第六章 并发集合 Java 7并发编程实战手册读书笔记

第六章 并发集合

a.Java提供两种在并发应用程序中使用的集合：

阻塞集合：这种集合包括添加和删除数据的操作。如果操作不能立即进行，是因为集合已满或者为空，该程序将被阻塞，直到操作可以进行。

非阻塞集合：这种集合也包括添加和删除数据的操作。如果操作不能立即进行，这个操作将返回null值或抛出异常，但该线程将不会阻塞。

b.一些常用的并发类集合包括：

- 非阻塞列表，使用ConcurrentLinkedDeque类
- 阻塞列表，使用LinkedBlockingDeque类
- 用在生产者与消费者数据的阻塞列表，使用LinkedTransferQueue类
- 使用优先级排序元素的阻塞列表，使用PriorityBlockingQueue类
- 存储迟迟元素的阻塞列表，使用DelayQueue类。
- 非阻塞可导航的map，使用ConcurrentSkipListMap类
- 随机数，使用ThreadLocalRandom类
- 原子变量，使用AtomicLong和AtomicIntegerArray类

1、使用非阻塞线程安全的列表 ConcurrentLinkedDeque

并发列表允许不同的线程在同一时刻添加或移除列表中的元素，而不会造成数据不一致。

a.非阻塞列表提供这些操作：如果操作不能立即完成(比如，你想要获取列表的元素而列表却是空的)，它将根据这个操作抛出异常或返回null值。

b.Java 7引进实现了非阻塞并发列表的ConcurrentLinkedDeque类。 ConcurrentLinkedDeque类提供更多方法来获取列表的元素：

1）getFirst()和getLast()：这些方法将分别返回列表的第一个和最后一个元素。它们不会从列表删除返回的元素。如果列表为空，这些方法将抛出NoSuchElementExcpetion异常。

2）peek()、peekFirst()和peekLast()：这些方法将分别返回列表的第一个和最后一个元素。它们不会从列表删除返回的元素。如果列表为空，这些方法将返回null值。

3）poll()、pollFirst()和pollLast()：这些方法将分别返回列表的第一个和最后一个元素。它们不会从列表删除返回的元素。如果列表为空，这些方法将返回null值。

4）remove()、removeFirst()、 removeLast()：这些方法将分别返回列表的第一个和最后一个元素。它们将从列表删除返回的元素。如果列表为空，这些方法将抛出NoSuchElementExcpetion异常。

```
public class AddTask implements Runnable {

 private ConcurrentLinkedDeque<String> list;
 public AddTask(ConcurrentLinkedDeque<String> list) {
  this.list=list;
 }

 @Override
 public void run() {
  String name=Thread.currentThread().getName();
  for (int i=0; i<10000; i++){
   list.add(name+": Element "+i);
  }
 }
}
public class PollTask implements Runnable {

 private ConcurrentLinkedDeque<String> list;
 public PollTask(ConcurrentLinkedDeque<String> list) {
  this.list=list;
 }

 @Override
 public void run() {
  for (int i=0; i<5000; i++) {
   list.pollFirst();
   list.pollLast();
  }
 }
```

```
}
public class Main {

 public static void main(String[] args) throws Exception {

  ConcurrentLinkedDeque<String> list=new ConcurrentLinkedDeque<>();
  for (int i=0; i<threads.length; i++){
   AddTask task=new AddTask(list);
   threads[i]=new Thread(task);
   threads[i].start();
  }
  System.out.printf("Main: %d AddTask threads have been launched\n",threads.length);
  for (int i=0; i<threads.length; i++) {
    threads[i].join();
  }
  System.out.printf("Main: Size of the List: %d\n",list.size());

  for (int i=0; i<threads.length; i++){
   PollTask task=new PollTask(list);
   threads[i]=new Thread(task);
   threads[i].start();
  }
  System.out.printf("Main: %d PollTask threads have been launched\n",threads.length);

  for (int i=0; i<threads.length; i++) {
   threads[i].join();
  }
  System.out.printf("Main: Size of the List: %d\n",list.size());
 }
}
```

结果

```
<terminated> Main (20) [Java Application] D:\Java\jre\bin\
Main: 100 AddTask threads have been launched
Main: Size of the List: 1000000
Main: 100 PollTask threads have been launched
Main: Size of the List: 0
```

2、使用阻塞线程安全的列表 LinkedBlockingDeque

a.阻塞列表与非阻塞列表的主要区别是，阻塞列表有添加和删除元素的方法，如果由于列表已满或为空而导致这些操作不能立即进行，它们将阻塞调用的线程，直到这些操作可以进行。

b.Java包含实现阻塞列表的LinkedBlockingDeque类。 LinkedBlockingDeque类同时提供方法用于添加和获取列表的元素，而不被阻塞，或抛出异常，或返回null值。这些方法是：

1）put()方法添加字符串到列表中。如果列表已满（因为你已使用固定大小来创建它），这个方法阻塞线程的执行，直到列表有可用空间。
2）take()方法从列表中获取字符串,如果列表为空，这个方法将阻塞线程的执行，直到列表中有元素。
3）takeFirst() 和takeLast()：这些方法分别返回列表的第一个和最后一个元素。它们从列表删除返回的元素。如果列表为空，这些方法将阻塞线程，直到列表有元素。

```
public class Client implements Runnable{

 private LinkedBlockingDeque<String> requestList;
 public Client (LinkedBlockingDeque<String> requestList) {
  this.requestList=requestList;
 }

 @Override
 public void run() {
  for (int i=0; i<3; i++) {
   for (int j=0; j<5; j++) {
    StringBuilder request=new StringBuilder();
    request.append(i);
    request.append(":");
    request.append(j);
```

```
    try {
     requestList.put(request.toString());
    } catch (InterruptedException e) {
     e.printStackTrace();
    }
    System.out.printf("Client: %s at %s.\n",request,new Date());
   }
   try {
    TimeUnit.SECONDS.sleep(2);
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
  System.out.printf("Client: End.\n");
 }
}

public class Main {

 public static void main(String[] args) throws Exception {

  LinkedBlockingDeque<String> list=new LinkedBlockingDeque<>(3);
  Client client=new Client(list);
  Thread thread=new Thread(client);
  thread.start();

  for (int i=0; i<5 ; i++) {
   for (int j=0; j<3; j++) {
    String request=list.take();
    System.out.printf("Main: Request: %s at %s. Size: %d\n",request,new Date(),list.size());
   }
   TimeUnit.MILLISECONDS.sleep(300);
  }
  System.out.printf("Main: End of the program.\n");
 }
}
```

结果

```
<terminated> Main (21) [Java Application] D:\Java\jre\bin\javaw.exe (2016年
Main: Request: 0:0 at Sun Nov 20 16:21:32 CST 2016. Size: 0
Client: 0:0 at Sun Nov 20 16:21:32 CST 2016.
Main: Request: 0:1 at Sun Nov 20 16:21:32 CST 2016. Size: 0
Client: 0:1 at Sun Nov 20 16:21:32 CST 2016.
Main: Request: 0:2 at Sun Nov 20 16:21:32 CST 2016. Size: 0
Client: 0:2 at Sun Nov 20 16:21:32 CST 2016.
Client: 0:3 at Sun Nov 20 16:21:32 CST 2016.
Client: 0:4 at Sun Nov 20 16:21:32 CST 2016.
Main: Request: 0:3 at Sun Nov 20 16:21:33 CST 2016. Size: 1
Main: Request: 0:4 at Sun Nov 20 16:21:33 CST 2016. Size: 0
Main: Request: 1:0 at Sun Nov 20 16:21:34 CST 2016. Size: 0
Client: 1:0 at Sun Nov 20 16:21:34 CST 2016.
Client: 1:1 at Sun Nov 20 16:21:34 CST 2016.
Client: 1:2 at Sun Nov 20 16:21:34 CST 2016.
Client: 1:3 at Sun Nov 20 16:21:34 CST 2016.
Main: Request: 1:1 at Sun Nov 20 16:21:35 CST 2016. Size: 2
Client: 1:4 at Sun Nov 20 16:21:35 CST 2016.
Main: Request: 1:2 at Sun Nov 20 16:21:35 CST 2016. Size: 2
Main: Request: 1:3 at Sun Nov 20 16:21:35 CST 2016. Size: 1
Main: Request: 1:4 at Sun Nov 20 16:21:35 CST 2016. Size: 0
Main: Request: 2:0 at Sun Nov 20 16:21:37 CST 2016. Size: 0
Client: 2:0 at Sun Nov 20 16:21:37 CST 2016.
Main: Request: 2:1 at Sun Nov 20 16:21:37 CST 2016. Size: 0
Client: 2:1 at Sun Nov 20 16:21:37 CST 2016.
Client: 2:2 at Sun Nov 20 16:21:37 CST 2016.
Client: 2:3 at Sun Nov 20 16:21:37 CST 2016.
Client: 2:4 at Sun Nov 20 16:21:37 CST 2016.
Main: Request: 2:2 at Sun Nov 20 16:21:37 CST 2016. Size: 2
Main: Request: 2:3 at Sun Nov 20 16:21:37 CST 2016. Size: 1
Main: Request: 2:4 at Sun Nov 20 16:21:37 CST 2016. Size: 0
Main: End of the program.
Client: End.
```

3、使用优先级排序的阻塞线程安全的列表 PriorityBlockingQueue

一个典型的需求是，当你需要使用一个有序列表的数据结构时，Java提供的PriorityBlockingQueue类就拥有这种功能。
对下面Demo的说明：
a.想要添加到PriorityBlockingQueue中的所有元素必须实现Comparable接口。所以Event类实现了这个接口的compareTo()方法，这个方法决定插入元素的位置。（校注：默认情况下）较大的元素将被放在队列的尾部。
b.阻塞数据结构（blocking data structure）是PriorityBlockingQueue的另一个重要特性。它有这样的方法，如果它们不能立即进行它们的操作，则阻塞这个线程直到它们的操作可以进行。
c.所有事件（此demo中为Event类）都有一个优先级属性。拥有更高优先级的元素将成为队列的第一个元素。当你已实现compareTo()方法,如果执行这个方法的事件拥有比作为参数传入的事件更高的优先级时,它将返回-1。在其他情况下，如果执行这个方法的事件拥有比作为参数传入的事件更低的优先级时，它将返回1。如果这两个对象拥有相同优先级，compareTo()方法将返回0。在这种情况下，PriorityBlockingQueue类并不能保证元素的顺序。

PriorityBlockingQueue类提供其他有趣的方法，以下是其中一些方法的描述：
1）clear()：这个方法删除队列中的所有元素。
2）take()：这个方法返回并删除队列中的第一个元素。如果队列是空的，这个方法将阻塞线程直到队列有元素。
3）put(E e)：E是用来参数化PriorityBlockingQueue类的类。这个方法将作为参数传入的元素插入到队列中。
4）peek()：这个方法返回列队的第一个元素，但不删除它。
5）poll()：检索并删除list第一元素。

```
public class Event implements Comparable<Event> {
 private int thread;
 private int priority;
 public Event(int thread, int priority){
  this.thread=thread;
  this.priority=priority;
 }
 public int getThread() {
  return thread;
 }
 public int getPriority() {
  return priority;
 }
 @Override
 public int compareTo(Event e) {
  if (this.priority>e.getPriority()) {
   return -1;
  } else if (this.priority<e.getPriority()) {
   return 1;
  } else {
   return 0;
  }
 }
}
public class Task implements Runnable {

 private int id;
 private PriorityBlockingQueue<Event> queue;
 public Task(int id, PriorityBlockingQueue<Event> queue) {
  this.id=id;
  this.queue=queue;
 }
 @Override
 public void run() {
  for (int i=0; i<1000; i++){
   Event event=new Event(id,i);
   queue.add(event);
  }
 }
}
public class Main {
 public static void main(String[] args) {
  PriorityBlockingQueue<Event> queue=new PriorityBlockingQueue<>();
  Thread taskThreads[]=new Thread[5];
  for (int i=0; i<taskThreads.length; i++){
   Task task=new Task(i,queue);
```

```
  taskThreads[i]=new Thread(task);
 }
 for (int i=0; i<taskThreads.length; i++) {
  taskThreads[i].start();
 }
 for (int i=0; i<taskThreads.length; i++) {
  try {
   taskThreads[i].join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
 System.out.printf("Main: Queue Size: %d\n",queue.size());
 for (int i=0; i<taskThreads.length*1000; i++){
  Event event=queue.poll();
  System.out.printf("Thread %s: Priority %d\n",event.getThread(),event.getPriority());
 }
 System.out.printf("Main: Queue Size: %d\n",queue.size());
 System.out.printf("Main: End of the program\n");
}
}
```

结果

```
<terminated> Main (22) [Java Application] D:\Java\jre\bi
Thread 3: Priority 6
Thread 4: Priority 5
Thread 3: Priority 5
Thread 2: Priority 5
Thread 1: Priority 5
Thread 0: Priority 5
Thread 3: Priority 4
Thread 4: Priority 4
Thread 0: Priority 4
Thread 2: Priority 4
Thread 1: Priority 4
Thread 3: Priority 3
Thread 1: Priority 3
Thread 2: Priority 3
Thread 4: Priority 3
Thread 0: Priority 3
Thread 0: Priority 2
Thread 3: Priority 2
Thread 2: Priority 2
Thread 1: Priority 2
Thread 4: Priority 2
Thread 0: Priority 1
Thread 1: Priority 1
Thread 4: Priority 1
Thread 2: Priority 1
Thread 3: Priority 1
Thread 2: Priority 0
Thread 1: Priority 0
Thread 3: Priority 0
Thread 4: Priority 0
Thread 0: Priority 0
Main: Queue Size: 0
Main: End of the program
```

4、使用线程安全的、带有延迟元素的列表 DelayedQueue（没看懂）

a.DelayedQueue类是Java API提供的一种有趣的数据结构，在这个类中，可以存储带有激活日期的元素。方法返回或抽取队列的元素将忽略未到期的数据元素。它们对这些方法来说是看不见的。

b.为了获取这种行为，你想要存储到DelayedQueue类中的元素必须实现Delayed接口。这个接口允许处理延迟对象，所以将实现存储在DelayedQueue对象的激活日期，这个激活时期将作为对象的剩余时间，直到激活日期到来。这个接口强制实现以下两种方法：
1）compareTo(Delayed o)：Delayed接口继承Comparable接口。如果执行这个方法的对象的延期小于作为参数传入的对象时，该方法返回一个小于0的值。如果执行这个方法的对象的延期大于作为参数传入的对象时，该方法返回一个大于0的值。如果这两个对象有相同的延期，该方法返回0。
2）getDelay(TimeUnit unit)：该方法返回与此对象相关的剩余延迟时间，以给定的时间单位表示。TimeUnit类是一个枚举类，有以下常量：DAYS、HOURS、 MICROSECONDS、MILLISECONDS、 MINUTES、 NANOSECONDS 和 SECONDS。

c.DelayQueue类提供其他方法，如下：
1）clear()：这个方法删除队列中的所有元素。
2）offer(E e)：E是代表用来参数化DelayQueue类的类。这个方法插入作为参数传入的元素到队列中。

3）peek()：这个方法检索，但不删除队列的第一个元素。
4）take()：这具方法检索并删除队列的第一个元素。如果队列中没有任何激活的元素，执行这个方法的线程将被阻塞，直到队列有一些激活的元素。

5、使用线性安全可遍历映射ConcurrentSkipListMap

a.Java API 提供的有趣的数据结构,并且你可以在并发应用程序中使用，即ConcurrentNavigableMap接口。实现ConcurrentNavigableMap接口的类存储以下两部分元素：
1)唯一标识元素的key
2)定义元素的剩余数据

b.Java API 也提供了这个接口的实现类，这个类是ConcurrentSkipListMap，它实现了非阻塞列表且拥有ConcurrentNavigableMap的行为。在内部实现中，它使用Skip List来存储数据。Skip List是基于并行列表的数据结构，它允许我们获取类似二叉树的效率。使用它，你可以得到一个排序的数据结构，这比排序数列使用更短的访问时间来插入、搜索和删除元素.

c.ConcurrentSkipListMap类有其他有趣的方法，这些方法如下：
1）headMap(K toKey)：K是参数化ConcurrentSkipListMap对象的Key值的类。返回此映射的部分视图，其键值小于 toKey。
2）tailMap(K fromKey)：K是参数化ConcurrentSkipListMap对象的Key值的类。返回此映射的部分视图，其键大于等于 fromKey。
3）putIfAbsent(K key, V Value)：如果key不存在map中，则这个方法插入指定的key和value。
pollLastEntry()：这个方法返回并删除map中最后一个元素的Map.Entry对象。
4）replace(K key, V Value)：如果这个key存在map中，则这个方法将指定key的value替换成新的value。

```
public class Contact {

 private String name;
 private String phone;
 public Contact(String name, String phone) {
  this.name=name;
  this.phone=phone;
 }
 public String getName() {
  return name;
       }
 public String getPhone() {
  return phone;
 }
}
public class Task implements Runnable {

 private ConcurrentSkipListMap<String, Contact> map;
 private String id;
 public Task (ConcurrentSkipListMap<String, Contact> map, String id) {
  this.id=id;
  this.map=map;
 }

 @Override
 public void run() {
  for (int i=0; i<1000; i++) {
   Contact contact=new Contact(id, String.valueOf(i+1000));
   map.put(id+contact.getPhone(), contact);
  }
 }
}
public class Main {

 public static void main(String[] args) {

  ConcurrentSkipListMap<String, Contact> map;
  map=new ConcurrentSkipListMap<>();
  Thread threads[]=new Thread[25];
  int counter=0;
  for (char i='A'; i<'Z'; i++) {
   Task task=new Task(map, String.valueOf(i));
   threads[counter]=new Thread(task);
```

```
  threads[counter].start();
  counter++;
 }
 for (int i=0; i<threads.length; i++) {
  try {
   threads[i].join();
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
 System.out.printf("Main: Size of the map: %d\n",map.size());

 Map.Entry<String, Contact> element;
 Contact contact;

 element=map.firstEntry();
 contact=element.getValue();
 System.out.printf("Main: First Entry: %s: %s\n",contact.getName(),contact.getPhone());

 element=map.lastEntry();
 contact=element.getValue();
 System.out.printf("Main: Last Entry: %s: %s\n",contact.getName(),contact.getPhone());

 System.out.printf("Main: Submap from A1996 to B1002: \n");
 ConcurrentNavigableMap<String, Contact> submap=map.subMap("A1996", "B1002");
 do {
  element=submap.pollFirstEntry();
  if (element!=null) {
   contact=element.getValue();
   System.out.printf("%s: %s\n",contact.getName(),contact.getPhone());
  }
 } while (element!=null);
}
}
```

结果

```
<terminated> Main (24) [Java Application] D:\Java\jre\l
Main: Size of the map: 25000
Main: First Entry: A: 1000
Main: Last Entry: Y: 1999
Main: Submap from A1996 to B1002:
A: 1996
A: 1997
A: 1998
A: 1999
B: 1000
B: 1001
```

6、生成并发随机数 ThreadLocalRandom

a.ThreadLocalRandom的current()方法是一个静态方法，它返回当前线程的ThreadLocalRandom对象，可以使用这个对象生成随机数.
b.ThreadLocalRandom类同样提供方法来生成long、float 和 double类型的数以及 Boolean值。这些方法允许你传入一个数值作为参数，然后生成0到这个数值之间的随机数。还有允许你传入两个参数的其他方法，然后生成在这两个参数数值之间的随机数。

```
public class TaskLocalRandom implements Runnable {

 public TaskLocalRandom() {
  ThreadLocalRandom.current();
 }
 @Override
 public void run() {
  String name=Thread.currentThread().getName();
  for (int i=0; i<10; i++){
   System.out.printf("%s: %d\n",name,ThreadLocalRandom.current().nextInt(10));
  }
 }
}
public class Main {
```

```
 public static void main(String[] args) {

  Thread threads[]=new Thread[3];
  for (int i=0; i<threads.length; i++) {
   TaskLocalRandom task=new TaskLocalRandom();
   threads[i]=new Thread(task);
   threads[i].start();
  }
 }
}
```

结果

```
<terminated> Main (25) [Java Application] D:\Java\jre\b
Thread-0: 8
Thread-0: 3
Thread-0: 3
Thread-0: 4
Thread-0: 8
Thread-0: 5
Thread-0: 6
Thread-0: 6
Thread-0: 8
Thread-0: 9
Thread-2: 1
Thread-2: 3
Thread-2: 1
Thread-2: 7
Thread-2: 4
Thread-2: 6
Thread-2: 2
Thread-2: 8
Thread-2: 5
Thread-2: 8
Thread-1: 8
Thread-1: 8
Thread-1: 4
Thread-1: 5
Thread-1: 9
Thread-1: 5
Thread-1: 6
Thread-1: 4
Thread-1: 8
Thread-1: 0
```

7、使用原子变量

a.在Java 1.5中就引入了原子变量，它提供对单个变量的原子操作。当你在操作一个普通变量时，你在Java实现的每个操作，在程序编译时会被转换成几个机器能读懂的指令。例如，当你分配一个值给变量，在Java你只使用了一个指令，但是当你编译这个程序时，这个指令就被转换成多个JVM 语言指令。这样子的话当你在操作多个线程且共享一个变量时，就会导致数据不一致的错误。

b.为了避免这样的问题，Java引入了原子变量。当一个线程正在操作一个原子变量时，即使其他线程也想要操作这个变量，类的实现中含有一个检查那步骤操作是否完成的机制。 基本上，操作获取变量的值，改变本地变量值，然后尝试以新值代替旧值。如果旧值还是一样，那么就改变它。如果不一样，方法再次开始操作。这个操作称为 Compare and Set（校对注：简称CAS，比较并交换的意思）。

c.原子变量不使用任何锁或者其他同步机制来保护它们的值的访问。他们的全部操作都是基于CAS操作。它保证几个线程可以同时操作一个原子对象也不会出现数据不一致的错误，并且它的性能比使用受同步机制保护的正常变量要好。

d.Java还有其他的原子类。例如：AtomicBoolean, AtomicInteger, 和 AtomicReference。

```
public class Account {
 private AtomicLong balance;
 public Account(){
  balance=new AtomicLong();
 }
 public long getBalance() {
  return balance.get();
 }
 public void setBalance(long balance) {
  this.balance.set(balance);
 }
 public void addAmount(long amount) {
  this.balance.getAndAdd(amount);
 }
 public void subtractAmount(long amount) {
```

```
  this.balance.getAndAdd(-amount);
 }
}

public class Bank implements Runnable {
 private Account account;
 public Bank(Account account) {
  this.account=account;
 }
 @Override
 public void run() {
  for (int i=0; i<10; i++){
   account.subtractAmount(1000);
  }
 }
}

public class Company implements Runnable {

 private Account account;
 public Company(Account account) {
  this.account=account;
 }
 @Override
 public void run() {
  for (int i=0; i<10; i++){
   account.addAmount(1000);
  }
 }
}
public class Main {

 public static void main(String[] args) {

  Account account=new Account();
  account.setBalance(1000);

  Company company=new Company(account);
  Thread companyThread=new Thread(company);
  Bank bank=new Bank(account);
  Thread bankThread=new Thread(bank);

  System.out.printf("Account : Initial Balance: %d\n",account.getBalance());

  companyThread.start();
  bankThread.start();

  try {
   companyThread.join();
   bankThread.join();
   System.out.printf("Account : Final Balance: %d\n",account.getBalance());
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
}
```

结果

```
<terminated> Main (26) [Java Application] D:\Java\j
Account : Initial Balance: 1000
Account : Final Balance: 1000
```

8、使用原子数组

a.CAS(compare-and-swap)操作为并发操作对象的提供更好的性能，CAS操作通过以下3个步骤来实现对变量值得修改：

1）获取当前内存中的变量的值

2）用一个新的临时变量(temporal variable)保存改变后的新值
3）如果当前内存中的值等于变量的旧值，则将新值赋值到当前变量；否则不进行任何操作

b.对于这个机制，你不需要使用任何同步机制，这样你就避免了 死锁，也获得了更好的性能。这种机制能保证多个并发线程对一个共享变量操作做到最终一致。

c.Java 在原子类中实现了CAS机制。这些类提供了compareAndSet() 方法；这个方法是CAS操作的实现和其他方法的基础。

d.Java 中还引入了原子数组，用来实现Integer类型和Long类型数组的操作，如AtomicIntegerArray 类来操作原子数组。

e.Java仅提供了另一个原子 array类。它是 AtomicLongArray 类，与 IntegerAtomicArray 类提供了相同的方法。这些类的一些其他的方法有：
get(int i): 返回array中第i个位置上的值
set(int I, int newValue): 设置array中第i个位置上的值为newValue

```
public class Decrementer implements Runnable {

 private AtomicIntegerArray vector;
 public Decrementer(AtomicIntegerArray vector) {
  this.vector=vector;
 }
 @Override
 public void run() {
  for (int i=0; i<vector.length(); i++) {
   vector.getAndDecrement(i);
  }
 }
}

public class Incrementer implements Runnable {

 private AtomicIntegerArray vector;
 public Incrementer(AtomicIntegerArray vector) {
  this.vector=vector;
 }

 @Override
 public void run() {

  for (int i=0; i<vector.length(); i++){
   vector.getAndIncrement(i);
  }
 }
}
public class Main {

 public static void main(String[] args) {

  final int THREADS=100;
  AtomicIntegerArray vector=new AtomicIntegerArray(1000);
  Incrementer incrementer=new Incrementer(vector);
  Decrementer decrementer=new Decrementer(vector);
  Thread threadIncrementer[]=new Thread[THREADS];
  Thread threadDecrementer[]=new Thread[THREADS];
  for (int i=0; i<THREADS; i++) {
   threadIncrementer[i]=new Thread(incrementer);
   threadDecrementer[i]=new Thread(decrementer);
   threadIncrementer[i].start();
   threadDecrementer[i].start();
  }

  for (int i=0; i<THREADS; i++) {
   try {
    threadIncrementer[i].join();
```

```
    threadDecrementer[i].join();
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }

  for (int i=0; i<vector.length(); i++) {
   if (vector.get(i)!=0) {
    System.out.println("Vector["+i+"] : "+vector.get(i));
   }
  }
  System.out.println("Main: End of the example");
 }
}
```

结果

```
<terminated> Main (27) [Java Application
Main: End of the example
```

结果分析：本例使用AtomicIntegerArray对象实现了下面两个不同的任务：

1）Incrementer 任务：使用getAndIncrement()增加数组中所有元素的值；

2）Decrementer 任务：使用getAndIncrement()方法较少数组的所有元素的值。

Main类中，执行了100个Incrementer 任务和100个Decrementer 任务，如果没有不一致的错误，数组中的元素都必须为0，所以程序结果为：Main: End of the example