链滴

Java 设计模式 (一)- 单例模式

作者：yechuan

原文链接：https://ld246.com/article/1588138998795

来源网站：链滴

Java设计模式(一)-单例模式

什么是单例模式

● 单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建模式，它提供了一种创建对象的最佳方式。

● 这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

实现单例模式的思路

1. **构造方法私有化**

保证系统中只有一个对象的实例，那么我们就需要去防止该对象一不小心被new出来，因此我们需要构造方法私有化

2. **创建静态方法获取实例**

外界不能通过new来获得对象实例了，然而我们需要获取一个实例，那么我们就需要提供一个可供外访问的返回对象实例的静态方法

3. **确保对象实例只有一个**

只对对象进行一次实例化，以后的获取都是获取第一次创建的对象实例

几种单例模式的区别

1. **饿汉模式**

- 饿汉模式是指，我先把对象创建好，等你需要的时候再来拿就可以了

```
public class Singleton1 {
  // 无论该对象是否被使用，均会创建，绝大数情况加浪费性能
   private static Singleton1 singleton = new Singleton_1();
 private static Singleton1() {
     }
 public Singleton1 getSingleton() {
   return singleton;
  }
}
```

● 这种模式是最为简单的，而且天生线程安全，但是有一个缺陷在于有资源浪费的嫌疑，不论你么时候使用，我在类加载的时候就一次性创建了

2. **懒汉模式**

● 因为饿汉模式存在浪费资源的嫌疑，懒汉模式应运而生

● 懒汉模式的意思是，我先不创建类的对象实例，等你需要的时候我再创建

```
public class Singleton2 {
  private static Singleton2 singleton = null;
  private Singleton2() {}

 // 在调用的时候再去创建对象
 // 在并发情况下无法保证项目中只有一个Singleton_2对象，违背单例模式的初衷
 public static Singleton2 getSingleton() {
    if (singleton == null) { // 线程1
        singleton = new Singleton2(); // 线程 2
    }
    return singleton;
 }
}
```

● 该对象实例只有在执行获取方法的时候才会去创建、解决了饿汉模式的资源浪费的缺陷

● 但是上述代码在并发情况下会出现创建多个对象的情况，例如，此时singleton为null，线程1与程2均会通过判断，进行对象实例化，因此线程1与线程2会各自创建一个对象

3. **synchronized同步锁**

● 因为可能出现外界多人同时访问getSingleton()方法，可能会出现因为并发问题导致类被实例化次，我们可以给获取对象方法加上锁**synchronized**来控制该方法在同一时刻只有一个线程能进入，保证该对象只会被实例化一次

```
public class Singleton3 {

 private static Singleton3 singleton = null;

 private Singleton3() {
 }

 // 这是最简单的解决方案，使用synchronized加锁来解决
 // 可以有多个线程进入getSingleton（）方法，但只会有一个线程进入if判断，其他线程等待
 public static Singleton3 getSingleton() {
    synchronized (Singleton3.class) {
        if (singleton == null) {
```

```
                singleton = new Singleton3();
            }
        }
        return singleton;
    }
}
```

- 我们通过加锁的方式保证了单例模式的安全性，但因为给获取对象的方法进行了加锁，多个线只有一个能进行判断，其他进程进入阻塞状态，等待上个进程执行结束后才能进行判断，影响性能

4. **双重检验锁**

- 我们可以使用双重检验锁的形式来优化

```
public class Singleton4 {

    private static Singleton4 singleton = null;

    private Singleton_3() {
    }
    public static Singleton4 getSingleton() {
        // 先进行判断，为null再进入同步代码块
        if (singleton == null) {
            synchronized (Singleton_3.class) {
                if (singleton == null) {
                    singleton = new Singleton_3();
                }
            }
        }
        return singleton;
    }
}
```

- 但是DCL也存在有缺陷，由于jvm的指令重排序，DCL偶尔也会存在失效的情况
- 创建一个对象这个操作并不是一个原子性的操作，jvm会生成三个指令
  - 指令1：给对象分配内存空间
  - 指令2：调用构造器，初始化对象属性
  - 指令3：将对象引用指向内存地址
- java编译器可能会对指令进行优化，可能会把指令执行顺序变为
  - 执行指令1：分配对象空间
  - 执行指令3：将对象引用执行内存地址
  - 执行指令2：调用构造器，初始化对象属性
- 我们来分析重排序后可能存在的问题
  - 在线程1执行到重排序后的第二步之后，然后cpu正好切换到线程2工作，且线程2也正好进了**getSingleton**操作，此时对象处于**创建了引用且分配了内存空间但未进行初始化的状态**，那么线程在执行 **if (singleton == null)**判断的时候，线程2发现对象不为null，那么线程2就获取到了一个有进行初始化的对象
- 所以在这种情况下，双重检验锁的方式会出现DCL失效的问题。

● 优化双重检验锁以及双重检验锁失效的问题非本文重点，有兴趣可参考 双重检验锁失效”的问题明以及相关优化以及翻译版本

5. **静态内部类**

● 当外部类内被访问时，并不会加载内部类，我们可以通过这一特性来进行设计单例

```
public class Singleton4 {
    private Singleton4() {
    }

    // 定义静态内部类
    private static class BuildSingleton4 {
        // 加载外部类时不会执行，只有加载 BuildSingleton4 时才会执行
        private static Singleton4 singleton = new Singleton4();
    }

    public Singleton4 getSingleton() {
        //当内部类第一次被访问时，创建对象实例
        return BuildSingleton4.singleton;
    }
}
```

● 当外部内被访问时，并不会加载内部类，所以只要不访问 **BuildSingleton4**这个内部类，**private static Singleton4 singleton = new Singleton4()不会被执行，只有当ingleton4.getSingleton**被调用时访问内部类的属性，此时才会将对象进行实例化，这就相当于实懒加载的效果，这样既解决了恶汉模式下可能造成资源浪费的问题，也避免了了懒汉模式下的并发问。