java并发编程笔记—CopyOnWriteArrayList

|  |  |
| --- | --- |
|  | 🢂 内容概览 |
|  | Why：此文档用来做什么？它存在的意义是什么？为解决什么问题？   |  | | --- | |  |   What：当前包含了那些内容？   |  | | --- | |  |   How：此文档应如何参考？   |  | | --- | |  |   Who：此文档适用于那些人员阅读参考？   |  | | --- | |  |   Summary：摘要   |  | | --- | |  |   Reference：参考文献   |  | | --- | | [Java多线程系列--“JUC集合”02之 CopyOnWriteArrayList](http://www.cnblogs.com/skywang12345/p/3498483.html)  Clonable接口的使用  [sun.misc.Unsafe类的使用](http://blog.csdn.net/fenglibing/article/details/17138079)  [CopyOnWriteArrayList解读](http://caoyaojun1988-163-com.iteye.com/blog/1754686) | |

目录

[1 描述 3](#_Toc460873600)

[2 类图 3](#_Toc460873601)

[3 主要实现 5](#_Toc460873602)

[3.1 基本属性定义 5](#_Toc460873603)

[3.2 构造器定义 5](#_Toc460873604)

[3.3 查找元素 6](#_Toc460873605)

[3.4 新增元素 7](#_Toc460873606)

[3.5 更新元素 8](#_Toc460873607)

[3.6 删除元素 8](#_Toc460873608)

[3.7 迭代器实现 10](#_Toc460873609)

[4 适用场景 12](#_Toc460873610)

[4.1 小数据量，读多写少 12](#_Toc460873611)

[5 缺点&权衡点 12](#_Toc460873612)

[5.1 写操作耗时更多 12](#_Toc460873613)

[5.2 集合占用内存更多 12](#_Toc460873614)

[6 应用案例 13](#_Toc460873615)

[7 相关知识点 13](#_Toc460873616)

[7.1 迭代器模式 13](#_Toc460873617)

[7.2 CopyOnWriteArraySet 13](#_Toc460873618)

[8 问题思考 13](#_Toc460873619)

# 描述

1. CopyOnWriteArrayList是List的一种线程安全的实现；
2. 其实现原理采用”CopyOnWrite”的思路（不可变元素），即所有写操作，包括：add，remove，set等都会触发底层数组的拷贝，从而在写操作过程中，不会影响读操作；避免了使用synchronized等进行读写操作的线程同步；
3. CopyOnWrite对于写操作来说代价很大，故不适合于写操作很多的场景；当遍历操作远远多于写操作的时候，适合使用CopyOnWriteArrayList；
4. 迭代器以”快照”方式实现，在迭代器创建时，引用指向List当前状态的底层数组，所以在迭代器使用的整个生命周期中，其内部数据不会被改变；并且集合在遍历过程中进行修改，也不会抛出ConcurrentModificationException；迭代器在遍历过程中，不会感知集合的add，remove，set等操作；
5. 因为迭代器指向的是底层数组的”快照”，因此也不支持对迭代器本身的修改操作，包括add，remove，set等操作，如果使用这些操作，将会抛出UnsupportedOperationException；
6. 相关Happens-Before规则：一个线程将元素放入集合的操作happens-before于其它线程访问/删除该元素的操作；

# 类图

|  |
| --- |
|  |

|  |
| --- |
|  |

主要角色：

1. Iterable接口：定义创建迭代器操作，赋予实现类创建迭代器的能力；
2. Collection接口：定义集合基本操作，所有集合类都需实现该接口；
3. List接口：定义列表基本操作，所有列表类都需要实现该接口；
4. Serializable接口：标记接口，允许对象序列化；
5. Cloneable接口：标记接口，允许对象被克隆，如果类没有实现Cloneable接口，调用类对象的clone方法抛出CloneNotSupportedException；
6. RandomAccess接口：标记接口，用来表明其支持快速（通常是固定时间）随机访问（比如：ArrayList支持快速随机访问，所以使用普通遍历方式效率更高；而LinkedList则更适合顺序访问，所以使用迭代器的方式效率更高）；JDK中推荐的是对List集合尽量要实现RandomAccess接口；
7. COWIterator：CopyOnWriteArrayList对应的迭代器；不支持写操作，当调用写操作时，将抛出UnsupportedOperationException异常；
8. COWSubList：调用subList方法时，CopyOnWriteArrayList对应的子列表对象，和CopyOnWriteArrayList共享同一个array对象，使用使用offset和size维护对array的位置偏移；且当CopyOnWriteArrayList调用写操作更新array引用时，COWSubList对应的操作会抛出ConcurrentModificationException；
9. COWSubListIterator：COWSubList对应的迭代器，底层通过COWIterator实现，使用offset和size维护对array的位置偏移；不支持写操作，当调用写操作时，将抛出UnsupportedOperationException异常；

# 主要实现

## 基本属性定义

|  |
| --- |
| */\*\*  \* 互斥锁,用于保护所有修改操作,通过Unsafe进行初始化  \*/* **final transient** ReentrantLock **lock** = **new** ReentrantLock();  */\*\*  \* 存放数据的底层数据结构,只能通过getArray()和setArray()两个方法访问;  \*/* **private transient volatile** Object[] **array**;  */\*\*  \* <h2>array属性的getter和setter</h2>  \* <p>声明为非private类型,以便于CopyOnWriteArraySet访问</p>  \*/* **final** Object[] getArray() {  **return array**; }  **final void** setArray(Object[] array) {  **this**.**array** = array; } |

1. **lock属性：**互斥锁，用于控制CopyOnWriteArrayList所有的写操作的同步，以及对应的COWSubList的所有读写操作的同步；
2. **array属性：**Object[]数组，用于存放集合元素的底层数据结构；array只允许CopyOnWriteArrayList定义的getter和setter方法访问；（原因？）

## 构造器定义

|  |
| --- |
| */\*\*  \* <h2>初始化array为空数组</h2>  \*/* **public** CopyOnWriteArrayList() {  setArray(**new** Object[0]); }  */\*\*  \* <h2>将集合参数中的元素作为当前集合对象的元素</h2>  \*/* **public** CopyOnWriteArrayList(Collection<? **extends** E> c) {  Object[] elements;  **if** (c.getClass() == CopyOnWriteArrayList.**class**) {  elements = ((CopyOnWriteArrayList<?>) c).getArray();  } **else** {  elements = c.toArray();  *//注:c.toArray()可能不一定返回Object[]* **if** (elements.getClass() != Object[].**class**) {  elements = Arrays.*copyOf*(elements, elements.**length**, Object[].**class**);  }  } }  */\*\*  \* <h2>将数组参数中的元素作为当前集合对象的元素</h2>  \*/* **public** CopyOnWriteArrayList(E[] toCopyIn) {  *//注:传入数组对象并不是传递引用,而是新建一个数组拷贝原有数组* setArray(Arrays.*copyOf*(toCopyIn, toCopyIn.**length**, Object[].**class**)); } |

1. 定义了3个构造函数，第一个构造函数为无参构造函数，默认初始化array为空数组；后两个构造函数分别接收Collection对象和数组对象作为参数，使用参数的元素作为初始化array的元素；
2. 使用数组对象作为参数时，需要通过浅拷贝的方式初始化array，而非直接使用数组对象的引用；

## 查找元素

|  |
| --- |
| @Override **public int** size() {  **return** getArray().**length**; }  @Override **public boolean** isEmpty() {  **return** size() == 0; }  **private static boolean** eq(Object o1, Object o2) {  **return** (o1 == **null**) ? o2 == **null** : o1.equals(o2); }  **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; }  **private static int** lastIndexOf(Object o, Object[] elements, **int** index) {  **if** (o == **null**) {  **for** (**int** i = index; i >= 0; i--) {  **if** (elements[i] == **null**) {  **return** i;  }  }  } **else** {  **for** (**int** i = index; i >= 0; i--) {  **if** (o.equals(elements[i])) {  **return** i;  }  }  }  **return** -1; }  @Override **public boolean** contains(Object o) {  Object[] elements = getArray();  **return** *indexOf*(o, elements, 0, elements.**length**) >= 0; }  @Override **public int** indexOf(Object o) {  Object[] elements = getArray();  **return** *indexOf*(o, elements, 0, elements.**length**); }  **public int** indexOf(E e, **int** index) {  Object[] elements = getArray();  **return** *indexOf*(e, elements, index, elements.**length**); }  **public int** lastIndexOf(Object o) {  Object[] elements = getArray();  **return** *lastIndexOf*(o, elements, elements.**length** - 1); }  **public int** lastIndexOf(E e, **int** index) {  Object[] elements = getArray();  **return** *lastIndexOf*(e, elements, index); } |

1. CopyOnWriteArrayList的所有读操作都不需要加锁；
2. 单个元素的访问，直接通过索引访问底层数组；
3. 查找遍历等涉及多个元素的读操作，都会针对array的快照进行，即每次操作开始前使用名为elements的Object[]局部变量存放array当前的引用。在遍历过程中，array发生变化时，遍历操作遍历的依然是原来的快照，从而保证了读操作不需要加锁，写操作也不会发生ConcurrentModificationException异常；

## 新增元素

|  |
| --- |
| @Override **public boolean** add(E e) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  Object[] newElements = Arrays.*copyOf*(elements, len + 1);  newElements[len] = e;  setArray(newElements);  **return true**;  } **finally** {  lock.unlock();  } } |

1. 所有的新增元素操作都需要使用lock加互斥锁；
2. 新增元素需要考虑添加元素的个数（单个/集合），添加元素的位置（末尾/非末尾）；

## 更新元素

|  |
| --- |
| @Override **public** E set(**int** index, E element) {  *//加锁* **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  E oldValue = get(elements, index);   *//待更新元素不是原来的元素,才执行copyOnWrite* **if** (oldValue != element) {  **int** len = elements.**length**;  Object[] newElements = Arrays.*copyOf*(elements, len);  newElements[index] = element;  setArray(newElements);  } **else** {  *// 注:再次更新引用,为了触发volatile的语义,通知所有线程* setArray(elements);  }   **return** oldValue;  } **finally** {  *//解锁* lock.unlock();  } } |

1. 同添加元素操作一样，更新元素也需要使用lock加互斥锁；
2. 需要注意的是，即使待更新元素和集合中元素引用相同，也需要执行setArray()操作，以便触发volatile语义，通知所有线程；

## 删除元素

|  |
| --- |
| @Override **public** E remove(**int** index) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] elements = getArray();  **int** len = elements.**length**;  E oldValue = get(elements, index);  *//计算需要移动的元素个数* **int** numMoved = len - index - 1;  *//如果删除元素在数组末尾* **if** (numMoved == 0) {  setArray(Arrays.*copyOf*(elements, len - 1));  } **else** {  Object[] newElements = **new** Object[len - 1];  System.*arraycopy*(elements, 0, newElements, 0, index);  System.*arraycopy*(elements, index + 1, newElements, index,  numMoved);  setArray(newElements);  }  **return** oldValue;  } **finally** {  lock.unlock();  } }  **private boolean** remove(Object o, Object[] snapshot, **int** index) {  **final** ReentrantLock lock = **this**.**lock**;  lock.lock();  **try** {  Object[] current = getArray();  **int** len = current.**length**;  *//如果此时已经进行过写操作* **if** (snapshot != current) {   *//校准索引* FINDINDEX:  {  *//在当前代码获取锁时,有可能len已经小于index,故需要取二者中较小的;* **int** prefix = Math.*min*(index, len);   *//情形1:[0,index)区间的元素有删除操作,导致index所指元素前移* **for** (**int** i = 0; i < prefix; i++) {  *//如果元素引用已经替换* **if** (current[i] != snapshot[i] && *eq*(o, current[i])) {  index = i;  **break** FINDINDEX;  }  }   *//情形2:当前数组删除元素较多,导致数组长度小于原先index* **if** (index >= len) {  **return false**;  }   *//情形3:[0,index)区间的元素没有删除操作,index所指元素未移动* **if** (current[index] == o) {  **break** FINDINDEX;  }   *//情形4:[0,index)区间有添加元素操作,导致index所指元素后移,重新定位元素索引* index = *indexOf*(o, current, index, len);  }   }  Object[] newElements = **new** Object[len - 1];  System.*arraycopy*(current, 0, newElements, 0, index);  System.*arraycopy*(current, index + 1,  newElements, index,  len - index - 1);  setArray(newElements);   **return true**;  } **finally** {  lock.unlock();  } }  @Override **public boolean** remove(Object o) {  Object[] snapshot = getArray();  **int** index = *indexOf*(o, snapshot, 0, snapshot.**length**);  **return** index >= 0 && remove(o, snapshot, index); } |

1. 删除元素时，同样需要加互斥锁，但出于效率考虑，在加锁前都检测待删除元素是否存在，如果不存在则不加锁，直接返回false；
2. 由于判断元素是否存在的操作时未加锁，不能保证删除方法执行过程中，其它线程进行写操作。故在加锁后，依然需要对索引进行校准，增加了删除操作实现难度；

## 迭代器实现

|  |
| --- |
| @Override **public** Iterator<E> iterator() {  **return new** COWIterator<E>(getArray(), 0); }  @Override **public** ListIterator<E> listIterator() {  **return new** COWIterator<E>(getArray(), 0); }  @Override **public** ListIterator<E> listIterator(**int** index) {  Object[] elements = getArray();  **int** len = elements.**length**;  **if** (index < 0 || index > len) {  **throw new** IndexOutOfBoundsException(**"Index: "** + index);  }   **return new** COWIterator<E>(elements, index); }  */\*\*  \* <h2>Copy-On-Write迭代器定义</h2>  \* <p>限定为final,表示不可以被继承</p>  \*/* **static final class** COWIterator<E> **implements** ListIterator<E> {  */\*\*  \* 限定为final,指定为不可变对象,存放arry快照  \*/* **private final** Object[] **snapshot**;  */\*\*  \* 迭代器索引  \*/* **private int cursor**;   **private** COWIterator(Object[] elements, **int** initialCursor) {  **cursor** = initialCursor;  **snapshot** = elements;  }   **public boolean** hasNext() {  **return cursor** < **snapshot**.**length**;  }   **public boolean** hasPrevious() {  **return cursor** > 0;  }   @SuppressWarnings(**"unchecked"**)  **public** E next() {  **if** (!hasNext()) {  **throw new** NoSuchElementException();  }  **return** (E) **snapshot**[**cursor**++];  }   @SuppressWarnings(**"unchecked"**)  **public** E previous() {  **if** (!hasPrevious()) {  **throw new** NoSuchElementException();  }  **return** (E) **snapshot**[--**cursor**];  }   **public int** nextIndex() {  **return cursor**;  }   **public int** previousIndex() {  **return cursor** - 1;  }   */\*\*  \* Not supported. Always throws UnsupportedOperationException.  \*  \** ***@throws*** *UnsupportedOperationException always; {****@code*** *remove}  \* is not supported by this iterator.  \*/* **public void** remove() {  **throw new** UnsupportedOperationException();  }   */\*\*  \* Not supported. Always throws UnsupportedOperationException.  \*  \** ***@throws*** *UnsupportedOperationException always; {****@code*** *set}  \* is not supported by this iterator.  \*/* **public void** set(E e) {  **throw new** UnsupportedOperationException();  }   */\*\*  \* Not supported. Always throws UnsupportedOperationException.  \*  \** ***@throws*** *UnsupportedOperationException always; {****@code*** *add}  \* is not supported by this iterator.  \*/* **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;  } } |

1. COWIterator迭代器实现是基于快照的，即在调用iterator()方法创建迭代器时，传入array的引用作为快照； 通过一个整型游标记录当前访问到元素的索引；
2. 因为迭代器是基于快照的，故所有读操作都无须加锁，且迭代过程中，对CopyOnWriteArrayList集合的修改不会影响到迭代操作，也不会抛出ConcurrentModificationException；
3. 注意：因为COWSubList是作为CopyOnWriteArrayList的子列表的，所有操作都依赖于array，所以COWSubList的所有操作都必须检测array是否有改变，并且读写操作都要加锁，以保证写操作替换array的时候，没有读操作同步进行；

# 适用场景

## 小数据量，读多写少

数据量较小，读操作尤其是遍历操作远多于写操作时候，适合使用CopyOnWriteArrayList。

# 缺点&权衡点

## 写操作耗时更多

不可变对象的每次写操作就要进行一次copy/new操作，带来的性能消耗随着copy的数据量显著增加，包括内存的消耗以及copy/new过程的时间消耗；故不适合copy/new数据量很大，并且写操作很多的场景。

## 集合占用内存更多

使用Copy-On-Write，如果短时间有大量读伴随着写，则会有很多”快照”引用得不到释放，占用大量内存。

# 应用案例

# 相关知识点

## 迭代器模式

## CopyOnWriteArraySet

# 问题思考

1. **CopyOnWriteList如何保证在遍历的过程中修改集合不会触发ConcurrentModificationException?**
2. **为什么CopyOnWriteList的迭代器不支持写操作？**
3. **如下代码为什么不直接使用array，而要通过getArray方法？**

|  |
| --- |
|  |

1. 加锁的代码为什么要按照如下方式写？是要防止锁的引用改变吗？

|  |
| --- |
|  |