

JC2002 Java 程序设计

第 10 天：高级并发（CS）

11 月 14 日星期二

JC2002 Java 程序设计

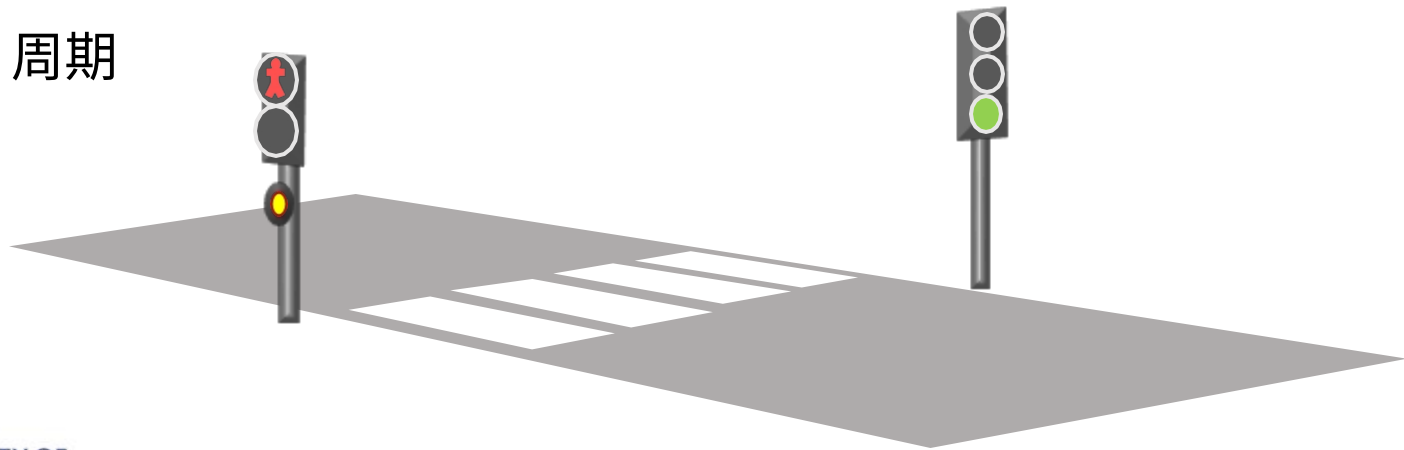
第 10 天，第 1 课时：定时事件和同步化

参考文献和学习目标

- 今天的会议主要基于
 - Deitel, H., 《**Java 如何编程**》，**早期对象**，第 23 章，2018 年
 - <https://docs.oracle.com/javase/tutorial/uiswing>
- 今天的课程结束后，您应该能够
 - 使用多线程执行定时事件
 - 在多线程应用程序中实施避免线程干扰和死锁的基本技术

定时事件的多线程处理

- 线程可用于执行定时事件
- 例如：交通信号灯，行人按下按钮即可请求绿灯并启动信号灯周期



例如：交通灯

- 我们可以将行人和交通信号灯周期作为两个线程来实施

交通灯线

灯光

线程睡

行人灯线

变了

眠

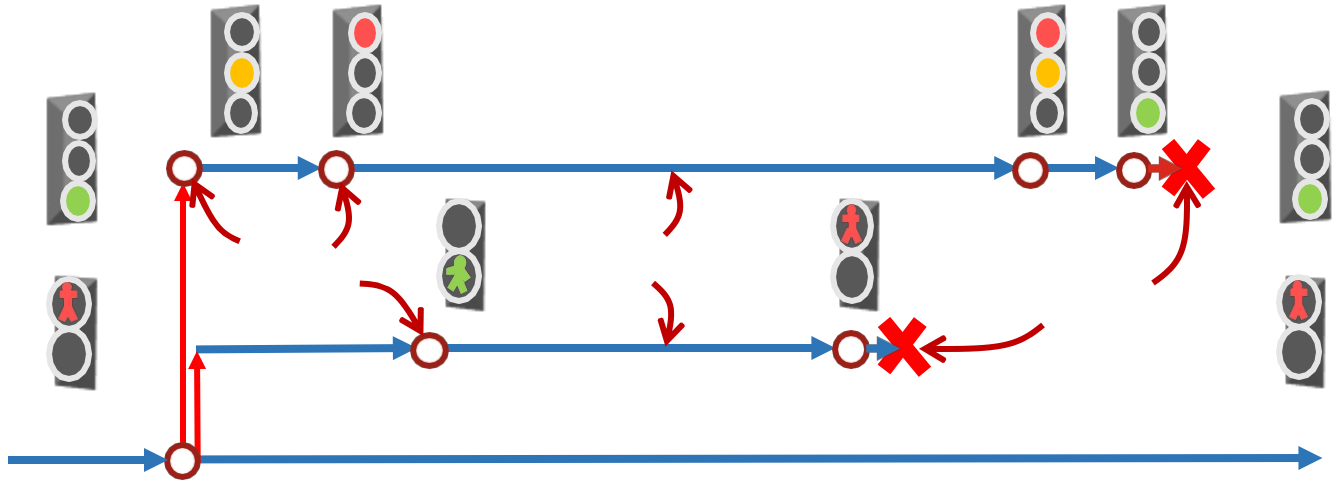
螺纹末

主线

端

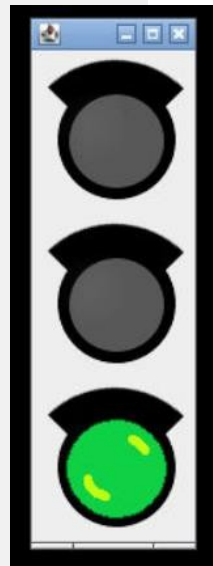
按钮

控



示例：定义汽车灯 (1)

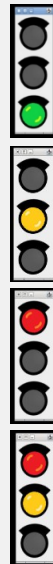
```
5  类 TrafficLights 扩展 JPanel {
6
7      JLabel topLight、middleLight、bottomLight;
8      ImageIcon off, red, yellow,
9      green;
10     TrafficLights() {
11         off = new ImageIcon("traffic_off.png");
12         red = new ImageIcon("traffic_red.png");
13         yellow = new ImageIcon("traffic_yellow.png");
14         green = new ImageIcon("traffic_green.png");
15         topLight = new JLabel();
16         topLight.setIcon(off);
17         middleLight = new JLabel();
18         middleLight.setIcon(off);
19         bottomLight = new JLabel();
20         bottomLight.setIcon(green);
```



```
21      GridLayout gridlayout = new GridLayout(3,1);
22      setLayout(gridlayout);
23      add(topLight);
24      add(middleLight);
25      add(bottomLight);
26  }
```

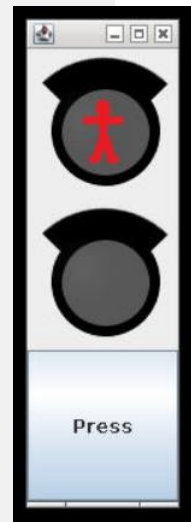

例如：定义汽车灯 (2)

```
27 public void setGreen() {
28     topLight.setIcon(off);
29     middleLight.setIcon(off);
30     bottomLight.setIcon(green);
31 }
32 公共 void setYellow() {
33     topLight.setIcon(off);
34     middleLight.setIcon(yellow);
35     bottomLight.setIcon(off);
36 }
37 public void setRed() {
38     topLight.setIcon(red);
39     middleLight.setIcon(off);
40     bottomLight.setIcon(off);
41 }
42 公共 void setRedAndYellow() {
43     topLight.setIcon(red);
44     middleLight.setIcon(yellow);
45     bottomLight.setIcon(off);
46 }
47 }
```



例如：定义行人灯 (1)

```
49 类 PedestrianLights 扩展 JPanel
50      实现 ActionListener {
51      JLabel topLight, bottomLight;
52      JButton 按钮;
53      关闭 ImageIcon, 等待, 开始;
54      字符串状态;
55      交通灯 trafficLights;
56      PedestrianLights(TrafficLights tl) {
57          交通灯 = tl;
58          status = new String("wait");
59          off = new ImageIcon("traffic_off.png");
60          wait = new ImageIcon("traffic_wait.png");
61          go = new ImageIcon("traffic_go.png");
62          topLight = new JLabel();
63          topLight.setIcon(wait);
```



```
64     bottomLight = new JLabel();
65     bottomLight.setIcon(off);
66     button = new JButton("Press");
67     GridLayout gridlayout = new GridLayout(3,1);
68     setLayout(gridlayout);
69     add(topLight);
70     add(bottomLight);
71     add(button);
```

例如：为行人设置照明灯 (2)

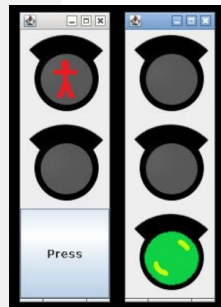
```
72         button.addActionListener(this);
73     }
74     public void setGo() {
75         topLight.setIcon(off);
76         bottomLight.setIcon(go);
77         status = new String("go")
78     ;
79 }

public void setWait() {
    topLight.setIcon(wait);
    bottomLight.setIcon(off);
    status = new String("wait")
;
}
```



示例：创建并显示图形用户界面

```
129 公共类 TrafficLightExample {
130      私人静态 void createAndShowGUI() {
131          JComponent trafficLights = new TrafficLights();
132          trafficLights.setOpaque(true);
133          JComponent pedestrianLights = new PedestrianLights((TrafficLights)trafficLights);
134          pedestrianLights.setOpaque(true);
135          JFrame plFrame = new JFrame("Pedestrian Lights");
136          plFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
137          JFrame tlFrame = new JFrame("Traffic Lights");
138          tlFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
139          plFrame.add(pedestrianLights);
140          plFrame.pack();
141          plFrame.setLocation(50,50);
142          plFrame.setVisible(true);
143          tlFrame.add(trafficLights);
144          tlFrame.pack();
145          tlFrame.setLocation(300,50);
149      public static void main(String[] args) {
```



```
150     javax.swing.SwingUtilities.invokeLater(new Runnable() {
151         public void run() {
152             createAndShowGUI();
153         }
154     });
155 }
156 }
```

例如：汽车运行周期

```
49 类 PedestrianLights 扩展 JPanel
50
...
84 实现 ActionListener {
85
...
86 public void startCycle() {
87     Thread trafficThread = new Thread(new Runnable() {
88         @Override
89         public void run()
90         {
91             try {
92                 trafficLights.setYellow();
93                 Thread.sleep(2000);
94                 trafficLights.setRed();
95                 Thread.sleep(5000);
96                 trafficLights.setRedAndYellow();
97                 Thread.sleep(1000);
98                 trafficLights.setGreen();
99                 button.setEnabled(true);
100             }
101             catch (InterruptedException e) {
102                 e.printStackTrace();
103             }
104         }
105     }
106 }
```



```
}  
)  
;
```


例如：为行人提供运行周期

```
104 Thread pedestrianThread = new Thread(new Runnable() {
105     @Override
106     public void run()
107     {
108         try {
109             button.setEnabled(false);
110             Thread.sleep(3000);
111             setGo();
112             Thread.sleep(3000);
113             setWait();
114         }
115         catch (InterruptedException e) {
116             e.printStackTrace();
117         }
118     }
119 }
120 });
121 trafficThread.start();
122 pedestrianThread.start();
```

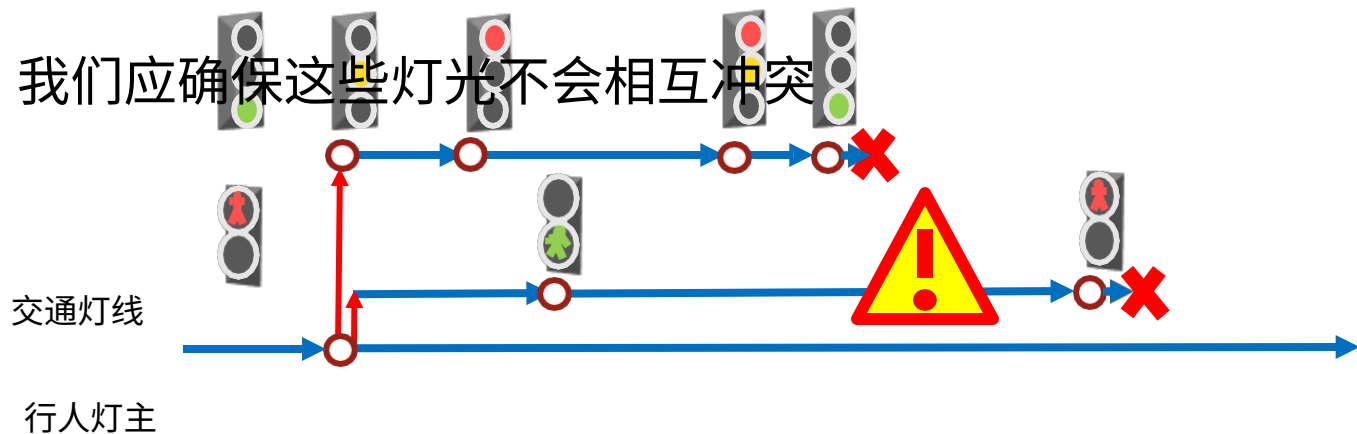
禁用按钮是为了避免新周期在旧周期停止前启动



```
124 public void actionPerformed(ActionEvent e) {
125     startCycle();
126 }
127 }
```

交通灯同步示例

- 当交通信号灯线程独立运行且计时错误时，汽车的绿灯有可能过早亮起
- 我们应确保这些灯光不会相互冲突



按钮按下

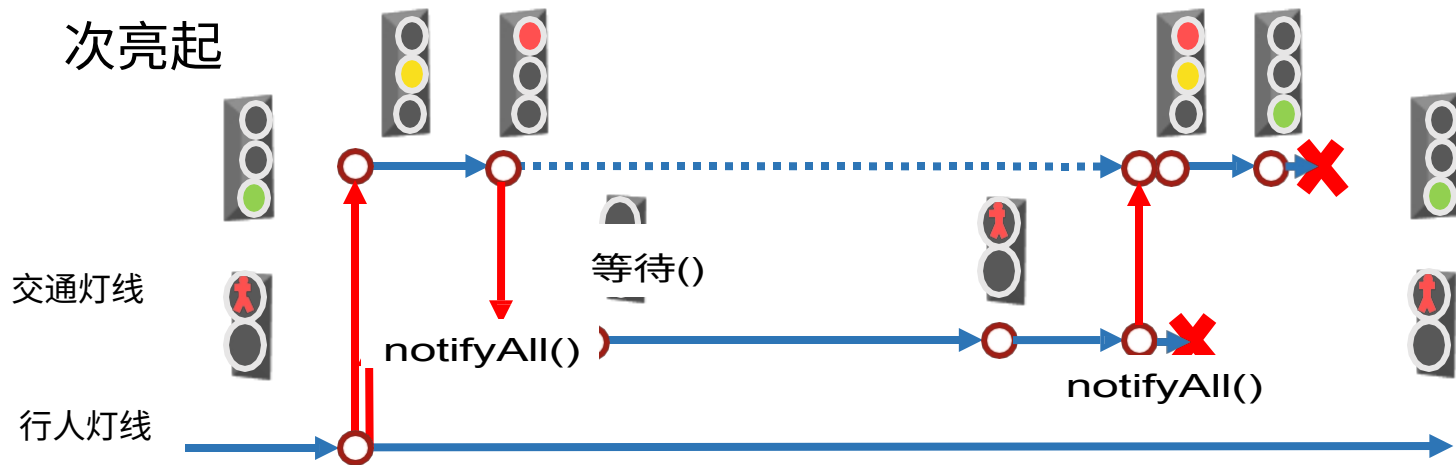
使用 `wait()` 和 `notify()`

- 通过 **`Object.wait()`** 和 **`Object.notifyAll()`** 可以控制从线程访问对象类或对象子类的任何对象（基本上是所有对象）
 - - 方法 `wait()` 会暂停当前线程的运行，直到另一个线程为同一对象发出通知，它才可以继续运行
 - 方法 `notifyAll()` 会向所有其他线程发送通知以便他们能够恢复
 - 方法 `notify()` 与 `notifyAll()` 类似，但只通知随机选择

的一个线程

例如：同步交通信号灯

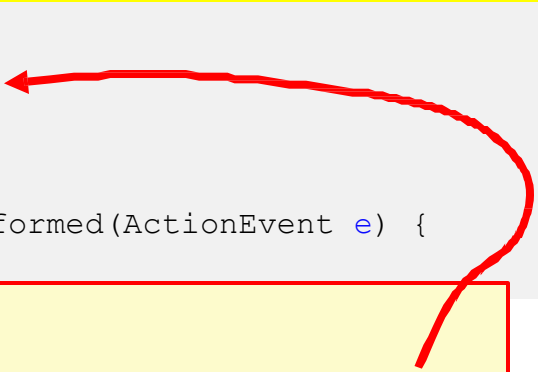
- 我们可以同步线程，使汽车的红绿灯始终等待行人的红灯再次亮起



按钮按下

同步交通信号灯 (1)

```
...  
88     ...  
89     public void startCycle() {  
90         TrafficLightThread tlCycle = new TrafficLightThread(this, tl);  
91         PedestrianLightThread plCycle = new PedestrianLightThread(this);  
92         tlCycle.start();  
93         plCycle.start();  
94     }  
95  
96     public void actionPerformed(ActionEvent e) {  
97         disableButton();  
98         startCycle();  
99     }
```



等待状态下启动

请注意，线程是同步的，因此 plCycle 在

我们为交通灯和行人灯定

义了 Thread 的自定义子类

同步交通信号灯 (2)

```
101 类 TrafficLightThread 扩展线程 {
102     私人 PedestrianLights pl;
103     私人交通灯 tl;
104     public TrafficLightThread(PedestrianLights pl, TrafficLights tl) {
105         this.pl = pl; this.tl = tl;
106     }
107     @Override
108     public void run() {
109         同步 (pl) {
110             try {
111                 tl.setYellow();
112                 Thread.sleep(2000);
113                 tl.setRed();
114                 pl.notifyAll();
115                 pl.wait();
116             }
117         }
118         tl.setRedAndYellow(); Thread.sleep(1000);
119     }
120 }
```

方法 notifyAll() 和 wait()
必须位于同步块内，以避免
出现异常

```
t  
l  
.n()  
s  
e  
t  
G  
r  
e  
e  
n  
(  
)  
;  
p  
l  
.e  
n  
a  
b  
l  
e  
B  
u  
t  
t  
o
```

```
121         catch (InterruptedException e) {  
122             e.printStackTrace();  
123         }  
124     }  
125 }  
126 }
```

同步交通信号灯 (3)

```
128 类 PedestrianLightThread 扩展线程 {
129     私人 PedestrianLights pl;
130     public PedestrianLightThread(PedestrianLights pl) {
131         this.pl = pl;
132     }
133     @Override
134     public void run() {
135         同步 (pl) {
136             try {
137                 Thread.sleep(3000);
138                 pl.setGo();
139             } catch (InterruptedException e) {
140                 Thread.currentThread().interrupt();
141             }
142         }
143     }
144 }
```

请注意，这个线程是在等待状态下启动的，因为它是同步的，而且是在另一个线程之后启动的！

```
Thread.sleep(5000); pl.setWait();
Thread.sleep(3000); pl.notifyAll();
}
```

```
144         catch
145             (InterruptedException
146              n e) {
147             e.printStackTrace();
148         }
149     }
}
```

生产者-消费者模式

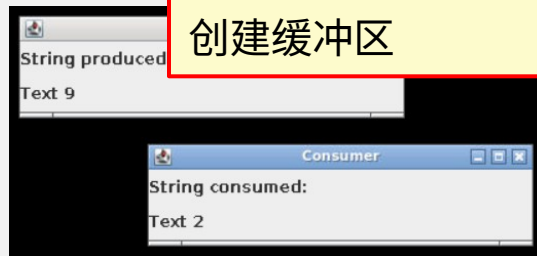
- 说明同步重要性的一个常见例子是
生产者和消费者问题
 - 生产者线程生产数据并将其添加到缓冲区中
 - 消费者线程消耗数据并从缓冲区中删除数据
 - 如果没有同步，线程可能会尝试在
导致问题



缓冲区

生产者-消费者范例图形用户界面

```
58 公共类 ProducerConsumerExample {
59      private static void createAndShowGUI() {
60          ArrayList<String> buffer = new ArrayList<>();
61          JPanel producerPanel = new JPanel();
62          producerPanel.setOpaque(true);
63          JLabel prodInfo = new JLabel("String produced:");
64          JLabel prodLabel = new JLabel();
65          ...
76          JLabel consInfo = new JLabel("String
77          consumed:"); JLabel consLabel = new JLabel();
78          ...
79          JPanel consumerPanel = new JPanel();
80          consumerPanel.setOpaque(true);
81          JLabel consInfo = new JLabel("String
82          consumed:"); JLabel consLabel = new JLabel();
83          ...
84          Producer prodThread = new Producer(buffer, prodLabel);
85          prodThread.start();
86          Consumer ConsThread = new Consumer(buffer, consLabel);
```



创建并启动生产者
和消费者线程

}
.
.
.

con
s
t
r
e
a
m
t
(
);

生产者主题

```
6  类 Producer 扩展线程 {
7      私有 ArrayList<String> buffer;
8      私有 JLabel label;
9      public Producer(ArrayList<String> buffer, JLabel label) {
10         this.buffer = buffer;
11         this.label = label;
12     }
13     @Override
14     public void run() {
15         for(int i = 0; i < 100; i++) {
16             同步 (缓冲) {
17                 String text = new String("Text " + i);
18                 System.out.println("Produced text: " + text);
19                 buffer.add(text);
20                 label.setText(text);
21             }
22         }
23     }
24 }
```

模拟制作文本项目之间的随机间隔

```
22     try {  
23         Random r = new Random();  
24         Thread.sleep(r.nextInt(800));  
25     }
```

```
26         e.printStackTrace();  
27     }  
28 }  
29 }  
30 }
```

消费者主题

```
32 class Consumer extends Thread {
33     private ArrayList<String> buffer;
34     private JLabel label;
35     public Consumer(ArrayList<String> buffer, JLabel label)
36     {
37         this.buffer = buffer;
38         this.label = label;
39     }
40     @Override
41     public void run() {
42         while(true) {
43             synchronized (buffer)
44             {
45                 if(!buffer.isEmpty()) {
46                     String text = buffer.remove(0);
47                     System.out.println("Consumed text: " + text);
48                     label.setText(text);
49                 }
50             }
51         }
52         try {
53             Thread.sleep(1000);
54         } catch (InterruptedException e) {}
55     }
56 }
```

```
$ java ProducerConsumerExample
```

制作	文本	文本	0
已消费	文本	文本	0
制作	文本	文本	1
制作	文本	文本	2
制作	文本	文本	3
制作	文本	文本	4
已消费	文本	文本	1
制作	文本	文本	5
制作	文本	文本	6
制作	文本	文本	7
已消费	文本	文本	2

模拟一秒处理
消耗时间

```
        e.printStackTrace();
    }
}
}
```

53
54
55
56
57

有问题或意见？

瓦伯丁

JC2002 Java 程序设计

第 10 天，第 2 次会议：有效性和高级并发性

有效性

- 应用程序及时执行的能力被称为作为其 *有效性*
- *死锁*、*饥饿*和*活锁*都会影响有效性
 - 当两个线程相互阻塞时就会出现 *死锁*
 - 当低优先级线程无法访问共享线程时，就会出现 *饿死现象*。资源，因为它们被高优先级的 "贪婪" 线程所保留
 - *活锁*与*死锁*类似，但线程不会被阻塞它们只是反应太慢，无法互相回应

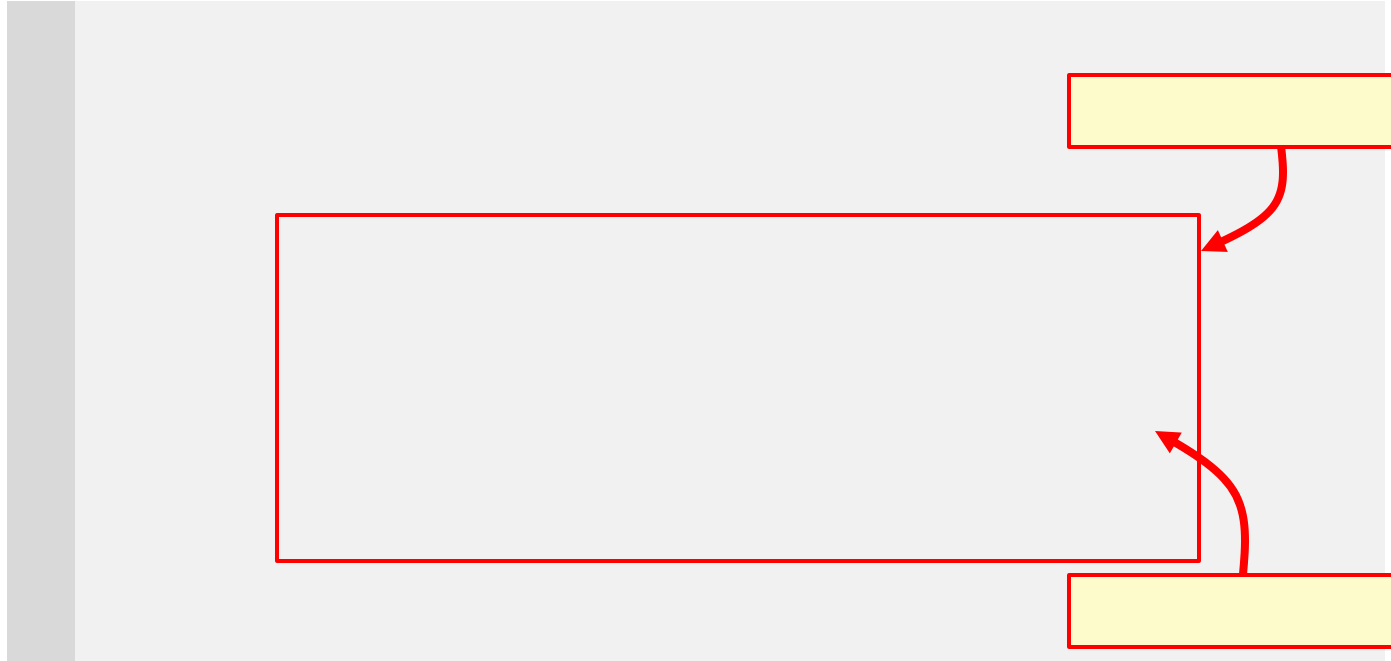
死锁示例：工人 1

```
1  公共类 DeadLockExample {
2      public static void main(String[] args) {
3          String hammer = new String("Hammer");
4          String nails = new String("Nails");
5          线程 Worker1 = 新线程 () {
6              public void run() {
7                  System.out.println("Worker 1 going to get hammer...")
8                  ;
9                  同步 (锤子) {
10                     System.out.println("Worker 1 got the hammer!");
11                     try { Thread.sleep(1000); } catch(Exception e) {}
12
13                     同步 (钉子) {
14                         System.out.println("Worker 1 got the nails!");
15                         System.out.println("Worker 1 does the work...");
16                         try { Thread.sleep(5000); } catch(Exception e) {}
17                         System.out.println("Worker 1 finished the work!")
18                         ;
19                     }
20                     System.out.println("Worker 1 returned the nails!");
21                 }
22             }
23         }
24     }
```

锁锤

```
20         System.out.println("Worker 1 returned the hammer!")
21     }
22     };
```

锁钉



死锁示例：工人 2

```
23 线程 worker2 = new Thread() { public
24      void run() {
25          System.out.println("Worker 2 going to get nails...");
26          同步 (钉子) {
27              同步 (锤子) {
28                  System.out.println("Worker 2 got the nails!");
29                  System.out.println("Worker 2 does the work...");
30                  try { Thread.sleep(5000); } catch (Exception e) {} ;
31                  System.out.println("Worker 2 finished the work!")
32                  ;
33              }
34          }
35          System.out.println("Worker 2 returned the hammer!");
36      }
37      System.out.println("Worker 2 returned the nails!");
38  };
39
40
41 worker1.start();
42
```

锁钉

锁锤

```
43     }  
44     worker2.start();  
}
```

死锁示例：输出

```
$ java DeadLockExample  
工人 1 去拿锤子...  
工人 1 拿到了锤子  
工人 2 去拿指甲...  
工人 2 拿到了钉子  
工人 2 去拿锤子...  
工人 1 去拿指甲...
```

- 程序陷入僵局，因为工人 1 拿着锤子，而工人 2 拿到了钉子，两个工人都无法继续.....

避免僵局

- 避免嵌套锁（同步块之间相互嵌套）
- 避免不必要的锁定：只锁定真正需要的对象
- 使用 *不可变* 对象，而不是通过同步锁定对象
尽可能
 - 如果一个对象在构建后其状态无法改变，那么它就是 *不可变的*
 - 不提供设置器方法，将所有字段定义为最终字段和私有字段
- 调用线程 *t* 的 **t.join()** 方法，让其他线程
在 *t* 结束后开始

- 定时版本 `join(m)` 最多等待 m 毫秒线程死亡

使用 join() 避免死锁的示例

```
1  公共类 DeadLockExample2 {  
2      public static void main(String[] args) {  
3          }  
4          String hammer = new String("Hammer");  
5          String nails = new String("Nails");  
6          ...  
7          ...  
8          ...  
9          try {  
10             worker1.start();  
11             worker1.join();  
12             worker2.start();  
13         }  
14         catch (InterruptedException e) {  
15             e.printStackTrace();  
16         }  
17     }  
18 }
```

与上例相同

}

使用 join() 避免死锁：输出

```
$ java DeadLockExample2
工人 1 去拿锤子...工人 1 拿到了锤子
工人 1 去拿钉子...工人 1 拿到了钉子
工人 1 完成工作...工人 1 完成工作!
工人 1 归还钉子工人 1 还锤子工人 2
去拿钉子...工人 2 拿到了钉子!
工人 2 去拿锤子...工人二号去拿钉子
工人 2 完成工作...工人 2 完成工作!
工人 2 归还锤子工人 2 归还钉子
$
```

高级并行性

- 到目前为止，本课程所讲解的并发是基于低级的应用程序接口对基本任务有用，但不适合更高级的任务
- 包 `java.util.concurrent` 提供了更先进的特点
 - 锁定对象，实现更复杂的同步功能
 - 执行器定义了用于启动和管理线程的高级应用程序接口
 - 用于管理和同步大型数据收集
 - 原子变量用于原子操作，无需同步

锁定对象

- 锁定对象的主要优势在于它们能够后退
试图获取锁
- 方法 **tryLock()** 可用于尝试锁定一个锁对象，如果无法
锁定（已经有人获得了锁），则返回 false。
- 也可以使用 **tryLock(m)** 的定时版本，即等待给定的超时 m
（毫秒）后才放弃
- 以及其他高级功能（不在本课程范围内）

使用锁对象避免死锁

```
1 import java.util.concurrent.locks.ReentrantLock;
2 public class LockExample {
3     public static void main(String[] args) {
4         ReentrantLock hammerLock = new ReentrantLock();
5         ReentrantLock nailLock = new ReentrantLock();
6         Thread worker1 = new Thread() {
7             public void run() {
8                 System.out.println("Worker 1 going to get hammer...");
9                 if(!hammerLock.tryLock()) {
10                     System.out.println("Hammer already taken!");
11                     返回;
12                 }
13                 System.out.println("Worker 1 got the hammer!");
14                 try { Thread.sleep(1000); } catch(Exception e) {}
15                 ...
16                 // ...尝试用同样的方法锁定钉子
17                 System.out.println("Worker 1 got the nails!");
18                 System.out.println("Worker 1 does the work...");
19                 try { Thread.sleep(5000); } catch(Exception e) {}
20                 System.out.println("Worker 1 finished the
21 work!");
22                 nailLock.unlock();
23                 System.out.println("Worker 1 returned the nails!");
24 ...
```

以类似方式实现
线程 Worker2

锁定示例：输出

```
$ java LockExample
工人 1 去拿锤子...
工人 2 去拿指甲...
工人 1 拿到了锤子
工人 1 去拿指甲...
工人 2 拿到了锤子
工人 2 去拿锤子...
```

钉子已经被拿走了!

锤子已经被拿走了

\$

- 检测到已使用的项目，避免出现死锁!

执行器接口

- java.util.concurrent 包中的执行器接口提供启动和管理任务（如线程）的方法
- 假设 r 是 Runnable，e 是 **Executor** 对象，那么就可以用 e.execute(r) 替换 (new Thread(r)).start();
；
 - 大多数执行器的实现都是为了处理由多个工作线程组成的线程池
 - 在大规模应用中的优势，例如需要以可扩展的方式协调大量线程

简单执行器示例 (1)

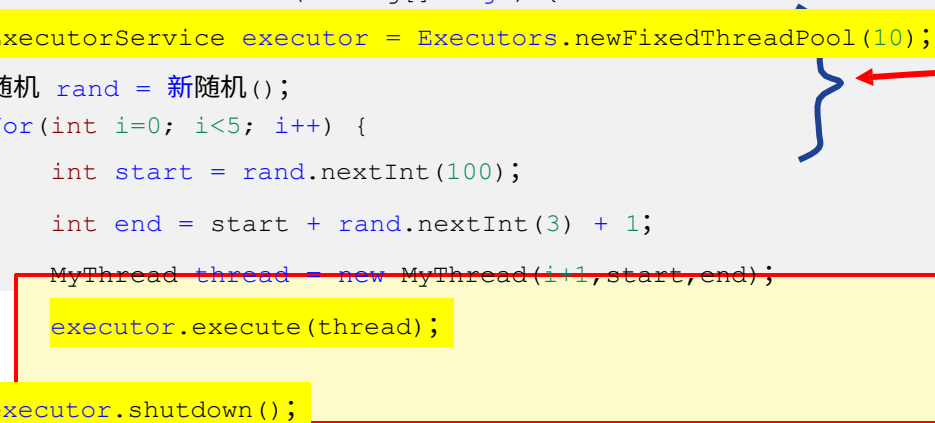
```
1  import java.util.concurrent.*;
2  import java.util.*;
3  类 MyThread 实现 Runnable {
4      int threadNum, start, end;
5      MyThread(int num, int start, int end) {
6          this.threadNum = num; this.start = start; this.end = end;
7      }
8      public void run() {
9          try {
10             for(int i = start; i <= end; i++) {
11                 System.out.printf("Thread #d, step %d\n", threadNum,i);
12                 随机 rand = 新随机();
13                 Thread.sleep(rand.nextInt(1000));
14             }
15         }
16         catch (InterruptedException e) {
```

执行自定义线程
照常

```
17         e.printStackTrace();
18     }
19 }
20 }
```

简单执行器示例 (2)

```
21 公共类 ExecutorExample {  
22      public static void main(String[] args) {  
23          ExecutorService executor = Executors.newFixedThreadPool(10);  
24          随机 rand = 新随机();  
25          for(int i=0; i<5; i++) {  
26              int start = rand.nextInt(100);  
27              int end = start + rand.nextInt(3) + 1;  
28              MyThread thread = new MyThread(i+1, start, end);  
29              executor.execute(thread);  
30          }  
31          executor.shutdown();  
32      }  
33  }
```



创建五个具有不同随机特征的线程，并通过执行器对象

执行它们

简单执行器示例：输出

```
$ java ExecutorExample
```

```
螺纹 螺 #1, 第 46  
          #5, 步
```

```
纹 螺纹 #4, 步骤 19  
          #3,
```

```
螺纹     #2, 步骤 49
```

```
$         #3, 步骤 49  
          #2,
```

```
          #1, 第 24
```

```
          #2,
```

```
          #3, 步
```

```
          #3, 步骤 50
```

```
          #4,
```

```
          #5, 步骤 25
```

```
          #4,  
          第 47
```

步

步骤 26

步骤 51

关于高级并发性的结束语

- 并发是一个非常复杂的话题，尤其是当多核平台
- 对于大多数程序员来说，低级应用程序接口已经足够，但对于处理大量数据和线程的高级应用程序来说，`java.util.concurrent` 的高级应用程序接口则是必不可少的。
- 欲了解更多详情：
 - <https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>

摘要

- 并发编程通常用于实现定时事件
 - 同步以及 `wait()` 和 `notify()` 方法可用于只允许一个线程同时访问资源，并使线程等待另一个线程。
- 多线程要求程序员考虑以下问题
线程干扰和死锁
 - 使用锁可以避免线程干扰，但这可能会导致死锁和其他延迟问题

有问题或意见？

瓦伯丁