二

63 单例模式的双重检查锁模式为什么必须加 volatile?

本课时我们主要讲解单例模式的双重检查锁模式为什么必须加 volatile?

什么是单例模式

单例模式指的是，保证一个类只有一个实例，并且提供一个可以全局访问的入口。

为什么需要使用单例模式

那么我们为什么需要单例呢？其中**一个理由，那就是为了节省内存、节省计算。**因为在很多情况下，我们只需要一个实例就够了，如果出现更多的实例，反而纯属浪费。

下面我们举一个例子来说明这个情况，以一个初始化比较耗时的类来说，代码如下所示：

```
public class ExpensiveResource {

    public ExpensiveResource() {

        field1 = // 查询数据库

        field2 = // 然后对查到的数据做大量计算

        field3 = // 加密、压缩等耗时操作

    }

}
```

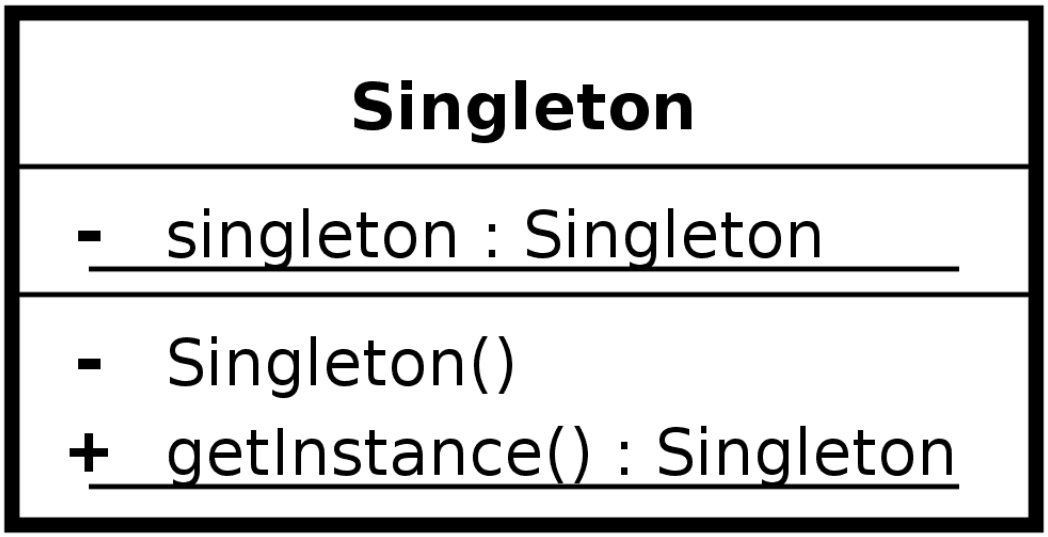
这个类在构造的时候，需要查询数据库并对查到的数据做大量计算，所以在第一次构造时，我们花了很多时间来初始化这个对象。但是假设数据库里的数据是不变的，我们就可以把这个对象保存在内存中，那么以后开发的时候就可以直接用这同一个实例了，不需要再次构建新实例。如果每次都重新生成新的实例，则会造成更多的浪费，实在没有必要。

接下来看看需要单例的第二个理由，那就是为了保证结果的正确。**比如我们需要一个全局

的计数器，用来统计人数，如果有多个实例，反而会造成混乱。

另外呢，就是为了方便管理。**很多工具类，我们只需要一个实例，那么我们通过统一的入口，比如通过 getInstance 方法去获取这个单例是很方便的，太多实例不但没有帮助，反而会让人眼花缭乱。

一般单例模式的类结构如下图所示：有一个私有的 Singleton 类型的 singleton 对象；同时构造方法也是私有的，为了防止他人调用构造函数来生成实例；另外还会有一个 public 的 getInstance 方法，可通过这个方法获取到单例。



双重检查锁模式的写法

单例模式有多种写法，我们重点介绍一下和 volatile 强相关的双重检查锁模式的写法，代码如下所示：

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {

    }

    public static Singleton getInstance() {

        if (singleton == null) {
```

```
                synchronized (Singleton.class) {
                    if (singleton == null) {
                        singleton = new Singleton();
                    }
                }
            }
            return singleton;
        }
    }
```

在这里我将重点讲解 getInstance 方法，方法中首先进行了一次 if (singleton == null) 的检查，然后是 synchronized 同步块，然后又是一次 if (singleton == null) 的检查，最后是 singleton = new Singleton() 来生成实例。

我们进行了两次 if (singleton == null) 检查，这就是“双重检查锁”这个名字的由来。这种写法是可以保证线程安全的，假设有两个线程同时到达 synchronized 语句块，那么实例化代码只会由其中先抢到锁的线程执行一次，而后抢到锁的线程会在第二个 if 判断中发现 singleton 不为 null，所以跳过创建实例的语句。再后面的其他线程再来调用 getInstance 方法时，只需判断第一次的 if (singleton == null)，然后会跳过整个 if 块，直接 return 实例化后的对象。

这种写法的优点是不仅线程安全，而且延迟加载、效率也更高。

讲到这里就涉及到了一个常见的问题，面试官可能会问你，“为什么要 double-check？去掉任何一次的 check 行不行？”

我们先来看第二次的 check，这时你需要考虑这样一种情况，有两个线程同时调用 getInstance 方法，由于 singleton 是空的，因此两个线程都可以通过第一重的 if 判断；然后由于锁机制的存在，会有一个线程先进入同步语句，并进入第二重 if 判断，而另外的一个线程就会在外面等待。

不过，当第一个线程执行完 new Singleton() 语句后，就会退出 synchronized 保护的区域，这时如果没有第二重 if (singleton == null) 判断的话，那么第二个线程也会创建一个实例，此时就破坏了单例，这肯定是不行的。

而对于第一个 check 而言，如果去掉它，那么所有线程都会串行执行，效率低下，所以两个 check 都是需要保留的。

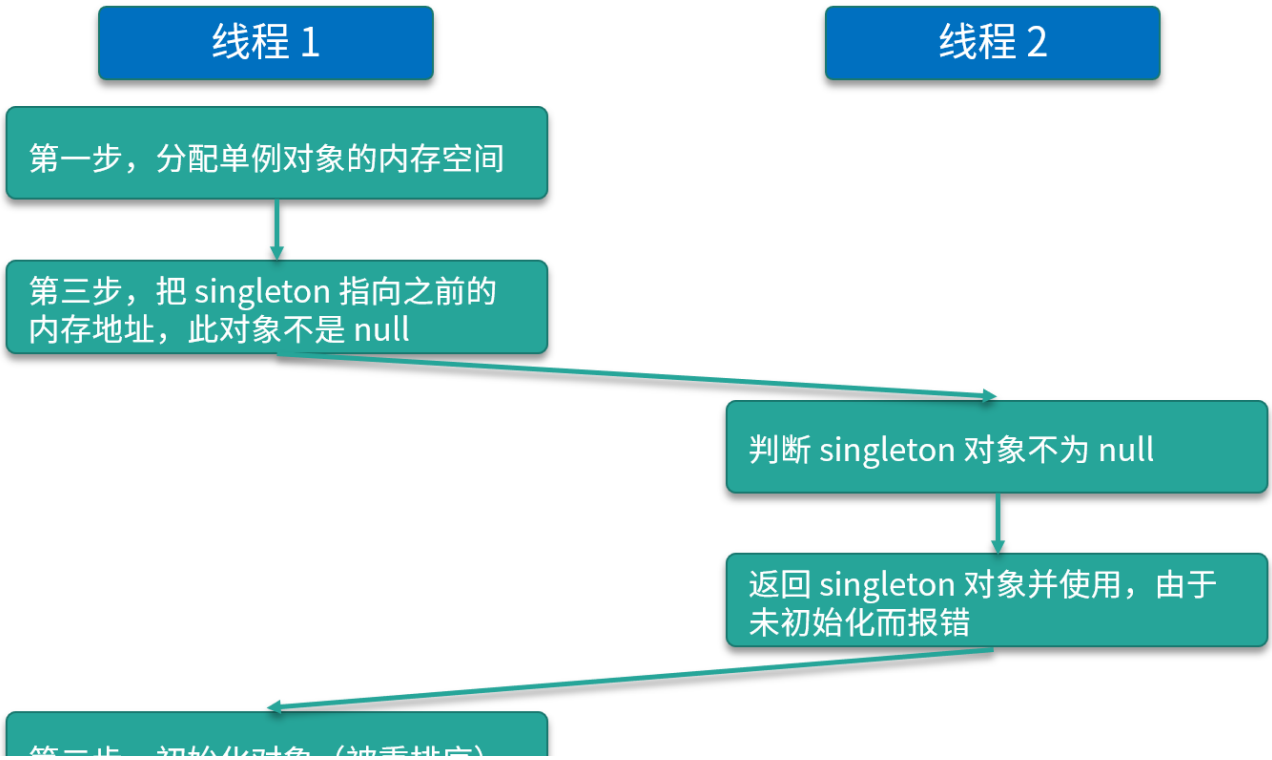
在双重检查锁模式中为什么需要使用 volatile 关键字

相信细心的你可能看到了，我们在双重检查锁模式中，给 singleton 这个对象加了 volatile 关键字，那**为什么要用 volatile 呢？**主要就在于 singleton = new Singleton()，它并非是一个原子操作，事实上，在 JVM 中上述语句至少做了以下这 3 件事：

- 第一步是给 singleton 分配内存空间；
- 然后第二步开始调用 Singleton 的构造函数等，来初始化 singleton；
- 最后第三步，将 singleton 对象指向分配的内存空间（执行完这步 singleton 就不是 null 了）。

这里需要留意一下 1-2-3 的顺序，因为存在指令重排序的优化，也就是说第2 步和第 3 步的顺序是不能保证的，最终的执行顺序，可能是 1-2-3，也有可能是 1-3-2。

如果是 1-3-2，那么在第 3 步执行完以后，singleton 就不是 null 了，可是这时第 2 步并没有执行，singleton 对象未完成初始化，它的属性的值可能不是我们所预期的值。假设此时线程 2 进入 getInstance 方法，由于 singleton 已经不是 null 了，所以会通过第一重检查并直接返回，但其实这时的 singleton 并没有完成初始化，所以使用这个实例的时候会报错，详细流程如下图所示：



线程 1 首先执行新建实例的第一步，也就是分配单例对象的内存空间，由于线程 1 被重排序，所以执行了新建实例的第三步，也就是把 singleton 指向之前分配出来的内存地址，在这第三步执行之后，singleton 对象便不再是 null。

这时线程 2 进入 getInstance 方法，判断 singleton 对象不是 null，紧接着线程 2 就返回 singleton 对象并使用，由于没有初始化，所以报错了。最后，线程 1 “姗姗来迟”，才开始执行新建实例的第二步——初始化对象，可是这时的初始化已经晚了，因为前面已经报错了。

使用了 volatile 之后，相当于是表明了该字段的更新可能是在其他线程中发生的，因此应确保在读取另一个线程写入的值时，可以顺利执行接下来所需的操作。在 JDK 5 以及后续版本所使用的 JMM 中，在使用了 volatile 后，会一定程度禁止相关语句的重排序，从而避免了上述由于重排序所导致的读取到不完整对象的问题的发生。

到这里关于“为什么要用 volatile” 的问题就讲完了，使用 volatile 的意义主要在于它可以防止避免拿到没完成初始化的对象，从而保证了线程安全。

总结

在本课时中我们首先介绍了什么是单例模式，以及为什么需要使用单例模式，然后介绍了双重检查锁模式这种写法，以及面对这种写法时为什么需要 double-check，为什么需要用 volatile？最主要的是为了保证线程安全。

上一页 下一页