# 参考文档：https://www.cnblogs.com/ngy0217/p/9006716.html

# [java单例模式几种实现方式](https://www.cnblogs.com/ngy0217/p/9006716.html)

# **单例模式的五种实现方式**

1. 饿汉式(线程安全，调用效率高，但是不能延时加载)：

public class ImageLoader{

     private static ImageLoader instance = new ImageLoader;

     private ImageLoader(){}

     public static ImageLoader getInstance(){

          return instance;

      }

}

一上来就把单例对象创建出来了，要用的时候直接返回即可，这种可以说是单例模式中最简单的一种实现方式。但是问题也比较明显。单例在还没有使用到的时候，初始化就已经完成了。也就是说，如果程序从头到位都没用使用这个单例的话，单例的对象还是会创建。这就造成了不必要的资源浪费。所以不推荐这种实现方式。

2.懒汉式(线程安全，调用效率不高，但是能延时加载)：

public class SingletonDemo2 {

//类初始化时，不初始化这个对象(延时加载，真正用的时候再创建)

private static SingletonDemo2 instance;

//构造器私有化

private SingletonDemo2(){}

//方法同步，调用效率低

public static synchronized SingletonDemo2 getInstance(){

if(instance==null){

instance=new SingletonDemo2();

}

return instance;

}

}

3.Double CheckLock实现单例：DCL也就是双重锁判断机制（由于JVM底层模型原因，偶尔会出问题，不建议使用）：

1 public class SingletonDemo5 {

2 private volatile static SingletonDemo5 SingletonDemo5;

3

4 private SingletonDemo5() {

5 }

6

7 public static SingletonDemo5 newInstance() {

8 if (SingletonDemo5 == null) {

9 synchronized (SingletonDemo5.class) {

10 if (SingletonDemo5 == null) {

11 SingletonDemo5 = new SingletonDemo5();

12 }

13 }

14 }

15 return SingletonDemo5;

16 }

17 }

4.静态内部类实现模式（线程安全，调用效率高，可以延时加载）

1 public class SingletonDemo3 {

2 3 private static class SingletonClassInstance{

4 private static final SingletonDemo3 instance=new SingletonDemo3(); 5 }

6

7 private SingletonDemo3(){}

8

9 public static SingletonDemo3 getInstance(){

10 return SingletonClassInstance.instance;

11 }

12

13 }

5.枚举类（线程安全，调用效率高，不能延时加载，可以天然的防止反射和反序列化调用）

1 public enum SingletonDemo4 {

2

3 //枚举元素本身就是单例

4 INSTANCE;

5

6 //添加自己需要的操作

7 public void singletonOperation(){

8 }

9 }

如何选用：

-单例对象 占用资源少，不需要延时加载，枚举 好于 饿汉

-单例对象 占用资源多，需要延时加载，静态内部类 好于 懒汉式