# 单例设计模式

目录

[单例设计模式 1](#_Toc22835655)

[**一、 懒汉（线程不安全）** 1](#_Toc22835656)

[**二、 懒汉（线程安全）** 1](#_Toc22835657)

[**三、 饿汉** 2](#_Toc22835658)

[**四、 饿汉（变种）** 2](#_Toc22835659)

[**五、 静态内部类** 3](#_Toc22835660)

[**六、 枚举** 3](#_Toc22835661)

[**七、 双重校验锁** 3](#_Toc22835662)

[**八、 总结** 4](#_Toc22835663)

1. **懒汉（线程不安全）**

public class Singleton {

private static Singleton instance;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。

1. **懒汉（线程安全）**

public class Singleton {

private static Singleton instance;

private Singleton() {

}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

1. **饿汉**

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton() {

}

public static Singleton getInstance() {

return instance;

}

}

这种方式基于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。而且不能延迟加载，某种意义上会增加系统的负载。

1. **饿汉（变种）**

public class Singleton {

private static final Singleton instance;

static {

instance = new Singleton();

}

private Singleton() {

}

public static Singleton getInstance() {

return instance;

}

}

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

1. **静态内部类**

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton() {

}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要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，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保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，这种方式相比第三和第四种方式就显得很合理。

1. **枚举**

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

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，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

1. **双重校验锁**

public class Singleton {

private volatile static Singleton singleton;

private Singleton() {

}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

这个是第二种方式的升级版，俗称双重检查锁定，在JDK1.5之后，双重检查锁定才能够正常达到单例效果。

1. **总结**

有两个问题需要注意：

     1、如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些servlet容器对每个servlet使用完全不同的类  装载器，这样的话如果有两个servlet访问一个单例类，它们就都会有各自的实例。

     2、如果Singleton实现了java.io.Serializable接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是：

private static Class getClass(String classname)

throws ClassNotFoundException {

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

if (classLoader == null)

classLoader = Singleton.class.getClassLoader();

return (classLoader.loadClass(classname));

}

 对第二个问题修复的办法是：

public class Singleton implements java.io.Serializable {

public static Singleton INSTANCE = new Singleton();

protected Singleton() {

}

private Object readResolve() {

return INSTANCE;

}

}

第三种和第五种方式，简单易懂，而且在JVM层实现了线程安全（如果不是多个类加载器环境），一般的情况下，我会使用第三种方式，只有在要明确实现lazy loading效果时才会使用第五种方式，另外，如果涉及到反序列化创建对象时我会试着使用枚举的方式来实现单例，不过，我一直会保证我的程序是线程安全的，而且我永远不会使用第一种和第二种方式，如果有其他特殊的需求，我可能会使用第七种方式，毕竟，JDK1.5已经没有双重检查锁定的问题了。

不过一般来说，第一种不算单例，第四种和第三种就是一种，如果算的话，第五种也可以分开写了。所以说，一般单例都是五种写法。懒汉，恶汉，双重校验锁，枚举和静态内部类。