# 单例模式

## 单例模式介绍：

所谓类的单例模式，就是采取一定的方法保证其在整个的软件系统中，对某个类**只能存在一个实例对象**，并且该类只提供一个取得其对象实例的方法（静态方法）。

比如Hibernate的SessionFactory，它充当存储源的代理，并负责创建Session对象。SessionFactory并不是轻量级的，一般情况下，一个项目通常只需要一个SessionFactory就够，这时就会使用到单例模式。

## 单例设计模式8种方式：

### 1、饿汉式（静态常量）

### 2、饿汉式（静态代码块）

### 3、懒汉式（线程不安全，无同步）：错误

### 4、懒汉式（线程安全，同步方法）：正确，但是不推荐

### 5、懒汉式（线程不安全，同步代码块）：错误

### 6、双重检查

### 7、静态内部类

### 8、枚举

【注】饿汉式指的是在类中先创建对象；懒汉式指的是在获取该对象实例的时候才创建对象。

## 饿汉式（静态常量）：

### 步骤：

1. 构造器私有化（防止外部的类创建该对象）
2. 类的内部创建对象
3. 向外暴露一个静态的公共方法

### 代码：

|  |
| --- |
| **public** **class** Singleton {  // 2.类的内部创建对象  **private** **static** **final** Singleton ***INSTANCE*** = **new** Singleton();    // 1.构造方法私有化  **private** Singleton() {}    // 3.向外暴露一个静态的公共方法  **public** **static** Singleton getInstance() {  **return** ***INSTANCE***;  }  } |

### 优缺点：

1. 优点：这种写法比较简单，就是在类加载的时候就完成实例化，避免了多线程问题你。
2. 缺点：在类加载的时候完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。
3. 这种方式基于classloader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，在单例模式中大多数都是调用getInstance方法，但是导致类装载的原因有很多，因此不能确定有其他的方式导致类加载，这种时候初始化instance就没有达到Lazy Loading的过小。【总的来说就是：我们主动使用 Singleton.getInstance() 方法确实是能够启动类加载，但是我们却无法保证Singleton这个类是否还有其他的类加载方式】
4. 结论：这种单例模式使用，可能造成内存浪费。

## 饿汉式（静态代码块）

这种写法和上面的【饿汉式（静态常量）】基本是一样的，只不过实例化对象的时候是放在静态代码块中的。

### 1、代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *instance*;    **static** {  *instance* = **new** Singleton();  }    **private** Singleton() {}    **public** **static** Singleton getInstance() {  **return** *instance*;  }  } |

## 五、懒汉式（线程不安全，无同步）：

### 1、代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *instance*;    **private** Singleton() {}    **private** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  **return** *instance*;  }  } |

【注】在多线程环境下，代码错误，不能使用

## 六、懒汉式（线程安全，同步方法）：

### 代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *instance*;    **private** Singleton() {}    **public** **static** **synchronized** Singleton getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  **return** *instance*;  }  } |

【注】没有错误，但是不推荐使用，因为synchronized同步方法影响效率

## 七、懒汉式（线程不安全，同步代码块）：

### 1、代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** Singleton *instance*;    **private** Singleton() {}    **public** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (Singleton.**class**) {  *instance* = **new** Singleton();  }  }  **return** *instance*;  }  } |

【注】在多线程环境下，没有起到线程安全的作用，不能使用

## 八、双重检查：

### 1、代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** **volatile** Singleton *instance*;    **private** Singleton() {}    **public** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (Singleton.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  }  }  **return** *instance*;  }  } |

## 九、静态内部类：

### 代码：

|  |
| --- |
| **public** **class** Singleton {  **private** **static** **volatile** Singleton *instance*;    **private** Singleton() {}    **public** **static** Singleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (Singleton.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** Singleton();  }  }  }  **return** *instance*;  }  } |

### 2、优缺点：

1）这种方式采用了类装载的机制来保证初始化实例时只有一个线程。

2）静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要的实例化时调用getInstance()方法，才会装载SingletonHandler类，从而完成Singleton的实例化。

3）类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程的安全，在类进行初始化的时候，别的线程是无法进入的。

## 枚举：

### 代码：

|  |
| --- |
| **public** **enum** Singleton {  ***INSTANCE***;    **public** **void** sayOk() {  System.***out***.println("ok~");  }  } |

### 优缺点：

1. 这借助JDK1.5中添加的枚举来实现单例模式，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。