设计模式 - 单例模式

寂然

五重境界

> 第 1 层：刚开始学编程不久，听说过什么是设计模式 ，但不了解
>
> 第 2 层：有很长时间的编程经验了，自己写了很多代码，其中用到了设计模式，但是自己却不知道
>
> 第 3 层：学习过了设计模式，发现自己已经在使用了，并且发现了一些新的模式挺好用的
>
> 第 4 层：阅读了很多别人写的源码和框架，其中看到了设计模式，能够领会设计模式的精妙和带来的好处
>
> 第 5 层：代码写着写着，自己都没有意识到使用了设计模式，并且熟练的写了出来

官方定义

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

举个最常见的例子，Spring 中的 bean 默认都是单例模式，每个bean定义只生成一个对象实例，每次getBean请

求获得的都是此实例

单例模式八种方式

那接下来我们来聊聊单例模式的八种实现方式，如下所示

> 饿汉式(静态常量)
>
> 饿汉式（静态代码块）
>
> 懒汉式(线程不安全)
>
> 懒汉式(线程安全，同步方法)

> 懒汉式(线程安全，同步代码块)
>
> 双重检查
>
> 静态内部类
>
> 枚举方式

饿汉式（静态常量）

```
class Singleton{

    //一：构造器的私有化 防止外部用构造器...
    private Singleton(){}

    //二：类的内部创建对象 final static
    private static final Singleton singleton = new Singleton();

    //三：对外提供公共的静态方法 返回该类唯一的对象实例
    public static Singleton getInstance(){

        return singleton;
    }
}
```

写法分析

优势 简单 避免多线程的同步问题

劣势 没有达到懒加载的效果 内存的浪费

饿汉式（静态代码块）

```
//方式二：静态代码块的方式
class Singleton{

    //构造器私有化
    private Singleton(){}

    //类的内部创建对象
    private static final Singleton singleton;

    static {
        singleton = new Singleton();
    }

    //对外提供公共的静态的方法
    public static Singleton getInstance(){
```

```
        return singleton;
    }
}
```

写法分析

优势：简单，避免了多线程同步问题

劣势：没有实现懒加载的效果 内存的浪费

懒汉式（线程不安全）

```
class Singleton{

    //构造器私有化
    private Singleton(){}

    //类的内部提供对象
    private static Singleton singleton;

    //对外提供公共的静态方法的时候，来判断
    public static Singleton getInstance(){

        if (singleton == null){

            singleton = new Singleton();
        }

        return singleton;
    }

}
```

写法分析

优势：起到了懒加载的效果 不会造成内存浪费

劣势：线程不安全 不推荐这种方式的

懒汉式（同步方法）

```
//加入同步处理  同步方法
class Singleton{

    //构造器私有化
    private Singleton(){}

    //类的内部提供对象
    private static Singleton singleton;
```

```
    //对外提供公共的静态方法的时候，来判断
    public static synchronized Singleton getInstance(){

        if (singleton == null){

            singleton = new Singleton();
        }

        return singleton;
    }

}
```

解决了线程安全问题，但是效率太低

懒汉式（同步代码块）

```
//加入同步处理 - 同步代码块的方式  不推荐的
class Singleton{

    //构造器私有化
    private Singleton(){}

    //类的内部提供对象
    private static Singleton singleton;

    //对外提供公共的静态方法的时候，来判断
    public static Singleton getInstance(){

        if (singleton == null){

            synchronized (Singleton.class){ //失去了意义

                singleton = new Singleton();
            }

        }

        return singleton;
    }

}
```

不推荐的，解决不了线程的安全问题

双重检查机制

```
class Singleton{

    private Singleton(){}

    private static Singleton singleton;

    //加入双重检查机制
    public static Singleton getInstance(){
        if (singleton == null){

            synchronized (Singleton.class){
                if (singleton == null){
                    singleton = new Singleton();
                }
            }
        }

        return singleton;
    }
}
```

写法分析

实际开发中，推荐使用这种方式？ 三点

线程安全

解决了线程安全问题

懒加载

不会造成内存的浪费

效率很高

效率也相对较高

可能出现的问题

我们认为的 new Singleton() 操作

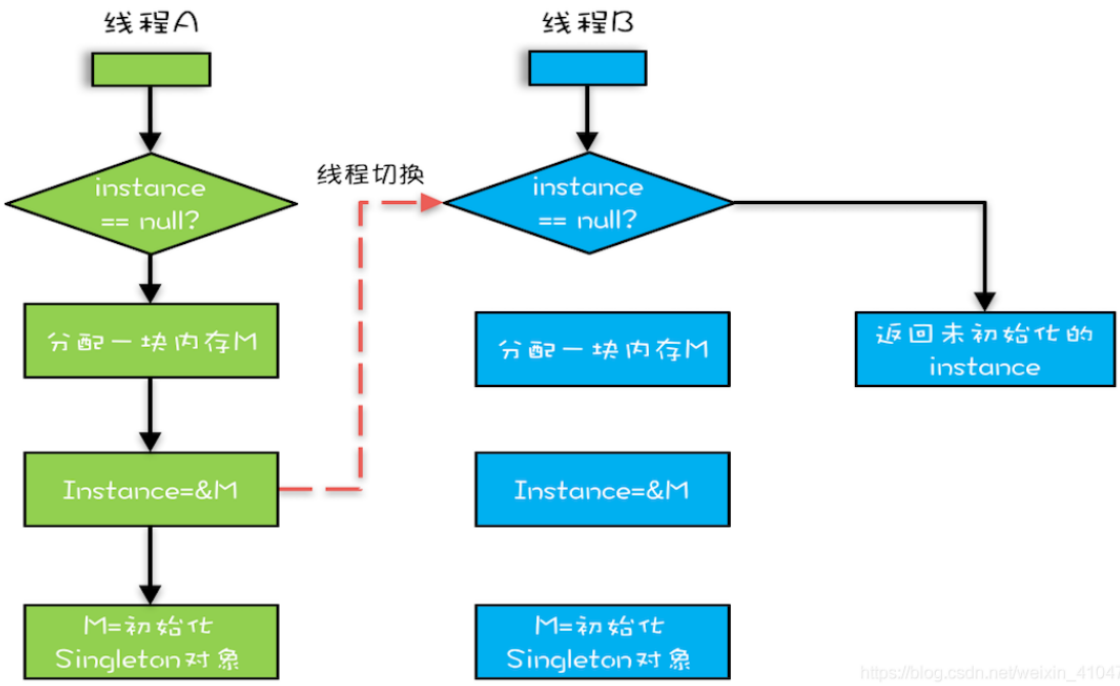
> 1）分配内存地址 M
>
> 2）在内存 M 上初始化Singleton 对象
>
> 3）将M的地址赋值给 instance 对象

JVM编译优化后（**指令重排**）可能的 new Singleton() 操作

> 1）分配内存地址 M
>
> 2）将M的地址赋值给instance变量
>
> 3）在内存M上初始化 Singleton 对象

空指针异常

异常发生过程



解决方式：关键字 Volatile 来禁止指令重排

扩展 - Volatile

轻量级的同步机制 （低配版） 没有保证原子性

三大特性？？

保证可见性

其中一个线程修改了主内存共享变量的值，要写回主内存，并要及时通知其他线程可见

没有保证原子性

没法（不能保证）不可分割，完整，要么同时成功，要么同时失败

禁止指令重排

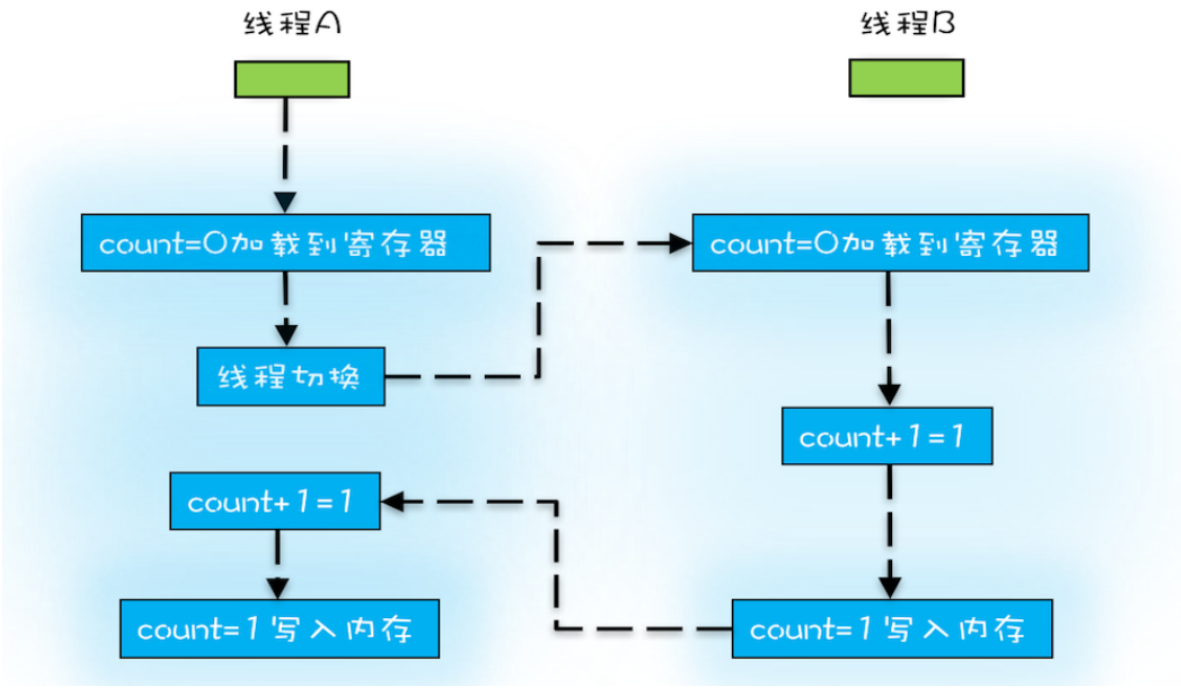
和底层内存屏障相关 避免多线程下出现指令乱序的情况

扩展-线程切换

Java的一条语句对应的cpu指令可能是多条，其中任意一条cpu指令在执行完都可能发生线程切换

count += 1，对应cpu 指令如下：

> 1）将变量count从内存加载到cpu寄存器
>
> 2）寄存器中 +1
>
> 3）将结果写入内存（缓存机制写入的可能是cpu而不是内存）



静态内部类

```
class Singleton{

    private Singleton(){}

    private static class SingletonInstance{

        public static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance(){

        return SingletonInstance.INSTANCE;
    }
}
```

写法分析

不会出现线程安全问题

JVM来帮我们保证了线程的安全性

利用静态内部类的特点，效率也很高，实际开发中推荐使用的

枚举方式

```
enum Singleton{

    INSTANCE; //属性
}
```

写法分析

不仅可以避免线程安全问题 推荐大家使用

注意事项

> 1，单例模式保证了系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，
>
> 使用单例模式可以提高系统性能
>
> 2，当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用 new

单例模式的使用场景

- 对于一些需要频繁创建销毁的对象
- 重量级的对象
- 经常使用到的对象
- 工具类对象
- 数据源，session。。。。

JDK源码分析

JDK中，java.lang.Runtime 就是经典的饿汉式单例模式

```
public class Runtime {
    private static Runtime currentRuntime = new Runtime();

    /**
     * Returns the runtime object associated with the current Java application.
     * Most of the methods of class <code>Runtime</code> are instance
     * methods and must be invoked with respect to the current runtime object.
     *
     * @return  the <code>Runtime</code> object associated with the current
     *          Java application.
     */
    public static Runtime getRuntime() {
        return currentRuntime;
    }

    /** Don't let anyone else instantiate this class */
    private Runtime() {}
```