链滴

代码审查：从 ArrayList 说线程安全

作者：mzlogin

原文链接：https://ld246.com/article/1615616954744

来源网站：链滴

本文从代码审查过程中发现的一个 ArrayList 相关的「线程安全」问题出发，来剖析和理解线程安全。

案例分析

前两天在代码 Review 的过程中，看到有小伙伴用了类似以下的写法：

```
List<String> resultList = new ArrayList<>();

paramList.parallelStream().forEach(v -> {
    String value = doSomething(v);
    resultList.add(value);
});
```

印象中 ArrayList 是线程不安全的，而这里会多线程改写同一个 ArrayList 对象，感觉这样的写法会问题，于是看了下 ArrayList 的实现来确认问题，同时复习下相关知识。

先贴个概念：

线程安全 是程式设计中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个程之间的共享变量，使程序功能正确完成。 ——维基百科

我们来看下 ArrayList 源码里与本话题相关的关键信息：

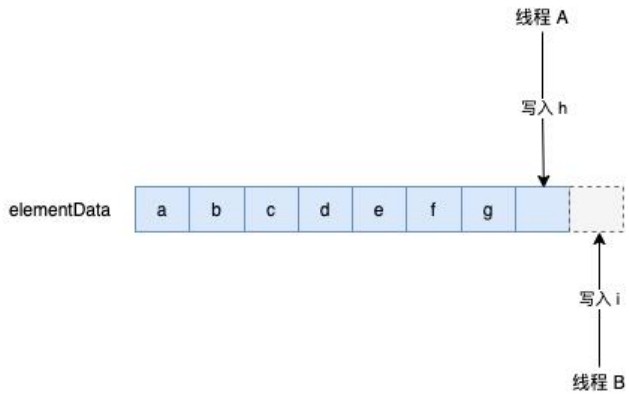
```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    // ...

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer...
     */
    transient Object[] elementData; // non-private to simplify nested class access

    /**
     * The size of the ArrayList (the number of elements it contains).
     */
    private int size;

    // ...

    /**
     * Appends the specified element to the end of this list...
     */
    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

    // ...
}
```

从中我们可以关注到关于 ArrayList 的几点信息：

1. 使用数组存储数据，即 elementData
2. 使用 int 成员变量 size 记录实际元素个数
3. add 方法逻辑与执行顺序：

- 执行 ensureCapacityInternal(size + 1)：确认elementData 的容量是否够用，不够用的话扩一半（申请一个新的大数组，将elementData 里的原有内容 copy 过去，然后将新的大数组赋值给elementData）
- 执行 elementData[size] = e;
- 执行 size++

为了方便理解这里讨论的「线程安全问题」，我们选一个最简单的执行路径来分析，假设有 A 和 B 个线程同时调用 ArrayList.add 方法，而此时 elementData 容量为 8，size 为 7，足以容纳一个新的元素，那么可能发生什么现象呢？



一种可能的执行顺序是：

- 线程 A 和 B 同时执行了 ensureCapacityInternal(size + 1)，因7 + 1 并没超过elementData 的量 8，所以并未扩容
- 线程 A 先执行 elementData[size++] = e;，此时size 变为 8
- 线程 B 执行 elementData[size++] = e;，因为elementData 数组长度为 8，却访问elementData 8]，数组下标越界

程序会抛出异常，无法正常执行完，根据前文提到的线程安全的定义，很显然这已经是属于线程不安的情况了。

构造示例代码验证

有了以上的理解之后，我们来写一段简单的示例代码，验证以上问题确实可能发生：

```
List<Integer> resultList = new ArrayList<>();
List<Integer> paramList = new ArrayList<>();
int length = 10000;
for (int i = 0; i < length; i++) {
    paramList.add(i);
}
paramList.parallelStream().forEach(resultList::add);
```

执行以上代码有可能表现正常，但更可能是遇到以下异常：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.j
va:62)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAcce
sorImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
    at java.util.concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:598)
    at java.util.concurrent.ForkJoinTask.reportException(ForkJoinTask.java:677)
    at java.util.concurrent.ForkJoinTask.invoke(ForkJoinTask.java:735)
    at java.util.stream.ForEachOps$ForEachOp.evaluateParallel(ForEachOps.java:160)
    at java.util.stream.ForEachOps$ForEachOp$OfRef.evaluateParallel(ForEachOps.java:174)
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:233)
    at java.util.stream.ReferencePipeline.forEach(ReferencePipeline.java:418)
    at java.util.stream.ReferencePipeline$Head.forEach(ReferencePipeline.java:583)
    at concurrent.ConcurrentTest.main(ConcurrentTest.java:18)
Caused by: java.lang.ArrayIndexOutOfBoundsException: 1234
    at java.util.ArrayList.add(ArrayList.java:465)
    at java.util.stream.ForEachOps$ForEachOp$OfRef.accept(ForEachOps.java:184)
    at java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1384)
    at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:482)
    at java.util.stream.ForEachOps$ForEachTask.compute(ForEachOps.java:291)
    at java.util.concurrent.CountedCompleter.exec(CountedCompleter.java:731)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1067)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1703)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:172)
```

从我这里试验的情况来看，length 值小的时候，因为达到容量边缘需要扩容的次数少，不易重现，将 ength 值调到比较大时，异常抛出率就很高了。

实际上除了抛出这种异常外，以上场景还可能造成数据覆盖/丢失、ArrayList 里实际存放的元素个数与 size 值不符等其它问题，感兴趣的同学可以继续挖掘一下。

解决方案

对这类问题常见的有效解决思路就是对共享的资源访问加锁。

我提出代码审查的修改意见后，小伙伴将文首代码里的

```
List<String> resultList = new ArrayList<>();
```

修改为了

```
List<String> resultList = Collections.synchronizedList(new ArrayList<>());
```

这样实际最终会使用 SynchronizedRandomAccessList，看它的实现类，其实里面也是加锁，它内持有一个 List，用 synchronized 关键字控制对 List 的读写访问，这是一种思路——使用线程安全的合类，对应的还可以使用 Vector 等其它类似的类来解决问题。

另外一种方思路是手动对关键代码段加锁，比如我们也可以将

```
resultList.add(value);
```

修改为

```
synchronized (mutex) {
    resultList.add(value);
}
```

小结

Java 8 的并行流提供了很方便的并行处理、提升程序执行效率的写法，我们在编码的过程中，对用到线程的地方要保持警惕，有意识地预防此类问题。

对应的，我们在做代码审查的过程中，也要对涉及到多线程使用的场景时刻绷着一根弦，在代码合入把好关，将隐患拒之门外。

参考

- 线程安全——维基百科