**先看一下什么是设计模式？**

**设计模式：(Design Pattern)是一套被反复使用，经过分类编目的，代码设计经验的总结。**

**目的：使代码可重用性，使代码更容易被他人理解，保证代码可靠性。**

**单例模式：**

**顾名思义，就是只能有一个，就如世界上没有两片一模一样的树叶，这里说的是，一个类只能有一个实例，整个项目系统都能访问这个实例**

**单例模式分为两大类，**

1. **懒汉模式：实例在第一次使用时创建**
2. **饿汉模式：实例在类装载时创建**

**饿汉模式：**

**第一种：**

**public class** Singleton {  
 *//使用private将构造方法私有化，以防外界通过该构造方法创建多个实例* **private** Singleton(){}  
 *//由于不能使用构造方法创建实例，所以需要在类的内部创建该类的唯一实例  
 //使用static修饰Singleton，在外界可以通过类名调用该实例 类名.成员名* **static** Singleton *singleton*=**new** Singleton();  
  
}

**第二种：**

**public class** Singleton {  
 *//通过使用private封装该实例，则需要get()实现对外界的开放* **private static** Singleton *instance*=**new** Singleton();  
 *//添加static 将该方法变成类所有 通过类名访问* **public static** Singleton getInstance(){  
 **return** *instance*;  
 }  
}

**推荐第二种方法**

**但是有个缺点，由于instance的实例化是在类加载的时候进行的，类加载由ClassLoader来实现，初始化太早，造成资源浪费，但是单例占用的资源较少，所以这种方式也是不错的。**

**类加载的时机**

1. **new一个对象时**
2. **使用反射创建它的实例**
3. **子类被加载时，如果父类还没有被加载，就先加载父类**
4. **JVM启动时执行主类会先被加载**

**懒汉模式：**

**单线程：**

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

**这种方法在单线程是没有问题的**

**如果是多线程的话，就要使用synchronized关键字**

**第一种：**

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

**这种方法虽然避免了可能会出现多个实例的情况，，但是会强制除了第一个线程之外其他的线程都要等待的情况，会对程序的执行效率造成负面影响**

**第二种：双重检查(Double-Check)**

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

**注意其中有两次判断if(singleton==null)这个叫双重检查(Double-Check)**

1. **第一个是为了解决第一种方法的效率问题，只有singleton为null的时候才能进入synchronized的代码段大大减少了几率**
2. **第二个就跟第一种方法一样为了防止可能出现多个实例。**

**但是这也是很小概率，想要更小就要理解原子操作，指令重排**

**原子操作atomic**

**原子操作就是不可分割的操作，在计算机中就是不会因为线程调度被打断的操作**

**比如：赋值m=6;要么赋值成功6，要么失败0，不会变成3或者其他，即使是在线程并发中**

**但是声明并赋值int m=6;就不是一个原子操作**

**对于这个语句，有两个操作：**

1. **声明变量m**
2. **给m赋值**

**这样就会有个中间状态，这样在多线程中，由于线程执行的顺序的不确定性，如果两个线程都使用m，就可能导致不稳定的结果出现**

**指令重排：**

**简单来说，就是计算机为了提高执行效率，做的一些优化，在不影响最终结果的情况下，可能会对一些语句进行顺序调整，**

**public void** test(){  
 **int** a; *// 1* a=6; *// 2* **int** b=6; *// 3* **int** c=a+b; *// 4*}

**比如，上面方法中的一段代码，正常的执行顺序：1234，但是由于指令重排的原因，可能会2134,3412等**

**另外语句3和4可能会被拆分执行，再重排，**

**那现在我们再看一下多线程单例模式第二种方法中，singleton=new Singleton();并非是一个原子操作，事实上在JVM中这句话做了3件事：**

1. **给singleton分配内存**
2. **调用Singleton的构造函数来初始化成员变量，形成实例**
3. **将singleton对象指向分配的内存空间（执行完这步singleton才是非null）**

**这3步在JVM即时编译中存在指令重排的优化，可能会出现，有一个singleton不为null，但是没有完成初始化，会报错。解决方案是给singleton声明上加volatile关键字**

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

**volatile关键字的一个作用就是防止指令重排，把singleton声明为volatile后，对它的操作就会有一个内存屏障，在它的赋值完成之前，就不会调用读操作**

**注意：volatile阻止的不是singleton=new Singleton()这句话内部的指令重排，而是保证了在一个写操作这三步完成之前，不会调用if(singleton==null)**

**再看看其他单例**

**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***;  
 }  
}

**这句话的巧妙在于**

1. **对于内部类SingletonHolder是一个饿汉式的单例实现，在SingletonHolder初始化的时候会有ClassLoader来保证同步，是INSTANCE是一个真单例。**
2. **由于SingletonHolder是一个内部类，只在外部类的Singleton的getInstance()中被使用，所以它被加载的时机也就是在getInstance()方法第一被调用的时候**

**枚举**

**public enum** Singleton {  
 ***INSTANCE***;  
 **public void** fun1(){  
 *//do something...* }  
}

**因为自动化序列机制，保证了线程的绝对安全，简单、高效、安全**