

Java 应用程序设计

线程编程

王晓东

wxd2870@163.com

中国海洋大学

April 24, 2018



本章学习目标

1. 线程基础
2. 线程控制
3. 线程的同步



大纲

线程基础

线程控制

线程的同步



接下来...

线程基础

线程控制

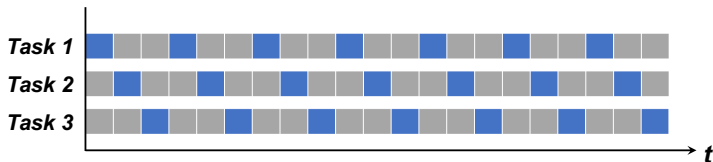
线程的同步



概念回顾

❖ 任务调度

- ▶ 大部分操作系统的任务调度是采用时间片轮转的抢占式调度方式，一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。
- ▶ CPU 的执行效率非常高，时间片非常短，在各个任务之间快速地切换，让人感觉像是多个任务在“同时进行”，这也就是我们所说的并发。



概念回顾

❖ 进程

- ▶ 进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。
- ▶ 进程一般由**程序段、数据段和进程控制块**三部分构成进程实体。



什么是线程

根据多任务原理，在一个程序内部也可以实现多个任务（顺序控制流）的并发执行，其中每个任务被称为线程（Thread）。更专业的表述为：**线程是程序内部的顺序控制流。**



线程和进程的区别和联系



1. 每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。
2. 线程作为“轻量的进程”，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。
3. 多进程——在操作系统中能同时运行多个任务（程序）。
4. 多线程——在同一应用程序中有多个顺序流同时执行。



线程和进程的区别和联系



1. 每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。
2. 线程作为“轻量的进程”，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。
3. 多进程——在操作系统中能同时运行多个任务（程序）。
4. 多线程——在同一应用程序中有多个顺序流同时执行。

多核与多线程

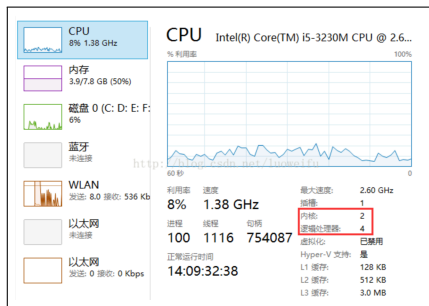
- ▶ 多核处理器是指在一个处理器上集成多个运算核心以提高并行计算能力，每一个处理核心对应一个内核线程（Kernel Thread, KLT）。
- ▶ 内核线程是直接由操作系统内核支持的线程，由内核来完成线程切换，内核通过操作调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。



多核与多线程

一般一个处理核心对应一个内核线程，比如单核处理器对应一个内核线程，双核处理器对应两个内核线程。

而现代计算机采用超线程技术将一个物理处理核心模拟成两个逻辑处理核心对应两个内核线程，一般是双核四线程、四核八线程。



课后自行学习

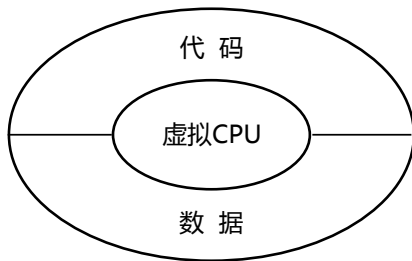
1. 超线程的概念
2. 内核线程与用户线程的映射



Java 线程的概念模型

在 Java 语言中，多线程的机制通过虚拟 CPU 来实现。

1. 虚拟的 CPU，由 `java.lang.Thread` 类封装和虚拟；
2. CPU 所执行的代码，传递给 `Thread` 类对象；
3. CPU 所处理的数据，传递给 `Thread` 类代码对象。



创建线程

Java 的线程是通过 `java.lang.Thread` 类来实现的。每个线程都是通过某个特定 `Thread` 对象所对应的方法 `run()` 来完成其操作的，方法 `run()` 称为线程体。

```
1 public class TestThread1 {  
2     public static void main(String args[]) {  
3         Runner1 r = new Runner1();  
4         Thread t = new Thread(r);  
5         t.start();  
6     }  
7 }  
8 class Runner1 implements Runnable {  
9     public void run() {  
10         for(int i=0; i<30; i++) {  
11             System.out.println("No.␣" + i);  
12         }  
13     }  
14 }
```



创建和启动线程的一般步骤

1. 定义一个类实现 Runnable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runnable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runnable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。



多线程

Java 中引入线程机制的目的在于实现多线程（Multi-Thread）并发执行任务，以及实现多任务之间的协同。

使用多线程

```
1 public class TestThread2 {
2     public static void main(String args[]) {
3         Runner2 r = new Runner2();
4         Thread t1 = new Thread(r);
5         Thread t2 = new Thread(r);
6         t1.start();
7         t2.start();
8     }
9 }
10 class Runner2 implements Runnable {
11     public void run() {
12         for(int i=0; i<20; i++) {
13             String s = Thread.currentThread().getName(); //获取当前运行中的线程对象
14             System.out.println(s + ": " + i);
15         }
16     }
17 }
```



多线程共享代码和数据

多线程之间可以共享代码和数据。

```
1 Runner2 r = new Runner2();  
2 Thread t1 = new Thread(r);  
3 Thread t2 = new Thread(r);
```

线程	虚拟 CPU	代码
t1	Thread 类对象	Runner2 类中的 run() 方法
t2	Thread 类对象	Runner2 类中的 run() 方法



创建线程的第二种方式

直接继承 Thread 类创建线程

```
1 public class TestThread3 {  
2     public static void main(String args[]) {  
3         Thread t = new Runner3();  
4         t.start();  
5     }  
6 }  
7 class Runner3 extends Thread {  
8     public void run() {  
9         for(int i=0; i<30; i++) {  
10             System.out.println("No. " + i);  
11         }  
12     }  
13 }
```

❖ 第二种创建线程的方式

1. 定义一个类继承 Thread 类，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建该 Thread 类的对象；
3. 调用该对象的 start() 方法。



两种创建线程的方式比较

❖ 使用 Runnable 接口创建线程

- ▶ 可以将虚拟 CPU、代码和数据分开，形成清晰的模型；
- ▶ 线程体 `run()` 方法所在的类还可以从其他类继承一些有用的属性或方法；
- ▶ 有利于保持程序风格的一致性。

❖ 直接继承 Thread 类创建线程

- ▶ Thread 子类无法再从其他类继承；
- ▶ 编写简单，`run()` 方法的当前对象就是线程对象，可直接操纵。



后台线程

❖ 相关概念

后台处理 也称为后台运行，是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程 (Daemon Thread)。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。



后台线程

❖ 相关概念

后台处理 也称为后台运行，是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程 (Daemon Thread)。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。



后台线程

❖ 相关概念

后台处理 也称为后台运行，是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程 (Daemon Thread)。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。



后台线程

Thread 类提供的与后台线程相关的方法包括：

1. 测试当前线程是否为守护线程，如果是则返回 true，否则返回 false。

```
1 public final boolean isDaemon()
```

2. 将当前线程标记为守护线程或用户线程，本方法必须在启动线程前调用。

```
1 public final void setDaemon(Boolean on)
```



后台线程

CODE sample.thread.DaemonThreadSample.java

```
1 public class DaemonThreadSample {
2     public static void main(String[] args) {
3         Thread t1 = new MyRunner(10);
4         t1.setName("用户线程t1");
5         t1.start();
6         Thread t2 = new MyRunner(10000);
7         t2.setDaemon(true);
8         t2.setName("后台线程t2");
9         t2.start();
10        for(int i = 0; i < 10; i++) {
11            System.out.println(Thread.currentThread().getName() + ":_" + i);
12        }
13        System.out.println("主线程结束");
14    }
15 }
16 class MyRunner extends Thread {
17     private int n;
18     public MyRunner(int n) {
19         this.n = n;
20     }
21     public void run() {
22         for(int i = 0; i < n; i++) {
23             System.out.println(this.getName() + ":_" + i);
24         }
25         System.out.println(this.getName() + "结束");
26     }
27 }
```



后台线程

❖ 对上述代码的分析

后台线程线程 t2 并没有如预期的输出数字 0-9999，而是提前终止。这是因为，待用户线程（这里包括主线程和线程 t1）全部运行结束后，JVM 检测到只剩下后台线程在运行的时候，就退出了当前应用程序的运行。

将上述代码中的“t2.setDaemon(true);”注释后，编译运行程序进行比较。



GUI 线程

GUI 程序运行过程中，系统会自动创建若干 GUI 线程，以提供 GUI 所需的功能，主要包括：

1. 窗体显示和重绘；
2. GUI 事件处理；
3. 关闭抽象窗口工具集等。



GUI 线程

使用 GUI 线程示例

CODE ♦ GUIThreadSample.java: Part 1

```
1  import java.awt.*;
2  import java.awt.event.*;

4  public class GUIThreadSample {
5      public static void main(String[] args) throws Exception {
6          Frame f = new Frame();
7          Button b = new Button("Press Me");
8          MyMonitor mm = new MyMonitor();
9          b.addActionListener(mm);
10         f.addWindowListener(mm);
11         f.add(b, "Center");
12         f.setSize(100, 60);
13         f.setVisible(true);
14         MyThreadViewer.view();
15     }
16 }

17 class MyMonitor extends WindowAdapter implements ActionListener {
18     public void actionPerformed(ActionEvent e) {
19         MyThreadViewer.view();
20     }
21 }
```



GUI 线程

CODE ♦ GUIThreadSample.java: Part 2

```
1  class MyThreadViewer {
2      public static void view() {
3          Thread current = Thread.currentThread();
4          System.out.println("当前线程名称: " + current.getName());
5          int total = Thread.activeCount();
6          System.out.println("活动线程总数: " + total + "个");
7          Thread[] threads = new Thread[total];
8          current.enumerate(threads);
9          for(Thread t: threads) {
10             String role = t.isDaemon() ? "后台线程" : "用户线程";
11             System.out.println("UUU-" + role + t.getName());
12         }
13         System.out.println("-----");
14     }
15 }
```



GUI 自动创建的线程

- ▶ AWT-Windows 线程

负责从操作系统获取底层事件通知，并将之发送到系统事件队列（EventQueue）等待处理。在其他平台上运行时，此线程的名字也会作相应变化，例如在 Unix 系统则为“AWT-Unix”。

- ▶ AWT-EventQueue-n 线程

- ▶ AWT-Shutdown 线程

- ▶ DestroyJavaVM 线程



GUI 自动创建的线程

- ▶ AWT-Windows 线程

- ▶ AWT-EventQueue-n 线程

也称事件分派线程，该线程负责从事件队列中获取事件，将之分派到相应的 GUI 组件（事件源）上，进而触发各种 GUI 事件处理对象，并将之传递给相应的事件监听器进行处理。

- ▶ AWT-Shutdown 线程

- ▶ DestroyJavaVM 线程



GUI 自动创建的线程

- ▶ AWT-Windows 线程
- ▶ AWT-EventQueue-n 线程
- ▶ AWT-Shutdown 线程
负责关闭已启用的抽象窗口工具，释放其所占用的资源，该线程将等到其他 GUI 线程均退出后才开始其清理工作。
- ▶ DestroyJavaVM 线程



GUI 自动创建的线程

- ▶ AWT-Windows 线程
- ▶ AWT-EventQueue-n 线程
- ▶ AWT-Shutdown 线程
- ▶ DestroyJavaVM 线程

在所有其他用户线程退出后，负责释放任意线程所占用系统资源并卸载 Java 虚拟机。该线程在主线程运行结束时由系统自动启动，但要等到所有其他用户线程均退出后才开始其卸载工作。



接下来...

线程基础

线程控制

线程的同步



线程的生命周期

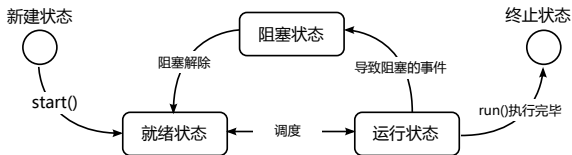
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

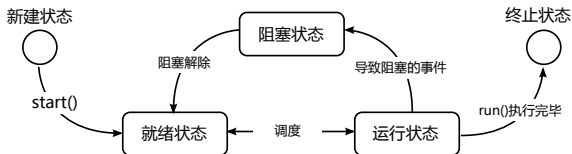
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

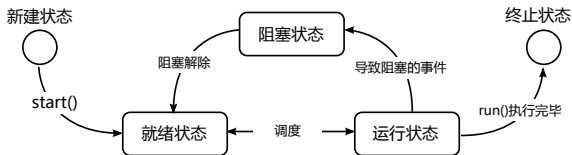
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

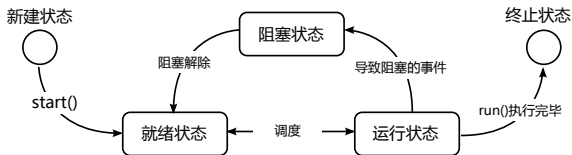
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程的生命周期

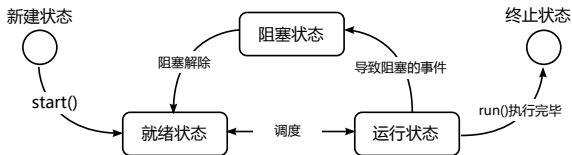
新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。



线程优先级

线程的优先级用数字来表示，范围从 1 到 10。主线程的缺省优先级是 5，子线程的优先级默认与其父线程相同。可以使用 Thread 类提供的方法获得和设置线程的优先级：

► 获取当前线程优先级

```
1 public final int getPriority();
```

► 设定当前线程优先级

```
1 public final void setPriority(int newPriority);
```

相关静态整型常量：

```
1 Thread.MIN_PRIORITY = 1  
2 Thread.MAX_PRIORITY = 10  
3 Thread.NORM_PRIORITY = 5
```



练习

请自行编写线程优先级测试代码。



线程串行化

在多线程程序中，如果在一个线程运行的过程中要用到另一个线程的运行结果，则可进行线程的串型化处理。


Thread 类提供的相关方法：

```
1 public final void join()  
2 public final void join(long millis)  
3 public final void join(long millis, int nanos)
```



线程串行化

实现线程的串行化

CODE  sample.thread.ThreadJoinSample.java

```
1 public class ThreadJoinSample {
2     public static void main(String[] args) {
3         JoinRunner r = new JoinRunner();
4         Thread t = new Thread(r);
5         t.start();
6         try {
7             t.join();
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        for(int i = 0; i < 50; i++) {
12            System.out.println("主线程: " + i);
13        }
14    }
15 }
16 class JoinRunner implements Runnable {
17     public void run() {
18         for(int i = 0; i < 50; i++) {
19             System.out.println("子线程: " + i);
20         }
21     }
22 }
```



线程串行化

❖ 线程串行化程序说明

- ▶ 主线程在执行过程中调用了线程 t 的 join() 方法，该方法导致当前线程（主线程）阻塞。
- ▶ 直到线程 t 运行终止后，主线程才会获得继续执行的机会，相当于将线程 t 串行加入到主线程中。



线程休眠

线程休眠，即暂停执行当前运行中的线程，使之进入阻塞状态，待经过指定的“延迟时间”后再醒来并转入到就绪状态。

Thread 类提供的相关方法：

```
1 public static void sleep(long millis)
2 public static void sleep(long millis, int nanos)
```



线程休眠

数字计数器

CODE sample.thread.DigitaltimerByThreadSleep.java

```
1  import javax.swing.*;
3  public class DigitalClock {
4      public static void main(String[] args) {
5          JFrame jf = new JFrame("Clock");
6          JLabel clock = new JLabel("clock");
7          clock.setHorizontalAlignment(JLabel.CENTER);
8          jf.add(clock, "Center");
9          jf.setSize(140, 80);
10         jf.setLocation(500, 300);
11         jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         jf.setVisible(true);
13         Thread t = new MyThread(clock);
14         t.start();
15     }
16 }
```



线程休眠

CODE ♦ DigitalTimer.java+

```
1  class MyThread extends Thread {  
2      private JLabel clock;  
3      private int i;  
4      public MyThread(JLabel clock) {  
5          this.clock = clock;  
6          this.i = 1;  
7      }  
8      public void run() {  
9          while(true) {  
10             clock.setText(String.valueOf(i++));  
11             try {  
12                 Thread.sleep(1000);  
13             } catch (InterruptedException e) {  
14                 e.printStackTrace();  
15             }  
16         }  
17     }  
18 }
```



线程让步

线程让步，让运行中的线程主动放弃当前获得的 CPU 处理机会，但不是使该线程阻塞，而是使之转入就绪状态。

Thread 类提供的相关方法：

```
1 public static void yield()
```

线程让步示例

CODE ▶ TestYield.java

```
1 import java.util.Date;

3 public class TestYield {
4     public static void main(String[] args) {
5         Thread t1 = new MyThread(false);
6         Thread t2 = new MyThread(true);
7         Thread t3 = new MyThread(false);
8         t1.start();
9         t2.start();
10        t3.start();
11    }
12 }
```



线程让步

CODE TestYield.java+

```
1  class MyThread extends Thread {
2      private boolean flag;
3      public MyThread(boolean flag) {
4          this.flag = flag;
5      }
6      public void setFlag(boolean flag) {
7          this.flag = flag;
8      }
9      public void run() {
10         long start = new Date().getTime();
11         for(int i = 0; i < 200; i++) {
12             if(flag) {
13                 Thread.yield();
14             }
15             System.out.println(this.getName() + ": " + i + "\t");
16         }
17         long end = new Date().getTime();
18         System.out.println("\n" + this.getName() + "执行时间: " + (end - start) + "ms");
19     }
20 }
```

从执行结果来看，由于设置了线程让步，thread-1（第二个线程）明显执行时间长。调用 `yield()` 方法只是令当前线程主动在时间片到期前使其他线程获得运行机会。



线程挂起与恢复

线程挂起 暂时停止当前运行中的线程，使之转入阻塞状态，并且不会自动恢复运行。

线程恢复 使得一个已挂起的线程恢复运行。

Thread 类提供的相关方法：

```
1 public final void suspend()  
2 public final void resume()
```

注意

suspend() 和 resume() 方法已不提倡使用，原因是 suspend() 方法挂起线程时并不释放其锁定的资源，这可能会影响到其他线程的执行，且容易导致线程死锁。



线程等待与通知

❖ 将运行中的线程转为阻塞状态的另外一种途径

调用该线程中被锁定资源（Java 对象）的 `wait()` 方法，该方法在 `Object` 类中定义，其功能是让当前线程等待，直到有其他线程调用了同一个对象的 `notify()` 或 `notifyAll()` 方法通知其结束等待，或是经历了约定的等待时间后，等待线程才会醒来，重新进入可执行状态。

❖ 等待线程与 `suspend()` 方法导致的线程挂起比较

- ▶ 线程挂起时不会释放所占用的资源；
- ▶ 线程等待时则会释放资源，以使其他线程获得运行机会。



线程等待与通知

❖ 将运行中的线程转为阻塞状态的另外一种途径

调用该线程中被锁定资源（Java 对象）的 `wait()` 方法，该方法在 `Object` 类中定义，其功能是让当前线程等待，直到有其他线程调用了同一个对象的 `notify()` 或 `notifyAll()` 方法通知其结束等待，或是经历了约定的等待时间后，等待线程才会醒来，重新进入可执行状态。

❖ 等待线程与 `suspend()` 方法导致的线程挂起比较

- ▶ 线程挂起时不会释放所占用的资源；
- ▶ 线程等待时则会释放资源，以使其他线程获得运行机会。



接下来...

线程基础

线程控制

线程的同步



临界资源问题

两个线程 A 和 B 在同时操纵 Stack 类的同一个实例 (栈), A 向栈里 push 一个数据, B 要从堆栈中 pop 一个数据。

❖ 代码

```
1 public class Stack {  
2     int idx = 0;  
3     char[ ] data = new char[6];  
  
5     public void push(char c) {  
6         data[idx] = c;  
7         idx++;  
8     }  
9     public char pop() {  
10        idx--;  
11        return data[idx];  
12    }  
13 }
```



临界资源问题

❖ 问题分析

1. 操作之前，假设 $data = |a|b| || ||$ ， $idx = 2$;
2. 线程 A 执行 `push` 中的第一个语句，将 `c` 推入堆栈; $data = |a|b|c| || ||$ ， $idx = 2$;
3. 线程 A 还未执行 `idx++` 语句，A 的执行被线程 B 中断，B 执行 `pop` 方法， $data = |a|b|c| || ||$ $idx = 1$;
4. 线程 A 继续执行 `push` 的第二个语句: $data = |a|b|c| || ||$ ， $idx = 2$;
5. 最后的结果相当于 `c` 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。



互斥锁

- ▶ Java 引入了**对象互斥锁**的概念来保证共享数据操作的完整性。
- ▶ 每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。
- ▶ 关键字**synchronized**来与对象的互斥锁联系。当某个对象用 synchronized 修饰时，表明该对象在任一时刻只能由一个线程访问。



synchronized 的用法

❖ 用于方法声明中，标明整个方法为同步方法

```
1 public synchronized void push(char c) {  
2     data[idx] = c;  
3     idx++;  
4 }
```

❖ 用于修饰语句块，标明整个语句块为同步块

```
1 // Other code  
2 public char pop() {  
3     synchronized(this) {  
4         idx--;  
5         return data[idx];  
6     }  
7     // Other code  
8 }
```



线程死锁

并发运行的多个线程间彼此等待、都无法运行的状态称为线程死锁。

为避免死锁，在线程进入阻塞状态时应尽量释放其锁定的资源，以为其他的线程提供运行的机会。

❖ 相关方法

- ▶ `public final void wait()`
- ▶ `public final void notify()`
- ▶ `public final void notifyAll()`



生产者—消费者问题

```
1 public class SyncStack {
2     private int index = 0;
3     private char[] data = new char[6];

4
5     public synchronized void push(char c) {
6         while(index == data.length) {
7             try {
8                 this.wait();
9             } catch (InterruptedException e) {
10            }
11        }
12        this.notify();
13        data[index] = c;
14        index++;
15        System.out.println("生产:_" + c);
16    }

17
18    public synchronized char pop() {
19        while(index == 0) {
20            try {
21                this.wait();
22            } catch (InterruptedException e) {
23            }
24        }
25        this.notify();
26        index--;
27        System.out.println("消费:_" + data[index]);
28        return data[index];
29    }
30 }
```



生产者—消费者问题

❖ 生产者

```
1 public class Producer implements Runnable {  
2     SyncStack stack;  
3     public Producer(SyncStack s) {  
4         stack = s;  
5     }  
6     public void run() {  
7         for(int i=0; i<20; i++) {  
8             char c = (char)(Math.random() * 26 + 'A');  
9             stack.push(c);  
10            try {  
11                Thread.sleep((int)(Math.random() * 300));  
12            } catch(InterruptedException e) {  
13                e.printStackTrace();  
14            }  
15        }  
16    }  
17 }
```



标题生产者—消费者问题

❖ 消费者

```
1 public class Consumer implements Runnable {  
2     SyncStack stack;  
3     public Consumer(SyncStack s) {  
4         stack = s;  
5     }  
6     public void run() {  
7         for(int i=0; i<20; i++) {  
8             char c = stack.pop();  
9             try {  
10                Thread.sleep((int)(Math.random() * 800));  
11            } catch(InterruptedException e) {  
12                e.printStackTrace();  
13            }  
14        }  
15    }  
16 }
```



THE END

wxd2870@163.com

