JUC集合: CopyOnWriteArrayList详解

> CopyOnWriteArrayList是ArrayList 的一个线程安全的变体，其中所有可变操作(add、set 等等)都是通过对底层数组进行一次新的拷贝来实现的。COW模式的体现。

面试问题去理解

- 请先说说非并发集合中Fail-fast机制?
- 再为什么说ArrayList查询快而增删慢?
- 对比ArrayList说说CopyOnWriteArrayList的增删改查实现原理? COW基于拷贝
- 再说下弱一致性的迭代器原理是怎么样的? `COWIterator<E>`
- CopyOnWriteArrayList为什么并发安全且性能比Vector好?
- CopyOnWriteArrayList有何缺陷，说说其应用场景?

CopyOnWriteArrayList源码分析

类的继承关系

CopyOnWriteArrayList实现了List接口，List接口定义了对列表的基本操作；同时实现了RandomAccess接口，表示可以随机访问(数组具有随机访问的特性)；同时实现了Cloneable接口，表示可克隆；同时也实现了Serializable接口，表示可被序列化。

```
public class CopyOnWriteArrayList<E> implements List<E>, RandomAccess, Cloneable,
java.io.Serializable {}
```

类的内部类

- COWIterator类

COWIterator表示迭代器，其也有一个Object类型的数组作为CopyOnWriteArrayList数组的快照，这种快照风格的迭代器方法在创建迭代器时使用了对当时数组状态的引用。此数组在迭代器的生存期内不会更改，因此不可能发生冲突，并且迭代器保证不会抛出 ConcurrentModificationException。创建迭代器以后，迭代器就不会反映列表的添加、移除或者更改。在迭代器上进行的元素更改操作(remove、set 和 add)不受支持。这些方法将抛出 UnsupportedOperationException。

```
static final class COWIterator<E> implements ListIterator<E> {
    /** Snapshot of the array */
    // 快照
    private final Object[] snapshot;
    /** Index of element to be returned by subsequent call to next.  */
    // 游标
```

```
    private int cursor;
    // 构造函数
    private COWIterator(Object[] elements, int initialCursor) {
        cursor = initialCursor;
        snapshot = elements;
    }
    // 是否还有下一项
    public boolean hasNext() {
        return cursor < snapshot.length;
    }
    // 是否有上一项
    public boolean hasPrevious() {
        return cursor > 0;
    }
    // next项
    @SuppressWarnings("unchecked")
    public E next() {
        if (! hasNext()) // 不存在下一项，抛出异常
            throw new NoSuchElementException();
        // 返回下一项
        return (E) snapshot[cursor++];
    }

    @SuppressWarnings("unchecked")
    public E previous() {
        if (! hasPrevious())
            throw new NoSuchElementException();
        return (E) snapshot[--cursor];
    }

    // 下一项索引
    public int nextIndex() {
        return cursor;
    }

    // 上一项索引
    public int previousIndex() {
        return cursor-1;
    }

    /**
        * Not supported. Always throws UnsupportedOperationException.
        * @throws UnsupportedOperationException always; {@code remove}
        *         is not supported by this iterator.
        */
    // 不支持remove操作
    public void remove() {
        throw new UnsupportedOperationException();
    }

    /**
        * Not supported. Always throws UnsupportedOperationException.
        * @throws UnsupportedOperationException always; {@code set}
        *         is not supported by this iterator.
        */
    // 不支持set操作
    public void set(E e) {
        throw new UnsupportedOperationException();
    }

    /**
```

```
         * Not supported. Always throws UnsupportedOperationException.
         * @throws UnsupportedOperationException always; {@code add}
         *         is not supported by this iterator.
         */
        // 不支持add操作
        public void add(E e) {
            throw new UnsupportedOperationException();
        }

        @Override
        public void forEachRemaining(Consumer<? super E> action) {
            Objects.requireNonNull(action);
            Object[] elements = snapshot;
            final int size = elements.length;
            for (int i = cursor; i < size; i++) {
                @SuppressWarnings("unchecked") E e = (E) elements[i];
                action.accept(e);
            }
            cursor = size;
        }
    }
```

类的属性

属性中有一个可重入锁，用来保证线程安全访问，还有一个Object类型的数组，用来存放具体的元素。当然，也使用到了反射机制和CAS来保证原子性的修改lock域。

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    // 版本序列号
    private static final long serialVersionUID = 8673264195747942595L;
    // 可重入锁
    final transient ReentrantLock lock = new ReentrantLock();
    // 对象数组，用于存放元素
    private transient volatile Object[] array;
    // 反射机制
    private static final sun.misc.Unsafe UNSAFE;
    // lock域的内存偏移量
    private static final long lockOffset;
    static {
        try {
            UNSAFE = sun.misc.Unsafe.getUnsafe();
            Class<?> k = CopyOnWriteArrayList.class;
            lockOffset = UNSAFE.objectFieldOffset
                (k.getDeclaredField("lock"));
        } catch (Exception e) {
            throw new Error(e);
        }
    }
}
```

类的构造函数

- 默认构造函数

```
public CopyOnWriteArrayList() {
    // 设置数组
    setArray(new Object[0]);
}
```

- CopyOnWriteArrayList(Collection<? extends E>)型构造函数　该构造函数用于创建一个按 collection 的迭代器返回元素的顺序包含指定 collection 元素的列表。

```
public CopyOnWriteArrayList(Collection<? extends E> c) {
    Object[] elements;
    if (c.getClass() == CopyOnWriteArrayList.class) // 类型相同
        // 获取c集合的数组
        elements = ((CopyOnWriteArrayList<?>)c).getArray();
    else { // 类型不相同
        // 将c集合转化为数组并赋值给elements
        elements = c.toArray();
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elements.getClass() != Object[].class) // elements类型不为Object[]类型
            // 将elements数组转化为Object[]类型的数组
            elements = Arrays.copyOf(elements, elements.length, Object[].class);
    }
    // 设置数组
    setArray(elements);
}
```

该构造函数的处理流程如下

- 判断传入的集合c的类型是否为CopyOnWriteArrayList类型，若是，则获取该集合类型的底层数组(Object[])，并且设置当前CopyOnWriteArrayList的数组(Object[]数组)，进入步骤③；否则，进入步骤②
- 将传入的集合转化为数组elements，判断elements的类型是否为Object[]类型(toArray方法可能不会返回Object类型的数组)，若不是，则将elements转化为Object类型的数组。进入步骤③
- 设置当前CopyOnWriteArrayList的Object[]为elements。
- CopyOnWriteArrayList(E[])型构造函数

该构造函数用于创建一个保存给定数组的副本的列表。

```
public CopyOnWriteArrayList(E[] toCopyIn) {
    // 将toCopyIn转化为Object[]类型数组，然后设置当前数组
    setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, Object[].class));
}
```

核心函数分析

对于CopyOnWriteArrayList的函数分析，主要明白Arrays.copyOf方法即可理解CopyOnWriteArrayList其他函数的意义。

copyOf函数

该函数用于复制指定的数组，截取或用 null 填充(如有必要)，以使副本具有指定的长度。

```
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {
    @SuppressWarnings("unchecked")
    // 确定copy的类型(将newType转化为Object类型，将Object[].class转化为Object类型，判断两者是否相等，若相等，则生成指定长度的Object数组
    // 否则,生成指定长度的新类型的数组)
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    // 将original数组从下标0开始，复制长度为(original.length和newLength的较小者),复制到copy数组中(也从下标0开始)
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

add函数

```
public boolean add(E e) {
    // 可重入锁
    final ReentrantLock lock = this.lock;
    // 获取锁
    lock.lock();
    try {
        // 元素数组
        Object[] elements = getArray();
        // 数组长度
        int len = elements.length;
        // 复制数组
        Object[] newElements = Arrays.copyOf(elements, len + 1);
        // 存放元素e
        newElements[len] = e;
        // 设置数组
        setArray(newElements);
        return true;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

此函数用于将指定元素添加到此列表的尾部，处理流程如下

- 获取锁(保证多线程的安全访问)，获取当前的Object数组，获取Object数组的长度为length，进入步骤②。
- 根据Object数组复制一个长度为length+1的Object数组为newElements(此时，newElements[length]为null)，进入下一步骤。
- 将下标为length的数组元素newElements[length]设置为元素e，再设置当前Object[]为newElements，释放锁，返回。这样就完成了元素的添加。

addIfAbsent方法

该函数用于添加元素(如果数组中不存在，则添加；否则，不添加，直接返回)，可以保证多线程环境下不会重复添加元素。

```
private boolean addIfAbsent(E e, Object[] snapshot) {
    // 重入锁
    final ReentrantLock lock = this.lock;
    // 获取锁
    lock.lock();
    try {
        // 获取数组
        Object[] current = getArray();
        // 数组长度
        int len = current.length;
        if (snapshot != current) { // 快照不等于当前数组，对数组进行了修改
            // Optimize for lost race to another addXXX operation
            // 取较小者
            int common = Math.min(snapshot.length, len);
            for (int i = 0; i < common; i++) // 遍历
                if (current[i] != snapshot[i] && eq(e, current[i])) // 当前数组的元素与快照的元素不相等并且e与当前元素相等
                    // 表示在snapshot与current之间修改了数组，并且设置了数组某一元素为e，已经存在
                    // 返回
                    return false;
            if (indexOf(e, current, common, len) >= 0) // 在当前数组中找到e元素
                    // 返回
                    return false;
        }
        // 复制数组
        Object[] newElements = Arrays.copyOf(current, len + 1);
        // 对数组len索引的元素赋值为e
        newElements[len] = e;
        // 设置数组
        setArray(newElements);
        return true;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

该函数的流程如下:

① 获取锁，获取当前数组为current，current长度为len，判断数组之前的快照snapshot是否等于当前数组current，若不相等，则进入步骤②；否则，进入步骤④

② 不相等，表示在snapshot与current之间，对数组进行了修改(如进行了add、 set、 remove等操作)，获取长度(snapshot与current之间的较小者)，对current进行遍历操作，若遍历过程发现snapshot与current的元素不相等并且current的元素与指定元素相等(可能进行了set操作)，进入步骤⑤，否则，进入步骤③

③ 在当前数组中索引指定元素，若能够找到，进入步骤⑤，否则，进入步骤④

④ 复制当前数组current为newElements，长度为len+1，此时newElements[len]为null。再设置newElements[len]为指定元素e，再设置数组，进入步骤⑤

⑤ 释放锁，返回。

set函数

此函数用于用指定的元素替代此列表指定位置上的元素，也是基于数组的复制来实现的。

```
public E set(int index, E element) {
    // 可重入锁
    final ReentrantLock lock = this.lock;
    // 获取锁
    lock.lock();
    try {
        // 获取数组
        Object[] elements = getArray();
        // 获取index索引的元素
        E oldValue = get(elements, index);

        if (oldValue != element) { // 旧值等于element
            // 数组长度
            int len = elements.length;
            // 复制数组
            Object[] newElements = Arrays.copyOf(elements, len);
            // 重新赋值index索引的值
            newElements[index] = element;
            // 设置数组
            setArray(newElements);
        } else {
            // Not quite a no-op; ensures volatile write semantics
            // 设置数组
            setArray(elements);
        }
        // 返回旧值
        return oldValue;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

remove函数

此函数用于移除此列表指定位置上的元素。

```
public E remove(int index) {
    // 可重入锁
    final ReentrantLock lock = this.lock;
    // 获取锁
    lock.lock();
    try {
        // 获取数组
        Object[] elements = getArray();
        // 数组长度
        int len = elements.length;
        // 获取旧值
        E oldValue = get(elements, index);
        // 需要移动的元素个数
        int numMoved = len - index - 1;
        if (numMoved == 0) // 移动个数为0
            // 复制后设置数组
            setArray(Arrays.copyOf(elements, len - 1));
        else { // 移动个数不为0
```

```
            // 新生数组
            Object[] newElements = new Object[len - 1];
            // 复制index索引之前的元素
            System.arraycopy(elements, 0, newElements, 0, index);
            // 复制index索引之后的元素
            System.arraycopy(elements, index + 1, newElements, index,
                             numMoved);
            // 设置索引
            setArray(newElements);
        }
        // 返回旧值
        return oldValue;
    } finally {
        // 释放锁
        lock.unlock();
    }
}
```

处理流程如下

① 获取锁，获取数组elements，数组长度为length，获取索引的值elements[index]，计算需要移动的元素个数(length - index - 1),若个数为0，则表示移除的是数组的最后一个元素，复制elements数组，复制长度为length-1，然后设置数组，进入步骤③；否则，进入步骤②

② 先复制index索引前的元素，再复制index索引后的元素，然后设置数组。

③ 释放锁，返回旧值。

CopyOnWriteArrayList示例

下面通过一个示例来了解CopyOnWriteArrayList的使用：在程序中，有一个PutThread线程会每隔50ms就向CopyOnWriteArrayList中添加一个元素，并且两次使用了迭代器，迭代器输出的内容都是生成迭代器时，CopyOnWriteArrayList的Object数组的快照的内容，在迭代的过程中，往CopyOnWriteArrayList中添加元素也不会抛出异常。

```
import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

class PutThread extends Thread {
    private CopyOnWriteArrayList<Integer> cowal;

    public PutThread(CopyOnWriteArrayList<Integer> cowal) {
        this.cowal = cowal;
    }

    public void run() {
        try {
            for (int i = 100; i < 110; i++) {
                cowal.add(i);
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
public class CopyOnWriteArrayListDemo {
    public static void main(String[] args) {
        CopyOnWriteArrayList<Integer> cowal = new CopyOnWriteArrayList<Integer>();
        for (int i = 0; i < 10; i++) {
            cowal.add(i);
        }
        PutThread p1 = new PutThread(cowal);
        p1.start();
        Iterator<Integer> iterator = cowal.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println();
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        iterator = cowal.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

运行结果(某一次)

```
0 1 2 3 4 5 6 7 8 9 100
0 1 2 3 4 5 6 7 8 9 100 101 102 103
```

更深入理解

*CopyOnWriteArrayList*的缺陷和使用场景

CopyOnWriteArrayList 有几个缺点：

- 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致young gc或者full gc
- 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个set操作后，读取到数据可能还是旧的,虽然CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求；

CopyOnWriteArrayList 合适读多写少的场景，不过这类慎用

因为谁也没法保证CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次add/set都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

*CopyOnWriteArrayList*为什么并发安全且性能比*Vector*好*?*

Vector对单独的add，remove等方法都是在方法上加了synchronized; 并且如果一个线程A调用size时，另一个线程B 执行了remove，然后size的值就不是最新的，然后线程A调用remove就会越界(这时就需要再加一个Synchronized)。这样就导致有了双重锁，效率大大降低，何必呢。于是vector废弃了，要用就用CopyOnWriteArrayList 吧。