

第 15 章 多线程编程

本章将要讲述 Java 中线程（Thread）相关的内容。它是一个全新的事物。为了理解本章的内容，需要用到前面学到的以下知识点。

- ❑ 方法的调用过程；
- ❑ 程序执行流程；
- ❑ 继承和覆盖；
- ❑ 接口和内部类；
- ❑ 类文件即是 Java 平台的可执行文件。

线程原本是操作系统中的一个概念。在绝大多数平台上，Java 平台中的线程其实就是利用了操作系统本身的线程。对于学习 Java 线程而言，最重要的内容是理解线程。在理解了线程之后，再去学习 Java 中常用的线程编程其实不难。除了介绍线程的概念，本章还会讲解 Java 线程的使用、多线程编程和线程同步的基本知识。这些都是最常用的线程编程技术。

本章 15.1 节用来讲述线程的概念，是本章中最重要的一节。对于线程这种抽象的概念，一次看不懂也是正常的。15.1 节是全章的基础，理解了 15.1 节的内容，本章剩余的内容就不难理解了。所以请读者在继续后面的内容之前，务必将 15.1 节的内容看懂。好，下面首先理解线程的概念。

15.1 线程——执行代码的机器

线程是编程中极其重要的一部分内容，但是对于初学线程的读者来说，它的概念显得过于抽象而不好理解。和程序的代码不同，线程是隐藏在程序背后的，对于编程者来说它是看不见摸不着的。为了形象地描绘线程的作用，本节将使用一个“CD 机模型”和“演奏会模型”来与线程进行类比。为了明白线程，首先需要了解 Java 程序是如何运行的。

15.1.1 线程——执行代码的基本单位

什么是线程呢？它不是 Java 语言语法的一部分。在 Java 中，线程可以说是一个“机器”，它的作用就是执行 Java 代码。换句话说，Java 中的代码，都是通过线程为基本单位来执行的。图 15-1 描绘了前面学习的从 Java 源代码到生成 Java 类文件的过程。

相信这个过程大家并不陌生，本章后面的内容对上面这个过程将不再叙述。生成了 Java 类文件之后，就是运行 Java 程序了。上段中说过，线程是 Java 中程序执行的基本单位，执行一个 Java 程序（有 main() 方法的 Java 类）的过程如图 15-2 所示。



图 15-1 生成 Java 类文件的过程

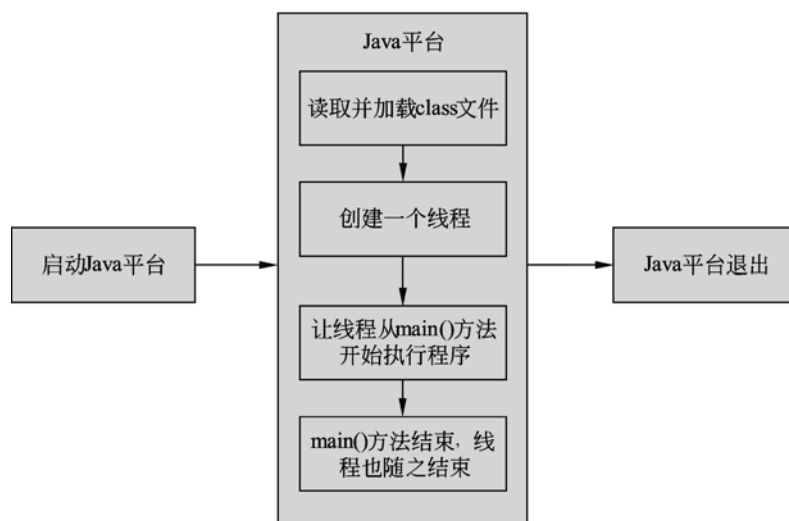


图 15-2 Java 程序执行过程

在图 15-2 中, 启动 Java 平台就是我们在命令行执行 java 命令, Java 平台退出就是 java 命令执行结束。中间的图表示了 Java 平台执行的过程。因为是在控制台上直接使用 java 命令执行一个类文件的, 所以很容易觉得 java 命令是执行 Java 代码的基本单位。实际上, java 命令是通过创建一个 Java 线程来执行 Java 代码的。

说明: Java 线程当然也是 Java 平台的一部分。在本章中为了突出 Java 线程, 从概念上将它从 Java 平台中剥离了出来单独讲解。

1. Java线程和CD机

从线程的角度来看, Java 平台更像是一个线程管理器。下面我们通过一个例子, 来说明类文件、Java 线程和 Java 平台的关系。大家都用过 CD 机, CD 机中读取 CD 碟片内容的部件就是 CD 机上的激光头。CD 和 Java 之间各个元素可以做如下类比, 如图 15-3 所示。

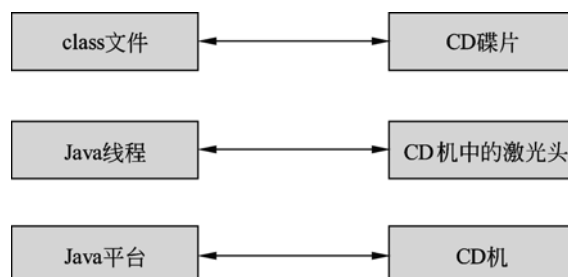


图 15-3 Java 和 CD 的对比

class 文件就如同 CD 碟片：class 文件中包含 Java 程序的可执行代码；CD 中包含着音乐文件。它们都是数据的载体。

Java 线程就如同 CD 中的激光头：Java 线程负责执行 class 文件中的代码；激光头负责读取并处理 CD 上的音乐文件。它们都是负责处理数据的。

Java 平台就如同 CD 机：Java 平台包含 Java 线程，然后，Java 线程还负责管理 Java 线程，包括创建 Java 线程，为 Java 线程提供各种资源（在这里不去深究是什么资源，可以将之理解为 Java 线程执行代码的各种基础条件）；CD 机包括激光头，同时它也管理着激光头，为激光头供电，同时还提供外壳、马达、播放控制、音频解码和音频输出等各种功能，没有这些功能，激光头本身无法处理数据。Java 平台和 CD 机可以说是独立的系统，可以完成一个功能；线程和激光头是它们中的核心部件，但是并不是可独立完成整个工作的部件。

2. 从CD机的工作机制看Java线程

通过上面这个例子，相信线程这个概念已经不完全陌生了。对于激光头，它会从 CD 的某个位置开始，按照顺序读取 CD 上的数据。那么线程的工作模式是怎样的呢？其实和激光头很类似——只要给线程一个“开头”，线程就会一直沿着这个“开头”执行下去。对于前面的所有例程来说，这个“开头”在图 15-2 中已经说明了，它就是我们再熟悉不过的 main()方法。也就是说，线程会从 main()方法开始执行程序。

CD 机的作用就是播放音乐，当 CD 在播放完 CD 后，激光头就会关闭，CD 机也会自动关机。同样的道理，Java 平台的作用就是执行 Java 程序代码。线程在执行完 main()方法后，也就结束了。而当 Java 平台发现自己里面的线程都退出以后，也就会退出。这时 Java 程序就运行完毕了。

那么，线程是如何执行代码的呢？这个超出了本书的范围。就好像使用 CD 机一样，使用者只要知道 CD 机中激光头是用来读取和处理 CD 碟片上的内容就可以了，没必要去追究它是如何读取 CD 碟片上的内容的。对于线程也是一样的道理。前面说过，线程是操作系统中的一个概念，所以“线程是如何执行程序的”这个问题属于操作系统课程中的内容。对于学习 Java 而言，在这里先知道如下几点就可以了。

- ❑ 线程是执行 Java 程序代码的基本单位。
- ❑ Java 线程也是 Java 平台的一部分。
- ❑ Java 线程是运行在 Java 平台内部的，Java 平台负责管理 Java 线程。
- ❑ Java 线程执行程序代码时，Java 平台为其提供各种所需的条件。
- ❑ 当线程执行完给它的方法后，就会退出。Java 平台中如果没有正在运行的线程，就代表程序执行完毕，Java 平台也就自动退出了。

本节的内容就先到这里。本节中使用的“CD 机模型”比较容易理解，但是它和线程还不是十分相似。在 15.1.2 节中将通过“演奏会模型”来加深对线程的理解。

15.1.2 演奏会模型

线程是隐藏在 Java 平台之中的，它的工作方式并没有展露在 Java 语法中。我们只能通过类比的方式来理解 Java 线程以及程序代码、Java 线程和 Java 平台三者之间的关系。

本节将要介绍的就是“演奏会模型”。演奏会并不是一个陌生的概念，可以把它看成是由一个指挥家、一个或多个演奏家、乐谱组成的事物。它的最终结果就是演奏乐谱。我们的这个演奏会模型与现实中的演奏会差不多，区别有以下几点。

- ❑ 演奏会中使用的乐谱不是纸质的乐谱，而是用一个显示器显示乐谱。
- ❑ 所有的乐谱都保存在一个存储设备上。所以在演奏会开始之前要先将乐谱输入到存储设备中。
- ❑ 每个演奏家使用一个单独的显示器来看乐谱，但是所有的显示器都从同一个存储设备上读取乐谱。
- ❑ 显示器每次只显示一小节乐谱内容，演奏家演奏完这个小节后，显示器会自动显示出乐谱下一小节的内容，直到乐谱结束。

演奏会的工作模式也不同。

- ❑ 首先将所有的乐谱输入到存储设备中。
- ❑ 指挥家按照演奏的进度，每当需要演奏一个乐谱的时候，这个指挥家首先请上一位演奏家，然后搬上一个显示器来显示需要演奏的乐谱。
- ❑ 演奏家按照显示器上的内容进行演奏。当演奏家演奏完当前小节后，显示器自动显示乐谱下一小节的内容。
- ❑ 当乐谱结束后，演奏家就退场了。当所有的演奏家都退场以后，指挥家就退场，演奏会就结束了。

1. Java线程和演奏会模型

下面用图 15-4 将 Java 程序和这个演奏会模型做一个类比。

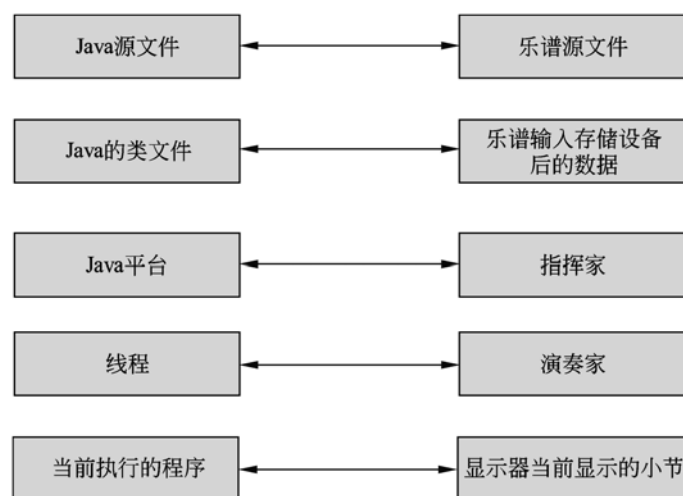


图 15-4 Java 线程和演奏会模型的类比

2. 运行中的Java线程和演奏会模型

通过图 15-4 可以更清晰地看出线程、Java 平台和程序代码之间的关系。一个演奏家专心演奏摆在面前的乐谱，就好像 Java 线程一行行的执行代码。Java 平台则是指挥家，管理

着整个程序，包括 Java 线程。Java 平台为了让 Java 线程能够执行代码，做了很多的工作。下面的图 15-5 将一个程序的执行过程和一个演奏会的执行过程做了一个类比。

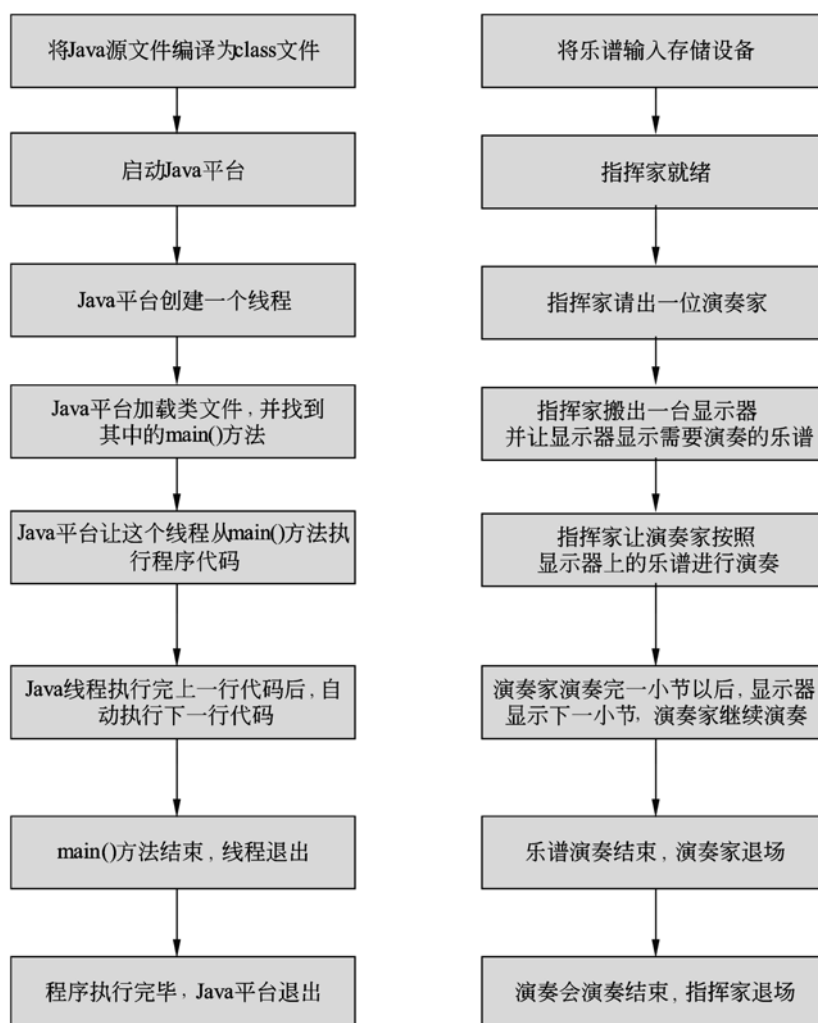


图 15-5 Java 程序执行流程和演奏会流程

下面以一个简单的例程来说明图 15-5 中所示的流程。

```

package com.javaeasy.execution;           // 例程所在的包
public class Execution {                   // 例程名
    public static void main(String[] args) { // main()方法，这就是程序执行
                                                // 的起点，也是线程执行的起点
        int i = 3 + 5;                     // 第一行代码是一个运算操作
        System.out.println(i);             // 第二行代码是一个方法调用
    }
}
  
```

上面的例程很简单，这里不再解释。使用如下的 `javac` 命令对例程的源代码进行编译。

```
javac com\javaeasy\execution\Execution.java
```


编译结束后，就会生成相应的类文件 `Execution.class`。`Execution` 类是有 `main()` 方法的，所以可以使用 `java` 命令来执行这个类文件。在使用如下命令执行 `Execution` 类的时候，

```
java com.javaeasy.execution.Execution
```

首先 Java 平台会启动，启动完毕后，Java 平台会创建一个线程。然后 Java 平台会读取给它的参数，也就是 `com.javaeasy.execution.Execution`。它是类的全限定名，Java 平台会根据这个名字来寻找需要执行的类，找到 `Execution.class` 文件后，就会把它读取并加载到程序中，然后让开始创建出来的线程去执行 `Execution` 类中的 `main()` 方法。

关于 `main()` 方法：`main()` 方法从语法上来说只是一个普通的方法，它的特殊之处在于“Java 平台会将 `main()` 方法作为一个程序的开始，让线程从这里开始执行程序”。这也是为什么所有可以直接被执行的 Java 程序都必须有这么一个 `main()` 方法的原因。`main()` 方法是一个约定俗成，无论 Java 平台以什么方法作为程序的开始都是没有关系的。

例程中的第 1 行是一个 `int` 变量的加法运算和 `int` 变量的赋值操作。线程在执行的时候就会为变量分配内存并进行运算。例程中的第 2 行是一个方法调用。对于方法调用，线程执行的时候会将这个方法“展开”（也就是进去被调用方法内部。在第 14 章讲解异常传递时，就在方法调用的时候将方法展开了），这样一个方法调用就变成了执行方法中的代码了。线程就会这样逐行代码地执行下去，直到 `main()` 方法结束。

 **说明：**线程始终都是在一行行地执行代码，遇到方法调用时，就进去被调用方法内部去一行行执行。当然，我们在阅读程序的时候，可以简单地认为方法调用就是一个运算。只有在必要的时候，我们才将一个方法调用“展开”。

通过上面的分析，线程这个概念已经了解的差不多了，后面将讲述线程编程的基础。本节的内容是后面内容的基础。

- ❑ 线程的作用就是从某个指定的方法（如 `main()` 方法）开始逐行执行代码。就好像演奏家按照乐谱一小节一小节演奏一样。
- ❑ 执行完指定的方法线程就结束了。
- ❑ 通过演奏会模型加深对线程的理解。

15.2 Java 中的线程编程

前面对“线程是什么”做了大量的讲述和类比。本节中将讲述如何使用线程编程。通过前面的内容我们知道，线程的作用就是“执行一个方法，执行完这个方法后，线程就结束了”。在 Java 中我们可以自由地使用线程：创建线程，指定它需要执行的方法，然后启动线程，这个线程就会执行下去了。请看本节的内容。

15.2.1 线程类 Thread

在 Java 语言中，线程被封装为 `Thread` 类（全限定名是 `java.lang.Thread`）。当然，线程核心的内容并不在这个类中，因为真正的线程是存在于 Java 平台中的。可以把这个类认

为是真正的线程的“代理人”。我们操作 `Thread` 类时，`Thread` 类就会操作真正的线程，这就好像“使用引用操作引用指向的对象”。

在程序中，可以按照需要创建和使用线程。当创建一个 `Thread` 类的实例时，在 Java 平台内部，一个真正的线程同时被创建了出来。其实使用线程很简单，根据前面对线程的介绍，使用线程的时候只需要关心下面两点就可以了。

- ❑ 如何指定线程需要执行的方法。我们知道，线程的作用就是执行一个方法，直到方法结束，线程也完成了使命。
- ❑ 如何启动一个线程。当创建出了线程，也指定了线程需要执行的方法，下面的事情就是“推动一下”，让线程启动起来。

下面就来讲述使用 Java 线程的第一种办法。

15.2.2 覆盖 `Thread` 类的 `run()` 方法

为了使用线程，首先需要学习 `Thread` 类中的两个重要方法。

- ❑ `Thread()`：这是 `Thread` 类的一个构造方法，它没有任何参数，所以说创建线程的实例也是很简单的，可以不提供任何参数。
- ❑ `void start()`：`start()` 方法就是启动线程的方法，这个方法是线程类中最核心的方法。当调用这个方法以后，它就会启动线程，并让线程去执行指定的方法，而这里说的“指定的方法”，就是下面要说的 `run()` 方法。
- ❑ `void run()`：`run()` 方法是 `Thread` 类中一个普通的方法，它的特殊之处仅仅在于 `start()` 方法会将它作为线程的起点。

好，知道了 `Thread` 类中这两个方法的作用后，如何让 `run()` 方法变成想让线程去执行的方法呢？这里就要用到继承和覆盖了。我们使用一个类去继承 `Thread` 类，然后为这个 `Thread` 类的子类添加一个 `run()` 方法，用来覆盖 `Thread` 类中原来的 `run()` 方法。那么，根据 Java 覆盖的原则，`start()` 方法再调 `run()` 方法的时候，其实就是调用的 `Thread` 类中子类的 `run()` 方法了。也就是说，“只要在 `Thread` 类的子类中的 `run()` 方法内部，编写需要让线程执行的代码”就可以了。

下面以一个例子来演示这种使用线程的方法，首先创建一个 `Thread` 类的子类，并覆盖 `Thread` 类中的 `run()` 方法。

```
package com.javaeasy.usethread;           // 程序所在的包
public class MyThread extends Thread {    // MyThread 类继承自 Thread 类

    public void run() {                    // 覆盖 Thread 类中的 run() 方法
        System.out.println("这是在另一个线程中执行的代码。");
                                           // 向控制台输出一行字
    }                                      // run() 方法结束
}
```

上面的类很简单。首先需要注意的就是 `MyThread` 类继承自 `Thread` 类，然后是 `MyThread` 类覆盖了 `Thread` 类中的 `run()` 方法。这样才能够让线程在启动后（调用 `start()` 方法后）执行到想让线程执行的内容。好，下面是一个使用 `MyThread` 类的例程。

```

package com.javaeasy.usethread;           // 程序所在的包
public class UseMyThread {                // 例程类

    public static void main(String[] args) { // main()方法
        MyThread thread = new MyThread(); // 创建一个 Thread 类的实例
        thread.start();                    // 启动一个新的线程
    }
}

```

运行上面的例程，控制台输出如下内容：

这是在另一个线程中执行的代码。

到这里，线程似乎还没有带来什么让人兴奋的特征。不过不着急，下面首先在图 15-6 中使用“演奏家模型”说明一下上面例程的执行过程（省略关于编译等无关的步骤）。

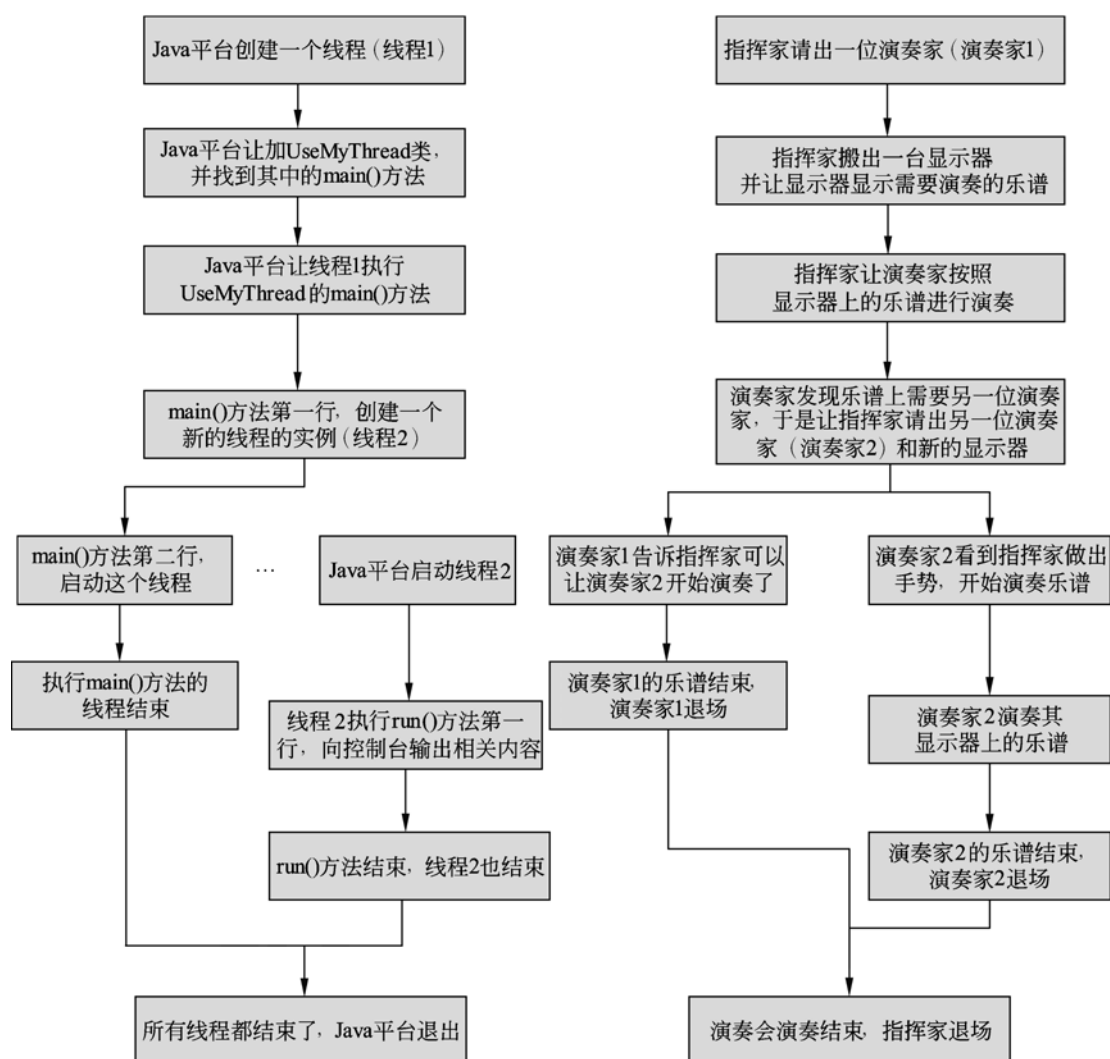


图 15-6 创建并启动线程的程序执行过程

图 15-6 可以看出如下几点内容。

- ❑ 新线程的创建和启动其实都是通过 Java 代码触发的。实际上，除了第一个线程（也就是启动程序的、运行 `main()` 方法的线程）是由 Java 平台直接创建的之外，其余的线程都是在 Java 代码中通过“创建 `Thread` 类的实例，启动线程”这种方式创建并启动的。
- ❑ 当启动一个新的线程时，其过程是：由 Java 代码通知 Java 平台，Java 平台再启动线程。例如在图 15-6 中，线程 1 启动线程 2 的过程实际上就是：线程 1 执行 `thread.start()`，这个方法在内部会让 Java 平台启动第二个线程。所以，启动线程 2 对于线程 1 来说，是一个很短的过程，因为启动线程的具体事情都是 Java 平台做的，线程 1 只是“通知”Java 平台要启动线程 2 而已，通知完了就继续执行代码，不等待线程 2。
- ❑ 只有所有的线程都退出以后，程序才会退出。

如果这个过程看上去太抽象，请对比着右边演奏会的过程进行理解。下面学习另一种使用 Java 线程的方式。

- ❑ `Thread` 类的 `start()` 方法是用来启动一个线程的。
- ❑ `Thread` 类的 `run()` 方法是一个线程启动后执行的方法。

15.2.3 使用 `Runnable` 接口

我们知道，Java 中的类只能够是单继承，也就是说，如果一个类为了使用线程而继承了 `Thread` 类，它就不能再继承别的类了。这很可能给编程带来不便。本节中介绍的就是一种脱离继承来使用线程的方法。这个方法的核心就是 `Runnable` 接口。

`Runnable` 接口的全限定名是 `java.lang.Runnable`。它其中只有一个抽象方法 `void run()`。为了了解如何在线程中使用 `Runnable` 接口，我们还需要看一下 `Thread` 类中的一个叫做 `target` 的属性和 `Thread` 类中的 `run()` 方法。`Thread` 类中有一个类型为 `Runnable` 的属性，叫做 `target`。而 `Thread` 类的 `run()` 方法用到了这个属性，`run()` 方法的代码如下：

```
public void run() {           // Thread 类的 run() 方法
    if (target != null) {     // 检查 target 属性是否为空，target 属性是 Runnable
                              // 类型的引用
        target.run();         // 如果不为空则执行 run() 方法
    }                         // 否则什么都不做
}                             // run() 方法结束
```

如何让 `target` 的值不为 `null` 呢？`Thread` 类的另一个构造方法就是用来给 `target` 属性赋值的，这个构造方法是 `Thread(Runnable)`。当调用这个构造方法时，传递过来的参数就会赋值给 `target` 属性。也就是说，如果直接使用 `Thread` 类也是可以的，步骤如下：

（1）实现 `Runnable` 接口，例如叫做 `MyRunnable`，并在 `MyRunnable` 类的 `run()` 方法里编写想要让线程做的事情。

（2）创建一个 `MyRunnable` 的实例。

（3）通过构造方法 `Thread(Runnable)` 来创建 `Thread` 类的实例。

这时再调用 `start()` 方法启动这个线程，执行的就是 `MyRunnable` 中 `run()` 方法的代码了。下面我们来使用以下这种方法，首先是 `MyRunnable` 类。

```

package com.javaeasy.usethread;                // 程序在的包

public class MyRunnable implements Runnable {    // 实现 Runnable 接口
    public void run() {                          // 实现 run() 方法
        System.out.println("这是在另一个线程中执行的代码。");
                                                // 向控制台输出一行字
    }                                            // run() 方法结束
}

```

MyRunnable 实现了 **Runnable** 接口，其 **run()** 方法就是线程会去执行的方法。然后是例程。

```

package com.javaeasy.usethread;

public class UseMyRunnable {
    public static void main(String[] args) {    // 例程的 main() 方法
        // 创建一个 MyRunnable 类的实例, MyRunnable, MyRunnable 实现了 Runnable
        // 接口
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);   // 调用 Thread 相应的构造
                                                // 方法, 传入参数
        thread.start();                         // 启动线程
    }
}

```

在例程中，按照步骤分别创建 **MyRunnable** 类的实例，调用 **Thread** 相应的构造方法，最后启动线程。因为 **Runnable** 是个接口，为了简单一些，还可以使用前面学到过的匿名类来实现相同的功能。使用匿名类的例程如下：

```

package com.javaeasy.usethread;

public class UseRunnable {
    public static void main(String[] args) {    // 测试类的 main() 方法
        // 创建一个线程, 参数为一个实现了 Runnable 接口的匿名类的实例
        Thread thread = new Thread(new Runnable() {
            public void run() {                  // 实现抽象方法 run()
                System.out.println("这是在另一个线程中执行的代码。");
            }
        });
        thread.start();                         // 启动线程
    }
}

```

例程 **UseRunnable** 其实和例程 **UseMyRunnable** 是一样的。当然，从本质上讲，无论是使用继承还是使用 **Runnable** 接口，其目的都是一样的。让线程执行我们写的一段代码。使用继承并覆盖 **run()** 方法也好；使用 **Runnable** 接口也好，都是为了指定线程执行的方法。本节不再给出程序执行的实例图，图 15-6 可以涵盖本节的程序流程。

下面我们回过头去看看图 15-6：当一个新的线程启动以后，程序就相当于是有两个同时在执行的线程。没错，事情就是这样的。就好像演奏会上的两个演奏家一样，两个演奏家是一起演奏各自的乐谱。两个线程也是各自执行自己的代码，彼此之间互不影响。但是事情到这里就开始变得有意思了：一个程序内有两个线程。好，下面让我们进入 15.2.4 节，看看两个线程的故事。

□ 使用 **Runnable** 接口来让线程执行自己编写的 **run()** 方法。

15.2.4 两个线程

前面介绍了如何设定线程执行的方法和如何启动一个线程。但是 `main()` 方法在启动完第二个线程后就直接退出了。本节我们来看一下如果 `main()` 方法在启动完第二个线程后不直接退出，会出现什么情况。下面在 `main()` 方法启动线程后，再添加一行向控制台输出一行字的代码，新的例程如下：

```
package com.javaeasy.usethread;           // 程序所在的包
public class UseMyThread {                 // 例程类


    public static void main(String[] args) { // main()方法
        MyThread thread = new MyThread();   // 创建一个 Thread 类的实例
        thread.start();                     // 启动一个新的线程
        System.out.println("这是在 main()方法中执行的代码。"); // 启动线程后，再输出一行代码
    }
}
```

在添加了这一行后，线程 1 和线程 2 有一段时间就是同时运行的了。先想想程序运行的结果，然后运行例程，控制台可能输出如下内容：

```
这是在 main()方法中执行的代码。
这是在另一个线程中执行的代码。
```

这个输出的结果确实令人吃惊：为什么首先启动的线程 2，但是控制台输出的第一行却是线程 1 所运行的 `main()` 方法的内容呢？原因正是我们前面所说的那样：线程在执行代码的时候是相对独立互不影响的。也就是说，哪个线程执行得快，哪个线程执行得慢，并不是确定的。下面用图 15-7 来演示这个过程。

在图 15-7 中，线程 2 启动的时间稍长了一些，线程 1 在通知 Java 平台去启动线程 2 之后，快速地执行了下一行代码，也就是向控制台输出一行字，而线程 2 则落在了线程 1 的后面，所以在控制台上看到的是那样的内容。当然，这只是一种可能性，也有可能线程 2 会跑到线程 1 的前面，我们可以认为这个是随机的。这点也是线程与演奏会模型最大的不同。我们假设演奏会模型里的演奏家也是这样的，演奏家彼此之间互不影响，各自演奏各自的部分，这就和线程中的情况很类似了。

 **并行：**有一个术语专门用来形容“多个事情同时进行”的情况，叫做并行。对于线程来说，就是“线程是并行的”或“线程并行执行”。

当然，这里还有一个角色——Java 平台。Java 平台作为所有线程的管理者，会让每个线程运行的时间差不多。就好像演奏会里的指挥家一样，虽然每个演奏家可能有快有慢，但是总体上来说会按照指挥家的指挥进行演奏。也就是说，我们不能确定线程 1 中的某行代码是否肯定在线程 2 的某行代码之前或之后执行，但是可以肯定的是线程 1 和线程 2 都在执行代码。好，15.3 节中我们通过一个相对复杂点的例程来进一步看看多个线程之间的故事。

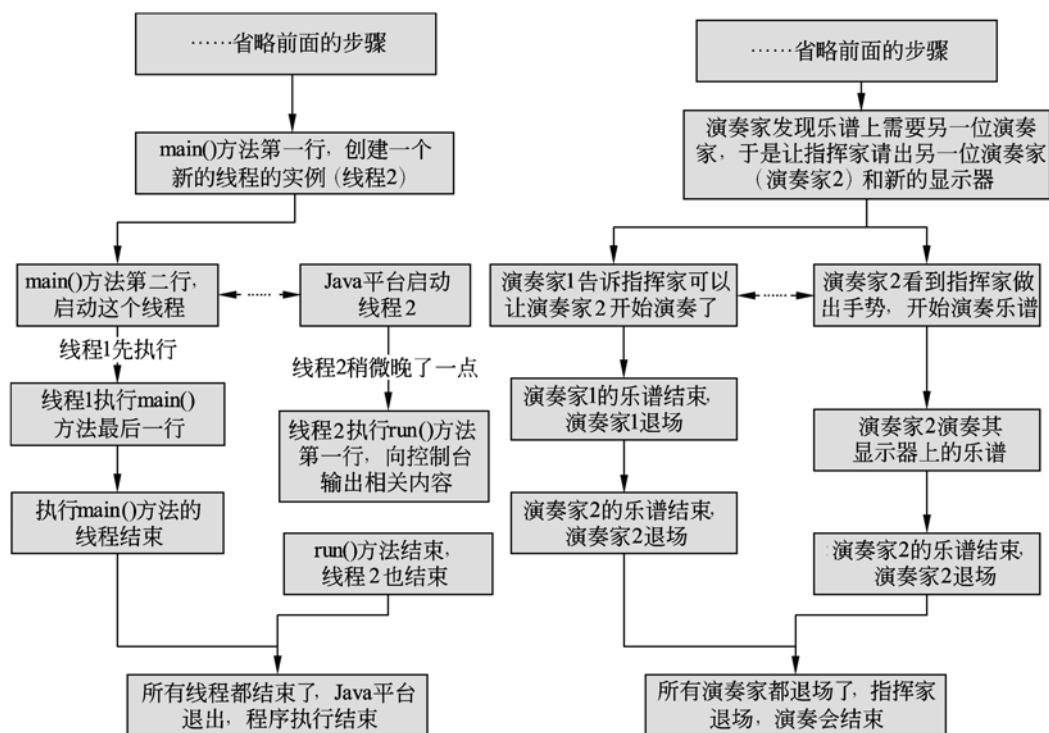


图 15-7 两个线程执行的快慢速度不能确定

主线程：对于那个“由 Java 平台创建的，用来执行 `main()` 方法的线程”，习惯上称之为“主线程”。当然，这个线程在执行代码等方面并没有任何特殊之处。对于上面说到的“线程 1”，在后面的内容中将称之为主线程。

- ❑ 启动一个线程对于 Java 程序来说是很快的，因为 Java 程序仅仅是“通知”Java 平台去启动线程而已，真正的启动工作是 Java 平台做的。
- ❑ 在一个 Java 程序中的多个线程是并行执行的，线程之间在执行代码的时候互不影响。

15.3 深入学习 Thread 类

前面的内容讲解了 `Thread` 类最基本的方法和用法，包括 `run()` 方法，一些构造方法以及 `start()` 方法。仅通过这些方法还不能够释放线程最大的能量。本节中将继续讲述 `Thread` 类中一些重要的方法和属性。包括让线程挂起、线程的名字和得到执行当前代码的线程。

15.3.1 线程的名字

本节中学习一下 `Thread` 类的 `name` 属性，它的类型是 `String`。它其实就是线程的名字，和演奏家的名字是一样的。在 `Thread` 类中，有 `String getName()` 和 `void setName (String)` 两个方法用来设置和获取这个属性的值。

同时 `Thread` 类还提供相应的构造方法，让 `Thread` 类的对象在创建的时候就有名字，在这里列出两个 `Thread` 类的构造方法。

- ❑ `Thread (String name)`：接受一个 `String` 实例为参数的 `Thread` 类构造方法，这个参数就将是这个线程的名字。
- ❑ `Thread (Runnable target, String name)`：接受一个 `Runnable` 实例和一个 `String` 实例为参数的 `Thread` 类构造方法。其中 `Runnable` 中的 `run()` 方法就是线程将要执行的方法；`String` 实例就将是这个线程的名字。


如果在创建一个 `Thread` 类实例的时候没有为 `Thread` 实例提供一个名字，那么 `Thread` 实例将使用一个默认的名字。我们可以认为 `Thread` 实例的名字就是演奏家的名字。好，下面通过一个例程来学习一下 `Thread` 类的 `name` 属性。首先给出一个 `Thread` 类的子类。

```
package com.javaeasy.threadname;           // 程序所在的包

public class ShowThreadName extends Thread { // 继承自 Thread 类
    public ShowThreadName() {                // 构造方法，没有参数
        super();                             // 调用父类相应的构造方法
    }
    public ShowThreadName(String name) {      // 构造方法，提供线程的名字
        super(name);                         // 调用父类相应的构造方法
    }

    public void run() {                      // 覆盖 run() 方法
        System.out.println("这个线程的名字是: " + this.getName());
        // 输出线程的名字
    }
}
```

类 `ShowThreadName` 很简单，首先它继承自 `Thread` 类。然后提供了两个构造方法，这两个构造方法都是直接调用父类相应参数的构造方法。如果通过没有参数的构造方法创建 `Thread` 类的实例，那么这个实例将会有默认的名字；如果通过有 `String` 参数的构造方法创建 `Thread` 类的实例，那么这个线程就会使用这个 `String` 实例作为名字。`run()` 方法仅仅是输出了线程的名字。

 **线程的默认名字**：对于在程序中创建的线程，线程的默认名字一般是“Thread-”加上一个递增的整数；而对于主线程，它的名字一般会被设置为 `main`。

下面是例程。

```
package com.javaeasy.threadname;           // 程序所在的包

public class ShowThreadNameMain {          // 例程
    public static void main(String[] args) { // main() 方法
        // 使用无参数的构造方法创建 Thread 类实例，这时它将有一个默认的名字
        ShowThreadName defaultName = new ShowThreadName();
        // 使用构造方法给线程指定一个名字
        ShowThreadName name = new ShowThreadName("线程的名字");
        // 启动两个线程
        defaultName.start();
        name.start();
    }
}
```

上面的例程中，首先以无参数的构造方法创建了一个 `ShowThreadName` 类的实例，然后通过用一个 `String` 实例为参数的构造方法创建了一个 `ShowThreadName` 类的实例。最后启动这两个线程，根据 `ShowThreadName` 的 `run()` 方法的内容可以知道，程序的运行结果是在控制台上输出这两个线程的名字。运行例程，控制输出。

```
这个线程的名字是：Thread-0
这个线程的名字是：线程的名字
```

通过 `setName(String)` 方法还可以更改线程的名字。`name` 属性的操作很简单，在这里就不给出例程了。当一个 Java 程序中有多个线程在运行的时候，给线程一个名字还是很有用的。

- ❑ `Thread` 类的实例的名字。
- ❑ 如果不通过构造方法给线程实例一个名字，那么它将拥有一个默认的名字。主线程的名字一般叫做 `main`。
- ❑ 可以通过 `setName` 和 `getName` 来设置/得到线程的名字。

15.3.2 得到当前的线程

既然线程是代码的执行器，那么每行代码在真正执行的时候，都是由某个线程负责的。如何得到这个线程呢？下面的内容给出了答案。

1. `Thread` 类的静态方法 `currentThread()`

我们知道，Java 的线程是执行 Java 程序的基本单位。也就是说，所有的 Java 代码最终都是由线程执行的，就好像所有的音符都是由演奏家演奏的。如果在程序中需要得到“执行当前代码的线程的引用”，那么就可以使用 `Thread` 类的静态方法 `Thread currentThread()`。这个方法的返回值是 `Thread` 的引用，这个引用所指向的 `Thread` 类的实例正是“指向执行当前代码的线程”。下面通过一个简单的程序来演示一下如何使用这个方法。

2. 使用 `Thread.currentThread()` 方法

下面将给出一个程序，展示 `Thread.currentThread()` 的用法。

```
package com.javaeasy.currentthread;           // 包名

public class PrintCurrentThreadName {          // 类名，此类不是 Thread 类的子类
    public void printCurrentThreadName() { // 打印当前线程名字的方法
        Thread currentThread = Thread.currentThread(); // 获得当前的线程
        String threadName = currentThread.getName();
                                                // 得到当前线程的名字
        System.out.println("执行代码的线程名叫做: " + threadName);
                                                // 向控制台输出当前线程的名字
    }
}
```

在上面的程序中，首先使用 `Thread.currentThread()` 方法得到执行当前代码的线程引用，并将它赋值给 `currentThread`，然后使用 `currentThread` 执行 `getName()` 方法来得到当前线程

的名字，最后将这个名称输出到控制台。

3. 在主线程中使用PrintCurrentThreadName

下面的例程使用了 `PrintCurrentThreadName` 类。

```
package com.javaeasy.currentthread;           // 程序包

public class CurrentThreadMain {               // 例程类
    public static void main(String[] args) {   // main()方法
        // 创建一个 CurrentThreadNamePrinter 类的实例
        CurrentThreadNamePrinter printer = new CurrentThreadNamePrinter();
        // 调用 printCurrentThreadName()方法，用来输出执行此方法的线程的名字
        printer.printCurrentThreadName();
    }
}
```

在上面的例程中，首先创建了一个 `CurrentThreadNamePrinter` 类的实例，然后通过这个实例调用了 `printCurrentThreadName()`方法。在 `printCurrentThreadName()`方法中，会使用 `Thread.currentThread()`得到“执行当前代码的线程的引用”，并通过这个引用来得到线程的名字，将它输出到控制台。运行例程，控制台输出如下内容：

```
执行代码的线程名叫做: main
```

在上面的例程中，没有创建新的线程，执行 `printCurrentThreadName()`方法的线程肯定就是主线程。我们前面说过，主线程的名字一般会被设置为 `main`。通过控制台的输出，也可以证明这一点。

4. 在新的线程中使用TestMultiply

下面的例程将创建一个新的线程，并在这个线程中执行 `printCurrentThreadName()`方法。

```
package com.javaeasy.currentthread;           // 包名

public class CurrentThreadMainII {            // 例程名
    public static void main(String[] args) {   // main()方法
        Runnable runnable = new Runnable() {
            // 通过匿名内部类创建一个 Runnable 的实例
            public void run() {                 // 实现抽象方法 run()
                // 创建一个 CurrentThreadNamePrinter 类的实例
                CurrentThreadNamePrinter printer = new CurrentThreadNamePrinter();
                // 调用 printCurrentThreadName()方法，用来输出执行此方法的线程的
                // 名字
                printer.printCurrentThreadName();
            }
        };
        // 使用 runnable 创建一个线程，线程名字叫做“线程-1”
        Thread thread = new Thread(runnable, "线程-1");
        thread.start();                         // 启动线程
    }
}
```

在上面的例程中，首先创建了一个实现 `Runnable` 接口的匿名内部类的实例（关于匿名

内部类的内容，请参见 13.5 节）。然后使用这个实例，创建了一个 `Thread` 类的实例，它的名字是“线程-1”，最后启动这个线程。运行例程，控制台输出如下内容：

```
执行代码的线程名叫做：线程-1
```

同样，`printCurrentThreadName()`方法获得了执行当前代码的线程名字。

5. 理解 `Thread.currentThread()` 方法

我们可以假设乐谱中有一个特殊的符号@，每当演奏家演奏这个“音符”的时候，就会报出自己的“编号”（注意，不是名字）。编号对每个演奏家都是不同的。通过这个编号就可以找到这个演奏家，那么这个编号其实就相当于 Java 中的引用的值。

对于一段程序，任何一个线程都可以去执行它。在程序中，就可以通过 `Thread.currentThread()` 来得到执行程序中的线程；同样，对于一段乐谱，任何一个演奏家也都可以演奏，而通过乐谱中的@符号就可以得到演奏家的编号。得到这个编号以后就可以在乐谱中来“指挥”演奏家（当然现实中的乐谱并没有这个功能）。图 15-8 说明了这个对应关系。

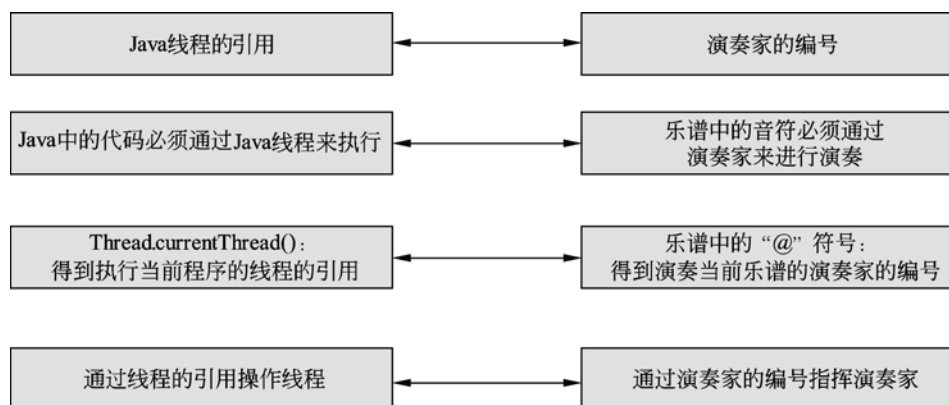


图 15-8 得到当前线程的引用和得到当前演奏家的编号

图 15-9 中，假设“演奏会”模型中的乐谱中有特殊的符号，可以通过演奏家的编号来指挥演奏家。而符号@就是得到演奏当前乐谱的演奏家。好，本节已经介绍了足够多的关于线程的内容，下面开始新的一节，让我们看看“多个线程的故事”。

- ❑ 理解 `Thread.currentThread()` 方法的作用：对于任何一段程序，肯定都是由一个线程来执行的，而 `Thread.currentThread()` 方法的返回值则正是“指向执行当前程序的线程的引用”。

15.3.3 让线程“沉睡”

在前面所有的程序中，其实都没有时间这个概念。线程会一直不停地向下执行代码，直到执行完毕。但有时程序需要停顿一下。例如，现在需要这样一个小程序。程序的功能是给用户提供加法运算测验，程序首先生成两个 0~100 的整数，将这两个整数输出到控制台上，而后程序会给用户 5 秒钟的思考时间，最后程序输出运算结果，供用户与自己的运

算结果比较。

整个程序是很简单的，唯一没有接触过的地方就是如何检查让程序“给用户 5 秒钟的思考时间”。通过前面的学习，我们知道线程是用来执行程序。也就是说，只要线程暂停执行程序，“沉睡”5 秒钟，那么目的就达到了。`Thread` 类中的静态方法 `void sleep(long)` 方法就是能够让“当前线程沉睡”的方法，下面首先看一下这个 `sleep()` 方法。

1. `Thread`类的静态方法`sleep()`

`sleep()`方法是一个静态方法，没有返回值，接受一个 `long` 类型的参数。这个参数的意义是“线程需要沉睡的毫秒数”。也就是说，如果参数为 5000，`sleep()`方法的执行结果就应该是让线程“沉睡”5 秒钟。5 秒钟后，线程会自动苏醒，并继续向下执行代码。

当一个线程在“沉睡”的时候，编程术语中有一个专门的名词来形容线程此时的状态，叫做“被挂起”或者“挂起”。相对应的，当一个线程在执行代码的时候，编程术语中称呼此时线程的状态为“运行”。

在这里需要解释一下线程沉睡的时间。`sleep()`方法并不能够让程序“严格”的沉睡指定的时间。例如当使用 5000 作为 `sleep()`方法的参数时，线程可能在实际被挂起 5000.001 毫秒后才会继续运行。当然，对于一般的应用程序来说，`sleep()`方法对时间控制的精度足够了。

`sleep()`方法会抛出一个类型为 `InterruptedException` 的异常。这个异常的含义就是“线程在处于挂起状态时，因为某种原因被打断了”。例如，当以 5000 为参数执行 `sleep()`方法时，线程应该挂起 5 秒钟左右，但是在线程挂起了 3 秒钟以后，“因为某种原因线程被打断了”，那么就会抛出这个异常。对于这个异常，如果没有特殊的需求，则这个异常是没有必要向外传递的，一般直接使用 `try-catch` 语句处理掉就可以了。

2. 加法测验程序

好，学习了线程的 `sleep()`方法以后，这个加法测验程序就很容易了。看下面的例程。

```
package com.javaeasy.threadsleep;

public class TestAddingInMain {
    public static void main(String[] args) {
        int a = (int) (100 * Math.random()); // 生成一个 0~100 的随机数
        int b = (int) (100 * Math.random()); // 生成另一个 0~100 的随机数
        System.out.println("请在 5 秒钟内计算出下面两个整数的和: " + a + "+" + b);
                                                // 输出两个随机数

        try {
            Thread.sleep(5000); // (1) 让线程挂起 5 秒钟
        } catch (InterruptedException e) { // sleep()方法可能抛出 InterruptedException 异常

            // 输出异常信息
            System.out.println("对不起，程序运行出错，错误信息为: " + e.getMessage());
            return; // 程序出错，不再向下执行
        }
        int result = a+b;
        System.out.println(a + "+" + b + "的运算结果是" + result);
                                                // 输出运算结果
    }
}
```

```
    }
}
```

在 `main()` 方法中，首先是生成两个 0~100 的 `int` 变量。前面讲过，`Math.random()` 会生成一个 0~1 的 `double` 类型的随机数，将这个数乘以 100，再将结果强制类型转换为 `int`，就得到了一个“0~100 的 `int` 变量”。程序向控制台输出这两个随机数后，会调用 `Thread` 类的 `sleep()` 方法进入挂起状态。5 秒钟后，如果发生了异常，那么程序会将错误信息输出并退出；否则，程序会继续向下执行，计算正确的结果并输出。运行例程，控制台输出如下内容：

```
请在 5 秒钟内计算出下面两个整数的和：85+53
85+53 的运算结果是 138
```

其中在第一行输出和第二行输出之间，会间隔 5 秒钟。

3. `sleep()` 方法是个静态方法

在这里必须强调的一点是：`sleep()` 方法是个静态方法。在上面的例程中，是用 `Thread` 直接调用 `sleep` 方法的。`sleep()` 方法的作用准确来说，应该是“让当前线程挂起”。关于“当前线程”这个概念已经在前面讲解了。也就是说，`Thread.sleep()` 方法会让 `Thread.currentThread()` 这个线程挂起指定的时间。下面我们用程序说明这一点。

首先写一个 `TestAdding` 类，用来完成加法测验的功能。

```
package com.javaeasy.threadsleep;                // 程序所在的包

public class {                                    // 没有继承自 Thread 类
    public void giveAddingTest () {                // 加法测验方法
        int a = (int) (100 * Math.random());      // 生成一个 0~100 的随机数
        int b = (int) (100 * Math.random());      // 生成另一个 0~100 的随机数
        System.out.println("请在 5 秒钟内计算出下面两个整数的和：" + a + "+" + b);
                                                // 输出两个随机数
        // 通过 Thread.currentThread() 得到当前线程，进而得到其名字
        String currThreadName = Thread.currentThread().getName();
        // 向控制台输出当前线程的名字
        System.out.println("执行当前代码的线程名叫做：" + currThreadName);
        try {
            Thread.sleep(5000);                    // 让当前线程挂起 5 秒钟
        } catch (InterruptedException e) {          // sleep() 方法可能抛出 InterruptedException 异常
                                                // rruptedException 异常

            // 输出异常信息
            System.out.println("对不起，程序运行出错，错误信息为：" + e.getMessage());
            return;                                  // 程序出错，不再向下执行
        }
        int result = a+b;
        System.out.println(a + "+" + b + "的运算结果是" + result);
                                                // 输出运算结果
    }
}
```

`TestAdding` 类中 `giveAddingTest()` 方法中的内容和上一个例程中 `main()` 方法的内容相似，在这里就不多做解释了。唯一的不同之处是在调用 `Thread.sleep()` 方法之前，会使用

`Thread.currentThread()`方法获得当前线程，并向控制台输出当前线程的名字。

下面使用一个例程来运行上面的程序。

```
package com.javaeasy.threadsleep;           // 程序所在的包

public class TestAddingMain {                // 例程名
    public static void main(String[] args) { // main()方法
        Runnable runnable = new Runnable() { // 通过匿名类创建 Runnable 的
                                                // 实例
            public void run() {                // 实现抽象方法 run()
                TestAdding adding = new TestAdding();
                                                // 创建 TestAdding 的实例
                adding.giveAddingTest();        // 调用 giveAddingTest()
                                                // 方法
            }
        };
        // 使用 runnable 创建一个线程实例，名字是"加法测试线程"
        Thread thread = new Thread(runnable, "加法测试线程");
        thread.start();
        System.out.println("主线程结束了");
    }
}
```

在上面的例程中，首先通过匿名内部类创建一个实现了 `Runnable` 接口的实例（关于匿名内部类的内容，请参见 13.5 节）。这个匿名类的 `run()`方法中会创建一个 `TestAdding` 类的实例并调用 `giveAddingTest()`方法。然后使用这个实例创建了一个名字为“加法测试线程”的线程。启动这个线程后，主线程输出一行字，然后就退出了。

这样，“加法测试线程”就会执行 `giveAddingTest()`方法了。也就是说 `giveAddingTest()`方法中 `Thread.currentThread()`所得到的就是这个名字叫做“加法测试线程”的线程了。而主线程将不会挂起。运行上面的例程，控制台输出如下内容：

```
主线程结束了
请在 5 秒钟内计算出下面两个整数的和：62+94
执行当前代码的线程名叫做：加法测试线程
62+94 的运算结果是 156
```

在实际运行程序的时候会发现，“主线程结束了”会首先输出，并没有等待 5 秒钟，这说明主线程确实没有挂起。第 3 行输出的线程的名字是“加法测试线程”，同时第 3 行和第 4 行输出之间会间隔 5 秒钟，这正说明了被挂起的是叫做“加法测试线程”的线程。所以说，`Thread.sleep()`方法虽然是个静态方法，但是它的作用是“让当前线程挂起”。

4. 理解线程挂起

前面的例程展示了如何让线程挂起。在编程中，经常会需要让一个线程挂起，等待用户的操作。例如上面的例程中，就是为了给用户 5 秒钟的时间计算。如何理解线程的挂起呢？我们可以认为线程的挂起就是和演奏家的“暂停演奏”是一样的。下面通过图 15-9 来对两者进行一个对比。

在图 15-9 中，线程在执行方法中的代码，演奏家在演奏乐谱中的音符。当线程执行到 `sleep(5000)` 这行代码的时候，就相当于演奏家看到乐谱上将有 5000 个 0 音符。假设 1

个音符耗时 1 毫秒，那么乐谱中的 5000 个 0 音符就是让演奏家在 5 秒内不做什么事情。这就和使用 `sleep()` 方法让线程挂起 5000 毫秒的结果是类似的。

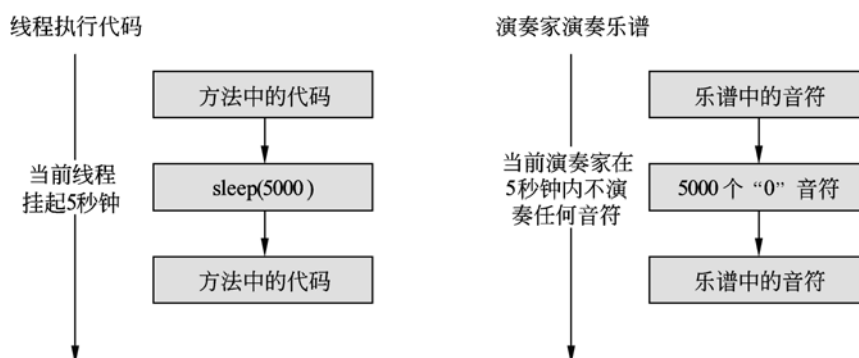


图 15-9 线程挂起与演奏家暂停演奏

- ❑ `Thread` 类的静态方法 `sleep()` 可以让“当前线程”挂起一段时间。理解 `sleep()` 方法必须先理解什么是“当前线程”。
- ❑ `sleep()` 方法不能绝对精确地控制线程挂起的时间，但是精确度对于一般的应用程序是足够的。
- ❑ `sleep()` 方法会抛出一个 `InterruptedException` 的异常。

15.4 多个线程的故事

很多时候程序需要有多个线程一起执行，每个线程分别负责不同的任务。实际上，学习线程编程的最重要的目的，就是让程序能够有多个线程一起执行。在本节中，将会使用一个“复印社模型”来展示多线程编程。

如果说单线程是一场只有一个演奏家的独奏会，那么多线程就好像是“一场有多个演奏家参与的演奏会”，有负责小提琴的，有负责大提琴的，有负责钢琴的。多线程让程序内容更加丰富，让程序变成一场气势恢宏的演奏会。好，下面就开始本节的内容。

15.4.1 一个有多个线程的程序

线程是代码的执行器。一个程序中其实可以有多个线程执行在代码。本节中就来演示一个程序中有多线程的情况。

1. 多线程编程

本节将在以前内容的基础上继续讲述“在一个 Java 平台上运行多个线程”的故事。在本节中使用的例程是这样的：创建一个 `Thread` 类的子类 `PrintNumberThread`，这个类中的 `run()` 方法就是向控制台输出多个数字。在主线程中，创建两个 `PrintNumberThread` 类的实例并启动它们。这时，程序中就会有两条线程在同时执行。好，下面首先看 `PrintNumberThread` 类的代码。

```

package com.javaeasy.multithreads;                // 程序所在的包

public class PrintNumberThread extends Thread {    // 继承自 Thread 类
    private int times;                             // times 属性，用来控制输出内容的
                                                    // 次数

    public PrintNumberThread(int times) {          // 构造方法，以一个 int 变量为参数
        this.times = times;                       // 给 times 属性赋值
    }

    public void run() {                            // 覆盖 Thread 类的 run() 方法
        for (int i = 0; i < times; i++) {         // for 循环，循环次数为 times 次
            // 生成需要输出的内容。其中 this.getName() 是得到线程的名字
            String content = this.getName() + "\t:\t" + i;
            try {
                this.sleep(1);                    // 让程序沉睡 1 毫秒
            } catch (InterruptedException e) {
                System.out.println("对不起，程序运行出错，错误信息为：" +
                    e.getMessage());
            }
            System.out.println(content);           // 向控制台输出内容
        }
        // 在 run() 方法的最后一行，输出线程结束的信息
        System.out.println("线程\"" + this.getName() + "\"程结束了。");
    }
}

```

PrintNumberThread 类并不复杂。它继承自 Thread 类，有一个 int 类型的 times 变量。PrintNumberThread 类照惯例覆盖了 Thread 类的 run。在这个 run() 方法中，有个循环 times 次的 for 语句，每次循环会生成一个由线程名字和递增变量 i 组成的字符串，然后将它输出到控制台。在 run() 方法的最后一行，向控制台输出线程结束的信息。

好，下面是例程的代码。

```

package com.javaeasy.multithreads;

public class RunMultiPrintNumThread {            // 例程
    public static void main(String[] args) {     // main() 方法
        // 分别创建两个 PrintNumberThread 的实例。
        PrintNumberThread threadOne = new PrintNumberThread(3);
        PrintNumberThread threadTwo = new PrintNumberThread(5);
        // 给两个 PrintNumberThread 的实例设置不同的名字
        threadOne.setName("线程 1");
        threadTwo.setName("线程 2");
        // 分别启动两个线程
        threadOne.start();
        threadTwo.start();
        System.out.println("主线程结束了。");    // main() 方法最后一行，输
                                                    // 出线程结束的语句
    }
}

```

在上面的例程中，分别创建了两个 PrintNumberThread 类的实例 threadOne 和 threadTwo，并分别给这两个实例设置不同的名字，然后分别启动这两个线程。main() 方法在输出一行文字后就结束了，同时主线程的任务也完成了，主线程将退出。

2. 好戏拉开序幕

下面来运行这个例程，控制台“有可能”会输出如下内容：

```
主线程结束了。
线程 1    :    0
线程 2    :    0
线程 2    :    1
线程 1    :    1
线程 2    :    2
线程 1    :    2
线程"线程 1"程结束了。
线程 2    :    3
线程 2    :    4
线程"线程 2"程结束了。
```

下面分析一下程序的执行过程。首先，在主线程中，创建了两个 `PrintNumberThread` 的实例，并启动这两个线程，然后主线程在输出“主线程结束了。”后就结束了。后面的内容都是由 `PrintNumberThread` 的两个线程输出的。

通过上面的输出内容可以看出，“线程 1 和线程 2 的执行顺序没有任何规律”。并不是“线程 1，线程 2，线程 1，线程 2……”这样按照顺序执行的。而且，上面的这个结果只是笔者某次在自己电脑上运行程序得到的结果。没错，如果再次执行这个程序，控制台的输出内容极可能是与这次不一样（相同也只是巧合）。

也就是说，如果不进行任何控制，多个线程之间的执行顺序是没有任何规律可言的，有可能是线程 1 执行 2 行，然后线程 2 再执行 3 行；也有可能是线程 1 执行 1 行，然后线程 2 再执行 1 行。在本例程中，这并没有什么影响，因为线程 1 和线程 2 之间并没有逻辑上的关联。但是对于绝大多数的多线程的程序来说，线程之间的关系并没有这么简单，如果不好好协调，则程序无法正确工作。在 15.4.2 节的内容中我们将讲解一个多线程带来的问题。

- ❑ 当一个程序中运行着多个线程的时候，Java 平台并不保证这多个线程之间执行代码的顺序。也就是说，这多个线程执行代码的速率是随机的。

15.4.2 复印社模型

本节将引入“复印社模型”，“CD 机模型”和“演奏会模型”用来帮助大家理解线程。这里的“复印社模型”则是用来展示“多线程之并行的问题与多线程相互协调的必要性”的。首先介绍一下这个“复印社模型”。

- ❑ 复印社模型中有一个或多个复印机。
- ❑ 复印社有一个经理，负责员工之间的协调工作。
- ❑ 复印社中员工的工作就是使用复印机复印稿件。
- ❑ 每个复印机只能由一个员工使用（这是关键点）。
- ❑ 如果员工找不到空闲的复印机，那么他必须等待。直到有复印机空闲下来，他才可以使使用复印机复印稿件。

在这个复印社模型中，“复印机”是一个新的东西。它可以被认为是某种“资源”。只有获得了这个资源，工作才能够继续，而这个资源每次只能由一个人来使用。我们在本节中将使用 Java 程序把这个“复印社模型”编写出来。在这里，复印机将使用一个叫做 Copier 的 Java 类来表示。“复印社模型”中的其他元素则与“演奏会模型”很类似。如图 15-10 所示为“复印社模型”和 Java 程序中各个元素的对应关系。

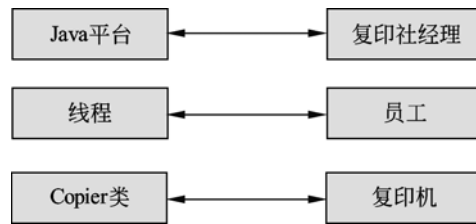


图 15-10 Java 程序与复印社模型的对应关系

下面完成这个复印社程序，复印社程序中有两个重要的类，第一个是 Copier 类，代表复印机；一个是 Employee 类，用来代表员工，它是线程类的子类。Employee 使用 Copier 进行复印工作。在复印期间，Employee 实例处于挂起状态。

1. Copier类

首先给出 Copier 类的代码。

```

package com.javaeasy.simplecopier;           // 程序在的包

public class Copier {                          // 复印机类
    private String name;                      // 复印机的名字

    public Copier(String name) {              // 构造方法，参数为复印机的名字
        this.name = name;
    }

    public String getName() {                 // 得到复印机的名字
        return name;
    }

    public void copyPages (int pages) {       // 复印
        Thread employee = Thread.currentThread();
                                                // 得到当前的“员工”线程

        // 向控制台输出哪个员工正在使用哪个复印机
        System.out.println(employee.getName() + "\t 正在使用复印机\t" +
            name);
        long time = pages * 1000;             // 假设复印一张纸需要 1 秒钟
        try {
            Thread.sleep(time);               // 以挂起线程代表工人正在忙着复印
        } catch (InterruptedException e) {
            System.out.println("对不起，程序运行出错，错误信息为：" +
                e.getMessage());
        }
        // 向控制台输出哪个员工使用完了哪个复印机
        System.out.println(employee.getName() + "\t 用完了打印机\t" + name);
    }
}
  
```

上面的 Copier 类并不复杂，首先是一个 String 类型的 name 属性，它是复印机的名字。构造方法和 getName() 方法都很简单，不再解释了。Copier 类最核心的方法就是 copyPages() 方法，它的作用是模拟复印机进行复印。

根据“复印社模型”，操作复印机的都是员工在操作复印机，而员工又是一个 Thread

类的子类。所以执行 `copyPages()` 方法的肯定是员工线程。在 `copyPages()` 方法中，首先获得代表员工的线程，然后获得员工的名字（也就是线程的名字），并向控制台输出哪个员工正在使用哪台复印机。假设每次复印一张纸需要 1 秒钟，在计算完需要“忙”多少秒之后，就让员工线程挂起，代表此员工正在忙着复印。忙完之后，向控制台输出哪个员工使用完了哪个复印机，方法也就执行完毕了。

2. Employee类

然后是员工线程类的代码。

```
package com.javaeasy.simplecopier;           // 包名

public class Employee extends Thread {        // 继承自 Thread 类的 Employee 类
    private int workTimes;                    // 该员工一天的工作份额
    private Copier copier;                    // 复印机

    // 构造方法，name 是线程的名字，也是员工的名字，剩下两个参数直接赋值给相应的属性
    public Employee(String name, int workTimes, Copier copier) {
        super(name);                          // 调用父类的构造方法
        this.workTimes = workTimes;
        this.copier = copier;
    }

    public void run() {                        // 覆盖 Thread 类的 run() 方法
        System.out.println(this.getName() + ": 开始工作。");
        // 开始一天的工作
        for (int i = 0; i < workTimes; i++) { // 工作 workTimes 次
            // 生成一个从 1~6 的随机数，代表此次需要复印的张数
            int pageAmount = (int) (5 * Math.random()) + 1;
            copier.copyPages (pageAmount); // 调用 printPages() 方法进行复印
        }
        // 完成了 workTimes 次循环后，员工完成了当天的工作份额，下班回家
        System.out.println(this.getName() + ": 完成了工作，下班。");
    }
}
```

`Employee` 类继承自 `Thread` 类，以线程的名字作为员工的名字，同时还有以下两个属性。

- ❑ `workTimes`: 一个 `int` 类型的变量，代表此员工一天工作的份额。
- ❑ `copier`: `Copier` 类型的实例，被此员工用来复印。

`Employee` 类的 `run()` 方法还是最重要的。在这个方法中，首先输出一条信息，代表此员工开始了一天的工作；在方法最后，也会输出一句话，代表此员工结束了一天的工作。`run()` 方法的中间部分就是进行 `workTimes` 次循环，每次都生成一个 1~6 的随机数，代表此次需要复印的张数。然后调用 `Copier` 的 `copyPages()` 方法，进行复印。我们知道，在 `copyPages()` 方法，此线程会根据工作量的不同而挂起相应的时间，代表员工正在工作。

15.4.3 一个简单的复印社例程

准备好了 `Copier` 类和 `Employee` 类，就可以开始写复印社类了。

1. 一个员工一台复印机

下面的 SimpleCopyShop 类是一个简单的复印社程序。

```
package com.javaeasy.simplecopier;           // 程序所在的包

public class SimpleCopyShop{                 // 例程
    public static void main(String[] args) { // main()方法
        Copier canon = new Copier("佳能");   // 创建一台佳能复印机
        Copier sharp = new Copier("夏普");   // 创建一台夏普复印机
        // 创建一个叫做 Simth 的员工线程，每天工作 2 次，并让他使用佳能复印机
        Employee simth = new Employee("Simth", 2, canon);
        // 创建一个叫做 John 的员工线程，每天工作 3 次并让他使用夏普复印机
        Employee john = new Employee("John", 3, sharp);
        simth.start();                       // 启动两个线程
        john.start();
    }
}
```

上面的例程很简单，创建两台复印机和两个员工线程，每个员工线程使用一台复印机。然后启动线程，每个员工线程就会按照设计好的步骤进行复印工作。运行例程，控制台输出如下内容：

```
Simth: 开始工作。
Simth  正在使用复印机    佳能
John:  开始工作。
John   正在使用复印机    夏普
John   用完了复印机      夏普
John   正在使用复印机    夏普
Simth  用完了复印机      佳能
Simth  正在使用复印机    佳能
Simth  用完了复印机      佳能
Simth: 完成了工作，下班。
John   用完了复印机      夏普
John   正在使用复印机    夏普
John   用完了复印机      夏普
John:  完成了工作，下班。
```

当然，每次运行程序的输出都可能是不一样的。因为给 John 和 Simth 分配了不同的复印机，所以这两位员工不存在“等待复印机”的情况。但是根据尝试判断，复印机大部分时间都是空闲的，所以不需要给不同的人分配不同的复印机，只要有一台公用的复印机就可以了。

2. 当多个员工共享一台复印机

下面的例程和上面的例程只有一点不同：两个员工只有一台复印机可以使用。那么会发生什么呢？首先看例程代码。

```
package com.javaeasy.simplecopier;

public class OneCopierCopyShop {
    public static void main(String[] args) {
        Copier canon = new Copier("佳能"); // 创建一台佳能复印机
    }
}
```

```

        // 创建一个叫做 Simth 的员工线程，每天工作 1 次，并让他使用佳能复印机
        Employee simth = new Employee("Simth", 1, canon);
        // 创建一个叫做 John 的员工线程，每天工作 2 次，也让他使用佳能复印机
        Employee john = new Employee("John", 2, canon);
        simth.start();                                // 启动两个线程
        john.start();
    }
}

```

运行例程，控制台输出如下内容：

```

Simth: 开始工作。
Simth  正在使用复印机      佳能
John:   开始工作。
John   正在使用复印机      佳能
Simth  用完了复印机        佳能
Simth: 完成了工作，下班。
John   用完了复印机        佳能
John   正在使用复印机      佳能
John   用完了复印机        佳能
John:  完成了工作，下班。

```

从控制台输出可以看出问题来了：在控制台的第 2 行，说明 **Simth** 正在使用佳能复印机，在控制台的第 5 行，说明 **Simth** 结束了此次复印机的使用。但是，在控制台的第 4 行，**John** 却在使用同一个佳能的复印机！这肯定是不能允许的，因为一台复印机同时只能有一个人在使用。

这时程序的状态就好像一个乱了套的复印社。每个员工都只顾做自己的事情，而相互没有协调。如何才能让“一台复印机只能由一个线程使用”呢？这时就需要请出“复印社经理（Java 平台）”，让他来负责协调“员工（线程）”之间的工作了。

□ 理解“多个员工线程同时访问一台复印机实例”的问题。