链滴

Java 中的写时复制容器

作者：kevin2020

原文链接：https://ld246.com/article/1514905087070

来源网站：链滴

Copy-On-Write简称COW，是一种用于程序设计中的优化策略。其基本思路是，从一开始大家都在享同一个内容，当某个人想要修改这个内容的时候，才会真正把内容Copy出去形成一个新的内容然再改，这是一种延时懒惰策略。从JDK1.5开始Java并发包里提供了两个使用CopyOnWrite机制实现并发容器,它们是CopyOnWriteArrayList和CopyOnWriteArraySet。CopyOnWrite容器非常有用，以在非常多的并发场景中使用到。

什么是CopyOnWrite容器

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离思想，读和写不同的容器。

CopyOnWriteArrayList的实现原理

在使用CopyOnWriteArrayList之前，我们先阅读其源码了解下它是如何实现的。以下代码是向ArrayList里添加元素，可以发现在添加的时候是需要加锁的，否则多线程写的时候会Copy出N个副本出来。

```
public boolean add(T e) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {

        Object[] elements = getArray();

        int len = elements.length;
        // 复制出新数组

        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 把新元素添加到新数组里

        newElements[len] = e;

        setArray(newElements);       // 把原数组引用指向新数组 很精髓

        return true;

    } finally {

        lock.unlock();

    }

}

final void setArray(Object[] a) {
    array = a;
}
```

读的时候不需要加锁，如果读的时候有多个线程正在向ArrayList添加数据，读还是会读到旧的数据，为写的时候不会锁住旧的ArrayList。

CopyOnWrite的应用场景

CopyOnWrite并发容器用于读多写少的并发场景。比如白名单，黑名单，商品类目的访问和更新场，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，但是某些关键字不许被搜索。这些不能被搜索的关键字会被放在一个黑名单当中，黑名单每天晚上更新一次。当用户搜时，会检查当前关键字在不在黑名单当中，如果在，则提示不能搜索。

CopyOnWrite的缺点

CopyOnWrite容器有很多优点，但是同时也存在两个问题，即内存占用问题和数据一致性问题。所在开发的时候需要注意一下。

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，内存里会同时驻扎个对象的内存，旧的对象和新写入的对象（注意:在复制的时候只是复制容器里的引用，只是在写的时会创建新对象添加到新容器里，而旧容器的对象还在使用，所以有两份对象内存）。如果这些对象占的内存比较大，比如说200M左右，那么再写入100M数据进去，内存就会占用300M，那么这个时候有可能造成频繁的Yong GC和Full GC。之前我们系统中使用了一个服务由于每晚使用CopyOnWrite制更新大对象，造成了每晚15秒的Full GC，应用响应时间也随之变长。

针对内存占用问题，可以通过压缩容器中的元素的方法来减少大对象的内存消耗，比如，如果元素全10进制的数字，可以考虑把它压缩成36进制或64进制。或者不使用CopyOnWrite容器，而使用其他并发容器，如ConcurrentHashMap。

数据一致性问题。CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。所如果你希望写入的的数据，马上能读到，请不要使用CopyOnWrite容器。

CopyOnWriteArraySet简介

它是线程安全的无序的集合，可以将它理解成线程安全的HashSet。有意思的是，CopyOnWriteArraSet和HashSet虽然都继承于共同的父类AbstractSet；但是，HashSet是通过散列表(HashMap)”实的，而CopyOnWriteArraySet则是通过“动态数组(CopyOnWriteArrayList)”实现的，并不是散列。

和CopyOnWriteArrayList类似，CopyOnWriteArraySet具有以下特性：

1. 它最适合于具有以下特征的应用程序：Set 大小通常保持很小，只读操作远多于可变操作，需要遍历期间防止线程间的冲突。

2. 它是线程安全的。

3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。

4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等 操作。

5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖不变的数组快照。

6. CopyOnWriteArraySet的“线程安全”机制，和CopyOnWriteArrayList一样，是通过volatile互斥锁来实现的。

CopyOnWriteArrayList简介

它相当于线程安全的ArrayList。和ArrayList一样，它是个可变数组；但是和ArrayList不同的时，它有以下特性：

1. 它最适合于具有以下特征的应用程序：List 大小通常保持很小，只读操作远多于可变操作，需要

遍历期间防止线程间的冲突。

2. 它是线程安全的。

3. 因为通常需要复制整个基础数组，所以可变操作（add()、set() 和 remove() 等等）的开销很大。

4. 迭代器支持hasNext(), next()等不可变操作，但不支持可变 remove()等操作。

5. 使用迭代器进行遍历的速度很快，并且不会与其他线程发生冲突。在构造迭代器时，迭代器依赖不变的数组快照。

CopyOnWriteArrayList原理和数据结构

CopyOnWriteArrayList的数据结构，如下文所示：

CopyOnWriteArrayList实现List接口

成员 lock reentrantlock

volatile array[]:object

说明：

1. CopyOnWriteArrayList实现了List接口，因此它是一个队列。

2. CopyOnWriteArrayList包含了成员lock。每一个CopyOnWriteArrayList都和一个互斥锁lock绑，通过lock，实现了对CopyOnWriteArrayList的互斥访问。

3. CopyOnWriteArrayList包含了成员array数组，这说明CopyOnWriteArrayList本质上通过数组现的。

下面从“动态数组”和“线程安全”两个方面进一步对CopyOnWriteArrayList的原理进行说明。

1. **CopyOnWriteArrayList的“动态数组”机制** -- 它内部有个“volatile数组”(array)来保持数据在“添加/修改/删除”数据时，都会新建一个数组，并将更新后的数据拷贝到新建的数组中，最后再该数组赋值给“volatile数组”。这就是它叫做CopyOnWriteArrayList的原因！CopyOnWriteArrayList就是通过这种方式实现的动态数组；不过正由于它在“添加/修改/删除”数据时，都会新建数组，以涉及到修改数据的操作，CopyOnWriteArrayList效率很

低；但是单单只是进行遍历查找的话，效率比较高。

2. **CopyOnWriteArrayList的“线程安全”机制** -- 是通过volatile和互斥锁来实现的。(01) CopyOWriteArrayList是通过“volatile数组”来保存数据的。一个线程读取volatile数组时，总能看到其它程对该volatile变量最后的写入；就这样，通过volatile提供了“读取到的数据总是最新的”这个机制的

保证。(02) CopyOnWriteArrayList通过互斥锁来保护数据。在“添加/修改/删除”数据时，会先“取互斥锁”，再修改完毕之后，先将数据更新到“volatile数组”中，然后再“释放互斥锁”；这样就达到了保护数据的目的。

接下来这几个方法是cowset实现时用到的

不存在则添加元素，可以发是直接新建一个数组，边判断边添加，若发现相同，则返回false，丢弃新的数组

```
public boolean addIfAbsent(E e) {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            // Copy while checking if already present.
           // This wins in the most common case where it is not present
            Object[] elements = getArray();
```

```
            int len = elements.length;
            Object[] newElements = new Object[len + 1];
            for (int i = 0; i < len; ++i) {
                if (eq(e, elements[i]))
                    return false; // exit, throwing away copy
               else
                    newElements[i] = elements[i];
            }
            newElements[len] = e;
           setArray(newElements);
            return true;
        } finally {
            lock.unlock();
        }
    }
```

将所有不存在的元素加入到list中，index方法在下方

```
public int addAllAbsent(Collectionextends E> c) {
    Object[] cs = c.toArray();
 if (cs.length == 0)
        return 0;
  Object[] uniq = new Object[cs.length];
 final ReentrantLock lock = this.lock;
  lock.lock();
 try {
        Object[] elements = getArray();
 int len = elements.length;
 int added = 0;
 for (int i = 0; i < cs.length; ++i) { // scan for duplicates
  Object e = cs[i];
 if (indexOf(e, elements, 0, len) < 0 &&
              indexOf(e, uniq, 0, added) < 0)
              uniq[added++] = e;
  }
        if (added > 0) {
            Object[] newElements = Arrays.copyOf(elements, len + added);
  System.arraycopy(uniq, 0, newElements, len, added);
  setArray(newElements);
  }
        return added;
  } finally {
        lock.unlock();
  }
}
```

元素查重 重复返回-1

```
* @param o element to search for
 * @param elements the array
 * @param index first index to search
 * @param fence one past last index to search
 * @return index of element, or -1 if absent
 */private static int indexOf(Object o, Object[] elements,
 int index, int fence) {
    if (o == null) {
```

```
        for (int i = index; i < fence; i++)
            if (elements[i] == null)
                return i;
    } else {
        for (int i = index; i < fence; i++)
            if (o.equals(elements[i]))
                return i;
    }
    return -1;
}
```