**简介：** 很多算法用尾递归方法表示会显得格外简明。编译器会自动把这种方法转换成循环，以提高程序的性能。但在 Java 语言规范中，并没有要求一定要作这种转换，因此，并不是所有的 Java 虚拟机（JVM）都会做这种转换。这就意味着在 Java 语言中采用尾递归方法将导致巨大的内存占用，而这并不是我们期望的结果。Eric Allen 在本文中阐述了动态编译将会保持语言的语义，而静态编译则通常不会。他说明了为什么这是一个重要问题，并提供了一段代码来帮助判断您的即时（JIT）编译器是否会在保持语言语义的同时做尾递归代码转换。

**尾递归及其转换**

相当多的程序包含有循环，这些循环运行的时间占了程序总运行时间的很大一部分。这些循环经常要反复更新不止一个变量，而每个变量的更新又经常依赖于其它变量的值。

由于这一特点，在尾递归函数和循环之间有一个很好的对应关系：可以简单地把每个递归调用看作是一个循环的多次迭代。但因为所有可变的参数值都一次传给了递归调用，所以比起循环来，在尾递归中可以更容易地得到更新值。而且，难以使用的 break 语句也常常为函数的简单返回所替代。如果把迭代看成是 *尾递归函数*，那么，就可以把这些变量看成是函数的参数。简单提醒一下：如果一个调用的返回值被作为调用函数的值立即返回，那么，这个递归调用就是尾递归；尾递归不必记住调用时调用函数的上下文。

但在 Java 编程中，用这种方式表示迭代将导致效率低下，因为大量的递归调用有导致堆栈溢出的危险。

解决方案比较简单：因为尾递归函数实际上只是编写循环的一种更简单的方式，所以就让编译器把它们自动转换成循环形式。这样您就同时利用了这两种形式的优点。

但是，尽管大家都熟知如何把一个尾递归函数自动转换成一个简单循环，Java 规范却不要求做这种转换。不作这种要求的原因大概是：通常在面向对象的语言中，这种转换不能静态地进行。相反地，这种从尾递归函数到简单循环的转换必须由 JIT 编译器动态地进行。

要理解为什么会是这样，考虑下面一个失败的尝试：在 Integers 集上，把 Iterator 中的元素相乘。

因为下面的程序中有一个错误，所以在运行时会抛出一个异常。但是，就象在本专栏以前的许多文章中已经论证的那样，一个程序抛出的精确异常（跟很棒的错误类型标识符一样）对于找到错误藏在程序的什么地方并没有什么帮助，我们也不想编译器以这种方式改变程序，以使编译的结果代码抛出一个不同的异常。

**清单 1. 一个把 Integer 集的 Iterator 中的元素相乘的失败尝试**

|  |
| --- |
| import java.util.Iterator;  public class Example {  public int product(Iterator i) {  return productHelp(i, 0);  }  int productHelp(Iterator i, int accumulator) {  if (i.hasNext()) {  return productHelp(i, accumulator \* ((Integer)i.next()).intValue());  }  else {  return accumulator;  }  }  } |

注意 product 方法中的错误。 product 方法通过把 accumulator 赋值为 0 调用 productHelp 。它的值应为 1。否则，在类 Example 的任何实例上调用 product 都将产生 0 值，不管 Iterator 是什么值。

假设这个错误终于被改正了，但同时，类 Example 的一个子类也被创建了，如清单 2 所示：

**清单 2. 试图捕捉象清单 1 这样的不正确的调用**

|  |
| --- |
| import java.util.\*;  class Example {  public int product(Iterator i) {  return productHelp(i, 1);  }  int productHelp(Iterator i, int accumulator) {  if (i.hasNext()) {  return productHelp(i, accumulator \* ((Integer)i.next()).intValue());  }  else {  return accumulator;  }  }  }  // And, in a separate file:  import java.util.\*;  public class Example2 extends Example {  int productHelp(Iterator i, int accumulator) {  if (accumulator < 1) {  throw new RuntimeException("accumulator to productHelp must be >= 1");  }  else {  return super.productHelp(i, accumulator);  }  }  public static void main(String[] args) {  LinkedList l = new LinkedList();  l.add(new Integer(0));  new Example2().product(l.listIterator());  }  } |

类 Example2 中的被覆盖的 productHelp 方法试图通过当 accumulator 小于“1”时抛出运行时异常来捕捉对 productHelp 的不正确调用。不幸的是，这样做将引入一个新的错误。如果 Iterator 含有任何 0 值的实例，都将使 productHelp 在自身的递归调用上崩溃。

现在请注意，在类 Example2 的 main 方法中，创建了 Example2 的一个实例并调用了它的 product 方法。由于传给这个方法的Iterator 包含一个 0，因此程序将崩溃。

然而，您可以看到类 Example 的 productHelp 是严格尾递归的。假设一个静态编译器想把这个方法的正文转换成一个循环，如清单 3 所示：

**清单 3. 静态编译不会优化尾调用的一个示例**

|  |
| --- |
| int productHelp(Iterator i, int accumulator) {  while (i.hasNext()) {  accumulator \*= ((Integer)i.next()).intValue();  }  return accumulator;  } |

于是，最初对 productHelp 的调用，结果成了对超类的方法的调用。超方法将通过简单地在 iterator 上循环来计算其结果。不会抛出任何异常。

用两个不同的静态编译器来编译这段代码，结果是一个会抛出异常，而另一个则不会，想想这是多么让人感到困惑。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-diag/part8/index.html#ibm-pcon)

**您的 JIT 会做这种转换吗？**

因此，如清单 3 中的示例所示，我们不能期望静态编译器会在保持语言语义的同时对 Java 代码执行尾递归转换。相反地，我们必须依靠 JIT 进行的动态编译。JIT 会不会做这种转换是取决于 JVM。

要判断您的 JIT 会否转换尾递归的一个办法是编译并运行如下小测试类：

**清单 4. 判断您的 JIT 能否转换尾递归**

|  |
| --- |
| public class TailRecursionTest {  private static int loop(int i) {  return loop(i);  }  public static void main(String[] args) {  loop(0);  }  } |

我们来考虑一下这个类的 loop 方法。这个方法只是尽可能长时间地对自身作递归调用。因为它永远不会返回，也不会以任何方式影响任何外部变量，因此如清单 5 所示替换其代码正文将保留程序的语义。

**清单 5. 一个动态转换**

|  |
| --- |
| public class TailRecursionTest {  private static int loop(int i) {  while (true) {  }  }  public static void main(String[] args) {  loop(0);  }  } |

而且，事实上这也就是足够完善的编译器所做的转换。

如果您的 JIT 编译器把尾递归调用转换成迭代，这个程序将无限期地运行下去。它所需的内存很小，而且不会随时间增加。

另一方面，如果 JIT 不做这种转换，程序将会很快耗尽堆栈空间并报告一个堆栈溢出错误。

我在两个 Java SDK 上运行这个程序，结果令人惊讶。在 SUN 公司的 Hotspot JVM（版本 1.3 ）上运行时，发现 Hotspot 不执行这种转换。缺省设置下，在我的机器上运行时，不到一秒钟堆栈空间就被耗尽了。

另一方面，程序在 IBM 的 JVM（版本 1.3 ）上咕噜噜运行时却没有任何问题，这表明 IBM 的 JVM 以这种方式转换代码。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-diag/part8/index.html#ibm-pcon)

**总结**

记住：我们不能寄希望于我们的代码会总是运行在会转换尾递归调用的 JVM 上。因此，为了保证您的程序在所有 JVM 上都有适当的性能，您应始终努力把那些最自然地符合尾递归模式的代码按迭代风格编写。

但是请注意：就象我们的示例所演示的那样，以这种方式转换代码时很容易引入错误，不论是由人工还是由软件来完成这种转换。