链滴

Java 之线程安全的容器

作者：YiHui

原文链接：https://ld246.com/article/1499265378926

来源网站：链滴

线程安全的容器

列表

线程安全的列表有 Vector，CopyOnWriteArrayList 两种，区别则主要在实现方式上，对锁的优化；后者主要采用的是 copy-on-write 思路，修改时，拷贝一份出来，修改完成之后替换

1. Vector 实现

vector 保证线程安全的原理比较简单粗暴，直接在方法上加锁

get 方法

```
public synchronized E get(int index) {
  if (index >= elementCount)
     throw new ArrayIndexOutOfBoundsException(index);

  return elementData(index);
}
```

set 方法

```
public synchronized E set(int index, E element) {
  if (index >= elementCount)
     throw new ArrayIndexOutOfBoundsException(index);

  E oldValue = elementData(index);
  elementData[index] = element;
  return oldValue;
}
```

add方法

```
 public synchronized boolean add(E e) {
       modCount++;
       ensureCapacityHelper(elementCount + 1);
       elementData[elementCount++] = e;
       return true;
   }
```

size方法

```
public synchronized int size() {
  return elementCount;
}
```

从上面几个最最常见的几个方法，就可以看出，这个实现非常的简单粗暴，全部上锁，肯定是线程安的问题了；相应的问题也很明显，效率妥妥的够了，即便全是读操作，都会有阻塞竞争，基本上完全没法忍的

2. CopyOnWriteArrayList 实现

使用了 copyOnWrite 机制，一句话，读时直接读，在修改时，先拷贝一份出来，在拷贝上进行修改完成之后替换掉之前的引用

下面主要看一下几个最常见的方法，是如何实现的，以此来研究下这套机制的玩法

size 方法

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

/** The array, accessed only via getArray/setArray. */
private transient volatile Object[] array;

/**
* Gets the array.  Non-private so as to also be accessible
* from CopyOnWriteArraySet class.
*/
final Object[] getArray() {
   return array;
}
```

对比一下 ArrayList 的获取size方法，有一个size属性记录的是链表的长度，直接返回的这个值；而CoyOnWriteArrayList 则是每次都去实时查数组的长度

```
/**
* The size of the ArrayList (the number of elements it contains).
*
* @serial
*/
private int size;

public int size() {
       return size;
}
```

为什么这么做？

get方法

```
/**
* {@inheritDoc}
*
* @throws IndexOutOfBoundsException {@inheritDoc}
*/
public E get(int index) {
   return get(getArray(), index);
}

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

```
}

// ArrayList
public E get(int index) {
   rangeCheck(index);

   return elementData(index);
}

E elementData(int index) {
   return (E) elementData[index];
}
```

和上面相同，同样是先调用 getArray() 方法，然后在进行相应的操作，如果不这么做，直接如 ArrayList 一样的调用方式时(如下)

- 假设数组长度为 3，现在获取index=2（即最后一个）的元素值
- rangeCheck 方法确认通过，elementData执行之前
- 现在一个线程，删除了一个元素，此时上面这个线程获取时，就会出现数组越界

如果是上面的执行逻辑，则不会如此，因为操作的依然是老的那个数组对应的引用；当发生修改时，在新的数组上进行的

add 方法

接下来则看一下具体的修改方法，是不是确实在新的数组上进行的操作，源码如下：

```
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();
   }
}
```

关注几点：

- 上锁： 这里表明，每次修改都只能有一个线程在执行
- 修改过程：
  - 拷贝原来内容到新的的数组上
  - 将元素添加在新的数组上
  - 更新列表中数组的引用，指向新的数组

set 方法

修改内容的值时，同样是先加锁，再修改，确保每次修改都是串行进行的；需要注意的一点是

- 若设置的value和原来的内容相等，则不需要修改引用
- 一定得确保释放锁

```
public E set(int index, E element) {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        Object[] elements = getArray();
        E oldValue = get(elements, index);

        if (oldValue != element) {
            int len = elements.length;
            Object[] newElements = Arrays.copyOf(elements, len);
            newElements[index] = element;
            setArray(newElements);
        } else {
            // Not quite a no-op; ensures volatile write semantics
            setArray(elements);
        }
        return oldValue;
    } finally {
        lock.unlock();
    }
}
```

小结

1. Vector：无论读写，全部加上了同步锁，导致多线程访问or修改时，锁的竞争，效率较低

2. CopyOnWriteArrayList：读不加锁，在修改时，加锁保证每次只有一个线程在修改列表；且修改逻辑都是先拷贝一个副本出来，然后在副本上进行修改，最后在替换列表中数组的引用

3. CopyOnWriteArraySet：内部数组其实就是一个 CopyOnWriteArrayList, 相关方法也是直接来自opyOnWriteArrayList

Map

线程安全的Map则主要是 HashTable ConcurrentHashMap， 后者采用了分段锁机制来提高并发访的性能

在便利时，操作Map的几种场景分析

1. 在遍历时，修改Map的引用（即用一个新的map替换这个值）
    - 仍旧遍历老的Map
2. 在遍历时，修改Map中的元素值
    - 会读取到修改后的值

3. 在遍历时，新增or删除元素

- HashMap 会抛异常； ConcurrentHashMap 可正常执行

1. HashTable

同 Vector 一样，通过对所有的方法添加 synchronized 关键字来确保的线程安全；缺点也很明显，率低，具体的几个方法源码如下，不多加说明了

```
public synchronized int size() {
   return count;
}

public synchronized Enumeration<K> keys() {
   return this.<K>getEnumeration(KEYS);
}

 public synchronized Enumeration<V> elements() {
   return this.<V>getEnumeration(VALUES);
}

public synchronized V get(Object key) {
   Entry<?,?> tab[] = table;
   int hash = key.hashCode();
   int index = (hash & 0x7FFFFFFF) % tab.length;
   for (Entry<?,?> e = tab[index] ; e != null ; e = e.next) {
       if ((e.hash == hash) && e.key.equals(key)) {
           return (V)e.value;
       }
   }
   return null;
}


public synchronized V put(K key, V value) {
   // Make sure the value is not null
   if (value == null) {
       throw new NullPointerException();
   }

   // Makes sure the key is not already in the hashtable.
   Entry<?,?> tab[] = table;
   int hash = key.hashCode();
   int index = (hash & 0x7FFFFFFF) % tab.length;
   @SuppressWarnings("unchecked")
   Entry<K,V> entry = (Entry<K,V>)tab[index];
   for(; entry != null ; entry = entry.next) {
       if ((entry.hash == hash) && entry.key.equals(key)) {
           V old = entry.value;
           entry.value = value;
           return old;
       }
   }
```

```
    addEntry(hash, key, value, index);
    return null;
}
```

2. ConcurrentHashMap

一个ConcurrentHashMap由多个segment组成，每一个segment都包含了一个HashEntry数组的hastable， 每一个segment包含了对自己的hashtable的操作，比如get，put，replace等操作，这些操发生的时候，对自己的hashtable进行锁定。由于每一个segment写操作只锁定自己的hashtable，以可能存在多个线程同时写的情况，性能无疑好于只有一个hashtable锁定的情况

简单来讲，就是每个 segment 的操作都是加锁的；而多个 segment 的操作可以是并发的

详解可以参考: Java集合---ConcurrentHashMap原理分析