关于 SwingWorker

Swing 中的并发-使用 SwingWorker 线程模式

本文介绍 Java SE 6 中的 SwingWorker 线程工作模式，翻译 Concurrency in Swing（[url]http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html[/url]）。
author: ZJ 2007-7-16
Blog: [url]http://zhangjunhd.blog.51cto.com/[/url]
本文将讨论并发机制在 Swing 编程中的应用。
谨慎地使用并发机制对 Swing 开发人员来说非常重要。一个好的 Swing 程序使用并发机制来创建不会失去响应的用户接口-不管是什么样的用户交互，程序总能够对其给出响应。创建一个有响应的程序，开发人员必须学会如何在 Swing 框架中使用多线程。
一个 Swing 开发人员将会与下面几类线程打交道：
[1]Initial threads（初始线程），此类线程将执行初始化应用代码。
[2]The event dispatch thread（事件派发线程）,所有的事件处理代码在这里执行。大多数与 Swing 框架交互的代码也必须执行这个线程。
[3]Worker threads（工作线程）,也称作 background threads（后台线程）,此类线程将执行所有消耗时间的任务。
开发人员不需要在代码中显式的创建这些线程：它们是由 runtime 或 Swing 框架提供的。开发人员的工作就是利用这些线程来创建具有响应的，持久的 Swing 程序。如同所有其他在 Java 平台上运行的程序，一个 Swing 程序可以创建额外的线程和线程池，这需要使用本文即将介绍的方法。本文将介绍以上这三种线程。工作线程的讨论将涉及到使用 javax.swing.SwingWorker 类。这个类有许多有用的特性，包括在工作线程任务与其他线程任务之间的通信与协作。

1．初始线程

每个程序都会在应用逻辑开始时生成一系列的线程。在标准的程序中，只有一个这样的线程：这个线程将调用程序主类中的 main 方法。在 applet 中初始线程是 applet 对象的构造子，它将调用 init 方法；这些 actions 可能在一个单一的线程中执行，或在两个或三个不同的线程中，这些都依据 Java 平台的具体实现。在本文中，我们称这类线程为初始线程（initial threads）。
在 Swing 程序中，初始线程没有很多事情要做。它们最基本的任务是创建一个 Runnable 对象，用于初始化 GUI 以及为那些用于执行事件派发线程中的事件的对象编排顺序。一旦 GUI 被创建，程序将主要由 GUI 事件驱动，其中的每个事件驱动将引起一个在事件派发线程中事件的执行。程序代码可以编排额外的任务给事件驱动线程（前提是它们会被很快的执行，这样才不会干扰事件的处理）或创建工作线程（用于执行消耗时间的任务）。
一个初始线程编排 GUI 创建任务是通过调用 javax.swing.SwingUtilities.invokeLater 或 javax.swing.SwingUtilities.invokeAndWait。这两个方法都带有一个唯一的参数：Runnable 用于定义新的任务。它们唯一的区别是：invokerLater 仅仅编排任务并返回；invokeAndWait 将等待任务执行完毕才返回。

看下面示例：

```
SwingUtilities.invokeLater(new Runnable()) {
    public void run() {
       createAndShowGUI();
    }
}
```

在 applet 中，创建 GUI 的任务必须被放入 init 方法中并且使用 invokeAndWait；否则，初始过程将有可能在 GUI 创建完之前完成，这样将有可能出现问题。在其他的情况下，编排 GUI 创建任务通常是初始线程中最后一个被执行的，所以使用 invokeLater 或 invokeAndWait 都可以。

为什么初始线程不直接创建 GUI？因为几乎所有的用于创建和交互 Swing 组件的代码必须在事件派发线程中执行。这个约束将在下文中讨论。

2．事件派发线程

Swing 事件的处理代码在一个特殊的线程中执行，这个线程被称为事件派发线程。大部分调用 Swing 方法的代码都在这个线程中被执行。这样做是必要的，因为大部分 Swing 对象是“非线程安全的”。

可以将代码的执行想象成在事件派发线程中执行一系列短小的任务。大部分任务被事件处理方法调用，诸如 ActionListener.actionPerformed。其余的任务将被程序代码编排，使用 invokeLater 或 invokeAndWait。在事件派发线程中的任务必须能够被快速执行完成，如若不然，未经处理的事件被积压，用户界面将变得“响应迟钝”。

如果你需要确定你的代码是否是在事件派发线程中执行，可调用 javax.swing.SwingUtilities.isEventDispatchThread。

3．工作线程与 SwingWorker

当一个 Swing 程序需要执行一个长时间的任务，通常将使用一个工作线程来完成。每个任务在一个工作线程中执行，它是一个 javax.swing.SwingWorker 类的实例。

SwingWorker 类是抽象类；你必须定义它的子类来创建一个 SwingWorker 对象；通常使用匿名内部类来这做这些。

SwingWorker 提供一些通信与控制的特征：

[1]SwingWorker 的子类可以定义一个方法，done。当后台任务完成的时候，它将自动的被事件派发线程调用。

[2]SwingWorker 类实现 java.util.concurrent.Future。这个接口允许后台任务提供一个返回值给其他线程。该接口中的方法还提供允许撤销后台任务以及确定后台任务是被完成了还是被撤销的功能。

[3]后台任务可以通过调用 SwingWorker.publish 来提供中间结果，事件派发线程将会调用该方法。

[4]后台任务可以定义绑定属性。绑定属性的变化将触发事件，事件派发线程将调用事件处理程序来处理这些被触发的事件。

4．简单的后台任务

下面介绍一个示例，这个任务非常简单，但它是潜在地消耗时间的任务。TumbleItem applet 导入一系列的图片文件。如果这些图片文件是通过初始线程导入的，那么将

在 GUI 出现之前有一段延迟。如果这些图片文件是在事件派发线程中导入的，那么 GUI 将有可能出现临时无法响应的情况。
为了解决这些问题，TumbleItem 类在它初始化时创建并执行了一个 StringWorker 类的实例。这个对象的 doInBackground 方法，在一个工作线程中执行，将图片导入一个 ImageIcon 数组，并且返回它的一个引用。接着 done 方法，在事件派发线程中执行，得到返回的引用，将其放在 applet 类的成员变量 imgs 中。这样做可以允许 TumbleItem 类立刻创建 GUI，而不必等待图片导入完成。
下面的示例代码定义和实现了一个 SwingWorker 对象。

```
SwingWorker worker = new SwingWorker<ImageIcon[], Void>() {
    @Override
    public ImageIcon[] doInBackground() {
        final ImageIcon[] innerImgs = new ImageIcon[nimgs];
        for (int i = 0; i < nimgs; i++) {
            innerImgs[i] = loadImage(i+1);
        }
        return innerImgs;
    }

    @Override
    public void done() {
        //Remove the "Loading images" label.
        animator.removeAll();
        loopslot = -1;
        try {
            imgs = get();
        } catch (InterruptedException ignore) {}
        catch (java.util.concurrent.ExecutionException e) {
            String why = null;
            Throwable cause = e.getCause();
            if (cause != null) {
                why = cause.getMessage();
            } else {
                why = e.getMessage();
            }
            System.err.println("Error retrieving file: " + why);
        }
    }
};
```

所有的继承自 SwingWorker 的子类都必须实现 doInBackground；实现 done 方法是可选的。

注意，SwingWorker 是一个范型类，有两个参数。第一个类型参数指定 doInBackground 的返回类型。同时也是 get 方法的类型，它可以被其他线程调用以获得来自于 doInBackground 的返回值。第二个类型参数指定中间结果的类型，这个例子没有返回中间结果，所以设为 void。
使用 get 方法，可以使对象 imgs 的引用（在工作线程中创建）在事件派发线程中得到使用。这样就可以在线程之间共享对象。
实际上有两个方法来得到 doInBackground 类返回的对象。
[1]调用 SwingWorker.get 没有参数。如果后台任务没有完成，get 方法将阻塞直到它完成。
[2]调用 SwingWorker.get 带参数指定 timeout。如果后台任务没有完成，阻塞直到它完成-除非 timeout 期满，在这种情况下，get 将抛出 java.util.concurrent.TimeoutException。

5．具有中间结果的任务

让一个正在工作的后台任务提供中间结果是很有用处的。后台任务可以调用 SwingWorker.publish 方法来做到这个。这个方法接受许多参数。每个参数必须是由 SwingWorker 的第二个类型参数指定的一种。
可以覆盖（override）SwingWorker.process 来保存由 publish 方法提供的结果。这个方法是由事件派发线程调用的。来自 publish 方法的结果集通常是由一个 process 方法收集的。
我们看一下 Filpper.java 提供的实例。这个程序通过一个后台任务产生一系列的随机布尔值测试 java.util.Random。就好比是一个投硬币试验。为了报告它的结果，后台任务使用了一个对象 FlipPair。

```
private static class FlipPair {
    private final long heads, total;
    FlipPair(long heads, long total) {
       this.heads = heads;
       this.total = total;
    }
}
```

heads 表示 true 的结果；total 表示总的投掷次数。
后台程序是一个 FilpTask 的实例：

```
private class FlipTask extends SwingWorker<Void, FlipPair> {
```

因为任务没有返回一个最终结果，这里不需要指定第一个类型参数是什么，使用 Void。在每次“投掷”后任务调用 publish：

```
@Override
protected Void doInBackground() {
    long heads = 0;
    long total = 0;
    Random random = new Random();
    while (!isCancelled()) {
       total++;
```

```
            if (random.nextBoolean()) {
                heads++;
            }
            publish(new FlipPair(heads, total));
        }
        return null;
    }
```

由于 publish 时常被调用，许多的 FlipPair 值将在 process 方法被事件派发线程调用之前被收集；process 仅仅关注每次返回的最后一组值，使用它来更新 GUI:

```
protected void process(List pairs) {
    FlipPair pair = pairs.get(pairs.size() - 1);
    headsText.setText(String.format("%d", pair.heads));
    totalText.setText(String.format("%d", pair.total));
    devText.setText(String.format("%.10g",
            ((double) pair.heads)/((double) pair.total) - 0.5));
}
```

6．取消后台任务

调用 SwingWorker.cancel 来取消一个正在执行的后台任务。任务必须与它自己的撤销机制一致。有两个方法来做到这一点：

[1]当收到一个 interrupt 时，将被终止。

[2]调用 SwingWorker.isCanceled，如果 SwingWorker 调用 cancel，该方法将返回 true。

7．绑定属性和状态方法

SwingWorker 支持 bound properties，这个在与其他线程通信时很有作用。提供两个绑定属性：progress 和 state。progress 和 state 可以用于触发在事件派发线程中的事件处理任务。

通过实现一个 property change listener，程序可以捕捉到 progress,state 或其他绑定属性的变化。

7.1The progress Bound Variable

Progress 绑定变量是一个整型变量，变化范围由 0 到 100。它预定义了 setter (the protected SwingWorker.setProgress)和 getter (the public SwingWorker.getProgress)方法。

7.2The state Bound Variable

State 绑定变量的变化反映了 SwingWorker 对象在它的生命周期中的变化过程。该变量中包含一个 SwingWorker.StateValue 的枚举类型。可能的值有：

[1]PENDING

这个状态持续的时间为从对象的建立知道 doInBackground 方法被调用。

[2]STARTED

这个状态持续的时间为 doInBackground 方法被调用前一刻直到 done 方法被调用前一刻。

[3]DONE
对象存在的剩余时间将保持这个状态。
需要返回当前 state 的值可调用 SwingWorker.getState。