双重检查锁定及单例模式

本文转载至 http://www.ibm.com/developerworks/cn/java/j-dcl.html

单例创建模式是一个通用的编程习语。和多线程一起使用时，必需使用某种类型的同步。在努力创建更有效的代码时，Java 程序员们创建了双重检查锁定习语，将其和单例创建模式一起使用，从而限制同步代码量。然而，由于一些不太常见的 Java 内存模型细节的原因，并不能保证这个双重检查锁定习语有效。它偶尔会失败，而不是总失败。此外，它失败的原因并不明显，还包含 Java 内存模型的一些隐秘细节。这些事实将导致代码失败，原因是双重检查锁定难于跟踪。在本文余下的部分里，我们将详细介绍双重检查锁定习语，从而理解它在何处失效。

单例创建习语

要理解双重检查锁定习语是从哪里起源的，就必须理解通用单例创建习语，如清单 1 中的阐释：

清单 1. 单例创建习语

```
import java.util.*;
class Singleton
{
  private static Singleton instance;
  private Vector v;
  private boolean inUse;

  private Singleton()
  {
    v = new Vector();
    v.addElement(new Object());
    inUse = true;
  }

  public static Singleton getInstance()
  {
    if (instance == null)         //1
      instance = new Singleton();  //2
    return instance;              //3
  }
}
```

此类的设计确保只创建一个 Singleton 对象。构造函数被声明为 private，getInstance() 方法只创建一个对象。这个实现适合于单线程程序。然而，当引入多线程时，就必须通过同步来保护 getInstance() 方法。如果不保护 getInstance() 方法，则可能返回 Singleton

对象的两个不同的实例。假设两个线程并发调用 getInstance() 方法并且按以下顺序执行调用：

线程 1 调用 getInstance() 方法并决定 instance 在 //1 处为 null。
线程 1 进入 if 代码块，但在执行 //2 处的代码行时被线程 2 预占。
线程 2 调用 getInstance() 方法并在 //1 处决定 instance 为 null。
线程 2 进入 if 代码块并创建一个新的 Singleton 对象并在 //2 处将变量 instance
分配给这个新对象。
线程 2 在 //3 处返回 Singleton 对象引用。
线程 2 被线程 1 预占。
线程 1 在它停止的地方启动，并执行 //2 代码行，这导致创建另一个 Singleton 对象。
线程 1 在 //3 处返回这个对象。

结果是 getInstance() 方法创建了两个 Singleton 对象，而它本该只创建一个对象。通过同步 getInstance() 方法从而在同一时间只允许一个线程执行代码，这个问题得以改正，如清单 2 所示：

清单 2. 线程安全的 getInstance() 方法

```
public static synchronized Singleton getInstance()
{
  if (instance == null)         //1
    instance = new Singleton();  //2
  return instance;              //3
}
```

清单 2 中的代码针对多线程访问 getInstance() 方法运行得很好。然而，当分析这段代码时，您会意识到只有在第一次调用方法时才需要同步。由于只有第一次调用执行了 //2 处的代码，而只有此行代码需要同步，因此就无需对后续调用使用同步。所有其他调用用于决定 instance 是非 null 的，并将其返回。多线程能够安全并发地执行除第一次调用外的所有调用。尽管如此，由于该方法是 synchronized 的，需要为该方法的每一次调用付出同步的代价，即使只有第一次调用需要同步。

为使此方法更为有效，一个被称为双重检查锁定的习语就应运而生了。这个想法是为了避免对除第一次调用外的所有调用都实行同步的昂贵代价。同步的代价在不同的 JVM 间是不同的。在早期，代价相当高。随着更高级的 JVM 的出现，同步的代价降低了，但出入 synchronized 方法或块仍然有性能损失。不考虑 JVM 技术的进步，程序员们绝不想不必要地浪费处理时间。因为只有清单 2 中的 //2 行需要同步，我们可以只将其包装到一个同步块中，如清单 3 所示：

清单 3. getInstance() 方法

```
public static Singleton getInstance()
```

```
{
  if (instance == null)
  {
    synchronized(Singleton.class) {
      instance = new Singleton();
    }
  }
  return instance;
}
```

清单 3 中的代码展示了用多线程加以说明的和清单 1 相同的问题。当 instance 为 null 时，两个线程可以并发地进入 if 语句内部。然后，一个线程进入 synchronized 块来初始化 instance，而另一个线程则被阻断。当第一个线程退出 synchronized 块时，等待着的线程进入并创建另一个 Singleton 对象。注意：当第二个线程进入 synchronized 块时，它并没有检查 instance 是否非 null。

双重检查锁定

为处理清单 3 中的问题，我们需要对 instance 进行第二次检查。这就是“双重检查锁定”名称的由来。将双重检查锁定习语应用到清单 3 的结果就是清单 4 。

清单 4. 双重检查锁定示例

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {  //1
      if (instance == null)          //2
        instance = new Singleton();  //3
    }
  }
  return instance;
}
```

双重检查锁定背后的理论是：在 //2 处的第二次检查使（如清单 3 中那样）创建两个不同的 Singleton 对象成为不可能。假设有下列事件序列：

线程 1 进入 getInstance() 方法。
由于 instance 为 null，线程 1 在 //1 处进入 synchronized 块。
线程 1 被线程 2 预占。

线程 2 进入 getInstance() 方法。
由于 instance 仍旧为 null，线程 2 试图获取 //1 处的锁。然而，由于线程 1 持有该锁，线程 2 在 //1 处阻塞。
线程 2 被线程 1 预占。
线程 1 执行，由于在 //2 处实例仍旧为 null，线程 1 还创建一个 Singleton 对象并将其引用赋值给 instance。
线程 1 退出 synchronized 块并从 getInstance() 方法返回实例。
线程 1 被线程 2 预占。
线程 2 获取 //1 处的锁并检查 instance 是否为 null。

由于 instance 是非 null 的，并没有创建第二个 Singleton 对象，由线程 1 创建的对象被返回。

双重检查锁定背后的理论是完美的。不幸地是，现实完全不同。双重检查锁定的问题是：并不能保证它会在单处理器或多处理器计算机上顺利运行。

双重检查锁定失败的问题并不归咎于 JVM 中的实现 bug，而是归咎于 Java 平台内存模型。内存模型允许所谓的“无序写入”，这也是这些习语失败的一个主要原因。

无序写入

为解释该问题，需要重新考察上述清单 4 中的 //3 行。此行代码创建了一个 Singleton 对象并初始化变量 instance 来引用此对象。这行代码的问题是：在 Singleton 构造函数体执行之前，变量 instance 可能成为非 null 的。

什么？这一说法可能让您始料未及，但事实确实如此。在解释这个现象如何发生前，请先暂时接受这一事实，我们先来考察一下双重检查锁定是如何被破坏的。假设清单 4 中代码执行以下事件序列：

线程 1 进入 getInstance() 方法。
由于 instance 为 null，线程 1 在 //1 处进入 synchronized 块。
线程 1 前进到 //3 处，但在构造函数执行之前，使实例成为非 null。
线程 1 被线程 2 预占。
线程 2 检查实例是否为 null。因为实例不为 null，线程 2 将 instance 引用返回给一个构造完整但部分初始化了的 Singleton 对象。
线程 2 被线程 1 预占。
线程 1 通过运行 Singleton 对象的构造函数并将引用返回给它，来完成对该对象的初始化。

此事件序列发生在线程 2 返回一个尚未执行构造函数的对象的时候。为展示此事件的发生情况，假设为代码行 instance =new Singleton(); 执行了下列伪代码：

```
instance =new Singleton();
mem = allocate();           //Allocate memory for Singleton object.
instance = mem;             //Note that instance is now non-null, but
                            //has not been initialized.
ctorSingleton(instance);    //Invoke constructor for Singleton passing
                            //instance.
```

这段伪代码不仅是可能的，而且是一些 JIT 编译器上真实发生的。执行的顺序是颠倒的，但鉴于当前的内存模型，这也是允许发生的。JIT 编译器的这一行为使双重检查锁定的问题只不过是一次学术实践而已。

为说明这一情况，假设有清单 5 中的代码。它包含一个剥离版的 getInstance() 方法。我已经删除了“双重检查性”以简化我们对生成的汇编代码（清单 6）的回顾。我们只关心 JIT 编译器如何编译 instance=new Singleton(); 代码。此外，我提供了一个简单的构造函数来明确说明汇编代码中该构造函数的运行情况。

清单 5. 用于演示无序写入的单例类

```
class Singleton
{
  private static Singleton instance;
  private boolean inUse;
  private int val;

  private Singleton()
  {
    inUse = true;
    val = 5;
  }
  public static Singleton getInstance()
  {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
}
```

清单 6 包含由 Sun JDK 1.2.1 JIT 编译器为清单 5 中的 getInstance() 方法体生成的汇编代码。

清单 6. 由清单 5 中的代码生成的汇编代码

```
;asm code generated for getInstance
054D20B0   mov        eax,[049388C8]      ;load instance ref
054D20B5   test       eax,eax             ;test for null
054D20B7   jne        054D20D7
054D20B9   mov        eax,14C0988h
054D20BE   call       503EF8F0            ;allocate memory
```

```
054D20C3   mov       [049388C8],eax      ;store pointer in
                        ;instance ref. instance
                        ;non-null and ctor
                        ;has not run
054D20C8   mov       ecx,dword ptr [eax]
054D20CA   mov       dword ptr [ecx],1   ;inline ctor - inUse=true;
054D20D0   mov       dword ptr [ecx+4],5 ;inline ctor - val=5;
054D20D7   mov       ebx,dword ptr ds:[49388C8h]
054D20DD   jmp       054D20B0
```

注: 为引用下列说明中的汇编代码行，我将引用指令地址的最后两个值，因为它们都以 054D20 开头。例如，B5 代表 test eax,eax。

汇编代码是通过运行一个在无限循环中调用 getInstance() 方法的测试程序来生成的。程序运行时，请运行 Microsoft Visual C++ 调试器并将其附到表示测试程序的 Java 进程中。然后，中断执行并找到表示该无限循环的汇编代码。

B0 和 B5 处的前两行汇编代码将 instance 引用从内存位置 049388C8 加载至 eax 中，并进行 null 检查。这跟清单 5 中的 getInstance() 方法的第一行代码相对应。第一次调用此方法时，instance 为 null，代码执行到 B9。BE 处的代码为 Singleton 对象从堆中分配内存，并将一个指向该块内存的指针存储到 eax 中。下一行代码，C3，获取 eax 中的指针并将其存储回内存位置为 049388C8 的实例引用。结果是，instance 现在为非 null 并引用一个有效的 Singleton 对象。然而，此对象的构造函数尚未运行，这恰是破坏双重检查锁定的情况。然后，在 C8 行处，instance 指针被解除引用并存储到 ecx。CA 和 D0 行表示内联的构造函数，该构造函数将值 true 和 5 存储到 Singleton 对象。如果此代码在执行 C3 行后且在完成该构造函数前被另一个线程中断，则双重检查锁定就会失败。

不是所有的 JIT 编译器都生成如上代码。一些生成了代码，从而只在构造函数执行后使 instance 成为非 null。针对 Java 技术的 IBM SDK 1.3 版和 Sun JDK 1.3 都生成这样的代码。然而，这并不意味着应该在这些实例中使用双重检查锁定。该习语失败还有一些其他原因。此外，您并不总能知道代码会在哪些 JVM 上运行，而 JIT 编译器总是会发生变化，从而生成破坏此习语的代码。

双重检查锁定：获取两个

考虑到当前的双重检查锁定不起作用，我加入了另一个版本的代码，如清单 7 所示，从而防止您刚才看到的无序写入问题。

清单 7. 解决无序写入问题的尝试

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {      //1
      Singleton inst = instance;         //2
```

```
    if (inst == null)
    {
      synchronized(Singleton.class) {  //3
        inst = new Singleton();        //4
      }
      instance = inst;                 //5
    }
   }
  }
  return instance;
}
```

看着清单 7 中的代码，您应该意识到事情变得有点荒谬。请记住，创建双重检查锁定是为了避免对简单的三行 getInstance() 方法实现同步。清单 7 中的代码变得难于控制。另外，该代码没有解决问题。仔细检查可获悉原因。

此代码试图避免无序写入问题。它试图通过引入局部变量 inst 和第二个 synchronized 块来解决这一问题。该理论实现如下：

线程 1 进入 getInstance() 方法。
由于 instance 为 null，线程 1 在 //1 处进入第一个 synchronized 块。
局部变量 inst 获取 instance 的值，该值在 //2 处为 null。
由于 inst 为 null，线程 1 在 //3 处进入第二个 synchronized 块。
线程 1 然后开始执行 //4 处的代码，同时使 inst 为非 null，但在 Singleton 的构造函数执行前。（这就是我们刚才看到的无序写入问题。）
线程 1 被线程 2 预占。
线程 2 进入 getInstance() 方法。
由于 instance 为 null，线程 2 试图在 //1 处进入第一个 synchronized 块。由于线程 1 目前持有此锁，线程 2 被阻断。
线程 1 然后完成 //4 处的执行。
线程 1 然后将一个构造完整的 Singleton 对象在 //5 处赋值给变量 instance，并退出这两个 synchronized 块。
线程 1 返回 instance。
然后执行线程 2 并在 //2 处将 instance 赋值给 inst。
线程 2 发现 instance 为非 null，将其返回。

这里的关键行是 //5。此行应该确保 instance 只为 null 或引用一个构造完整的 Singleton 对象。该问题发生在理论和实际彼此背道而驰的情况下。由于当前内存模型的定义，清单 7 中的代码无效。Java 语言规范（Java Language Specification，JLS）要求不能将 synchronized 块中的代码移出来。但是，并没有说不能将 synchronized 块外面的代码移入 synchronized 块中。

JIT 编译器会在这里看到一个优化的机会。此优化会删除 //4 和 //5 处的代码，组合并且生成清单 8 中所示的代码。

清单 8. 从清单 7 中优化来的代码。

```
public static Singleton getInstance()
{
  if (instance == null)
  {
    synchronized(Singleton.class) {      //1
      Singleton inst = instance;         //2
      if (inst == null)
      {
        synchronized(Singleton.class) {  //3
          //inst = new Singleton();      //4
          instance = new Singleton();
        }
        //instance = inst;               //5
      }
    }
  }
  return instance;
}
```

如果进行此项优化，您将同样遇到我们之前讨论过的无序写入问题。
用 volatile 声明每一个变量怎么样？

另一个想法是针对变量 inst 以及 instance 使用关键字 volatile。根据 JLS（参见
参考资料），声明成 volatile 的变量被认为是顺序一致的，即，不是重新排序的。但是试图使用
volatile 来修正双重检查锁定的问题，会产生以下两个问题：
这里的问题不是有关顺序一致性的，而是代码被移动了，不是重新排序。

即使考虑了顺序一致性，大多数的 JVM 也没有正确地实现
volatile。第二点值得展开讨论。假设有清单 9 中的代码：

清单 9. 使用了 volatile 的顺序一致性

```
class test
{
  private volatile boolean stop = false;
  private volatile int num = 0;

  public void foo()
  {
    num = 100;    //This can happen second
    stop = true;  //This can happen first
    //...
```

```
  }

  public void bar()
  {
    if (stop)
      num += num;  //num can == 0!
  }
  //...
}
```

根据 JLS，由于 stop 和 num 被声明为 volatile，它们应该顺序一致。这意味着如果 stop 曾经是 true，num 一定曾被设置成 100。尽管如此，因为许多 JVM 没有实现 volatile 的顺序一致性功能，您就不能依赖此行为。因此，如果线程 1 调用 foo 并且线程 2 并发地调用 bar，则线程 1 可能在 num 被设置成为 100 之前将 stop 设置成 true。这将导致线程见到 stop 是 true，而 num 仍被设置成 0。使用 volatile 和 64 位变量的原子数还有另外一些问题，但这已超出了本文的讨论范围。有关此主题的更多信息，请参阅 参考资料。

解决方案

底线就是：无论以何种形式，都不应使用双重检查锁定，因为您不能保证它在任何 JVM 实现上都能顺利运行。JSR-133 是有关内存模型寻址问题的，尽管如此，新的内存模型也不会支持双重检查锁定。因此，您有两种选择：
接受如清单 2 中所示的 getInstance() 方法的同步。放弃同步，而使用一个 static 字段。选择项 2 如清单 10 中所示

清单 10. 使用 static 字段的单例实现

```
class Singleton
{
  private Vector v;
  private boolean inUse;
  private static Singleton instance = new Singleton();

  private Singleton()
  {
    v = new Vector();
    inUse = true;
    //...
  }

  public static Singleton getInstance()
  {
    return instance;
  }
```

```
}
```

清单 10 的代码没有使用同步，并且确保调用 static getInstance() 方法时才创建 Singleton。如果您的目标是消除同步，则这将是一个很好的选择。

String 不是不变的

鉴于无序写入和引用在构造函数执行前变成非 null 的问题，您可能会考虑 String 类。假设有下列代码：

```
private String str;
//...
str = new String("hello");
```

String 类应该是不变的。尽管如此，鉴于我们之前讨论的无序写入问题，那会在这里导致问题吗？答案是肯定的。考虑两个线程访问 String str。一个线程能看见 str 引用一个 String 对象，在该对象中构造函数尚未运行。事实上，清单 11 包含展示这种情况发生的代码。注意，这个代码仅在我测试用的旧版 JVM 上会失败。IBM 1.3 和 Sun 1.3 JVM 都会如期生成不变的 String。

清单 11. 可变 String 的例子

```
class StringCreator extends Thread
{
  MutableString ms;
  public StringCreator(MutableString muts)
  {
    ms = muts;
  }
  public void run()
  {
    while(true)
      ms.str = new String("hello");         //1
  }
}
class StringReader extends Thread
{
  MutableString ms;
  public StringReader(MutableString muts)
  {
    ms = muts;
```

```
  }
  public void run()
  {
    while(true)
    {
      if (!(ms.str.equals("hello")))        //2
      {
        System.out.println("String is not immutable!");
        break;
      }
    }
  }
}
class MutableString
{
  public String str;                        //3
  public static void main(String args[])
  {
    MutableString ms = new MutableString();  //4
    new StringCreator(ms).start();           //5
    new StringReader(ms).start();            //6
  }
}
```

此代码在 //4 处创建一个 MutableString 类，它包含了一个 String 引用，此引用由 //3 处的两个线程共享。在行 //5 和 //6 处，在两个分开的线程上创建了两个对象 StringCreator 和 StringReader。传入一个 MutableString 对象的引用。StringCreator 类进入到一个无限循环中并且使用值“hello”在 //1 处创建 String 对象。StringReader 也进入到一个无限循环中，并且在 //2 处检查当前的 String 对象的值是不是“hello”。如果不行，StringReader 线程打印出一条消息并停止。如果 String 类是不变的，则从此程序应当看不到任何输出。如果发生了无序写入问题，则使 StringReader 看到 str 引用的惟一方法绝不是值为“hello”的 String 对象。
在旧版的 JVM 如 Sun JDK 1.2.1 上运行此代码会导致无序写入问题。并因此导致一个非不变的 String。

结束语

为避免单例中代价高昂的同步，程序员非常聪明地发明了双重检查锁定习语。不幸的是，鉴于当前的内存模型的原因，该习语尚未得到广泛使用，就明显成为了一种不安全的编程结构。重定义脆弱的内存模型这一领域的工作正在进行中。尽管如此，即使是在新提议的内存模型中，双重检查锁定也是无效的。对此问题最佳的解决方案是接受同步或者使用一个 static field。

本文链接: 【】（）