从单例模式窥探类初始化过程中的同步处理机制

原创 xyzso1z 2020-06-21 02:09:26 104 收藏 原力计划 编辑 版权
分类专栏： # 并发

在Java多线程中，有时候需要采用**延迟初始化来降低初始化类和创建对象的开销**。双重检查锁定是常见的延迟初始化技术。但它是一个错误的用法。本文将分析双重检查锁定的错误根源，以及两种线程安全的延迟初始化方案。

一、 双重检查锁定的由来

下面代码是单例模式中比较常见的写法（**错误的、不安全**）：

```
public class UnsafeLazyInitialization {

        private static UnsafeLazyInitialization instance;

        public static UnsafeLazyInitialization getInstance(){
                if (instance==null) {                           //1: A 线程执行
                instance=new UnsafeLazyInitialization();  // 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` 引用的对象没有完成初始化（具体原因后面会详细讲解）。

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(**正确的、低效率**)：

```
public class UnsafeLazyInitialization {

        private static UnsafeLazyInitialization instance;

        public synchronized static UnsafeLazyInitialization getInstance(){
                if (instance==null) {
                        instance=new UnsafeLazyInitialization();
                }
                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()` 方法被多个线程频繁的调用，将会导致程序执行性能的下降。反之，如果 `getInstance()` 方法不会被多个线程频繁调用，那么这个延迟初始化方案将能提供令人满意的性能。

对于上述方法，有一个优化技巧(只有在未初始化时才做同步操作）：双重检查锁定 (Double-Checked Locking) 。通过双重检查锁定来降低同步的开销。代码如下：

```
1  public class DoubleCheckedLocking {       // 1
2          private static DoubleCheckedLocking instance;      // 2
3
4          public static DoubleCheckedLocking getInstance(){  // 3
5                  if (instance==null) {     // 4：第一次检查
6                          synchronized (DoubleCheckedLocking.class) { // 5: 加锁
7                                  if (instance==null) {  // 6：第二次检查
8                                          instance=new DoubleCheckedLocking(); // 7： 此
9                                  }
10                         }
11                 }
12                 return instance;
13         }
14
15 }
```

如上面代码所示,如果第一次检查 `instance` 不为 `null` ，那么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。上面代码表面上看起来两全其美。

- 多个线程试图在同一事件创建对象时，会通过加锁来报正直有一个线程创建对象。
- 在对象创建好之后，执行 `getInstance()` 方法不需要获取锁，直接返回已创建好的对象。

双重检查锁定看起来很完美，但这是一个错误的优化！在代码执行到第 4 行，**代码读取到 `instance` 不为 null 时，`instance` 引用的对象有可能还没有完成初始化**，这个问题引入下面重点知识对象初始化流程。

二、 问题的根源

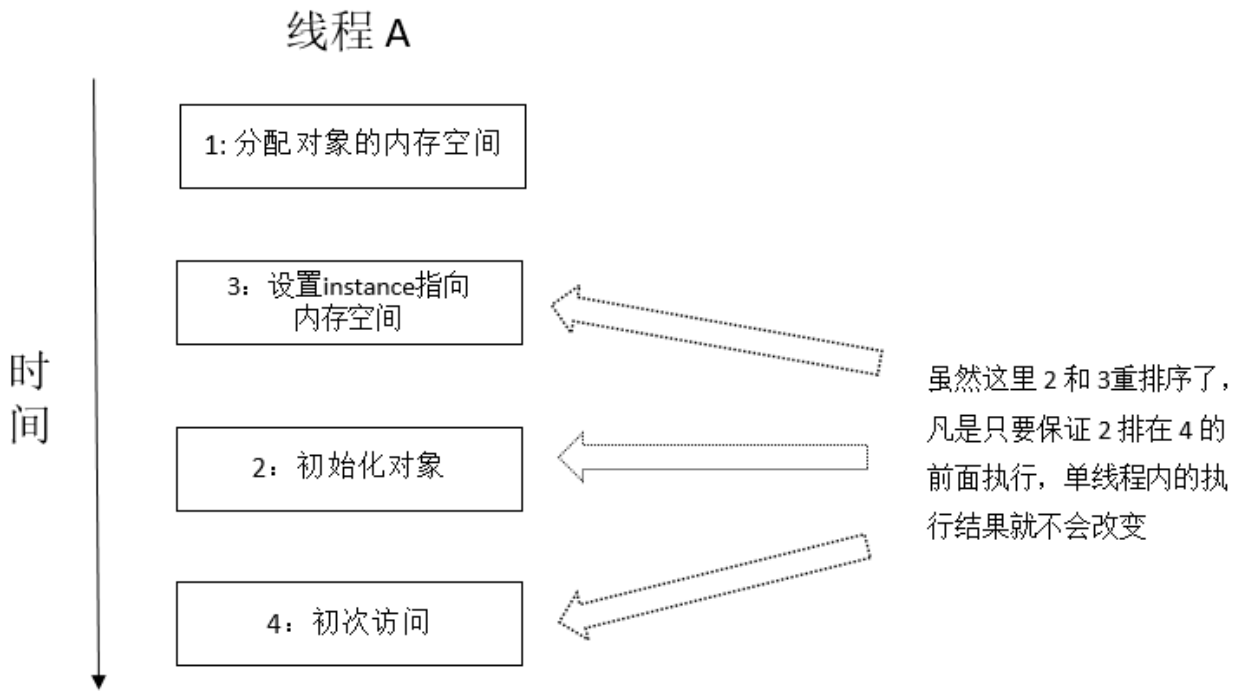
上面的双重检查锁定示例代码的第 7 行(`instance = new DoubleCheckedLocking()`) 创建了一个对象。这一行代码可以分解为如下的 3 行伪代码：

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

上面 3 行伪代码中的 2 和 3 之间，可能会被重排序。2 和 3 之间重排序会后的执行如下：

```
memory = allocate(); // 1： 分配对象的内存空间
instance = memory;   // 3: 设置instance 指向 2 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
```

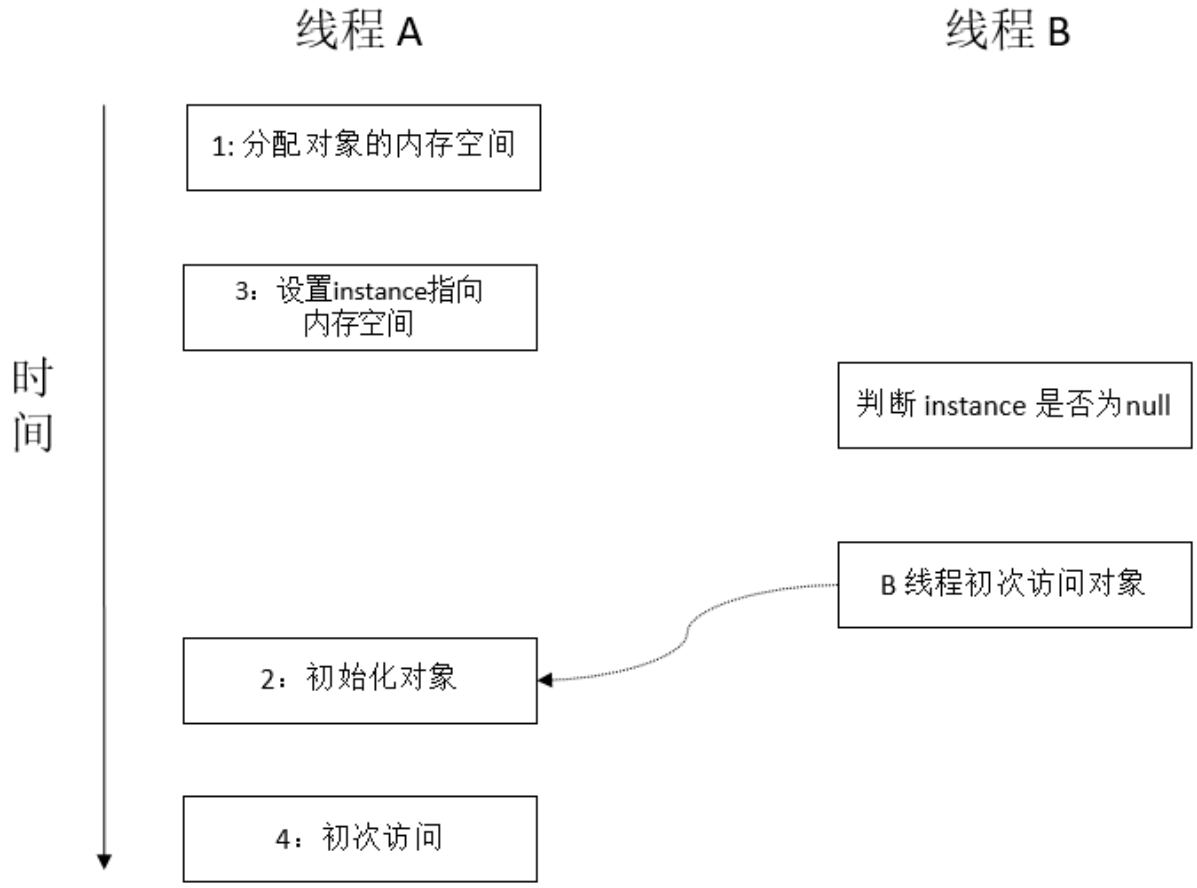
跟据Java语言规范，所有线程在执行Java程序时必须遵守 intra-thread semantics 。intra-thread semantics 保证重排序不会改变单线程内的程序执行结果。换句话说，**intra-thread semantics 允许那些在单线程内，不会改变单线程程序执行结果的重排序**。上面 3 行伪代码的 2 和 3 之间虽然被重排序了，但这个重排序并不会违反 intra-thread semantics 。这个重排序在没有改变单线程程序执行结果的前提下，可以提高程序的执行性能。
为了更好地理解 intra-thread semantics ，可以看下图(假设一个 线程A 在构造对象后，立即访问这个对象)：



线程执行的时序图

https://blog.csdn.net/xyzso1z

如上图，**只要保证 2 排在 4 的前面，即使 2 和 3 之间重排序了，也不会违反 intranet-thread semantics。**
下面我们看看多线程并发执行的情况，如下图：



多线程执行的时序图

https://blog.csdn.net/xyzso1z

由于单线程内要遵守 intra-thread semantice，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 线程B 按上图的时序执行时，B线程 将看到一个还没有被初始化的对象。

再回到 `DoubleCheckedLocking` 示例代码的第 7 行(`instance=new DoubleCheckedLocking()`) 如果发生重排序，另一个并发执行的 线程B 就有可能在第 4 行(`if (instance==null)`) 判断 `instance` 不为 `null` 。**线程B 接下来将访问instance 所引用的对象，但此时这个对象可能还没有被 线程A 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。
   下面将对这两种方案进行讲解实现。

三、 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(`DoubleCheckedLocking`)，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：

```
public class DoubleCheckedLocking {
        private volatile static DoubleCheckedLocking instance;

        public static DoubleCheckedLocking getInstance(){
                if (instance==null) {
                        synchronized (DoubleCheckedLocking.class) {
                                if (instance==null) {
                                        instance=new DoubleCheckedLocking();
                                }
                        }
                }
                return instance;
        }

}
```

当声明对象的引用为 `volatile` 后，上代码中 2 和 3 之间的重排序在多线程环境中将会禁止。

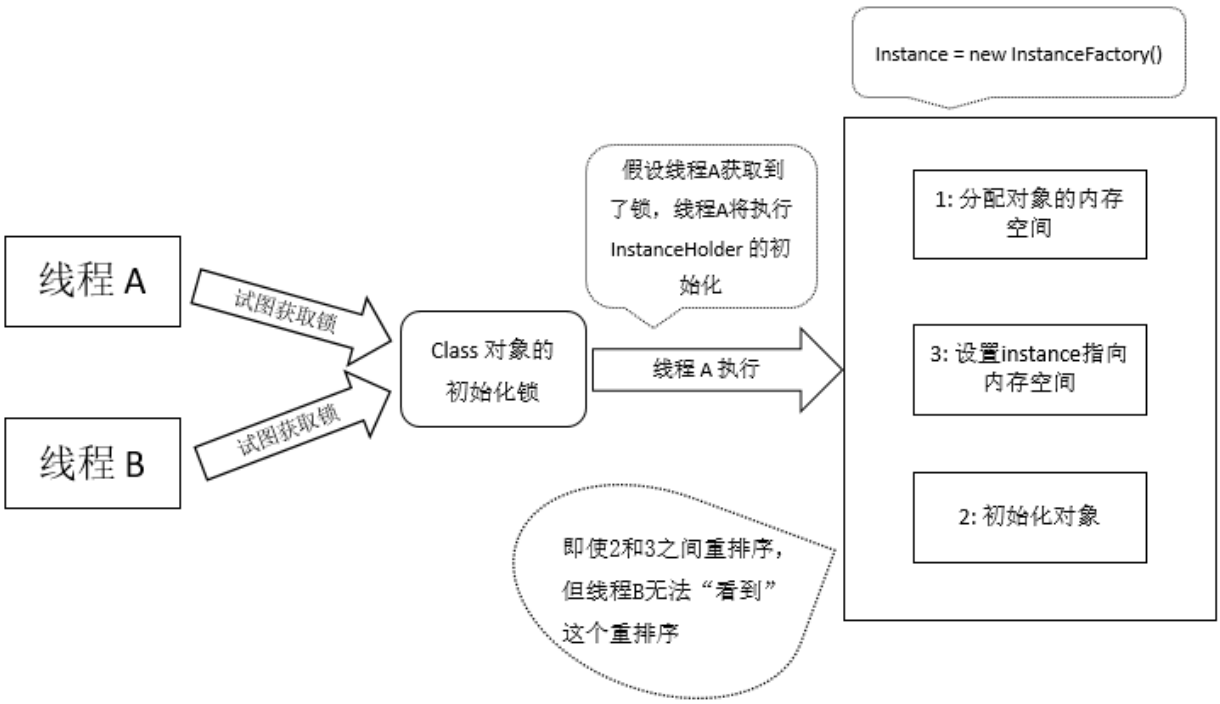
四、 基于类初始化的解决方案

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话初始化方案(Initialization On Demand Holder idiom)。

```
public class InstanceFactory {
        private static class InstanceHolder{
                public static InstanceFactory instance = new InstanceFactory();
        }

        public static InstanceFactory getInstace(){
                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，下面是执行的示意图：



两个线程并发执行的示意图 https://blog.csdn.net/xyzso1z

这个方案的实质是：**允许 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。**
初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。在首次发生下列任意一种情况时，一个类或接口类型T将被立即初始化：

1. T 是一个类，而且一个T类型的实例被创建。
2. T 是一个类，且T中声明的一个静态方法被调用。
3. T 中声明的一个静态字段被赋值。
4. T 中声明的一个静态字段被使用，而且这个字段不是一个常量字段。
5. T 是一个顶级类，而且一个断言语句嵌套在T内部被执行。

在 `InstanceFactory` 示例中，首次执行 `getInstaance()` 方法的现成将导致 `InstanceHolder` 类被初始化(符合情况4)。

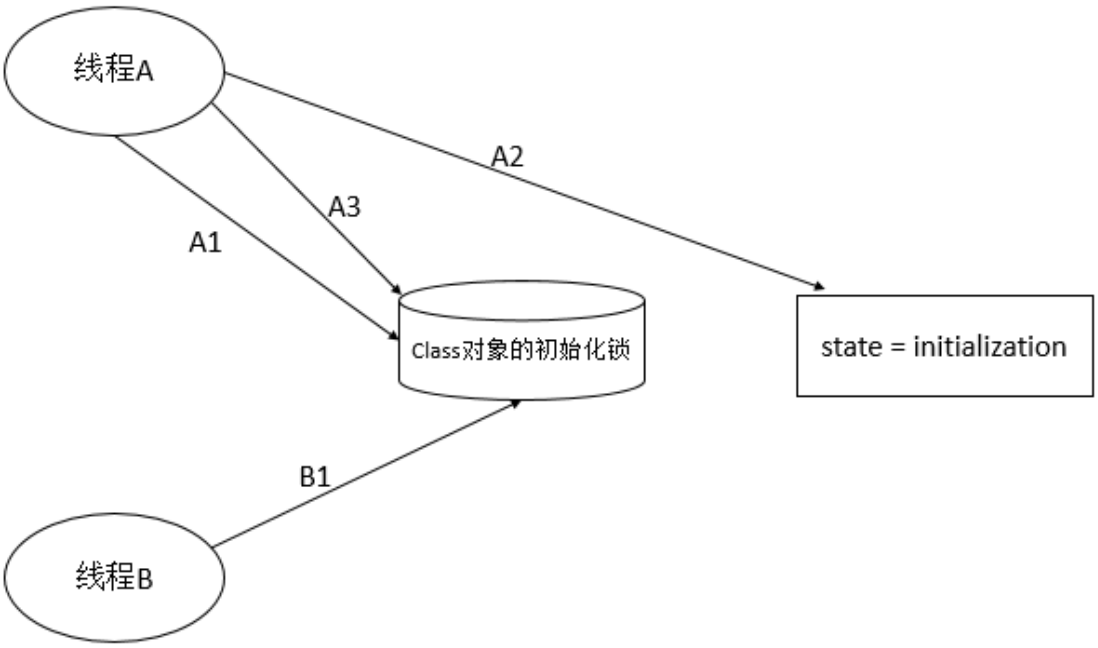
五、类的初始化流程

由于Java语言是多线程的，多个线程可能在同一时间尝试去初始化同一个类或接口。因此，在Java中初始化一个类或者接口时，需要做细致的同步处理。
Java语言规范规定，**对于每一个类或者接口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在类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已经被初始化过了。**
对于类或接口的初始化，Java语言规范制定了精巧而复杂的类初始化处理过程。Java初始化一个类或接口的处理过程如下(这里对类初始化处理过程的说明，省略了与本无关的部分。同时为了更好的说明类初始化过程中的同步处理机制，我们人为的把类初始化的处理过程分为了5个阶段)。

第 1 阶段

通过在 `Class` 对象上同步(即获取 `Class` 对象的初始化锁),来控制类或接口的初始化。这个获取锁的线程会一直等待，知道当前线程能够获取到这个初始化锁。

假设 `Class` 对象当前还没有被初始化(初始化状态 `state` ，此时被标记为 `state = noInitialization`)，且有两个线程A 和 B 试图同时初始化这个 `Class` 对象。如图：



类初始化—第一阶段

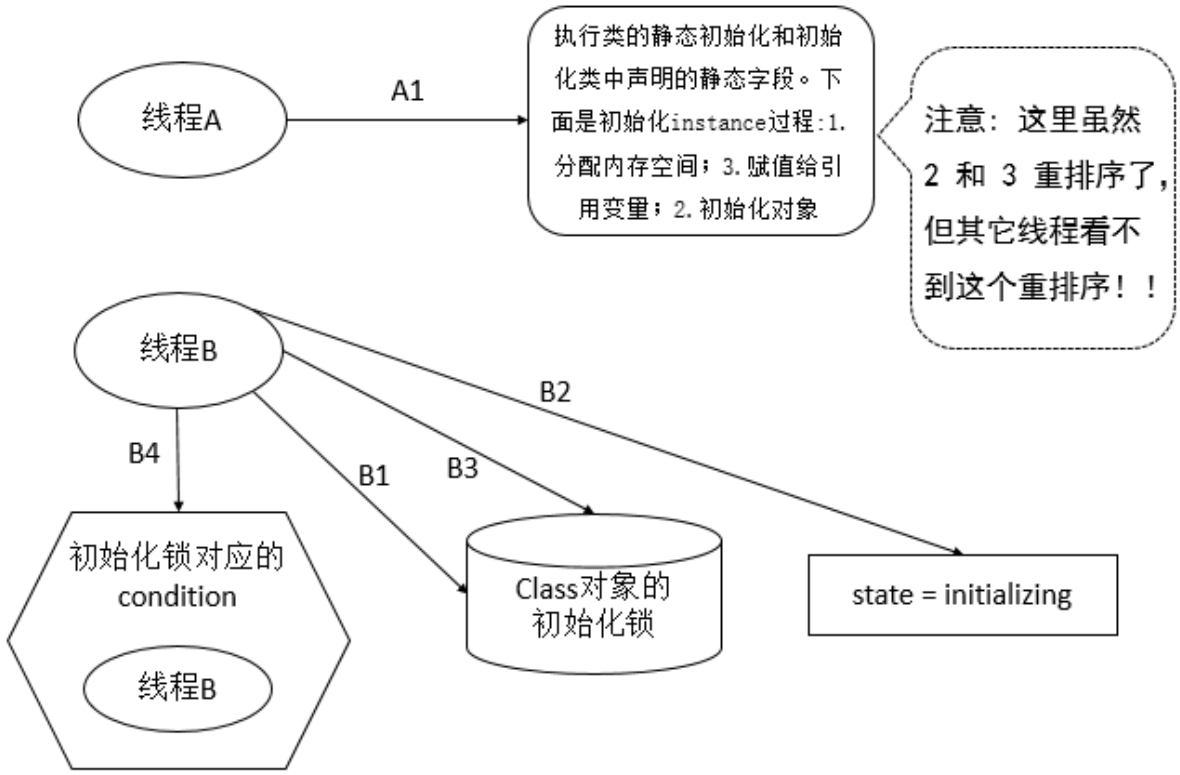
https://blog.csdn.net/xyzso1z

第 1 阶段的执行时序表：

时间	线程 A	线程B
t1	A1: 尝试获取Class对象的初始化锁。这里假设线程A获取到了初始化锁	B1: 尝试获取Class对象的初始化锁，由了锁，线程B将一直等待获取初
t2	A2: 线程A看到对象还未被初始化(因为读取到 `state = noInitialization`)，线程A设置 `state = initializing`	
t3	A3: 线程A释放初始化锁	

第 2 阶段

线程A 执行类的初始化，同时线程B 在初始化锁对应的condition 上等待。



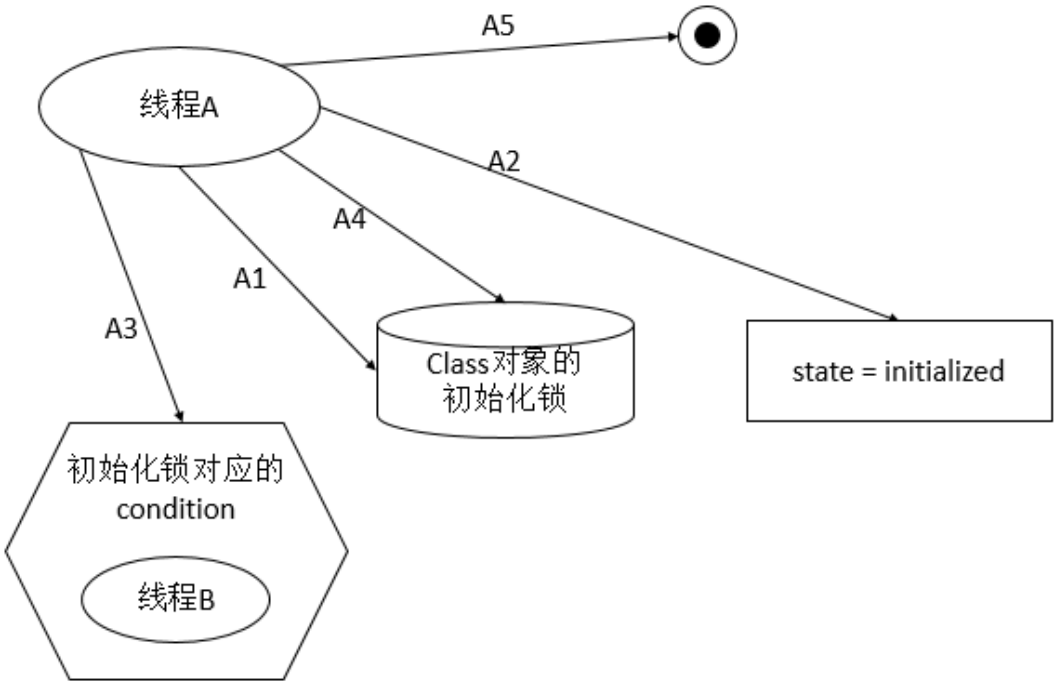
类初始化—第 2 阶段 https://blog.csdn.net/xyzso1z

第 2 阶段的执行时序表：

时间	线程 A	线程B
t1	A1: 执行类的静态初始化和初始化类中声明的静态字段	B1: 获取到初始化锁
t2		B2: 读取到 `state = initi`
t3		B3: 释放初始化锁
t4		B4: 在初始化锁的conditi

第 3 阶段

线程A 设置 `state = initialized` ,然后唤醒在 `condition` 中等待的所有线程。



类初始化—第3阶段

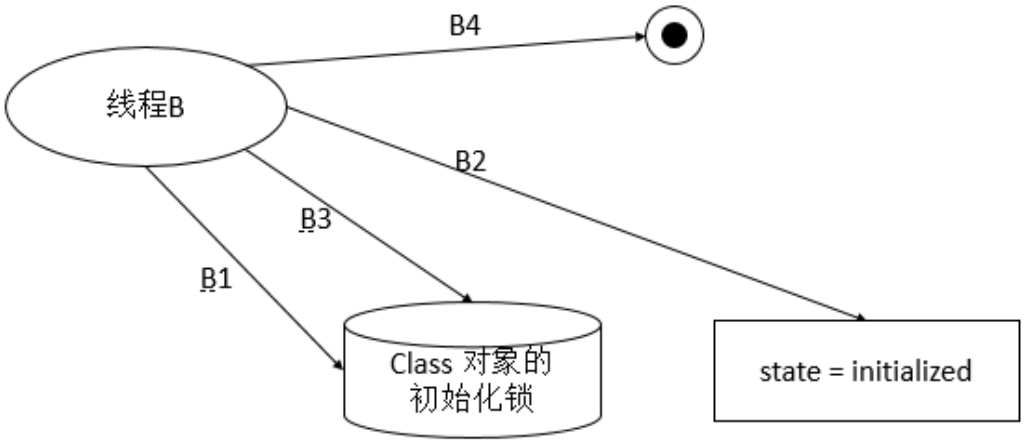
https://blog.csdn.net/xyzso1z

第 3 阶段的执行时序表：

时间	线程 A
t1	A1: 获取初始化锁
t2	A2: 设置 `state = initialized`
t3	A3: 唤醒在condition中等待的所有线程
t4	A4: 释放初始化锁
t5	A5: 线程A的初始化处理过程完成

第 4 阶段

线程B 结束类的初始化处理

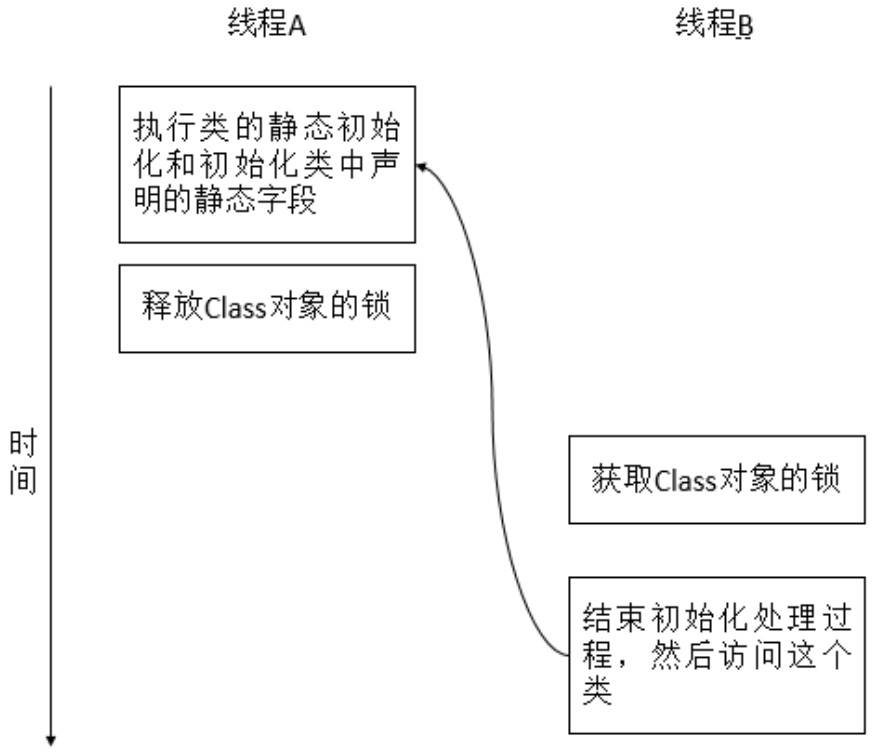


类初始化—第4阶段

https://blog.csdn.net/xyzso1z

第 4 阶段的执行时序表：

时间	线程 B
t1	B1: 获取初始化锁
t2	B2: 读取到 `state = initialized`
t3	B3: 释放初始化锁
t4	B4: 线程B的类初始化处理过程完成



多线程执行的时序图

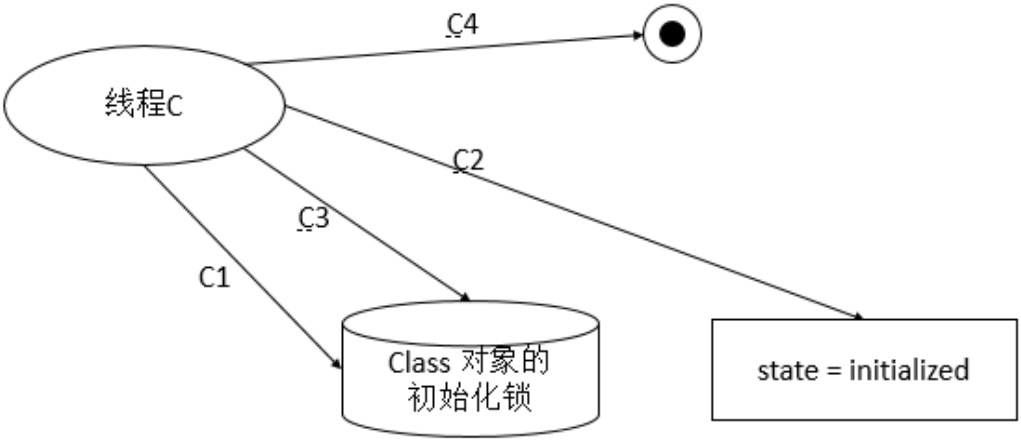
https://blog.csdn.net/xyzso1z

线程A 在第 2 阶段的A1执行类的初始化，并在第 3 阶段的A4释放初始化锁；线程B在第 4 阶段的B1获取同一个初始化锁，并在第 4 阶段的B4之后才开始访问这个类。

根据Java内存模型规范的锁规则，这里将存在如下的happens-before关系：**线程A 执行类的初始化时的写入操作（执行类的静态初始化和初始化类中的声明的静态字段），线程B一定能看到。**

第 5 阶段

线程C执行类的初始化的处理



类初始化—第5阶段

https://blog.csdn.net/xyzso1z

第 5 阶段的执行时序表：

时间	线程 C
t1	C1: 获取初始化锁
t2	C2: 读取到 `state = initialized`
t3	C3: 释放初始化锁
t4	C4: 线程B的类初始化处理过程完成

在第 3 阶段之后，类已经完成了初始化。因此线程C 在第 5 阶段的类初始化处理过程相对简单一些（**前面的线程A 和 B 的初始化处理过程都经历了两次获取-释放锁，而线程C 的类初始化处理只需要经历一次锁获取-释放**）。

> 注意： 这里的 `condition` 和 `state` 标记是本文虚构的，Java语言规范并没有硬性规定一定要使用 `condition` 和 `state` 标记。JVM的具体实现只要实现类似的功能即可。

六、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化字段的开销。**在大多数的时候，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，请使用上面介绍的基于类初始化方案。**

本文选自《Java并发编程的艺术》