<https://mp.weixin.qq.com/s?__biz=MzIyNDU2ODA4OQ==&mid=2247483927&idx=1&sn=215abdc5fa5151cefcb7dc2ee5b90835&chksm=e80db461df7a3d77b3472a391e6a72f22c333bd01ac1455dbcccb91f72ed79cbd919e17345ef&scene=21#wechat_redirect>

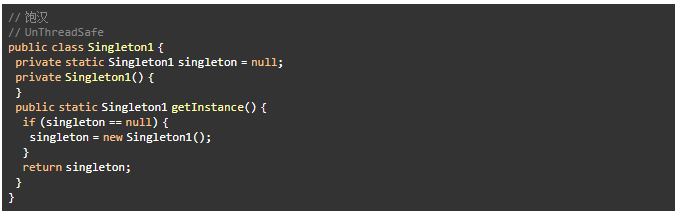
# 单例模式有几种写法？

## 饱汉模式

饱汉是变种最多的单例模式。我们从饱汉出发，通过其变种逐渐了解实现单例模式时需要关注的问题。

#### 基础的饱汉

饱汉，即已经吃饱，不着急再吃，饿的时候再吃。所以他就先不初始化单例，等第一次使用的时候再初始化，即“懒加载”。

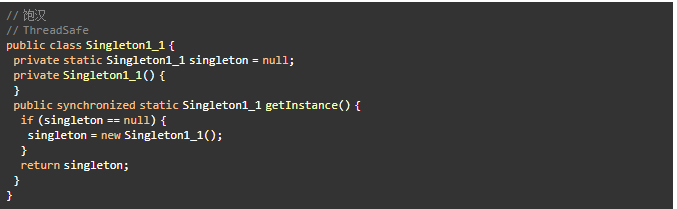


饱汉模式的核心就是懒加载。好处是更启动速度快、节省资源，一直到实例被第一次访问，才需要初始化单例；小坏处是写起来麻烦，大坏处是线程不安全，if语句存在竞态条件。

写起来麻烦不是大问题，可读性好啊。因此，单线程环境下，基础饱汉是笔者最喜欢的写法。但多线程环境下，基础饱汉就彻底不可用了。下面的几种变种都在试图解决基础饱汉线程不安全的问题。

#### 饱汉 - 变种 1

最粗暴的犯法是用synchronized关键字修饰getInstance()方法，这样能达到绝对的线程安全。

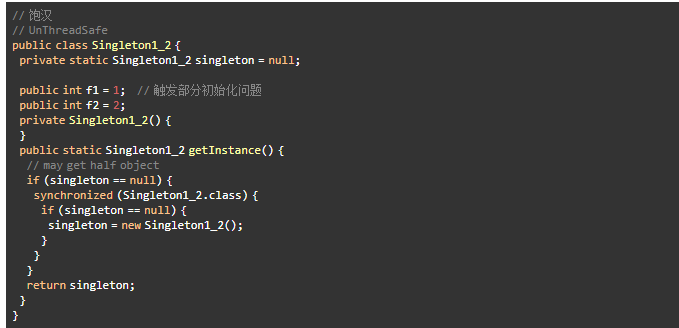


变种1的好处是写起来简单，且绝对线程安全；坏处是并发性能极差，事实上完全退化到了串行。单例只需要初始化一次，但就算初始化以后，synchronized的锁也无法避开，从而getInstance()完全变成了串行操作。性能不敏感的场景建议使用。

#### 饱汉 - 变种 2

变种2是“臭名昭著”的DCL 1.0。

针对变种1中单例初始化后锁仍然无法避开的问题，变种2在变种1的外层又套了一层check，加上synchronized内层的check，即所谓“双重检查锁”（Double Check Lock，简称DCL）。



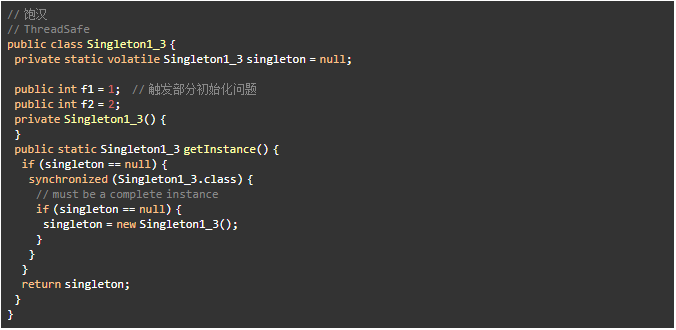
变种2的核心是DCL，看起来变种2似乎已经达到了理想的效果：懒加载+线程安全。可惜的是，正如注释中所说，DCL仍然是线程不安全的，由于指令重排序，你可能会得到“半个对象”，即”部分初始化“问题。详细在看完变种3后，可参考下面这篇文章，这里不再赘述。

https://monkeysayhi.github.io/2016/11/29/volatile关键字的作用、原理/

#### 饱汉 - 变种 3

变种3专门针对变种2，可谓DCL 2.0。

针对变种3的“半个对象”问题，变种3在instance上增加了volatile关键字，原理见上述参考。

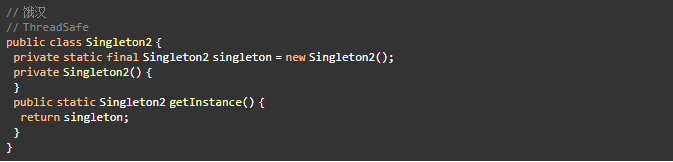


多线程环境下，变种3更适用于性能敏感的场景。但后面我们将了解到，就算是线程安全的，还有一些办法能破坏单例。

当然，还有很多方式，能通过与volatile类似的方式防止部分初始化。读者可自行阅读内存屏障相关内容，但面试时不建议主动装逼。

## 饿汉模式

与饱汉相对，饿汉很饿，只想着尽早吃到。所以他就在最早的时机，即类加载时初始化单例，以后访问时直接返回即可。

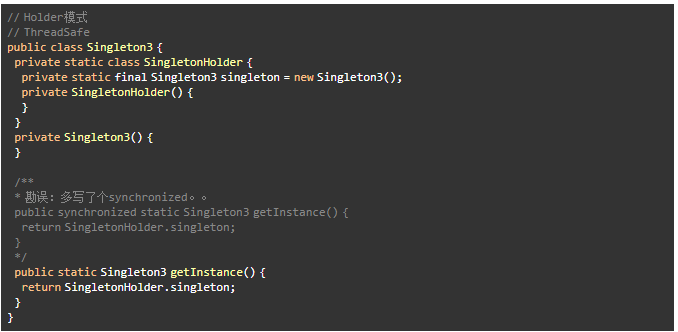


饿汉的好处是天生的线程安全（得益于类加载机制），写起来超级简单，使用时没有延迟；坏处是有可能造成资源浪费（如果类加载后就一直不使用单例的话）。

值得注意的时，单线程环境下，饿汉与饱汉在性能上没什么差别；但多线程环境下，由于饱汉需要加锁，饿汉的性能反而更优。

## Holder模式

我们既希望利用饿汉模式中静态变量的方便和线程安全；又希望通过懒加载规避资源浪费。Holder模式满足了这两点要求：核心仍然是静态变量，足够方便和线程安全；通过静态的Holder类持有真正实例，间接实现了懒加载。



相对于饿汉模式，Holder模式仅增加了一个静态内部类的成本，与饱汉的变种3效果相当（略优），都是比较受欢迎的实现方式。同样建议考虑。

## 枚举模式

用枚举实现单例模式，相当好用，但可读性是不存在的。

#### 基础的枚举

将枚举的静态成员变量作为单例的实例：



代码量比饿汉模式更少。但用户只能直接访问实例Singleton4.SINGLETON——事实上，这样的访问方式作为单例使用也是恰当的，只是牺牲了静态工厂方法的优点，如无法实现懒加载。

## 丑陋但好用的语法糖

Java的枚举是一个“丑陋但好用的语法糖”。

#### 枚举型单例模式的本质

通过反编译打开语法糖，就看到了枚举类型的本质，简化如下：

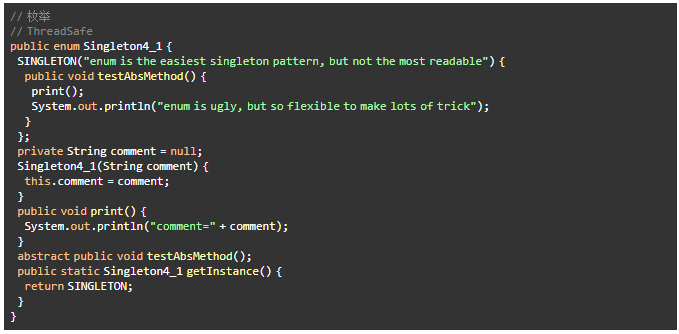


本质上和饿汉模式相同，区别仅在于公有的静态成员变量。

#### 用枚举实现一些trick

这一部分与单例没什么关系，可以跳过。如果选择阅读也请认清这样的事实：虽然枚举相当灵活，但如何恰当的使用枚举有一定难度。一个足够简单的典型例子是TimeUnit类，建议有时间耐心阅读。

上面已经看到，枚举型单例的本质仍然是一个普通的类。实际上，我们可以在枚举型型单例上增加任何普通类可以完成的功能。要点在于枚举实例的初始化，可以理解为实例化了一个匿名内部类。为了更明显，我们在Singleton4\_1中定义一个普通的私有成员变量，一个普通的公有成员方法，和一个公有的抽象成员方法，如下：



这样，枚举类Singleton4\_1中的每一个枚举实例不仅继承了父类Singleton4\_1的成员方法print()，还必须实现父类Singleton4\_1的抽象成员方法testAbsMethod()。

## 总结

上面的分析都忽略了反射和序列化的问题。通过反射或序列化，我们仍然能够访问到私有构造器，创建新的实例破坏单例模式。此时，只有枚举模式能天然防范这一问题。反射和序列化笔者还不太了解，但基本原理并不难，可以在其他模式上手动实现。

下面继续忽略反射和序列化的问题，做个总结回味一下：

