在 Java 实现正确的双重检查锁（Double-Checked Locking）

双重检查锁定模式（也被称为"双重检查加锁优化"，"锁暗示"（Lock hint））是一种软件设计模式用来减少并发系统中竞争和同步的开销。双重检查锁定模式首先验证锁定条件（第一次检查），只有通过锁定条件验证才真正的进行加锁逻辑并再次验证条件（第二次检查）。

该模式在某些语言在某些硬件平台的实现可能是不安全的。有的时候，这一模式被看做是反模式。

它通常用于减少加锁开销，尤其是为多线程环境中的单例模式实现“惰性初始化”。惰性初始化的意思是直到第一次访问时才初始化它的值。本文将介绍双重检查锁在 Java 中如何实现。

在实现单例模式时，如果未考虑多线程的情况，就容易写出下面的错误代码：

```
public class Singleton {
   private static Singleton uniqueSingleton;

   private Singleton() {
   }

   public Singleton getInstance() {
      if (null == uniqueSingleton) {
         uniqueSingleton = new Singleton();
      }
      return uniqueSingleton;
   }
}
```

在多线程的情况下，这样写可能会导致uniqueSingleton有多个实例。比如下面这种情况，考虑有两个线程同时调用getInstance()：

Time	Thread A	Thread B
T1	检查到uniqueSingleton为空	
T2		检查到uniqueSingleton为空
T3		初始化对象A
T4		返回对象A
T5	初始化对象B	
T6	返回对象B	

可以看到，uniqueSingleton被实例化了两次并且被不同对象持有。完全违背了单例的初衷。

加锁

出现这种情况，第一反应就是加锁，如下：

```
public class Singleton {
   private static Singleton uniqueSingleton;

   private Singleton() {
   }

   public synchronized Singleton getInstance() {
      if (null == uniqueSingleton) {
         uniqueSingleton = new Singleton();
      }
      return uniqueSingleton;
   }
}
```

这样虽然解决了问题，但是因为用到了synchronized，会导致很大的性能开销，并且加锁其实只需要在第一次初始化的时候用到，之后的调用都没必要再进行加锁。

双重检查锁

双重检查锁（double checked
locking）是对上述问题的一种优化。先判断对象是否已经被初始化，再决定要不要加锁。

错误的双重检查锁

```
public class Singleton {
   private static Singleton uniqueSingleton;

   private Singleton() {
   }

   public Singleton getInstance() {
      if (null == uniqueSingleton) {
         synchronized (Singleton.class) {
            if (null == uniqueSingleton) {
               uniqueSingleton = new Singleton();   // error
```

```
            }
        }
    }
    return uniqueSingleton;
  }
}
```

如果这样写，运行顺序就成了：

1. 检查变量是否被初始化(不去获得锁)，如果已被初始化则立即返回。
2. 获取锁。
3. 再次检查变量是否已经被初始化，如果还没被初始化就初始化一个对象。

执行双重检查是因为，如果多个线程同时了通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么剩余通过了第一次检查的线程就不会再去实例化对象。

这样，除了初始化的时候会出现加锁的情况，后续的所有调用都会避免加锁而直接返回，解决了性能消耗的问题。

隐患

上述写法看似解决了问题，但是有个很大的隐患。实例化对象的那行代码（标记为error的那行），实际上可以分解成以下三个步骤：

1. 分配内存空间
2. 初始化对象
3. 将对象指向刚分配的内存空间

但是有些编译器为了性能的原因，可能会将第二步和第三步进行重排序，顺序就成了：

1. 分配内存空间
2. 将对象指向刚分配的内存空间
3. 初始化对象

现在考虑重排序后，两个线程发生了以下调用：

Time	Thread A	Thread B
T1	检查到uniqueSingleton为空	
T2	获取锁	
T3	再次检查到uniqueSingleton为空	
T4	为uniqueSingleton分配内存空间	

Time	Thread A	Thread B
T5	将uniqueSingleton指向内存空间	
T6		检查到uniqueSingleton不为空
T7		访问uniqueSingleton（此时对象还未完成初始化）
T8	初始化uniqueSingleton	

在这种情况下，T7时刻线程B对uniqueSingleton的访问，访问的是一个初始化未完成的对象。

正确的双重检查锁

```
public class Singleton {
   private volatile static Singleton uniqueSingleton;

   private Singleton() {
   }

   public Singleton getInstance() {
      if (null == uniqueSingleton) {
         synchronized (Singleton.class) {
            if (null == uniqueSingleton) {
               uniqueSingleton = new Singleton();
            }
         }
      }
      return uniqueSingleton;
   }
}
```

为了解决上述问题，需要在uniqueSingleton前加入关键字volatile。使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

至此，双重检查锁就可以完美工作了。

参考资料：

1. 双重检查锁定模式
2. 如何在Java中使用双重检查锁实现单例
3. 双重检查锁定与延迟初始化

本文链接: 【】（）