java并发包

## 一 同步控制工具

### 重量级锁 Synchronized

类锁，对象锁，变量锁

#### 1.1.1 类锁

就是对整个静态的class文件加锁，也就是说一个地方用到了这个class文件，其他地方，就要等待获取到class的锁。典型的比如我们的单例模式。就是控制类级别的锁。

class ThreadEF {

public void MethodA(){

synchronized(ThreadEF.class){ }

}

public static synchronized void MethodB(){}

}

如上代码所示，如果一个地方正在使用这个类，那么其他地方就不能够再使用这个类了。这就是类级别的锁。然而方法A,与B就是互斥的。也就是说这两种写法都是类锁的写法。

#### 1.1.2 对象锁

class ThreadEG {

public void MethodA(){

synchronized(this){ }

}

public synchronized void MethodB(){}

}

如上所示的对象加锁方法就是对象锁，一个类可以有多个实例，然而多个实例直接各自调用各自的方法是不会影响的，但是如上所示加了对象锁之后，那么这个实例是不可以对同时调用A与B方法的，这两个方法是同步的。但是如果是两个实例的话，这两个方法就是异步的。如果两个实例要实现同步，那么就得把锁上升到类锁。

#### 1.1.3 变量锁

class ThreadEH {

private Object object=new Object();

public void MethodA(){

synchronized(object){}

}

public void MethodB(){

synchronized (object){ }

}

}

如上代码所示，是通过变量来实现，A,B方法的同步的，然而这样的同步其实也是对象锁的特例。它无非还是让对象的A,B两个方法不能同时执行。那么上面的index++问题，大家应该知道怎么处理了吧。那就是类锁来解决。

### 重入索 ReentrantLock

**public** **class** ReetrantLock1 **implements** Runnable {

**private** **static** **int** *index*=0;

**public** **static** ReentrantLock *lock*=**new** ReentrantLock(**true**);

@Override

**public** **void** run() {

*lock*.lock();

**try** {

**for**(**int** i=0;i<10000;i++){

*index*++;

}

} **catch** (Exception e) {

// **TODO**: handle exception

} **finally**{

*lock*.unlock();

}

}

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

Thread thread=**new** Thread(**new** ReetrantLock1());

Thread thread2=**new** Thread(**new** ReetrantLock1());

thread.start();

thread2.start();

thread.join();

thread2.join();

System.***out***.println(*index*);

}

}

#### 1.2.1 ReentrantLock与synchronized比较

(1) 可以指定公平性。而synchronized锁默认是不公平的锁，重入锁的公平锁可以保证每个线程都有执行的机会。Synchronized不公平锁就不可以了，它会使得优先级比较低的线程处于饥饿状态。

(2) Synchronized的性能明显要弱于重入锁,但是实际开发中由于吞吐量的问题，不至于因为锁导致巨大的性能影响。所以基本还是使用Synchronized。

(3) Synchronized是jvm 自动进行管理的，自动获取锁，自动释放锁。Lock是手动管理的需要手动的申请锁资源或者释放锁资源。

(4) Synchronized不支持锁中断要么等待，要么锁定。而ReentrantLock是支持锁中断的,在某些特殊情况下，锁中断也是非常有必要的。

比如死锁的代码

**public** **class** ReetranLockInterrupt {

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

ReentrantLock lock1=**new** ReentrantLock();

ReentrantLock lock2=**new** ReentrantLock();

ThreadM threadM=**new** ThreadM(lock1, lock2);

ThreadH threadH=**new** ThreadH(lock1, lock2);

threadM.start();

threadH.start();

}

}

**class** ThreadM **extends** Thread{

**private** ReentrantLock lock1=**null**;

**private** ReentrantLock lock2=**null**;

**public** ThreadM(ReentrantLock lock1,ReentrantLock lock2) {

**this**.lock1=lock1;

**this**.lock2=lock2;

}

@Override

**public** **void** run() {

lock1.lock();

**try** {

Thread.*sleep*(1000);

lock2.lock();

}**catch** (InterruptedException e) {

}**finally**{

lock2.unlock();

lock1.unlock();

}

}

}

**class** ThreadH **extends** Thread{

**private** ReentrantLock lock1=**null**;

**private** ReentrantLock lock2=**null**;

**public** ThreadH(ReentrantLock lock1,ReentrantLock lock2) {

**this**.lock1=lock1;

**this**.lock2=lock2;

}

@Override

**public** **void** run() {

lock2.lock();

**try** {

Thread.*sleep*(1000);

lock1.lock();

} **catch** (InterruptedException e) {

}**finally**{

lock1.unlock();

lock2.unlock();

}

}

}

如上所示是死锁的代码，如果是Synchronized,那么这个锁是不会轻易解锁的，那么lock怎么进行解锁呢。

**public** **class** ReetranLockInterrupt {

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

ReentrantLock lock1=**new** ReentrantLock();

ReentrantLock lock2=**new** ReentrantLock();

ThreadM threadM=**new** ThreadM(lock1, lock2);

ThreadH threadH=**new** ThreadH(lock1, lock2);

threadM.start();

threadH.start();

Thread.*sleep*(5000);

threadH.interrupt();

threadM.interrupt();

}

}

**class** ThreadM **extends** Thread{

**private** ReentrantLock lock1=**null**;

**private** ReentrantLock lock2=**null**;

**public** ThreadM(ReentrantLock lock1,ReentrantLock lock2) {

**this**.lock1=lock1;

**this**.lock2=lock2;

}

@Override

**public** **void** run() {

**try** {

lock1.lockInterruptibly();

Thread.*sleep*(1000);

lock2.lockInterruptibly();

}**catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally**{

System.***out***.println("ThreadM---退出中断");

lock2.unlock();

lock1.unlock();

}

}

}

**class** ThreadH **extends** Thread{

**private** ReentrantLock lock1=**null**;

**private** ReentrantLock lock2=**null**;

**public** ThreadH(ReentrantLock lock1,ReentrantLock lock2) {

**this**.lock1=lock1;

**this**.lock2=lock2;

}

@Override

**public** **void** run() {

**try** {

lock2.lockInterruptibly();

Thread.*sleep*(1000);

lock1.lockInterruptibly();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally**{

System.***out***.println("ThreadH---退出中断");

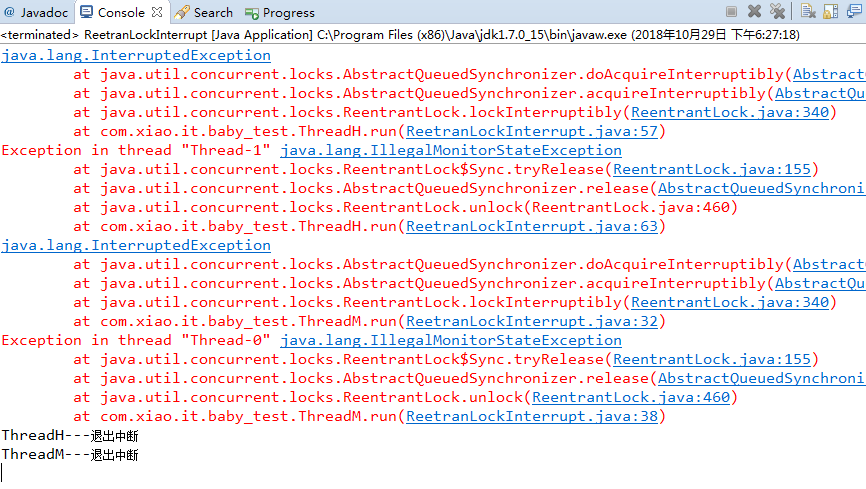
lock1.unlock();

lock2.unlock();

}

}

}



如上图所示ReetrantLock提供了锁的中断机制。这样可以有效的解决死锁问题。

### 重入条件 newCondition

**public** **class** LockConditions {

**private** **static** String *name*;

**private** **static** String *sex*;

**private** **static** Lock *lock*=**new** ReentrantLock();

**public** **static** Condition *condition*=*lock*.newCondition();

**static** **boolean** *flag*=**false**;

/\*\*

\* 写线程

\* **@author** Administrator

\*/

**static** **class** WriteThread **extends** Thread{

**int** num=0;

**public** WriteThread(){}

**public** **void** run(){

**while**(**true**){

**try** {

*lock*.lock();

**if**(*flag*){

*condition*.await();

}

**if**(num%2==1){

*name*="张三";

*sex*="男";

}**else**{

*name*="李四";

*sex*="女";

}

num++;

*flag*=**true**;

*condition*.signal();

}**catch**(Exception lock){

lock.printStackTrace();

}**finally** {

*lock*.unlock();

}

}

}

}

/\*\*

\* 读线程

\* **@author** Administrator

\*

\*/

**static** **class** ReadThread **extends** Thread{

**public** ReadThread(){ }

**public** **void** run(){

**while**(**true**){

**try** {

*lock*.lock();

**if**(!*flag*){

*condition*.await();

}

Thread.*sleep*(1000);

System.***out***.println(*name*+":"+*sex*);

*flag*=**false**;

*condition*.signal();

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}**finally**{

*lock*.unlock();

}

}

}

}

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

WriteThread thread=**new** WriteThread();

thread.start();

ReadThread readThread=**new** ReadThread();

readThread.start();

}

}

如上所示重入锁中的await(),notify(),notifyAll(),类似于对象锁中的wait()、notify()、notifyAll()是定义在Object类里的方法，可以用来控制线程的状态。这三个方法最终调用的都是jvm级的native方法。随着jvm运行平台的不同可能有些许差异。如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。如果对象调用了notify方法就会通知某个正在等待这个对象的控制权的线程可以继续运行。如果对象调用了notifyAll方法就会通知所有等待这个对象控制权的线程继续运行。

### 限时等待锁 lock.tryLock()

还是刚才重入锁中的代码,假设我们还是用相同的方式实现死锁，只是我们换一下锁的方式，把lock换成trylock

**public** **class** ReetranLockInterrupt {

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

ReentrantLock lock1 = **new** ReentrantLock();

ReentrantLock lock2 = **new** ReentrantLock();

ThreadM threadM = **new** ThreadM(lock1, lock2);

ThreadH threadH = **new** ThreadH(lock1, lock2);

threadM.start();

threadH.start();

}

}

**class** ThreadM **extends** Thread {

**private** ReentrantLock lock1 = **null**;

**private** ReentrantLock lock2 = **null**;

**public** ThreadM(ReentrantLock lock1, ReentrantLock lock2) {

**this**.lock1 = lock1;

**this**.lock2 = lock2;

}

@Override

**public** **void** run() {

**while**(**true**){

**if**(lock1.tryLock()){

**try** {

**if**(lock2.tryLock()){

System.***out***.println("事情做完了");

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

lock2.unlock();

}

}

lock1.unlock();

}

}

}

**class** ThreadH **extends** Thread {

**private** ReentrantLock lock1 = **null**;

**private** ReentrantLock lock2 = **null**;

**public** ThreadH(ReentrantLock lock1, ReentrantLock lock2) {

**this**.lock1 = lock1;

**this**.lock2 = lock2;

}

@Override

**public** **void** run() {

**while**(**true**){

**if**(lock2.tryLock()){

**try** {

**if**(lock1.tryLock()){

System.***out***.println("事情做完了");

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

lock1.unlock();

}

}

lock2.unlock();

}

}

}

### 1.5 信号量 Semaphore

Semaphore是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore可以用来构建一些对象池，资源池之类的，比如数据库连接池，我们也可以创建计数为1的Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态,这个类似于操作系统概念中的pv操作，如果不放信号量就不能做PV操作。它的用法如下：

availablePermits函数用来获取当前可用的资源数量

wc.acquire(); //申请资源

wc.release();// 释放资源

举个例子

需求: 一个厕所只有3个坑位，但是有10个人来上厕所，那怎么办？假设10的人的编号分别为1-10，并且1号先到厕所，10号最后到厕所。那么1-3号来的时候必然有可用坑位，顺利如厕，4号来的时候需要看看前面3人是否有人出来了，如果有人出来，进去，否则等待。同样的道理，4-10号也需要等待正在上厕所的人出来后才能进去，并且谁先进去这得看等待的人是否有素质，是否能遵守先来先上的规则。

代码:

**public** **class** Semaphones **implements** Runnable {

**private** Semaphore semaphore;

**public** Semaphones(Semaphore semaphore) {

**this**.semaphore=semaphore;

}

@Override

**public** **void** run() {

**try**{

// 剩下的资源(剩下的茅坑)

**int** availablePermits = semaphore.availablePermits();

**if** (availablePermits > 0) {

System.***out***.println(Thread.*currentThread*().getName()+"天助我也,终于有茅坑了...");

} **else** {

System.***out***.println(Thread.*currentThread*().getName()+"怎么没有茅坑了...");

}

//申请茅坑 如果资源达到3次，就等待

semaphore.acquire();

System.***out***.println(Thread.*currentThread*().getName()+"终于轮我上厕所了..爽啊");

Thread.*sleep*(**new** Random().nextInt(1000)); // 模拟上厕所时间。

System.***out***.println(Thread.*currentThread*().getName()+"厕所上完了...");

semaphore.release();

}**catch**(Exception e){

e.printStackTrace();

}

}

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

//三个茅坑

Semaphore semaphore=**new** Semaphore(3);

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

**for**(**int** i=0;i<10;i++){

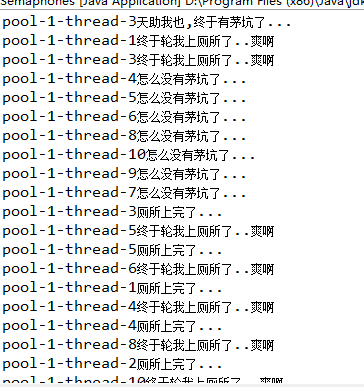
executorService.execute(**new** Semaphones(semaphore));

}

}

}

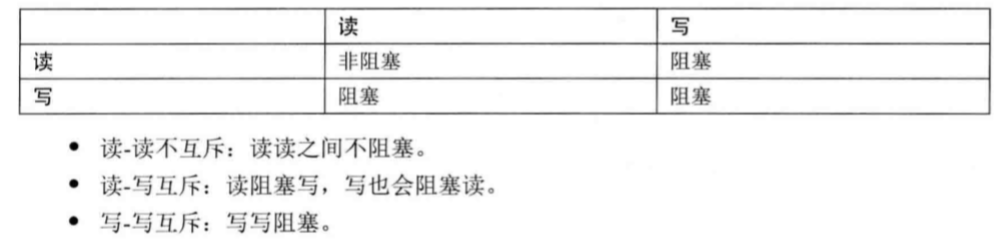
运行结果



这就是信号量的作用，它可以限定资源的数量， 资源用完以后等待，直到下一波释放资源以后再进行资源的抢占。类似于操作系统的PV操作。

### 1.6 读写锁 ReentrantReadWriteLock

读写分离可以有效的减少锁的竞争，以提升系统的性能。用锁分离机制来提升系统的性能非常容易理解。比如线程A1,A2,A3进行写操作。线程B1,B2,B3进行读操作。如果使用重入锁或者内部锁，理论上不管是读写，写读，读读都是串行操作的。然而往往读数据是不会破坏数据一致性的。所以读写所提供了以上的访问优势。



这在读操作明显多于写操作，并且读操作比较耗时的场景下，可以明显提高读写性能。

**public** **class** LockConditions {

**private** **static** String *name*;

**private** **static** String *sex*;

**private** **static** ReentrantReadWriteLock *readAndWritelock*=**new** ReentrantReadWriteLock();

**private** **static** Lock *readLock*= *readAndWritelock*.readLock();

**private** **static** Lock *wirtLock*=*readAndWritelock*.writeLock();

**static** **boolean** *flag*=**false**;

/\*\*

\* 写线程

\* **@author** Administrator

\*/

**static** **class** WriteThread **extends** Thread{

**private** Lock lock=**null**;

**private** **int** num=0;

**public** WriteThread(Lock lock,**int** num){

**this**.lock=lock;

**this**.num=num;

}

**public** WriteThread(){}

**public** **void** run(){

**try** {

lock.lock();

Thread.*sleep*(2000);

**if**(num%2==1){

*name*="张三";

*sex*="男";

}**else**{

*name*="李四";

*sex*="女";

}

}**catch**(Exception lock){

lock.printStackTrace();

}**finally** {

lock.unlock();

}

}

}

/\*\*

\* 读线程

\* **@author** Administrator

\*

\*/

**static** **class** ReadThread **extends** Thread{

**private** Lock lock=**null**;

**public** ReadThread(Lock lock){

**this**.lock=lock;

}

**public** **void** run(){

**try** {

lock.lock();

Thread.*sleep*(3000);

System.***out***.println(*name*+":"+*sex*);

} **catch** (Exception e) {

e.printStackTrace();

}**finally**{

lock.unlock();

}

}

}

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

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

**for**(**int** i=0;i<10;i++){

executorService.execute(**new** WriteThread(*wirtLock*,i));

}

**for**(**int** i=0;i<100;i++){

executorService.execute(**new** ReadThread(*readLock*));

}

executorService.shutdown();

}

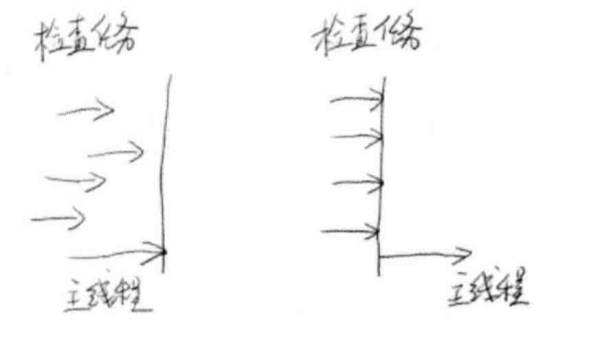
}

如上代码段所示，明显可以看出读操作的次数，明显多于写操作的次数，而且读操作又比较耗时，此时采用读写锁的分离技术就可以明显的提高系统性能。

### 1.7 计数器 CountDownLatch

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。中文界把它叫做倒计时门闩。

原理图



如上图所示我们要等待所有的子线程完成所有的任务以后主线程才开始执行，它的代码类似于写了多个线程，依次使用了join()方法进行了等待操作。代码示例

**public** **class** CountDownLocach **implements** Runnable {

**private** **static** CountDownLatch *countDownLatch*;

**public** CountDownLocach(CountDownLatch countDownLatch) {

**this**.*countDownLatch*=countDownLatch;

}

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(1000);

System.***out***.println(Thread.*currentThread*().getName()+":"+"执行完毕");

*countDownLatch*.countDown();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

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

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

CountDownLatch countDownLatch=**new** CountDownLatch(10);

**for**(**int** i=0;i<10;i++){

executorService.execute(**new** CountDownLocach(countDownLatch));

}

**try** {

countDownLatch.await();

System.***out***.println("所有线程执行完毕了 可以执行主线程了");

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

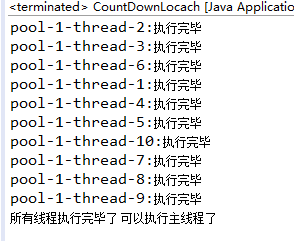
e.printStackTrace();

}

}

}

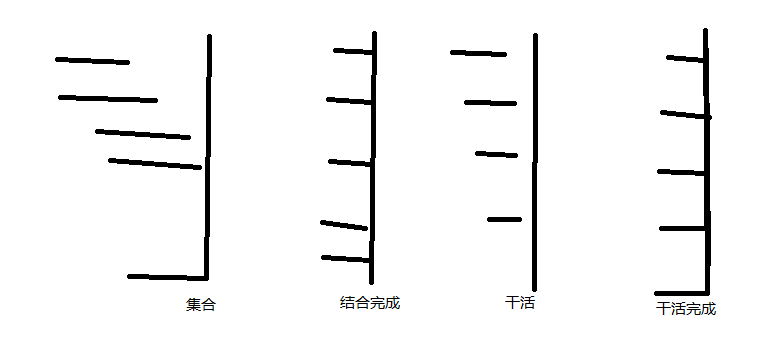
执行结果



### 1.8 循环栅栏 CyclicBarrier

CyclicBarrier初始化时规定一个数目，然后计算调用了CyclicBarrier.await()进入等待的线程数。当线程数达到了这个数目时，所有进入等待状态的线程被唤醒并继续。

 CyclicBarrier就象它名字的意思一样，可看成是个障碍， 所有的线程必须到齐后才能一起通过这个障碍。 CyclicBarrier初始时还可带一个Runnable的参数， 此Runnable任务在CyclicBarrier的数目达到后，所有其它线程被唤醒前被执行。



循环栏杆的工作原理如上图所示，他可以多次控制信号量实际上类似于CountDownLatch

这样的话他可以等待集合完成以后，在开始干某一件事情，对于有序的并发来说也算是一个不错的选择。

**public** **class** CyclicBarrireA **implements** Runnable{

**private** CyclicBarrier cyclicBarriter;

**private** **static** **boolean** *flag*=**true**;

**public** CyclicBarrireA(CyclicBarrier cyclicBarriter){

**this**.cyclicBarriter=cyclicBarriter;

}

@Override

**public** **void** run() {

doGather();

**try** {

cyclicBarriter.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BrokenBarrierException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**if**(*flag*){

System.***out***.println("集合完成");

*flag*=**false**;

}

doWork();

**try** {

cyclicBarriter.await();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (BrokenBarrierException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**public** **void** doGather(){

System.***out***.println(Thread.*currentThread*().getName()+":"+"集合");

}

**public** **void** doWork(){

System.***out***.println(Thread.*currentThread*().getName()+":"+"干活");

}

**public** **static** **void** main(String[] args) **throws** InterruptedException, BrokenBarrierException {

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

CyclicBarrier cyclicBarriter=**new** CyclicBarrier(10);

**for**(**int** i=0;i<10;i++){

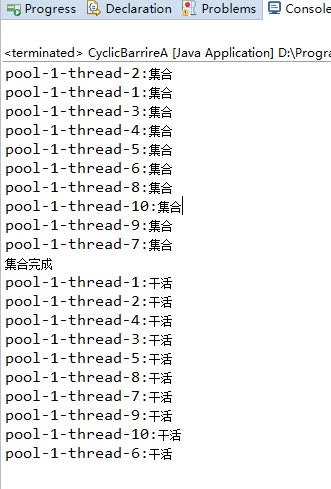
executorService.execute(**new** CyclicBarrireA(cyclicBarriter));

}

executorService.shutdown();

}

}

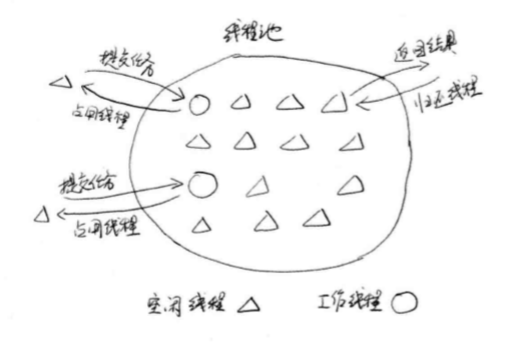


上图是执行结果，由上图可以看出，它可以在线程里面控制所有的线程进行同步等待。

## 二 线程池

### 2.1 线程池与优缺点

为了避免频繁的创建于销毁线程。我们可以让线程进行服用。如果大家进行过数据库开发。对数据库的连接池并不会陌生。为了避免频繁的创建于销毁数据库连接。我们可以使用数据库连接池来维护数据库的连接，使其处于一个被激活的状态。当需要连接数据库时，不是创建一个数据库连接，而是在连接池中获取一个已经开启的数据库连接。线程池也是这样的例子，当你用完线程以后不是直接关闭它，而是把这个线程放回到线程池中。

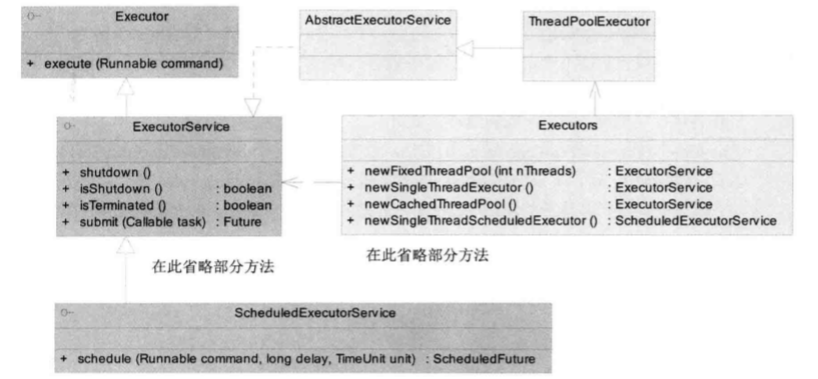


线程池的优点

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。  
第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。  
第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，  
还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用线程池，必须对其实现原理了如指掌。

### 2.2 线程池原理

就是如上图所示的概念，在java中 jdk为我们提供了线程池的实现。我们可以直接调用，或者进行改造。



如上图所示是线程池的模型。

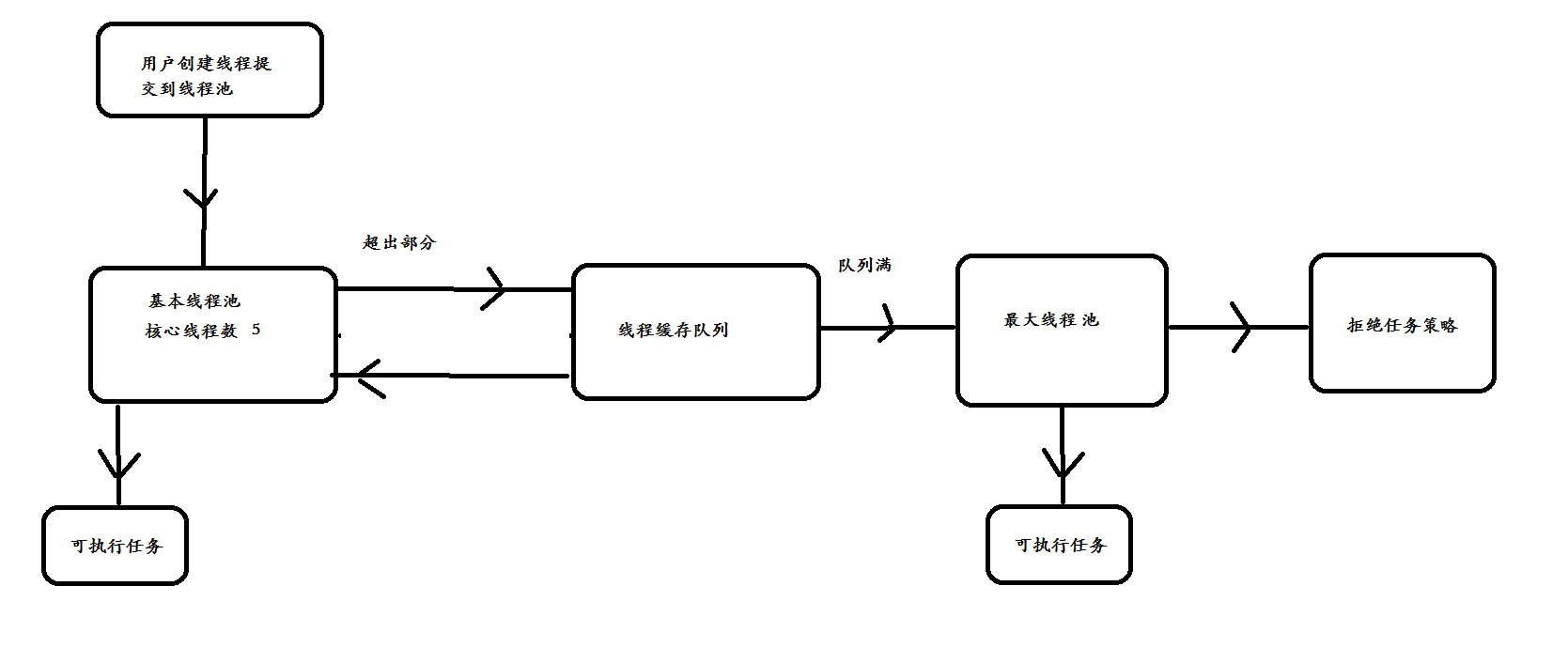
Executor框架的最顶层实现是ThreadPoolExecutor类，Executors工厂类中提供的newScheduledThreadPool、newFixedThreadPool、newCachedThreadPool方法其实也只是ThreadPoolExecutor的构造函数参数不同而已。通过传入不同的参数，就可以构造出适用于不同应用场景下的线程池，那么它的底层原理是怎样实现的呢，这篇就来介绍下ThreadPoolExecutor线程池的运行过程。

corePoolSize： 核心池的大小。 当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中

maximumPoolSize： 线程池最大线程数，它表示在线程池中最多能创建多少个线程；

keepAliveTime： 表示线程没有任务执行时最多保持多久时间会终止。

unit： 参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性



如上图所示

1、判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

### 2.3 线程池分类

Java通过Executors（jdk1.5并发包）提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程

newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

#### 2.3.1 newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

|  |
| --- |
| **// 无限大小线程池 jvm自动回收**  **ExecutorService newCachedThreadPool = Executors.*newCachedThreadPool*();**  **for (int i = 0; i < 10; i++) {**  **final int temp = i;**  **newCachedThreadPool.execute(new Runnable() {**  **@Override**  **public void run() {**  **try {**  **Thread.*sleep*(100);**  **} catch (Exception e) {**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",i:" + temp);**  **}**  **});**  **}** |

**总结: 线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。**

#### 2.3.2 newFixedThreadPool

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下：

|  |
| --- |
| **ExecutorService newFixedThreadPool = Executors.*newFixedThreadPool*(5);**  **for (int i = 0; i < 10; i++) {**  **final int temp = i;**  **newFixedThreadPool.execute(new Runnable() {**  **@Override**  **public void run() {**  **System.*out*.println(Thread.*currentThread*().getId() + ",i:" + temp);**  **}**  **});**  **}** |

**总结:因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。**

**定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()**

#### 2.3.3 newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

|  |
| --- |
| ScheduledExecutorService newScheduledThreadPool = Executors.*newScheduledThreadPool*(5);  for (int i = 0; i < 10; i++) {  final int temp = i;  newScheduledThreadPool.schedule(new Runnable() {  public void run() {  System.*out*.println("i:" + temp);  }  }, 3, TimeUnit.*SECONDS*);  } |

#### 2.3.4 newSingleThreadExecutor

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

|  |
| --- |
| **ExecutorService newSingleThreadExecutor = Executors.*newSingleThreadExecutor*();**  **for (int i = 0; i < 10; i++) {**  **final int index = i;**  **newSingleThreadExecutor.execute(new Runnable() {**  **@Override**  **public void run() {**  **System.*out*.println("index:" + index);**  **try {**  **Thread.*sleep*(200);**  **} catch (Exception e) {}**  **}**  **});**  **}** |

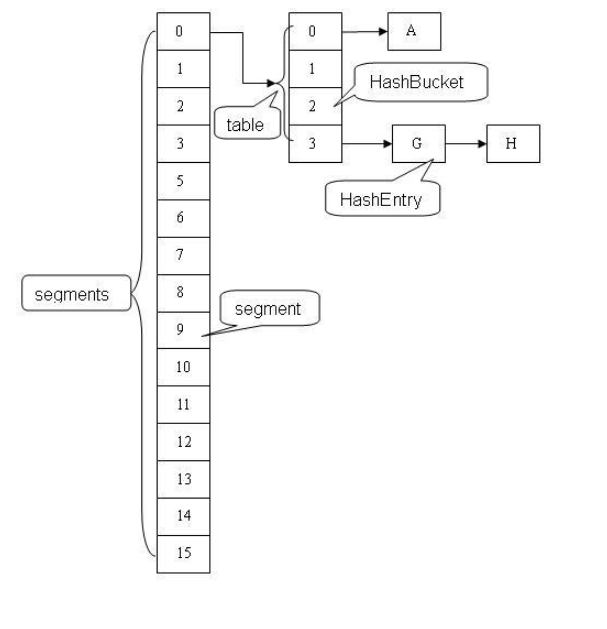
注意: 结果依次输出，相当于顺序执行各个任务。

如上代码所示就是线程池的使用代码，如上我创建了一个固定有十个工作线程的线程池。

## 三 并发容器

#### 3.1 concurrentHashMap

是一个线程安全的hashMap，它可以解决多线程环境下，hashmap线程不安全的问题。解决hashmap线程不安全的问题的方式当然还有Collections.*synchronizedMap*(**new** HashMap())这种方式也可以实现hashMap的同步。然而concurrentHashMap是一个更高效的，并发的hashmap



如上是concurrentHashMap的数据结构，可以看出concurrentHashMap室友16个segment组成的。每个segment属于自己的分段锁。也就是说concurrentHashMap最大支持16个并发的线程进行写操作。如果简单的理解的话我们可以说hashtable是线程安全的。而concurrentHashMap是有16个小的hashtable构成的。不过实际上不是这样的。每个segment

对应一张表。这张表又是一个HashBucket与多个HashEntry组成的。也就是说每个segment对应一个通过链表存储解决冲突的哈希表。

至于什么是Hash表，还有冲突，请读者自行翻看《数据结构的书》解决冲突的方式，包括散列，再散列，链地址法。而这里所采用的方法就是链地址法。所以基于这样的分段锁的机制可以有效的保证数据的一致性与并发访问的效率。

#### 3.2 CopyOnWriteArrayList

这个集合已经自动实现了读写锁的功能了，正如我们前在讲解ReentrantReadWriteLock

的时候已经提到过的，加锁的方式不应该影响读与读的操作，下面这个集合就是专门为读写操作设置的不需要手动加锁的线程安全的集合类。

**public** **class** CopyOnWriteList **implements** Runnable{

**private** List list=**null**;

**private** ConcurrentLinkedQueue concurrentLinkedQueue=**null**;

**private** CountDownLatch countdown=**null**;

**private** CopyOnWriteArrayList copyOnWriteArrayList=**null**;

**private** **int** num;

**public** CopyOnWriteList(CopyOnWriteArrayList copyOnWriteArrayList,List list,ConcurrentLinkedQueue linkedQueue,CountDownLatch countdown,**int** num) {

**this**.list=list;

**this**.concurrentLinkedQueue=linkedQueue;

**this**.countdown=countdown;

**this**.num=num;

**this**.copyOnWriteArrayList=copyOnWriteArrayList;

}

@Override

**public** **void** run() {

**try** {

**if**(num==0){

copyOnWriteArrayList.add("1");

}**else**{

Object peek = copyOnWriteArrayList.get(0);

**new** Random().nextInt();

}

countdown.countDown();

} **catch** (Exception e) {

e.printStackTrace();

}

}

@SuppressWarnings({ "rawtypes", "rawtypes" })

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

//执行时间:=74 li size:=30

List list=Collections.*synchronizedList*(**new** ArrayList());

//执行时间:=71 li size:=30

ConcurrentLinkedQueue concurrentLinkedQueue=**new** ConcurrentLinkedQueue();

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

CountDownLatch countdown=**new** CountDownLatch(100030);

//执行时间:=64 li size:=30

CopyOnWriteArrayList copyOnWriteArrayList=**new** CopyOnWriteArrayList();

**long** timeA = System.*currentTimeMillis*();

**for**(**int** i=0;i<30;i++){

executorService.execute(**new** CopyOnWriteList(copyOnWriteArrayList,list, concurrentLinkedQueue,countdown,0));

}

**for**(**int** i=0;i<100000;i++){

executorService.execute(**new** CopyOnWriteList(copyOnWriteArrayList,list, concurrentLinkedQueue,countdown,1));

}

countdown.await();

**long** timeB = System.*currentTimeMillis*();

//System.out.println("执行时间:="+(timeB-timeA)+" li size:="+list.size());

System.***out***.println("执行时间:="+(timeB-timeA)+" li size:="+copyOnWriteArrayList.size());

//long timeC = System.currentTimeMillis();

// System.out.println("执行时间:="+(timeC-timeB)+" li size:="+concurrentLinkedQueue.size());

executorService.shutdown();

}

}

如上代码所示，不同的集合在读写比较大的情况下的使用情况，由于线程数还相对比较小，所以呢就只能看出微小的性能差异。

#### 3.3 concurrentlinkedqueue

同concurrentHashMap类似，这是一个支持并发且线程安全的链表,它的底层使用了CAS无锁操作。我们都知道ArrayList是不支持线程安全的。Collections.*synchronizedList*(**new** ArrayList())通过集合工具类我们可以实现ArrayList的线程安全。

**public** **class** Dbss **implements** Runnable{

**private** List list=**null**;

**private** ConcurrentLinkedQueue concurrentLinkedQueue=**null**;

**private** CountDownLatch countdown=**null**;

**public** Dbss(List list,ConcurrentLinkedQueue linkedQueue,CountDownLatch countdown) {

**this**.list=list;

**this**.concurrentLinkedQueue=linkedQueue;

**this**.countdown=countdown;

}

@Override

**public** **void** run() {

**try** {

**for**(**int** i=0;i<1000000;i++){

concurrentLinkedQueue.add(i);

}

countdown.countDown();

} **catch** (Exception e) {

e.printStackTrace();

}

}

@SuppressWarnings({ "rawtypes", "rawtypes" })

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

//执行时间:=2886 li size:=10000000

List list=Collections.*synchronizedList*(**new** ArrayList());

//执行时间:=1446 li size:=10000000

ConcurrentLinkedQueue concurrentLinkedQueue=**new** ConcurrentLinkedQueue();

ExecutorService executorService=Executors.*newFixedThreadPool*(10);

CountDownLatch countdown=**new** CountDownLatch(10);

**long** timeA = System.*currentTimeMillis*();

**for**(**int** i=0;i<10;i++){

executorService.execute(**new** Dbss(list, concurrentLinkedQueue,countdown));

}

countdown.await();

**long** timeB = System.*currentTimeMillis*();

//System.out.println("执行时间:="+(timeB-timeA)+" li size:="+list.size());

System.***out***.println("执行时间:="+(timeB-timeA)+" li size:="+concurrentLinkedQueue.size());

executorService.shutdown();

}

}

如上所示代码可以看出concurrentLinkedQueue的代码所示，性能明显可以快了近一倍，但当我们使用concurrentLinkedQueue.size()的时候，因为这个要遍历整个集合，所以这个方法的性能不会特别好。所以这里在判断集合为空的时候，尽量用你isempty而不要用size()==0这样不会导致性能问题。concurrentLinkedQueue性能好，但是concurrentLinkedQueue所需要的内存空间也比较大。所以这里如果在计算次数比较多的情况下，要分配给足额的jvm堆内存。

#### 3.4 Blockingqueue

**public** **class** ConcurrentCalculator {

**private** **static** BlockingDeque<Msg> *addQ*=**new** LinkedBlockingDeque<ConcurrentCalculator.Msg>();

**private** **static** BlockingDeque<Msg> *mulQ*=**new** LinkedBlockingDeque<ConcurrentCalculator.Msg>();

**private** **static** BlockingDeque<Msg> *divQ*=**new** LinkedBlockingDeque<ConcurrentCalculator.Msg>();

**static** **class** AddThread **implements** Runnable{

@Override

**public** **void** run() {

**while**(**true**){

**try** {

Msg take = *addQ*.take();

**int** i = take.getI();

**int** j = take.getJ();

take.setJ(i+j);

*mulQ*.add(take);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**static** **class** Multiple **implements** Runnable{

@Override

**public** **void** run() {

**while**(**true**){

**try** {

Msg take = *mulQ*.take();

**int** j = take.getJ();

take.setJ(j\*10);

*divQ*.add(take);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**static** **class** DivDid **implements** Runnable{

@Override

**public** **void** run() {

**while**(**true**){

Msg take;

**try** {

take = *divQ*.take();

**int** j = take.getJ();

take.setJ(j/5);

**long** sxs=take.getI()+take.getJ();

System.***out***.println(take.getObj()+" "+sxs);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**static** **class** Msg{

**private** **int** i;

**private** **int** j;

**private** String obj=**null**;

**public** **int** getI() {

**return** i;

}

**public** **void** setI(**int** i) {

**this**.i = i;

}

**public** **int** getJ() {

**return** j;

}

**public** **void** setJ(**int** j) {

**this**.j = j;

}

**public** String getObj() {

**return** obj;

}

**public** **void** setObj(String obj) {

**this**.obj = obj;

}

}

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

**new** Thread(**new** AddThread()).start();

**new** Thread(**new** Multiple()).start();

**new** Thread(**new** DivDid()).start();

**for**(**int** i=0;i<1000;i++){

**for**(**int** j=0;j<1000;j++){

Msg msg=**new** Msg();

msg.setI(i);

msg.setJ(j);

msg.setObj("i:="+i+" j:="+j);

*addQ*.add(msg);

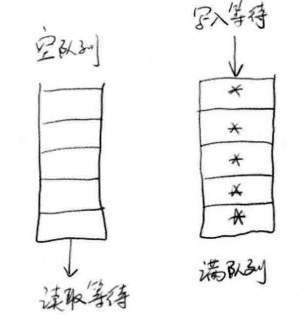
}

}

}

}

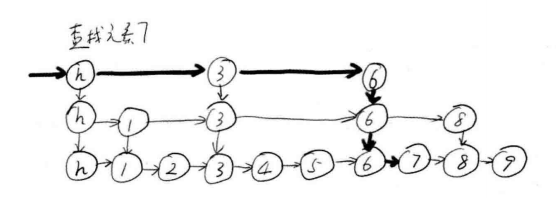
Blockingqueue是一个接口，它名下有很多的阻塞队列。



他的工作模式就是当队列为空的时候消费线程等待，当队列满的时候写入线程等待。如上代码段是一个并行计算的代码块，通过blockingqueue实现了。流水线的并行计算，是一个通过阻塞队列与多线程实现，把顺序执行的步奏并行化的例子。一般来说阻塞队列非常适合于生产者，消费者模式的线程。比如小区的物业管理与用户的意见的意见箱，比如我们的MQ作为系统之间通信的阻塞队列。通过中间件就可以做到系统之间能够平滑升级。

#### 3.5 concurrentskiplistmap

Concurrentskiplistmap的底层实现是通过调表的方式，底层通过CAS与微粒度锁来解决并发的线程安全问题。调表结构的特点就是查询性能快，有序。如果需要存储键值对而且还要保证键的有序性，那么调表就是不二的选择。



**public** **class** ConcurrentSki **implements** Runnable{

**private** ConcurrentSkipListMap concurrentSkipListMap=**null**;

**private** CountDownLatch countdown=**null**;

**public** ConcurrentSki(ConcurrentSkipListMap concurrentSkipListMap,CountDownLatch countdown){

**this**.concurrentSkipListMap=concurrentSkipListMap;

**this**.countdown=countdown;

}

@Override

**public** **void** run() {

**this**.concurrentSkipListMap.put(Thread.*currentThread*().getName(), "fghjk");

countdown.countDown();

}

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

CountDownLatch countdown=**new** CountDownLatch(10000);

ConcurrentSkipListMap concurrentSkipListMap=**new** ConcurrentSkipListMap();

ExecutorService executorService=Executors.*newFixedThreadPool*(100);

**for**(**int** i=0;i<100;i++){

executorService.execute(**new** ConcurrentSki(concurrentSkipListMap,countdown));

}

countdown.await();

Set entrySet = concurrentSkipListMap.entrySet();

Iterator iterator = entrySet.iterator();

**while**(iterator.hasNext()){

Entry<Integer, Integer> next = (Entry<Integer, Integer>) iterator.next();

System.***out***.println("key:="+next.getKey()+" value"+next.getValue());

}

}

}

如上代码所示，可以看出这个结果是一个按照key排序后的结果，而且也并没有出现线程不安全的问题。性能也得到了提高。

