发件人：方堃 fangkun119@hotmail.com
主题：
日期：2021年3月4日 下午2:04
收件人：

CH21 多线程(D)：新类库构件

2016年10月1日 星期六
16:55

目录

1. 概述 【目录】

库	java.util.concurrent

2.CountDownLatch 【目录】

功能	设置task数量	向CountDownLatch上设置一个初始值(表示要等待的task数量)
	等待task全部完成	对CountDownLatch上做任何await()操作都将被阻塞，直到计数器为0，此时阻塞结束
	task与CountDownLatch的交互	Task结束时可以调用countDown()来减小计数值

注意	只能被触发一次	需要重置计数器，可考虑CyclicBarrier
	只有await阻塞	Task调用countDown()不会被阻塞
	Task之间的同步	需要Task们自己来处理
	CountDownLatch只保证WaitedTask完成后WaitingTask才开始运行，多个WaitedTask之间的同步需要单独的机制，多个WaitingTask之间也是	

例子

典型的应用场景是，将一块工作分成n个独立的子任务：CoundDownLatch初始值设置为n、每完成一部分CountDownLatch减1；阻塞这块工作完成的代码挂起在CountDownLatch的await()上

处理子任务的线程	`import java.util.concurrent.*;` `import java.util.*;` `import static net.mindview.util.Print.*;` `// Performs some portion of a task:` `class TaskPortion implements Runnable {` `  private static int counter = 0;` `  private final int id = counter++;` `  private static Random rand = new Random(47);` `  private final CountDownLatch latch;` `  TaskPortion(CountDownLatch latch) {` `  this.latch = latch;` `  }` `  public void run() {` `  try {` `  doWork();` `  latch.countDown();` `  } catch(InterruptedException ex) {` `  // Acceptable way to exit` `  }` `  }` `  public void doWork() throws InterruptedException {` `  TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));` `  print(this + "completed");` `  }` `  public String toString() {` `  return String.format("%1$-3d ", id);` `  }` `}`	`//处理子任务的线程` `class TaskWorkerRunnable implements Runnable {` `  ...` `  //与等待子任务的线程共用一个CountDownLatch(构造函数传入)` `  private final CountDownLatch countDownLatch;` `  TaskWorkerRunnabvle(CountDownLatch latch) {` `  this.countDownLatch = latch;` `  }` `  //线程` `  public void run() {` `  doWork();` `  countDownLatch.countDown();` `  }` `  ...` `}`
等待子任务的线程	`// Waits on the CountDownLatch:` `class WaitingTask implements Runnable {` `  private static int counter = 0;` `  private final int id = counter++;` `  private final CountDownLatch latch;` `  WaitingTask(CountDownLatch latch) {` `  this.latch = latch;` `  }` `  public void run() {` `  try {` `  latch.await();` `  print("Latch barrier passed for " + this);` `  } catch(InterruptedException ex) {` `  print(this + " interrupted");` `  }` `  }` `  public String toString() {` `  return String.format("WaitingTask %1$-3d ", id);` `  }` `}`	`class TaskWaitRunnable implements Runnable {` `  //与处理子任务的线程共用一个CountDownLatch(构造函数传入)` `  private final CountDownLatch countDownLatch;` `  TaskWaitRunnable(CountDownLatch latch) {` `  this.countDownLatch latch;` `  }` `  //线程函数` `  public void run() {` `  try {` `  //阻塞，等待子任务全部完成` `  countDownLatch.await();` `  } catch (InterruptedException e) {` `  ...` `  }` `  //做子任务全部完成以后才能做的事情` `  ...` `  }` `  ...` `}`
main	`public class CountDownLatchDemo {` `  static final int SIZE = 100;`	`ExecutorServicer exec = Executors.newCachedThreadPool();`

```
  public static void main(String[] args) throws Exception {
    ExecutorService exec = Executors.newCachedThreadPool();
    // All must share a single CountDownLatch object:
    CountDownLatch latch = new CountDownLatch(SIZE);
    for(int i = 0; i < 10; i++)
      exec.execute(new WaitingTask(latch));
    for(int i = 0; i < SIZE; i++)
      exec.execute(new TaskPortion(latch));
    print("Launched all tasks");
    exec.shutdown(); // Quit when all tasks complete
  }
} /* (Execute to see output) *///:~
```

```
// SIZE个子任务将被等待
CountDownLatch countDownLatch = new CountDownLatch(SIZE);
// 10个线程，等待SIZE个子任务全部完成
for (int i = 0; i < 10; ++i)
  exec.execute(new TaskWaitRunnable());
// SIZE个线程，每个线程完成一个子任务
for (int i = 0; i < SIZE; ++i)
  exec.execute(new TaskWorkerRunnable());
// 关闭executorService不再接受新的Runnable
exec.shutdown();
```

3.CyclicBarrier 循环栅栏【目录】

功能及用法	协作方	一组worker线程，一个barrier作业	
	协作模式	stage_1	worker[1][2][3]...[N]并发运行，每个线程完成一个作业后，进入阻塞阶段（通过调用barrier.awati()）
		stage_2	所有worker都完成作业时，最后一个调用barrier.await()的线程负责执行barrier作业
			下一轮（如果barrier作业没有决定中断/停止所有worker线程），进入state_1，开始下一轮循环（可重复）

例子：赛马：n匹马抽象成n个子任务，所有马都完成任务之后，一场比赛才宣告结束

赛马（工作线程）

```
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
  private static int counter = 0;
  private final int id = counter++;
  private int strides = 0;
  private static Random rand = new Random(47);
  private static CyclicBarrier barrier;
  public Horse(CyclicBarrier b) { barrier = b; }
  public synchronized int getStrides() { return strides; }
  public void run() {
    try {
      while(!Thread.interrupted()) {
        synchronized(this) {
          strides += rand.nextInt(3); // Produces 0, 1 or 2
        }
        barrier.await();
      }
    } catch(InterruptedException e) {
      // A legitimate way to exit
    } catch(BrokenBarrierException e) {
      // This one we want to know about
      throw new RuntimeException(e);
    }
  }
  public String toString() { return "Horse " + id + " "; }
  public String tracks() {
    StringBuilder s = new StringBuilder();
    for(int i = 0; i < getStrides(); i++)
      s.append("*");
    s.append(id);
    return s.toString();
  }
}
```

```
//工作线程
class Horse implements Runnable {
 ...
 //CyclicBarrier多个Horse及HorseRace共享，构造函数传入
 .
 public Horse(CyclicBarrier barrier) {
  cyclicBarrier = barrier;
 }
 //当前跑了多远，线程之间也要同步
 public synchronized in getStrides() { return strides; }
 public String tracks() {
   ...
  for (int i = 0; i < getStrides(); ++i) {
    ...
 }
 //线程
 public void run() {
  try {
   while(!Thread.interrupted()) {c
    synchronized(this) {
     strides += rand.nextInt(3); //0,1 or 2
    }
    //通知barrier本worker的在这一轮作业中的任务已经完成
    //阻塞等待barrier进入下一轮
    barrier.await();
   }
  } catch (InterruptedException e) {
   ...
  } catch (BrokenBarrierException e) {
   throw new RuntimeException(e);
  }
 }
 //
}
```

比赛（栅栏）

```
public class HorseRace {
  static final int FINISH_LINE = 75;
  private List<Horse> horses = new ArrayList<Horse>();
  private ExecutorService exec =
    Executors.newCachedThreadPool();
  private CyclicBarrier barrier;
  public HorseRace(int nHorses, final int pause) {
    barrier = new CyclicBarrier(nHorses, new Runnable() {
      public void run() {
        StringBuilder s = new StringBuilder();
        for(int i = 0; i < FINISH_LINE; i++)
          s.append("="); // The fence on the racetrack
        print(s);
        for(Horse horse : horses)
          print(horse.tracks());
        for(Horse horse : horses)
          if(horse.getStrides() >= FINISH_LINE) {
            print(horse + "won!");
            exec.shutdownNow();
            return;
          }
        try {
          TimeUnit.MILLISECONDS.sleep(pause);
        } catch(InterruptedException e) {
          print("barrier-action sleep interrupted");
        }
      }
    });
    for(int i = 0; i < nHorses; i++) {
      Horse horse = new Horse(barrier);
      horses.add(horse);
      exec.execute(horse);
    }
  }
```

```
//比赛(栅栏)
public class HorseRace {
 ...
 //工作线程
 private List<Horse> horse = new ArrayList<House>();
 private ExecutorService exec = Executors.newCachedThreadPool();
 //栅栏
 private CyclicBarrier cyclicBarrier;
 //比赛封装在构造函数中
 pbulic HorseRace(int horseNum, final int pause) {
  //创建CyclicBarrier，horseNum个worker
  //new Runnable()由每一轮最后一个进入barrier的线程执行
  cyclicBarrier = new CyclicBarrier(horseNum, new Runnable(){
   public void run() {
    ...
    for (Horse horse : horses)
     print(horse.tracks()); //打印horse跑的距离(互斥)
    for (Horse horse : horses)
     if (horse.getStrides() >= FINISH_LINE) {
      exec.shutdownNow(); //用中断结束工作线程(horse)
      return; //此轮barrier作业结束
     }
    ...
    //sleep for ${pause} seconds
   }
  });g
  //启动工作线程
  for (int i = 0; i < nHorses; ++i) {
   Horse horse = new Horse(barrier);
   horses.add(horse);
   exec.execute(horse);
  };
```

			//即使main执行完毕，其他线程也不会立即被回收(要等到线程结束) //CyclicBarrier被各个线程引用，也不会立即被回收 }
	main	public static void main(String[] args) { int nHorses = 7; int pause = 200; if(args.length > 0) { // Optional argument int n = new Integer(args[0]); nHorses = n > 0 ? n : nHorses; } if(args.length > 1) { // Optional argument int p = new Integer(args[1]); pause = p > -1 ? p : pause; } new HorseRace(nHorses, pause); } } /* (Execute to see output) *///:~	main() parse参数，new一个HorseRace出来

4.DelayQueue【目录】

<table>
<tr><td>用途</td><td>存放元素</td><td>实现了Delayed接口的对象</td><td>需要实现getDelayed()函数，表示还需要多少纳秒，Delayed Task才可以执行</td></tr>
<tr><td></td><td></td><td>Delayed接口继承Comparable<Delayed></td><td>需要实现compareTo(Delayed other)函数，执行时间比other早时返回-1,晚返回1</td></tr>
<tr><td></td><td>存取</td><td>存入数量</td><td>是无界的BlockingQueue，存放元素数量没有限制，不会阻塞</td></tr>
<tr><td></td><td></td><td>取元素</td><td>有Delayed对象到期可以执行时
取出Delayed对象
无Delayed对象到期时
take()
阻塞
poll(timeOut，TimeUnit)
阻塞
poll()
返回null</td></tr>
<tr><td></td><td>顺序</td><td colspan="2">delay最小的最先取出
这种特性也是DelayedQueue成为优先级队列的一种变体（与优先级队列不同的是，融入了基于超时的时间控制）</td></tr>
</table>

例子

DelayedTask

```
//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
  private static int counter = 0;
  private final int id = counter++;
  private final int delta;
  private final long trigger;
  protected static List<DelayedTask> sequence =
    new ArrayList<DelayedTask>();
  public DelayedTask(int delayInMilliseconds) {
    delta = delayInMilliseconds;
    trigger = System.nanoTime() +
      NANOSECONDS.convert(delta, MILLISECONDS);
    sequence.add(this);
  }
  public long getDelay(TimeUnit unit) {
    return unit.convert(
      trigger - System.nanoTime(), NANOSECONDS);
  }
  public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
  }
  public void run() { printnb(this + " "); }
  public String toString() {
    return String.format("[%1$-4d]", delta) +
      " Task " + id;
  }
  public String summary() {
    return "(" + id + ":" + delta + ")";
  }
  public static class EndSentinel extends DelayedTask {
    private ExecutorService exec;
    public EndSentinel(int delay, ExecutorService e) {
      super(delay);
      exec = e;
    }
    public void run() {
      for(DelayedTask pt : sequence) {
        printnb(pt.summary() + " ");
      }
      print();
      print(this + " Calling shutdownNow()");
      exec.shutdownNow();
    }
  }
}
```

```
//向DelayedQueue中放置的Task元素，实现Runnable接口是因为要run，实现Delayed接口是满足DelayedQueue对元素的要求
class DelayedTask implements Runnable, Delayed {
  ...
  private final int delta_ms; //任务创建后,延迟多少ms触发任务执行
  private final long trigger_tm_us; //任务执行触发时间
  //task_list被所有DelayedTask共享，遍历打印Task时用
  protected static List<DelayedTask> task_list
                    = new ArrayList<DelayedTask>;
  //构造函数: 设置延迟时间,任务执行时间,追加到task_list
  public DelayedTask(int delayMilliSec) {
    delta = delayMilliSec;
    trigger_tm_us
      = System.nanoTime() + NANOSECONDS.convert(delta,
MILLISEDONDS);
    task_list.add(this);
  }
  //接口要求提供，距离函数调用，还需要多长时间Task才可以执行
  public long getDelay(TimeUnit unit) {
    return unit.convert(trigger - System.nanoTime(), NANOSECONDS);
  }
  //比较函数：接口要求提供
  public int compareTo(Delayed arg) P{
    //函数内部再转型回DelayedTask
    DelayedTask that = (DelayedTask)arg;
    if (trigger < that.trigger) return -1; //trigger小的先出队
    if (trigger > that.trigger) return 1;
    return 0;
  }
  //DelayedTask从DelayedQueue中取出并运行时执行的函数
  public void run() { printnb(this + " "); }
  ...
  //哨兵节点，用来标记队列中最后一个任务的
  //执行到这个Task时会引发特殊操作，例如统计报表，结束工作线程之类
  public static class EndSentinel extends DelayedTask {
    private ExecutorService execSvc;
    public EndSentinel(int delay, ExecutorService exec) {
      //将delay设置的比所有任务都长，才能保证在最后执行
      //但是DelayQueue#take()函数会阻塞，任务到期时才能取出
      super(delay);
      exec = e;
    }
    public void run() {
      for(DelayedTask task : taskList) {
        printnb(task.summary() + " ");
      }
      execSvc.shutdownNow(); //中断消费者线程
    }
  }
}
```

DelayedTaskConsumer

```
class DelayedTaskConsumer implements Runnable {
  private DelayQueue<DelayedTask> q;
  public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
    this.q = q;
```

```
//消费者线程
class DelayedTaskConsumer implements Runnable {
  //DelayQueue<DelayedTask>从构造函数参数传入，与生产者共享
```

		} public void run() { try { while(!Thread.interrupted()) q.take().run(); // Run task with the current thread } catch(InterruptedException e) { // Acceptable way to exit } print("Finished DelayedTaskConsumer"); } }	private DelayQueue<DelayedTask> delayQueue; public DelayedTaskConsumer(DelayQueue<DelayedTask> q) { delayQueue = q; } //线程操作 public void run() { try { while(!Thread.interrupted()) { //没阻塞时收到中断信号 delayQueue.take().run(); //阻塞，等到队列中有一个对象到期时取出 } } catch(InterruptedException e) { //阻塞时收到中断信号 ... } } }
	main	public class DelayQueueDemo { public static void main(String[] args) { Random rand = new Random(47); ExecutorService exec = Executors.newCachedThreadPool(); DelayQueue<DelayedTask> queue = new DelayQueue<DelayedTask>(); // Fill with tasks that have random delays: for(int i = 0; i < 20; i++) queue.put(new DelayedTask(rand.nextInt(5000))); // Set the stopping point queue.add(new DelayedTask.EndSentinel(5000, exec)); exec.execute(new DelayedTaskConsumer(queue)); } } /* Output: [128] Task 11 [200] Task 7 [429] Task 5 [520] Task 18 [555] Task 1 [961] Task 4 [998] Task 16 [1207] Task 9 [1693] Task 2 [1809] Task 14 [1861] Task 3 [2278] Task 15 [3288] Task 10 [3551] Task 12 [4258] Task 0 [4258] Task 19 [4522] Task 8 [4589] Task 13 [4861] Task 17 [4868] Task 6 (0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868) (7:200) (8:4522) (9:1207) (10:3288) (11:128) (12:3551) (13:4589) (14:1809) (15:2278) (16:998) (17:4861) (18:520) (19:4258) (20:5000) [5000] Task 20 Calling shutdownNow() Finished DelayedTaskConsumer *///:~	//线程管理 ExecutorService execSvc = Executors.newCachedThreadPool(); //DelayedQueue DelayQueue<DelayedTask> delayQueue = new DelayQueue<DelayedTask>(); for (int i = 0; i < 20; ++i) { //DelayQueuy#put(DelayedTask)：因为队列无界所以不会阻塞 //rand.nextInt(5000)：返回数字大于0小于5000，表示任务要延迟多久执行 delayQueue.put(new DelayedTask(rand.nextInt(5000)); } //DelayQueue#add(DelayedTask)：添加元素到DelayQueue中，函数无特别说明 //EndSentinel的延迟时间是5000ms，比所有其他DelayedTask都晚，最后执行 delayQueue.add(new DelayedTask.EndSentinel(5000, execSvc)); //启动消费者线程 execSvc.execute(new DelayecdTaskConsumer(delayQueue));

5.PriorityBlockingQueue【目录】

功能	优先级阻塞队列，PrioritizedTask被赋予一个优先级数字，优先级高的先出队		
例子	PrioritizedTask	//: concurrency/PriorityBlockingQueueDemo.java import java.util.concurrent.*; import java.util.*; import static net.mindview.util.Print.*; class PrioritizedTask implements Runnable, Comparable<PrioritizedTask> { private Random rand = new Random(47); private static int counter = 0; private final int id = counter++; private final int priority; protected static List<PrioritizedTask> sequence = new ArrayList<PrioritizedTask>(); public PrioritizedTask(int priority) { this.priority = priority; sequence.add(this); } public int compareTo(PrioritizedTask arg) { return priority < arg.priority ? 1 : (priority > arg.priority ? -1 : 0); } public void run() { try { TimeUnit.MILLISECONDS.sleep(rand.nextInt(250)); } catch(InterruptedException e) { // Acceptable way to exit } print(this); } public String toString() { return String.format("[%1$-3d]", priority) + " Task " + id; } public String summary() { return "(" + id + ":" + priority + ")"; } public static class EndSentinel extends PrioritizedTask { private ExecutorService exec; public EndSentinel(ExecutorService e) { super(-1); // Lowest priority in this program exec = e; } public void run() { int count = 0; for(PrioritizedTask pt : sequence) { printnb(pt.summary()); if(++count % 5 == 0) print(); } print(); print(this + " Calling shutdownNow()"); exec.shutdownNow(); } } }	class PrioritizedTask implements Runnable, Comparable<PrioritizedTask> { //所有PrioritizedTask共享这个队列，保存所有Task的引用 protected static List<PrioritizedTask> taskList = new ArrayList<PrioritizedTask>(); //priority private final in priority; public PrioritizedTask(int priority) { this.priority = priority; taskList.add(this); } //Task比较 public int compareTo(PrioritizedTask other) { if (priority > other.priority) {return -1;} //优先级高的先出队 else if (priority < other.priority) {return 1;} else {return 0;} } //Task任务 public void run() { .. } ... //哨兵节点，优先级最低，最后执行，用来发中断信号结束线程运行 public static class EndSentinel extends PrioritizedTask { private ExecutorService exec; public EndSentinel(ExecutorService exec) { super(-1); //lowest priority } public void run() { //打印summary for (PrioritizedTask pt : taskList) { ... } //结束线程运行 execSvc.shutdownNow(); } } }
	生产者	class PrioritizedTaskProducer implements Runnable { private Random rand = new Random(47); private Queue<Runnable> queue; private ExecutorService exec; public PrioritizedTaskProducer(Queue<Runnable> q, ExecutorService e) { queue = q; exec = e; // Used for EndSentinel }	• 随机向PriorityQueue加入各种PriorityTask（优先级为[0,10）内的随机数 包括在运行中，sleep一段时间后，再插入新的PriorityTask • 最后再插入一个EndSentinel • priorityQueue.add(new PrioritzedTask(rand.nextInt(10));

		<pre>public void run() { // Unbounded queue; never blocks. // Fill it up fast with random priorities: for(int i = 0; i < 20; i++) { queue.add(new PrioritizedTask(rand.nextInt(10))); Thread.yield(); } // Trickle in highest-priority jobs: try { for(int i = 0; i < 10; i++) { TimeUnit.MILLISECONDS.sleep(250); queue.add(new PrioritizedTask(10)); } // Add jobs, lowest priority first: for(int i = 0; i < 10; i++) queue.add(new PrioritizedTask(i)); // A sentinel to stop all the tasks: queue.add(new PrioritizedTask.EndSentinel(exec)); } catch(InterruptedException e) { // Acceptable way to exit } print("Finished PrioritizedTaskProducer"); } }</pre>	
	消费者	<pre>class PrioritizedTaskConsumer implements Runnable { private PriorityBlockingQueue<Runnable> q; public PrioritizedTaskConsumer(PriorityBlockingQueue<Runnable> q) { this.q = q; } public void run() { try { while(!Thread.interrupted()) // Use current thread to run the task: q.take().run(); } catch(InterruptedException e) { // Acceptable way to exit } print("Finished PrioritizedTaskConsumer"); } }</pre>	• 消费者 不断从PriorityBlockingQueue中取出元素并运行 priorityQueue.take().run()
	main	<pre>public class PriorityBlockingQueueDemo { public static void main(String[] args) throws Exception { Random rand = new Random(47); ExecutorService exec = Executors.newCachedThreadPool(); PriorityBlockingQueue<Runnable> queue = new PriorityBlockingQueue<Runnable>(); exec.execute(new PrioritizedTaskProducer(queue, exec)); exec.execute(new PrioritizedTaskConsumer(queue)); } } /* (Execute to see output) *///:~</pre>	• main • 创建ExecutorService • PriorityBlockingQueue • 创建生产者线程 • 创建消费者线程

6. ScheduledExecutor 【目录】

功能	用来将Runnable任务设置为在某个时间点运行，或者按照固定的间隔周期运行
例子	用ScheduledExecutor实现了一个GreenHouseSchedular 1.各种任务通过实现Runnable接口来编写 2.创建一个ScheduledThreadPoolExecutor，内含10个Core Thread 3.向ScheduledThreadPoolExecutor注册各种任务，一次性的，周期执行的

Runnable	<pre>class LightOn implements Runnable { public void run() { // Put hardware control code here to // physically turn on the light. System.out.println("Turning on lights"); light = true; } } class LightOff implements Runnable { public void run() { // Put hardware control code here to // physically turn off the light. System.out.println("Turning off lights"); light = false; } } class WaterOn implements Runnable { public void run() { // Put hardware control code here. System.out.println("Turning greenhouse water on"); water = true; } } class WaterOff implements Runnable { public void run() { // Put hardware control code here. System.out.println("Turning greenhouse water off"); water = false; } } class ThermostatNight implements Runnable { public void run() { // Put hardware control code here. System.out.println("Thermostat to night setting"); setThermostat("Night"); } } class ThermostatDay implements Runnable { public void run() { // Put hardware control code here. System.out.println("Thermostat to day setting"); setThermostat("Day"); } } class Bell implements Runnable { public void run() { System.out.println("Bing!"); } } class Terminate implements Runnable { public void run() { System.out.println("Terminating");</pre>	//各种Runnable（实现为GreenHouseSchedular的内部类） //run函数都是设置标志位，硬件根据标志位来执行对应的功能 //被设置的标志位都是GreenHouseScheduler的成员变量 class LightOn implements Runnable { public void run() { light = true; } } class LightOff implements Runnable { public void run() { light = false; } } class WaterOn implements Runnable { public void run() { water = true; } } class WatterOff implements Runnable { public void run() { water = false; } } class ThermoistatNight implements Runnable { public void run() { setThermostat("Night"); } } class ThermostatDay implements Runnable { public void run() { setThermostat("Day"); } } class Bell implements Runnable { public void run() { ... } }

	`` scheduler.shutdownNow(); // Must start a separate task to do this job, // since the scheduler has been shut down: new Thread() { public void run() { for(DataPoint d : data) System.out.println(d); } }.start(); } } ```	``` //Terminate比较特殊 class Terminate implements Runnable { public void run() { //关闭Scheduler，应该会发送中断信号给coreThread scheduler.shutdownNow(); //启动一个线程，统计当前任务情况 new Thread() { public void run() { for(DataPoint d : data) System.out.println(d); } }.start(); } } ```
任务调度	``` import java.util.concurrent.*; import java.util.*; public class GreenhouseScheduler { private volatile boolean light = false; private volatile boolean water = false; private String thermostat = "Day"; public synchronized String getThermostat() { return thermostat; } public synchronized void setThermostat(String value) { thermostat = value; } ScheduledThreadPoolExecutor scheduler = new ScheduledThreadPoolExecutor(10); public void schedule(Runnable event, long delay) { scheduler.schedule(event,delay,TimeUnit.MILLISECONDS); } public void repeat(Runnable event, long initialDelay, long period) { scheduler.scheduleAtFixedRate(event, initialDelay, period, TimeUnit.MILLISECONDS); } ```	``` //任务调度 public class GreenhouseScheduler { //温室的各项标志位，上面的一批Runnable的run()函数操作这些标志位 //通过volatile让标志位对所有CPU可见 private volatile boolean light = false; private volatile boolean water = false; private String thermostat = "Day"; //只被synchronized函数操作 ... //ScheduledThreadPoolExecutor用一个无界推理和corePoolSizePool //个线程实现，这个例子中corePoolSizePool = 10 ScheduledThreadPoolExecutor scheduledThreadPoolExecutor = new SchedculedThreadPoolExecutor(10); //添加一次性的调度任务 public void schedule(Runnable event, long delay) { scheduledThreadPoolExecutor.schedule(event, delay, TimeUnit.MILLISECONDS); } //添加周期执行的调度任务 public void repeat(Runnable event, long initDelay, long period) { scheduledThreadPoolExecutor.schedule(event, initDelay, period, TimeUnit.MILLISECONDS); } ```
数据记录	``` // New feature: data collection static class DataPoint { final Calendar time; final float temperature; final float humidity; public DataPoint(Calendar d, float temp, float hum) { time = d; temperature = temp; humidity = hum; } public String toString() { return time.getTime() + String.format(" temperature: %1$.1f humidity: %2$.2f", temperature, humidity); } } ```	``` //数据记录（实现为GreenHouseSchedular的内部类） //是个辅助类，用来打印<时间,温度,湿度>这样的温室状况 static class DataPoint { ... } ```
域初始化	``` private Calendar lastTime = Calendar.getInstance(); { // Adjust date to the half hour lastTime.set(Calendar.MINUTE, 30); lastTime.set(Calendar.SECOND, 00); } private float lastTemp = 65.0f; private int tempDirection = +1; private float lastHumidity = 50.0f; private int humidityDirection = +1; private Random rand = new Random(47); List<DataPoint> data = Collections.synchronizedList(new ArrayList<DataPoint>()); ```	``` //DataPoint放在sychronizedList中，不用担心同步问题 ```
数据记录线程	``` class CollectData implements Runnable { public void run() { System.out.println("Collecting data"); synchronized(GreenhouseScheduler.this) { // Pretend the interval is longer than it is: lastTime.set(Calendar.MINUTE, lastTime.get(Calendar.MINUTE) + 30); // One in 5 chances of reversing the direction: if(rand.nextInt(5) == 4) tempDirection = -tempDirection; // Store previous value: lastTemp = lastTemp + tempDirection * (1.0f + rand.nextFloat()); if(rand.nextInt(5) == 4) humidityDirection = -humidityDirection; lastHumidity = lastHumidity + humidityDirection * rand.nextFloat(); // Calendar must be cloned, otherwise all // DataPoints hold references to the same lastTime. // For a basic object like Calendar, clone() is OK. data.add(new DataPoint((Calendar)lastTime.clone(), lastTemp, lastHumidity)); } } } ```	数据记录线程： 单独的线程，记录各项指标的变化记录
main	``` public static void main(String[] args) { GreenhouseScheduler gh = new GreenhouseScheduler(); gh.schedule(gh.new Terminate(), 5000); // Former "Restart" class not necessary: gh.repeat(gh.new Bell(), 0, 1000); gh.repeat(gh.new ThermostatNight(), 0, 2000); gh.repeat(gh.new LightOn(), 0, 200); gh.repeat(gh.new LightOff(), 0, 400); gh.repeat(gh.new WaterOn(), 0, 600); gh.repeat(gh.new WaterOff(), 0, 800); gh.repeat(gh.new ThermostatDay(), 0, 1400); gh.repeat(gh.new CollectData(), 500, 500); } } /* (Execute to see output) *///:~ ```	``` //设置各种任务调度 //5000ms后退出 gh.schedule(gh.new Terminate(), 5000); //其他的时周期执行的任务 gh.repeat(gh.new Bell(), 0, 1000); ... gh.repeat(gh.new CollectData(), 500, 500); ```

7. Semaphore 【目录】

功能	允许N个任务同时访问一个资源（与concurrent.lock，内建的synchronized锁不同）	
例子	用Semaphore来同步对象池	
	对象池	
	胖对象	
	CheckOut线程	

对象池

```
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
  private int size;
  private List<T> items = new ArrayList<T>();
  private volatile boolean[] checkedOut;
  private Semaphore available;
  public Pool(Class<T> classObject, int size) {
    this.size = size;
    checkedOut = new boolean[size];
    available = new Semaphore(size, true);
    // Load pool with objects that can be checked out:
    for(int i = 0; i < size; ++i)
      try {
        // Assumes a default constructor:
        items.add(classObject.newInstance());
      } catch(Exception e) {
        throw new RuntimeException(e);
      }
  }
  public T checkOut() throws InterruptedException {
    available.acquire();
    return getItem();
  }
  public void checkIn(T x) {
    if(releaseItem(x))
      available.release();
  }
  private synchronized T getItem() {
    for(int i = 0; i < size; ++i)
      if(!checkedOut[i]) {
        checkedOut[i] = true;
        return items.get(i);
      }
    return null; // Semaphore prevents reaching here
  }
  private synchronized boolean releaseItem(T item) {
    int index = items.indexOf(item);
    if(index == -1) return false; // Not in the list
    if(checkedOut[index]) {
      checkedOut[index] = false;
      return true;
    }
    return false; // Wasn't checked out
  }
} ///:~
```

```
public class Pool<T> {
 // 资源
 private int size;
 private List<T> itemList = new ArrayList<T>();
 // 指定资源是否已经被某个线程占用
 private volatile boolean[] checkedOut;
 // 取资源时控制阻塞/等待的信号量
 private Semaphore semaphore;
 // 初始化成员变量
 public Pool(Class<T> class, int size) {
  this.size = size;
  checkedOut = new boolean[size];
  semaphore = new Semaphore(size, /*fair=*/true);
  for (int i = 0; i < size; ++i)
   try {
    itemList.add(class.newInstance())l
   } catch(Exception e) {
    throw new RuntimeException(e);
   }
  }
 }
 // 取资源
 public T checkOut() throws InterruptedException {
  //对信号量做P操作，资源全部被占用时，会阻塞
  semaphore.acquire();
  //getItem是synchronized函数
  return getItem();
 }
 public synchronized T getItem() {
  for (int i = 0; i < size; ++i) {
   if (!checkedOut[i]) {
    checkedOut[i] = true; //第一个没checkedOut得资源
    return itemList.get(i);
   }
  }
  return null; //走不到这一行，因为有semaphore保护
 }
 // 归还资源
 public void checkIn(T x) {
  // releaseItem时synchronized函数
  if (releaseItem(x)) {
   //资源释放完成后，再对信号量做V操作
   semaphore.release();
  }
 }
 private synchronized boolean releaseItem(T item) {
  int index = itemList.indexOf(item);
  if (index == -1) { return false; }
  if (checkedOut[index]) {
   checkedOut[index] = false;
   return true;
  }
  return false;
 }
}
```

胖对象

```
public class Fat {
  private volatile double d; // Prevent optimization
  private static int counter = 0;
  private final int id = counter++;
  public Fat() {
    // Expensive, interruptible operation:
    for(int i = 1; i < 10000; i++) {
      d += (Math.PI + Math.E) / (double)i;
    }
  }
  public void operation() { System.out.println(this); }
  public String toString() { return "Fat id: " + id; }
} ///:~
```

Fat对象，构造成本非常大，因此需要用对象池来缓存

CheckOut线程

```
  //: concurrency/SemaphoreDemo.java
  // Testing the Pool class
  import java.util.concurrent.*;
  import java.util.*;
  import static net.mindview.util.Print.*;
  // A task to check a resource out of a pool:
  class CheckoutTask<T> implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private Pool<T> pool;
    public CheckoutTask(Pool<T> pool) {
      this.pool = pool;
    }
    public void run() {
      try {
        T item = pool.checkOut();
        print(this + "checked out " + item);
        TimeUnit.SECONDS.sleep(1);
        print(this +"checking in " + item);
        pool.checkIn(item);
```

```
class CheackOutRunnable<T> implemenbts Runnable {
 ...
 //对象池多线程共享，由构造函数传入
 private Pool<T> pool;
 public CheackOutRunnable(Pool<T> pool) {
  this.pool = pool;
 }

 public void run() {
  try {
   //从对象池取对象，资源都被占用时会阻塞
   T item = pool.checkOut();
   TimeUnit.SECONDS.sleep(1);
   //把对象归还给对象池
   pool.checkIn(item);
  } catch (InterruptedException e) {
   ...
```

		`` } catch(InterruptedException e) { // Acceptable way to terminate } } public String toString() { return "CheckoutTask " + id + " "; } } ```	} } ... }
	main	``` public class SemaphoreDemo { final static int SIZE = 25; public static void main(String[] args) throws Exception { final Pool<Fat> pool = new Pool<Fat>(Fat.class, SIZE); ExecutorService exec = Executors.newCachedThreadPool(); for(int i = 0; i < SIZE; i++) exec.execute(new CheckoutTask<Fat>(pool)); print("All CheckoutTasks created"); List<Fat> list = new ArrayList<Fat>(); for(int i = 0; i < SIZE; i++) { Fat f = pool.checkOut(); printnb(i + ": main() thread checked out "); f.operation(); list.add(f); } Future<?> blocked = exec.submit(new Runnable() { public void run() { try { // Semaphore prevents additional checkout, // so call is blocked: pool.checkOut(); } catch(InterruptedException e) { print("checkOut() Interrupted"); } } }); TimeUnit.SECONDS.sleep(2); blocked.cancel(true); // Break out of blocked call print("Checking in objects in " + list); for(Fat f : list) pool.checkIn(f); for(Fat f : list) pool.checkIn(f); // Second checkIn ignored exec.shutdown(); } } /* (Execute to see output) *///:~ ```	//创建对象池 final Pool<Fat> pool = new Pool<Fat>(Fat.class, SIZE); //启动线程 ExecutorService execSvc = Executors.newCachedThreadPool(); for (int i = 0; i < SIZE; ++i) { execSvc.execute(new CheckoutRunnable<Fat>(pool); } //把对象池抽空 List<Fat> list = new ArrauyList<Fat>(); for (int i = 0; i < SIZE; ++i) { Fat fat = pool.checkOut(); fat.operator(); list.add(fat); } //启动一个线程，尝试从已被抽空的对象池中取出Fat对象 Futrue<?> future = execSvc.submit(new Runnable() { public void run() { try { pool.checkOut(); } catch (InterruptedException e) { ... } }); //阻塞了两秒钟，仍然没有取出Fat对象，线程仍然阻塞 TimeUnit.SECONDS.sleep(2); //中断取资源线程，取资源线程正阻塞在semaphore.acquire()上，这个操作也可以被中断 future.cancel(true); //归还资源 for (Fat fat : list) pool.checkIn(fat); //重复的归还没有效果 for (Fat fat : list) pool.checkIn(fat); //关闭ExecutorService，不再接受新的线程提交 execSvc.shutdown();

8. Exchanger【目录】

用途	使用过程	任务A拥有Obj M，任务B拥有Obj N 其中一个调用exchanger.exchange()阻塞，直到另一个任务也调用exchange.change()，两个任务在Exchanger中交换对象 任务A拥有Obj N，任务A拥有Obj M	
	场景	对象生产代价很高，一个任务生产对象，一个任务消费对象	
例子	生产者	``` import java.util.concurrent.*; import java.util.*; import net.mindview.util.*; class ExchangerProducer<T> implements Runnable { private Generator<T> generator; private Exchanger<List<T>> exchanger; private List<T> holder; ExchangerProducer(Exchanger<List<T>> exchg, Generator<T> gen, List<T> holder) { exchanger = exchg; generator = gen; this.holder = holder; } public void run() { try { while(!Thread.interrupted()) { for(int i = 0; i < ExchangerDemo.size; i++) holder.add(generator.next()); // Exchange full for empty: holder = exchanger.exchange(holder); } } catch(InterruptedException e) { // OK to terminate this way. } } } ```	class ExechangerProducer<T> implements Runnable { //Generator，Exchanger以及存储产品的itemList， //都从构造函数传入，生产者和消费者共享 private Generator<T> generator; private Exchanger<List<T>> exchanger; //与消费者共享 private List<T> itemList; //与消费者交换 ExchangerProducer(...) {...} //生产者线程 public void run() { try { while (!Thread.interrupted()) { //生产size个产品 for (int i = 0; i < globalConf.size; ++) { itemList.add(generator.next()); } //提交给消费者线程，阻塞等待消费者线程来取 //参数是: myItemList；返回值是：peerItemList itemList = exchanger.exchange(itemList); } } catch (InterruptedException e) { //通过中断来结束线程运行 ... } } }
	消费者	``` class ExchangerConsumer<T> implements Runnable { private Exchanger<List<T>> exchanger; private List<T> holder; private volatile T value; ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder){ exchanger = ex; this.holder = holder; } public void run() { try { while(!Thread.interrupted()) { ```	class ExchangerConsumer<T> implements Runnable { //临时变量 private volatile T tmpTVal; //Exchange,存储产品的itemList都从构造函数传入 private List<T> itemList; private Exchanger<List<T>> exchanger; ExchangerConsumer(...) {...} //消费者线程 public void run() {

		holder = exchanger.exchange(holder); for(T x : holder) { value = x; // Fetch out value holder.remove(x); // OK for CopyOnWriteArrayList } } } catch(InterruptedException e) { // OK to terminate this way. } System.out.println("Final value: " + value); } }	try { while (!Thread.interrupted()) { //与生产者交换List<Item> //参数是: myItemList；返回值是： peerItemList itemList = exchanger.exchange(itemList); //消费itemList中的Item for (T t : itemList) { tmpTVal = t; itemList.remove(tmpTVal); } } } catch (InterruptedException e) { ... } } }
	main	public class ExchangerDemo { static int size = 10; static int delay = 5; // Seconds public static void main(String[] args) throws Exception { if(args.length > 0) size = new Integer(args[0]); if(args.length > 1) delay = new Integer(args[1]); ExecutorService exec = Executors.newCachedThreadPool(); Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>(); List<Fat> producerList = new CopyOnWriteArrayList<Fat>(), consumerList = new CopyOnWriteArrayList<Fat>(); exec.execute(new ExchangerProducer<Fat>(xc, BasicGenerator.create(Fat.class), producerList)); exec.execute(new ExchangerConsumer<Fat>(xc,consumerList)); TimeUnit.SECONDS.sleep(delay); exec.shutdownNow(); } } /* Output: (Sample) Final value: Fat id: 29999 *///:~	//生产者、消费者所用的Item存放容器 //CopyOnWriteArrayList参考下一节《CopyOnWriteArrayList》 List<Fat> producerList = new CopyOnWriteArrayList<Fat>(); consumerList = new CopyOnWriteArrayList<Fat>(); //Exchanger Exchanger<List<Fat>> exchanger = new Exchanger<List<Fat>>(); //生产者消费者线程 ExecutorService exec = Executors.newCachedThreadPool(); exec.execute(new ExchangerProducer<Fat>(exchanger, BasicGenerator.create(Fat.class), produerList)); exec.execute(new ExchangerConsumer<Fat>(exchanger,consumerList));