java多线程之延迟初始化

阅读 79 收藏 2 2018-07-23 原文链接: click.aliyun.com

有时候我们可能推迟一些高开销的对象的初始化操作,并且只有在使用这些对象时才进行初始化,开发者可以采用延迟初始化来实现该需求。但是要正确实现线程安全的延迟初始化还是需要一些技巧的,否则很容易出现问题。下面是一个非线程安全的延迟初始化的例子:

在UnsafeLazyInit类中,假设A线程执行代码1的同时,B线程执行代码2,此时线程A可能会看到instance对象还没有完成初始化。

对于UnsafeLazyInit类,我们可以对getInstance方法做同步处理来实现线程安全的延迟初始化,代码如下:





由于对getInstance方法做了同步处理,将导致性能开销,如果getInstance方法被多个线程频繁调用的话,将会导致程序执行性能的下降,而如果getInstance不会被多个线程频繁调用,那么这个方案将会提供令人满意的性能。

对于synchronized方法可能带来的程序执行性能的下降,我们可以使用一种"聪明"的技巧:双重检查锁定(Double-Checked Locking)来降低同步的开销。下面是使用双重检查锁来实现延迟初始化的示例代码:

```
class DoubleCheckedLocking {

private static Instance instance;

public static Instance getInstance() {

    if(instance == null) {

        synchronized(DoubleCheckedLocking.class) {

        if(instance == null) {

            instance = new Instance();

        }

    }

    return instance;
}
```

按照上面的代码,如果第一次检查instance不为null,则不需要执行下面的加锁和二次检查与初始化操作,因此可以大大降低synchronized带来的性能开销,似乎是两全其美的实现方式。

双重检查锁定看起来似乎很完美,但这是一个错误的优化!在线程执行1:第一次检查时,代码读取到 instance不为null,其实instance有肯能还没有完成初始化,该问题的根源就在于:重排序。 在创建instance实例时,instance = new Instance()这行代码可以分解为如下3行伪代码:

```
memory = allocate(); //1: 分配对象的内存空间
ctorInstance(memory); //2: 初始化对象
instance = memory; //3: 设置instance指向刚分配的内存
```

上述伪代码中的2和3之间,可能会发生重排序,重排序后的执行顺序如下:

```
memory = allocate(); //1: 分配对象的内存空间 instance = memory; //3: 设置instance指向刚分配的内存 注意: 此时对象还没有被初始化! ctorInstance(memory); //2: 初始化对象
```





一次检查时instance不为null,线程B接下来将访问instance所引用的对象,但此时该对象可能还没有被A线程初始化,也就是会访问一个未被初始化的对象。

知道了这个问题根源以后,可以有两个办法来实现线程安全的延迟初始化:

- 1.不允许2和3重排序。
- 2.允许2和3重排序,但不允许其它线程"看到"这个重排序。

1.不允许2和3重排序,只需对双重检查锁定做小小的修改即可,我们把instance声明为volatile型,就可以实现线程安全的延迟初始化,示例代码如下:

当对象声明为volatile后,伪代码中的2和3的重排序,在多线程环境中将被禁止。

2.允许2和3重排序,但不允许其它线程"看到"这个重排序。

JVM在类的初始化阶段(即在Class被加载后,且被线程使用之前),会执行类的初始化。在执行类的初始化期间,JVM会去获取一个锁,这个锁可以同步多个线程对一个类的初始化。基于这个特性,我们可以在允许2和3重排序的情况下,实现线程安全的延迟初始化。

```
public class InstanceFactory {
    private static class InstanceHolder {
        public static Instance instance = new Instance();
    }
    public static Instance getInstance() {
        return InstanceHolder.instance: //这里将导致InstanceHolder类被初始化。
```





这个方案的实质是: 允许2和3重排序, 但是不允许非构造线程(如线程B) "看到"这个重排序。

在InstanceFactory中,首次执行getInstance方法的线程(如线程A)将导致InstanceHolder类被初始 化,但是如果多个线程同时调用getInstance方法,将会怎样呢?

Java语言规范规定,对于每一个类或接口C,都有一个唯一的初始化锁LC与之对应,从C到LC的映 射,由JVM的具体实现去自由实现。JVM在初始化期间会获取这个初始化锁,并且每个线程至少获取 一次锁来确保这个类被初始化了。

这个过程比较冗长,这里不做过多描述,总之就是JVM通过初始化锁同步了多个线程同时初始化一个 对象的操作,保证类不会被多次初始化。

通过对比基于volatile的双重检查锁定的方案和基于类初始化的方案,我们发现基于类初始化的方案更 加简洁。但基于volatile的双重检查锁定方案有一个额外优势:除了可以对静态字段实现延迟初始化 外,还可以对实例字段实现延迟初始化。

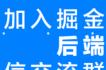
在设计模式中,有一个单例模式(Singleton),该模式比较常用,我们可以使用基于volatile的双重检 查锁定和基于类初始化的方案去创建单例对象,在实际工作中,我一般是使用基于类初始化的方案去 实现单例模式。

Java

JVM

键弧 属于你的 技术圈







・微信交流群

相关热门文章

【面经分享】互联网寒冬、7面阿里、终获Offer、定级P6+

敖丙 ♥ 73 ▶ 32





