http://www.importnew.com/21889.html

在java 1.5中，提供了一些非常有用的辅助类来帮助我们进行并发编程，比如CountDownLatch，CyclicBarrier和Semaphore，今天我们就来学习一下这三个辅助类的用法。

以下是本文目录大纲：

一.CountDownLatch用法

二.CyclicBarrier用法

三.Semaphore用法

一.CountDownLatch用法

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。

CountDownLatch类只提供了一个构造器：

1

public CountDownLatch(int count) { }; //参数count为计数值

然后下面这3个方法是CountDownLatch类中最重要的方法：

public void await() throws InterruptedException { }; //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行

public boolean await(long timeout, TimeUnit unit) throws InterruptedException { }; //和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行

public void countDown() { }; //将count值减1

下面看一个例子大家就清楚CountDownLatch的用法了：

public class Test {

public static void main(String[] args) {

final CountDownLatch latch = new CountDownLatch(2);

new Thread(){

public void run() {

try {

System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");

Thread.sleep(3000);

System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");

latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

};

}.start();

new Thread(){

public void run() {

try {

System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");

Thread.sleep(3000);

System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");

latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

};

}.start();

try {

System.out.println("等待2个子线程执行完毕...");

latch.await();

System.out.println("2个子线程已经执行完毕");

System.out.println("继续执行主线程");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

执行结果：

线程Thread-0正在执行

线程Thread-1正在执行

等待2个子线程执行完毕...

线程Thread-0执行完毕

线程Thread-1执行完毕

2个子线程已经执行完毕

继续执行主线程

**二.CyclicBarrier用法**

字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了。

CyclicBarrier类位于java.util.concurrent包下，CyclicBarrier提供2个构造器：

public CyclicBarrier(int parties, Runnable barrierAction) {

}

public CyclicBarrier(int parties) {

}

参数parties指让多少个线程或者任务等待至barrier状态；参数barrierAction为当这些线程都达到barrier状态时会执行的内容。

然后CyclicBarrier中最重要的方法就是await方法，它有2个重载版本：

public int await() throws InterruptedException, BrokenBarrierException { };

public int await(long timeout, TimeUnit unit)throws InterruptedException,BrokenBarrierException,TimeoutException { };

第一个版本比较常用，用来挂起当前线程，直至所有线程都到达barrier状态再同时执行后续任务；

第二个版本是让这些线程等待至一定的时间，如果还有线程没有到达barrier状态就直接让到达barrier的线程执行后续任务。

下面举几个例子就明白了：

假若有若干个线程都要进行写数据操作，并且只有所有线程都完成写数据操作之后，这些线程才能继续做后面的事情，此时就可以利用CyclicBarrier了：

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N);

for(int i=0;i<N;i++)

new Writer(barrier).start();

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

cyclicBarrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println("所有线程写入完毕，继续处理其他任务...");

}

}

}

执行结果：

线程Thread-0正在写入数据...

线程Thread-3正在写入数据...

线程Thread-2正在写入数据...

线程Thread-1正在写入数据...

线程Thread-2写入数据完毕，等待其他线程写入完毕

线程Thread-0写入数据完毕，等待其他线程写入完毕

线程Thread-3写入数据完毕，等待其他线程写入完毕

线程Thread-1写入数据完毕，等待其他线程写入完毕

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

从上面输出结果可以看出，每个写入线程执行完写数据操作之后，就在等待其他线程写入操作完毕。

当所有线程线程写入操作完毕之后，所有线程就继续进行后续的操作了。

如果说想在所有线程写入操作完之后，进行额外的其他操作可以为CyclicBarrier提供Runnable参数：

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N,new Runnable() {

@Override

public void run() {

System.out.println("当前线程"+Thread.currentThread().getName());

}

});

for(int i=0;i<N;i++)

new Writer(barrier).start();

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

cyclicBarrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println("所有线程写入完毕，继续处理其他任务...");

}

}

}

运行结果：

1

2

3

4

5

6

7

8

9

10

11

12

13

线程Thread-0正在写入数据...

线程Thread-1正在写入数据...

线程Thread-2正在写入数据...

线程Thread-3正在写入数据...

线程Thread-0写入数据完毕，等待其他线程写入完毕

线程Thread-1写入数据完毕，等待其他线程写入完毕

线程Thread-2写入数据完毕，等待其他线程写入完毕

线程Thread-3写入数据完毕，等待其他线程写入完毕

当前线程Thread-3

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

所有线程写入完毕，继续处理其他任务...

从结果可以看出，当四个线程都到达barrier状态后，会从四个线程中选择一个线程去执行Runnable。

下面看一下为await指定时间的效果：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N);

for(int i=0;i<N;i++) {

if(i<N-1)

new Writer(barrier).start();

else {

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.printStackTrace();

}

new Writer(barrier).start();

}

}

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

try {

cyclicBarrier.await(2000, TimeUnit.MILLISECONDS);

} catch (TimeoutException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"所有线程写入完毕，继续处理其他任务...");

}

}

}

执行结果：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

线程Thread-0正在写入数据...

线程Thread-2正在写入数据...

线程Thread-1正在写入数据...

线程Thread-2写入数据完毕，等待其他线程写入完毕

线程Thread-0写入数据完毕，等待其他线程写入完毕

线程Thread-1写入数据完毕，等待其他线程写入完毕

线程Thread-3正在写入数据...

java.util.concurrent.TimeoutException

Thread-1所有线程写入完毕，继续处理其他任务...

Thread-0所有线程写入完毕，继续处理其他任务...

at java.util.concurrent.CyclicBarrier.dowait(Unknown Source)

at java.util.concurrent.CyclicBarrier.await(Unknown Source)

at com.cxh.test1.Test$Writer.run(Test.java:58)

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(Unknown Source)

at java.util.concurrent.CyclicBarrier.await(Unknown Source)

at com.cxh.test1.Test$Writer.run(Test.java:58)

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(Unknown Source)

at java.util.concurrent.CyclicBarrier.await(Unknown Source)

at com.cxh.test1.Test$Writer.run(Test.java:58)

Thread-2所有线程写入完毕，继续处理其他任务...

java.util.concurrent.BrokenBarrierException

线程Thread-3写入数据完毕，等待其他线程写入完毕

at java.util.concurrent.CyclicBarrier.dowait(Unknown Source)

at java.util.concurrent.CyclicBarrier.await(Unknown Source)

at com.cxh.test1.Test$Writer.run(Test.java:58)

Thread-3所有线程写入完毕，继续处理其他任务...

上面的代码在main方法的for循环中，故意让最后一个线程启动延迟，因为在前面三个线程都达到barrier之后，等待了指定的时间发现第四个线程还没有达到barrier，就抛出异常并继续执行后面的任务。

另外CyclicBarrier是可以重用的，看下面这个例子：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N);

for(int i=0;i<N;i++) {

new Writer(barrier).start();

}

try {

Thread.sleep(25000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("CyclicBarrier重用");

for(int i=0;i<N;i++) {

new Writer(barrier).start();

}

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

cyclicBarrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"所有线程写入完毕，继续处理其他任务...");

}

}

}

执行结果：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

线程Thread-0正在写入数据...

线程Thread-1正在写入数据...

线程Thread-3正在写入数据...

线程Thread-2正在写入数据...

线程Thread-1写入数据完毕，等待其他线程写入完毕

线程Thread-3写入数据完毕，等待其他线程写入完毕

线程Thread-2写入数据完毕，等待其他线程写入完毕

线程Thread-0写入数据完毕，等待其他线程写入完毕

Thread-0所有线程写入完毕，继续处理其他任务...

Thread-3所有线程写入完毕，继续处理其他任务...

Thread-1所有线程写入完毕，继续处理其他任务...

Thread-2所有线程写入完毕，继续处理其他任务...

CyclicBarrier重用

线程Thread-4正在写入数据...

线程Thread-5正在写入数据...

线程Thread-6正在写入数据...

线程Thread-7正在写入数据...

线程Thread-7写入数据完毕，等待其他线程写入完毕

线程Thread-5写入数据完毕，等待其他线程写入完毕

线程Thread-6写入数据完毕，等待其他线程写入完毕

线程Thread-4写入数据完毕，等待其他线程写入完毕

Thread-4所有线程写入完毕，继续处理其他任务...

Thread-5所有线程写入完毕，继续处理其他任务...

Thread-6所有线程写入完毕，继续处理其他任务...

Thread-7所有线程写入完毕，继续处理其他任务...

从执行结果可以看出，在初次的4个线程越过barrier状态后，又可以用来进行新一轮的使用。而CountDownLatch无法进行重复使用。

三.Semaphore用法

Semaphore翻译成字面意思为 信号量，Semaphore可以控同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

Semaphore类位于java.util.concurrent包下，它提供了2个构造器：

1

2

3

4

5

6

public Semaphore(int permits) { //参数permits表示许可数目，即同时可以允许多少线程进行访问

sync = new NonfairSync(permits);

}

public Semaphore(int permits, boolean fair) { //这个多了一个参数fair表示是否是公平的，即等待时间越久的越先获取许可

sync = (fair)? new FairSync(permits) : new NonfairSync(permits);

}

下面说一下Semaphore类中比较重要的几个方法，首先是acquire()、release()方法：

1

2

3

4

public void acquire() throws InterruptedException { } //获取一个许可

public void acquire(int permits) throws InterruptedException { } //获取permits个许可

public void release() { } //释放一个许可

public void release(int permits) { } //释放permits个许可

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。

release()用来释放许可。注意，在释放许可之前，必须先获获得许可。

这4个方法都会被阻塞，如果想立即得到执行结果，可以使用下面几个方法：

1

2

3

4

public boolean tryAcquire() { }; //尝试获取一个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

public boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException { }; //尝试获取一个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

public boolean tryAcquire(int permits) { }; //尝试获取permits个许可，若获取成功，则立即返回true，若获取失败，则立即返回false

public boolean tryAcquire(int permits, long timeout, TimeUnit unit) throws InterruptedException { }; //尝试获取permits个许可，若在指定的时间内获取成功，则立即返回true，否则则立即返回false

另外还可以通过availablePermits()方法得到可用的许可数目。

下面通过一个例子来看一下Semaphore的具体使用：

假若一个工厂有5台机器，但是有8个工人，一台机器同时只能被一个工人使用，只有使用完了，其他工人才能继续使用。那么我们就可以通过Semaphore来实现：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

public class Test {

public static void main(String[] args) {

int N = 8; //工人数

Semaphore semaphore = new Semaphore(5); //机器数目

for(int i=0;i<N;i++)

new Worker(i,semaphore).start();

}

static class Worker extends Thread{

private int num;

private Semaphore semaphore;

public Worker(int num,Semaphore semaphore){

this.num = num;

this.semaphore = semaphore;

}

@Override

public void run() {

try {

semaphore.acquire();

System.out.println("工人"+this.num+"占用一个机器在生产...");

Thread.sleep(2000);

System.out.println("工人"+this.num+"释放出机器");

semaphore.release();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

执行结果：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

工人0占用一个机器在生产...

工人1占用一个机器在生产...

工人2占用一个机器在生产...

工人4占用一个机器在生产...

工人5占用一个机器在生产...

工人0释放出机器

工人2释放出机器

工人3占用一个机器在生产...

工人7占用一个机器在生产...

工人4释放出机器

工人5释放出机器

工人1释放出机器

工人6占用一个机器在生产...

工人3释放出机器

工人7释放出机器

工人6释放出机器

下面对上面说的三个辅助类进行一个总结：

1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；

而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；

另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。