

Java语言程序设计

配套教材由清华大学出版社出版发行

第8章 多线程并发编程



中國農業大學

阚道宏

第8章 多线程并发编程

- 计算机可以同时运行多个程序，这样就能在同一台计算机上同时做不同的事情
- 在单个CPU上同时运行多个程序将采用**分时**（time-sharing）技术
 - 把CPU的运行时间划分成很小的**时间片**（time slice），然后按时间片轮流执行各程序
 - 如果程序在用完一个时间片之后未能完成执行，则被暂时挂起，等待下一轮继续执行
- 采用分时技术同时执行多个程序的方式被称为**并发**（concurrency）
- **操作系统**全权负责管理和调度多个程序的并发执行，程序员在编程时不需要做什么事情



第8章 多线程并发编程

- 本章内容
 - [8.1 多线程并发程序](#)
 - [8.2 多线程编程及并发调度](#)
 - [8.3 多线程之间的并发与互斥](#)
 - [8.4 多线程之间的协同](#)
 - [8.5 定时执行的线程](#)
 - [8.6 Swing框架中的线程](#)



8.1 多线程并发程序

- 进程与线程

- 进程

- 操作系统为每个加载到内存执行的程序创建一个**进程**（process）
 - 进程可理解为一个**运行环境**（execution environment），具有运行程序所需的**计算资源**和**存储资源**
 - 每个进程运行一个程序，多个进程就可以同时运行多个程序，这就是**进程并发**
 - 多个进程通过**分时技术**分享CPU的计算资源，通过**地址空间映射技术**分享内存的存储资源
 - **操作系统**全权负责进程的创建、管理、调度和删除，并为进程分配CPU时间片和内存空间
 - **程序员**可以忽略进程的存在，编程时通常不需要为进程做什么事情



8.1 多线程并发程序

- 进程与线程

- 线程

- 程序可能需要完成比较复杂的任务，可以将复杂任务分解成多个小的子任务
 - 如果将完成子任务的**指令序列**（即语句序列）称作一个**算法**，那么一个程序可以包含多个算法
 - 程序中的多个算法是按顺序依次执行的，我们称之为**串行**执行
 - 程序可以为其中的**算法**单独创建**线程**（thread），每个线程负责执行一个算法
 - 多个线程之间各自独立运行，通过**分时技术**可以同时执行多个算法，这就是**线程并发**
 - 一个**多线程并发程序**在运行时看起来就像是同时在做多件事情



8.1 多线程并发程序

- 进程与线程
 - 进程与线程的关系
 - 进程包含线程
 - 程序所创建的线程包含于运行该程序的进程之中。一个线程可理解为是包含于进程内部的一个可独立运行算法的**运行环境**
 - 在进程中创建线程，其目的是为了对进程的**计算资源**做进一步细分
 - 程序员可运用线程技术对程序做更直接、更精细的并发控制
 - 一个进程可以包含多个线程
 - 同一线程的算法内部是**串行**执行的，而不同线程的算法之间则是通过分时技术**并发**执行的
 - 每个线程所执行的算法是一个指令序列，我们将其称作是一个**指令执行流**。一个线程包含一个指令执行流
 - 一个进程中可以有多个线程，因此一个进程可能包含多个并发执行的指令执行流
 - 进程中的**存储资源**可以被进程中各线程所运行的算法共享
 - 不同线程的算法之间虽然是各自独立执行的，但它们需要协同工作
 - 通过访问共同的存储资源，同一进程中不同线程的算法之间可以共享数据，进而实现多线程协同工作



8.1 多线程并发程序

- 进程与线程

- 进程与线程的关系

- 进程和线程**总结**

- 一个进程运行一个程序。操作系统为每个加载到内存执行的程序创建一个进程
 - 一个线程运行一个算法。一个程序可以划分成多个算法，将算法分散交由不同的线程去执行，这样可以实现多线程并发执行。程序员负责创建线程，并为线程指定所要运行的算法
 - 一个进程可以包含多个线程。同一进程中的多个线程之间虽然各自独立运行，但需要共享数据，这样才能实现多线程协同工作



8.1 多线程并发程序


- 单线程串行程序

- 每个Java程序在运行时都会单独创建一个进程。该进程自动包含一个由系统创建的**主线程**（main thread），用于运行程序的主方法**main()**。主线程从主方法的第一条语句开始执行，直到最后一条语句执行结束，或遇到return语句中途退出时为止
- 同一线程的算法内部是**串行**执行的。如果程序员没有为程序额外创建线程，那么该程序在运行时将只有一个主线程。主线程按串行方式执行程序中的指令序列，这是一种**单线程串行**程序
- 如果程序的**主方法**调用某个**子方法**，则主线程在执行到方法调用语句时将暂停主方法的执行，转去执行子方法，执行结束后再返回主方法继续执行。这种程序仍然属于单线程串行程序，因为主方法与子方法的算法是在同一线程中按串行方式执行的



例8-1 一个模拟音乐播放器的单线程串行Java演示程序（JPlayerST.java）

```
1 public class JPlayerST {           // 主类：单线程串程序
2     public static void main(String[] args) { // 主方法
3         // 下载算法：模拟网络下载，显示5次信息"Downloading ..."
4         int count = 1;    // 显示计数
5         while (count <= 5) { // 循环显示5次信息
6             System.out.println("Downloading ..." +count);
7             count++;
8             // 每显示一次信息后让算法休眠（暂停）0.1秒，模拟下载过程
9             try { // 捕获并处理可能抛出的勾选异常
10                 Thread.sleep(100); // 可能抛出勾选异常InterruptedException
11             }
12             catch (InterruptedException e) { // 捕捉并处理异常
13                 System.out.println( e.getMessage() );
14                 return; // 退出主方法，程序结束
15             }
16         }
17         play(); // 下载完成后，调用子方法play()，模拟播放音乐
18     }
19
20     static void play() { // 子方法
21         // 播放算法：模拟播放音乐，显示5次信息"Playing ..."
22         int count = 1;    // 显示计数
23         while (count <= 5) { // 循环显示5次信息
24             System.out.println("\t Playing ..." +count);
25             count++;
26             // 每显示一次信息后让算法休眠（暂停）0.1秒。模拟
27             try { // 捕获并处理可能抛出的勾选异常
28                 Thread.sleep(100); // 可能抛出勾选异常InterruptedException
29             }
30             catch (InterruptedException e) { // 捕捉并处理异常
31                 System.out.println( e.getMessage() );
32                 return; // 退出子方法，返回主方法
33             }
34         }
35     } }
```



Problems @ Javadoc Declaration Console

<terminated> JPlayerST [Java Application] C:\Java\jre

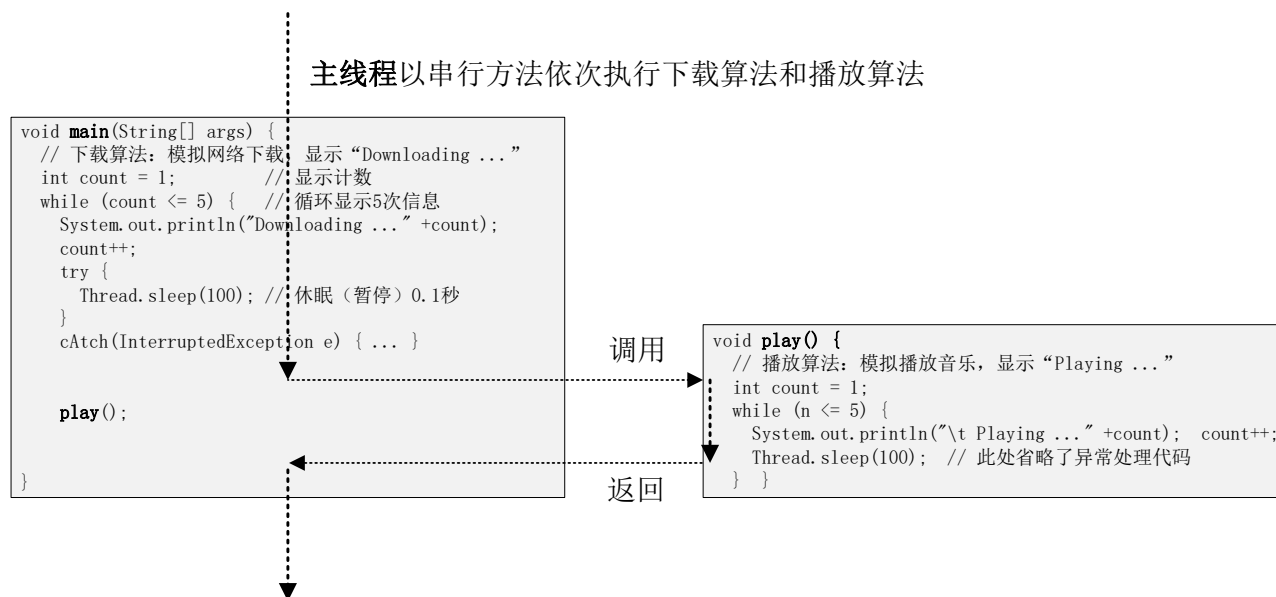
Downloading ...1
Downloading ...2
Downloading ...3
Downloading ...4
Downloading ...5
 Playing ...1
 Playing ...2
 Playing ...3
 Playing ...4
 Playing ...5

8.1 多线程并发程序

- 单线程串行程序

- 单线程串行程序的执行流程

有**下载**和**播放**两个算法，但只有一个**主线程**。



8.1 多线程并发程序

- 单线程串行程序
 - 线程的休眠

```
try { // 捕获并处理可能抛出的勾选异常
    Thread.sleep(100); // 可能抛出勾选异常
}
catch (InterruptedException e) { // 捕捉并处理异常
    System.out.println( e.getMessage() );
    return; // 退出主方法，程序结束
}
```



8.1 多线程并发程序

- 多线程并发程序
 - 让音乐播放器程序能边下载，边播放
 - Java API
 - 可运行接口 **Runnable**
 - 线程类 **Thread**
 - 将算法放入一个线程中运行
 - 将算法封装成一个可运行的 **算法对象**
 - 创建 **线程对象**，在线程对象中运行算法对象
 - 由程序员创建的线程对象被统称为 **子线程**（sub thread）。子线程在启动后将与主线程并发执行，分头执行各自的算法
 - 一个程序可以 **包含** 多个线程，其中一个是系统自动创建的主线程，其余的则是由程序员创建的子线程
 - 多个线程之间通过 **分时技术** 并发执行，这样的程序被称为是 **多线程并发程序**



例 8-2 一个模拟音乐播放器的多线程并发 Java 演示程序 (JPlayerMT.java)

```

1  public class JPlayerMT {                                // 主类：多线程并发程序
2      public static void main(String[] args) {           // 主方法
3          // 将播放算法放入单独的线程中去执行
4          PlayAlgorithm a = new PlayAlgorithm();          // 创建一个可运行的播放算法对象 a
5          Thread t = new Thread(a);                      // 新建子线程 t，在 t 中运行算法对象 a
6          t.start();                                      // 启动子线程 t
7
24     class PlayAlgorithm implements Runnable {           // 可运行的算法类 PlayAlgorithm
25         public void run() {                             // 描述算法的方法 run
26             // 播放算法：模拟播放音乐，显示 5 次信息 "Playing ..."
27             int count = 1;                               // 显示计数
28             while (count <= 5) { // 循环显示 5 次信息
29                 System.out.println("\t Playing ..." + count); count++;
30                 // 每显示一次信息后让程序休眠（暂停）0.1 秒。模拟播放过程
31                 try { // 捕获并处理可中断异常
32                     Thread.sleep(100);
33                 }
34                 catch (InterruptedException e) {
35                     System.out.println(e.getMessage());
36                     return; // 退出
37                 }
38             }
39             // 执行完算法代码后退出 run 方法
40         } }
22 } }

```

Problems @ Javadoc Declaration Console

```

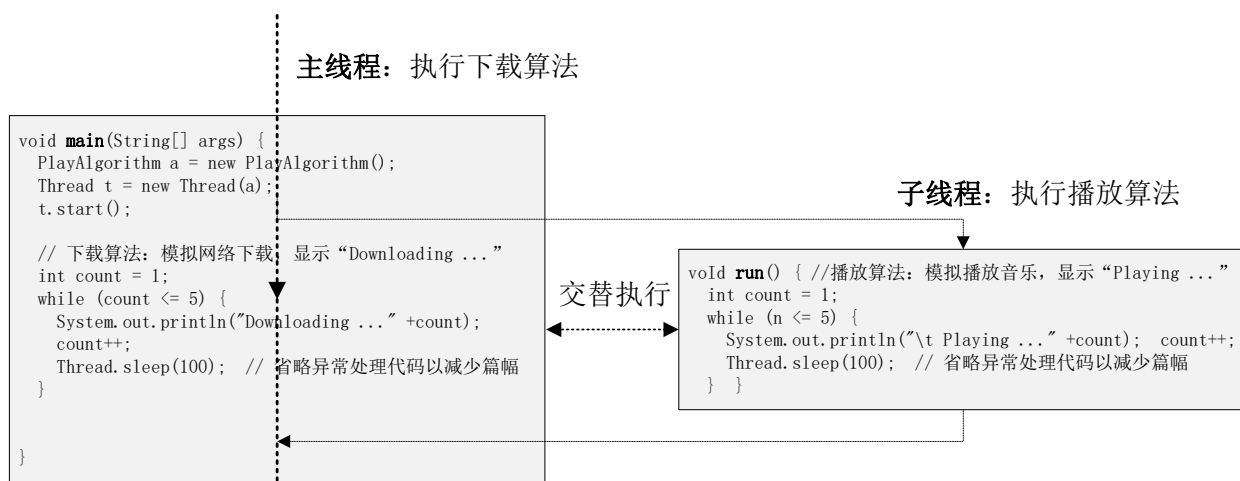
<terminated> JPlayerMT [Java Application] C:\Java\jre
Downloading ...1
    Playing ...1
Downloading ...2
    Playing ...2
    Playing ...3
Downloading ...3
    Playing ...4
Downloading ...4
Downloading ...5
    Playing ...5

```



8.1 多线程并发程序

- 多线程并发程序
 - 多线程并发程序的执行流程
 - 主线程**：运行下载算法
 - 子线程**：是在主线程中创建的，运行播放算法



多线程并发执行具有**随机性**



8.2 多线程编程及并发调度

- 3个非常重要的概念
 - 算法
 - 运行算法的线程
 - 多线程在执行时的并发调度
- 算法
 - 算法是程序中完成某种功能的指令序列（即语句序列），一个程序可以包含多个算法
 - 如果将算法代码封装成可以被线程运行的算法对象，然后将它单独放入一个线程，那么这个算法就能与其他线程里的算法并发执行
 - Java API提供了一个“可运行的”接口Runnable，用于将算法封装成可被线程运行的算法对象



8.2 多线程编程及并发调度

- 算法
 - “可运行的” 接口 **Runnable**

java.lang. Runnable 接口说明文档			
@FunctionalInterface			
public interface Runnable			
	修饰符	功能接口成员	功能说明
1		void run()	描述需在线程中执行的算法代码

```
class 算法类名 implements Runnable {  
    public void run() {  
        ..... // 此处编写需被并发执行的算法代码  
    }  
}
```

PlayAlgorithm a = **new** PlayAlgorithm(); // 创建一个可运行的播放算法对象a

```
Runnable a = new Runnable() { // 改用匿名类直接创建一个播放算法对象a  
    public void run() {  
        ..... // 此处编写需并发执行的播放算法  
    }  
};
```



中国农业大学

阚道宏

java.lang.Thread类说明文档			
public class Thread			
extends Object			
implements Runnable			
	修饰符	类成员（节选）	功能说明
1	static	int MAX_PRIORITY	优先级常量，最大优先级
2	static	int MIN_PRIORITY	优先级常量，最小优先级
3	static	int NORM_PRIORITY	优先级常量，一般优先级
4	static	void sleep (long millis)	休眠，释放执行权
5	static	void yield ()	释放执行权
6	static	Thread currentThread ()	返回当前正在执行的线程
7	static	boolean interrupted ()	检查当前线程是否有中止请求
8		Thread (Runnable target)	构造方法
9		Thread (Runnable target, String name)	构造方法
10		void start ()	启动线程
11		void run ()	描述在线程中执行的算法
12		void setPriority (int newPriority)	设置线程优先级
13		int getPriority ()	获取线程优先级
14		void setName (String name)	设置线程名
15		String getName ()	获取线程名
16		long getId ()	获取线程号
17		boolean isAlive ()	检查线程是否还未结束
18		void interrupt ()	请求中止线程
19		Thread.State getState ()	获取线程状态
20		void setDaemon (boolean on)	设置后台（守护）线程
21		boolean isDaemon ()	检查是否后台线程
.....			



8.2 多线程编程及并发调度

- 线程

- 一个多线程并发程序包含多个线程，其中一个是系统自动创建的主线程，其余的则是由程序员创建的子线程
- 每个线程都有线程号（ID）、线程名称（name）、优先级（priority）和状态（state）等属性
- 调用线程对象的start()方法启动线程，计算机将在线程中执行算法对象的run()方法。该线程与所在进程的其他线程（包括主线程）一起并发执行
- 当执行完算法对象run()方法中的所有代码，或遇到return语句中途退出时，线程即宣告结束
- 线程中的算法对象可以在执行过程中响应外部中断请求，并使用return语句退出run()方法，结束线程
- 当程序进程中的主线程和所有子线程都执行结束后，则退出程序，进程也随之结束
- 可以将子线程设置为守护（daemon）线程，或称后台线程。只有当主线程及所有非守护子线程结束后，进程才会结束。进程结束时，守护线程会自动结束



```

17 27 class SubDaemon implements Runnable { // 算法类 2
18 28     public void run() { // 描述算法的方法 run()
19 29         int count = 1; // 显示计数
20 30         while (true) { // 循环显示信息，死循环
21 31             System.out.println("\t SubDaemon ..." + count); count++;
22 32             try { // 捕获并处理可能抛出的勾选异常
23 33                 Thread.sleep(30); // 休眠 0.03 秒
24 34             }
25 35             catch (InterruptedException e) { // 捕捉并处理异常
26 36                 System.out.println( e.getMessage() ); return;
27 37             }
28 38         }
29 39     } }

```

```

8     int count = 1; // 显示计数
9     while (count <= 5) { // 循环显示 5 次信息
10         System.out.println("Main ..." + count);
11     }
12     if ( t1.isAlive() ) // 如果子线程 t1 还未运行
13         t1.interrupt(); // 请求中止子线程 t
14     // 子线程 t2 是守护线程，在进程结束时将
15 } }
16

```

Problems @ Javadoc Declaration Console

```

<terminated> JThreadTest [Java Application] C:\Java\j
Main ...1
        SubDaemon ...1
        Sub ...1
Main ...2
        Sub ...2
Main ...3
Main ...4
Main ...5
        Sub ...3

```

8.2 多线程编程及并发调度

- 线程

- 定义算法类时直接继承线程类**Thread**

例8-4 一个通过继承线程类Thread来定义算法类的Java演示程序（JSubThreadTest.java）

```
1 public class JSubThreadTest {           // 主类
2     public static void main(String[] args) { // 主方法
3         SubThread t = new SubThread(); // 创建包含算法的线程对象t
4         t.start();                      // 启动线程，运行其中的算法
5     }
6 }
7
8 class SubThread extends Thread { // 通过继承线程类Thread来定义算法类
9     public void run() { // 重写run()方法，编写需在线程中运行的算法代码
10         System.out.println("Hello from a thread!");
11     }
12 }
```



8.2 多线程编程及并发调度

- 多线程的并发调度
 - 多线程并发程序在执行时，只有拿到**CPU控制权**的线程才能执行
 - Java虚拟机**负责线程的管理和执行调度
 - 程序员**无法掌控线程在什么时候执行，也很难精确控制哪个线程先执行，哪个后执行
 - 线程的**5种**状态

java.lang.Thread.State内部类说明文档			
public static enum Thread.State extends Enum<Thread.State>			
	修饰符	枚举常量（节选）	功能说明
1		NEW	已创建但还未启动的线程
2		RUNNABLE	处于可运行（就绪）状态的线程
3		BLOCKED	处于阻塞状态的线程
4		WAITING	处于等待状态的线程
5		TIMED_WAITING	处于定时等待状态的线程
6		TERMINATED	已执行结束的线程
.....			



8.2 多线程编程及并发调度

- 多线程的并发调度
 - 线程对象的优先级
 - 线程优先级从1到10
 - 1级为**最低**优先级，10级为**最高**优先级，线程**默认**优先级是5级
 - 在线程中创建另一个新线程，新线程具有与当前线程相同的优先级
 - 线程对象可以通过方法成员**setPriority()**修改优先级，或通过**getPriority()**获得自己当前的优先级
 - 多线程与CPU
 - 单核CPU：分时技术，**并发**执行各个线程
 - 多核或多CPU：在不同的运行核或CPU上运行线程，**并行**执行



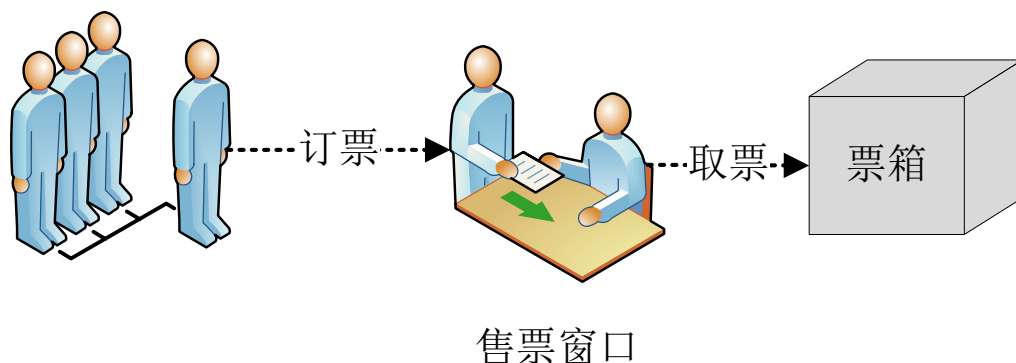
8.3 多线程之间的并发与互斥

- 多线程并发程序的各个算法之间虽然是**独立执行的**，但它们可能需要**共享数据**，这样才能实现多线程协同工作
 - 为什么需要使用多线程并发？
 - 并发时为什么要共享数据？
 - 多线程共享数据时需要考虑哪些问题？



8.3 多线程之间的并发与互斥

- 一个单线程售票服务程序







— 售票窗口的服务流程

- **售票准备**。向客户询问出发日期、目的地等购票信息
- **售票**。查看票箱的剩余票数，如果有余票，则显示售票成功，并将剩余票数减一；否则显示无票，售票失败



例 8-5 一个单线程串行的 Java 售票服务演示程序 (JTicketST.java)

<pre> 1 public class JTicketST { 2 public static void main(String[] args) { 3 TicketBox tBox = new TicketBox(); 4 TicketWindow tw = new TicketWindow(tBox); 5 // 模拟售票前的准备工作 6 long start = System.currentTimeMillis(); 7 for (int i = 0; i < 10; i++) { 8 // 模拟售票前的准备工作 9 long end = System.currentTimeMillis(); 10 System.out.println("准备时间: " + (end - start)); 11 } 12 } 13 class TicketBox { 14 private int tickets = 10; 15 public TicketBox() { 16 tickets = 10; 17 } 18 } 19 </pre>	<pre> 20 class TicketWindow { // 提供售票服务 21 private TicketBox tBox; // 从 TicketBox 中获取票数 22 public TicketWindow(TicketBox p) // 构造 23 { tBox = p; } 24 public void prepare() { // 模拟售票前的准备工作 25 System.out.println(Thread.currentThread().getName() + 26 " 开始售票前的准备工作"); 27 Thread.sleep(100); // 休眠 100ms 28 } 29 catch (InterruptedException e) // 捕捉 sleep() 方法可能抛出的异常 30 { System.out.println(e.getMessage()); return; } 31 } 32 public void sale() { // 具体的售票算法 33 int tickets = tBox.get(); // 读取剩余票数 34 if (tickets > 0) { // 如果有票 35 tickets--; // 售出一张票, 将剩余票数减一 36 tBox.set(tickets); // 设置票箱的剩余票数 37 System.out.println(Thread.currentThread().getName() + 38 ": 成功, 剩余票数 " + tickets); 39 } 40 else System.out.println(Thread.currentThread().getName() + ": 无票"); // 无票 41 } 42 public void serviceAlgorithm() { // 描述完整售票服务流程的算法 43 prepare(); // 模拟售票前的准备工作 44 sale(); // 模拟售票 45 } 46 } </pre>	<div>  Problems  Javadoc  Declaration  Console </div> <pre> <terminated> JTicketST [Java Application] C:\Java\jre main: 购票前准备 ... main: 成功, 剩余票数 3 main: 购票前准备 ... main: 成功, 剩余票数 2 main: 购票前准备 ... main: 成功, 剩余票数 1 main: 购票前准备 ... main: 成功, 剩余票数 0 main: 购票前准备 ... main: 无票 用时: 0.503秒 </pre>
--	--	--

例 8-6 一个多线程并发的 Java 售票服务演示程序 (JTicketM

```

1 public class JTicketMT { // 多线程并发的售票演示
2     public static void main(String[] args) {
3         TicketBox tb = new TicketBox(4);
4         TicketWindow tw = new TicketWindow(t
5         // 模拟 5 个售票窗口同时向 5 位客户
6         Thread t1 = new Thread(tw, "窗口 1");
7         Thread t2 = new Thread(tw, "窗口 2");
8         Thread t4 = new Thread(tw, "窗口 4");
9         // 启动 5 个子线程, 5 个售票窗口同时

```

Problems @ Javadoc Declaration Console

<terminated> JTicketMT [Java Application] C:\Java\jre

窗口4: 购票前准备 ...
 窗口1: 购票前准备 ...
 窗口2: 购票前准备 ...
 窗口3: 购票前准备 ...
 窗口5: 购票前准备 ...
 窗口2: 成功, 剩余票数 3
 窗口3: 成功, 剩余票数 2
 窗口4: 成功, 剩余票数 3
 窗口1: 成功, 剩余票数 3
 窗口5: 成功, 剩余票数 3
 用时: 0.101秒

多线程能提高程序效率
 多线程之间存在互斥操作

```

14 24
15 25 class TicketWindow { // 提供售票服务的售票窗口类
16 26 class TicketWindow implements Runnable { // 实现 Runnable 接口才能在线程中运行
17 27     ..... // 此处同例 8-5 中的代码第 21~41 行, 省略代码
18 28 public void serviceAlgorithm() { // 描述完整售票服务流程的算法
19 29     public void run() { // 必须实现 run()方法, 描述可在线程中运行的售票服务算法
20 30         prepare(); // 模拟售票前的准备工作
21 31         sale(); // 模拟售票
22 32     } }

```

8.3 多线程之间的并发与互斥

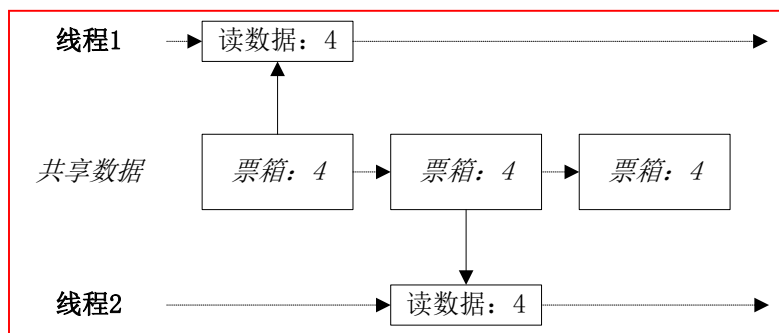
- 多线程中的互斥操作
 - 多线程并发程序包含**多个**线程，每个线程运行一个算法
 - 执行时各线程轮流**切换**，并发执行算法
 - 如果两个线程中的算法不能**重叠交叉**执行，则这两个算法被称为是**互斥**操作
 - 什么样的操作是互斥操作？
 - 如果多个线程**共享数据**，则在不同线程中**同时**访问共享数据就可能**是互斥**操作
 - 多线程售票服务程序
 - 在5个线程中同时运行售票服务算法，从同一票箱获取票务数据
 - 票务数据就是一个被多线程共享的数据



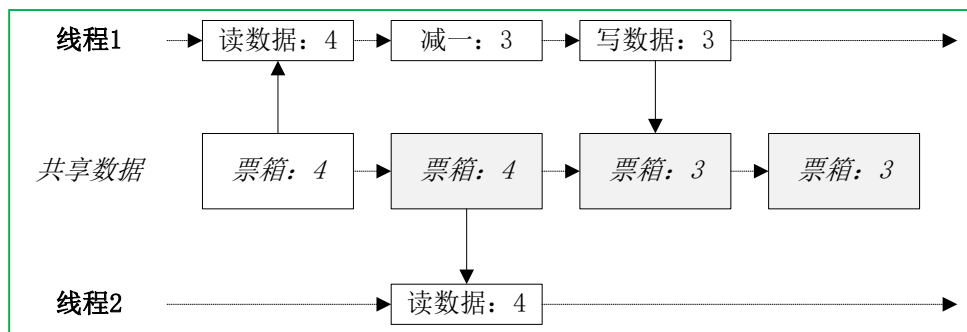
8.3 多线程之间的并发与互斥

- 3种并发访问形式

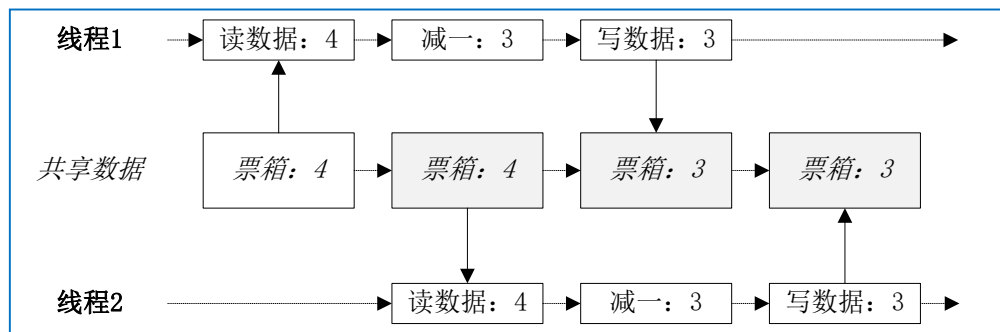
- 读-读



- 改-读



- 改-改



8.3 多线程之间的并发与互斥

- 多线程中的互斥操作
 - 多线程并发程序需要对不同线程中运行的互斥操作做特殊处理，否则会产生错误的运行结果
 - 产生错误的原因在于，本来不能重叠交叉执行的互斥操作会因线程切换而出现了重叠交叉
 - 这种现象在单线程串行程序中不会发生，它是多线程并发程序所特有的问题
 - Java语言提供了一种同步（synchronization）机制



8.3 多线程之间的并发与互斥

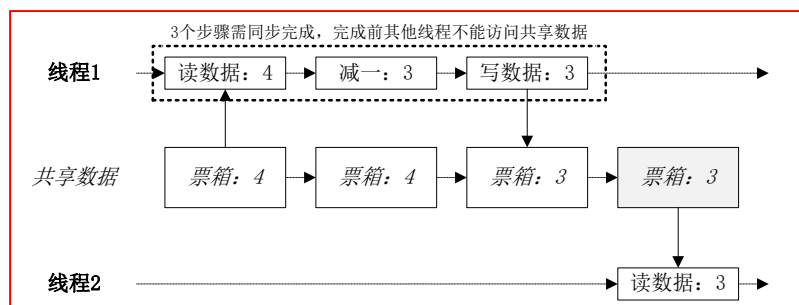
- 多线程中的互斥操作
 - 对互斥操作算法进行**同步**
 - 使用关键字**synchronized**将互斥操作算法定义成一个**同步方法**（synchronized method）
 - 一旦某个线程中的同步方法开始执行，所有其他线程中与该同步方法互斥的方法都不会被执行，直到同步方法所包含的指令序列执行结束，这就是Java语言里的**同步机制**
 - 同步方法所包含的指令序列必须**一次性**同步执行完成，其执行过程不会因线程切换而被其他线程里的互斥方法中途**打断**



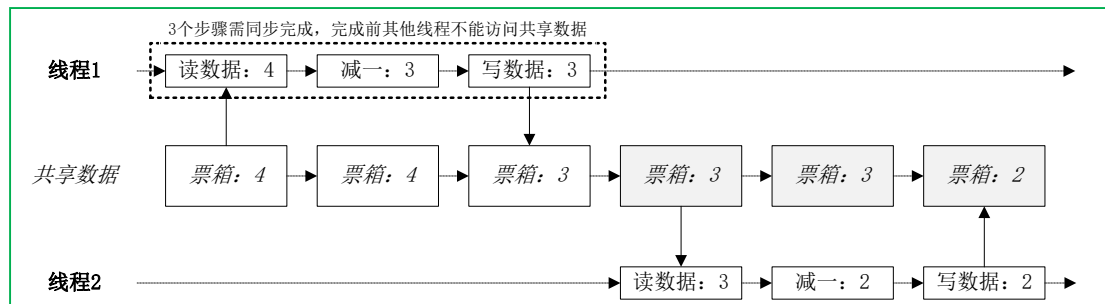
8.3 多线程之间的并发与互斥

- 多线程中的互斥操作
 - 对互斥操作算法进行同步

— 改-读



— 改-改



中國農業大學

閻道宏

8.3 多线程之间的并发与互斥

- 多线程中的互斥操作

- 将互斥操作算法**定义**成同步方法

synchronized 方法类型 方法名

..... // 此处编写互斥操作的

}



```
<terminated> JTicketMT [Java Application] C:\Java\jre
窗口2: 购票前准备 ...
窗口1: 购票前准备 ...
窗口4: 购票前准备 ...
窗口5: 购票前准备 ...
窗口3: 购票前准备 ...
窗口5: 成功, 剩余票数 3
窗口1: 成功, 剩余票数 2
窗口2: 成功, 剩余票数 1
窗口3: 成功, 剩余票数 0
窗口4: 无票
用时: 0.101秒
```

- 为多线程售票服务程序添加同步机制

- 对售票窗口类**TicketWindow**中的售票方法**sale()**进行同步
 - 定义售票方法**sale()**时增加关键字**synchronized**, 将其定义为同步方法



例8-7 对售票方法sale()进行同步的售票窗口类TicketWindow

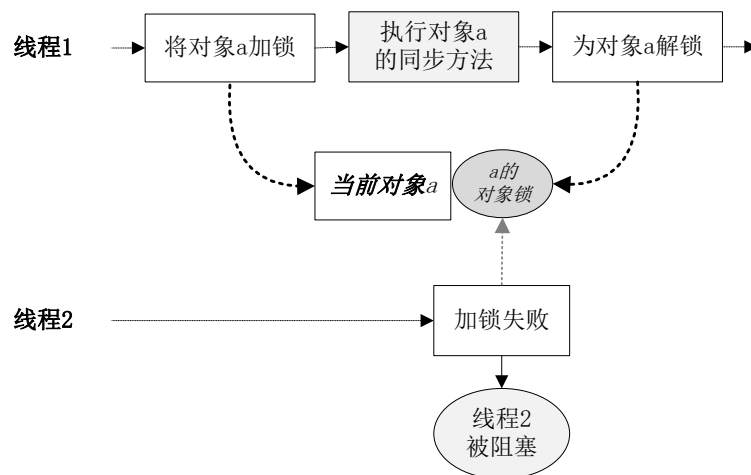
```
1 class TicketWindow implements Runnable { // 可运行的售票窗口类TicketWindow
2     private TicketBox tBox;          // 从票箱tBox取票
3     public TicketWindow(TicketBox p)    // 构造方法
4     { tBox = p; }
5     public void prepare() { // 模拟售票前的一些准备工作，例如询问出发日期、目的地等
6         System.out.println(Thread.currentThread().getName() + ": 购票前准备 ...");
7         try {
8             Thread.sleep(100); // 休眠（暂停）0.1秒，模拟购票前的准备工作
9         }
10        catch(InterruptedException e) // 捕捉sleep方法可能抛出的异常
11        { System.out.println( e.getMessage() ); return; }
12    }
13    // public void sale() { // 同步之前的售票方法
14    public synchronized void sale() { // 同步之后的售票方法
15        int tickets = tBox.get(); // 读取剩余票数
16        if (tickets > 0) { // 如果有票
17            tickets --; // 售出一张票，将剩余票数减一
18            tBox.set(tickets); // 设置票箱的剩余票数
19            System.out.println(Thread.currentThread().getName() +
20                               ": 成功，剩余票数 " +tickets);
21        }
22        else System.out.println(Thread.currentThread().getName() + ": 无票"); // 无票
23    }
24    public void run() { // 描述可在线程中运行的售票窗口算法run()
25        prepare(); // 模拟售票前的准备工作
26        sale(); // 模拟售票
27    }
28 }
```



8.3 多线程之间的并发与互斥

- Java同步机制的实现原理

- Java语言通过**同步机制**来控制多线程并发执行互斥操作时不会出现重叠交叉
- Java语言是通过对**当前对象**进行**加锁-解锁**来实现同步机制的



- 什么是对象锁？
- 被锁定的当前对象是哪个对象？
 - 如果线程所运行的同步方法属于**对象a**，则对象a就是被锁定的当前对象
 - 如果两个线程中运行的同步方法属于**同一个对象**，则两个线程锁定的就是同一个对象
 - 如果两个线程运行的同步方法属于**不同对象**，则两个线程锁定的就是不同对象



8.3 多线程之间的并发与互斥

- Java同步机制的实现原理
 - 同步语句（synchronized statement）

```
synchronized( 对象名 ) {  
    ..... // 此处编写互斥操作的算法代码  
}  
  
// public synchronized void sale() { // 将售票算法定义成一个同步方法  
public void sale() { // 改用同步语句对售票算法进行同步  
    synchronized( this ) { // 对当前对象（即该方法所属的售票窗口对象）加锁  
        int tickets = tBox.get(); // 在大括号中编写需要同步的售票算法代码  
        if (tickets > 0) { // 有票有票  
            tickets --; // 售出一张票，将剩余票数减一  
            tBox.set(tickets); // 设置票箱的剩余票数  
            System.out.println(Thread.currentThread().getName() + ": 成功，剩余票数 " + tickets);  
        }  
        else System.out.println(Thread.currentThread().getName() + ": 无票"); // 无票  
    }  
}
```



8.3 多线程之间的并发与互斥

- Java同步机制的实现原理
 - 同步语句（synchronized statement）
 - 同步语句可锁定任何对象
 - 同步方法只能通过锁定**当前对象**来实现同步
 - 理论上，同步语句可以锁定**任何对象**来实现同步
 - 如果多线程共享数据，同步语句一般是通过锁定被**共享**的数据对象来实现同步的
 - 同步语句可实现更细粒度的并发控制
 - 同步方法所同步的是**整个方法**，即方法中的所有指令
 - 同步语句可以只对方法中的**部分指令**进行同步



8.3 多线程之间的并发与互斥

- Java同步机制的实现原理

- 对共享数据加锁

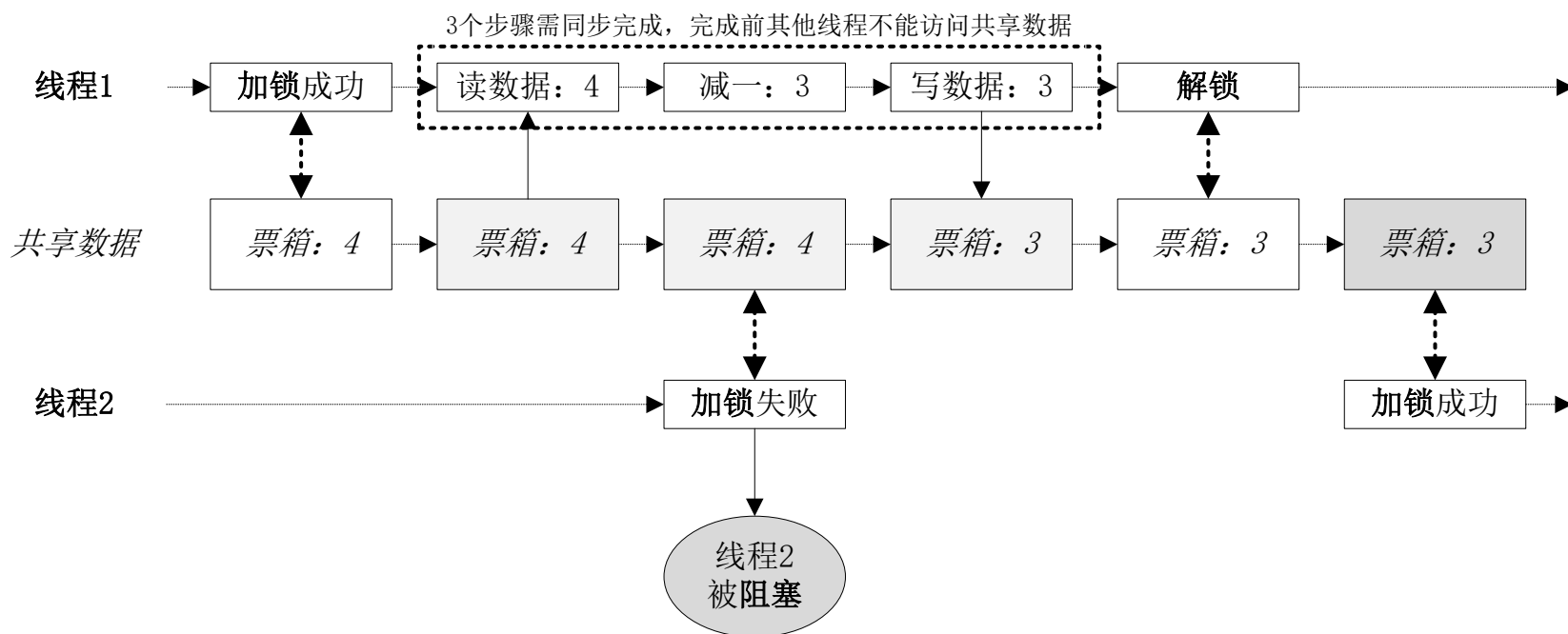
- 数据虽然能被多线程共享，但往往不能被同时操作，这是多线程互斥操作产生的主要原因
 - 使用同步语句直接锁定被共享的数据对象，这样可以对处理数据的算法进行同步，避免多线程同时操作共享数据
 - 多线程售票服务程序：锁定票箱对象tBox

```
public void sale() {    // 使用同步语句对售票算法进行同步    // synchronized( this ) { // 修改前：对当前对象加锁    synchronized( tBox ) { // 修改后：直接锁定票箱对象tBox，对售票算法进行同步        .....    // 售票算法从tBox中取票。代码保持不变，此处省略    }    }
```
 - 直接锁定共享数据，然后对处理算法进行同步，这样所编写出的程序在逻辑上更加清晰，易于理解



8.3 多线程之间的并发与互斥

- Java同步机制的实现原理
 - 对共享数据加锁



例8-8 一个典型的Java多线程并发数据处理程序代码框架

```
1 class Data1 { ..... } // 描述待处理数据1的类
2 class Data2 { ..... } // 描述待处理数据2的类
3
4 class Algorithm implements Runnable { // 描述数据处理算法的算法类
5     private Data1 d1; // 指向待处理的数据1
6     private Data2 d2; // 指向待处理的数据2
7     private void Algorithm(Data1 p1, Data2 p2) { // 构造方法
8         { d1 = p1; d2 = p2; } // 初始化待处理的数据
9     private void process1() { // 数据1的处理算法
10         synchronized( d1 ) // 对处理数据1的算法代码进行同步
11         { ..... } // 具体的算法代码
12     }
13     private void process2() { // 数据2的处理算法
14         synchronized( d2 ) // 对处理数据2的算法代码进行同步
15         { ..... } // 具体的算法代码
16     }
17     public void run() { // 实现run()方法，描述可在线程中运行的数据处理算法
18         process1(); // 处理数据1
19         process2(); // 处理数据2
20     }
21 }
22
23 public class JDataProcessingMT { // 多线程并发数据处理程序的主类
24     public static void main(String[] args) { // 主方法
25         Data1 dObj1 = new Data1(); // 创建待处理的数据对象1
26         Data2 dObj2 = new Data2(); // 创建待处理的数据对象2
27         Algorithm a = new Algorithm(dObj1, dObj2); // 创建算法对象a
28         // 创建多个子线程，同时执行算法对象a，实现多线程并发数据处理
29         Thread t1 = new Thread( a ); Thread t2 = new Thread( a ); .....
30         t1.start(); t2.start(); ..... // 启动子线程，开始并发执行
31     } }
```



8.4 多线程之间的协同

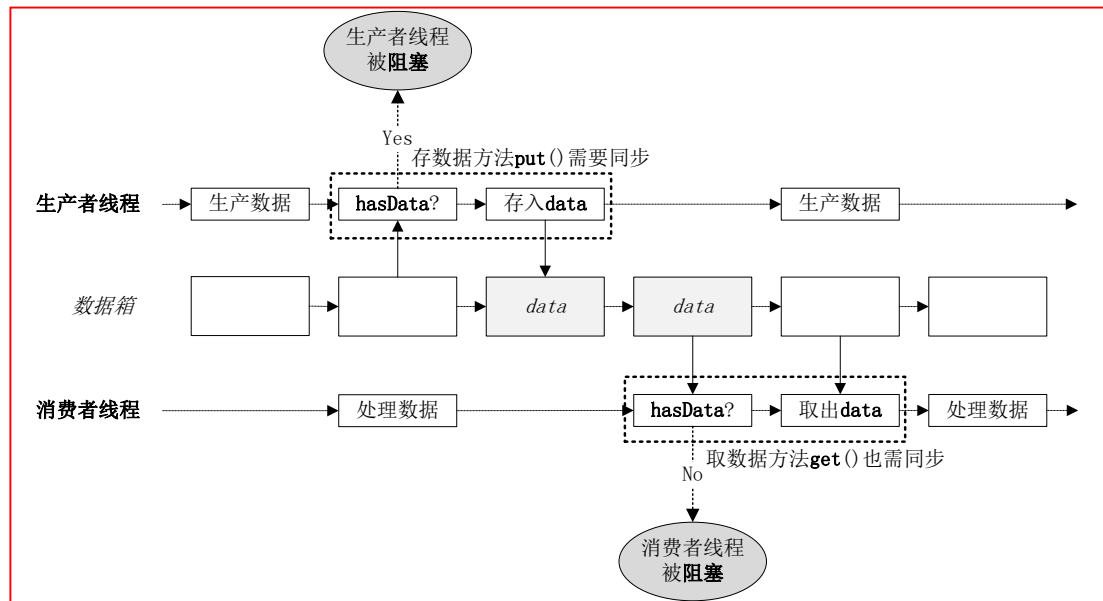
- 数据“**采集-处理**”系统
 - **处理模块**等待采集模块提交数据，如果发现有数据则取出处理，否则进入等待状态
 - **采集模块**检查自己提交的数据是否已被处理，如果已处理则提交下一个数据，否则也进入等待状态
- 类似的系统还有业务“**申请-审批**”系统、订单“**下单-处理**”系统等
- 可以将这类系统抽象成一种“**生产者-消费者**”（producer-consumer）数据处理模式



8.4 多线程之间的协同

- “生产者-消费者”模式

对**数据箱**的操作需要同步
对数据箱的操作顺序是“**先存后取，取完再存**”



8.4 多线程之间的协同

- 编写多线程“**生产者-消费者**”模式数据处理程序
 - 需要**同步**线程间的互斥操作
 - 控制各线程的运行**次序**
- 多线程之间的**协同**（coordination）
- Java语言在同步机制的基础上又增加了一种**等待-唤醒**（wait-notify）机制



8.4 多线程之间的协同

- 守护代码块
 - 只有当数据箱中有数据时，消费者才能去取数据
 - 在设计取数据方法`get()`时，需首先检查“数据箱中有数据”这个条件是是否成立。如果条件不成立则应持续检查直至条件成立，待条件成立后再取出数据箱中的数据
 - 取数据方法`get()`的算法代码结构

```
int get() {           // 从数据箱取数据的方法
    while (hasData != true) {} // 循环检查条件“数据箱中有数据”，直至条件成立
    return data;        // 条件成立后，取出并返回数据箱中的数据data
}
```
 - 持续检查算法条件直至条件成立，然后再进行后续处理，这样的算法代码块被称为**守护代码块**（`guarded block`，或称为**警戒代码块**）
 - 同理，向数据箱存数据方法`put()`的算法代码也是一个守护代码块结构



8.4 多线程之间的协同

- 守护代码块

例8-9 一个具有同步机制的数据箱类DataBox示例代码（DataBox.java）

```
1 class DataBox { // 存放共享数据的数据箱类
2     private boolean hasData = false; // 数据标记：标记数据箱里的数据是否有效
3     private int data = 0; // 数据
4
5     public synchronized void put(int x) { // 生产者调用该方法，存入数据x
6         while (hasData == true) { } // 如已有数据则空转等待，直到消费者将它取走
7         data = x; hasData = true; // 存入数据x，设置数据标记为true（有效）
8     }
9
10    public synchronized int get() { // 消费者调用该方法，取出数据
11        while (hasData != true) { } // 如没有数据，则空转等待，直到生产者存入数据
12        int x = data; hasData = false; // 取出数据，设置数据标记为false（无效）
13        return x; // 返回所取出的数据
14    }
15 }
```

数据标记 **hasData**
同步方法中的“**空转等待**”：死锁（deadlock）



8.4 多线程之间的协同

- Java等待-唤醒机制

- 空转等待
- 阻塞等待

在同步方法中使用阻塞等待，当前线程会释放对象锁，然后暂停执行并进入阻塞状态，CPU转去执行其他线程；被阻塞的线程需要由其他线程唤醒，唤醒后再继续执行，这就是Java语言的“等待-唤醒”机制。

- 阻塞等待wait和阻塞唤醒notifyAll

- 根类Object

- 阻塞等待方法wait(): 阻塞状态与对象锁
- 阻塞唤醒方法notifyAll(): 被唤醒后的阻塞线程转入就绪状态



例8-10 一个具有同步机制和等待-唤醒机制的数据箱类DataBox示例代码（DataBox.java）

```
1 class DataBox {           // 存放共享数据的数据箱类
2     private boolean hasData = false; // 数据标记：标记数据箱里的数据是否有效
3     private int data = 0;      // 数据
4
5     public synchronized void put(int x) {    // 生产者调用该方法，存入数据x
6         while (hasData == true) { // 如已有数据则阻塞等待，直到消费者将它取走
7             try { // 方法wait()可能会抛出勾选异常InterruptedException
8                 wait(); // 当前的生产者线程被阻塞，等待被消费者线程唤醒
9             } catch (InterruptedException e) {}
10        }
11        data = x; hasData = true; // 存入数据x，设置数据标记为true（有效）
12        notifyAll(); // 唤醒被阻塞的消费者线程，通知它们可以取数据了
13    }
14
15    public synchronized int get() { // 消费者调用该方法，取出数据
16        while (hasData != true) { // 如没有数据，则阻塞等待，直到生产者存入数据
17            try { // 方法wait()可能会抛出勾选异常InterruptedException
18                wait(); // 当前的消费者线程被阻塞，等待被生产者线程唤醒
19            } catch (InterruptedException e) {}
20        }
21        int x = data; hasData = false; // 取出数据，设置数据标记为false（无效）
22        notifyAll(); // 唤醒被阻塞的生产者线程，通知它们可以生产数据了
23        return x;
24    }
25 }
```

阻塞等待**wait()**可能会抛出勾选异常**InterruptedException**
阻塞等待-唤醒中**wait()**和**notifyAll()**的对应关系



8.4 多线程之间的协同

- Java等待-唤醒机制
 - 线程安全类
 - 综合运用Java语言的**同步**机制和**等待-唤醒**机制
 - 并发访问线程安全类的对象时**不需要**再添加Java语言的同步机制或等待-唤醒机制
 - Java API可能会有“线程安全”和“非线程安全”两个版本
 - 字符串类
 - 线程安全版本：**String**、StringBuffer
 - 非线程安全版本：**StringBuilder**
 - 集合类
 - 线程安全版本：Vector、Hashtable
 - 非线程安全版本：**ArrayList**、**LinkedList**、**HashSet**、**HashMap**等
 - 开发单线程程序应选用**非线程安全**的版本



例 8-

1

2

3

4

5

6

7

8

9

10

```
11 class Producer implements Runnable { // 生产者算法
12     private DataBox dBox; // 存放数据的容器
13     // 以下为模拟的原始数据，将被依次存入数据箱
14     private int[] x = { 1, 3, 5, 7, 9, -1 }; // -1: 数据结束标志
15     public Producer(DataBox d) // 构造方法
16     {
17         dBox = d;
18     }
19     public void run() { // 模拟生产的算法
20         while (true) {
21             int x = dBox.get(); // 从数据箱中取出一个数据，计算其平方值
22             if (x == -1) { // -1: 数据结束标志
23                 System.out.println( "\t" + Thread.currentThread().getName() + ": 数据结束");
24                 return;
25             }
26             System.out.println( "\t" + Thread.currentThread().getName() + ": " + x*x );
27             try {
28                 Thread.sleep(100); // 休眠（暂停）0.1 秒，模拟复杂的数据处理过程
29             } catch (InterruptedException e) {}
30         }
31     }
32 }
33
34 class Consumer implements Runnable { // 消费者算法
35     private DataBox dBox; // 存放数据的容器
36     public Consumer(DataBox d) // 构造方法
37     {
38         dBox = d;
39     }
40     public void run() { // 模拟消费的算法
41         while (true) {
42             int x = dBox.get(); // 从数据箱中取出一个数据，计算其平方值
43             if (x == -1) { // -1: 数据结束标志
44                 System.out.println( "\t" + Thread.currentThread().getName() + ": 数据结束");
45                 return;
46             }
47             System.out.println( "\t" + Thread.currentThread().getName() + ": " + x*x );
48             try {
49                 Thread.sleep(100); // 休眠（暂停）0.1 秒，模拟复杂的数据处理过程
50             } catch (InterruptedException e) {}
51         }
52     }
53 }
```

Problems @ Javadoc Declaration Console

<terminated> JProducerConsumer [Java Application]

```
Producer: 1
Consumer: 1
Producer: 3
Consumer: 9
Producer: 5
Consumer: 25
Producer: 7
Consumer: 49
Producer: 9
Consumer: 81
Producer: -1
Consumer: 数据结束
```


8.5 定时执行的线程

- 本地日期时间类LocalDateTime

java.time.LocalDateTime类说明文档			
public final class LocalDateTime			
extends Object			
implements Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable			
	修饰符	类成员（节选）	功能说明
1	static	LocalDateTime now ()	获取表示本地当前时间的对象
2	static	LocalDateTime of (int year, int month, int dayOfMonth, int hour, int minute, int second)	创建一个本地时间对象
3	static	LocalDateTime parse (CharSequence text)	创建一个本地时间对象
4		int getYear ()	获取年（月、日）
5		int getHour ()	获取时（分、秒）
6		LocalDateTime plusYears (long years)	增加年（月、日）
7		LocalDateTime plusHours (long seconds)	增加时（分、秒）
8		LocalDateTime minusYears (long years)	减少年（月、日）
9		LocalDateTime minusHours (long seconds)	减少时（分、秒）
10		int compareTo (ChronoLocalDateTime<?> other)	比较时间的先后
.....			



8.5 定时执行的线程

- 本地日期时间类LocalDateTime

例8-12 一个本地日期时间类LocalDateTime的Java演示程序（JLocalDateTimeTest.java）

```
1 import java.time.*;      // 导入java.time包中与时间相关的类
2 public class JLocalDateTimeTest { // 测试类：测试本地日期时间类LocalDateTime
3     public static void main(String[] args) { // 主方法
4         LocalDateTime t = LocalDateTime.now(); // 获取本地计算机系统的当前时间
5         System.out.println( t );           // 显示当前时间
6         System.out.println( "getYear():    " +t.getYear() );    // 年
7         System.out.println( "getMonthValue(): " +t.getMonthValue() ); // 月
8         System.out.println( "getDayOfMonth(): " +t.getDayOfMonth() ); // 日
9         System.out.println( "getHour():     " +t.getHour() );    // 时
10        System.out.println( "getMinute():   " +t.getMinute() );   // 分
11        System.out.println( "getSecond():   " +t.getSecond() );   // 秒
12    } }
```



8.5 定时执行的线程

- 定时执行的线程
 - 定时任务类**TimerTask**和定时器类**Timer**
 - 定时任务类**TimerTask**。定时任务类TimerTask实现了**Runnable**接口。它是一个**算法类**，用于描述定时执行的操作任务
 - 定时器类**Timer**。用于**创建**定时执行的守护线程（即后台线程），并在其中**执行**定时任务类TimerTask的算法对象



8.5 定时执行的线程

- 定时执行的线程

例8-13 一个定时显示本地系统时间的Java演示程序（JTimerTest）

```
1 import java.util.*; // 导入java.util包中的类（其中包括类
2 import java.time.*; // 导入java.time包中与时间相关的类
3
4 class ShowTime extends TimerTask { // 继承TimerTask定
5     public int count = 0; // 记录被定时执行的次数
6     public void run() { // 实现run()方法，编写需要定时执
7         count++;
8         System.out.println( count +": " +LocalDateTime.now() ); // 显示系统时间
9     } }
10
11 public class JTimerTest { // 测试类：定时执行某个程序任务
12     public static void main(String[] args) { // 主方法
13         Timer t = new Timer("Show time"); // 创建一个定时器类Timer的对象t
14         ShowTime st = new ShowTime(); // 创建一个显示时间的定时任务对象st
15         t.schedule(st, 0, 2000); // 创建后台线程并在其中每隔2秒执行一次任务对象st
16         while (st.count < 5) { // 检查定时执行的次数，执行5次后结束
17             try {
18                 Thread.sleep(1000); // 休眠1秒
19             } catch (InterruptedException e) {}
20         }
21         t.cancel(); // 结束（取消）定时任务
22         System.out.println( "TimerTask stopped" );
23     } }
```

Problems @ Javadoc Declaration Console

<terminated> JTimerTest [Java Application] C:\Java\jre

```
1: 2018-06-19T09:43:35.107
2: 2018-06-19T09:43:36.982
3: 2018-06-19T09:43:38.985
4: 2018-06-19T09:43:40.988
5: 2018-06-19T09:43:42.992
TimerTask stopped
```



8.5 定时执行的线程

- 定时执行的线程

java.util.TimerTask类说明文档			
public abstract class TimerTask extends Object implements Runnable			
	修饰符	类成员（节选）	功能说明
1	protected	TimerTask()	构造方法
2	abstract	void run()	定义需要定时执行的算法代码
3		void cancel()	取消定时任务

java.util.Timer类说明文档			
public class Timer extends Object			
	修饰符	类成员（节选）	功能说明
1		Timer()	构造方法
2		Timer(String name)	构造方法
3		void schedule (TimerTask task, long delay, long period)	创建一个定时执行程序任务task的守护线程
4		void cancel()	终止定时线程



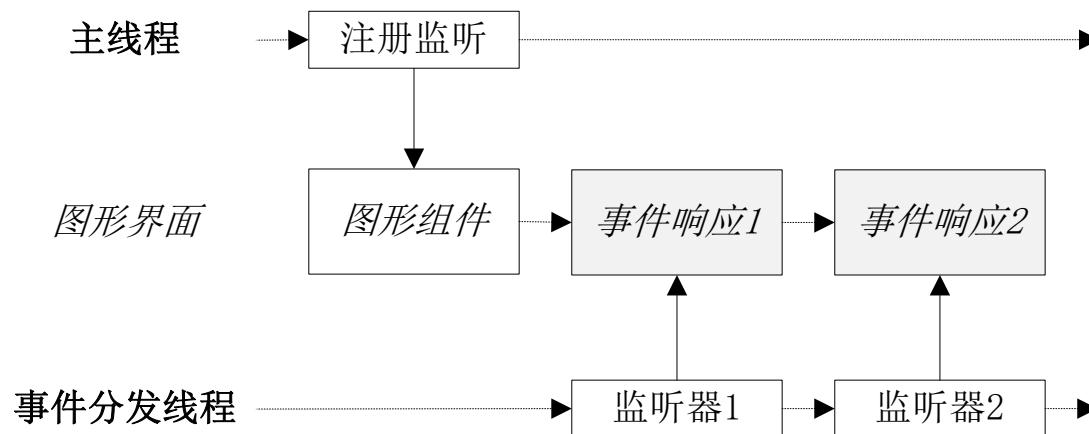
8.6 Swing框架中的线程

- 事件分发线程
 - Swing框架在内部会用到一个重要的线程，这就是**事件分发线程**（event dispatch thread）
 - 所有响应组件事件的监听器算法代码都会在事件分发线程中执行
 - 程序首先为界面上的图形组件注册响应事件的监听器
 - 当用户操作图形组件时，Java虚拟机会自动执行其注册监听器中的算法代码，对用户事件进行处理和响应
 - Swing框架将所有监听器算法代码都安排在事件分发线程中运行



8.6 Swing框架中的线程

- 事件分发线程



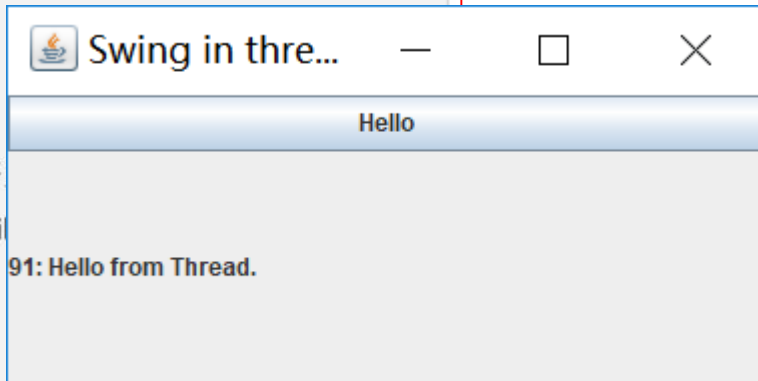
- 在事件分发线程中运行的监听器**算法代码**应能在短时间内完成执行



例 8-14 在线程中操作图形组件的完整 Java 演示代码 (JSwingInThread.java)

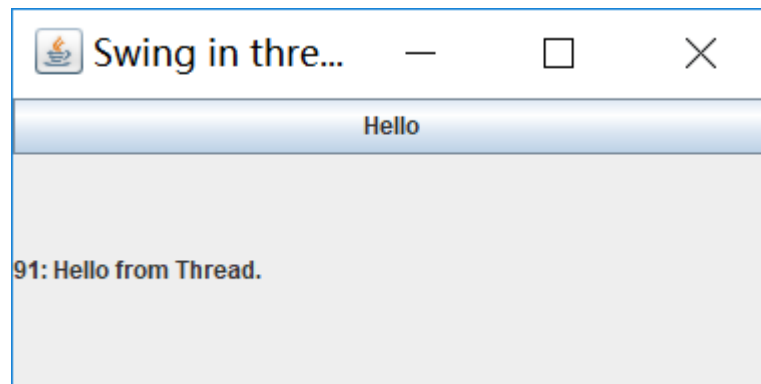
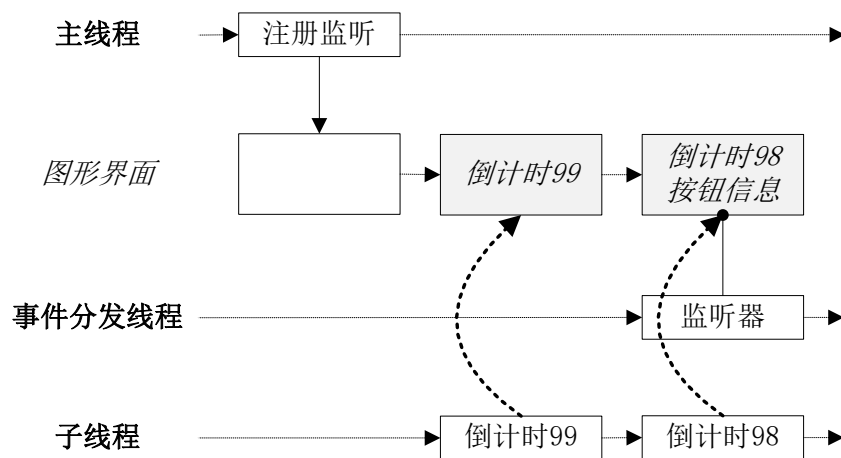
```
1  import java.awt.*;           // 导入 java.awt 包中的类
2  import java.awt.event.*;     // 导入 java.awt.event 包中定义的事件类
3  import javax.swing.*;        // 导入 javax.swing 包中的类
4  public class JSwingInThread {           // 测试类
5      public static void main(String[] args) {           // 主方法
6          JFrame w = new JFrame("Swing in threads"); // 创建程序
7          w.setSize(400, 200); w.setLocation(100, 100); w.setVisible(true);
8          w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9          // 在窗口中添加一个 JButton 按钮, 一个 JLabel 标签
10         JButton btn = new JButton("Hello"); // 创建一个按钮
11         JLabel msg = new JLabel(""); // 创建一个显示信息的标签
12         w.add(msg);
13         w.validate();
14         // 单线程中运行
15         ActionThread t = new ActionThread(msg);
16         t.start();
17         // 等待线程运行完毕
18         t.join();
19         btn.setText(msg.getText());
20         // 下面运行主线程
21         CountDown t2 = new CountDown(msg);
22         t2.start();
23     } }
```

```
25 class CountDown implements Runnable { // 可在线程中运行的倒计时算法类
26     private JLabel msg;                // 显示倒计时信息的标签
27     private int count;                 // 计数器
28     public CountDown(JLabel lab)       // 构造方法
29     { msg = lab; }
30     public void run() {                // 实现 Runnable 接口规定的抽象方法
31         for (int n = 99; n >= 0; n--) { // 倒计时 100 次
32             count = n;
33             msg.setText(count + ": Hello from Thread."); // 子线程: 在标签上显示倒计时信息
34             try { Thread.sleep(500); } // 休眠 0.5 秒
35             catch (InterruptedException e) {}
36         }
37     } }
```



8.6 Swing框架中的线程

- 在线程中操作图形组件



— **Swing框架要求**: 在线程中操作图形组件, 应当将算法代码交由事件分发线程统一执行



8.6 Swing框架中的线程

- 通过事件分发线程操作图形界面
 - 实现**Runnable**接口，将操作图形组件的算法代码**包装**成一个可在线程中运行的**算法对象**
 - 调用javax.swing.**SwingUtilities**类中的静态方法**invokeLater()**，将算法对象**提交**给**事件分发线程**统一调度、执行



— 举例

`msg.setText(count + ": Hello from Thread.");` // 子线程：在标签上显示倒计时信息

```
SwingUtilities.invokeLater( new Runnable() { // 使用匿名类的形式包装算法对象
    public void run() { // 实现接口Runnable规定的抽象方法
        msg.setText(count + ": Hello from Thread."); // 被包装的算法代码
    }
});
```

```
SwingUtilities.invokeLater( ()-> { msg.setText(count + ": Hello from Thread."); } );
```

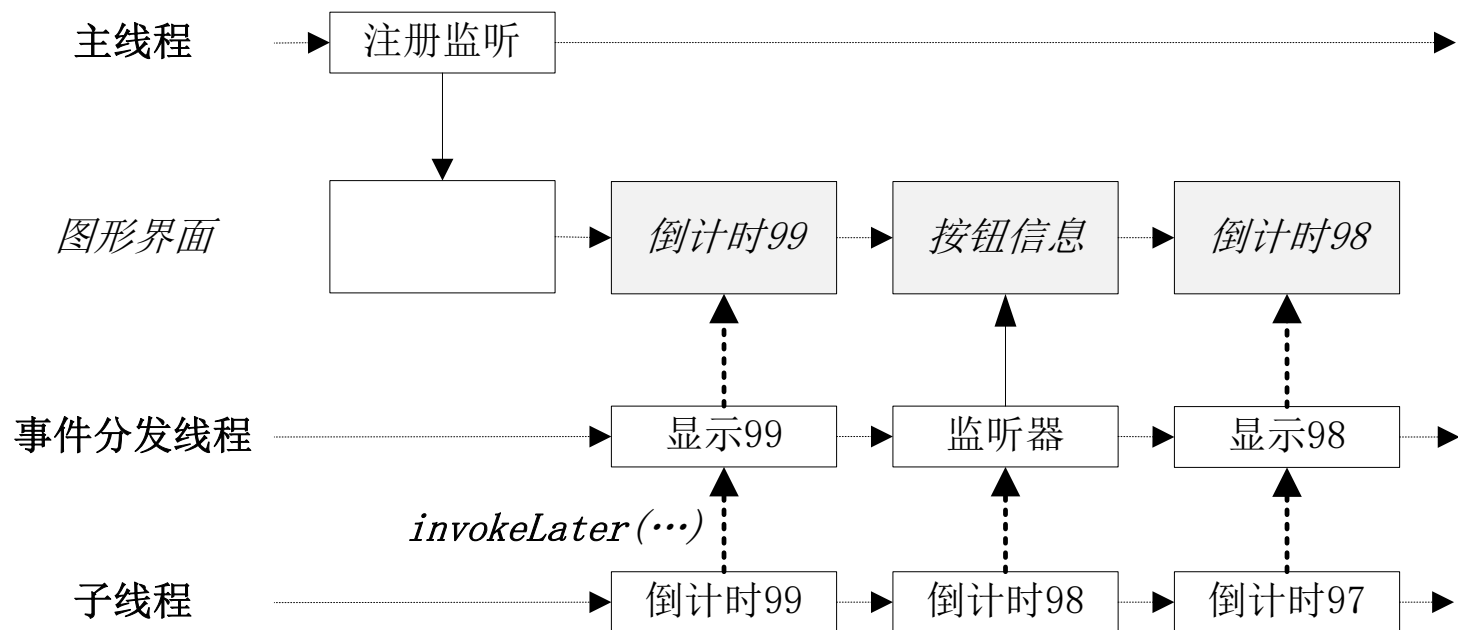


中國農業大學

閻道宏

8.6 Swing框架中的线程

- 通过事件分发线程操作图形界面



8.6 Swing框架中的线程

- 多线程并发绘图
 - 将**绘图算法**包装成可在线程中运行的**算法对象**
 - 调用方法**invokeLater()**将其提交给**事件分发线程**去统一调度、执行
 - 多线程并发绘图程序Java演示代码



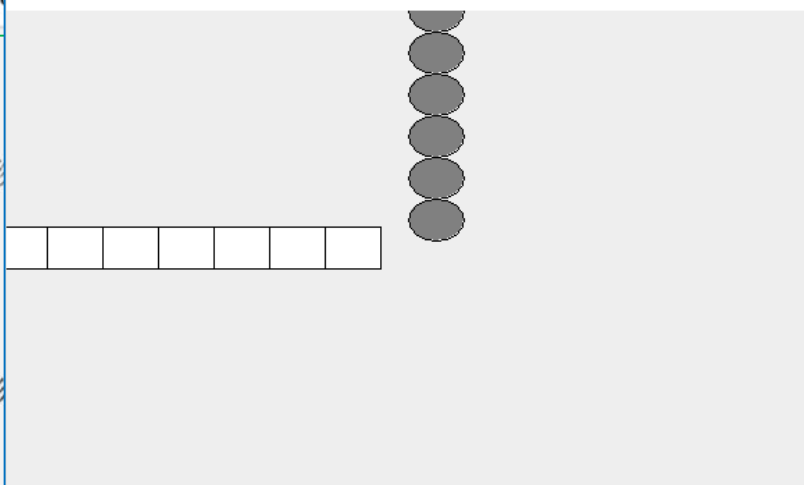
例 8-15 一个完整的多线程并发绘图程序 Java 演示代码 (JDrawInThre

```

1  import java.awt. 24      public void drawShape() {
2  import javax.swing. 25          Graphics g = win.getGraphics();
3  public class JDraw 26          if (shape == 1) { // 绘制长方形
4      public static 27              g.setColor( Color.WHITE );
5          JFrame win = new JFrame( "多线程绘图" ); 28              g.setColor( Color.BLACK );
6          win.setSize( 400, 400 ); 29          }
7          win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); 30          else { // 绘制椭圆形
8          // 创建并启动多线程 31              g.setColor( Color.GRAY );
9          DrawThread dt = new DrawThread( win ); 32              g.setColor( Color.BLACK ); g.drawOval(x, y, w, h);
10         dt.start(); 33          }
11         Thread t1 = dt.getThread(); 34         if (shape == 1) x += w; // 长方形向右移动
12         t1.start(); 35         else y += h; // 椭圆形向下移动
13     } } 36         if (x >= win.getWidth() || y >= win.getHeight() ) isStop = true; // 超出边界时停止绘图
14     class DrawRunnable implements Runnable { 37     }
15         private JFrame win; 38         public void run() { // 实现抽象方法 run(), 描述在线程中运行的绘图算法
16         private int x = 0; 39         while (isStop == false) { // 循环绘图, 直到超出窗口边界
17         private int y = 0; 40             SwingUtilities.invokeLater( ()->{ drawShape(); } ); // 在线程中绘图
18         private boolean isStop = false; 41             try { Thread.sleep(200); } catch (InterruptedException e) {} // 休眠 0.2 秒
19         public DrawRunnable( JFrame win ) { 42         }
20             this.win = win; 43     } }
21         if (shape == 1) { x = 0; y = win.getHeight() / 2; } // 1-长方形: 从左到右绘图
22         else { x = win.getWidth() / 2; y = 0; } // 2-椭圆形: 从上到下绘图
23     }

```

在线程中绘图



第8章 多线程并发编程

- 本章学习要点

- 多线程是一种**高级**编程技术。多线程可以提高CPU使用率，改善用户体验。在多核或多CPU计算机系统中，使用多线程可以明显提高程序的运行速度
- 要准确理解多线程编程中的三个要素
 - 可以运行的**算法对象**，算法对象具有**run()**方法
 - 运行算法对象的**线程对象**，线程对象是**Thread**类的对象
 - 被多个线程共享的**数据对象**，操作这些数据对象时需要启用同步（**synchronized**）机制，多线程协同时还需要使用等待-唤醒（**wait-notify**）机制
- 多线程编程比较复杂，学习时应仔细**阅读并理解**本章提供的**演示程序**，然后尝试自己**重写**一遍

