Java 理论与实践: 做个好的（事件）侦听器

**简介：** 观察者模式在 Swing 开发中很常见，在 GUI 应用程序以外的场景中，它对于消除组件的耦合性也非常有用。但是，仍然存在一些侦听器登记和调用方面的常见缺陷。在 Java 理论与实践 的这一期中，Java 专家 Brian Goetz 就如何做一个好的侦听器，以及如何对您的侦听器也友好，提供了一些感觉很好的建议。

Swing 框架以事件侦听器的形式广泛利用了观察者模式（也称为发布-订阅模式）。Swing 组件作为用户交互的目标，在用户与它们交互的时候触发事件；数据模型类在数据发生变化时触发事件。用这种方式使用观察者，可以让控制器与模型分离，让模型与视图分离，从而简化 GUI 应用程序的开发。

“四人帮”的 *设计模式* 一书（参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp07265/#resources)）把观察者模式描述为：定义对象之间的“一对多”关系，这样一个对象改变状态时，所有它的依赖项都会被通知，并自动更新。观察者模式支持组件之间的松散耦合；组件可以保持它们的状态同步，却不需要直接知道彼此的标识或内部情况，从而促进了组件的重用。

AWT 和 Swing 组件（例如 JButton 或 JTable）使用观察者模式消除了 GUI 事件生成与它们在指定应用程序中的语义之间的耦合。类似地，Swing 的模型类，例如 TableModel 和 TreeModel，也使用观察者消除数据模型表示 与视图生成之间的耦合，从而支持相同数据的多个独立的视图。Swing 定义了 Event 和 EventListener 对象层次结构；可以生成事件的组件，例如 JButton（可视组件） 或 TableModel（数据模型），提供了 addXxxListener() 和 removeXxxListener() 方法，用于侦听器的登记和取消登记。这些类负责决定什么时候它们需要触发事件，什么时候确实触发事件，以及什么时候调用所有登记的侦听器。

为了支持侦听器，对象需要维护一个已登记的侦听器列表，提供侦听器登记和取消登记的手段，并在适当的事件发生时调用每个侦听器。使用和支持侦听器很容易（不仅仅在 GUI 应用程序中），但是在登记接口的两边（它们是支持侦听器的组件和登记侦听器的组件）都应当避免一些缺陷。

**线程安全问题**

通常，调用侦听器的线程与登记侦听器的线程不同。要支持从不同线程登记侦听器，那么不管用什么机制存储和管理活动侦听器列表，这个机制都必须是线程安全的。Sun 的文档中的许多示例使用 Vector 保存侦听器列表，它解决了部分问题，但是没有解决全部问题。在事件触发时，触发它的组件会考虑迭代侦听器列表，并调用每个侦听器，这就带来了并发修改的风险，比如在侦听器列表迭代期间，某个线程偶然想添加或删除一个侦听器。

**管理侦听器列表**

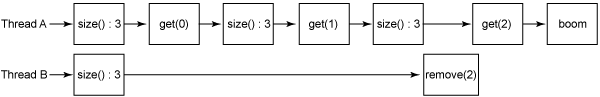
假设您使用 Vector<Listener> 保存侦听器列表。虽然 Vector 类是线程安全的（意味着不需要进行额外的同步就可调用它的方法，没有破坏 Vector 数据结构的风险），但是集合的迭代中包含“检测然后执行”序列，如果在迭代期间集合被修改，就有了失败的风险。假设迭代开始时列表中有三个侦听器。在迭代 Vector 时，重复调用 size() 和 get() 方法，直到所有元素都检索完，如清单 1 所示：

**清单 1. Vector 的不安全迭代**

|  |
| --- |
| Vector<Listener> v;  for (int i=0; i<v.size(); i++)  v.get(i).eventHappened(event); |

但是，如果恰好就在最后一次调用 Vector.size() 之后，有人从列表中删除了一个侦听器，会发生什么呢？现在，Vector.get() 将返回 null （这是对的，因为从上次检测 vector 的状态以来，它的状态已经变了），而在试图调用eventHappened() 时，会抛出 NullPointerException。这是“检测然后执行”序列的一个示例 —— 检测是否存在更多元素，如果存在，就取得下一元素 —— 但是在存在并发修改的情况下，检测之后状态可能已经变化。图 1 演示了这个问题：

**图 1. 并发迭代和修改，造成意料之外的失败**

这个问题的一个解决方案是在迭代期间持有对 Vector 的锁；另一个方案是克隆 Vector 或调用它的 toArray() 方法，在每次发生事件时检索它的内容。所有这两个方法都有性能上的问题：第一个的风险是在迭代期间，会把其他想访问侦听器列表的线程锁在外面；第二个则要创建临时对象，而且每次事件发生时都要拷贝列表。

如果用迭代器（Iterator）去遍历侦听器列表，也会有同样的问题，只是表现略有不同； iterator() 实现不抛出NullPointerException，它在探测到迭代开始之后集合发生修改时，会抛出 ConcurrentModificationException。同样，也可以通过在迭代期间锁定集合防止这个问题。

java.util.concurrent 中的 CopyOnWriteArrayList 类，能够帮助防止这个问题。它实现了 List，而且是线程安全的，但是它的迭代器不会抛出 ConcurrentModificationException，遍历期间也不要求额外的锁定。这种特性组合是通过在每次列表修改时，在内部重新分配并拷贝列表内容而实现的，这样，遍历内容的线程不需要处理变化 —— 从它们的角度来说，列表的内容在遍历期间保持不变。虽然这听起来可能没效率，但是请记住，在多数观察者情况下，每个组件只有少量侦听器，遍历的数量远远超过插入和删除的数量。所以更快的迭代可以补偿较慢的变化过程，并提供更好的并发性，因为多个线程可以同时迭代列表。

**初始化的安全风险**

从侦听器的构造函数中登记它很诱惑人，但是这是一个应当避免的诱惑。它仅会造成“失效侦听器（lapsed listener）的问题（我稍后讨论它），而且还会造成多个线程安全问题。清单 2 显示了一个看起来没什么害处的同时构造和登记侦听器的企图。问题是：它造成到对象的“this”引用在对象完全构造完成之前转义。虽然看起来没什么害处，因为登记是构造函数做的最后一件事，但是看到的东西是有欺骗性的：

**清单 2. 事件侦听器允许“this”引用转义，造成问题**

|  |
| --- |
| public class EventListener {  public EventListener(EventSource eventSource) {  // do our initialization  ...  // register ourselves with the event source  eventSource.registerListener(this);  }  public onEvent(Event e) {  // handle the event  }  } |

在继承事件侦听器的时候，会出现这种方法的一个风险：这时，子类构造函数做的任何工作都是在 EventListener 构造函数运行之后进行的，也就是在 EventListener 发布之后，所以会造成争用情况。在某些不幸的时候，清单 3 中的 onEvent 方法会在列表字段还没初始化之前就被调用，从而在取消 final 字段的引用时，会生成非常让人困惑的NullPointerException 异常：

**清单 3. 继承清单 2 的 EventListener 类造成的问题**

|  |
| --- |
| public class RecordingEventListener extends EventListener {  private final ArrayList<Event> list;  public RecordingEventListener(EventSource eventSource) {  super(eventSource);  list = Collections.synchronizedList(new ArrayList<Event>());  }  public onEvent(Event e) {  list.add(e);  super.onEvent(e);  }  } |

即使侦听器类是 final 的，不能派生子类，也不应当允许“this”引用在构造函数中转义 —— 这样做会危害 Java 内存模型的某些安全保证。如果“this”这个词不会出现在程序中，就可让“this”引用转义；发布一个非静态内部类实例可以达到相同的效果，因为内部类持有对它包围的对象的“this”引用的引用。偶然地允许“this”引用转义的最常见原因，就是登记侦听器，如清单 4 所示。**事件侦听器不应当在构造函数中登记！**

这登记了一个没有构造完成的对象  
**清单 4. 通过发布内部类实例，显式地允许“this”引用转义**

|  |
| --- |
| public class EventListener2 {  public EventListener2(EventSource eventSource) {  eventSource.registerListener(  new EventListener() {  public void onEvent(Event e) {  eventReceived(e);  }  });  }  public void eventReceived(Event e) {  }  } |

**侦听器线程安全**

使用侦听器造成的第三个线程安全问题来自这个事实：侦听器可能想访问应用程序数据，而调用侦听器的线程通常不直接在应用程序的控制之下。如果在 JButton 或其他 Swing 组件上登记侦听器，那么会从 EDT 调用该侦听器。侦听器的代码可以从 EDT 安全地调用 Swing 组件上的方法，但是如果对象本身不是线程安全的，那么从侦听器访问应用程序对象会给应用程序增加新的线程安全需求。

Swing 组件生成的事件是用户交互的结果，但是 Swing 模型类是在 fireXxxEvent() 方法被调用的时候生成事件。这些方法又会在调用它们的线程中调用侦听器。因为 Swing 模型类不是线程安全的，而且假设被限制在 EDT 内，所以对fireXxxEvent() 的任何调用也都应当从 EDT 执行。如果想从另外的线程触发事件，那么应当用 Swing 的 invokeLater() 功能让方法转而在 EDT 内调用。一般来说，要注意调用事件侦听器的线程，还要保证它们涉及的任何对象或者是线程安全的，或者在访问它们的地方，受到适当的同步（或者是 Swing 模型类的线程约束）的保护。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp07265/#ibm-pcon)

**失效侦听器**

不管什么时候使用观察者模式，都耦合着两个独立组件 —— 观察者和被观察者，它们通常有不同的生命周期。登记侦听器的后果之一就是：它在被观察对象和侦听器之间建立起很强的引用关系，这种关系防止侦听器（以及它引用的对象）被垃圾收集，直到侦听器取消登记为止。在许多情况下，侦听器的生命周期至少要和被观察的组件一样长 —— 许多侦听器会在整个应用程序期间都存在。但是在某些情况下，应当短期存在的侦听器最后变成了永久的，它们这种无意识的拖延的证据就是应用程序性能变慢、高于必需的内存使用。

“失效侦听器”的问题可以由设计级别上的不小心造成：没有恰当地考虑包含的对象的寿命，或者由于松懈的编码。侦听器登记和取消登记应当结对进行。但是即使这么做，也必须保证是在正确的时间执行取消登记。清单 5 显示了会造成失效侦听器的编码习惯的示例。它在组件上登记侦听器，执行某些动作，然后取消登记侦听器：

**清单 5. 有造成失效侦听器风险的代码**

|  |
| --- |
| public void processFile(String filename) throws IOException {  cancelButton.registerListener(this);  // open file, read it, process it  // might throw IOException  cancelButton.unregisterListener(this);  } |

清单 5 的问题是：如果文件处理代码抛出了 IOException —— 这是很有可能的 —— 那么侦听器就永远不会取消登记，这就意味着它永远不会被垃圾收集。取消登记的操作应当在 finally 块中进行，这样，processFile() 方法的所有出口都会执行它。

有时推荐的一个处理失效侦听器的方法是使用弱引用。虽然这种方法可行，但是实现起来很麻烦。要让它工作，需要找到另外一个对象，它的生命周期恰好是侦听器的生命周期，并安排它持有对侦听器的强引用，这可不是件容易的事。

另外一项可以用来找到隐藏失效侦听器的技术是：防止指定侦听器对象在指定事件源上登记两次。这种情况通常是 bug 的迹象 —— 侦听器登记了，但是没有取消登记，然后再次登记。不用检测问题，就能缓解这个问题的影响的一种方式是：使用 Set 代替 List 来存储侦听器；或者也可以检测 List，在登记侦听器之前检查是否已经登记了，如果已经登记，就抛出异常（或记录错误），这样就可以搜集编码错误的证据，并采取行动。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp07265/#ibm-pcon)

**其他侦听器问题**

在编写侦听器时，应当一直注意它们将要执行的环境。不仅要注意线程安全问题，还需要记住：侦听器也可以用其他方式为它的调用者把事情搞糟。侦听器 *不该* 做的一件事是：阻塞相当长一段时间（长得可以感觉得到）；调用它的执行上下文很可能希望迅速返回控制。如果侦听器要执行一个可能比较费时的操作，例如处理大型文本，或者要做的工作可能阻塞，例如执行 socket IO，那么侦听器应当把这些操作安排在另一个线程中进行，这样它就可以迅速返回它的调用者。

对于不小心的事件源，侦听器会造成麻烦的另一个方式是：抛出未检测的异常。虽然大多数时候，我们不会故意抛出未检测异常，但是确实有些时候会发生这种情况。如果使用清单 1 的方式调用侦听器，列表中的第二个侦听器就会抛出未检测异常，那么不仅后续的侦听器得不到调用（可能造成应用程序处在不一致的状态），而且有可能把执行它的线程破坏掉，从而造成局部应用程序失败。

在调用未知代码（侦听器就是这样的代码）时，谨慎的方式是在 try-catch 块中执行它，这样，行为有误的侦听器不会造成更多不必要的破坏。对于抛出未检测异常的侦听器，您可能想自动对它取消登记，毕竟，抛出未检测异常就证明侦听器坏掉了。（您可能还想记录这个错误或者提醒用户注意，好让用户能够知道为什么程序停止像期望的那样继续工作。）清单 6 显示了这种方式的一个示例，它在迭代循环内部嵌套了 try-catch 块：

**清单 6. 健壮的侦听器调用**

|  |
| --- |
| List<Listener> list;  for (Iterator<Listener> i=list.iterator; i.hasNext(); ) {  Listener l = i.next();  try {  l.eventHappened(event);  }  catch (RuntimeException e) {  log("Unexpected exception in listener", e);  i.remove();  }  } |

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp07265/#ibm-pcon)

**结束语**

观察者模式对于创建松散耦合的组件、鼓励组件重用非常有用，但是它有一些风险，侦听器的编写者和组件的编写者都应当注意。在登记侦听器时，应当一直注意侦听器的生命周期。如果侦听器的寿命应当比应用程序的短，那么请确保取消它的登记，这样它就可以被垃圾收集。在编写侦听器和组件时，请注意它包含的线程安全性问题。侦听器涉及的任何对象，都应当是线程安全的，或者是受线程约束的对象（例如 Swing 模型），侦听器应当确定自己正在正确的线程中执行。