设计模式java语言实现之单例模式

前言:

> 作者：韩数
>
> Github：https://github.com/hanshuaikang
>
> 时间:2019-01-26
>
> Jdk版本：1.8

定义:

> 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用范围:

> 一个全局使用的类频繁地创建与销毁。当某些如打印机程序只需要一个操作实例的时候(多个实例同时调用一台打印机打印文件可能会出现问题)

优/缺点

> 优点:
>
> 在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。
>
> 避免对资源的多重占用（比如写文件操作)
>
> 缺点:
>
> 没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化

前提引入:

> 略

单例模式是设计模式中较为简单的设计模式之一，同时又在实际的开发过程中广泛使用的一种设计模式，本次，我们将从基础版开始，使用4种不同的方式来实现单例模式。

代码实战:

1.基础版

这种方式实现的单例模式是我们日常使用较多的设计模式，采用了延迟加载来减少系统资源不必要的开支，但如果多个线程同时调用 **getInstance** 方法获取Singleton的实例时，可能会出现问题。

优/缺点:

> 优点:易于实现
>
> 缺点:线程不安全，在多线程下不能正常工作

代码如下:

```
/***
 *
 * @author 韩数
 * 一般单例模式实现，采用延迟加载方式实现
 *
 */

public class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

}
```

2.线程安全基础版:

代码如下:

优/缺点:

> 优点:线程安全，没有加锁，执行效率会高一点
>
> 缺点:容易产生垃圾对象，类加载时就初始化，浪费内存

```
/***
 *
 * @author 韩数
 * 常用的单例模式实现
```

```
 * 线程安全，不足之处，可能会损失一部分性能，非延迟加载
 *
 */


public class Singleton {

    //在类初始化的时候就实例化对象,所以不存在多线程的安全问题
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

3.线程安全加强版:

优/缺点:

> 优点:线程安全，延迟加载
>
> 缺点:必须加锁，效率比较低

代码如下:

```
/**
 *
 * @author 韩数
 * 线程安全方式实现单例模式，缺点是，每次都会调用synchronized的关键字修饰的方法，会损失一定的性能
 *
 */

public class Singleton {


    private static Singleton uniqueInstance;

    private Singleton() {}

    //synchronized修饰该方法，保证每次只有一个线程进入该方法，从而避免产生多个Singleton对象实例。
    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
```

```
    }
}
```

4.线程安全旗舰版:

双检锁/双重校验锁（DCL，即 double-checked locking）：可以在保证安全性的情况下，兼顾高性能

优/缺点:

> 优点:线程安全，延迟加载，执行效率高
>
> 缺点:实现难度比前几种稍显复杂

关于 volatile 关键字，感兴趣的伙伴可以去这个博客看一下，写得非常好

https://www.cnblogs.com/dolphin0520/p/3920373.html

代码如下:

```
/***
 *
 * @author 韩数
 * 线程安全，延迟加载方式实现单例模式
 *
 */

public class Singleton {

    private volatile static Singleton uniqueInstance;

    // 一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：
    //保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是        立即可见的。
    //禁止进行指令重排序。

    private Singleton() {}

    public static Singleton getInstance() {
        //延迟加载
        if (uniqueInstance == null) {
            //加线程锁
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```

总结:

本篇文章较为简单的阐述了单例模式的优缺点，使用场景，以及4种不同的实现方式，当然，现实生活中，单例模式的实现绝非只有本文中的4种，如果对单例模式的其他实现方式感兴趣的话，大家可以去互联网上查询相关的资料。

写在最后:

欢迎大家给小星星，您的星星是我写下去的不竭动力!

源码部分请移步本人Github下载:

Github地址:

Github：https://github.com/hanshuaikang/design-pattern-java

参考资料:

菜鸟教程:http://www.runoob.com/design-pattern/strategy-pattern.html

Head frist of 设计模式