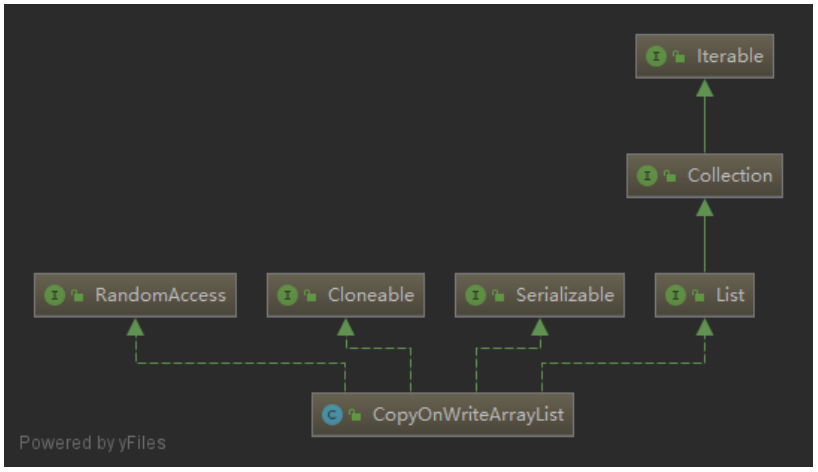
死磕 java集合之CopyOnWriteArrayList源码分析

原创 唐彤 彤哥读源码 2019-03-31

简介

CopyOnWriteArrayList是ArrayList的线程安全版本，内部也是通过数组实现，“写操作（add）时候复制旧数组到新数组”每次对数组的修改都完全拷贝一份新的数组来修改，修改完了再替换掉老数组，这样保证了只阻塞写操作，不阻塞读操作，实现读写分离。（“写操作”使用Reetrantlock加锁解锁，读取get操作不需要加锁，适合多读少写的环境）

继承体系



CopyOnWriteArrayList实现了List, RandomAccess, Cloneable, java.io.Serializable等接口。

CopyOnWriteArrayList实现了List，提供了基础的添加、删除、遍历等操作。

CopyOnWriteArrayList实现了RandomAccess，提供了随机访问的能力。

CopyOnWriteArrayList实现了Cloneable，可以被克隆。

CopyOnWriteArrayList实现了Serializable，可以被序列化。

源码解析

属性

```
/** 用于修改时加锁 */
```

```
final transient ReentrantLock lock = new ReentrantLock();

/** 真正存储元素的地方，只能通过getArray()/setArray()访问 */
private transient volatile Object[] array;
```

（1）lock

用于修改时加锁，使用transient修饰表示不自动序列化。

（2）array

真正存储元素的地方，使用transient修饰表示不自动序列化，使用volatile修饰表示一个线程对这个字段的修改另外一个线程立即可见。

问题：为啥没有size字段？且听后续分解。

CopyOnWriteArrayList()构造方法

创建空数组。

```
public CopyOnWriteArrayList() {
    // 所有对array的操作都是通过setArray()和getArray()进行
    setArray(new Object[0]);
}

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

CopyOnWriteArrayList(Collection c)构造方法

如果c是CopyOnWriteArrayList类型，直接把它的数组赋值给当前list的数组，注意这里是浅拷贝，两个集合共用同一个数组。

如果c不是CopyOnWriteArrayList类型，则进行拷贝把c的元素全部拷贝到当前list的数组中。

```
public CopyOnWriteArrayList(Collection<? extends E> c) {
    Object[] elements;
    if (c.getClass() == CopyOnWriteArrayList.class)
        // 如果c也是CopyOnWriteArrayList类型
        // 那么直接把它的数组拿过来使用。两个CopyOnWriterArrayList对象共用同一个 private transient volatile Object[] array数组
        elements = ((CopyOnWriteArrayList<?>)c).getArray();
    else {
        // 否则调用其toArray()方法将集合元素转化为数组
        elements = c.toArray();
        // 这里c.toArray()返回的不一定是Object[]类型
        // 详细原因见ArrayList里面的分析
        if (elements.getClass() != Object[].class)
            elements = Arrays.copyOf(elements, elements.length, Object[].class);
    }
    setArray(elements);
}
```

CopyOnWriteArrayList(E[] toCopyIn)构造方法

把toCopyIn的元素拷贝给当前list的数组。

```
public CopyOnWriteArrayList(E[] toCopyIn) {
    setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, Object[].class));
}
```

add(E e)方法

添加一个元素到末尾。

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

```
    try {
        // 获取旧数组
        Object[] elements = getArray();
        int len = elements.length;
        // 将旧数组元素拷贝到新数组中
        // 新数组大小是旧数组大小加1，所以就不存在ArrayList的扩容问题，数组刚好能放下所有的元素
        Object[] newElements = Arrays.copyOf(elements, len + 1);  调用c++的数组赋值
       // 将元素放在最后一位
        newElements[len] = e;
        setArray(newElements);旧数组没有任何引用，后期被GC回收
        return true;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

（1）加锁；

（2）获取元素数组；

（3）新建一个数组，大小为原数组长度加1，并把原数组元素拷贝到新数组；

（4）把新添加的元素放到新数组的末尾；

（5）把新数组赋值给当前对象的array属性，覆盖原数组；

（6）解锁；

add(int index, E element)方法

添加一个元素在指定索引处。

```
public void add(int index, E element) {
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        // 获取旧数组
        Object[] elements = getArray();
        int len = elements.length;
        // 检查是否越界，可以等于len
        if (index > len || index < 0)
            throw new IndexOutOfBoundsException("Index: "+index+
                                                ", Size: "+len);
        Object[] newElements;
        int numMoved = len - index;
        if (numMoved == 0)
            // 如果插入的位置是最后一位
            // 那么拷贝一个n+1的数组，其前n个元素与旧数组一致
            newElements = Arrays.copyOf(elements, len + 1);
        else {
            // 如果插入的位置不是最后一位
            // 那么新建一个n+1的数组
            newElements = new Object[len + 1];
            // 拷贝旧数组前index的元素到新数组中
            System.arraycopy(elements, 0, newElements, 0, index);
            // 将index及其之后的元素往后挪一位拷贝到新数组中
            // 这样正好index位置是空出来的
            System.arraycopy(elements, index, newElements, index + 1,
                             numMoved);
        }
        // 将元素放置在index处
        newElements[index] = element;
        setArray(newElements);
    } finally {
        // 释放锁
        lock.unlock();
    }
```

```
}
```

（1）加锁；

（2）检查索引是否合法，如果不合法抛出IndexOutOfBoundsException异常，注意这里index等于len也是合法的；

（3）如果索引等于数组长度（也就是数组最后一位再加1），那就拷贝一个len+1的数组；

（4）如果索引不等于数组长度，那就新建一个len+1的数组，并按索引位置分成两部分，索引之前（不包含）的部分拷贝到新数组索引之前（不包含）的部分，索引之后（包含）的位置拷贝到新数组索引之后（不包含）的位置，索引所在位置留空；

（5）把索引位置赋值为待添加的元素；

（6）把新数组赋值给当前对象的array属性，覆盖原数组；

（7）解锁；

addIfAbsent(E e)方法

添加一个元素如果这个元素不存在于集合中。

```
public boolean addIfAbsent(E e) {
    // 获取元素数组，取名为快照
    Object[] snapshot = getArray();
    // 检查如果元素不存在,直接返回false
    // 如果存在再调用addIfAbsent()方法添加元素
    return indexOf(e, snapshot, 0, snapshot.length) >= 0 ? false :
        addIfAbsent(e, snapshot);
}

private boolean addIfAbsent(E e, Object[] snapshot) {//
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        // 重新获取旧数组
        Object[] current = getArray();
        int len = current.length;
        // 如果快照与刚获取的数组不一致
        // 说明有修改
        if (snapshot != current) {
            // 重新检查元素是否在刚获取的数组里
            int common = Math.min(snapshot.length, len);
            for (int i = 0; i < common; i++)
                // 到这个方法里面了，说明元素不在快照里面
                if (current[i] != snapshot[i] && eq(e, current[i]))
                    return false;
            if (indexOf(e, current, common, len) >= 0)
                    return false;
        }
        // 拷贝一份n+1的数组
        Object[] newElements = Arrays.copyOf(current, len + 1);
        // 将元素放在最后一位
        newElements[len] = e;
        setArray(newElements);
        return true;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

（1）检查这个元素是否存在于数组快照中；

（2）如果存在直接返回false，如果不存在调用addIfAbsent(E e, Object[] snapshot)处理；

（3）加锁；

（4）如果当前数组不等于传入的快照，说明有修改，检查待添加的元素是否存在于当前数组中，如果存在直接返回false；

（5）拷贝一个新数组，长度等于原数组长度加1，并把原数组元素拷贝到新数组中；

（6）把新元素添加到数组最后一位；

（7）把新数组赋值给当前对象的array属性，覆盖原数组；

（8）解锁；

get(int index)

获取指定索引的元素，支持随机访问，时间复杂度为O(1)。

```
public E get(int index) {
    // 获取元素不需要加锁
    // 直接返回index位置的元素
    // 这里是没有做越界检查的，因为数组本身会做越界检查
    return get(getArray(), index);
}

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

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

（1）获取元素数组；

（2）返回数组指定索引位置的元素；

remove(int index)方法

删除指定索引位置的元素。

```
public E remove(int index) {
    final ReentrantLock lock = this.lock;
    // 加锁
    lock.lock();
    try {
        // 获取旧数组
        Object[] elements = getArray();
        int len = elements.length;
        E oldValue = get(elements, index);
        int numMoved = len - index - 1;
        if (numMoved == 0)
            // 如果移除的是最后一位
            // 那么直接拷贝一份n-1的新数组，最后一位就自动删除了
            setArray(Arrays.copyOf(elements, len - 1));
        else {
            // 如果移除的不是最后一位
            // 那么新建一个n-1的新数组
            Object[] newElements = new Object[len - 1];
            // 将前index的元素拷贝到新数组中
            System.arraycopy(elements, 0, newElements, 0, index);
            // 将index后面(不包含)的元素往前挪一位
            // 这样正好把index位置覆盖掉了，相当于删除了
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            setArray(newElements);
        }
        return oldValue;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

（1）加锁；

（2）获取指定索引位置元素的旧值；

（3）如果移除的是最后一位元素，则把原数组的前len-1个元素拷贝到新数组中，并把新数组赋值给当前对象的数组属性；

（4）如果移除的不是最后一位元素，则新建一个len-1长度的数组，并把原数组除了指定索引位置的元素全部拷贝到新数组中，并把新数组赋值给当前对象的数组属性；

（5）解锁并返回旧值；

size()方法

返回数组的长度。

```
public int size() {
    // 获取元素个数不需要加锁
    // 直接返回数组的长度
    return getArray().length;
}
```

总结

(1)CopyOnWriteArrayList使用ReentrantLock重入锁加锁，保证线程安全；任何对数组的修改都要先上锁修改完毕后解锁。

(2)CopyOnWriteArrayList的写操作都要先拷贝一份新数组，在新数组中做修改，修改完了再用新数组替换老数组，所以空间复杂度是O(n)，性能比较低下； “数组写修改操作都要拷贝一份新的数组，旧数组被GC回收”

(3)CopyOnWriteArrayList的读操作支持随机访问，时间复杂度为O(1)；

(4)CopyOnWriteArrayList采用读写分离的思想，读操作不加锁，写操作加锁，且写操作占用较大内存空间，所以适用于读多写少的场合；

(5)CopyOnWriteArrayList只保证最终一致性，不保证实时一致性；

彩蛋

为什么CopyOnWriteArrayList没有size属性?

因为每次修改都是拷贝一份正好可以存储目标个数元素的数组，所以不需要size属性了，数组的长度就是集合的大小，而不像ArrayList数组的长度实际是要大于集合的大小的。

比如，add(E e)操作，先拷贝一份n+1个元素的数组，再把新元素放到新数组的最后一位，这时新数组的长度为len+1了，也就是集合的size了。
