# 单例设计模式

在某些系统中，为了节省系统内存资源、保证数据的一致性，要求某些类只产生一个实例，这就是所谓的单例模式。

# 单例模式的定义和特点

单例模式的定义：单例类只有一个实例，且实例由单例类创建。例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，以及出现各个窗口显示内容的不一致的错误。

在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

单例模式有3个特点：

1. 单例类最多存在一个实例。
2. 单例类负责创建自己的唯一实例。
3. 单例类对外提供一个访问实例的接口。

# 单例模式的主要意图

避免某些全局使用的类频繁的创建和销毁，减少系统开销，提升程序性能。

# 单例模式的实现思路

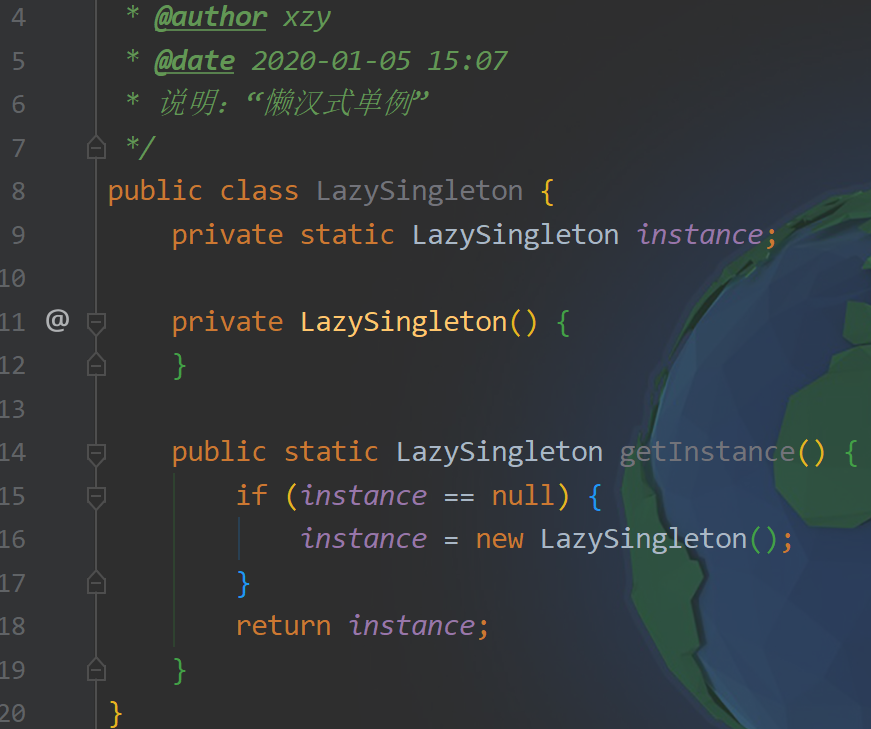
单例模式是最简单的软件设计模式之一。通常，普通类的构造函数的访问权限是公有的（public），外部类可以用过“new 构造函数()”的形式创建任意多个类的实例。将构造函数的访问权限设置为私有的（private），能够阻止外部类任意创建类的实例。

在将构造函数的访问权限设置为私有的同时，在类中添加一个静态私有实例，并向外提供一个静态的公有方法用于访问实例，即可实现类的实例数量的控制。

# **Talk is cheap, Show me the code**

单例模式主要有以下集中实现方式。

## 懒汉式（线程不安全）



**实例创建时机**：首次尝试获取实例时

**是否线程安全**：否

**实现难度**：易

**描述**：这是实现单例模式的最基本方式，这种方式存在的最大问题是线程不安全，所以从严格意义上讲，它并不算单例模式（如果多个线程同时调用getInstance()方法，每个线程可能返回不同的实例）

## 懒汉式（线程安全）



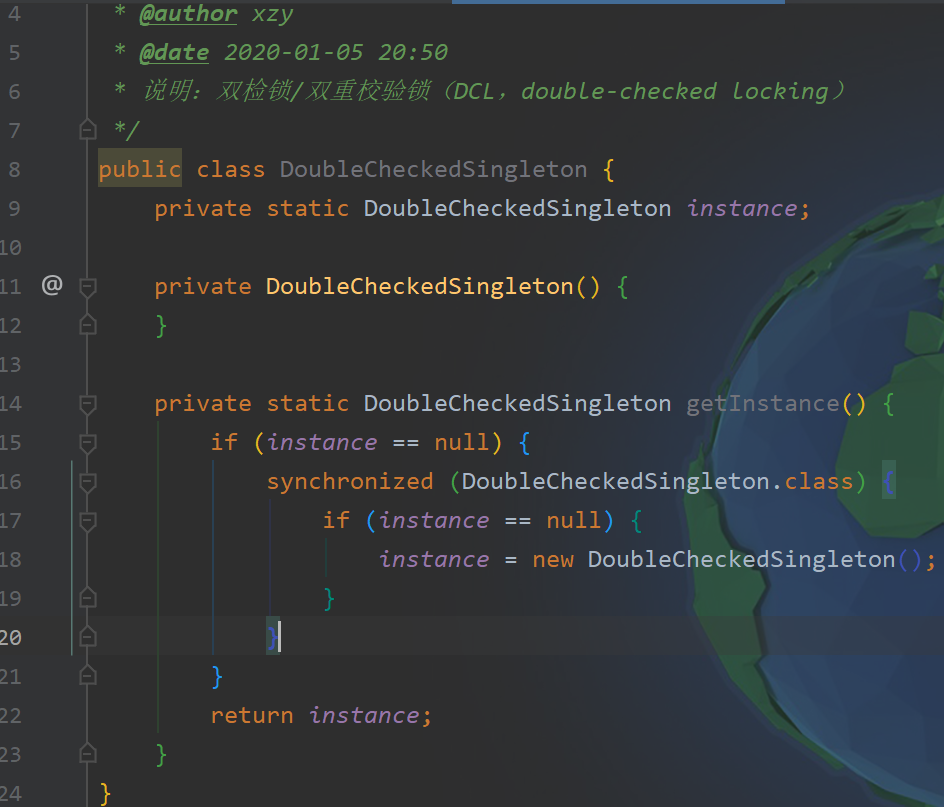
**实例创建时机**：首次尝试获取实例时

**是否线程安全**：是

**实现难度**：易

**描述**：这种方式很好的支持了懒加载，并且能够在多线程中很好的工作，但是，效率偏低，因为只有创建实例的时候才需要解决线程安全问题，后续获取创建好的实例其实不需要枷锁。

## 双检锁/双重校验锁（DCL,double-checked locking）



**实例创建时机**：首次尝试获取实例时

**是否线程安全**：是

**实现难度**：较复杂

**描述**：这种方式其实是对线程安全的懒汉模式的一种改进，即解决了线程安全问题，又保证了系统效率。

getInstance()方法的执行顺序为：

1. 判断单例类是否已经实例化
2. 如果已经实例化则返回实例
3. 如果没有实例化则获取锁，获取锁以后再次判断单例类是否已经实例，如果没有实例化则创建实例，否则返回实例。

## 饿汉式



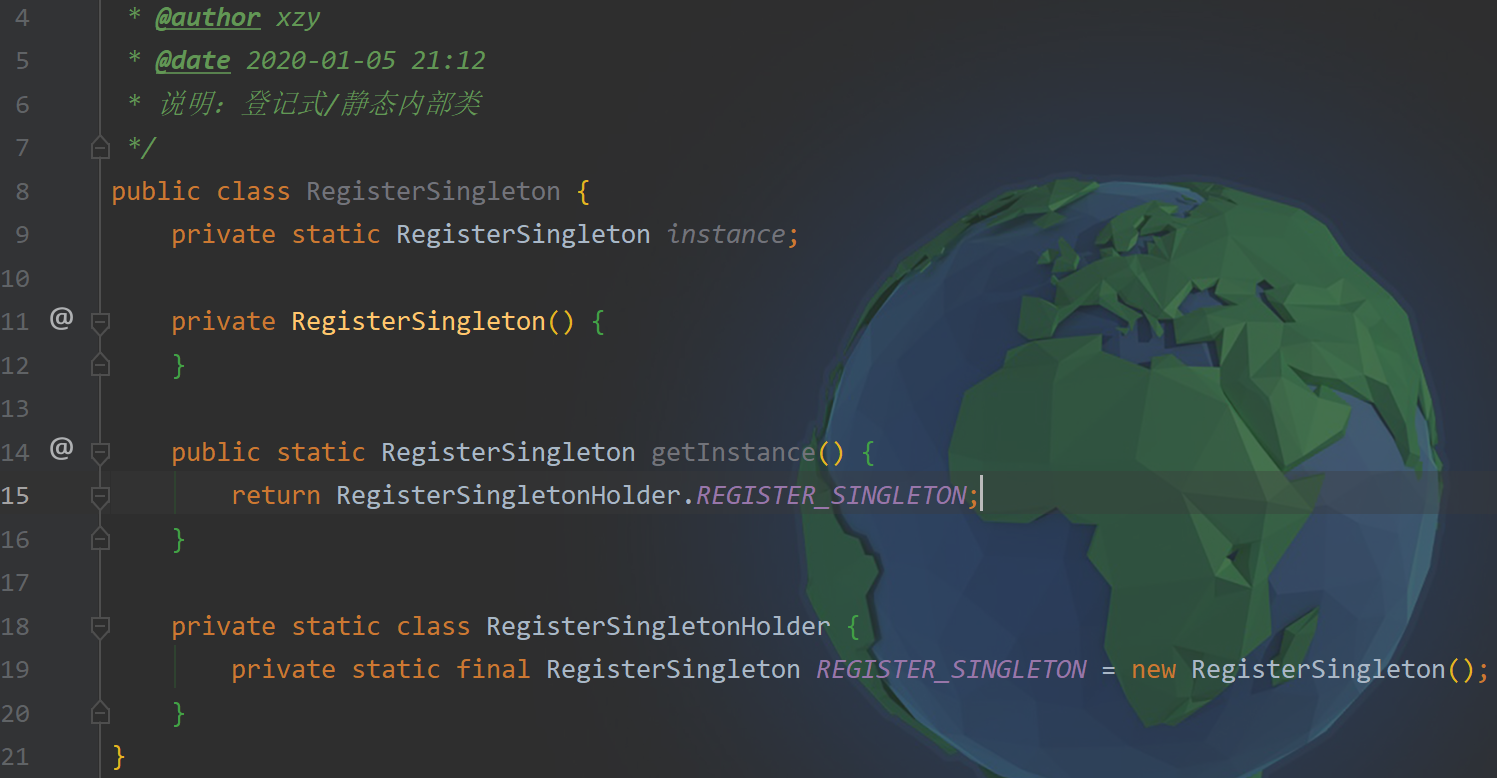
**实例创建时机**：在类被加载的时候就创建实例

**是否线程安全**：是

**实现难度**：易

**描述**：这种方式比较常用，但容易产生垃圾对象。因为实例在类加载时就创建完成，所以避免了线程安全问题，但是可能会造成内存的浪费。

## 登记式/静态内部类



**实例创建时机**：首次尝试获取实例时

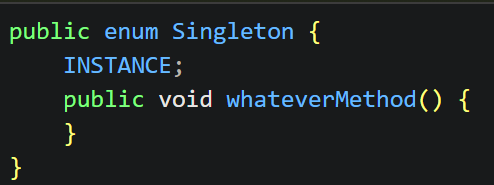
**是否线程安全**：是

**实现难度**：一般

**描述**：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

这种方式同样利用了类加载机制来保证创建实例时只有一个线程，它跟饿汉式不同的是：饿汉式只要单例类被装载了，那么实例就会被创建（没有达到懒加载效果），而这种方式是单例类被装载了，实例也不一定创建。因为SingletonHolder 类没有被主动使用，只有调用 getInstance()方法时，才会显式装载SingletonHolder类，从而创建单例类的实例。

## 枚举



是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种实现方式还没有被广泛采用，但这是实现单例模式的最佳方法。它更简洁，自动支持序列化机制，绝对防止多次实例化。

这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还自动支持序列化机制，防止反序列化重新创建新的对象，绝对防止多次实例化。不过，由于 JDK1.5 之后才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，也很少用。

不能通过 reflection attack 来调用私有构造方法。

# 单例模式的适用场景

单例模式适合在具有以下特点的场景中使用。

1. 在应用场景中，某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等。
2. 当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。
3. 当某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

简单的说，当您想要控制实例的数量，节省系统资源的时候，可以考虑使用单例模式。

# 单例模式的优点

1. 由于类的实例只有一个，内存的开销将被降低。
2. 由于实例由单例类创建，避免了频繁创建、销毁实例产生的开销。
3. 避免多个实例对系统资源的多重占用（如文件操作）。

# 单例模式的缺点

1、由于单利模式中没有抽象层，因此单例类的扩展有很大的困难，违背了依赖倒置原则。

2、单例类的职责过重，在一定程度上违背了“单一职责原则”。

# 单例模式的拓展

单例模式可以拓展为有限多例模式，即使用某种集合存储若干个实例供客户端存取使用。