对于多线程编程经验较少的程序员而言，开发多核系统软件将面临两个方面的问题：首先，并发会给 Java 程序引入新的缺陷，如数据速度和死锁，它们是非常难以复现和发现的。其次，许多程序员并不知道特定多线程编程方法的微妙细节，而这可能会导致代码错误。

为了避免给并发程序引入缺陷，Java 程序员必须了解如何识别缺陷在多线程代码中很可能出现的关键位置，然后才能够编写出没有缺陷的软件。在本文中，我们将帮助 Java 开发人员在理解并发编程早期和中期会遇到的问题。我们并不会关注于常见的 Java 并发缺陷模式，如双重检查锁、循环等待和等待不在循环内项目，我们将介绍 6 个鲜为人知的模式，但是却经常出现在真实的 Java 应用程序中。事实上，我们的前两个例子就是在两个流行的 Web 服务器上发现的缺陷。

**1. Jetty 的一个反模式**

我们要介绍的第一个并发缺陷是在广泛使用的开源 HTTP 服务器 Jetty 上发现的。这是已经过 Jetty 社区确认的一个真实缺陷（见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-concurrencybugpatterns/index.html#resources) 的缺陷报告）。

**清单 1. 在一个易变（volatile）域上不获取锁的情况下执行非原子操作**

|  |
| --- |
| // Jetty 7.1.0,  // org.eclipse.jetty.io.nio,  // SelectorManager.java, line 105  **private volatile** int \_set;  ......  **public void** register(SocketChannel channel, Object att)  {  int s=\_set++;  ......  }  ......  **public void** addChange(Object point)  {  **synchronized** (\_changes)  {  ......  }  } |

[清单 1](http://www.ibm.com/developerworks/cn/java/j-concurrencybugpatterns/index.html#listing1) 中的错误有以下几个部分：

* 首先，\_set 被声明为 volatile，这表示这个域可以由多个线程访问。
* 但是， \_set++ 并不是原子操作，这意味着它不会以单个不可分割操作执行。相反，它只是包含三个具体操作序列的简写方法：read-modify-write。
* 最后， \_set++ 并没有锁保护。如果方法 register 同时由多个线程调用，那么它会产生一个竞争状态，导致出现错误的 \_set值。

您的代码也可能和 Jetty 一样出现这种类型的错误，所以让我更详细地分析一下它是如何发生的。

**缺陷模式构成元素**

分析它的逻辑执行代码有助于弄清楚这个缺陷模式。变量 *i* 的操作，如

|  |
| --- |
| i++  --i  i += 1  i -= 1  i \*= 2 |

等，另外就是非原子操作（即 read-modify-write）。如果您知道 volatile 关键字在 Java 语言中仅仅保证变量的可见性，而不保证原子性，那么这应该会引起您的注意。一个易变域上的不受锁保护的非原子操作*可能* 会产生一个竞争状况 — 但是只有在多个线程并发访问非原子操作时才可能出现。

在一个线程安全的程序中，只有一个写线程能够修改这个变量；而其他的线程则可以读取 volatile 声明变量的最新值。

所以，代码是否有问题取决于有多少线程能够并发地访问这个操作。如果这个非原子操作仅仅由一个线程调用，由于是有一个开始联合关系或者外部锁，那么这样的编码方法也是线程安全的。

一定要谨记 volatile 关键字在 Java 代码中仅仅保证这个变量是可见的：它不保证原子性。在那些非原子且可由多个线程访问的易变操作中，一定不能够依赖于 volatile 的同步机制。相反，要使用 java.util.concurrent 包的同步语句、锁类和原子类。它们在设计上能够保证程序是线程安全的。

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

**2. 在易变域上的同步**

在 Java 语言中，我们使用了同步语句来获取互斥锁，这可以保护多线程系统的共享资源访问。然而，易变域的同步中会有一个漏洞，它可能破坏互斥。解决的方法是一定要将同步的域声明为 private final。让我们先来仔细看看问题是如何产生的。

**修改域上的同时访问锁**

同步语句是由同步域所引用对象保护的，而不是由域本身保护的。如果一个同步域是易变的（这意味着这个域在初始化之后可能在程序的其他位置赋值），这很可能不是有用的语义，因为不同的线程可能同时访问不同的对象。

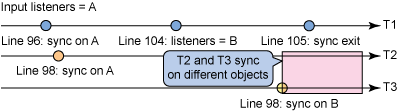
您可以在清单 2 中看到这个问题，这是节选自开源 Web 应用服务器 Tomcat 的代码片断：

**清单 2. Tomcat 的错误**

|  |
| --- |
| 96: **public void** addInstanceListener(InstanceListener listener) {  97:  98: **synchronized** (listeners) {  99: InstanceListener results[] =  100: **new** InstanceListener[listeners.length + 1];  101: **for** (int i = 0; i < listeners.length; i++)  102: results[i] = listeners[i];  103: results[listeners.length] = listener;  104: listeners = results;  105: }  106:  107:} |

假设 listeners 引用的是数组 A，而线程 T1 首先获取数组 A 的锁，然后开始创建数组 B。同时，T2 开始执行，并且由于数据 A 的锁而被阻挡。当 T1 完成数组 B 的 listeners 设置后，退出这个语句，T2 会锁住数组 A，然后开始复制数组 B。然后 T3 开始执行，并锁住数组 B。因为它们获得了不同的锁，T2 和 T3 现在可以同时复制数组 B。

图 1 更进一步地说明了这个执行顺序：

**图 1. 由于易变域的同步而失去互斥锁**  
 

无数的意外行为可能会导致这种情况出现。至少，其中一个新的监听器可能会丢失，或者其中一个线程可能会发生ArrayIndexOutOfBoundsException 异常（由于 listeners 引用及其长度可能在方法的任意时刻发生变化）。

好的做法是总是将同步域声明为 private final，这能够保证锁对象保持不变，并且保证了互斥（mutex）。

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

**3. java.util.concurrent 锁泄漏**

一个实现 java.util.concurrent.locks.Lock 接口的锁控制着多个线程是如何访问一个共享资源的。这些锁不需要使用语句结构，所以它们比同步方法或语句更灵活。然而，这种灵活性可能导致编码错误，因为不使用语句的锁是不会自动释放的。如果一个Lock.lock() 调用没有在同一个实例上执行相应的 unlock() 调用，其结果就可能造成一个锁泄漏。

如果忽视关键代码中的方法行为，我们就很容易造成 java.util.concurrent 锁泄漏，有可能抛出的异常。您可以从清单 3 的代码看到这一点，其中 accessResource 方法在访问共享资源时抛出了一个 InterruptedException 异常。结果，unlock() 是不会被调用的。

**清单 3. 分析一个锁泄漏**

|  |
| --- |
| **private final** Lock lock = new ReentrantLock();  **public void** lockLeak() {  lock.lock();  try {  // access the shared resource  accessResource();  lock.unlock();  } catch (Exception e) {}  **public void** accessResource() throws InterruptedException {...} |

要保证锁得到释放，我们只需要在每一个 lock 之后对应执行一个 unlock 方法，而且它们应该置于 try-finally 复杂语句中。清单 4 说明了这种方法：

**清单 4. 总是将 unlock 调用置于 finally 语句中**

|  |
| --- |
| **private final** Lock lock = new ReentrantLock();  **public void** lockLeak() {  lock.lock();  try {  // access the shared resource  accessResource();  } catch (Exception e) {}  finally {  lock.unlock();  }  public void accessResource() throws InterruptedException {...} |

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

**4. 同步语句的性能优化**

有一些并发缺陷有时不会使代码出错，但是它们可能会降低应用程序的性能。考虑清单 5 中的 synchronized 语句：

**清单 5. 带有不变代码的同步语句**

|  |
| --- |
| public class Operator {  private int generation = 0; //shared variable  private float totalAmount = 0; //shared variable  private final Object lock = new Object();  public void workOn(List<Operand> operands) {  synchronized (lock) {  int curGeneration = generation; //requires synch  float amountForThisWork = 0;  for (Operand o : operands) {  o.setGeneration(curGeneration);  amountForThisWork += o.amount;  }  totalAmount += amountForThisWork; //requires synch  generation++; //requires synch  }  }  } |

清单 5 代码中两个共享变量的访问是同步且正确的，但是如果仔细检查，您会注意到 synchronized 语句所需要进行的计算过多。我们可以通过调整代码顺序来解决这个问题，如清单 6 所示：

**清单 6. 没有不变代码的同步语句**

|  |
| --- |
| public void workOn(List<Operand> operands) {  int curGeneration;  float amountForThisWork = 0;  synchronized (lock) {  int curGeneration = generation++;  }  for (Operand o : operands) {  o.setGeneration(curGeneration);  amountForThisWork += o.amount;  }  synchronized (lock)  totalAmount += amountForThisWork;  }  } |

第二个版本代码在多核机器上执行效果会更好。其原因是清单 5 的同步代码阻止了并行执行。这个方法循环可能会消耗大量的计算时间。在清单 6 中，循环被移出同步语句，所以它可能由多个线程并行执行。一般而言，在保证线程安全的前提下要尽可能地简化同步语句。

**其他方面……**

您可能会有疑问，是否使用 AtomicInteger 和AtomicFloat 来表示清单 5 和 6 中的两个共享变量，然后去掉所有同步代码，是否会更好一些。这取决于其他方法是如何处理这些变量的，以及它们之间有什么样的依赖关系。

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

**5. 多阶段访问**

假设您的应用程序有两个表：第一个表将员工姓名映射到一个员工号，另一个将这个员工号映射到一个薪水记录。这些数据需要支持并发访问和更新，而您可以通过线程安全的 ConcurrentHashMap 实现，如清单 7 所示：

**清单 7. 两个阶段的访问**

|  |
| --- |
| public class Employees {  private final ConcurrentHashMap<String,Integer> nameToNumber;  private final ConcurrentHashMap<Integer,Salary> numberToSalary;  ... various methods for adding, removing, getting, etc...  public int geBonusFor(String name) {  Integer serialNum = nameToNumber.get(name);  Salary salary = numberToSalary.get(serialNum);  return salary.getBonus();  }  } |

这种方法看起来是线程安全的，但是事实上不是这样的。它的问题是 getBonusFor 方法并不是线程安全的。在获取这个序列号和使用它获取薪水之间，另一个线程可能从两个表删除员工信息。在这种情况下，第二个映射访问可能会返回 null，并抛出一个异常。

保证每一个 Map 本身的线程安全是不够的。它们之间存在一个依赖关系，而且访问这两个 Map 的一些操作必须是原子操作。在这里，您可以使用非线程安全的容器（如 java.util.HashMap），然后使用显式的同步语句来保护每一个访问，从而实现线程安全。然后这个同步语句可以在需要时包含这两个访问。

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

**6. 对称锁死锁**

可以考虑使用一个线程安全容器类 — 一个保证用户操作线程安全的数据结构。（这与 java.util 中的大多数容器不同，它不需要用户同步容器的使用。）在清单 8 中，一个可修改的成员变量负责保存数据，而一个锁对象则保护所有对它的访问。

**清单 8. 一个线程安全的容器**

|  |
| --- |
| public <E> class ConcurrentHeap {  private E[] elements;  private final Object lock = new Object(); //protects elements  public void add (E newElement) {  synchronized(lock) {  ... //manipulate elements  }  }  public E removeTop() {  synchronized(lock) {  E top = elements[0];  ... //manipulate elements  return top;  }  }  } |

现在让我添加一个方法，使用另一个实例，并将它的所有元素添加到当前的实例中。这个方法需要访问这两个实例的 elements 成员，如清单 9 所示：

**清单 9. 下面代码会产生一个死锁**

|  |
| --- |
| public void addAll(ConcurrentHeap other) {  synchronized(other.lock) {  synchronized(this.lock) {  ... //manipulate other.elements and this.elements  }  }  } |

**不仅仅发生在容器上**

这个对称锁死锁场景是很常见的，因为它出现在 Java 1.4 版本上，其中 Collections.synchronized 方法返回的一些同步容器会发生死锁。但是，不仅仅容器容易受到对称锁死锁的影响。如果一个类有一个方法使用同一个类的其他实例作为参数，那么这个类对这两个实例成员的操作也必须是原子的。其中 compareTo 和 equals 方法就是很好的两个例子。

您认识到了死锁的可能性吗？假设一个程序只有两个实例 heap1 和heap2。如果其中一个线程调用了 heap1.addAll(heap2)，而另一个线程同时调用 heap2.addAll(heap1)，那么这两个线程就可能遇到死锁。换言之，假设第一个线程获得了 heap2 的锁，但是它开始执行之前，第二个线程就开始执行方法，同时获取了 heap1 锁。结果，每一个线程都会等待另一个线程所保持的锁。

您可以通过确定实例顺序来防止对称锁死锁，这样当需要获取两个实例的锁时，其顺序是动态计算得到的，并决定哪一个锁先获取到。Brian Goetz 在他撰写的书 *Java Concurrency in Practice* 中详细讨论这个方法（见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-concurrencybugpatterns/index.html#resources)）。

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

**结束语**

许多 Java 开发人员都只是刚刚开始了解多核环境并发程序的编写方法。在这个过程中，我们要放弃所掌握的单线程编写方法，而使用更复杂的多线程环境方法。研究并发缺陷模式是一个发现多线程编程问题的好方法，也有利于掌握这些方法的微妙之处。

您可以学习从整体上把握缺陷模式的方法，这样在您编写代码或检查代码时就能够发现一些特定的问题线索。您也可以使用一些静态分析工具来发现问题。FindBugs 就是一个能够查找代码中可能的缺陷模式的开源静态分析工具。事实上，FindBugs 可用于检查本文讨论的第二和第三个缺陷模式。

静态分析工具的另一个常见的缺点是它们会产生错误的警告，这样您可能会浪费更多时间去检查本来不是缺陷的问题。但是，现在出现了新的更适合于测试并发程序的动态分析工具。其中两种是 IBM® Multicore Software Development Kit (MSDK) 和 ConcurrentTesting (ConTest)，它们都可以从 IBM alphaWorks 上免费下载。

**参考资料**

**学习**

* [*Java Concurrency in Practice*](http://my.safaribooksonline.com/programming/java/0321349601)（Brian Goetz，Addison-Wesley，2006 年）：Brian Goetz 撰写的一本权威书籍，介绍了如何在多个实例上同步时防止死锁。
* [*Java 理论与实践*](http://www.ibm.com/developerworks/cn/java/j-jtp/)：Goetz 的另一个长期维护的 developerWorks 系列文章，从中可了解并发编程建议，包括 [线程池和工作队列](http://www.ibm.com/developerworks/cn/java/j-jtp0730/) 的深入讨论（2002 年 7 月），[java.util.concurrent.lock](http://www.ibm.com/developerworks/cn/java/j-jtp10264/)（2004 年 10 月），[volatile 变量](http://www.ibm.com/developerworks/cn/java/j-jtp06197.html)（2007 年 6 月）和[java.util.concurrent 中的分叉-联合](http://www.ibm.com/developerworks/cn/java/j-jtp11137.html)（2007 年 11 月）。
* “[关于 java.util.concurrent 您不知道的 5 件事，第 1 部分](http://www.ibm.com/developerworks/cn/java/j-5things4.html?ca=drs-)”（Ted Neward，developerWorks，2010 年 5 月）：讨论CopyOnWriteArrayList、BlockingQueue 和 ConcurrentMap 等类如何改变用于并发编程的标准 Java 容器类。
* “[使用 GPars 解决常见并发问题](http://www.ibm.com/developerworks/cn/java/j-gpars/)”（Alex Miller，developerWorks，2010 年 9 月）：一个基于 Groovy 的并发编程库封装了所有的并发编程模型，如分叉/联合、参与者、代理和执行者等。
* [Jetty-1187: Non-atomic self-increment operation on volatile field \_set in class SelectorManager](http://jira.codehaus.org/browse/JETTY-1187)：这篇缺陷报告详细讨论了本文提到的 Jetty 反模式及其解决方法。
* “[FindBugs，第 1 部分: 提高代码质量](http://www.ibm.com/developerworks/cn/java/j-findbug1/)”（Chris Grindstaff，developerWorks，2004 年 5 月）：了解使用良好的静态分析工具的好处，然后了解如何使用一个最好的静态分析工具。
* “[IBM intros Multicore SDK](http://www.drdobbs.com/high-performance-computing/218400937)”（Dr Dobb's Journal，2009 年 7 月）：用于开发多核平台上并行程序的 IBM alphaWorks 工具套件的概述。
* 浏览 [Java 技术书店](http://www.ibm.com/developerworks/apps/SendTo?bookstore=safari) 中关于这个技术主题及其他技术主题的书籍。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：这里有数百篇关于 Java 编程各个方面的文章。

**获得产品和技术**

* [IBM Multicore SDK](http://alphaworks.ibm.com/tech/msdk)：这个工具套件可用于查找 Java 多线程程序中的数据竞争、死锁和锁争用。
* [IBM ConcurrentTesting 工具](http://alphaworks.ibm.com/tech/contest)：ConcurrentTesting 可用于多线程应用程序的单元测试，它有助于减少并行和分布式 Java 程序中与并发有关的缺陷。