链滴

设计模式 |06 单例模式

作者：qq692310342
原文链接：https://ld246.com/article/1567740797920
来源网站：链滴

开头说几句

博主的博客地址： https://www.jeffcc.top/

博主学习设计模式用的书是Head First的《设计模式》，强烈推荐配套使用！

什么是单例模式

权威定义：确保一个类只有一个实例，并提供一个全局访问点。

博主的理解：虽说单例模式对于许多人来说并不难，但是其中也是有很多需要注意的细节的。

设计方案

方案一：“急切”的单例

思路：所谓急切，是指我们在一开始的时候就创建出类的单例实例，不管有无实际需求。满足了单例计模式的需求。

```
package singleton;

/**
 * 急切式单例
 */
public class SingletonEagerly {

//    静态变量设置全局的单例singletonEagerly
    public static SingletonEagerly singletonEagerly = new SingletonEagerly();
    //    私有化构造器
    private SingletonEagerly() {
```

```
    }
}
```

设计关键：私有化构造器,静态变量new一个类实现单例；

设计问题：如果一直不需要这个单例，或者系统设计中有一大堆的单例，那么会影响性能！

方案二 “懒汉式”单例

思路：类提供一个供对外访问的静态方法getInstance()来对外提供一个全局访问点，然后通过私有化造器，实现单例模式。

```
package singleton;

/**
 * 懒汉式单例模式
 */
public class SingletonLazy {
//    设置一个私有的静态变量定义单例
    private static SingletonLazy singletonLazy;
//    私有化构造器
    private SingletonLazy(){};

    /**
     * 对外提供一个获取单例的全局访问点 需要公开
     * @return
     */
    public static SingletonLazy getInstance(){
//        进行判断
        if(singletonLazy==null){
            return new SingletonLazy();
        }
        return singletonLazy;
    }

}
```

设计关键：私有化构造器、对外提供getInstance获取单例

设计问题：无法应对多线程同时访问的情况，如果出现线程A进入到getInstance的判断为null之后被程B抢占了资源，则B也会重新建立一个实例，而线程A也会建立一个实例，这就会出现两个单例的情，不满足要求。

方案三 多线程模式

思路：通过synchronized关键字让getInstance方法变为同部方法就能轻松解决问题了

```
package singleton;

/**
 * 懒汉式单例模式
 */
public class SingletonThread {
```

```
//    设置一个私有的静态变量定义单例
    private static SingletonThread SingletonThread;
//    私有化构造器
    private SingletonThread(){};

    /**
     * 对外提供一个获取单例的全局访问点 需要公开
     * synchronized保证为同步方法
     * @return
     */
    public static synchronized SingletonThread getInstance(){
//        进行判断
        if(SingletonThread==null){
            return new SingletonThread();
        }
        return SingletonThread;
    }

}
```

设计关键：使用synchronized关键字实现线程同步

设计问题：每当需要获取实例都要通过这个同步方法，效率会下降至少一百倍；实际上只有第一次是要进行同步操作的，创建实例之后的同步就成了累赘了！

方案四 双重检查加锁的单例模式

思路：双重检查机制让方法只会第一次被锁上，以后的情况则不会出现需要同步的情况。

```
package singleton;

/**
 * 懒汉式单例模式
 */
public class Singleton {
    //    设置一个私有的静态变量定义单例
//    这里注意我们给实例加了一个volatile关键字
//    实现了多线程环境下，变量只有一个版本！ 保证了可见性！
    private static volatile Singleton singleton;

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

    ;

    /**
     * 对外提供一个获取单例的全局访问点 需要公开
     * synchronized保证同步整个类
     *
     * @return
     */
    public static Singleton getInstance() {
```

```
//        第一重判断
        if (singleton == null) {
//            第一次初始化才会进入这里同步方法
            synchronized (Singleton.class) {
//                第二重判断
                if (singleton == null) {
                    return new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

设计关键：给实例加了一个volatile关键字，实现了多线程环境下，变量只有一个版本！ 保证了可见！同时我们给getInstance方法提供了双重检查机制，让方法不会一直被同步。

设计问题：会有一些复杂

方案五 私有化静态内部类

```
package singleton;

/**
 * 静态私有内部类、支持多并发、效率高、
 */
public class Singleton1 {
    private Singleton1() {}

    /**
     * 静态化一个内部类 用于创建单例
     */
    private static class SingletonHolder {
        private static Singleton1 instance = new Singleton1();
    }

    public static Singleton1 getInstance() {
        return SingletonHolder.instance;
    }
}
```

设计关键：通过静态私有化内部类，高效、且在需要的时候才会被创建 高效支持高并发

回到定义

容易发现其实上面的四个单例模式的方案都是可以满足单例设计模式的要求的，至于我们又该如何做选择，就要看具体的需求了，在确定性能以及资源的限制下，我们可以选择适合自己的单例模式，譬如果没有什么限制，使用方案一急切式的单例模式也是可以的，但是这个直接使用全局变量的问题是我们可能会造成在系统日渐复杂时候，用这么多的全局变量指向许多小对象会使得命名空间被“污染；同时这个单例的控制权不在我么自己的手中，而是掌握在了JVM的手中。

Java中使用到的单例模式

spirng框架

对于最常用的spring框架来说，我们经常用spring来帮我们管理一些无状态的bean，其默认设置为例，这样在整个spring框架的运行过程中，即使被多个线程访问和调用，这些“无状态”的bean就会存在一个，为他们服务。那么“无状态”bean指的是什么呢？

1. 无状态：当前我们托管给spring框架管理的javabean主要有service、mybatis的mapper、一些util，这些bean中一般都是与当前线程会话状态无关的，没有自己的属性，只是在方法中会处理相应的辑，每个线程调用的都是自己的方法，在自己的方法栈中。

2. 有状态：指的是每个用户有自己特有的一个实例，在用户的生存期内，bean保持了用户的信息，“有状态”；一旦用户灭亡（调用结束或实例结束），bean的生命期也告结束。即每个用户最初都得到一个初始的bean，因此在将一些bean如User这些托管给spring管理时，需要设置为prototype例，因为比如user，每个线程会话进来时操作的user对象都不同，因此需要设置为多例。

优势：

1. 减少了新生成实例的消耗，spring会通过反射或者cglib来生成bean实例这都是耗性能的操作，其给对象分配内存也会涉及复杂算法；

2. 减少jvm垃圾回收；

3. 可以快速获取到bean；

劣势：

单例的bean一个最大的劣势就是要时刻注意线程安全的问题，因为一旦有线程间共享数据变很可能发问题。

log4j

在使用log4j框架时也注意到了其使用的是单例，当然也为了保证单个线程对日志文件的读写时不出题。

END

2019年9月6日11:33:04