目录

15 CopyOnWriteArrayList 源码解析和设计思路

更新时间：2019-09-26 09:36:57

古之立大事者，不唯有超世之才，亦必有坚韧不拔之志。

——苏轼

引导语

在 ArrayList 的类注释上，JDK 就提醒了我们，如果要把 ArrayList 作为共享变量的话，是线程不安全的，推荐我们自己加锁或者使用 Collections.synchronizedList 方法，其实 JDK 还提供了另外一种线程安全的 List，叫做 CopyOnWriteArrayList，这个 List 具有以下特征：

1. 线程安全的，多线程环境下可以直接使用，无需加锁；
2. 通过锁 + 数组拷贝 + volatile 关键字保证了线程安全；
3. 每次数组操作，都会把数组拷贝一份出来，在新数组上进行操作，操作成功之后再赋值回去。

1 整体架构

从整体架构上来说，CopyOnWriteArrayList 数据结构和 ArrayList 是一致的，底层是个数组，只不过 CopyOnWriteArrayList 在对数组进行操作的时候，基本会分四步走：

1. 加锁；
2. 从原数组中拷贝出新数组；
3. 在新数组上进行操作，并把新数组赋值给数组容器；
4. 解锁。

除了加锁之外，CopyOnWriteArrayList 的底层数组还被 volatile 关键字修饰，意思是一旦数组被修改，其它线程立马能够感知到，代码如下：

```
private transient volatile Object[] array;
```

目录

1.1 类注释

我们看看从 CopyOnWriteArrayList 的类注释上能得到哪些信息：

1. 所有的操作都是线程安全的，因为操作都是在新拷贝数组上进行的；
2. 数组的拷贝虽然有一定的成本，但往往比一般的替代方案效率高；
3. 迭代过程中，不会影响到原来的数组，也不会抛出 ConcurrentModificationException 异常。

接着我们来看下 CopyOnWriteArrayList 的核心方法源码。

2 新增

新增有很多种情况，比如说：新增到数组尾部、新增到数组某一个索引位置、批量新增等等，操作的思路还是我们开头说的四步，我们拿新增到数组尾部的方法举例，来看看底层源码的实现：

```
// 添加元素到数组尾部
public boolean add(E e) {
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        // 得到所有的原数组
        Object[] elements = getArray();
        int len = elements.length;
        // 拷贝到新数组里面，新数组的长度是 + 1 的，因为新增会多一个元素
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 在新数组中进行赋值，新元素直接放在数组的尾部
        newElements[len] = e;
        // 替换掉原来的数组
        setArray(newElements);
        return true;
    // finally 里面释放锁，保证即使 try 发生了异常，仍然能够释放锁
    } finally {
        lock.unlock();
    }
}
```


从源码中，我们发现整个 add 过程都是在持有锁的状态下进行的，通过加锁，来保证同一时刻只能有一个线程能够对同一个数组进行 add 操作。

除了加锁之外，还会从老数组中创建出一个新数组，然后把老数组的值拷贝到新数组上，这时候就有一个问题：都已经加锁了，为什么需要拷贝数组，而不是在原来数组上面进行操作呢，原因主要为：

1. volatile 关键字修饰的是数组，如果我们简单的在原来数组上修改其中某几个元素的值，是无法触发可见性的，我们必须通过修改数组的内存地址才行，也就说要对数组进行重新赋值才行。
2. 在新的数组上进行拷贝，对老数组没有任何影响，只有新数组完全拷贝完成之后，外部才能访问到，降低了在赋值过程中，老数组数据变动的影响。

简单 add 操作是直接添加到数组的尾部，接着我们来看下指定位置添加元素的关键源码（部分源码）：

```
// 如果要插入的位置正好等于数组的末尾，直接拷贝数组即可
if (numMoved == 0)
    newElements = Arrays.copyOf(elements, len + 1);
else {
// 如果要插入的位置在数组的中间，就需要拷贝 2 次
// 第一次从 0 拷贝到 index。
// 第二次从 index+1 拷贝到末尾。
    newElements = new Object[len + 1];
    System.arraycopy(elements, 0, newElements, 0, index);
    System.arraycopy(elements, index, newElements, index + 1,
        numMoved);
}
// index 索引位置的值是空的，直接赋值即可。
newElements[index] = element;
// 把新数组的值赋值给数组的容器中
setArray(newElements);
```

从源码中可以看到，当插入的位置正好处于末尾时，只需要拷贝一次，当插入的位置处于中间时，此时我们会把原数组一分为二，进行两次拷贝操作。

最后还有个批量新增操作，源码我们就不贴了，底层也是拷贝数组的操作。

2.1 小结

从 add 系列方法可以看出，CopyOnWriteArrayList 通过加锁 + 数组拷贝+ volatile 来保证了线程安全，每一个要素都有着其独特的含义：

1. 加锁：保证同一时刻数组只能被一个线程操作；
2. 数组拷贝：保证数组的内存地址被修改，修改后触发 volatile 的可见性，其它线程可以立马知道数组已经被修改；
3. volatile：值被修改后，其它线程能够立马感知最新值。

3 个要素缺一不可，比如说我们只使用 1 和 3 ，去掉 2，这样当我们修改数组中某个值时，并不会触发 volatile 的可见特性的，只有当数组内存地址被修改后，才能触发把最新值通知给其他线程的特性。

3 删除

接着我们来看下指定数组索引位置删除的源码：

```
// 删除某个索引位置的数据
public E remove(int index) {
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        // 先得到老值
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        // 如果要删除的数据正好是数组的尾部，直接删除
        if (numMoved == 0)
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            // 如果删除的数据在数组的中间，分三步走
```

```
            Object[] newElements = new Object[len - 1];
            System.arraycopy(elements, 0, newElements, 0, index);
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
```

步骤分为三步：

1. 加锁；
2. 判断删除索引的位置，从而进行不同策略的拷贝；
3. 解锁。

代码整体的结构风格也比较统一：锁 + try finally +数组拷贝，锁被 final 修饰的，保证了在加锁过程中，锁的内存地址肯定不会被修改，finally 保证锁一定能够被释放，数组拷贝是为了删除其中某个位置的元素。

4 批量删除

数组的批量删除很有意思，接下来我们来看下 CopyOnWriteArrayList 的批量删除的实现过程：


```
// 批量删除包含在 c 中的元素
public boolean removeAll(Collection<?> c) {
    if (c == null) throw new NullPointerException();
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        int len = elements.length;
        // 说明数组有值，数组无值直接返回 false
        if (len != 0) {
            // newlen 表示新数组的索引位置，新数组中存在不包含在 c 中的元素
            int newlen = 0;
            Object[] temp = new Object[len];
            // 循环，把不包含在 c 里面的元素，放到新数组中
            for (int i = 0; i < len; ++i) {
                Object element = elements[i];
                // 不包含在 c 中的元素，从 0 开始放到新数组中
                if (!c.contains(element))
                    temp[newlen++] = element;
            }
            // 拷贝新数组，变相的删除了不包含在 c 中的元素
            if (newlen != len) {
                setArray(Arrays.copyOf(temp, newlen));
                return true;
            }
        }
        return false;
    } finally {
        lock.unlock();
```

目录

从源码中，我们可以看到，我们并不会直接对数组中的元素进行挨个删除，而是先对数组中的值进行循环判断，把我们不需要删除的数据放到临时数组中，最后临时数组中的数据就是我们不需要删除的数据。

不知道大家有木有似曾相识的感觉，ArrayList 的批量删除的思想也是和这个类似的，所以我们在需要删除多个元素的时候，最好都使用这种批量删除的思想，而不是采用在 for 循环中使用单个删除的方法，单个删除的话，在每次删除的时候都会进行一次数组拷贝(删除最后一个元素时不会拷贝)，很消耗性能，也耗时，会导致加锁时间太长，并发大的情况下，会造成大量请求在等待锁，这也会占用一定的内存。

5 其它方法

5.1 indexOf

indexOf 方法的主要用处是查找元素在数组中的下标位置，如果元素存在就返回元素的下标位置，元素不存在的话返回 -1，不但支持 null 值的搜索，还支持正向和反向的查找，我们以正向查找为例，通过源码来说明一下其底层的实现方式：

```
// o：我们需要搜索的元素
// elements：我们搜索的目标数组
// index：搜索的开始位置
// fence：搜索的结束位置
private static int indexOf(Object o, Object[] elements,
                           int index, int fence) {
    // 支持对 null 的搜索
    if (o == null) {
        for (int i = index; i < fence; i++)
            // 找到第一个 null 值，返回下标索引的位置
            if (elements[i] == null)
                return i;
    } else {
        // 通过 equals 方法来判断元素是否相等
        // 如果相等，返回元素的下标位置
        for (int i = index; i < fence; i++)
            if (o.equals(elements[i]))
                return i;
    }
    return -1;
}
```

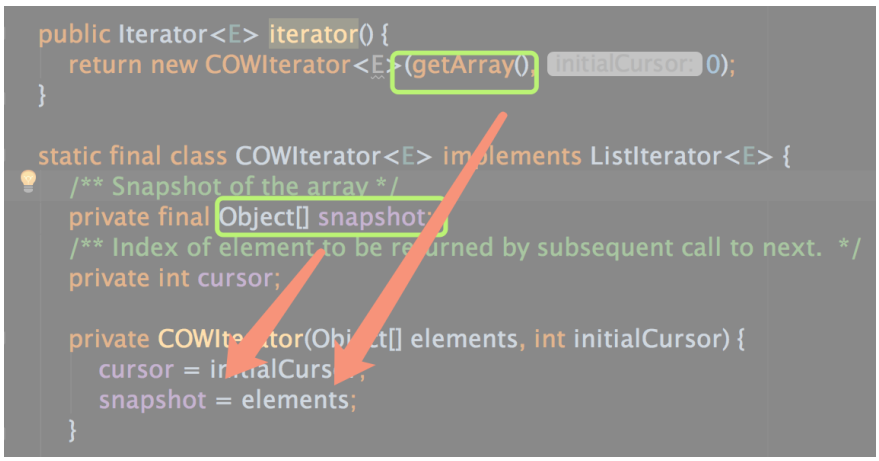

indexOf 方法在 CopyOnWriteArrayList 内部使用也比较广泛，比如在判断元素是否存在时（contains），在删除元素方法中校验元素是否存在时，都会使用到 indexOf 方法，indexOf 方法通过一次 for 循环来查找元素，我们在调用此方法时，需要注意如果找不到元素时，返回的是 -1，所以有可能我们会对这个特殊值进行判断。

5.2 迭代

在 CopyOnWriteArrayList 类注释中，明确说明了，在其迭代过程中，即使数组的原值被改变，也不会抛出 ConcurrentModificationException 异常，其根源在于数组的每次变动，都会生成新的数组，不会影响老数组，这样的话，迭代过程中，根本就不会发生迭代数组的变动，我们截几个图说明一下：

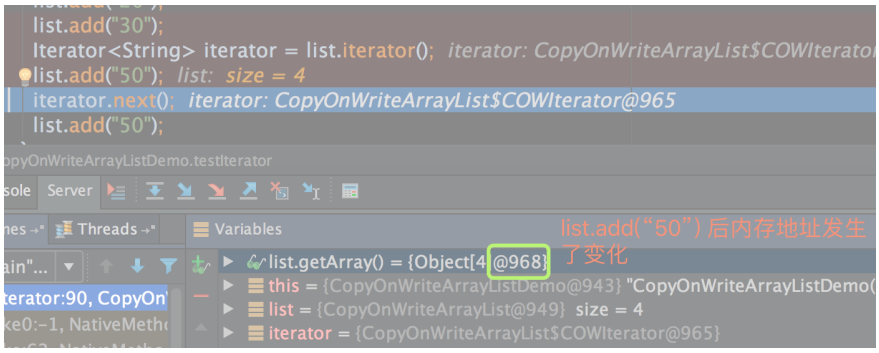
目录

可以看到迭代器是直接持有原数组的引用：



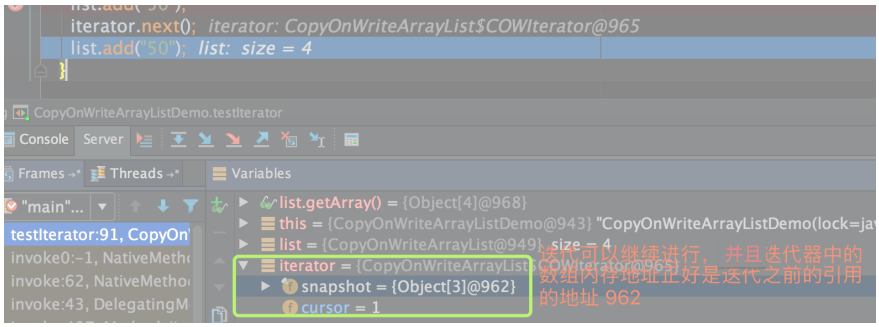
2. 我们写了一个 demo，在 CopyOnWriteArrayList 迭代之后，往 CopyOnWriteArrayList 里面新增值，从下图中可以看到在 CopyOnWriteArrayList 迭代之前，数组的内存地址是 962，请记住这个数字：

3. CopyOnWriteArrayList 迭代之后，我们使用 add(“50”) 代码给数组新增一个数据后，数组内存地址发生了变化，内存地址从原来的 962 变成了 968，这是因为 CopyOnWriteArrayList 的 add 操作，会生成新的数组，所以数组的内存地址发生了变化：



4. 迭代继续进行时，我们发现迭代器中的地址仍然是迭代之前引用的地址，是 962，而不是新的数组的内存地址：

目录



从上面 4 张截图，我们可以得到迭代过程中，即使 CopyOnWriteArrayList 的结构发生变动了，也不会抛出 ConcurrentModificationException 异常的原因：CopyOnWriteArrayList 迭代持有的是老数组的引用，而 CopyOnWriteArrayList 每次的数据变动，都会产生新的数组，对老数组的值不会产生影响，所以迭代也可以正常进行。

6 总结

当我们需要在线程不安全场景下使用 List 时，建议使用 CopyOnWriteArrayList，CopyOnWriteArrayList 通过锁 + 数组拷贝 + volatile 之间的相互配合，实现了 List 的线程安全，我们抛弃 Java 的这种实现，如果让我们自己实现，你又将如何实现呢？

← 14 简化工作：Guava Lists Maps 实际工作运用和源码

16 ConcurrentHashMap 源码解析和设计思路 →

精选留言 8

欢迎在这里发表留言，作者筛选后可公开显示

qq_oreo_5

这个重入锁是全局的老师 就是add remove 用的是一把锁，是不能add时remove的

0 回复 2019-11-27

Sicimike

老师，文中说“如果我们简单的在原来数组上修改其中某几个元素的值，是无法触发可见性的，我们必须通过修改数组的内存地址才行，也就说要对数组进行重新赋值才行”。是因为线程工作内存中只拷贝了数组的引用，如果只改变数组中的值，那么线程工作内存中的值（数组引用）没有改变，所以不会触发可见性。是这样吗？

0 回复 2019-11-12

文贺 回复 Sicimike

是的，底层原理主要是因为机器的 CPU 有多个导致的，我们在《03 Java 常用关键字理解》时有说到volatile关键字

回复 2019-11-17 10:54:19

目录

0 回复 2019-10-04

文贺 回复 **licly**

你说的操作数组是直接操作 array 的意思么？，array 是 private 的，无法直接操作哦。

回复 2019-10-08 12:54:19

licly 回复 **文贺**

可是add方法就是CopyOnWriteArrayList类里的方法，就算array用private修饰，也可以直接访问的呀。为什么要在add里面使用setArray(newElements)呢，而不是直接this.array=newElements

回复 2019-10-08 19:23:40

文贺 回复 **licly**

哦，明白了，你说是在 CopyOnWriteArrayList 中为什么不直接用 this.array = 这种形式，非要用 set，我个人觉得两者都可以用，可能和作者习惯有关，你在项目中应该也碰到这两种写法，在一个 DTO 中写一端逻辑，有的同学喜欢用 setXXX，有的同学喜欢用 XXX =。

回复 2019-10-08 19:42:23

你存在我脑海里

那老师，这种方式跟用Collections.synchronizeArrayList方式比较，哪种比较好呢？什么情景选择这两种不同的当时实现并发安全呢？

0 回复 2019-09-27

文贺 回复 **你存在我脑海里**

个人看法哈，可能不完整也不全。在需要List是线程安全的时候，在生产环境，这两种写法我都看过有人用过，两者没有谁好谁坏之分。但从两者的并发实现来看，有一个区别就是 synchronizedList 底层实现对读和写都加锁了(读写同一个锁，写时就无法读，读时也无法写)，而 CopyOnWriteArrayList 只对写进行了加锁，读是不加锁的，写时是可以读的，在读多的场景下，可能 CopyOnWriteArrayList 性能更好一些。 我个人观察，使用 CopyOnWriteArrayList 更多一些，我也更倾向于 CopyOnWriteArrayList，说的不全，仅供参考哈。

回复 2019-09-29 11:40:51

Elylic

ArrayList查询快而增删慢，CopyOnWriteArrayList的并发安全性能好！

0 回复 2019-09-27

---

soberyang

这个效率比Synchronize包裹方式的效率低很多吧

0 回复 2019-09-26

文贺 回复 **soberyang**

同学你好，这个主要看 synchronized 的代码怎么写了，看到了 synchronized 实现的代码才好比较。平时要用的话，如果数组不是很大的话，可以直接使用 CopyOnWriteArrayList，用的放心。

回复 2019-09-27 13:00:04

weixin_小马哥_04277912 回复 **soberyang**

所以他更适合读操作多于写操作的场景，因为读操作没有加锁。

回复 2019-09-27 16:47:12

目录

爱玩乒乓球的仲长芮澜

老师，可不可以在专栏后面加餐两篇，写一写注解和xml相关的源码，迫切需要呀！

1 回复 2019-09-26

文贺 回复 **爱玩乒乓球的仲长芮澜**

同学你好，注解和xml相关的源码范围太大了，你可以想几个你迫切需要的点，我写两篇手记是可以的，或者加我微信，不懂的可以随时问我：luanqiu0，加我备注一下来自慕课网哈。

回复 2019-09-27 13:04:29

海怪

老师这个迭代为什么不用考虑线程安全的问题呢？如果有人更新的数组，而我还在迭代旧的数据那不就会造成结果不一致了吗？

0 回复 2019-09-26

文贺 回复 **海怪**

同学你好，线程安全并不代表能得到最新实时的数据哈，正因为不管有没有更新，迭代时都是旧的数据，才保证了线程安全。

回复 2019-09-27 13:02:45

weixin_小马哥_04277912 回复 **海怪**

CopyOnWriteArrayList保证的是最终一致性，也就是当下一次获取的时候可以拿到最新的值，这也是volatile的关键，所以他的所有读操作都没有加锁。

回复 2019-09-27 16:45:08

千学不如一看，千看不如一练