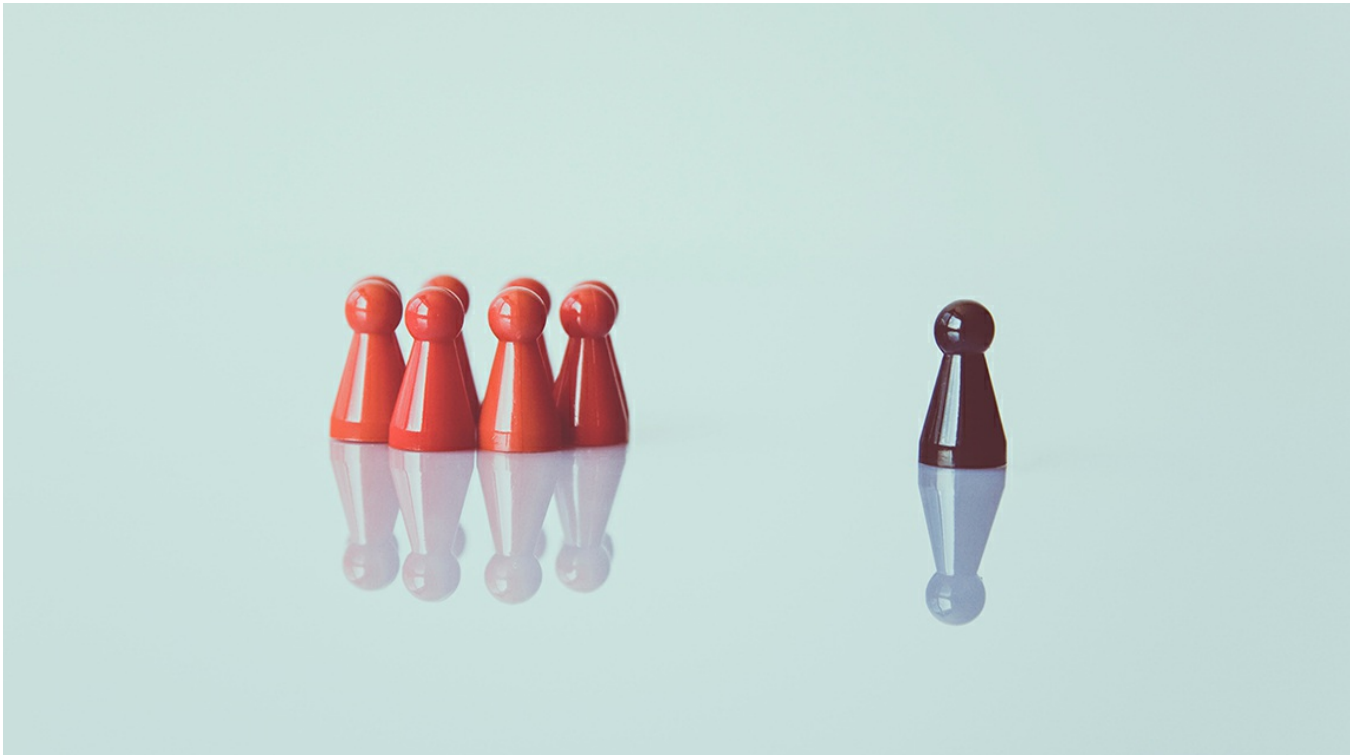


## 05 | ArrayList还是LinkedList？使用不当性能差千倍

2019-05-30 刘超



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。

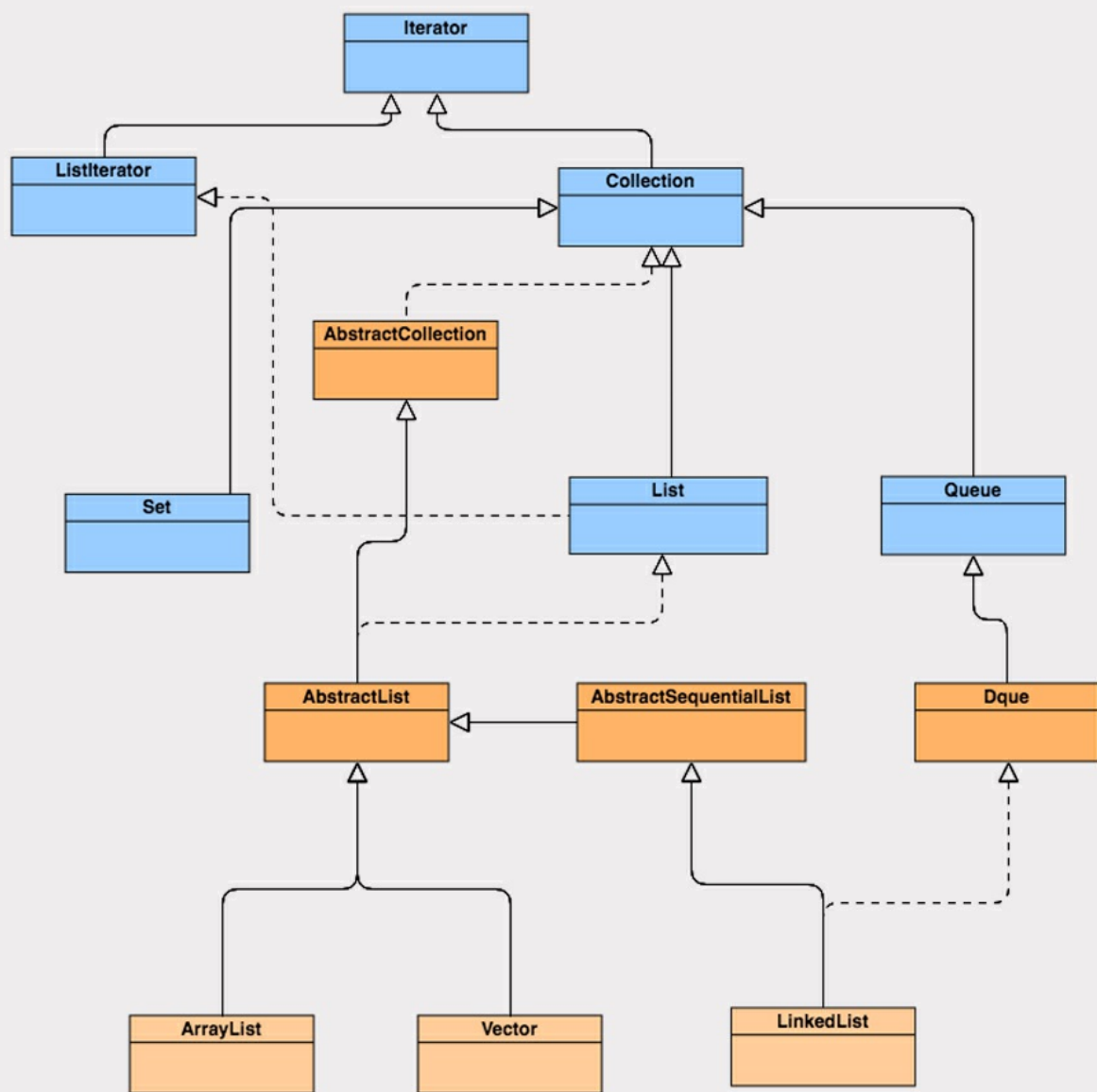
很多同学在面试的时候，经常会被问到集合的相关问题，比较常见的有ArrayList和LinkedList的区别。

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

如果你对这几道测试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”。

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

## 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**default\_capacity**等组成，其中初始化容量默认大小为10。

```
//默认初始化容量
private static final int DEFAULT_CAPACITY = 10;

//对象数组
transient Object[] elementData;

//数组长度
private int size;
```

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。

```
public ArrayList(int initialCapacity) {
    //初始化容量不为零时，将根据初始化值创建数组大小
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
    } else if (initialCapacity == 0) { //初始化容量为零时，使用默认的空数组
        this.elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new IllegalArgumentException("Illegal Capacity: "+
                                         initialCapacity);
    }
}

public ArrayList() {
    //初始化默认为空数组
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

### 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，另外一种添加元素到任意位置。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
        size - index);
    elementData[index] = element;
    size++;
}
```

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。

```

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}

private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}

```

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，是不会有元素复制排序过程的。

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。

```

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;
}

```

## 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。

```

public E get(int index) {
    rangeCheck(index);

    return elementData(index);
}

E elementData(int index) {
    return (E) elementData[index];
}

```

## 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。

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

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加快捷了。

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。

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



## 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。

```
transient int size = 0;
transient Node<E> first;
transient Node<E> last;
```

## 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 (Ee)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。

```
public boolean add(E e) {
    linkLast(e);
    return true;
}

void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

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。

```

public void add(int index, E element) {
    checkPositionIndex(index);

    if (index == size)
        linkLast(element);
    else
        linkBefore(element, node(index));
}

void linkBefore(E e, Node<E> succ) {
    // assert succ != null;
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e, succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    size++;
    modCount++;
}

```

#### 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点。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要高于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，就从后往前找。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時候，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。

```

public static void main(String[] args)
{
    ArrayList<String> list = new ArrayList<String>();
    list.add("a");
    list.add("a");
    list.add("b");
    list.add("b");
    list.add("c");
    list.add("c");
    remove(list); //删除指定的“b”元素

    for(int i=0; i<list.size(); i++)("c")()(s : list)
    {
        System.out.println("element : " + s)list.get(i)
    }
}

```

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:

```

public static void remove(ArrayList<String> list)
{
    Iterator<String> it = list.iterator();

    while (it.hasNext()) {
        String str = it.next();

        if (str.equals("b")) {
            it.remove();
        }
    }
}

```

写法2:

```
public static void remove(ArrayList<String> list)
{
    for (String s : list)
    {
        if (s.equals("b"))
        {
            list.remove(s);
        }
    }
}
```

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。



# Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



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。

精选留言



陆离

👍 19

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`的`remove`，而不是迭代器的`remove`。

这样就会造成`modCount != expectedModCount`，进而抛出异常。

2019-05-30

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



刘天若 Warner

7

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

2019-05-30

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`iteator`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



皮皮

6

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`exceptedmodcount`，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`exceptedmodcount=modcount`操作，所以后续调用`next`时判断就不会报异常了。

2019-05-30

作者回复

关键在用谁的`remove`方法。

2019-05-30





叮咛一班



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。

2019-05-30

作者回复

对的，使用普通循环也需要注意。

2019-05-30



mickle

2

第二种不行吧，会报并发修改异常的

2019-05-30



老杨同志

2

写法一正确，写法二会快速失败

2019-05-30



建国

1

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？

2019-06-10

作者回复

这个是随机的，因为分配的内存地址不是连续的。

2019-06-13



业余草

1

请问：List<A> list = new ArrayList<>();

```
for(int i=0;i++;i<1000){
```

```
A a = new A();
```

```
list.add(a);
```

```
}
```

和

```
和 这个 List<A> list = new ArrayList<>();
```

```
A a;
```

```
for(int i=0;i++;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！

2019-05-31

作者回复

没啥区别的，可以实际操作试试

2019-05-31





晓杰

1

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

2019-05-30

作者回复

对的！

2019-05-30



每天晒白牙

1

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

2019-05-30

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



gavin

0

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？

2019-06-11

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。

2019-06-13



[]

0

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equals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

0

”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

0

**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？

2019-06-09

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,则从前先后查找;
    // 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。

2019-06-10



农夫三拳

0

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...

2019-06-06

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

👍 0

第二种remove后加个return就不报错了吧

2019-06-05

作者回复

是不会报错了，但剩余的业务就无法进行下去了

2019-06-06



行者

👍 0

所以好像遇到这种问题，不能一上来就定性，ArrayList就是擅长随即访问，LinkedList就是擅长增加、删除。要基于不同的情况去分析问题，会让你有新的发现。

2019-06-01



约书亚

👍 0

实际场景使用中linked list的效率应该还要更低吧？因为要考虑到内存结构紧凑的问题。array list在删除时候移动元素，很可能是在一个cache line上操作，会很快，但linked list就未必了：写测试代码，linked list的元素总是连贯的。但实际使用场景一定是不连贯的。

2019-06-01

作者回复

很赞成，需要根据不同的业务场景考虑，实际场景中的问题更复杂。

2019-06-03



Loubobooo

👍 0

这一道我会。如果有看过阿里java规约就知道，在集合中进行remove操作时，不要在foreach循环里进行元素的remove/add操作。remove元素请使用Iterator方式，如果并发操作，需要对Iterator对象加锁。

<!-- 规约第七条 -->

2019-06-01

作者回复

👍

2019-06-01



清风拂面

👍 0

文稿关于从头部和尾部插入新元素所用时间那一块反了

2019-06-01

编辑回复

同学你好～你是想说ArrayList从头插入元素和从尾部插入元素的速度问题吗？从头部插入，存在数组复制，从尾部不存在。所以从尾部插入的速度要比头部快。

2019-06-01