java并发编程笔记--CopyOnWriteArraySet

|  |  |
| --- | --- |
|  | 🢂 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”03之 CopyOnWriteArraySet](http://www.cnblogs.com/skywang12345/p/3498497.html) | |

目录

[1 描述 3](#_Toc461198995)

[2 类图 3](#_Toc461198996)

[3 主要实现 5](#_Toc461198997)

[3.1 基本属性定义 5](#_Toc461198998)

[3.2 构造器定义 5](#_Toc461198999)

[3.3 基本方法 5](#_Toc461199000)

[3.4 查找元素 6](#_Toc461199001)

[3.5 新增元素 7](#_Toc461199002)

[3.6 删除元素 7](#_Toc461199003)

[3.7 迭代器实现 7](#_Toc461199004)

[4 适用场景 8](#_Toc461199005)

[4.1 小数据量，读多写少 8](#_Toc461199006)

[5 缺点&权衡点 8](#_Toc461199007)

[5.1 写操作耗时更多 8](#_Toc461199008)

[5.2 集合占用内存更多 8](#_Toc461199009)

[6 应用案例 8](#_Toc461199010)

[7 相关知识点 8](#_Toc461199011)

[7.1 CopyOnWriteArrayList 8](#_Toc461199012)

# 描述

1. CopyOnWriteArraySet是Set的一种线程安全的实现，实现无序集合。
2. 和CopyOnWriteArrayList类似，也是通过CopyOnWrite，避免了读写互斥；使用空间换取读操作性能的提升；（空间换时间）
3. CopyOnWriteArraySet底层使用CopyOnWriteArrayList实现，因为Set中不能包含重复元素，所以CopyOnWriteArrayList专门提供了addIfAbsent()和addAllAbsent()这两个添加元素的API，用于在添加元素时做去重操作；

# 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. Set接口：定义Set的基本操作，所有Set都需要实现该接口；
4. Serializable接口：标记接口，允许对象序列化；
5. AbstractCollection：实现集合操作的骨架，其余继承的子类只需要实现iterator和size方法即可使用；
6. AbstractSet：作为Set集合实现的操作骨架，仅实现了equals，hashCode，removeAll方法；

# 主要实现

## 基本属性定义

|  |
| --- |
| */\*\*  \* 用于存放Set的底层数据,并提供copy-on-write操作  \*/* **private final** CopyOnWriteArrayList<E> **al**; |

1. **al属性：**通过关联方式使用 CopyOnWriteArrayList作为CopyOnWriteArraySet的底层结构，基本上所有方法都是直接调用al的相应方法实现；

## 构造器定义

|  |
| --- |
| **public** CopyOnWriteArrayLSet() {  **al** = **new** CopyOnWriteArrayList<>(); }  **public** CopyOnWriteArrayLSet(Collection<? **extends** E> c) {  *//如果集合类型与当前类型相同,则直接强制类型转换,并将元素添加到底层CopyOnWriteArrayList中  //注意:一定是重新拷贝,而不是指向引用* **if** (c.getClass() == CopyOnWriteArrayLSet.**class**) {  CopyOnWriteArrayLSet<E> cc = (CopyOnWriteArrayLSet<E>) c;  **al** = **new** CopyOnWriteArrayList<>(cc);  }  *//如果不相同,则去重添加元素* **else** {  **al** = **new** CopyOnWriteArrayList<>();  **al**.addAllAbsent(c);  } } |

1. 定义了2个构造函数，第一个构造函数为无参构造函数，默认初始化al；后1个构造函数接收Collection对象作为参数，使用参数的元素作为初始化al的属性；
2. 使用Collection对象作为参数时，需要通过浅拷贝的方式初始化al，而非直接使用对象的引用；

## 基本方法

需要注意的是，CopyOnWriteArraySet的equals方法效率为O(n^2)，所以只适合小数据量场景使用；

|  |
| --- |
| **private static boolean** eq(Object o1, Object o2) {  **return** (o1 == **null**) ? o2 == **null** : o1.equals(o2); }  @Override **public boolean** equals(Object o) {   *//如果引用相等,则相等* **if** (o == **this**) {  **return true**;  }   *//如果不是集合,则为假* **if** (!(o **instanceof** Set)) {  **return false**;  }   Set<?> set = (Set<?>) o;  Iterator<?> it = set.iterator();   *// 注意:代码的比较效率为O(n^2),所以只适合于小数据量的比较* Object[] elements = **al**.getArray();  **int** len = elements.**length**;   **boolean**[] matched = **new boolean**[len];  **int** k = 0;  outer:  **while** (it.hasNext()) {   *//如果长度不相等,则为假* **if** (++k > len) {  **return false**;  }   *//遍历集合中每个元素,进行比较* Object x = it.next();  **for** (**int** i = 0; i < len; ++i) {   *//比较每个元素,如果相同,则设置标记为true* **if** (!matched[i] && *eq*(x, elements[i])) {  matched[i] = **true**;  **continue** outer;  }  }  **return false**;  }   **return** k == len; } |

## 查找元素

|  |
| --- |
| @Override **public boolean** contains(Object o) {  **return al**.contains(o); }  @Override **public boolean** containsAll(Collection<?> c) {  **return al**.containsAll(c); }  @Override **public int** size() {  **return al**.size(); }  **public** Object[] toArray() {  **return al**.toArray(); }  **public** <T> T[] toArray(T[] a) {  **return al**.toArray(a); }  @Override **public void** forEach(Consumer<? **super** E> action) {  **al**.forEach(action); }  @Override **public boolean** isEmpty() {  **return al**.isEmpty(); } |

## 新增元素

|  |
| --- |
| @Override **public boolean** add(E e) {  **return al**.addIfAbsent(e); }  @Override **public boolean** addAll(Collection<? **extends** E> c) {  **return al**.addAllAbsent(c) > 0; } |

## 删除元素

|  |
| --- |
| @Override **public boolean** remove(Object o) {  **return al**.remove(o); }  @Override **public boolean** removeAll(Collection<?> c) {  **return al**.removeAll(c); }  @Override **public boolean** removeIf(Predicate<? **super** E> filter) {  **return al**.removeIf(filter); }  @Override **public void** clear() {  **al**.clear(); }  @Override **public boolean** retainAll(Collection<?> c) {  **return al**.retainAll(c); } |

## 迭代器实现

|  |
| --- |
| @Override **public** Iterator<E> iterator() {  **return al**.iterator(); } |

# 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CopyOnWriteArraySet。

# 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

## CopyOnWriteArrayList

1. 问题思考：就底层实现而言，CopyOnWriteArraySet相比HashSet的效率如何？