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**static修饰的getInstance静态方法**，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。

# 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：

## 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。这种方式不存在多线程安全问题。

## 懒汉式(延迟加载)：当程序第一次访问单件模式实例时才进行创建。(重点)

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。

# 饿汉式

**饿汉式本身就是线程安全的**，饿汉式在**类创建**的同时就实例化**一个静态对象出来**，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成。

### //饿汉式单例模式

public class Singleton {

private static volatile Singleton instance = new Singleton();

**private 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;

}

}

# 延迟加载(懒汉式)：双重检查判断加锁机制

## 不安全的延迟加载:线程不安全

/\*\*

\* 延迟加载模式，俗称懒汉式

\* 在多线程情况下，存在线程安全问题；只适用于单线程

\*/

public class Singleton {

private static volatile Singleton instance = null;

private Singleton(){}

public static Singleton getInstance(){

**if(instance == null)**

**instance = new Singleton();**

return instance;

}

}

## 一次判断加锁：线程安全但不高效

/\*\*

\* **无论instance有没有被创建都要加锁，加锁是一个十分耗时的操作。**

\*/

public class OneIfSingleton {

//双重判断，这是最基本的方法

private static volatile OneIfSingleton instance = null;

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

private OneIfSingleton(){}

public static OneIfSingleton getInstance(){

**synchronized (lock){**

if(instance == null)

instance = new OneIfSingleton();

}

return instance;

}

}

或者同步方法

public class Singleton {

private static Singleton instance;

private Singleton (){}

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

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

缺点：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**。

## 双重检查延迟加载<重点：分析volatile的作用>：线程安全且高效

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**。

优势：**线程安全的**同时避免了**每次都同步的性能损耗**。

**volatile**关键字的作用：假设没有关键字volatile的情况下，两个线程A、B，都是第一次调用该单例方法，线程A先执行**instance = new Instance()，该构造方法是一个非原子操作，编译后生成多条字节码指令**，由于**JAVA的指令重排序**，可能会先执行instance的赋值操作，该操作实际只是在内存中开辟一片存储对象的区域后直接返回内存的引用，之后instance便不为空了，但是实际的初始化操作却还没有执行，**如果就在此时线程B进入，就会看到一个不为空的但是不完整（没有完成初始化）的Instance对象**，所以**需要加入volatile关键字，禁止指令重排序优化**，从而安全的实现单例。

补充：**volatile两个作用①可见性；②禁止指令重排序**。

**private static volatile DualIfSingleton instance = null;**

**这里的延迟加载不是真正的延迟加载，静态变量instance已经被初始化，只不过是null而已，对于静态内部类实现的单例，才是真正的延迟加载。将单例对象instance放到静态内部类中实现加载。**

/\*

创建一个字节数组，充当锁对象

\*/

public class DualIfSingleton {

//双重判断，这是最基本的方法

**private static volatile DualIfSingleton instance = null;**

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

private DualIfSingleton(){}

public static DualIfSingleton getInstance(){

if(instance == null){

synchronized (lock){

if(instance == null)

instance = new DualIfSingleton();

}

}

return instance;

}

}

利用可重入锁ReentrantLock

/\*

利用ReentrantLock可重入锁

\*/

public class DualIfSingleton {

//双重判断，这是最基本的方法

private static **volatile** DualIfSingleton instance = null;

private static ReentrantLock lock = new ReentrantLock();

private DualIfSingleton(){}

public static DualIfSingleton getInstance(){

if(instance == null){

**lock.lock();**

if(instance == null)

instance = new DualIfSingleton();

**lock.unlock();**

}

return instance;

}

}

# 静态内部类方式(高档)

public class **NestedSingleton** {

private NestedSingleton(){}//构造方法私有化

public static NestedSingleton getInstance(){

return **SingletonHolder.instance**;

}

private **static** class **SingletonHolder**{//静态内部类(单例对象持有者)

private SingletonHolder(){}

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

}

}

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（没有达到真正 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。

这个解决方案被称为Lazy initialization holder class 模式，这个模式综合使用了java的**类级内部类**和**多线程缺省同步锁**的知识，很巧妙的同时实现了**延迟加载和线程安全**。

**懒加载（Lazy initialization）**：因为**内部静态类**是要在有引用了以后才会装载到内存的。所以在你第一次调用getInstance()之前，SingletonHolder是没有被装载进来的，只有在你第一次调用了**getInstance()**之后，里面涉及到了**return SingletonHolder.instance;** 产生了对SingletonHolder的引用，内部静态类的实例才会真正装载。这也就是**懒加载**的意思。

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

//可以在INSTANCE后面定义枚举元素特有的属性与方法

//其他方法，如name属性的set与get

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。

# 类级内部类介绍

1. 什么是类级内部类？

内部类分为**类级内部类**和**对象级内部类**。

简单点说，类级内部类指的是，有**static修饰的成员内部类**。如果没有static修饰的成员式内部类被称为**对象级内部类**。

（2）类级内部类相当于其**外部类的static成分**，它的对象与外部类对象间不存在依赖关系，因此可以直接创建。而对象级内部类的实例，是绑定在外部对象实例中的。

（3）类级内部类中，可以定义静态的方法。**在静态方法中只能引用外部类中的静态成员方法或变量**。

（4）类级内部类相当于**其外部类的成员**，**只有在第一次被使用的时候才会被装载<重要>**。

# 多线程缺省同步锁的知识：

大家都知道，在多线程开发中，为了解决并发问题，主要是通过使用synchronized来加互斥锁进行同步控制，但是在某些情况下，JVM已经隐含的为您执行了同步，这些情况下就不用自己再来进行同步控制了。

这些情况包括：

（1）由**静态初始化器**（在**静态字段上或static{}块中的初始化器**）初始化数据时

（2）访问final字段时；

（3）在创建线程之前创建对象时；

（4）线程可以看见它将要处理的对象时；

解决方案的思路

（1）要想很简单的实现线程安全，可以采用**静态初始化器**的方式，它可以由JVM来保证线程的安全性。比如前面的**饿汉式实现方式**。但是这样一来，不是会浪费一定的空间吗？因为这种实现方式，会在类装载的时候就初始化对象，不管你需不需要。

（2） 如果现在**有一种方法能够让类装载的时候不去初始化对象**，那不就解决问题了？一种可行的方式就是**采用类级内部类**，在这个类级内部类里面去创建对象实例。这样一来，只要不使用到这个类级内部类， 那就**不会创建对象实例**，从而同步实现**延迟加载和线程安全**。

关于 **JVM来保证线程的安全性** 这句话的意思：利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗。

见“**JVM的类加载机制**”。