单例模式的双重检查

尉昌达 +关注
2017.06.24 16:42* 字数 1329 阅读 543 评论 0 喜欢 0

你也可以写文赚赞赏
扫码下载简书 App
人工智能怎么入门？
零基础 | 4个月入门 | 个性化辅导
立抢席位
广告

1.双重检查锁定的由来

下面是非线程安全的延迟初始化对象的示例代码。

```java
public class UnsafeLazyInitialization {
    private static Instance instance;
    public static Instance getInstance(){
        if(instance ==null)                  //1:A线程执行
            instance = new Instance();       //2:B线程执行
        return instance;
    }
}
```

在UnsafeLazyInitialization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引用的对象还没有完成初始化（原因之后分析）
对于UnsafeLazyInitialization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。示例代码如下。

```java
public class safeLazyInitialization {
    private static Instance instance;
    public synchronized static Instance getInstance(){
        if(instance ==null)
            instance = new Instance();
        return instance;
    }
}
```

由于对getInstance()方法做了同步处理，synchronized将导致性能开销。如果getInstance（）被多个线程调用，将导致程序性能下降。反之，那么这个延迟初始化方案能提供令人满意的性能。

在早期的JVM中，synchronized（甚至是无竞争的synchronized）存在着巨大性能开销。因此，出现双重检查锁定。以下是示例代码。

```java
public class DoubleCheckedLocking {                    //1
    private static Instance instance;                  //2
    public  static Instance getInstance(){             //3
        if(instance ==null) {                          //4:第一次检查
            synchronized (DoubleCheckedLocking.class) { //5: 加锁
                if (instance == null)                  //6: 第二次检查
                    instance = new Instance();         //7: 问题的根源处在这里
            }                                          //8
        }                                              //9
        return instance;                               //10
    }                                                  //11
}
```

如上面的代码所示，如果第一次检查instance不为null，那就不需要执行下面的加锁和初始化操作。因此，可以大幅降低synchronized带来的性能开销。

这样似乎很完美，但这是一个错误的优化！在线程执行到第4行，代码读取到instance不为null时，instance引用的对象可能还没有完成初始化。

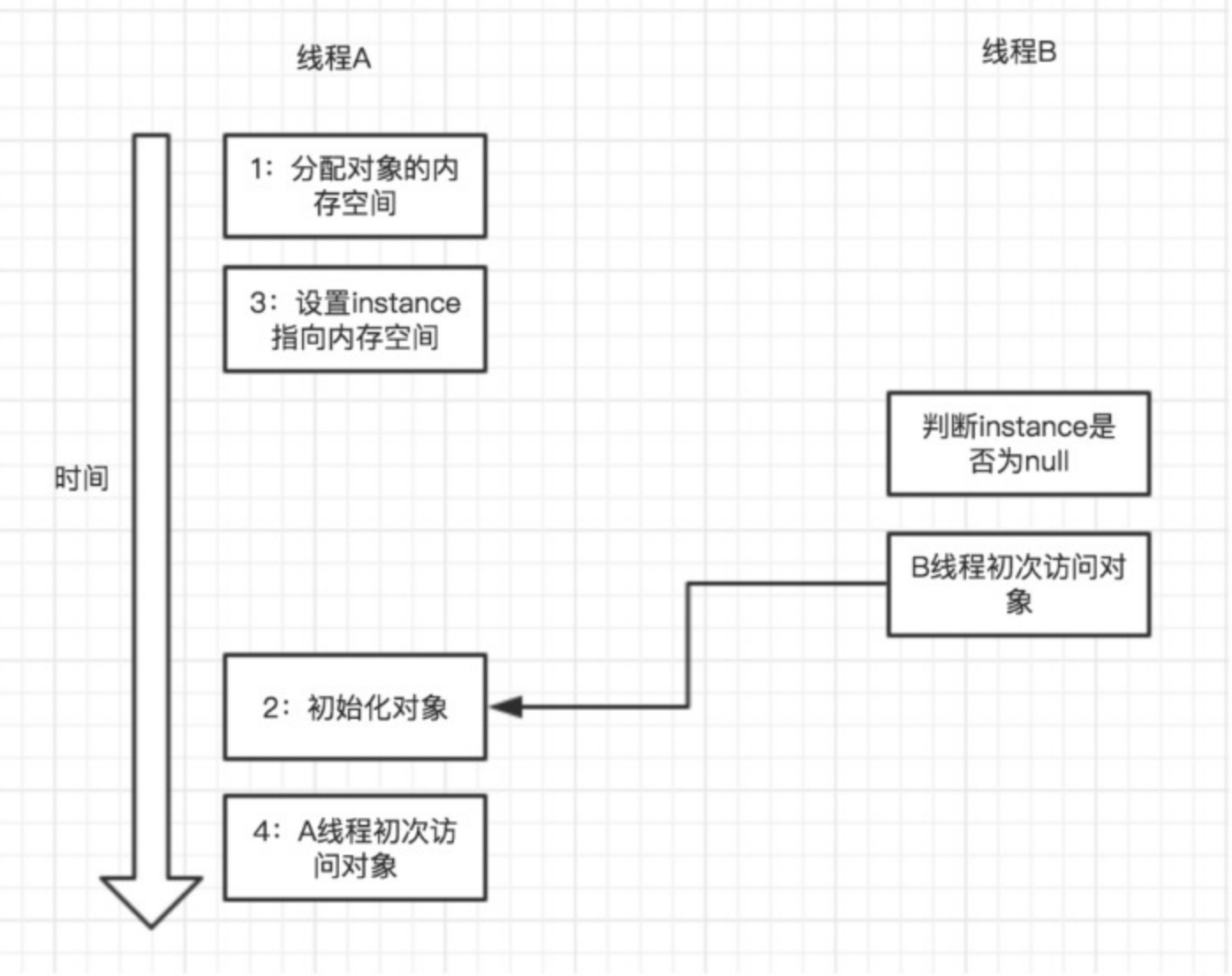
2.问题的根源

前面的双重检查示例代码第7行创建了一个对象。这一行代码可以分解为如下的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指向刚分配的内存地址
```

上面3行伪代码中的2和3之间，可能会被重排序。2和3重排序之后的执行时序如下。

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



多线程执行时序图

由于单线程内要遵守intra-thread semantics,从而能保证A线程的执行结果不会被改变。但是，当线程A和B按上图时序执行时，B线程将看到一个还没有被初始化的对象。

回到主题，DoubleCheckedLocking代码第7行（instance=new Instance()；）如果发生重排序，拎一个并发执行的线程B就有可能在第4行判断instance不为null。线程B接下来访问instance所引用的对象，但此时这个对象可能还没有被A线程初始化！

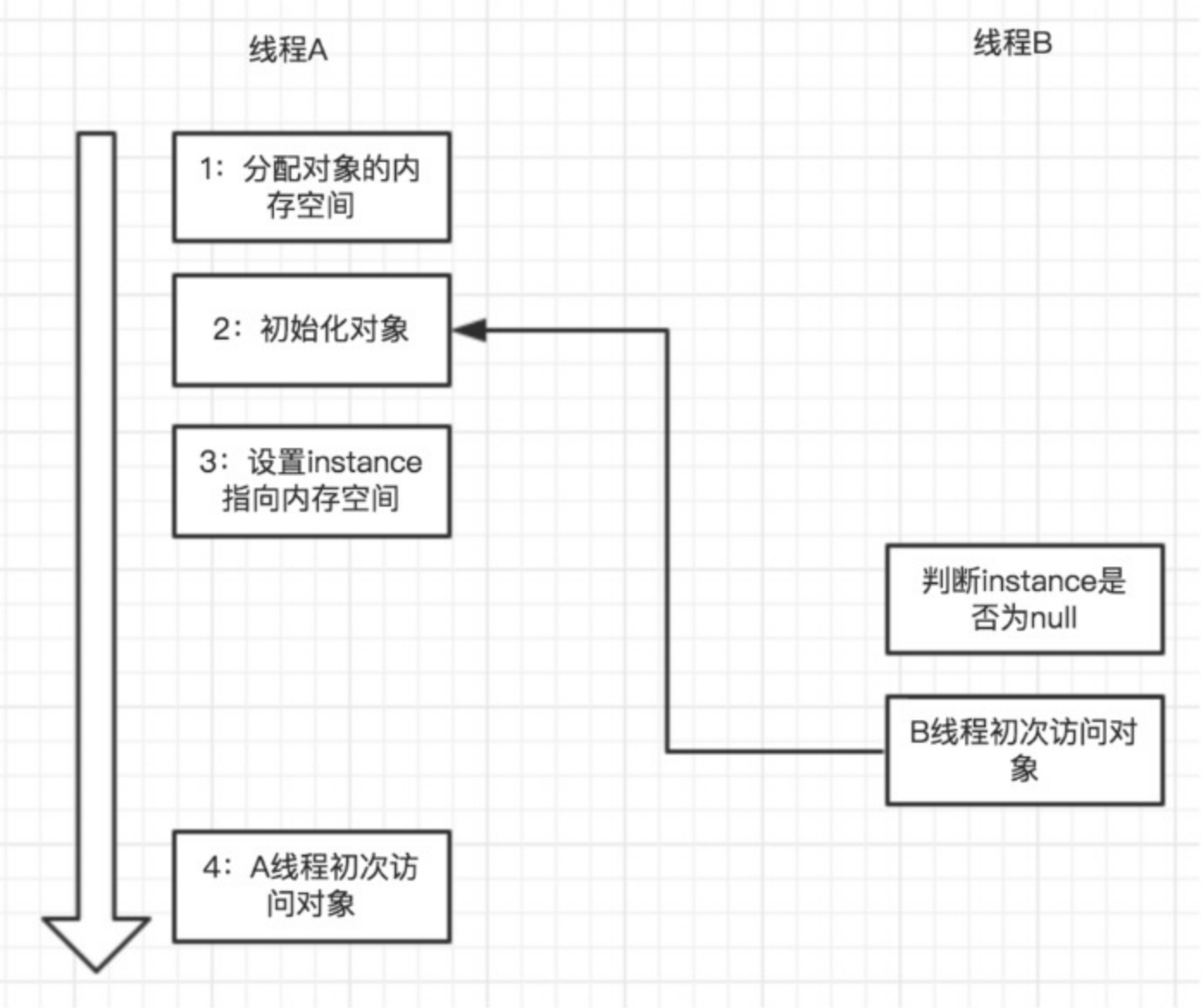
在知晓了问题发生的根源之后，我们可以想出两个办法来实现线程安全的延迟初始化。
1.不允许2和3重排序
2.允许2和3重排序，但不允许其他线程"看到"这个重排序。
基于上面这两点，提出两个解决方案。

3.1基于volatile的解决方案

对于前面的基于双重检查锁定来实现延迟初始化的方案，只需要做一点小的修改（把instance声明为volatile型），就可以实现线程安全的延迟初始化。请看下面的示例代码。

```java
public class SafeDoubleCheckedLocking {
    private volatile static Instance instance;
    public  static Instance getInstance(){
        if(instance ==null) {
            synchronized (SafeDoubleCheckedLocking.class) {
                if (instance == null)
                    instance = new Instance();            //instance为volatile，现在
            }
        }
        return instance;
    }
}
```

当声明对象的引用为volatile后，之前的3行伪代码中的2和3之间的重排序，在多线程环境中将会被禁止。上面的示例代码江安如下的时序执行。



多线程执行时序图

这个方案是通过禁止上图2和3之间的重排序，来保证线程安全的延迟初始化。

3.2基于类初始化的解决方案

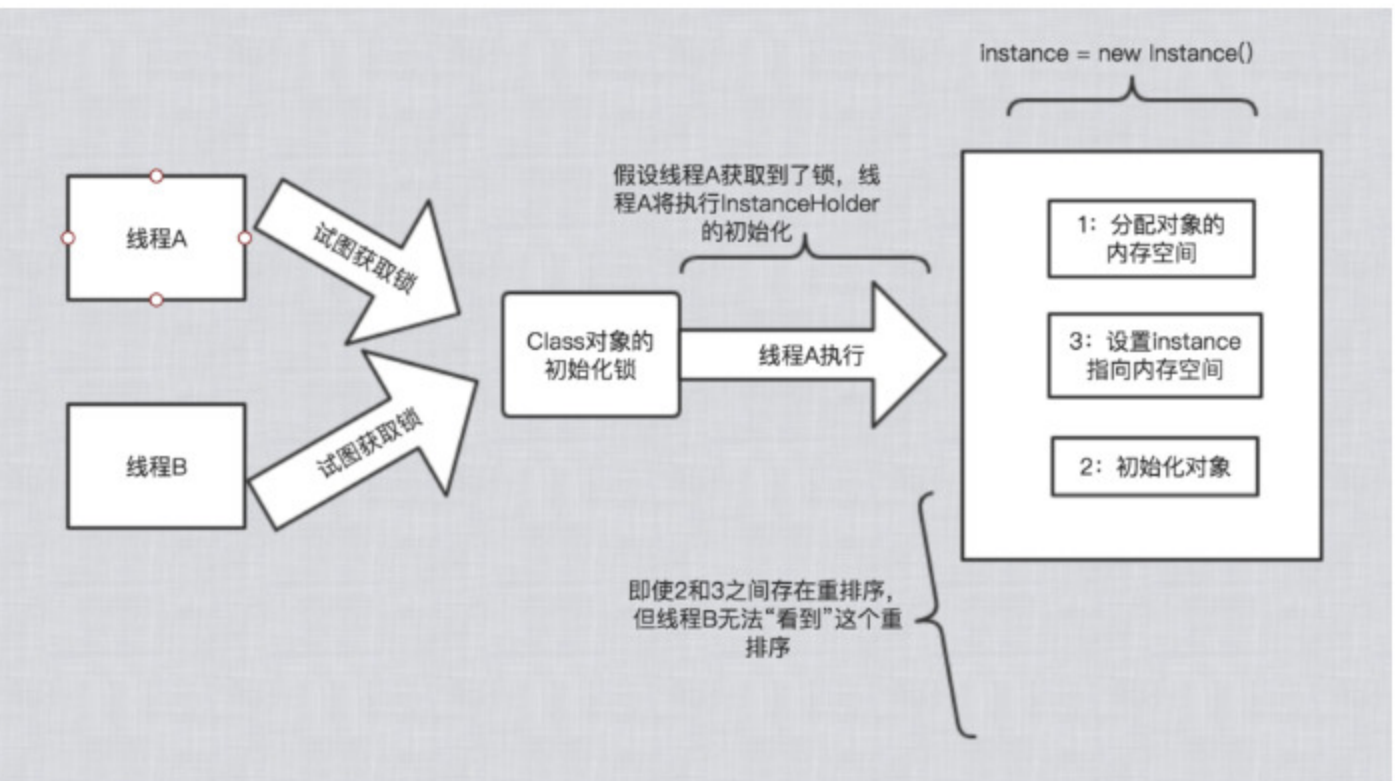
JVM在类的初始化阶段（即在Class被加载后，且被线程使用之前），会执行类的初始化。在执行类的初始化期间，JVM会去获取一个锁。这个锁可以同步多个线程对同一个类的初始化。

基于这个特性可以实现另一种线程安全的延迟初始化方案。

```java
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
    }
}
```

假设两个线程并发执行getInstance（）方法，下面是执行示意图。



两个线程并发执行的示意图

这个方案的实质是：允许之前的3行伪代码中的2和3重排序，但不允许非构造线程（这里指线程B）"看到"这个重排序。

4.总结

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，我们会发现基于类初始化的方案的实现代码更简洁。但基于volatile的双重检查锁定的方案有一个额外的优势：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。

字段延迟初始化降低了初始化类或创建实例的开销，但增加了访问被延迟初始化的字段的开销。在大多数时候，正常的初始化要优于延迟初始化。如果确实需要对实例字段使用线程安全的延迟初始化，请使用上面介绍的基于volatile的延迟初始化方案；如果确实需要对静态字段使用线程安全的延迟初始化，请使用基于类初始化的方案。

——摘自《*Java并发编程的艺术*》

小礼物走一走，来简书关注我
赞赏支持

日记本 举报文章 © 著作权归作者所有

尉昌达
写了 9454 字，被 15 人关注，获得了 4 个喜欢 +关注

喜欢 更多分享

下载简书 App
随时随地发现和创作内容

登录 后发表评论

评论

智慧如你，不想发表一点想法咩~

推荐阅读 更多精彩内容

油画棒，画得和油画一样棒
起这个标题，只是因为很顺口。 虽然油画棒带了油画两个字，并且也确实能表现出油画般的质感，但其实两者各有所长，各有特色。 画的时候，感觉也完全不一样。 对于油画，我才刚开始玩，画了也不超过十幅涂鸦，所以没什么发言权。 对于油画棒，已经用完两盒24色了，几十张涂鸦。 我总结出来一个诀窍就是，尽量涂厚一些，不要期望一次完工，往往要涂很多遍，不停叠加。 在这个过程中，颜料还会相互融合，形成新的颜色，也是乐趣之所在。 跟油画一样，白色颜料是最费的，所以可以单独买几十支白色油画棒。 我用的是樱花艺术家24色，延展性不错，没有颗粒杂志，没有异味。 关于纸张，效果最好的是油画纸，丝一般的顺滑.........
南蛮文子

英语长难句解读，记住这个公式就够了！
不背单词和语法，高效学英语→
广告

「你写好文，我送热门」简书头号玩家公会"送你上热...
注：本计划由"简书头号玩家公会"共同起草。 "简书头号玩家公会" 由持钻的简书原创作者组成。 "送你上热门"的名字，来自区块链内容平台 币乎 的民间活动。 这个活动由 币乎 用户自发组织。截至目前，这个活动已经成功帮助135位低权重优秀作者，登上币乎热门，为内容社区建设做出了不可磨灭的贡献。 实施简书版"送你上热门" 计划，和建立 简书头号玩家公会 的初衷，详见后文。 （一）计划宗旨 送优质原创文章上热门。 （二）实施方法 每周一三五，各选择1篇文章，由 简书头号玩家公会 成员合力点赞支持，送优质文章登上热门榜。 （三）目标成果 为简书优质作者提供热文榜曝光机会，切实提升收益和影响力。 （...
何思源

370调查组今日解散，关于活着，我想对自己说三句话
文/北崖 2018年11月30日， 马航MH370失踪事件调查组正式解散。 拾遗君今日发了篇文章， 有很多家属的故事， 每读一遍，都让人泪流满面。 也许，马航MH370离我们很远， 但生死，其实离我们很近。 关于活着，我想对自己说三句话。 1. 第一句话:走在时光前面， 能不怀念，就不要怀念。 生命中，有些人，再也不能相见。 他们突然离场， 不给人机会说声再见。 5个月前， 导师晚上跑步，再也没有回来， 没有留下一句话。 他爱喝酒，非常爱， 一场酒，会喝很长很长时间。 我特别头疼陪酒， 可是，还有酒喝吗？ 15年前， 父亲罹患癌症，没有挨过那年的冬天， 走得那天...
我是北崖君

比你有钱，比你有才，比你努力，还有什么理由埋怨？
连续看了三天的排行榜，关于热榜的信息，简书上有好几篇文章，从不同的角度分析，说得都很有道理。