Java 理论与实践: 变还是不变

**简介：** 不变对象具有许多能更方便地使用它们的特性，包括不严格的同步需求和不必考虑数据讹误就能自由地共享和高速缓存对象引用。尽管不变性可能未必对于所有类都有意义，但大多数程序中至少有一些类将受益于不可变。在本月的 Java 理论与实践中，Brian Goetz 说明了不变性的一些长处和构造不变类的一些准则。

不变对象是指在实例化后其外部可见状态无法更改的对象。Java 类库中的 String 、 Integer 和 BigDecimal 类就是不变对象的示例 ― 它们表示在对象的生命期内无法更改的单个值。

**不变性的长处**

如果正确使用不变类，它们会极大地简化编程。因为它们只能处于一种状态，所以只要正确构造了它们，就决不会陷入不一致的状态。您不必复制或克隆不变对象，就能自由地共享和高速缓存对它们的引用；您可以高速缓存它们的字段或其方法的结果，而不用担心值会不会变成失效的或与对象的其它状态不一致。不变类通常产生最好的映射键。而且，它们本来就是线程安全的，所以不必在线程间同步对它们的访问。

**自由高速缓存**

因为不变对象的值没有更改的危险，所以可以自由地高速缓存对它们的引用，而且可以肯定以后的引用仍将引用同一个值。同样地，因为它们的特性无法更改，所以您可以高速缓存它们的字段和其方法的结果。

如果对象是可变的，就必须在存储对其的引用时引起注意。请考虑清单 1 中的代码，其中排列了两个由调度程序执行的任务。目的是：现在启动第一个任务，而在某一天启动第二个任务。

**清单 1. 可变的 Date 对象的潜在问题**

|  |
| --- |
| Date d = new Date();  Scheduler.scheduleTask(task1, d);  d.setTime(d.getTime() + ONE\_DAY);  scheduler.scheduleTask(task2, d); |

因为 Date 是可变的，所以 scheduleTask 方法必须小心地用防范措施将日期参数复制（可能通过 clone() ）到它的内部数据结构中。不然， task1 和 task2 可能都在明天执行，这可不是所期望的。更糟的是，任务调度程序所用的内部数据结构会变成讹误。在编写象 scheduleTask() 这样的方法时，极其容易忘记用防范措施复制日期参数。如果忘记这样做，您就制造了一个难以捕捉的错误，这个错误不会马上显现出来，而且当它暴露时人们要花较长的时间才会捕捉到。不变的Date 类不可能发生这类错误。

**固有的线程安全**

大多数的线程安全问题发生在当多个线程正在试图并发地修改一个对象的状态（写-写冲突）时，或当一个线程正试图访问一个对象的状态，而另一个线程正在修改它（读-写冲突）时。要防止这样的冲突，必须同步对共享对象的访问，以便在对象处于不一致状态时其它线程不能访问它们。正确地做到这一点会很难，需要大量文档来确保正确地扩展程序，还可能对性能产生不利后果。只要正确构造了不变对象（这意味着不让对象引用从构造函数中转义），就使它们免除了同步访问的要求，因为无法更改它们的状态，从而就不可能存在写-写冲突或读-写冲突。

不用同步就能自由地在线程间共享对不变对象的引用，可以极大地简化编写并发程序的过程，并减少程序可能存在的潜在并发错误的数量。

**在恶意运行的代码面前是安全的**

把对象当作参数的方法不应变更那些对象的状态，除非文档明确说明可以这样做，或者实际上这些方法具有该对象的所有权。当我们将一个对象传递给普通方法时，通常不希望对象返回时已被更改。但是，使用可变对象时，完全会是这样的。如果将 java.awt.Point 传递给诸如 Component.setLocation() 的方法，根本不会阻止 setLocation 修改我们传入的 Point的位置，也不会阻止 setLocation 存储对该点的引用并稍后在另一个方法中更改它。（当然， Component 不这样做，因为它不鲁莽，但是并不是所有类都那么客气。）现在， Point 的状态已在我们不知道的情况下更改了，其结果具有潜在危险 ― 当点实际上在另一个位置时，我们仍认为它在原来的位置。然而，如果 Point 是不变的，那么这种恶意的代码就不能以如此令人混乱而危险的方法修改我们的程序状态了。

**良好的键**

不变对象产生最好的 HashMap 或 HashSet 键。有些可变对象根据其状态会更改它们的 hashCode() 值（如清单 2 中的StringHolder 示例类）。如果使用这种可变对象作为 HashSet 键，然后对象更改了其状态，那么就会对 HashSet 实现引起混乱 ― 如果枚举集合，该对象仍将出现，但如果用 contains() 查询集合，它就可能不出现。无需多说，这会引起某些混乱的行为。说明这一情况的清单 2 中的代码将打印“false”、“1”和“moo”。

**清单 2. 可变 StringHolder 类，不适合用作键**

|  |
| --- |
| public class StringHolder {  private String string;  public StringHolder(String s) {  this.string = s;  }  public String getString() {  return string;  }  public void setString(String string) {  this.string = string;  }  public boolean equals(Object o) {  if (this == o)  return true;  else if (o == null || !(o instanceof StringHolder))  return false;  else {  final StringHolder other = (StringHolder) o;  if (string == null)  return (other.string == null);  else  return string.equals(other.string);  }  }  public int hashCode() {  return (string != null ? string.hashCode() : 0);  }  public String toString() {  return string;  }  ...  StringHolder sh = new StringHolder("blert");  HashSet h = new HashSet();  h.add(sh);  sh.setString("moo");  System.out.println(h.contains(sh));  System.out.println(h.size());  System.out.println(h.iterator().next());  } |

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02183/#ibm-pcon)

**何时使用不变类**

不变类最适合表示抽象数据类型（如数字、枚举类型或颜色）的值。Java 类库中的基本数字类（如 Integer 、 Long 和Float ）都是不变的，其它标准数字类型（如 BigInteger 和 BigDecimal ）也是不变的。表示复数或精度任意的有理数的类将比较适合于不变性。甚至包含许多离散值的抽象类型（如向量或矩阵）也很适合实现为不变类，这取决于您的应用程序。

## Flyweight 模式

不变性启用了 Flyweight 模式，该模式利用共享使得用对象有效地表示大量细颗粒度的对象变得容易。例如，您可能希望用一个对象来表示字处理文档中的每个字符或图像中的每个像素，但这一策略的幼稚实现将会对内存使用和内存管理开销产生高得惊人的花费。Flyweight 模式采用工厂方法来分配对不变的细颗粒度对象的引用，并通过仅使一个对象实例与字母“a”对应来利用共享缩减对象数。有关 Flyweight 模式的更多信息，请参阅经典书籍 Design Patterns（Gamma 等著；请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp02183/#resources)）。

Java 类库中不变性的另一个不错的示例是 java.awt.Color 。在某些颜色表示法（如 RGB、HSB 或 CMYK）中，颜色通常表示为一组有序的数字值，但把一种颜色当作颜色空间中的一个特异值，而不是一组有序的独立可寻址的值更有意义，因此将 Color作为不变类实现是有道理的。

如果要表示的对象是多个基本值的容器（如：点、向量、矩阵或 RGB 颜色），是用可变对象还是用不变对象表示？答案是……要看情况而定。要如何使用它们？它们主要用来表示多维值（如像素的颜色），还是仅仅用作其它对象的一组相关特性集合（如窗口的高度和宽度）的容器？这些特性多久更改一次？如果更改它们，那么各个组件值在应用程序中是否有其自己的含义呢？

事件是另一个适合用不变类实现的好示例。事件的生命期较短，而且常常会在创建它们的线程以外的线程中消耗，所以使它们成为不变的是利大于弊。大多数 AWT 事件类都没有作为严格的不变类来实现，而是可以有小小的修改。同样地，在使用一定形式的消息传递以在组件间通信的系统中，使消息对象成为不变的或许是明智的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02183/#ibm-pcon)

**编写不变类的准则**

编写不变类很容易。如果以下几点都为真，那么类就是不变的：

* 它的所有字段都是 final
* 该类声明为 final
* 不允许 this 引用在构造期间转义
* 任何包含对可变对象（如数组、集合或类似 Date 的可变类）引用的字段：
  + 是私有的
  + 从不被返回，也不以其它方式公开给调用程序
  + 是对它们所引用对象的唯一引用
  + 构造后不会更改被引用对象的状态

最后一组要求似乎挺复杂的，但其基本上意味着如果要存储对数组或其它可变对象的引用，就必须确保您的类对该可变对象拥有独占访问权（因为不然的话，其它类能够更改其状态），而且在构造后您不修改其状态。为允许不变对象存储对数组的引用，这种复杂性是必要的，因为 Java 语言没有办法强制不对 final 数组的元素进行修改。注：如果从传递给构造函数的参数中初始化数组引用或其它可变字段，您必须用防范措施将调用程序提供的参数或您无法确保具有独占访问权的其它信息复制到数组。否则，调用程序会在调用构造函数之后，修改数组的状态。清单 3 显示了编写一个存储调用程序提供的数组的不变对象的构造函数的正确方法（和错误方法）。

**清单 3. 对不变对象编码的正确和错误方法**

|  |
| --- |
| class ImmutableArrayHolder {  private final int[] theArray;  // Right way to write a constructor -- copy the array  public ImmutableArrayHolder(int[] anArray) {  this.theArray = (int[]) anArray.clone();  }  // Wrong way to write a constructor -- copy the reference  // The caller could change the array after the call to the constructor  public ImmutableArrayHolder(int[] anArray) {  this.theArray = anArray;  }  // Right way to write an accessor -- don't expose the array reference  public int getArrayLength() { return theArray.length }  public int getArray(int n) { return theArray[n]; }  // Right way to write an accessor -- use clone()  public int[] getArray() { return (int[]) theArray.clone(); }  // Wrong way to write an accessor -- expose the array reference  // A caller could get the array reference and then change the contents  public int[] getArray() { return theArray }  } |

通过一些其它工作，可以编写使用一些非 final 字段的不变类（例如， String 的标准实现使用 hashCode 值的惰性计算），这样可能比严格的 final 类执行得更好。如果类表示抽象类型（如数字类型或颜色）的值，那么您还会想实现 hashCode()和 equals() 方法，这样对象将作为 HashMap 或 HashSet 中的一个键工作良好。要保持线程安全，不允许 this 引用从构造函数中转义是很重要的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02183/#ibm-pcon)

**偶尔更改的数据**

有些数据项在程序生命期中一直保持常量，而有些会频繁更改。常量数据显然符合不变性，而状态复杂且频繁更改的对象通常不适合用不变类来实现。那么有时会更改，但更改又不太频繁的数据呢？有什么方法能让 有时更改的数据获得不变性的便利和线程安全的长处呢？

util.concurrent 包中的 CopyOnWriteArrayList 类是如何既利用不变性的能力，又仍允许偶尔修改的一个良好示例。它最适合于支持事件监听程序的类（如用户界面组件）使用。虽然事件监听程序的列表可以更改，但通常它更改的频繁性要比事件的生成少得多。

除了在修改列表时， CopyOnWriteArrayList 并不变更基本数组，而是创建新数组且废弃旧数组之外，它的行为与 ArrayList类非常相似。这意味着当调用程序获得迭代器（迭代器在内部保存对基本数组的引用）时，迭代器引用的数组实际上是不变的，从而可以无需同步或冒并发修改的风险进行遍历。这消除了在遍历前克隆列表或在遍历期间对列表进行同步的需要，这两个操作都很麻烦、易于出错，而且完全使性能恶化。如果遍历比插入或除去更加频繁（这在某些情况下是常有的事）， CopyOnWriteArrayList 会提供更佳的性能和更方便的访问。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp02183/#ibm-pcon)

**结束语**

使用不变对象比使用可变对象要容易得多。它们只能处于一种状态，所以始终是一致的，它们本来就是线程安全的，可以被自由地共享。使用不变对象可以彻底消除许多容易发生但难以检测的编程错误，如无法在线程间同步访问或在存储对数组或对象的引用前无法克隆该数组或对象。在编写类时，问问自己这个类是否可以作为不变类有效地实现，总是值得的。您可能会对回答常常是肯定的而感到吃惊。