**单例模式**

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

**注意：**

* **1、单例类只能有一个实例。**
* **2、单例类必须自己创建自己的唯一实例。**
* **3、单例类必须给所有其他对象提供这一实例。**

**介绍**

**意图：**保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**主要解决：**一个全局使用的类频繁地创建与销毁。

**何时使用：**当您想控制实例数目，节省系统资源的时候。

**如何解决：**判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

**关键代码：**构造函数是私有的。

**应用实例：** 1、一个党只能有一个主席。 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

**优点：** 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。

**缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。**

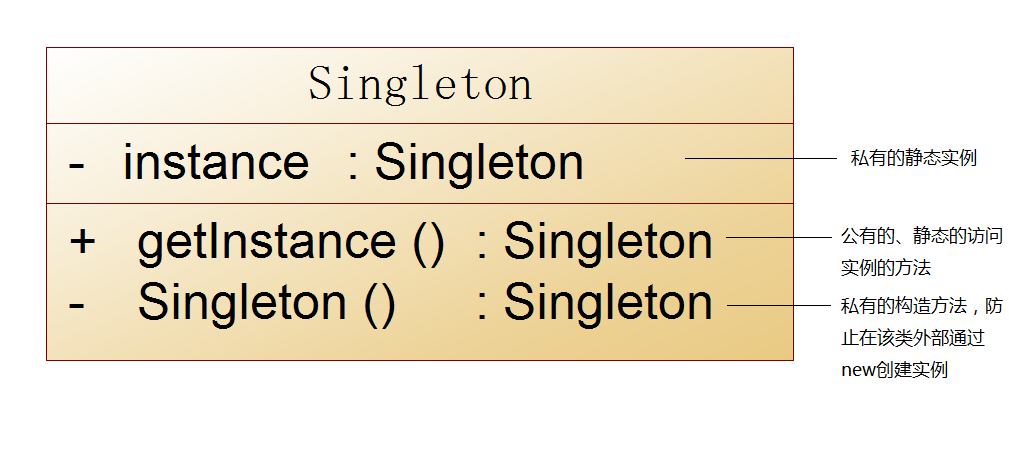
**使用场景：** 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

**实现**

我们将创建一个 *SingleObject* 类。*SingleObject* 类有它的私有构造函数和本身的一个静态实例。

*SingleObject* 类提供了一个静态方法，供外界获取它的静态实例。*SingletonPatternDemo*，我们的演示类使用 *SingleObject* 类来获取 *SingleObject* 对象。



**·** + ：表示public

**·** - ：表示private

**·** #：表示protected（friendly也归入这类）

**步骤 1**

创建一个 Singleton 类。

*SingleObject.java*

public class SingleObject {

//创建 SingleObject 的一个对象

private static SingleObject instance = new SingleObject();

//让构造函数为 private，这样该类就不会被实例化

private SingleObject(){}

//获取唯一可用的对象

public static SingleObject getInstance(){

return instance;

}

public void showMessage(){

System.out.println("Hello World!");

}

}

**步骤 2**

从 singleton 类获取唯一的对象。

*SingletonPatternDemo.java*

public class SingletonPatternDemo {

public static void main(String[] args) {

//不合法的构造函数

//编译时错误：构造函数 SingleObject() 是不可见的

//SingleObject object = new SingleObject();

//获取唯一可用的对象

SingleObject object = SingleObject.getInstance();

//显示消息

object.showMessage();

}

}

**步骤 3**

验证输出。

Hello World!

Java中单例(Singleton)模式是一种广泛使用的设计模式。单例模式的主要作用是保证在Java程序中，某个类只有一个实例存在。一些管理器和控制器常被设计成单例模式。

       单例模式有很多好处，它能够避免实例对象的重复创建，不仅可以减少每次创建对象的时间开销，还可以节约内存空间；能够避免由于操作多个实例导致的逻辑错误。如果一个对象有可能贯穿整个应用程序，而且起到了全局统一管理控制的作用，那么单例模式也许是一个值得考虑的选择。

       单例模式有很多种写法，大部分写法都或多或少有一些不足。下面将分别对这几种写法进行介绍。

1. **饿汉模式**

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **class** Singleton{
2. **private** **static** Singleton instance = **new** Singleton();
3. **private** Singleton(){}
4. **public** **static** Singleton getInstance (){
5. **return** instance;
6. }
7. }

**是否 Lazy 初始化：**否

**是否多线程安全：**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。  
**优点：没有加锁，执行效率会提高。  
缺点：类加载时就初始化，浪费内存。**  
它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

       从代码中我们看到，类的构造函数定义为private的，**保证其他类不能实例化此类，然后提供了一个静态实例并返回给调用者**。饿汉模式是最简单的一种实现方式，饿汉模式在类加载的时候就对实例进行创建，实例在整个程序周期都存在。它的好处是只在类加载的时候创建一次实例，不会存在多个线程创建多个实例的情况，避免了多线程同步的问题。**它的缺点也很明显，即使这个单例没有用到也会被创建，而且在类加载之后就被创建，内存就被浪费了。**

       这种实现方式适合单例占用内存比较小，在初始化时就会被用到的情况。但是，如果单例占用的内存比较大，或单例只是在某个特定场景下才会用到，使用饿汉模式就不合适了，这时候就需要用到懒汉模式进行延迟加载。

**2、懒汉模式**

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **class** Singleton{
2. **private** **static** Singleton instance = **null**;
3. **private** Singleton(){}
4. **public** **static** Singleton getInstance (){
5. **if**(**null** == instance){
6. instance = **new** Singleton();
7. }
8. **return** instance;
9. }
10. }

**是否 Lazy 初始化：**是

**是否多线程安全：**否

**实现难度：**易

**描述：**这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。**因为没有加锁 synchronized，所以严格意义上它并不算单例模式。**  
这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

 懒汉模式中单例是在需要的时候才去创建的，如果单例已经创建，再次调用获取接口将不会重新创建新的对象，而是直接返回之前创建的对象。如果某个单例使用的次数少，并且创建单例消耗的资源较多，那么就需要实现单例的按需创建，这个时候使用懒汉模式就是一个不错的选择。但是这里的懒汉模式并没有考虑线程安全问题，在多个线程可能会并发调用它的getInstance()方法，导致创建多个实例，因此需要加锁解决线程同步问题，实现如下。

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **class** Singleton{
2. **private** **static** Singleton instance = **null**;
3. **private** Singleton(){}
4. **public** **static** **synchronized** Singleton getInstance (){
5. **if**(**null** == instance){
6. instance = **new** Singleton();
7. }
8. **return** instance;
9. }
10. }

**是否 Lazy 初始化：**是

**是否多线程安全：**是

**实现难度：**易

**描述：**这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，**99% 情况下不需要同步。**  
**优点：第一次调用才初始化，避免内存浪费。  
缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。**  
getInstance () 的性能对应用程序不是很关键（该方法使用不太频繁）。

**3、双重校验锁**

       加锁的懒汉模式看起来即解决了线程并发问题，又实现了延迟加载，然而它存在着性能问题，依然不够完美。synchronized修饰的同步方法比一般方法要慢很多，如果多次调用getInstance()，累积的性能损耗就比较大了。**因此就有了双重校验锁，先看下它的实现代码。**

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **class** Singleton {
2. **private** **static** Singleton instance = **null**;
3. **private** Singleton(){}
4. **public** **static** Singleton getInstance() {
5. **if** (instance == **null**) {
6. **synchronized** (Singleton.**class**) {
7. **if** (instance == **null**) {//2
8. instance = **new** Singleton();
9. }
10. }
11. }
12. **return** instance;
13. }
14. }

       可以看到上面在同步代码块外多了一层instance为空的判断。由于单例对象只需要创建一次，如果后面再次调用getInstance()只需要直接返回单例对象。因此，大部分情况下，调用getInstance()都不会执行到同步代码块，从而提高了程序性能。不过还需要考虑一种情况，假如两个线程A、B，A执行了if (instance == null)语句，它会认为单例对象没有创建，此时线程切到B也执行了同样的语句，B也认为单例对象没有创建，然后**两个线程依次执行同步代码块，并分别创建了一个单例对象。**为了解决这个问题，还需要在同步代码块中增加if (instance == null)语句，也就是上面看到的代码2。

       我们看到双重校验锁即实现了延迟加载，又解决了线程并发问题，同时还解决了执行效率问题，是否真的就万无一失了呢？

       这里要提到Java中的指令重排优化**（就是前面博客在讲volitile关键字提及到的有序性）**。所谓指令重排优化是指在不改变原语义的情况下，通过调整指令的执行顺序让程序运行的更快。JVM中并没有规定编译器优化相关的内容，也就是说JVM可以自由的进行指令重排序的优化。

       这个问题的关键就在于由于指令重排优化的存在，**导致初始化Singleton和将对象地址赋给instance字段的顺序是不确定的。**在某个线程创建单例对象时，在构造方法被调用之前，就为该对象分配了内存空间并将对象的字段设置为默认值。此时就可以将1.分配的内存地址赋值给instance字段了，然而该2.对象可能还没有初始化**（即2还没有执行）**。若紧接着另外一个线程来调用getInstance，3.取到的就是**状态不正确的对象**，程序就会出错。（这就是多线程本身存在的有序性问题。）

       以上就是双重校验锁会失效的原因，不过还好在JDK1.5及之后版本**增加了volatile关键字**。volatile的一个语义是禁止指令重排序优化，也就**保证了instance变量被赋值的时候对象已经是初始化过的(当然还有可见性，把它在工作内存赋值过后立刻刷到主存)**，从而避免了上面说到的问题。代码如下：

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **class** Singleton {
2. **private** **static** **volatile** Singleton instance = **null**;
3. **private** Singleton(){}
4. **public** **static** Singleton getInstance() {
5. **if** (instance == **null**) {
6. **synchronized** (Singleton.**class**) {
7. **if** (instance == **null**) {
8. instance = **new** Singleton();
9. }
10. }
11. }
12. **return** instance;
13. }
14. }

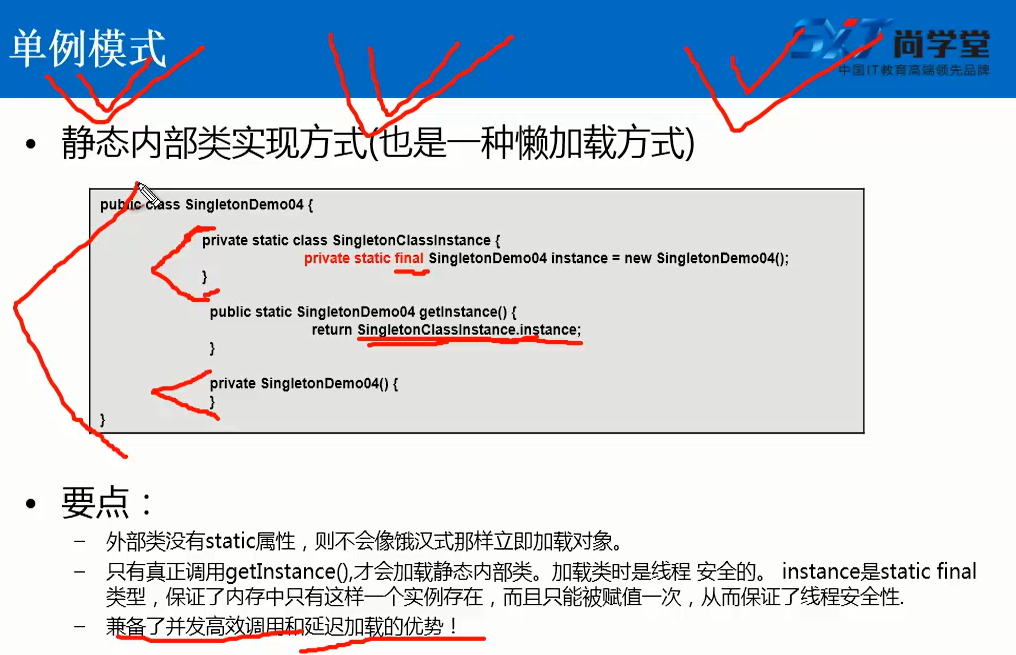
**JDK 版本：**JDK1.5 起

**是否 Lazy 初始化：**是

**是否多线程安全：**是

**实现难度：**较复杂

**描述：**这种方式采用双锁机制，安全且在多线程情况下能保持高性能。  
getInstance() 的性能对应用程序很关键。



### /静态内部类

**是否 Lazy 初始化：**是

**是否多线程安全：**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 classloder 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，**那么 instance 就会被实例化**（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，**instance 不一定被初始化。**因为 SingletonHolder 类没有被主动使用，只有通过显式调用 getInstance 方法时，才会显式装载 SingletonHolder 类，**从而实例化 instance。**想象一下，**如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，**因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

**代码实例：**

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

**5、枚举（面试还是写3 4种最好了，这种面试可以提一下）**

       再来看本文要介绍的最后一种实现方式：枚举。

**[java]** [view plain](https://blog.csdn.net/goodlixueyong/article/details/51935526) [copy](https://blog.csdn.net/goodlixueyong/article/details/51935526)

1. **public** **enum** Singleton{
2. instance;
3. **public** **void** whateverMethod(){}
4. }

       上面提到的四种实现单例的方式都有共同的缺点：

1）需要额外的工作来实现序列化，否则每次反序列化一个序列化的对象时都会创建一个新的实例。

2）可以使用反射强行调用私有构造器（如果要避免这种情况，可以修改构造器，让它在创建第二个实例的时候抛异常）。

       而枚举类很好的解决了这两个问题，使用枚举除了线程安全和防止反射调用构造器之外，还提供了自动序列化机制，防止反序列化的时候创建新的对象。因此，《Effective Java》作者推荐使用的方法。不过，在实际工作中，很少看见有人这么写。