CopyOnWriteArrayList 源码解析 | MrBird

> CopyOnWriteArrayList 为线程安全的 ArrayList，这节分析下 CopyOnWriteArrayList 的源码，基于 JDK1.8。

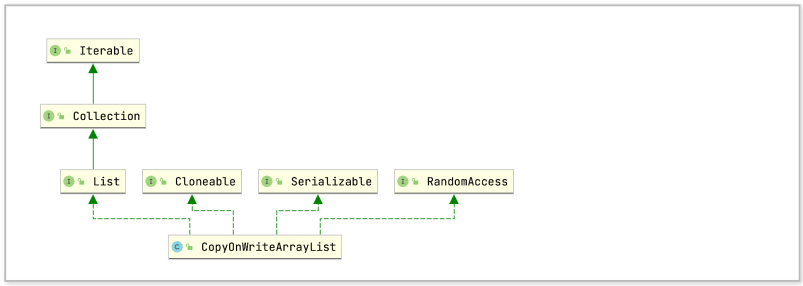
CopyOnWriteArrayList 源码解析

2020-08-18 | Visit count 1058054

CopyOnWriteArrayList 为线程安全的 ArrayList，这节分析下 CopyOnWriteArrayList 的源码，基于 JDK1.8。

类结构

CopyOnWriteArrayList 类关系图：



CopyOnWriteArrayList 实现了 List 接口的所有方法，主要包含如下两个成员变量：

```
// 可重入锁，用于对写操作加锁
final transient ReentrantLock lock = new ReentrantLock
```

```
// Object类型数组，存放数据，volatile修饰，目的是一个线程X
private transient volatile Object[] array;
```

CopyOnWriteArrayList 中并没有和容量有关的属性或者常量，下面通过对一些常用方法的源码解析，就可以知道原因。

方法解析

构造函数

CopyOnWriteArrayList() 空参构造函数：

```
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}

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

无参构造函数直接创建了一个长度为 0 的 Object 数组。

CopyOnWriteArrayList(Collection<? extends E> c) ：

```
public CopyOnWriteArrayList(Collection<? extends E> c)
    Object[] elements;
    if (c.getClass() == CopyOnWriteArrayList.class)
        // 如果集合类型就是CopyOnWriteArrayList，则直接将
        elements = ((CopyOnWriteArrayList<?>)c).getArra
    else {
        // 如果不是CopyOnWriteArrayList类型，则将集合转换
        elements = c.toArray();
        // 就如ArrayList源码分析所述那样，c.toArray()返回
        if (elements.getClass() != Object[].class)
            elements = Arrays.copyOf(elements, element
    }
    // 设置array值
    setArray(elements);
}
```

CopyOnWriteArrayList(E[] toCopyIn) ：

```
public CopyOnWriteArrayList(E[] toCopyIn) {
    // 入参为数组，拷贝一份赋值给array
```

```
        setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, (
    }
```

add(E e)

add(E e) 往 CopyOnWriteArrayList 末尾添加元素：

```
public boolean add(E e) {
    // 获取可重入锁
    final ReentrantLock lock = this.lock;
    // 上锁，同一时间内只能有一个线程进入
    lock.lock();
    try {
        // 获取当前array属性值
        Object[] elements = getArray();
        // 获取当前array数组长度
        int len = elements.length;
        // 复制一份新数组，新数组长度为当前array数组长度+1
        Object[] newElements = Arrays.copyOf(elements,
        // 在新数组末尾添加元素
        newElements[len] = e;
        // 新数组赋值给array属性
        setArray(newElements);
        return true;
    } finally {
        // 锁释放
        lock.unlock();
    }
}

final Object[] getArray() {
    return array;
}
```

可以看到，add 操作通过 ReentrantLock 来确保线程安全。通过 add 方法，我们也可以看出 CopyOnWriteArrayList 修改操作的基本思想为：复制一份新的数组，新数组长度刚好能够容纳下需要添加的元素；在新数组里进行操作；最后将新数组赋值给 array 属性，替换旧数组。这种思想也称为 “写时复制”，所以称为 CopyOnWriteArrayList。

此外，我们可以看到 CopyOnWriteArrayList 中并没有类似于 ArrayList 的 grow 方法扩容的操作。

add(int index, E element)

add(int index, E element) 指定下标添加指定元素：

```
public void add(int index, E element) {
    // 获取可重入锁
    final ReentrantLock lock = this.lock;
    // 上锁，同一时间内只能有一个线程进入
    lock.lock();
    try {
        // 获取当前array属性值
        Object[] elements = getArray();
         // 获取当前array数组长度
        int len = elements.length;
        // 下标检查
        if (index > len || index < 0)
            throw new IndexOutOfBoundsException("Index
                                                ", Size
        Object[] newElements;
        int numMoved = len - index;
        if (numMoved == 0)
            // numMoved为0，说明是在末尾添加，过程和add(E
            newElements = Arrays.copyOf(elements, len +
        else {
            // 否则创建一个新数组，数组长度为旧数组长度值+
            newElements = new Object[len + 1];
            // 分两次复制，分别将index之前和index+1之后的
            System.arraycopy(elements, 0, newElements,
            System.arraycopy(elements, index, newElemen
                             numMoved);
        }
        // 在新数组的index位置添加指定元素
        newElements[index] = element;
        // 新数组赋值给array属性，替换旧数组
        setArray(newElements);
    } finally {
        // 锁释放
        lock.unlock();
    }
}
```

set(int index, E element)

set(int index, E element) 设置指定位置的值：

```
public E set(int index, E element) {
    // 获取可重入锁
    final ReentrantLock lock = this.lock;
    // 上锁，同一时间内只能有一个线程进入
    lock.lock();
    try {
        // 获取当前array属性值
        Object[] elements = getArray();
        // 获取当前array指定index下标值
        E oldValue = get(elements, index);
        if (oldValue != element) {
                // 如果新值和旧值不相等
            int len = elements.length;
            // 复制一份新数组，长度和旧数组一致
```

```
                Object[] newElements = Arrays.copyOf(elemer
                // 修改新数组index下标值
                newElements[index] = element;
                // 新数组赋值给array属性，替换旧数组
                setArray(newElements);
            } else {
                // 即使新值和旧值一致，为了确保volatile语义，
                setArray(elements);
            }
            return oldValue;
        } finally {
            // 释放锁
            lock.unlock();
        }
    }

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

remove(int index)

remove(int index) 删除指定下标元素：

```
    public E remove(int index) {
        // 获取可重入锁
        final ReentrantLock lock = this.lock;
        // 上锁，同一时间内只能有一个线程进入
        try {
            // 获取当前array属性值
            Object[] elements = getArray();
            // 获取当前array长度
            int len = elements.length;
            // 获取旧值
            E oldValue = get(elements, index);
            int numMoved = len - index - 1;
            if (numMoved == 0)
                    // 如果删除的是最后一个元素，则将当前arra
                    // 新数组长度为旧数组长度-1，这样刚好截去
                setArray(Arrays.copyOf(elements, len - 1));
            else {
                    // 分段复制，将index前的元素和index+1后的
                    // 新数组长度为旧数组长度-1
                Object[] newElements = new Object[len - 1];
                System.arraycopy(elements, 0, newElements,
                System.arraycopy(elements, index + 1, newE
                                 numMoved);
                // 设置array
                setArray(newElements);
            }
            return oldValue;
        } finally {
            // 锁释放
            lock.unlock();
```

```
        }
    }
```

可以看到，CopyOnWriteArrayList 中的增删改操作都是在新数组中进行的，并且通过加锁的方式确保同一时刻只能有一个线程进行操作，操作完后赋值给 array 属性，替换旧数组，旧数组失去了引用，最终由 GC 回收。

get(int index)

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

可以看到， `get(int index)` 操作是分两步进行的：

1. 通过 `getArray()` 获取 array 属性值；
2. 获取 array 数组 index 下标值。

这个过程并没有加锁，所以在并发环境下可能出现如下情况：

1. 线程 1 调用 `get(int index)` 方法获取值，内部通过 `getArray()` 方法获取到了 array 属性值；
2. 线程 2 调用 CopyOnWriteArrayList 的增删改方法，内部通过 `setArray` 方法修改了 array 属性的值；
3. 线程 1 还是从旧的 array 数组中取值。

所以 `get` 方法是弱一致性的。

size()

```
public int size() {
    return getArray().length;
}
```

`size()` 方法返回当前 array 属性长度，因为 CopyOnWriteArrayList 中的 array 数组每次复制都刚好能够容

纳下所有元素，并不像 ArrayList 那样会预留一定的空间。所以 CopyOnWriteArrayList 中并没有 size 属性，元素的个数和数组的长度是相等的。

迭代器

```
public Iterator<E> iterator() {
    return new COWIterator<E>(getArray(), 0);
}
static final class COWIterator<E> implements ListIterat
    /** Snapshot of the array */
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent c
    private int cursor;

    private COWIterator(Object[] elements, int initialC
        cursor = initialCursor;
        snapshot = elements;
    }

    public boolean hasNext() {
        return cursor < snapshot.length;
    }
    ......
}
```

可以看到，迭代器也是弱一致性的，并没有在锁中进行。如果其他线程没有对 CopyOnWriteArrayList 进行增删改的操作，那么 snapshot 还是创建迭代器时获取的 array，但是如果其他线程对 CopyOnWriteArrayList 进行了增删改的操作，旧的数组会被新的数组给替换掉，但是 snapshot 还是原来旧的数组的引用：

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArra
list.add("hello");
Iterator<String> iterator = list.iterator();
list.add("world");
while (iterator.hasNext()){
    System.out.println(iterator.next());
}
```

输出结果仅为 hello。

总结

1. CopyOnWriteArrayList 体现了写时复制的思想，增删改操作都是在复制的新数组中进行的；

2. CopyOnWriteArrayList 的取值方法是弱一致性的，无法确保实时取到最新的数据；

3. CopyOnWriteArrayList 的增删改方法通过可重入锁确保线程安全；

4. CopyOnWriteArrayList 线程安全体现在多线程增删改不会抛出 `java.util.ConcurrentModificationException` 异常，并不能确保数据的强一致性；

5. 同一时刻只能有一个线程对 CopyOnWriteArrayList 进行增删改操作，而读操作没有限制，并且 CopyOnWriteArrayList 增删改操作都需要复制一份新数组，增加了内存消耗，所以 CopyOnWriteArrayList 适合读多写少的情况。

全文完