33 CopyOnWriteArrayList 有什么特点?

本课时我们主要讲解 CopyOnWriteArrayList 有什么特点。

故事要从诞生 CopyOnWriteArrayList 之前说起。其实在 CopyOnWriteArrayList 出现之前，我们已经有了 ArrayList 和 LinkedList 作为 List 的数组和链表的实现，而且也有了线程安全的 Vector 和 Collections.synchronizedList() 可以使用。所以首先就让我们来看下线程安全的 Vector 的 size 和 get 方法的代码：

```
public synchronized int size() {
    return elementCount;
}
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    return elementData(index);
}
```

可以看出，Vector 内部是使用 synchronized 来保证线程安全的，并且锁的粒度比较大，都是方法级别的锁，在并发量高的时候，很容易发生竞争，并发效率相对比较低。在这一点上，Vector 和 Hashtable 很类似。

并且，前面这几种 List 在迭代期间不允许编辑，如果在迭代期间进行添加或删除元素等操作，则会抛出 ConcurrentModificationException 异常，这样的特点也在很多情况下给使用者带来了麻烦。

所以从 JDK1.5 开始，Java 并发包里提供了使用 CopyOnWrite 机制实现的并发容器 CopyOnWriteArrayList 作为主要的并发 List，CopyOnWrite 的并发集合还包括 CopyOnWriteArraySet，其底层正是利用 CopyOnWriteArrayList 实现的。所以今天我们以 CopyOnWriteArrayList 为突破口，来看一下 CopyOnWrite 容器的特点。

适用场景

- **读操作可以尽可能的快，而写即使慢一些也没关系**

在很多应用场景中，读操作可能会远远多于写操作。比如，有些系统级别的信息，往往只需要加载或者修改很少的次数，但是会被系统内所有模块频繁的访问。对于这种场景，我们最希望看到的就是读操作可以尽可能的快，而写即使慢一些也没关系。

- **读多写少**

黑名单是最典型的场景，假如我们有一个搜索网站，用户在这个网站的搜索框中，输入关键字搜索内容，但是某些关键字不允许被搜索。这些不能被搜索的关键字会被放在一个黑名单中，黑名单并不需要实时更新，可能每天晚上更新一次就可以了。当用户搜索时，会检查当前关键字在不在黑名单中，如果在，则提示不能搜索。这种读多写少的场景也很适合使用 CopyOnWrite 集合。

读写规则

- **读写锁的规则**

读写锁的思想是：读读共享、其他都互斥（写写互斥、读写互斥、写读互斥），原因是由于读操作不会修改原有的数据，因此并发读并不会有安全问题；而写操作是危险的，所以当写操作发生时，不允许有读操作加入，也不允许第二个写线程加入。

- **对读写锁规则的升级**

CopyOnWriteArrayList 的思想比读写锁的思想又更进一步。为了将读取的性能发挥到极致，CopyOnWriteArrayList 读取是完全不用加锁的，更厉害的是，**写入也不会阻塞读取操作，也就是说你可以在写入的同时进行读取**，只有写入和写入之间需要进行同步，也就是不允许多个写入同时发生，但是在写入发生时允许读取同时发生。这样一来，读操作的性能就会大幅度提升。

特点

- **CopyOnWrite的含义**

从 CopyOnWriteArrayList 的名字就能看出它是满足 CopyOnWrite 的 ArrayList，CopyOnWrite 的意思是说，当容器需要被修改的时候，不直接修改当前容器，而是先将当前容器进行 Copy，复制出一个新的容器，然后修改新的容器，**完成修改之后，再将原容器的引用指向新的容器。**这样就完成了整个修改过程。

这样做的好处是，CopyOnWriteArrayList 利用了“不变性”原理，因为容器每次修改都是创建新副本，所以对于旧容器来说，其实是不可变的，也是线程安全的，无需进一步的同步操作。我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素，也不会有修改。

CopyOnWriteArrayList 的所有修改操作（add，set等）都是通过创建底层数组的新副本来实现的，所以 CopyOnWrite 容器也是一种读写分离的思想体现，读和写使用不同的容器。

- **迭代期间允许修改集合内容**

我们知道 ArrayList 在迭代期间如果修改集合的内容，会抛出 ConcurrentModificationException 异常。让我们来分析一下 ArrayList 会抛出异常的原因。

在 ArrayList 源码里的 ListItr 的 next 方法中有一个 checkForComodification 方法，代码如下：

```
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```

这里会首先检查 modCount 是否等于 expectedModCount。modCount 是保存修改次数，每次我们调用 add、remove 或 trimToSize 等方法时它会增加，expectedModCount 是迭代器的变量，当我们创建迭代器时会初始化并记录当时的 modCount。后面迭代期间如果发现 modCount 和 expectedModCount 不一致，就说明有人修改了集合的内容，就会抛出异常。

和 ArrayList 不同的是，CopyOnWriteArrayList 的迭代器在迭代的时候，如果数组内容被修改了，CopyOnWriteArrayList 不会报 ConcurrentModificationException 的异常，因为迭代器使用的依然是旧数组，只不过迭代的内容可能已经过时了。演示代码如下：

```
/**
* 描述： 演示CopyOnWriteArrayList迭代期间可以修改集合的内容
*/
public class CopyOnWriteArrayListDemo {
    public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>(new Integer
```

```
        System.out.println(list); //[1, 2, 3]
        //Get iterator 1
        Iterator<Integer> itr1 = list.iterator();
        //Add one element and verify list is updated
        list.add(4);
        System.out.println(list); //[1, 2, 3, 4]
        //Get iterator 2
        Iterator<Integer> itr2 = list.iterator();
        System.out.println("====Verify Iterator 1 content====");
        itr1.forEachRemaining(System.out::println); //1,2,3
        System.out.println("====Verify Iterator 2 content====");
        itr2.forEachRemaining(System.out::println); //1,2,3,4
    }
}
```

这段代码会首先创建一个 CopyOnWriteArrayList，并且初始值被赋为 [1, 2, 3]，此时打印出来的结果很明显就是 [1, 2, 3]。然后我们创建一个叫作 itr1 的迭代器，创建之后再添加一个新的元素，利用 list.add() 方法把元素 4 添加进去，此时我们打印出 List 自然是 [1, 2, 3, 4]。我们再创建一个叫作 itr2 的迭代器，在下方把两个迭代器迭代产生的内容打印出来，这段代码的运行结果是：

```
[1, 2, 3]
[1, 2, 3, 4]
====Verify Iterator 1 content====
1
2
3
====Verify Iterator 2 content====
1
2
```

```
3

4
```

可以看出，这两个迭代器打印出来的内容是不一样的。第一个迭代器打印出来的是 [1, 2, 3]，而第二个打印出来的是 [1, 2, 3, 4]。虽然它们的打印时机都发生在第四个元素被添加之后，但它们的创建时机是不同的。由于迭代器 1 被创建时的 List 里面只有三个元素，后续无论 List 有什么修改，对它来说都是无感知的。

以上这个结果说明了，CopyOnWriteArrayList 的迭代器一旦被建立之后，如果往之前的 CopyOnWriteArrayList 对象中去新增元素，在迭代器中既不会显示出元素的变更情况，同时也不会报错，这一点和 ArrayList 是有很大区别的。

缺点

这些缺点不仅是针对 CopyOnWriteArrayList，其实同样也适用于其他的 CopyOnWrite 容器：

- **内存占用问题**

因为 CopyOnWrite 的写时复制机制，所以在进行写操作的时候，内存里会同时驻扎两个对象的内存，这一点会占用额外的内存空间。

- **在元素较多或者复杂的情况下，复制的开销很大**

复制过程不仅会占用双倍内存，还需要消耗 CPU 等资源，会降低整体性能。

- **数据一致性问题**

由于 CopyOnWrite 容器的修改是先修改副本，所以这次修改对于其他线程来说，并不是实时能看到的，只有在修改完之后才能体现出来。如果你希望写入的的数据马上能被其他线程看到，CopyOnWrite 容器是不适用的。

源码分析

- **数据结构**

```
/** 可重入锁对象 */
final transient ReentrantLock lock = new ReentrantLock();
```

```
/** CopyOnWriteArrayList底层由数组实现，volatile修饰，保证数组的可见性 */
private transient volatile Object[] array;
/**
* 得到数组
*/
final Object[] getArray() {
    return array;
}
/**
* 设置数组
*/
final void setArray(Object[] a) {
    array = a;
}
/**
* 初始化CopyOnWriteArrayList相当于初始化数组
*/
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}
```

在这个类中首先会有一个 ReentrantLock 锁，用来保证修改操作的线程安全。下面被命名为 array 的 Object[] 数组是被 volatile 修饰的，可以保证数组的可见性，这正是存储元素的数组，同样，我们可以从 getArray()、setArray 以及它的构造方法看出，CopyOnWriteArrayList 的底层正是利用数组实现的，这也符合它的名字。

- **add 方法**

```
public boolean add(E e) {
    // 加锁
    final ReentrantLock lock = this.lock;
```

```
        lock.lock();
        try {
            // 得到原数组的长度和元素
            Object[] elements = getArray();
            int len = elements.length;
            // 复制出一个新数组
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            // 添加时，将新元素添加到新数组中
            newElements[len] = e;
            // 将volatile Object[] array 的指向替换成新数组
            setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }
```

add 方法的作用是往 CopyOnWriteArrayList 中添加元素，是一种修改操作。首先需要利用 ReentrantLock 的 lock 方法进行加锁，获取锁之后，得到原数组的长度和元素，也就是利用 getArray 方法得到 elements 并且保存 length。之后利用 Arrays.copyOf 方法复制出一个新的数组，得到一个和原数组内容相同的新数组，并且把新元素添加到新数组中。完成添加动作后，需要转换引用所指向的对象，利用 setArray(newElements) 操作就可以把 volatile Object[] array 的指向替换成新数组，最后在 finally 中把锁解除。

总结流程：在添加的时候首先上锁，并复制一个新数组，增加操作在新数组上完成，然后将 array 指向到新数组，最后解锁。

上面的步骤实现了 CopyOnWrite 的思想：写操作是在原来容器的拷贝上进行的，并且在读取数据的时候不会锁住 list。而且可以看到，如果对容器拷贝操作的过程中有新的读线程进来，那么读到的还是旧的数据，因为在那个时候对象的引用还没有被更改。

下面我们来分析一下读操作的代码，也就是和 get 相关的三个方法，分别是 get 方法的两个重载和 getArray 方法，代码如下：

```
public E get(int index) {
    return get(getArray(), index);
}
final Object[] getArray() {
    return array;
}
private E get(Object[] a, int index) {
    return (E) a[index];
}
```

可以看出，get 相关的操作没有加锁，保证了读取操作的高速。

- **迭代器 COWIterator 类**

这个迭代器有两个重要的属性，分别是 Object[] snapshot 和 int cursor。其中 snapshot 代表数组的快照，也就是创建迭代器那个时刻的数组情况，而 cursor 则是迭代器的游标。迭代器的构造方法如下：

```
private COWIterator(Object[] elements, int initialCursor) {
    cursor = initialCursor;
    snapshot = elements;
}
```

可以看出，迭代器在被构建的时候，会把当时的 elements 赋值给 snapshot，而之后的迭代器所有的操作都基于 snapshot 数组进行的，比如：

```
public E next() {
    if (! hasNext())
        throw new NoSuchElementException();
    return (E) snapshot[cursor++];
}
```

在 next 方法中可以看到，返回的内容是 snapshot 对象，所以，后续就算原数组被修改，

这个 snapshot 既不会感知到，也不会受影响，执行迭代操作不需要加锁，也不会因此抛出异常。迭代器返回的结果，和创建迭代器的时候的内容一致。

以上我们对 CopyOnWriteArrayList 进行了介绍。我们分别介绍了在它诞生之前的 Vector 和 Collections.synchronizedList() 的特点，CopyOnWriteArrayList 的适用场景、读写规则，还介绍了它的两个特点，分别是写时复制和迭代期间允许修改集合内容。我们还介绍了它的三个缺点，分别是内存占用问题，在元素较多或者复杂的情况下复制的开销大问题，以及数据一致性问题。最后我们对于它的重要源码进行了解析

上一页 下一页