



Java性能调优实战  
刘超  
金山软件西山居技术经理

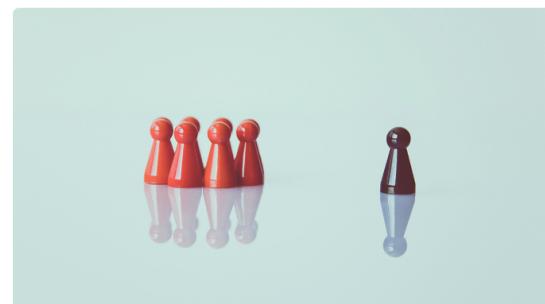
[查看详情](#)

5786 人已学习

[九局台词 \(1讲\)](#)[模块一 · 概述 \(2讲\)](#)[模块二 · Java编程性能调优 \(10讲\)](#)[03 | 字符串性能优化不容小觑，百M内存轻松存储几十G数据](#)[04 | 慎重使用正则表达式](#)[05 | ArrayList还是LinkedList? 使用不当性能差千倍](#)[加餐 | 推荐几款常用的性能测试工具](#)[06 | Stream如何提高遍历集合效率?](#)[07 | 深入浅出HashMap的设计与优化](#)[08 | 网络通信优化之I/O模型：如何解决](#)

## 05 | ArrayList还是LinkedList? 使用不当性能差千倍

刘超 2019-05-30



00:00 讲述: 李良 大小: 14.10M

你好，我是刘超。

集合作为一种存储数据的容器，是我们日常开发中使用最频繁的对象类型之一。JDK 为开发者提供了一系列的集合类型，这些集合类型使用不同的数据结构来实现。因此，不同的集合类型，使用场景也不同。

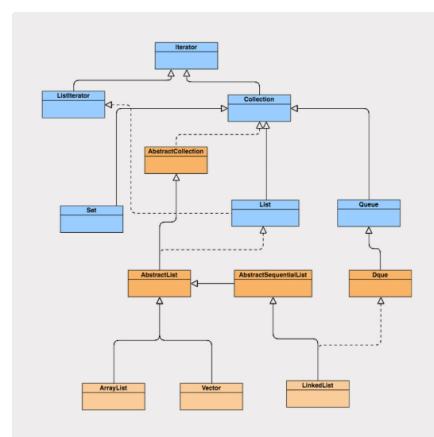
很多同学在面试的时候，经常会被问到集合的相关问题，比较常见的有 ArrayList 和 LinkedList 的区别。

相信大部分同学都能回答上：“ArrayList 是基于数组实现，LinkedList 是基于链表实现。”

而在回答使用场景的时候，我发现大部分同学的答案是：“ArrayList 和 LinkedList 在新增、删除元素时，LinkedList 的效率要高于 ArrayList，而在遍历的时候，ArrayList 的效率要高于 LinkedList。”这个回答是否准确呢？今天这一讲就带你验证。

### 初识 List 接口

在学习 List 集合类之前，我们先来通过这张图，看下 List 集合类的接口和类的实现关系：



我们可以看到 ArrayList、Vector、LinkedList 集合类继承了 AbstractList 抽象类，而 AbstractList 实现了 List 接口，同时也继承了 AbstractCollection 抽象类。ArrayList、Vector、LinkedList 又根据自我定位，分别实现了各自的功能。

ArrayList 和 Vector 使用了数组实现，这两者的实现原理差不多，LinkedList 使用了双向链表实现。基础铺垫就到这里，接下来，我们就详细地分析下 ArrayList 和 LinkedList 的源码实现。

### ArrayList 是如何实现的？

ArrayList 很常用，先来几道测试题，自检下你对 ArrayList 的了解程度。

**问题 1：**我们在查看 ArrayList 的实现类源码时，你会发现对象数组 elementData 使用了 transient 修饰，我们知道 transient 关键字修饰该属性，则表示该属性不会被序列化，然而我们并没有看到文档中说明 ArrayList 不能被序列化，这是为什么？

**问题 2：**我们在使用 ArrayList 进行新增、删除时，经常被提醒“使用 ArrayList 做新增删除操作会影响效率”。那是不是 ArrayList 在大量新增元素的场景下效率就一定会变慢呢？

**问题 3：**如果你使用 for 循环以及迭代循环遍历一个 ArrayList，你会使用哪种方式呢？原因是什

如果你对这几道测试题都没有一个全面的了解，那就跟我一起从数据结构、实现原理以及源码角度重新认识下 ArrayList 吧。

## 1.ArrayList 实现类

ArrayList 实现了 List 接口，继承了 AbstractList 抽象类，底层是数组实现的，并且实现了自增扩容数组大小。

ArrayList 还实现了 Cloneable 接口和 Serializable 接口，所以他可以实现克隆和序列化。

ArrayList 还实现了 RandomAccess 接口。你可能对这个接口比较陌生，不知道具体的用处。通过代码我们可以发现，这个接口其实是一个空接口，什么也没有实现，那 ArrayList 为什么要去实现它呢？

其实 RandomAccess 接口是一个标志接口，他标志着“只要实现该接口的 List 类，都能实现快速随机访问”。

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
3
```

## 2.ArrayList 属性

ArrayList 属性主要由数组长度 size、对象数组 elementData、初始化容量 defaultCapacity 等组成，其中初始化容量默认大小为 10。

```
1 // 默认初始化容量
2 private static final int DEFAULT_CAPACITY = 10;
3 // 对象数组
4 transient Object[] elementData;
5 // 数组长度
6 private int size;
7
```

从 ArrayList 属性来看，它没有被任何的多线程关键字修饰，但 elementData 被关键字 transient 修饰了。这就是我在上面提到的第一道测试题：transient 关键字修饰该字段则表示该属性不会被序列化，但 ArrayList 其实是实现了序列化接口，这到底是怎么回事呢？

这还得从“ArrayList 是基于数组实现”开始说起，由于 ArrayList 的数组是基于动态扩增的，所以并不是所有被分配的内存空间都存储了数据。

如果采用外部序列化法实现数组的序列化，会序列化整个数组。ArrayList 为了避免这些没有存储数据的内存空间被序列化，内部提供了两个私有方法 writeObject 以及 readObject 来自我完成序列化与反序列化，从而在序列化与反序列化数组时节省了空间和时间。

因此使用 transient 修饰数组，是防止对象数组被其他外部方法序列化。

## 3.ArrayList 构造函数

ArrayList 类实现了三个构造函数，第一个是创建 ArrayList 对象时，传入一个初始化值；第二个是默认创建一个空数组对象；第三个是传入一个集合类型进行初始化。

当 ArrayList 新增元素时，如果所存储的元素已经超过其已有大小，它会计算元素大小后再进行动态扩容，数组的扩容会导致整个数组进行一次内存复制。因此，我们在初始化 ArrayList 时，可以通过第一个构造函数合理指定数组初始大小，这样有助于减少数组的扩容次数，从而提高系统性能。

```
1 public ArrayList(int initialCapacity) {
2     // 初始化容量不为零时，将根据初始化值创建数组大小
3     if (initialCapacity > 0) {
4         this.elementData = new Object[initialCapacity];
5     } else if (initialCapacity == 0) { // 初始化容量为零时，使用默认的空数组
6         this.elementData = EMPTY_ELEMENTDATA;
7     } else {
8         throw new IllegalArgumentException("Illegal Capacity: " +
9             initialCapacity);
10    }
11 }
12
13 public ArrayList() {
14     // 初始化默认为空数组
15     this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
16 }
17
```

## 4.ArrayList 新增元素

ArrayList 新增元素的方法有两种，一种是直接将元素加到数组的末尾，另外一种是添加元素到任意位置。

```
1 public boolean add(E e) {
2     ensureCapacityInternal(size + 1); // Increments modCount!!
3     elementData[size++] = e;
4     return true;
5 }
6
7 public void add(int index, E element) {
8     rangeCheckForAdd(index);
9
10    ensureCapacityInternal(size + 1); // Increments modCount!!
11    System.arraycopy(elementData, index, elementData, index + 1,
12                      size - index);
13    elementData[index] = element;
14    size++;
15 }
16
```

两个方法的相同之处是在添加元素之前，都会先确认容量大小，如果容量够大，就不用进行扩容；如果容量不够大，就会按照原来数组的1.5倍大小进行扩容，在扩容之后需要将数组复制到新分配的内存地址。

```
1 private void ensureExplicitCapacity(int minCapacity) {
2     modCount++;
3
4     // overflow-conscious code
5     if (minCapacity - elementData.length > 0)
6         grow(minCapacity);
7 }
8 private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
9
10 private void grow(int minCapacity) {
11     // overflow-conscious code
12     int oldCapacity = elementData.length;
13     int newCapacity = oldCapacity + (oldCapacity >> 1);
14     if (newCapacity - minCapacity < 0)
15         newCapacity = minCapacity;
16     if (newCapacity - MAX_ARRAY_SIZE > 0)
17         newCapacity = hugeCapacity(minCapacity);
18     // minCapacity is usually close to size, so this is a win:
19     elementData = Arrays.copyOf(elementData, newCapacity);
20 }
21
```

当然，两个方法也有不同之处，添加元素到任意位置，会导致在该位置后的所有元素都需要重新排列，而将元素添加到数组的末尾，在没有发生扩容的前提下，是不会有关于元素复制排序的过程的。

这里你就可以找到第二道测试题的答案了。如果我们在初始化时就比较清楚存储数据的大小，就可以在 ArrayList 初始化时指定数组容量大小，并且在添加元素时，只在数组末尾添加元素，那么 ArrayList 在大量新增元素的场景下，性能并不会变差，反而比其他 List 集合的性能要好。

## 5.ArrayList 删除元素

ArrayList 的删除方法和添加任意位置元素的方法是有些相同的。ArrayList 在每一次有效的删除元素操作之后，都要进行数组的重组，并且删除的元素位置越靠前，数组重组的开销就越大。

```
1 public E remove(int index) {
2     rangeCheck(index);
3
4     modCount++;
5     E oldValue = elementData(index);
6
7     int numMoved = size - index - 1;
8     if (numMoved > 0)
9         System.arraycopy(elementData, index+1, elementData, index,
10                         numMoved);
11    elementData[--size] = null; // clear to let GC do its work
12
13    return oldValue;
14 }
15
```

## 6.ArrayList 遍历元素

由于 ArrayList 是基于数组实现的，所以在获取元素的时候是非常快捷的。

```
1 public E get(int index) {
2     rangeCheck(index);
3
4     return elementData(index);
5 }
6
7 E elementData(int index) {
8     return (E) elementData[index];
9 }
10
```

## LinkedList 是如何实现的？

虽然 LinkedList 与 ArrayList 都是 List 类型的集合，但 LinkedList 的实现原理却和 ArrayList 大相径庭，使用场景也不太一样。

LinkedList 是基于双向链表数据结构实现的，LinkedList 定义了一个 Node 结构，Node 结构中包含了 3 个部分：元素内容 item、前指针 prev 以及后指针 next，代码如下。

```
1 private static class Node<E> {
2     E item;
3     Node<E> next;
4     Node<E> prev;
5
6     Node(Node<E> prev, E element, Node<E> next) {
7         this.item = element;
8         this.next = next;
9         this.prev = prev;
10    }
11 }
```

总结一下，LinkedList 就是由 Node 结构对象连接而成的一个双向链表。在 JDK1.7 之前，LinkedList 中只包含了一个 Entry 结构的 header 属性，并在初始化的时候默认创建一个空的 Entry，用来做 header，前后指针指向自己，形成一个循环双向链表。

在 JDK1.7 之后，LinkedList 做了很大的改动，对链表进行了优化。链表的 Entry 结构换成了 Node，内部组成基本没有改变，但 LinkedList 里面的 header 属性去掉了，新增了一个 Node 结构的 first 属性和一个 Node 结构的 last 属性。这样做有以下几点好处：

- first/last 属性能更清晰地表达链表的链头和链尾概念;
- first/last 方式可以在初始化 LinkedList 的时候节省 new 一个 Entry;
- first/last 方式最重要的性能优化是链头和链尾的插入删除操作更快捷了。

这里同 ArrayList 的讲解一样，我将从数据结构、实现原理以及源码分析等几个角度带你深入了解 LinkedList。

## 1.LinkedList 实现类

LinkedList 类实现了 List 接口、Deque 接口，同时继承了 AbstractSequentialList 抽象类，LinkedList 既实现了 List 类型又有 Queue 类型的特点；LinkedList 也实现了 Cloneable 和 Serializable 接口，同 ArrayList 一样，可以实现克隆和序列化。

由于 LinkedList 存储数据的内存地址是不连续的，而是通过指针来定位不连续地址，因此，LinkedList 不支持随机快速访问，LinkedList 也就不能实现 RandomAccess 接口。

```
1 public class LinkedList<E>
2     extends AbstractSequentialList<E>
3     implements List<E>, Deque<E>, Cloneable, java.io.Serializable
4
```

复制代码

## 2.LinkedList 属性

我们前面讲到了 LinkedList 的两个重要属性 first/last 属性，其实还有一个 size 属性。我们可以看到这三个属性都被 transient 修饰了，原因很简单，我们在序列化的时候不会只对头尾进行序列化，所以 LinkedList 也是自行实现 readObject 和 writeObject 进行序列化与反序列化。

```
1 transient int size = 0;
2 transient Node<E> first;
3 transient Node<E> last;
4
```

复制代码

## 3.LinkedList 新增元素

LinkedList 添加元素的实现很简洁，但添加的方式却有很多种。默认的 add(E e) 方法是将添加的元素加到队尾，首先是将 last 元素置换到临时变量中，生成一个新的 Node 节点对象，然后将 last 引用指向新节点对象，之前的 last 对象的前指针指向新节点对象。

```
1 public boolean add(E e) {
2     linkLast(e);
3     return true;
4 }
5
6 void linkLast(E e) {
7     final Node<E> l = last;
8     final Node<E> newNode = new Node<E>(l, e, null);
9     last = newNode;
10    if (l == null)
11        first = newNode;
12    else
13        l.next = newNode;
14    size++;
15    modCount++;
16 }
17
```

复制代码

LinkedList 也有添加元素到任意位置的方法，如果我们是将元素添加到任意两个元素的中间位置，添加元素操作只会改变前后元素的前后指针，指针将会指向添加的新元素，所以相比 ArrayList 的添加操作来说，LinkedList 的性能优势明显。

```
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 void linkBefore(E e, Node<E> succ) {
11     // assert succ != null;
12     final Node<E> pred = succ.prev;
13     final Node<E> newNode = new Node<E>(pred, e, succ);
14     succ.prev = newNode;
15     if (pred == null)
16         first = newNode;
17     else
18         pred.next = newNode;
19     size++;
20     modCount++;
21 }
```

复制代码

## 4.LinkedList 删除元素

在 LinkedList 删除元素的操作中，我们首先要通过循环找到要删除的元素，如果要删除的位置处于 List 的前半段，就从前往后找；若其位置处于后半段，就从后往前找。

这样做的话，无论要删除较为靠前或较为靠后的元素都是非常高效的，但如果 List 拥有大量元素，移除的元素又在 List 的中间段，那效率相对来说会很低。

## 5.LinkedList 遍历元素

LinkedList 的获取元素操作实现跟 LinkedList 的删除元素操作基本类似，通过分前后半段来循

环查找到对应的元素。但是通过这种方式来查询元素是非常低效的，特别是在 for 循环遍历的情况下，每一次循环都会去遍历半个 List。

所以在 LinkedList 循环遍历时，我们可以使用 iterator 方式迭代循环，直接拿到我们的元素，而不需要通过循环查找 List。

## 总结

前面我们已经从源码的实现角度深入了解了 ArrayList 和 LinkedList 的实现原理以及各自的特點。如果你能充分理解这些内容，很多实际应用中的相关性能问题也就迎刃而解了。

就像如果现在还有人跟你说，“ArrayList 和 LinkedList 在新增、删除元素时，LinkedList 的效率要高于 ArrayList，而在遍历的时候，ArrayList 的效率要高于 LinkedList”，你还会表示赞同吗？

现在我们不妨通过几组测试来验证一下。这里因为篇幅限制，所以我就直接给出测试结果了，对应的测试代码你可以访问[Github](#)查看和下载。

### 1.ArrayList 和 LinkedList 新增元素操作测试

- 从集合头部位置新增元素
- 从集合中间位置新增元素
- 从集合尾部位置新增元素

测试结果 (花费时间):

- ArrayList>LinkedList
- ArrayList<LinkedList
- ArrayList<LinkedList

通过这组测试，我们可以知道 LinkedList 添加元素的效率未必高于 ArrayList。

由于 ArrayList 是数组实现的，而数组是一块连续的内存空间，在添加元素到数组头部的时候，需要对头部以后的数据进行复制重排，所以效率很低；而 LinkedList 是基于链表实现，在添加元素的时候，首先会通过循环查找到添加元素的位置，如果要添加的位置处于 List 的前半段，就从前往后找；若其位置处于后半段，则从后往前找。因此 LinkedList 添加元素到头部是非常高效的。

同上可知，ArrayList 在添加元素到数组中间时，同样有部分数据需要复制重排，效率也不是很高；LinkedList 将元素添加到中间位置，是添加元素最低效率的，因为靠近中间位置，在添加元素之前的循环查找是遍历元素最多的操作。

而在添加元素到尾部的操作中，我们发现，在没有扩容的情况下，ArrayList 的效率要高于 LinkedList，这是因为 ArrayList 在添加元素到尾部的时候，不需要复制重排数据，效率非常高。而 LinkedList 虽然也不用循环查找元素，但 LinkedList 中多了 new 对象以及变换指针指向对象的过程，所以效率要低于 ArrayList。

说明一下，这里我是基于 ArrayList 初始化容量足够，排除动态扩容数组容量的情况下进行的测试，如果有动态扩容的情况，ArrayList 的效率也会降低。

### 2.ArrayList 和 LinkedList 删除元素操作测试

- 从集合头部位置删除元素
- 从集合中间位置删除元素
- 从集合尾部位置删除元素

测试结果 (花费时间):

- ArrayList>LinkedList
- ArrayList<LinkedList
- ArrayList<LinkedList

ArrayList 和 LinkedList 删除元素操作测试的结果和添加元素操作测试的结果很接近，这是一样的原理，我在这里就不重复讲解了。

### 3.ArrayList 和 LinkedList 遍历元素操作测试

- for(); 循环
- 迭代器迭代循环

测试结果 (花费时间):

- ArrayList<LinkedList
- ArrayList≈LinkedList

我们可以看到，LinkedList 的 for 循环性能是最差的，而 ArrayList 的 for 循环性能是最好的。

这是因为 LinkedList 基于链表实现的，在使用 for 循环的时候，每一次 for 循环都会去遍历半个 List，所以严重影响了遍历的效率；ArrayList 则是基于数组实现的，并且实现了 RandomAccess 接口标志，意味着 ArrayList 可以实现快速随机访问，所以 for 循环效率非常高。

LinkedList 的迭代循环遍历和 ArrayList 的迭代循环遍历性能相当，也不会太差，所以在遍历 LinkedList 时，我们要切忌使用 for 循环遍历。

## 思考题

我们通过一个使用 for 循环遍历删除操作 ArrayList 数组的例子，思考下 ArrayList 数组的删除操作应该注意的一些问题。

```
1 public static void main(String[] args)
2 {
3     ArrayList<String> list = new ArrayList<String>();
4     list.add("a");
5     list.add("a");
6     list.add("b");
7     list.add("b");
8     list.add("c");
9     list.add("c");
10    remove(list); // 删掉指定的“b”元素
11
12    for(int i=0; i<list.size(); i++)("c")(){}(s : list)
13    {
14        System.out.println("element : " + s.list.get(i)
15    }
16}
17
```

从上面的代码来看，我定义了一个 ArrayList 数组，里面添加了一些元素，然后我通过 remove 删除指定的元素。请问以下两种写法，哪种是正确的？

写法 1：

```
1 public static void remove(ArrayList<String> list)
2 {
3     Iterator<String> it = list.iterator();
4
5     while (it.hasNext()) {
6         String str = it.next();
7
8         if (str.equals("b")) {
9             it.remove();
10        }
11    }
12}
13
14
```

写法 2：

```
1 public static void remove(ArrayList<String> list)
2 {
3     for (String s : list)
4     {
5         if (s.equals("b"))
6         {
7             list.remove(s);
8         }
9     }
10}
11
```

期待在留言区看到你的答案。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起学习。



© 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。

胡晓

由作者筛选后的优质留言将会公开显示。欢迎踊跃留言。

Command + Enter 发表

0/2000字

提交留言

精选留言(41)

陆离

对于arraylist和linkedlist的性能以前一直都是人云亦云，大家都说是这样那就这样吧，我也从来没有自己去验证过，没想过因操作位置的不同差异还挺大。  
当然这里面有一个前提，那就是arraylist的初始大小要足够大。

思考题是第一个是正确的，第二个虽然用的是foreach语法糖，遍历的时候用的也是迭代器遍历，但是在remove操作时使用的是原始数组list.remove，而不是迭代器的remove。这样就会造成modCount != expectedModCount，进而抛出异常。

作者回复：陆离同学一直保持非常稳定的发挥，答案非常准确！

2019-05-30

22

刘天若Warner

老师，为什么第二种就会抛出'ConcurrentModificationException 异常呢，我觉得第一种迭代器会抛这个异常啊

作者回复：for()循环[这里指的不是for(..)]是一个语法糖，这里会被解释为迭代器，在使用迭代器遍历时，ArrayList内部创建了一个内部迭代器iterator，在使用next()方法来取下一个元素时，会使用ArrayList里保存的一个用来记录List修改次数的变量modCount，与iterator保存了一个expectedModCount来表示期望的修改次数进行比较，如果不相等则会抛出异常；

而在foreach循环中调用list中的remove()方法，会走到fastRemove()方法，该方法不是iterator中的方法，而是ArrayList中的方法，在该方法只做了modCount++，而没有同步到expectedModCount。

当再次遍历时，会先调用内部类iterator中的hasNext(),再调用next(),在调用next()方法时，会对modCount和expectedModCount进行比较，此时两者不一致，就抛出了ConcurrentModificationException异常。

所以关键是用ArrayList的remove还是iterator中的remove。

2019-05-30

8

皮皮

第一种写法正确，第二种会报错，原因是上述两种写法都有用到list内部迭代器iterator，而在迭代器内部有一个属性是expectedmodcount，每次调用next和remove方法时会检查该值和list内部的modcount是否一致，不一致会报异常。问题中的第二种写法remove (e)，会在每次调用时modcount++，虽然迭代器的remove方法也会调用list的这个remove (e) 方法，但每次调用后还有一个expectedmodcount=modcount操作，所以下次调用next判断就不会报异常了。

作者回复：关键在用谁的remove方法。

2019-05-30

6

建国

老师，您好，linkList查找元素通过分前后半段，每次查找都要遍历半个list，怎么就知道元素是出于前半段还是后半段的呢？

作者回复：这个是随机的，因为分配的内存地址不是连续的。

2019-06-10

2

业余草

```
请问：List<A> list = new ArrayList<>();
for(int i=0;i<j;i<1000){
    A a = new A();
    list.add(a);
}
和
这个 List<A> list = new ArrayList<>();
A a;
for(int i=0;i<j;i<1000){
    a = new A();
    list.add(a);
}
```

效率上有差别吗？不说new ArrayList<>(); 初始化问题。单纯说创建对象这一块。谢谢！

作者回复：没啥区别的，可以实际操作试试

2019-05-31

2

计科一班

写法一正确。  
虽然都是调用了remove方法，但是两个remove方法是不同的。  
写法二是有可能会报ConcurrentModificationException异常。  
所以在ArrayList遍历删除元素时使用iterator方式或者普通的for循环。

作者回复：对的，使用普通循环也需要注意。

2019-05-30

2

mickle

第二种不行吧，会报并发修改异常的

2019-05-30

2

老杨同志

写法一正确，写法二会快速失败

2019-05-30

... 2

晓杰

写法2不正确，使用for循环遍历元素的过程中，如果删除元素，由于modCount != expectedModCount，会抛出ConcurrentModificationException异常

作者回复：对的！

2019-05-30

... 1

每天晒白牙

需要用迭代器方式删除  
for循环遍历删除会抛并发修改异常

作者回复：是的，不要使用迭代器循环时用ArrayList的remove方法，具体分析可以看留言区。

2019-05-30

... 1

我戒酒了

```
ArrayListTest 的这个测试方法笔误写错了吧  
public static void addFromMidTest(int DataNum) {  
    ArrayList<String> list = new ArrayList<String>(DataNum);  
    int i = 0;
```

```
    long timeStart = System.currentTimeMillis();  
    while (i < DataNum) {  
        int temp = list.size();  
        list.add(temp/2+"aaavv"); //正确写法list.add(temp/2, "aaavv");  
        i++;  
    }  
    long timeEnd = System.currentTimeMillis();  
  
    System.out.println("ArrayList从集合中间位置新增元素花费的时间" + (timeEnd - timeStart));  
}
```

作者回复：对的，感谢细心的你提醒

2019-07-02

... 1

吃胖了再减肥再吃再健身之谜之不...

老师，我用的测试代码试了好多次，在“从集合尾部位置新增元素”这个场景下，我测试的结果是“ArrayList<LinkedList>”，你代码里面的1000000，一千万次遍历时，会有少量的情况出现“ArrayList<LinkedList>”，所以我讲遍历次数增加到10000000，一千万次遍历，测试结果如下

第一次：

```
ArrayList从集合尾部位置新增元素花费的时间4690  
LinkedList从集合尾部位置新增元素花费的时间2942
```

第二次：

```
ArrayList从集合尾部位置新增元素花费的时间4655  
LinkedList从集合尾部位置新增元素花费的时间2798
```

第三次：

```
ArrayList从集合尾部位置新增元素花费的时间5126  
LinkedList从集合尾部位置新增元素花费的时间2960
```

从这个场景看来，大数据量遍历的情况下，LinkedList新增数据比较快，不知道我这个验证的结果是否正确，期待老师指教，谢谢！

作者回复：老师没有在查询时同时新增测试，考虑的只是单个场景下的测试。ArrayList新增需要固定一个初始化大小，如果默认初始化大小，则会在新增时出现扩容的情况，这样性能反而会降低。

2019-07-02

... 1

无忧

老师，你好。ArrayList删除元素时将对应位置元素设置为null。代码注释说GC会回收，请问回收是什么时候触发呢。

如果元素为空就会被回收的话，ArrayList在扩容时未使用的数组部分会不会也是空的被回收呢？

```
public E remove(int index) {  
    rangeCheck(index);  
  
    modCount++;  
    E oldValue = elementData(index);  
  
    int numMoved = size - index - 1;  
    if (numMoved > 0)  
        System.arraycopy(elementData, index+1, elementData, index,  
                         numMoved);  
    elementData[--size] = null; // clear to let GC do its work  
  
    return oldValue;  
}
```

2019-06-20

... 1

无忧

老师你好。ArrayLists删除元素时  
public E remove(int index) {  
 rangeCheck(index);

modCount++;

```
E oldValue = elementData(index);  
  
int numMoved = size - index - 1;  
if (numMoved > 0)  
    System.arraycopy(elementData, index+1, elementData, index,  
                    numMoved);  
elementData[--size] = null; // clear to let GC do its work  
  
return oldValue;  
}  
2019-06-20
```

码德纽@宝

我个人感觉arraylist和linkedlist性能最大的区别在于arraylist需要重新组排和扩容的开销。

2019-06-19

gavin

老师好，怎么确定操作集合是从头部、中间、还是尾部操作的呢？

作者回复：arraylist的add方法默认是从尾部操作，delete方法就是根据自己指定的位置来删除；  
linkedlist的add方法也是默认从尾部插入元素，delete方法也是根据指定的元素来删除。

2019-06-11

码德纽@宝

源码粘贴不完。大概描述一下

方法1 最后是通过调用迭代器remove(int index)，是直接删除对应下标的元素。  
方法2 最终是 如果b存在，那么调用list的remove (Object o) , list的remove是删除指定对象eqluse为true的第一个元素。

方法2 其实是转换为迭代器遍历，迭代器遍历的过程中使用了list的删除，导致迭代器下标越界。

顺便说下，自认为还有方法3  
public static void remove3(ArrayList<String> list){  
 while (list.remove("b")){  
 }  
}  
这种方式删除元素 好像也是可以的。

2019-06-10

Bruce

"之前的 last 对象的前指针指向新节点对象。"

这句话 为什么是前指针呢 代码里写的是 l.next = newNode;

2019-06-10

JasonZ

linkedlist使用iterator比普通for循环效率高，是由于遍历次数少，这是为什么？有什么文档可以参考么？

作者回复：因为for循环需要遍历链表，每循环一次就需要遍历一次指定节点前的数据，源码如下：

```
// 获取双向链表中指定位置的节点  
private Entry<E> entry(int index) {  
    if (index < 0 || index >= size)  
        throw new IndexOutOfBoundsException("Index: "+index+  
                                              ", Size: "+size);  
    Entry<E> e = header;  
    // 获取index处的节点。  
    // 若index < 双向链表长度的1/2,则从前向后查找;  
    // 否则,从后向前查找。  
    if (index < (size >> 1)) {  
        for (int i = 0; i <= index; i++)  
            e = e.next;  
    } else {  
        for (int i = size; i > index; i--)  
            e = e.previous;  
    }  
    return e;  
}
```

而iterator在第一次拿到一个数据后，之后的循环中会使用iterator中的next()方法采用的是顺序访问。

2019-06-09

农夫三拳

老师，可以在代码块多加一些注释吗？有些变量和方法不是很明白。原谅我比较菜...

编辑回复：收到，和老师说过了，这讲的会尽快加上，感谢你的建议！

2019-06-06

z.l

