链滴

为什么双重检查锁模式需要 volatile？

作者：9526xu
原文链接：https://ld246.com/article/1565836825381
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

双重检查锁定（Double check locked）模式经常会出现在一些框架源码中，目的是为了延迟初始化量。这个模式还可以用来创建单例。下面来看一个 Spring 中双重检查锁定的例子。

```java
private volatile Map<String, Object> handlerMappings;

/**
 * Load the specified NamespaceHandler mappings lazily.
 */
private Map<String, Object> getHandlerMappings() {
    Map<String, Object> handlerMappings = this.handlerMappings;
    // 第一次检查
    if (handlerMappings == null) {
        synchronized (this) {
            handlerMappings = this.handlerMappings;
            // 第二次检查
            if (handlerMappings == null) {
                // 正式开始加载资源
                if (logger.isTraceEnabled()) {
                    logger.trace("Loading NamespaceHandler mappings from [" + this.handlerMappingsLocation + "]");
                }
                try {
                    Properties mappings =
                            PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation, this.classLoader);
                    if (logger.isTraceEnabled()) {
                        logger.trace("Loaded NamespaceHandler mappings: " + mappings);
                    }
                    handlerMappings = new ConcurrentHashMap<>(mappings.size());
                    CollectionUtils.mergePropertiesIntoMap(mappings, handlerMappings);
                    this.handlerMappings = handlerMappings;
                }
                catch (IOException ex) {
                    throw new IllegalStateException(
                            "Unable to load NamespaceHandler mappings from location [" + this.handlerMappingsLocation +
"]", ex);
                }
            }
        }
    }
    return handlerMappings;
}
```

这个例子中需要将配置文件加载到 handlerMappings中，由于读取资源比较耗时，所以将动作放到正需要 handlerMappings 的时候。我们可以看到 handlerMappings 前面使用了volatile 。有没有过为什么一定需要 volatile？虽然之前了解了双重检查锁定模式的原理，但是却忽略变量使用了 volatile。

下面我们就来看下这背后的原因。

错误的延迟初始化例子

想到延迟初始化一个变量，最简单的例子就是取出变量进行判断。

原文链接：为什么双重检查锁模式需要 volatile ？

```java
private  Cache cache;

public Cache getCache() {
    if (cache == null) {
        cache=new Cache();
    }
    return cache;
}
```

这个例子在单线程环境交易正常运行，但是在多线程环境就有可能会抛出空指针异常。为了防止这种况，我们需要使用 synchronized 。这样该方法在多线程环境就是安全的，但是这么做就会导致每次用该方法获取与释放锁，开销很大。

深入分析可以得知只有在初始化的变量的需要真正加锁，一旦初始化之后，直接返回对象即可。

所以我们可以将该方法改造以下的样子。

```java
private Cache cache;

public Cache getCache() {
    if (cache == null) {
        synchronized (this) {
            if (cache == null) {
                cache = new Cache();
            }
        }

    }
    return cache;
}
```

这个方法首先判断变量是否被初始化，没有被初始化，再去获取锁。获取锁之后，再次判断变量是否初始化。第二次判断目的在于有可能其他线程获取过锁，已经初始化改变量。第二次检查还未通过，会真正初始化变量。

这个方法检查判定两次，并使用锁，所以形象称为双重检查锁定模式。

这个方案缩小锁的范围，减少锁的开销，看起来很完美。然而这个方案有一些问题却很容易被忽略。

new 实例背后的指令

这个被忽略的问题在于 Cache cache=new Cache() 这行代码并不是一个原子指令。使用 javap -c 令，可以快速查看字节码。

```
// 创建 Cache 对象实例，分配内存
  0: new          #5                  // class com/query/Cache
  // 复制栈顶地址，并再将其压入栈顶
  3: dup
// 调用构造器方法，初始化 Cache 对象
  4: invokespecial #6                 // Method "<init>":()V
// 存入局部方法变量表
```

7: astore_1

从字节码可以看到创建一个对象实例，可以分为三步：

1. 分配对象内存
2. 调用构造器方法，执行初始化
3. 将对象引用赋值给变量。

虚拟机实际运行时，以上指令可能发生重排序。以上代码 2,3 可能发生重排序，但是并不会重排序 1 顺序。也就是说 1 这个指令都需要先执行，因为 2,3 指令需要依托 1 指令执行结果。

Java 语言规规定了线程执行程序时需要遵守 **intra-thread semantics**。**intra-thread semantics ** 保证重排序不会改变单线程内的程序执行结果。这个重排序在没有改变单线程程序的执行结果的前提，可以提高程序的执行性能。

虽然重排序并不影响单线程内的执行结果，但是在多线程的环境就带来一些问题。

	线程1	线程2
t1	分配内存	
t2	变量赋值	
t3		判断对象是否为 null
t4		由于对象不为 null，访问改对象
t5	初始化对象	

上面错误双重检查锁定的示例代码中，如果线程 1 获取到锁进入创建对象实例，这个时候发生了指令排序。当线程1 执行到 t3 时刻，线程 2 刚好进入，由于此时对象已经不为 Null，所以线程 2 可以自访问该对象。然后该对象还未初始化，所以线程 2 访问时将会发生异常。

volatile 作用

正确的双重检查锁定模式需要需要使用 volatile。volatile主要包含两个功能。

1. 保证可见性。使用 volatile 定义的变量，将会保证对所有线程的可见性。
2. 禁止指令重排序优化。

由于 volatile 禁止对象创建时指令之间重排序，所以其他线程不会访问到一个未初始化的对象，从而证安全性。

注意，volatile禁止指令重排序在 JDK 5 之后才被修复

使用局部变量优化性能

重新查看 Spring 中双重检查锁定代码。

```java
private volatile Map<String, Object> handlerMappings;

/**
 * Load the specified NamespaceHandler mappings lazily.
 */
private Map<String, Object> getHandlerMappings() {
    Map<String, Object> handlerMappings = this.handlerMappings;
    // 第一次检查
    if (handlerMappings == null) {
        synchronized (this) {
            handlerMappings = this.handlerMappings;
            // 第二次检查
            if (handlerMappings == null) {
                // 正在开始加载资源
                if (logger.isTraceEnabled()) {
                    logger.trace("Loading NamespaceHandler mappings from [" + this.handlerMappingsLocation + "]");
                }
                try {
                    Properties mappings =
                            PropertiesLoaderUtils.loadAllProperties(this.handlerMappingsLocation, this.classLoader);
                    if (logger.isTraceEnabled()) {
                        logger.trace("Loaded NamespaceHandler mappings: " + mappings);
                    }
                    handlerMappings = new ConcurrentHashMap<>(mappings.size());
                    CollectionUtils.mergePropertiesIntoMap(mappings, handlerMappings);
                    this.handlerMappings = handlerMappings;
                }
                catch (IOException ex) {
                    throw new IllegalStateException(
                            "Unable to load NamespaceHandler mappings from location [" + this.handlerMappingsLocation +
"]", ex);
                }
            }
        }
    }
    return handlerMappings;
}
```

可以看到方法内部使用局部变量，首先将实例变量值赋值给该局部变量，然后再进行判断。最后内容写入局部变量，然后再将局部变量赋值给实例变量。

使用局部变量相对于不使用局部变量，可以提高性能。主要是由于 volatile 变量创建对象时需要禁止令重排序，这就需要一些额外的操作。

总结

对象的创建可能发生指令的重排序，使用 volatile 可以禁止指令的重排序，保证多线程环境内的系统全。

帮助文档

双重检查锁定与延迟初始化

有关“双重检查锁定失效”的说明