

单例模式

整理自：《java与模式》之单例模式

在阎宏博士的《JAVA与模式》一书中开头是这样描述单例模式的：

作为对象的创建模式，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。

单例模式的结构

单例模式的特点：

- 单例类只能有一个实例。
- 单例类必须自己创建自己的唯一实例。
- 单例类必须给所有其他对象提供这一实例。

饿汉式单例类

```
public class EagerSingleton {  
    private static EagerSingleton instance = new EagerSingleton();  
    /**  
     * 私有默认构造子  
     */  
    private EagerSingleton(){}  
    /**  
     * 静态工厂方法  
     */  
    public static EagerSingleton getInstance(){  
        return instance;  
    }  
}
```

上面的例子中，在这个类被加载时，静态变量instance会被初始化，此时类的私有构造子会被调用。这时候，单例类的唯一实例就被创建出来了。

饿汉式其实是一种比较形象的称谓。既然饿，那么在创建对象实例的时候就比较着急，饿了嘛，于是在装载类的时候就创建对象实例。

```
private static EagerSingleton instance = new EagerSingleton();
```

饿汉式是典型的空间换时间，当类装载的时候就会创建类的实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断，节省了运行时间。

懒汉式单例类

```
public class LazySingleton {  
    private static LazySingleton instance = null;  
    /**  
     * 私有默认构造子  
     */  
    private LazySingleton(){}  
    /**  
     * 静态工厂方法  
     */  
    public static synchronized LazySingleton getInstance(){  
        if(instance == null){  
            instance = new LazySingleton();  
        }  
        return instance;  
    }  
}
```

上面的懒汉式单例类实现里对静态工厂方法使用了同步化，以处理多线程环境。

懒汉式其实是一种比较形象的称谓。既然懒，那么在创建对象实例的时候就不着急。会一直等到马上要使用对象实例的时候才会创建，懒人嘛，总是推脱不开的时候才会真正去执行工作，因此在装载对象的时候不创建对象实例。

```
private static LazySingleton instance = null;
```

懒汉式是典型的时间换空间，就是每次获取实例都会进行判断，看是否需要创建实例，浪费判断的时间。当然，如果一直没有人使用的话，那就不会创建实例，则节约内存空间

由于懒汉式的实现是线程安全的，这样会降低整个访问的速度，而且每次都要判断。那么有没有更好的方式实现呢？

双重检查加锁

可以使用“双重检查加锁”的方式来实现，就可以既实现线程安全，又能够使性能不受很大的影响。那么什么是“双重检查加锁”机制呢？

所谓“双重检查加锁”机制，指的是：并不是每次进入getInstance方法都需要同步，而是先不同步，进入方法后，先检查实例是否存在，如果不存在才进行下面的同步块，这是第一重检查，进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

“双重检查加锁”机制的实现会使用关键字 `volatile`，它的意思是：被 `volatile` 修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。

注意：在java1.4及以前版本中，很多JVM对于 `volatile` 关键字的实现的问题，会导致“双重检查加锁”的失败，因此“双重检查加锁”机制只只能用在java5及以上的版本。

```
public class Singleton {
    private volatile static Singleton instance = null;
    private Singleton(){}
    public static Singleton getInstance(){
        //先检查实例是否存在，如果不存在才进入下面的同步块
        if(instance == null){
            //同步块，线程安全的创建实例
            synchronized (Singleton.class) {
                //再次检查实例是否存在，如果不存在才真正的创建实例
                if(instance == null){
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这种实现方式既可以实现线程安全地创建实例，而又不会对性能造成太大的影响。它只是第一次创建实例的时候同步，以后就不需要同步了，从而加快了运行速度。

提示：由于volatile关键字可能会屏蔽掉虚拟机中一些必要的代码优化，所以运行效率并不是很高。因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用“双重检查加锁”机制来实现线程安全的单例，但并不建议大量采用，可以根据情况来选用。

根据上面的分析，常见的两种单例实现方式都存在小小的缺陷，那么有没有一种方案，既能实现延迟加载，又能实现线程安全呢？

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，很巧妙地同时实现了延迟加载和线程安全。

1.相应的基础知识

- 什么是类级内部类？

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

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。

类级内部类中，可以定义静态的方法。在静态方法中只能够引用外部类中的静态成员方法或者成员变量。

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。

- 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。但是在某些情况中，JVM已经隐含地为您执行了同步，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

2. 解决方案的思路

要想很简单地实现线程安全，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，不管你需要不需要。

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，从而同时实现延迟加载和线程安全。

示例代码如下：

```
public class Singleton {

    private Singleton(){}
    /**
     * 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例
     * 没有绑定关系，而且只有被调用到时才会装载，从而实现了延迟加载。
     */
    private static class SingletonHolder{
        /**
         * 静态初始化器，由JVM来保证线程安全
         */
        private static Singleton instance = new Singleton();
    }

    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

当getInstance方法第一次被调用的时候，它第一次读取SingletonHolder.instance，导致SingletonHolder类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建Singleton的实例，由于是静态的域，因此只会在虚拟机装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

单例和枚举

按照《高效Java 第二版》中的说法：单元素的枚举类型已经成为实现Singleton的最佳方法。用枚举来实现单例非常简单，只需要编写一个包含单个元素的枚举类型即可。

```
public enum Singleton {  
    /**  
     * 定义一个枚举的元素，它就代表了Singleton的一个实例。  
     */  
  
    uniqueInstance;  
  
    /**  
     * 单例可以有自己的操作  
     */  
    public void singletonOperation(){  
        //功能处理  
    }  
}
```

使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。