单例设计模式Singleton

# 单例Singleton设计模式

## 简单介绍

**单例设计模式(Singleton)**算是设计模式中最简单的设计模式，使用的范围不是很广，但是一般都会用到，比如线程池对象等。

## 单例设计模式的思想

**需求分析**：如果有一个类不想让外界实例化，可以创建将所有的构造方法私有化(private修饰)，，这样外界就不可对该类进行实例化了。对于任何一个类，默认有1个空参数的公共的构造方法，将此覆盖且私有化，外界就访问不了。(注意：将构造方法私有化，该类也不能被继承，因为子类实例化的时候必须需要调用父类的构造方法，所有可以通过final修饰和构造方法私有化两种方法禁止被继承)

## 实现单例设计的步骤：

### 构造方法需要私有化：private修饰。

### 在类内部声明一个静态的类引用；对象的创建可以**静态创建或延迟加载创建**，必须保证只能创建一次；

### 对外提供public的静态static方法，一般为getInstance方法，返回该单例对象。

## 实现单例设计的主要是三类方法

延迟加载(懒汉式)(**最重要**)、饿汉式、通过枚举类实现。

# 延迟加载(懒汉式)：双重检查判断加锁机制

不加锁的懒汉式(不安全)==>加锁一次判断懒汉式(多数是没有必要的同步，效率低)==>双重检查判断加锁

//**懒汉式单例模式**：安全且高效

**public** **class** Singleton {

**private** **static** Singleton *instance*;//注意必须是静态的

**private** **static** **byte**[] *lock* = **new** **byte**[1];

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){

// synchronized(**Singleton.class**){

//可以利用Singleton.class作为锁对象，也可以自定义1个1字节的字节数组作为锁对象；也可以利用可重入锁ReentrantLock

**synchronized**(*lock*){

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

或者通过**ReentrantLock加锁**：

// 或者使用ReentrantLock锁

**public** **class** Singleton {

private Singleton(){}

private static Singleton instance = null;

private static ReentrantLock lock = new ReentrantLock(false);

public static Singleton getInstance(){

if(instance == null){//第一次判断的目的：防止执行获取锁的操作

**lock.lock();**

if(instance == null){//第二次判断的目的：再次判断instance是否实例化

instance = new Singleton();

}

**lock.unlock();**

}

return instance;

}

}

# 饿汉式：不存在并发安全问题

### //饿汉式单例模式

**public** **class** Singleton {

**private** **static** Singleton *instance* = **new** Singleton();

**public** **static** Singleton getInstance(){

**return** *instance*;

}

}

### 等价方式，静态实例在静态代码块中创建：

public class Singleton {

private Singleton(){}

private static Singleton instance = null;

**static{**

**instance = new Singleton();**

**}**

public static Singleton getInstance(){

return instance;

}

}

### 利用**静态内部类**方式

public class Singleton {

private static class SingletonHolder { //静态内部类

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，只有显式通过调用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，我想让它延迟加载，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。

# 如何选择饿汉式还是懒汉式？

下面对单件模式的懒汉式与饿汉式进行简单介绍：

1、饿汉式：在**程序启动或单件模式类被加载**的时候，单件模式实例就已经被创建。

2、懒汉式：当程序**第一次访问单件模式实例**时才进行创建。

如何选择：如果单件模式实例在系统中**经常会被用到**，饿汉式是一个不错的选择。

反之如果单件模式在系统中会**很少用到或者几乎不会用到**，那么懒汉式是一个不错的选择。

1. **利用枚举类Enum实现单例模式**

使用Enum实现**单例模式**是最简单的方法。

public enum SingletonByEnum {

**INSTANCE;//**枚举类型定义单例十分简单，只需要定义一个枚举元素就可以

//其他方法

private String name;

public void setName(String name){

this.name = name;

}

public String getName(){

return name;

}

}

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**，在深度分析Java的枚举类型----枚举的线程安全性及序列化问题中有详细介绍枚举的线程安全问题和序列化问题，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。

# 懒汉式：线程不安全不高效的示例

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

**instance = new Singleton();**

}

return instance;

}

}

懒汉：线程安全，但不高效：

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**。