39      }
40
41      public E set(int index, E element) {
42          rangeCheck(index);
43          checkForComodification();
44          return l.set(index+offset, element);
45      }
46
47      public ListIterator<E> listIterator(final int index) {
48          checkForComodification();
49          ...
50      }
51
52      private void checkForComodification() {
53          if (ArrayList.this.modCount != this.modCount)
54              throw new ConcurrentModificationException();
55      }
56      ...
57  }
58 }

```


第一，ArrayList 维护了一个叫作 modCount 的字段，表示集合结构性修改的次数。所谓结构性修改，指的是影响 List 大小的修改，所以 add 操作必然会改变 modCount 的值。

第二，分析第 21 到 24 行的 subList 方法可以看到，获得的 List 其实是**内部类 SubList**，并不是普通的 ArrayList，在初始化的时候传入了 this。

第三，分析第 26 到 39 行代码可以发现，这个 SubList 中的 parent 字段就是原始的 List。SubList 初始化的时候，并没有把原始 List 中的元素复制到独立的变量中保存。我们可以认为 SubList 是原始 List 的视图，并不是独立的 List。双方对元素的修改会相互影响，而且 SubList 强引用了原始的 List，所以大量保存这样的 SubList 会导致 OOM。

第四，分析第 47 到 55 行代码可以发现，遍历 SubList 的时候会先获得迭代器，比较原始 ArrayList modCount 的值和 SubList 当前 modCount 的值。获得了 SubList 后，我们为原始 List 新增了一个元素修改了其 modCount，所以判等失败抛出 ConcurrentModificationException 异常。

既然 SubList 相当于原始 List 的视图，那么避免相互影响的修复方式有两种：

一种是，不直接使用 subList 方法返回的 SubList，而是重新使用 new ArrayList，在构造方法传入 SubList，来构建一个独立的 ArrayList；

另一种是，对于 Java 8 使用 Stream 的 skip 和 limit API 来跳过流中的元素，以及限制流中元素的个数，同样可以达到 SubList 切片的目的。

 复制代码

```
1 //方式一：
2 List<Integer> subList = new ArrayList<>(list.subList(1, 4));
3
4 //方式二：
5 List<Integer> subList = list.stream().skip(1).limit(3).collect(Collectors.toLi:
```

修复后代码输出如下：

 复制代码

```
1 [2, 3, 4]
2 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 2
```

可以看到，删除 SubList 的元素不再影响原始 List，而对原始 List 的修改也不会再出现 List 迭代异常。

一定要让合适的数据结构做合适的事情

在介绍 [并发工具](#) 时，我提到要根据业务场景选择合适的并发工具或容器。在使用 List 集合类的时候，不注意使用场景也会遇见两个常见误区。

第一个误区是，使用数据结构不考虑平衡时间和空间。

首先，定义一个只有一个 int 类型订单号字段的 Order 类：

[复制代码](#)


```
1 @Data
2 @NoArgsConstructor
3 @AllArgsConstructor
4 static class Order {
5     private int orderId;
6 }
```

然后，定义一个包含 elementCount 和 loopCount 两个参数的 listSearch 方法，初始化一个具有 elementCount 个订单对象的 ArrayList，循环 loopCount 次搜索这个 ArrayList，每次随机搜索一个订单号：

[复制代码](#)

```
1 private static Object listSearch(int elementCount, int loopCount) {
2     List<Order> list = IntStream.rangeClosed(1, elementCount).mapToObj(i -> new
3     IntStream.rangeClosed(1, loopCount).forEach(i -> {
4         int search = ThreadLocalRandom.current().nextInt(elementCount);
5         Order result = list.stream().filter(order -> order.getOrderId() == sea
6         Assert.assertTrue(result != null && result.getOrderId() == search);
7     });
8     return list;
9 }
```


随后，定义另一个 `mapSearch` 方法，从一个具有 `elementCount` 个元素的 `Map` 中循环 `loopCount` 次查找随机订单号。`Map` 的 `Key` 是订单号，`Value` 是订单对象：

 复制代码

```
1 private static Object mapSearch(int elementCount, int loopCount) {
2     Map<Integer, Order> map = IntStream.rangeClosed(1, elementCount).boxed().collect(Collectors.toMap(i -> i, i -> new Order(i)));
3     IntStream.rangeClosed(1, loopCount).forEach(i -> {
4         int search = ThreadLocalRandom.current().nextInt(elementCount);
5         Order result = map.get(search);
6         Assert.assertTrue(result != null && result.getOrderID() == search);
7     });
8     return map;
9 }
```

我们知道，搜索 `ArrayList` 的时间复杂度是 $O(n)$ ，而 `HashMap` 的 `get` 操作的时间复杂度是 $O(1)$ 。所以，要对大 `List` 进行单值搜索的话，可以考虑使用 `HashMap`，其中 `Key` 是要搜索的值，`Value` 是原始对象，会比使用 `ArrayList` 有非常明显的性能优势。

如下代码所示，对 100 万个元素的 `ArrayList` 和 `HashMap`，分别调用 `listSearch` 和 `mapSearch` 方法进行 1000 次搜索：

 复制代码

```
1 int elementCount = 1000000;
2 int loopCount = 1000;
3 Stopwatch stopWatch = new Stopwatch();
4 stopWatch.start("listSearch");
5 Object list = listSearch(elementCount, loopCount);
6 System.out.println(ObjectSizeCalculator.getObjectSize(list));
7 stopWatch.stop();
8 stopWatch.start("mapSearch");
9 Object map = mapSearch(elementCount, loopCount);
10 stopWatch.stop();
11 System.out.println(ObjectSizeCalculator.getObjectSize(map));
12 System.out.println(stopWatch.prettyPrint());
```

可以看到，仅仅是 1000 次搜索，`listSearch` 方法耗时 3.3 秒，而 `mapSearch` 耗时仅仅 108 毫秒。

 复制代码

1 20861992

```

2 72388672
3 Stopwatch '': running time = 3506699764 ns
4 -----
5 ns          %      Task name
6 -----
7 3398413176  097%  listSearch
8 108286588   003%  mapSearch

```

即使我们要搜索的不是单值而是条件区间，也可以尝试使用 HashMap 来进行“搜索性能优化”。如果你的条件区间是固定的话，可以提前把 HashMap 按照条件区间进行分组，Key 就是不同的区间。

的确，如果业务代码中有频繁的大 ArrayList 搜索，使用 HashMap 性能会好很多。类似，如果要对大 ArrayList 进行去重操作，也不建议使用 contains 方法，而是可以考虑使用 HashSet 进行去重。说到这里，还有一个问题，使用 HashMap 是否会牺牲空间呢？

为此，我们使用 ObjectSizeCalculator 工具打印 ArrayList 和 HashMap 的内存占用，可以看到 ArrayList 占用内存 21M，而 HashMap 占用的内存达到了 72M，是 List 的三倍多。进一步使用 MAT 工具分析堆可以再次证明，ArrayList 在内存占用上性价比很高，77% 是实际的数据（如第 1 个图所示，16000000/20861992），而 HashMap 的“含金量”只有 22%（如第 2 个图所示，16000000/72386640）。

Class Name	Objects	Shallow Heap ▾
<Regex>	<Numeric>	<Numeric>
org.geekbang.time.commonmistakes.collection.listvsmap.ListVsMapApplication\$Order	1,000,000	16,000,000
java.lang.Object[]	1	4,861,968
java.util.ArrayList	1	24
Total: 3 entries	1,000,002	20,861,992

Class Name	Objects	Shallow Heap ▾
<Regex>	<Numeric>	<Numeric>
java.util.HashMap\$Node	1,000,000	32,000,000
org.geekbang.time.commonmistakes.collection.listvsmap.ListVsMapApplication\$Order	1,000,000	16,000,000
java.lang.Integer	999,873	15,997,968
java.util.HashMap\$Node[]	1	8,388,624
java.util.HashMap	1	48
Total: 5 entries	2,999,875	72,386,640

所以，在应用内存吃紧的情况下，我们需要考虑是否值得使用更多的内存消耗来换取更高的性能。这里我们看到的是平衡的艺术，空间换时间，还是时间换空间，只考虑任何一个方面都是不对的。

第二个误区是，过于迷信教科书的大 O 时间复杂度。

数据结构中要实现一个列表，有基于连续存储的数组和基于指针串联的链表两种方式。在 Java 中，有代表性的实现是 ArrayList 和 LinkedList，前者背后的数据结构是数组，后者则是（双向）链表。


在选择数据结构的时候，我们通常会考虑每种数据结构不同操作的时间复杂度，以及使用场景两个因素。查看 [这里](#)，你可以看到数组和链表大 O 时间复杂度的显著差异：

对于数组，随机元素访问的时间复杂度是 $O(1)$ ，元素插入操作是 $O(n)$ ；

对于链表，随机元素访问的时间复杂度是 $O(n)$ ，元素插入操作是 $O(1)$ 。


那么，在大量的元素插入、很少的随机访问的业务场景下，是不是就应该使用 LinkedList 呢？接下来，我们写一段代码测试下两者随机访问和插入的性能吧。

定义四个参数一致的方法，分别对元素个数为 elementCount 的 LinkedList 和 ArrayList，循环 loopCount 次，进行随机访问和增加元素到随机位置的操作：

 复制代码


```
1 //LinkedList访问
2 private static void linkedListGet(int elementCount, int loopCount) {
3     List<Integer> list = IntStream.rangeClosed(1, elementCount).boxed().collect(Collectors.toList());
4     IntStream.rangeClosed(1, loopCount).forEach(i -> list.get(ThreadLocalRandom.current().nextInt(list.size())));
5 }
6
7 //ArrayList访问
8 private static void arrayListGet(int elementCount, int loopCount) {
9     List<Integer> list = IntStream.rangeClosed(1, elementCount).boxed().collect(Collectors.toList());
10    IntStream.rangeClosed(1, loopCount).forEach(i -> list.get(ThreadLocalRandom.current().nextInt(list.size())));
11 }
12
13 //LinkedList插入
14 private static void linkedListAdd(int elementCount, int loopCount) {
15     List<Integer> list = IntStream.rangeClosed(1, elementCount).boxed().collect(Collectors.toList());
16     IntStream.rangeClosed(1, loopCount).forEach(i -> list.add(ThreadLocalRandom.current().nextInt(list.size() + 1), list.get(i)));
17 }
18
19 //ArrayList插入
20 private static void arrayListAdd(int elementCount, int loopCount) {
21     List<Integer> list = IntStream.rangeClosed(1, elementCount).boxed().collect(Collectors.toList());
22     IntStream.rangeClosed(1, loopCount).forEach(i -> list.add(ThreadLocalRandom.current().nextInt(list.size() + 1), list.get(i)));
23 }
```

测试代码如下，10 万个元素，循环 10 万次：

 复制代码

```
1 int elementCount = 100000;
2 int loopCount = 100000;
3 Stopwatch stopWatch = new Stopwatch();
4 stopWatch.start("linkedListGet");
5 linkedListGet(elementCount, loopCount);
6 stopWatch.stop();
7 stopWatch.start("arrayListGet");
8 arrayListGet(elementCount, loopCount);
9 stopWatch.stop();
10 System.out.println(stopWatch.prettyPrint());
11
12
13 Stopwatch stopWatch2 = new Stopwatch();
14 stopWatch2.start("linkedListAdd");
15 linkedListAdd(elementCount, loopCount);
16 stopWatch2.stop();
17 stopWatch2.start("arrayListAdd");
18 arrayListAdd(elementCount, loopCount);
19 stopWatch2.stop();
20 System.out.println(stopWatch2.prettyPrint());
```

运行结果可能会让你大跌眼镜。在随机访问方面，我们看到了 ArrayList 的绝对优势，耗时只有 11 毫秒，而 LinkedList 耗时 6.6 秒，这符合上面我们所说的时间复杂度；**但，随机插入操作居然也是 LinkedList 落败，耗时 9.3 秒，ArrayList 只要 1.5 秒：**

 复制代码

```
1 -----
2 ns          %      Task name
3 -----
4 6604199591  100%   linkedListGet
5 011494583   000%   arrayListGet
6
7
8 Stopwatch '': running time = 10729378832 ns
9 -----
10 ns          %      Task name
11 -----
12 9253355484  086%   linkedListAdd
13 1476023348  014%   arrayListAdd
```

翻看 LinkedList 源码发现，插入操作的时间复杂度是 $O(1)$ 的前提是，你已经有了那个要插入节点的指针。但，在实现的时候，我们需要先通过循环获取到那个节点的 Node，然后再执行插入操作。前者也是有开销的，不可能只考虑插入操作本身的代价：

 复制代码

```
1 public void add(int index, E element) {
2     checkPositionIndex(index);
3
4     if (index == size)
5         linkLast(element);
6     else
7         linkBefore(element, node(index));
8 }
9
10 Node<E> node(int index) {
11     // assert isElementIndex(index);
12
13     if (index < (size >> 1)) {
14         Node<E> x = first;
15         for (int i = 0; i < index; i++)
16             x = x.next;
17         return x;
18     } else {
19         Node<E> x = last;
20         for (int i = size - 1; i > index; i--)
21             x = x.prev;
22         return x;
23     }
24 }
```

所以，对于插入操作，LinkedList 的时间复杂度其实也是 $O(n)$ 。继续做更多实验的话你会发现，在各种常用场景下，LinkedList 几乎都不能在性能上胜出 ArrayList。

讽刺的是，LinkedList 的作者约书亚·布洛克（Josh Bloch），在其推特上回复别人时说，虽然 LinkedList 是我写的但我从来不用，有谁会真的用吗？



Joshua Bloch

@joshbloch

Effective Java author, API Designer, Swell guy

□ Silicon Valley

□ joshbloch.com

□ 加入于 2009年7月



这告诉我们，任何东西理论上和实际上是有差距的，请勿迷信教科书的理论，最好在下定论之前实际测试一下。抛开算法层面不谈，由于 CPU 缓存、内存连续性问题，链表这种数据结构的实现方式对性能并不友好，即使在它最擅长的场景都不一定可以发挥威力。

重点回顾

今天，我分享了若干和 List 列表相关的错误案例，基本都是由“想当然”导致的。

第一，想当然认为，`Arrays.asList` 和 `List.subList` 得到的 List 是普通的、独立的 `ArrayList`，在使用时出现各种奇怪的问题。

`Arrays.asList` 得到的是 `Arrays` 的内部类 `ArrayList`，`List.subList` 得到的是 `ArrayList` 的内部类 `SubList`，不能把这两个内部类转换为 `ArrayList` 使用。

`Arrays.asList` 直接使用了原始数组，可以认为是共享“存储”，而且不支持增删元素；`List.subList` 直接引用了原始的 List，也可以认为是共享“存储”，而且对原始 List 直接进行结构性修改会导致 `SubList` 出现异常。

对 `Arrays.asList` 和 `List.subList` 容易忽略的是，新的 List 持有了原始数据的引用，可能会导致原始数据也无法 GC 的问题，最终导致 OOM。

第二，想当然认为，`Arrays.asList` 一定可以把所有数组转换为正确的 List。当传入基本类型数组的时候，List 的元素是数组本身，而不是数组中的元素。

第三，想当然认为，内存中任何集合的搜索都是很快的，结果在搜索超大 ArrayList 的时候遇到性能问题。我们考虑利用 HashMap 哈希表随机查找的时间复杂度为 $O(1)$ 这个特性来优化性能，不过也要考虑 HashMap 存储空间上的代价，要平衡时间和空间。

第四，想当然认为，链表适合元素增删的场景，选用 LinkedList 作为数据结构。在真实场景中读写增删一般是平衡的，而且增删不可能只是对头尾对象进行操作，可能在 90% 的情况下都得不到性能增益，建议使用之前通过性能测试评估一下。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

思考与讨论

最后，我给你留下与 ArrayList 在删除元素方面的坑有关的两个思考题吧。

1. 调用类型是 Integer 的 ArrayList 的 remove 方法删除元素，传入一个 Integer 包装类的数字和传入一个 int 基本类型的数字，结果一样吗？
2. 循环遍历 List，调用 remove 方法删除元素，往往会遇到 ConcurrentModificationException 异常，原因是什么，修复方式又是什么呢？

你还遇到过与集合类相关的其他坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 09 | 数值计算：注意精度、舍入和溢出问题

下一篇 加餐1 | 带你吃透课程中Java 8的那些重要知识点（上）

精选留言 (14)

 写留言



Darren 置顶

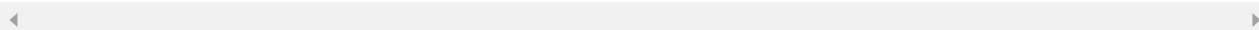
2020-03-31

哈哈，好巧，前两年有段时间比较闲，研究ArrayList和LinkedList，也对于所谓的ArrayList查询快，增删慢以及LinkedList查询慢，增删快提出过疑问，也做过类似的实验，然后去年给19年校招生入职培训的时候还专门分享过。要打破常规思维，多问为什么，要多听多看，多实验。

回答下问题：...

展开 

作者回复：点赞



5



2020-03-31

思考题1:

不一样,

remove 的两实现,

包装类型调用的是 `boolean remove(Object o);`方法, 本质上是寻找集合中是否有该元素, 有则删除。...

展开 ▾



3



秋水

2020-03-31

真的是迷信了LinkedList

展开 ▾



2



失火的夏天

2020-03-31

1.remove包装类数字是删除对象, 基本类型的int数字是删除下标。

2.好像是modcount和什么东西对不上来着, 具体忘记了, 看看其他大佬怎么说。解决这玩意就是改用迭代器遍历, 调用迭代器的remove方法。

话说到这个linkedlist, 真是感觉全面被arraylist压制。那这数据结构还留着干嘛呢? 为...

展开 ▾

作者回复: 1. 从完备性角度说sdk需要这样的数据结构

2. 就这个数据结构本身实现上并无问题

3. 也完全不是一无是处任何场景都没有优势

还是留着吧



2



eazonshaw

2020-03-31

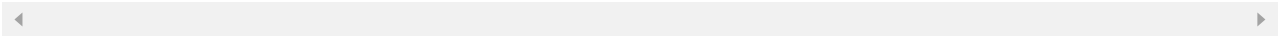
思考题:

1. 不一样。使用 ArrayList 的 remove方法, 如果传参是 Integer类型的话, 表示的是删除元素, 如果传参是int类型的话, 表示的是删除相对应索引位置的元素。

同时, 做了个小实验, 如果是String类型的ArrayList, 传参是Integer类型时, remove方法只是返回false, 视为元素不存在。...

展开 ▾

作者回复: 这个回复挺有我的写作风格 😊



👍 1



2020-03-31

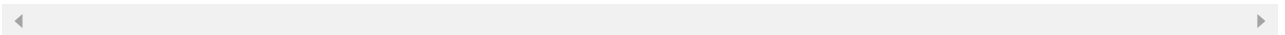
思考题2:

便利通常的实现方式for冒号的实现，其实底层还是用Iterator 删除元素，查看class文件大概是这样：

```
Iterator var2 = list.iterator();...
```

展开 ▾

作者回复: 🐣，源码ListRemoveApplication中也有我的实现



👍 1



2020-03-31

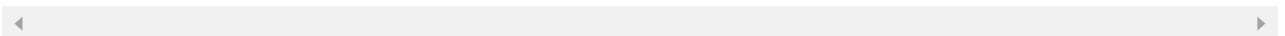
感触颇深：

Arrays的asList和subList，使用过程中需要谨慎，甚至可以考虑直接不用。

要熟悉数据结构。ArrayList 和 HashMap就是典型对比，ArrayList更适合随机访问，节约内存空间，大多数情况下性能不错。但，因为其本质上是数组，所以，无法实现快速找到想要的值。...

展开 ▾

作者回复: 很不错的总结



👍 1



终结者999号

2020-04-01

在转成stream的时候，linkedlist是不是要好于ArrayList呢？



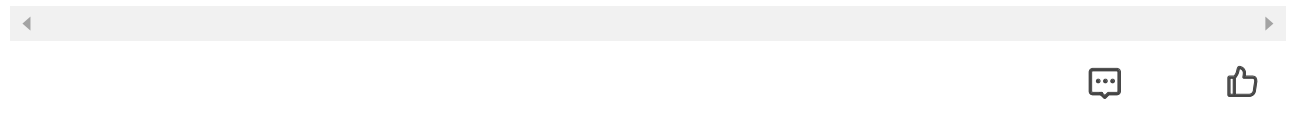
csyangchsh

2020-03-31

ArrayList分配的内存空间是连续的，对CPU Cache很友好。LinkedList还要包装成Node

e, 又增加了开销。这个测试使用JMH, 根据CPU Cache大小, 定义不同的元素个数, 可能更严谨一点。

作者回复: 是的, JMH做微基准测试更严谨一些



mgs2002

2020-03-31

第二题可以使用并发容器CopyOnWriteArrayList解决, 删除和添加都是在快照上面的, 不会影响原有的List

展开 ▾



Wiggle Wiggle

2020-03-31

这么看来 LinkedList 没有什么优势, 随机插入败了, 随机删除也差不多, 只剩尾插了。尾插其实也没有多少优势, 最多就是arrayList底层满了以后需要扩容, linkedList 不需要。估算下来, 大概只有在插入只是海量尾插、查询只是遍历的情况下才有点优势

展开 ▾



pedro

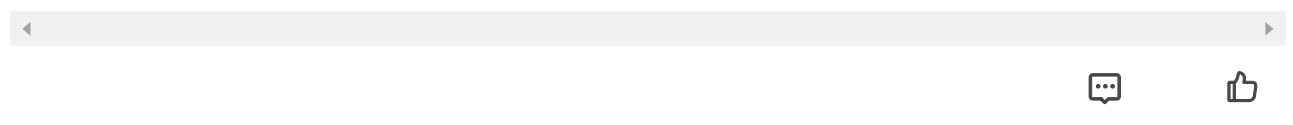
2020-03-31

第二个问题, 使用 for-each 或者 iterator 进行迭代删除 remove 时, 容易导致 next() 检测的 modCount 不等于 expectedModCount 从而引发 ConcurrentModificationException。

在单线程下, 推荐使用 next() 得到元素, 然后直接调用 remove(), 注意是无参的 remove; 多线程情况下还是使用并发容器吧 😊

展开 ▾

作者回复: 🙏

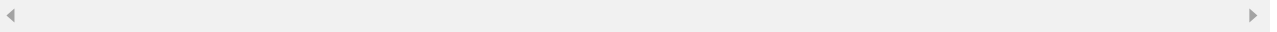


hellojd

2020-03-31

学习到了老师的探索精神, linedlist随机插入性能居然不高, 刷新了认知。

作者回复: :)



梦倚栏杆

2020-03-31

1. 刚好是看到案例时我想问的问题，答案知道，但是为什么呢？是规定为了区分？那为什么创建数组时可以自动装箱呢？

2. 用迭代器可以，但是为什么其实也不能说出个所以然

...

展开 ▾

作者回复: java 1.2的时候List接口已经是这样了，泛型后面出来的

