链滴

GOF —— 单例模式

作者：zh847707713

原文链接：https://ld246.com/article/1603876998573

来源网站：链滴

单例模式（Singleton）

单例模式(SIngleton)的目的是为了保证一个进程中，某个类有且仅有一个实例。

因为这个类只有一个实例，因此，该类不能允许 new 的方式创建实例。

- **单例的构造方法必须是 private**，这样就防止了调用方自己创建实例。
- 既然不能通过new 创建实例，只能通过访问静态成员变量的方式了，而且 **成员变量不能定义成public**（调用者可以直接通过Singleton.instance 的方式更改 ），所以需要提供一个 获取静态成员变量的静态方法。

饿汉式

```
/**
 * 饿汉式
 * 类加载到内存后，就实例化一个单例，JVM 保证线程安全
 * 简单实用，推荐使用！
 * 唯一缺点：不管用到与否，类加载时就完成实例化
 *  （话说你不用的，你装在它干啥？）
 */
public class Singleton01 {

    private static final Singleton01 INSTANCE = new Singleton01();

    private Singleton01() {
    }

    public static Singleton01 getInstance(){
        return INSTANCE;
    }

}
```

懒汉式

饿汉式虽然很简洁、而且有 JVM 线程安全，但是 饿汉式，不管用没用到这个实例，都会进行初始化。

为了达到按需初始化的目的，人们对饿汉式进行了更改。

线程不安全的懒汉式

```
/**
 * lazy loading
 * 也称懒汉式
 * 虽然达到了按需初始化的目的，但却带来了线程不安全的问题
 */
public class Singleton02 {

    private static Singleton02 INSTANCE ;

    private Singleton02(){}
```

```
    public static Singleton02 getInstance(){
        if(INSTANCE == null){                // ①
            INSTANCE = new Singleton02();    // ②
        }
        return INSTANCE;
    }

}
```

为啥线程不安全?

假设有两个线程同时调用 getInstance() 方法，当线程1 走到上面代码 ① 处时，判断 INSTANCE 变为空，走到了 if 方法里面，这时候 线程2 刚好在 线程 1 执行 代码 ② 之前 走到了 代码 ①处，并判断 NSTANCE 变量也为空，则会出现 INSTANCE 被赋值两次的情况。

线程安全的懒汉式

既然线程不安全，我们可以通过在静态方法上添加 synchronized 关键字，来进行线程同步

```
/**
 *  lazy loading
 *
 *  也称懒汉式
 *  虽然达到了按需初始化的目的，但却带来了线程不安全的问题
 *  可以通过 synchronized 解决问题，但也带来效率低下
 *
 */
public class Singleton03 {

    private static Singleton03 INSTANCE;

    private Singleton03(){}

    public static synchronized Singleton03 getInstance(){
        if(INSTANCE == null){
            INSTANCE = new Singleton03();
        }
        return INSTANCE;
    }


}
```

虽然解决了线程安全的问题，但是效率比较低下。

因为每次调用 getInstance() 方法的时候，都会加锁

线程不安全的双重检查锁定

```
/**
```

```
 * 双重检查锁定
 */
public class Singleton05 {

    private static Singleton05 INSTANCE;

    private Singleton05(){

    }

    public static Singleton05 getInstance(){
        if(INSTANCE == null){
            synchronized (Singleton05.class){
                if(INSTANCE == null){
                    INSTANCE = new Singleton05();
                }
            }
        }
        return INSTANCE;
    }

}
```

有效的解决了性能的问题，一旦 INSTANCE 被初始化，则后续的调用都不会加锁

但是这个虽然用到了同步，但是依然会有线程安全的问题。

即 cpu 对指令的重排序，会出现 INSTANCE 为 null 的情况。因为 CPU 会对 new Singleton05() 的分配内存和赋值指令排序，会有 分配内存之后被 直接返回的情况出现，所以线程不安全。

线程安全的双重检查锁定

```
/**
 * 双重检查锁定
 */
public class Singleton05 {

    private static volatile Singleton05 INSTANCE;

    private Singleton05(){

    }

    public static Singleton05 getInstance(){
        if(INSTANCE == null){
            synchronized (Singleton05.class){
                if(INSTANCE == null){
                    INSTANCE = new Singleton05();
                }
            }
        }
        return INSTANCE;
    }
```

```
}
```

只需要对 INSTANCE 变量加上 volatile 关键字，该改关键字会阻止 cpu 的指令重排序优化。

静态内部类

```
/**
 *  静态内部类的方式
 *  JVM 保证线程安全
 *  加载外部类时不会加载内部类，这样可以实现懒加载
 */
public class Singleton06 {

    private Singleton06(){
    }

    private static class SingletonHandler{
        private static final Singleton06 INSTANCE = new Singleton06();
    }

    public static Singleton06 getInstance(){
        return SingletonHandler.INSTANCE;
    }

}
```

比较完美的一种方式。

枚举的方式

```
public enum Singleton07 {

    INSTANCE;

    private String name = "world";

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```