第 4 章 使用开源软件 Amino 构建并发应用程序

在实际的并发线程应用程序中，常常会用到数组、树、图、集合等数据结构，而这些结构也涉及到并发线程所遇到的安全问题。采用 Amino 组件可以很方便地实现线程安全的数据结构。本章将介绍 Amino 组件在 Java 多线程中的使用。

4.1 开源软件 Amino 介绍

Amino是Apache旗下的开源软件。读者可以访问http://amino-cbbs.sourceforge.net/得到其最新版本。面向并发编程，它有以下特点：

1） 可操作性和良好的伸缩性

2） 跨平台性

3） 无论在 Java、C++或其他流行语言中，编程风格一致

4） 适用于多核的各种操作系统

5） 可以进行并发编程正确性的测试

本章将介绍 Amino 的 Java 版。Amino Java 类库将提供优化后的并发线程组件，适用于 JDK6.0 及其以后的版本。

Amino Java 类库将涉及下面四个方面的内容：

1） 数据结构

该组件将提供一套免锁的集合类。因为这些数据结构采用免锁的运算法则来生成，所以，它们将拥有基本的免锁组件的特性，如可以避免不同类型的死锁，不同类型的线程初始化顺序等。

2） 并行模式

Amino 将为应用程序提供一个或几个大家熟知的并行计算模式。采用这些并行模式可以使开发者起到事半功倍的效果，这些模式包括 Master-Worker、Map-reduce、Divide and conquer, Pipeline 等，线程调度程序可以与这些模式类协同工作，提供了开发效率。

3） 并行计算中的一般功能

Amino 将为应用程序提供并行计算中常用的方法，例如：

a. String、Sequence 和 Array 的处理方面。如 Sort、Search、Merge、Rank、Compare、Reverse、 Shuffle、Rotate 和 Median 等

b. 处理树和图的方法：如组件连接，树生成，最短路径，图的着色等

4）原子和 STM（软件事务内存模型）

下面的程序可以简单地演示使用 Amino 的例子：

```
// LogServerGood.java
package    org.amino.logserver;

import    org.amino.ds.lockfree.LockFreeQueue;
public    class    LogServerGood {
   /*Standard Queue interface*/
   private    Queue<String> queue;
   public    LogServerGood() throws    IOException {
/*Amino components are compatible with standard interface whenever    possible*/
     queue = new    LockFreeQueue<String>();
     }
}
```

4.2 无锁（Lock-Free）数据结构

我们知道，在传统的多线程环境下，我们需要共享某些数据，但为了避免竞争条件引致数据出现不一致的情况，某些代码段需要变成基于锁（Lock based）的原子操作去执行。加锁可以让某一线程可以独占共享数据，避免竞争条件，确保数据一致性。从好的一面来说，只要互斥体是在锁状态，就可以放心地进行任何操作，不用担心其它线程会闯进来搞坏你的共享数据。

然而，正是这种在互斥体的锁状态下可以为所欲为的机制同样也带来了很大的问题。例如，可以在锁期间读键盘或进行某些耗时较长的 I/O 操作，这种阻塞意味着其它想要占用正占用着的互斥体的线程只能被搁在一旁等着。更糟的是有可能引起死锁。基于锁（Lock based）的多线程设计更可能引发死锁、优先级倒置、饥饿等情况，令到一些线程无法继续其进度。

在 Amino 类库中，主要算法将使用锁无关的（Lock-Free）的数据结构。

锁无关（Lock-Free）算法，顾名思义，即不牵涉锁的使用。这类算法可以在不使用锁的情况下同步各个线程。对比基于锁的多线程设计，锁无关算法有以下优势：

- 对死锁、优先级倒置等问题免疫：它属于非阻塞性同步，因为它不使用锁来协调各个线程，所以对死锁、优先级倒置等由锁引起的问题免疫；
- 保证程序的整体进度：由于锁无关算法避免了死锁等情况出现，所以它能确保线程是在运行当中，从而确保程序的整体进度；
- 性能理想：因为不涉及使用锁，所以在普遍的负载环境下，使用锁无关算法可以

得到理想的性能提升。

自 JDK5 推出之后，包 java.util.concurrent.atomic 中的一组类为实现锁无关算法提供了重要的基础。锁无关数据结构是线程安全的，在使用时无需再编写额外代码去确保竞争条件不会出现。

而在锁无关多线程编程的世界里，几乎任何操作都是无法原子地完成的。只有很小一集操作可以被原子地进行，这一限制使得锁无关编程的难度大大地增加了。锁无关编程带来的好处是在线程进展和线程交互方面，借助于锁无关编程，你能够对线程的进展和交互提供更好的保证。

经过十几年的发展，锁无关的数据结构已经非常成熟，性能并不逊色于传统的实现方式。虽然编写锁无关算法十分困难的，但因为数据结构是经常被重用的部分，开发者可以使用现成的 API（如 Amino）轻易让程序进入锁无关的世界。

首先，一个“等待无关”的程序可以在有限步之内结束，而不管其它线程的相对速度如何。

一个“锁无关”的程序能够确保执行它的所有线程中至少有一个能够继续往下执行。这便意味着有些线程可能会被任意地延迟，然而在每一步都至少有一个线程能够往下执行。尽管有些线程的进度可能不如其它线程来得快，但系统作为一个整体总是在“前进”的。而基于锁的程序则无法提供上述的任何保证。一旦任何线程持有了某个互斥体并处于等待状态，那么其它任何想要获取同一互斥体的线程就只好站着干瞪眼；一般来说，基于锁的算法无法摆脱“死锁”或“活锁”的阴影，前者如两个线程互相等待另一个所占有的互斥体，后者如两个线程都试图去避开另一个线程的锁行为，就像两个在狭窄桥面上撞了个照面的家伙，都试图去给对方让路，结果像跳舞似的摆来摆去最终还是面对面走不过去。

2003 年，Maurice Herlihy因他在 1991 年发表的开创性论文“Wait-Free Synchronization”（ http://www.podc.org/dijkstra/2003.html）而获得了分布式编程的Edsger W. Dijkstra奖。在论文中，Herlihy证明了哪些原语对于构造锁无关数据结构来说是好的，哪些则是不好的。他证明了一些简单的结构就足以实现出任何针对任意数目的线程的锁无关算法。例如，Herlihy证明了原语Compare-and-swap（CAS） 是实现锁无关数据结构的通用原语。CAS 可以原子地比较一个内存位置的内容及一个期望值，如果两者相同，则用一个指定值取替这个内存位置里的内容，并且提供结果指示这个操作是否成功。很多现代的处理器已经提供了 CAS 的硬件实现，例如在 x86 架构下的 CMPXCHG8 指令。而在 Java 下，位于 java.util.concurrent.atomic 内的 AtomicReference<V> 类亦提供了 CAS 原语的实现，并且有很多其他的扩展功能。

下面我们来简单的了解一下硬件同步指令的工作方式：

在进行多处理时，现代 CPU 都可以通过检测或者阻止其他处理器的并发访问来更新共享内存，最通用的方法是实现 CAS（比较并转换）指令，例如在 Intel 处理器中 CAS 是通过 cmpxchg 指令实现的。CAS 操作过程是：当处理器要更新一个内存位置的值的时候，它首先将目前内存位置的值与它所知道的修改前的值进行对比（要知道在多处理的时候，你要更新的内存位置上的值有可能被其他处理更新过，而你全然不知），如果内存位置目前的值与期望的原值相同（说明没有被其他处理更新过），那么就将新的值写入内存位置；而如果不同（说明有其他处理在我不知情的情况下改过这的值咯），那么就什么也不做，不写入新的值（现在最新的做法是定义内存值的版本号，根据版本号的改变来判断内存值是否被修改，一般情况下，比较内存值的做法已经满足要求了）。CAS 的价值所在就在于它是在硬件级别实现的，速度那是相当的快。JDK5.0 中的原子类就是利用了现代处理器中的这个特性，可以在不进行锁定的情况下，进行共享属性访问的同步。

下面我们将举例说明锁无关栈（Stack）的实现方法。

栈能以数组或者链表作为底下的储存结构，虽然采取链表为基础的实现方式会占用多一点空间去储存代表元素的节点，但却可避免处理数组溢出的问题。故此我们将以链表作为栈的基础，本文不打算展开对栈数据结构的论述，仅给出相应的实现代码：

```
// 锁无关的栈实现
import java.util.concurrent.atomic.*;
 class Node<T> {
     Node<T> next;
     T value;
     public Node(T value, Node<T> next) {
          this.next = next;
          this.value = value;
     }
 }
 public class Stack<T> {
     AtomicReference<Node<T>> top = new AtomicReference<Node<T>>();
     public void push(T value) {
          boolean sucessful = false;
          while (!sucessful) {
               Node<T> oldTop = top.get();
               Node<T> newTop = new Node<T>(value, oldTop);
               sucessful = top.compareAndSet(oldTop, newTop);
          };
     }
```

```
        public T peek() {
            return top.get().value;
        }
        public T pop() {
            boolean sucessful = false;
            Node<T> newTop = null;
            Node<T> oldTop = null;
            while (!sucessful) {
                oldTop = top.get();
                newTop = oldTop.next;
                sucessful = top.compareAndSet(oldTop, newTop);
            }
            return oldTop.value;
        }
    }
```

成员数据 top 的类型为 AtomicReference<Node<T>> ， AtomicReference<V> 这个类可以对 top 数据成员施加 CAS 操作，亦即是可以允许 top 原子地和一个期望值比较，两者相同的话便用一个指定值取代。

4.3 应用 Amino 提供的数据结构

Amino Java 并发类库提供了应用程序常用的一些数据结构，如集合、树和图等，下面将分别举例说明。

4.3.1 简单集合

在 Amino 并发类库提供了 List,Queque,Set,Vector,Dirctionary,Stack,Deque 等数据结构，采用 Lock-Free 数据结构，可以确保线程安全。

1、 List

在 java.util.*包中，List 接口继承了 Collection 并声明了类集的新特性。使用一个基于零的下标，元素可以通过它们在列表中的位置被插入和访问。一个列表可以包含重复元素。Collection 接口是构造集合框架的基础，它声明所有类集合都将拥有的核心方法。

下面的例子中将采用 Amino 提供的线程安全的 `LockFreeList` 集合类。

【例 4.1】 采用并发线程的方式，构造共享 List 集合

```
// ListTest.java
    package org.amino.test;
  import java.util.List;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeList; //Amino 提供的无锁数据结构

public class ListTest {
    private static final int ELEMENT_NUM = 80;
    public static void main(String[] argvs) {
        ExecutorService exec = Executors.newCachedThreadPool();
        final List<String> listStr = new LockFreeList<String>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new ListInsTask(listStr));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(500, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Size of list is " + listStr.size());
        for (int i = 1; i <= ELEMENT_NUM; ++i) {
            if (!listStr.contains(i)) {
                System.out.println("didn't find " + i);
            }
        }
    }
}
class ListInsTask implements Runnable {
    private static AtomicInteger count = new AtomicInteger();
    List list;
    public ListInsTask(List l) {
        list = l;
    }
    public void run() {
        if ( list.add(count.incrementAndGet())) {
            System.out.println("List Size= " + list.size() );
        }else{
                System.out.println("did not insert " + count.get());
          }
    }
}
```

程序运行结果可能（根据计算机具体情况而变化）如下：

```
List Size= 1
List Size= 2
List Size= 3
……
List Size= 33
List Size= 34
List Size= 35
List Size= 80
List Size= 79
List Size= 78
……
List Size= 38
List Size= 37
List Size= 36
Size of list is 80
```

该程序在线程池中运行，可以看出，线程的调度是并发和抢先式的。线程的结束和创建的顺序是不一样的，但依然保证了结果的正确性。

对上面的程序进行简单的修改，使用 Amino 提供的 LockFreeOrderedList 类，就可以实现有序的线程安全的 List 集合。

【例 4.2】 采用并发线程的方式，实现无锁结构的有序 List 集合

```
// OrderedListTest.java
package org.amino.test;
import java.util.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeOrderedList;
public class OrderedListTest{
    private static final int ELEMENT_NUM = 80;
    public static void main(String[] argvs) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        final List<Integer> listStr = new LockFreeOrderedList<Integer>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new ListInsTask1(listStr));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(500, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
                }

                System.out.println("Size of list is " + listStr.size());
                Thread.sleep(600L);
                Iterator iterator = listStr.iterator();
                int nn=1;
                while (iterator.hasNext()) {
                    System.out.println( "After order:"+nn+"="+(Integer) iterator.next() );
nn++;
                }
            }
        }

        class ListInsTask1 implements Runnable {
            private static AtomicInteger count = new AtomicInteger();
            List list;
            public ListInsTask1(List l) {
                list = l;
            }
            public void run() {
                int rom;
                rom=(int)(1000 * java.lang.Math.random());
                System.out.println("rom="+rom);
                if ( list.add(count. addAndGet ( rom ) ) ){
                    System.out.println("List Size= " + list.size() );
                }else{
                    System.out.println("did not insert " + count.get());
                }
            }
        }
```

该程序将得到一个有序的共享集合序列 List。部分结果如下：

```
。。。。。。
After order:1=324
After order:2=852
After order:3=1291
After order:4=1640
After order:5=1754
After order:6=2560
After order:7=3377
After order:8=3594
。。。。。。  // 省略余下的部分。
```

2. Set

Set 接口定义了一个集合。它继承了 Collection 并说明了不允许重复元素的类集的特性。因此，如果试图将重复元素加到集合中时，add( )方法将返回 false。下面例子将使用 Amino 提供的无数锁的线程安全的 LockFreeSet.

【例 4.3】 采用并发线程的方式，无锁结构的 Set 集合

```
// SetTest.java
package org.amino.test;
import java.util.Iterator;
import java.util.Set;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeSet;
public class SetTest {
 private static final int ELEMENT_NUM = 80;
 public static void main(String[] argvs) throws Exception {
 ExecutorService exec = Executors.newCachedThreadPool();
 final Set<Integer> setStr = new LockFreeSet<Integer>();
 Future[] results = new Future[ELEMENT_NUM];
 for (int i = 0; i < ELEMENT_NUM; ++i) {
        results[i] = exec.submit(new SetInsTask(setStr));
 }
 try {
                for (int i = 0; i < ELEMENT_NUM; ++i) {
                        results[i].get();
                }
        } catch (InterruptedException e) {
                e.printStackTrace();
        } catch (ExecutionException e) {
                e.printStackTrace();
        }
        exec.shutdown();
        try {
                exec.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
                e.printStackTrace();
        }
        System.out.println("Size of set is " + setStr.size());
        for (int i = 1; i <= ELEMENT_NUM; ++i) {
                if (!setStr.contains(i)) {
```

```
                    System.out.println("didn't find " + i);
                }
            }
        Thread.sleep(600L);
        Iterator iterator = setStr.iterator();
            int nn=1;
            while (iterator.hasNext()) {
                System.out.println( "After insert:"+nn+"="+(Integer) iterator.next() ); nn++;
            }
        }
    }
    class SetInsTask implements Runnable {
        private static AtomicInteger count = new AtomicInteger();
        Set set;
        public SetInsTask(Set q) {
            set = q;
        }
        public void run() {
            if (!set.add(count.incrementAndGet())) {
                System.out.println("did not insert " + count.get());
            }
        }
    }
```

程序运行结果可能（根据计算机具体情况而变化）如下：

```
Size of set is 80
After insert:1=68
After insert:2=21
After insert:3=42
After insert:4=63
……
After insert:78=71
After insert:79=41
After insert:80=60
```

4.3.2 树

在树的种类中，二叉树是一种常见的数据结构。从二叉树的递归定义可知，一棵非空的二叉树由根结点及左、右子树这三个基本部分组成。下面的例子将使用 Amino 提供的无锁线程安全的二叉树

【例 4.4】无锁结构的二叉树

```
    // TreeTest.java
package org.amino.test;
package org.amino.examples;
import java.util.Queue;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import org.amino.ds.lockfree.LockFreeQueue;
import org.amino.mcas.LockFreeBSTree;

public class TreeTest {
 private static final int ELEMENT_NUM = 1000;
 public static void main(String[] argvs) {
 ExecutorService exec = Executors.newCachedThreadPool();
final LockFreeBSTree<Integer, String> bstree = new LockFreeBSTree<Integer, String>();
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            exec.submit(new InsertTask(bstree));
        }
        exec.shutdown();
        try {
            exec.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Boolean result = true;
        for (int i = 1; i < ELEMENT_NUM; ++i) {
            if (bstree.find(i) == null) {
                System.out.println("didn't find " + i);
                result = false;
            }
        }

        if(result) {
            System.out.println("Test successfully!");
        }
       }
}

class InsertTask implements Runnable {
 private static AtomicInteger count = new AtomicInteger();
 LockFreeBSTree tree;

 public InsertTask(LockFreeBSTree tr) {
        tree = tr;
 }
 public void run() {
```

```
            int c = count.incrementAndGet();
            if(tree.update(c, c) != null) {
                System.out.println("did not insert " + c);
            }
        }
    }
```

程序运行结果如下：

```
Test successfully!
```

结果说明该集合建立成功。

4.3.3 图

一般的图算法涉及对图属性和类型、图搜索、有向图、最小生成树、最短路径以及网络流等研究。图形数据结构在工程设计、地理空间信息、模式识别等多方面有广泛的用途。Amino组件中实现了线程并发情况下线程安全的无锁数据结构Graph接口。Graph接口的内容如下：

```
package org.amino.ds.graph;
import java.util.Collection;
public interface Graph<E> extends Collection<E>, Cloneable {
    Collection<Node<E>> getNodes(E e);
    Collection<Node<E>> getAllNodes();
    Collection<Edge<E>> getEdges(Node<E> start, Node<E> end);
    Collection<Edge<E>> getEdges(E start, E end);
    Node<E> addNode(E e);
    Node<E> addNode(Node<E> node);
    boolean addAllNodes(Collection<Node<E>> nodes);
    boolean addEdge(E start, E end, double weight);
    boolean addEdge(Node<E> start, Node<E> end, double weight);
    boolean addEdge(Edge<E> edge);
    Collection<AdjacentNode<E>> getLinkedNodes(Node<E> node);
    Collection<Edge<E>> getLinkedEdges(Node<E> node);
    boolean removeEdge(Node<E> start, Node<E> end);
    boolean removeEdge(Edge<E> edge);
    boolean removeEdge(E start, E end);
    boolean removeNode(Node<E> node);
    boolean containsEdge(E start, E end);
    Graph<E> clone() throws CloneNotSupportedException;
    boolean containsNode(Node<E> start);
}
```

从上面的接口中可以看出，在Graph中实现了对节点、边的操作。由于Gragh的操作涉及太多的代码行，本章中没有给出响应的实例。有兴趣的读者可以参考amino-cbbs-0.3.1.jar和它的原代码，以及网上的案例，其网址是

http://amino-cbbs.sourceforge.net/qs_cpp_examples.html。（作者完成本书时，Amino组件正处于0.3.1版的阶段，很多功能还没有开发出来。）

4.4 Amino 使用的模式和调度算法

在Amino并发库中，将使用的模式和调度算法有：Master-Worker, Map-reduce, Divide and conquer, Pipeline等几种。本节将对Master-Worker作简单介绍。

Master－Worker 是一类典型的并行计算。在这类应用中，存在一个 Master，由它将一个大问题进行分割，分割后的各个小问题送给各个 Worker 进行计算，最后由 Master 将所有 Worker 计算结果进行汇总。在这一类的应用中，Master 只进行少量的计算，而主要的计算工作由各个 Worker 进行。

Master－Worker 并行计算模式分为静态模式和动态模式。在静态模式中，计算过程在不同的进度中进行。首先，所有的被分割后的各个小问题同时被分派给 Worker，然后 Worker 开始紧张的计算。在动态模式中，分派问题和计算小问题是同时动态进行的。

在Amino开源代码中提供了Master－Worker工厂模式，代码如下：

```
package org.amino.pattern.internal;

/**
 * Classes for a MasterWorker Factory.
 * @author blainey
 *
 */
public class MasterWorkerFactory {
 /**
  *
  * @param <X> input type
  * @param <Y> result type
  * @param r work item
  * @return    StaticMasterWorker
  */
 public static <X,Y> MasterWorker<X,Y> newStatic(Doable<X,Y> r) {
       return new StaticMasterWorker<X,Y>(r);
 }

 /**
  *
  * @param <X> input type
  * @param <Y> result type
```

```
     * @param r work item
     * @param numWorkers number of workers (threads)
     * @return StaticMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newStatic(Doable<X,Y> r, int numWorkers) {
        return new StaticMasterWorker<X,Y>(r,numWorkers);
    }

    /**
     *
     * @param <X> input type
     * @param <Y> result type
     * @param r work item
     * @return DynamicMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newDynamic(DynamicWorker<X,Y> r) {
        return new DynamicMasterWorker<X,Y>(r);
    }

    /**
     *
     * @param <X> input type
     * @param <Y> result type
     * @param r work item
     * @param numWorkers number of workers
     * @return DynamicMasterWorker
     */
    public static <X,Y> MasterWorker<X,Y> newDynamic(DynamicWorker<X,Y> r, int
numWorkers) {
        return new DynamicMasterWorker<X,Y>(r,numWorkers);
    }
   }
```

从上面的代码中，我们可以看出Amino提供了StaticMasterWorker、DynamicMasterWorker两种底层的Master-Worker算法。

AbstractMasterWorker.java提供了Master-Worker算法的中基本的实现，由于代码太长，请读者参阅Amino的提供的源代码

对于StaticMasterWorker算法，他继承了AbstractMasterWorker类，其实现如下：

```
package org.amino.pattern.internal;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
import org.amino.scheduler.internal.AbstractScheduler;
```

```
    /**
     * Classes for a static MasterWorker, where upper bound of master workers is fixed once work
     * is initiated.
     *
     * @param <S> input type.
     * @param <T> result type.
     */
    class StaticMasterWorker<S,T> extends AbstractMasterWorker<S,T> {
     protected Queue<WorkItem> workQ = new ConcurrentLinkedQueue<WorkItem>();

     /**
      * @author ganzhi
      *
      */
     private class WorkWrapper implements Runnable {
          private Doable<S,T> w;

          public void run() {
               while (true) {
                    /*
                    // Go wait in the staff lounge
                    if (!waitInLounge()) break;
                    */

                    workerPool.startWork();
                    try {
                         while(true) {
                              final WorkItem input = workQ.poll();
                              if (input == null) break;

                              final T output = w.run(input.value());
                              resultMap.put(input.key(),output);
                         }
                    } finally {
                         workerPool.complete();
                         break;
                    }
               }
          }

          /**
           *
           * @param w work item
```

```
             */
            public WorkWrapper (Doable<S,T> w) {
                this.w = w;
            }
    }

    /**
     *
     * @param r work item
     */
    public StaticMasterWorker(Doable<S,T> r) {
            this(r,AbstractScheduler.defaultNumberOfWorkers());
    }
    /**
     *
     * @param r work item
     * @param numWorkers size of worker pool.
     */
    public StaticMasterWorker(Doable<S,T> r, int numWorkers) {
            super(numWorkers);

            Runnable run = new WorkWrapper(r);
            for (int i=0; i<numWorkers; i++) workerPool.createWorker(i, run);
    }

    public boolean isStatic() { return true; }

    /**
       * {@inheritDoc}
       */
    public ResultKey submit(S w) {
            if (workerPool.anyStarted()) return null;

            ResultKey key = new ResultKeyImpl();
            boolean added = workQ.offer(new WorkItem(w,key));

            // The queue is unbounded so should fail to add new entries only in out of memory
situations
            assert added;

            return key;
    }
   }
```

4.5 Amino 的简单使用

下面讲解一个使用的Amino的集合的简单使用。在本例中，我们将采用字符串作为集合对象的值，然后对其进行排序及搜索等操作。

【例4.5】 多线程状态下LockFreeVector的使用及排序

```
package sample.amino;
import java.util.*;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

import org.amino.ds.lockfree.*;
import org.amino.alg.sort.*;

public class StringDealSort {
    private static final int ELEMENT_NUM = 80;

    public static void main(String[] argvs) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();

final LockFreeVector<String> vectorStr = new LockFreeVector<String>();

        Future[] results = new Future[ELEMENT_NUM];
        for (int i = 0; i < ELEMENT_NUM; ++i) {
            results[i] = exec.submit(new VectorInsTaskSort(vectorStr));
        }

        try {
            for (int i = 0; i < ELEMENT_NUM; ++i) {
                results[i].get();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        exec.shutdown();
        try {
```

```
                exec.awaitTermination(60, TimeUnit.SECONDS);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Size of set is " + vectorStr.size());

            for (int i = 1; i <= ELEMENT_NUM; ++i) {
                if (!vectorStr.contains(new Integer(i).toString())) {
                    System.out.println("didn't find " + i);
                }
            }

            Thread.sleep(10L);
            QuickSorter qs=new QuickSorter();
            qs.sort(vectorStr);

            Iterator iterator = vectorStr.iterator();
            int nn=1;
            while (iterator.hasNext()) {
                System.out.println(  "After  insert:"+nn+"="+(String)  iterator.next()  );
nn++;
            }
        }

    }

    class VectorInsTaskSort implements Runnable {
        private static AtomicInteger count = new AtomicInteger();
        LockFreeVector vector;

        public VectorInsTaskSort(LockFreeVector q) {
            vector = q;
        }

        public void run() {
            int rom=0;
            rom=(int)(26 * java.lang.Math.random());
            String romstr=new Integer(rom).toString();
   if( vector.contains(romstr) ) {
    if( vector.contains( new Integer(rom+26).toString() ) ) {
        if( vector.contains( new Integer(rom+26*2).toString() ) ) {
            if( vector.contains( new Integer(rom+26*3).toString() ) ) {
                    vector.add( new Integer(rom+26*4).toString() );
```

```
                                }else{
                                    vector.add( new Integer(rom+26*3).toString() );
                }
                }else{
                vector.add( new Integer(rom+26*2).toString() );
            }
          }else{
            vector.add( new Integer(rom+26).toString() );
          }
     }else{
    vector.add(romstr);
     }
    }
    }
```

在本例中，我们在每个线程中向vector添加随机字符串对象，均采用了比较，如果存在，则在原来的字符值上再加上26，变成字符对象后然后再行添加，如此算法重复4次后，结果基本上没有重复的。

从本例中可以看出，vector的操作是线程安全的。

Linux公社（LinuxIDC.com）于2006年9月25日注册并开通网站，Linux现在已经成为一种广受关注和支持的一种操作系统，IDC是互联网数据中心，LinuxIDC就是关于Linux的数据中心。

LinuxIDC.com提供包括Ubuntu，Fedora，SUSE技术，以及最新IT资讯等Linux专业类网站。

并被收录到Google 网页目录-计算机 > 软件 > 操作系统 > Linux 目录下。

Linux公社（LinuxIDC.com）设置了有一定影响力的Linux专题栏目。

包括:

Ubuntu专题

Fedora专题

RedHat专题

SUSE专题

红旗Linux专题

Android专题