# [Java单例模式(Singleton)以及实现](http://www.cnblogs.com/cielosun/p/6582333.html)

一. 什么是单例模式

因程序需要，有时我们只需要某个类同时保留一个对象，不希望有更多对象，此时，我们则应考虑单例模式的设计。

二. 单例模式的特点

1. 单例模式只能有一个实例。

2. 单例类必须创建自己的唯一实例。

3. 单例类必须向其他对象提供这一实例。

三. 单例模式VS静态类

在知道了什么是单例模式后，我想你一定会想到静态类，“既然只使用一个对象，为何不干脆使用静态类？”，这里我会将单例模式和静态类进行一个比较。

1. 单例可以继承和被继承，方法可以被override，而静态方法不可以。

2. 静态方法中产生的对象会在执行后被释放，进而被GC清理，不会一直存在于内存中。

3. 静态类会在第一次运行时初始化，单例模式可以有其他的选择，即可以延迟加载。

4. 基于2， 3条，由于单例对象往往存在于DAO层（例如sessionFactory），如果反复的初始化和释放，则会占用很多资源，而使用单例模式将其常驻于内存可以更加节约资源。

5. 静态方法有更高的访问效率。

6. 单例模式很容易被测试。

几个关于静态类的误解：

误解一：静态方法常驻内存而实例方法不是。

实际上，特殊编写的实例方法可以常驻内存，而静态方法需要不断初始化和释放。

误解二：静态方法在堆(heap)上，实例方法在栈(stack)上。

实际上，都是加载到特殊的不可写的代码内存区域中。

静态类和单例模式情景的选择：

情景一：不需要维持任何状态，仅仅用于全局访问，此时更适合使用静态类。

情景二：需要维持一些特定的状态，此时更适合使用单例模式。

四. 单例模式的实现

1. 懒汉模式

[复制代码](javascript:void(0);)

public class SingletonDemo {

private static SingletonDemo instance;

private SingletonDemo(){

}

public static SingletonDemo getInstance(){

if(instance==null){

instance=new SingletonDemo();

}

return instance;

}

}

[复制代码](javascript:void(0);)

如上，通过提供一个静态的对象instance，利用private权限的构造方法和getInstance()方法来给予访问者一个单例。

缺点是，没有考虑到线程安全，可能存在多个访问者同时访问，并同时构造了多个对象的问题。之所以叫做懒汉模式，主要是因为此种方法可以非常明显的lazy loading。

针对懒汉模式线程不安全的问题，我们自然想到了，在getInstance()方法前加锁，于是就有了第二种实现。

2. 线程安全的懒汉模式

[复制代码](javascript:void(0);)

public class SingletonDemo {

private static SingletonDemo instance;

private SingletonDemo(){

}

public static synchronized SingletonDemo getInstance(){

if(instance==null){

instance=new SingletonDemo();

}

return instance;

}

}

[复制代码](javascript:void(0);)

然而并发其实是一种特殊情况，大多时候这个锁占用的额外资源都浪费了，这种打补丁方式写出来的结构效率很低。

3. 饿汉模式

[复制代码](javascript:void(0);)

public class SingletonDemo {

private static SingletonDemo instance=new SingletonDemo();

private SingletonDemo(){

}

public static SingletonDemo getInstance(){

return instance;

}

}

[复制代码](javascript:void(0);)

直接在运行这个类的时候进行一次loading，之后直接访问。显然，这种方法没有起到lazy loading的效果，考虑到前面提到的和静态类的对比，这种方法只比静态类多了一个内存常驻而已。

4. 静态类内部加载

[复制代码](javascript:void(0);)

public class SingletonDemo {

private static class SingletonHolder{

private static SingletonDemo instance=new SingletonDemo();

}

private SingletonDemo(){

System.out.println("Singleton has loaded");

}

public static SingletonDemo getInstance(){

return SingletonHolder.instance;

}

}

[复制代码](javascript:void(0);)

使用内部类的好处是，静态内部类不会在单例加载时就加载，而是在调用getInstance()方法时才进行加载，达到了类似懒汉模式的效果，而这种方法又是线程安全的。

5. 枚举方法

enum SingletonDemo{

INSTANCE;

public void otherMethods(){

System.out.println("Something");

}

}

Effective Java作者Josh Bloch 提倡的方式，在我看来简直是来自神的写法。解决了以下三个问题：

(1)自由序列化。

(2)保证只有一个实例。

(3)线程安全。

如果我们想调用它的方法时，仅需要以下操作：

public class Hello {

public static void main(String[] args){

SingletonDemo.INSTANCE.otherMethods();

}

}

这种充满美感的代码真的已经终结了其他一切实现方法了。

6. 双重校验锁法

[复制代码](javascript:void(0);)

public class SingletonDemo {

private volatile static SingletonDemo instance;

private SingletonDemo(){

System.out.println("Singleton has loaded");

}

public static SingletonDemo getInstance(){

if(instance==null){

synchronized (SingletonDemo.class){

if(instance==null){

instance=new SingletonDemo();

}

}

}

return instance;

}

}

[复制代码](javascript:void(0);)

接下来我解释一下在并发时，双重校验锁法会有怎样的情景：

STEP 1. 线程A访问getInstance()方法，因为单例还没有实例化，所以进入了锁定块。

STEP 2. 线程B访问getInstance()方法，因为单例还没有实例化，得以访问接下来代码块，而接下来代码块已经被线程1锁定。

STEP 3. 线程A进入下一判断，因为单例还没有实例化，所以进行单例实例化，成功实例化后退出代码块，解除锁定。

STEP 4. 线程B进入接下来代码块，锁定线程，进入下一判断，因为已经实例化，退出代码块，解除锁定。

STEP 5. 线程A获取到了单例实例并返回，线程B没有获取到单例并返回Null。

理论上双重校验锁法是线程安全的，并且，这种方法实现了lazyloading。