**设计模式 实验4-单例模式延伸**

一.简答题（共1题）

编写一个类LimitInstanceClass，该类最多可以实例化指定个数实例。实例的个数用配置文件InstanceLimit.cfg指定。例如，如果InstanceLimit.cfg的内容为2，则LimitInstanceClass最多可以同时存在2个对象。LimitInstanceClass的对象有一个整型成员变量id，保存对象的编号；有一个boolean型变量isBusy，如果该变量的值为true，表示该对象正在被使用，否则该对象空闲。getInstance()方法判断是否存在空闲的对象，存在将该空闲对象的isBusy置为true，并返回该对象；如果不存在空闲对象则返回null。LimitInstanceClass有一个release()方法，该方法将对象的isBusy置为false。LimitInstanceClass的对象还有一个String类型的成员变量accessMessage，以及一个成员方法writeAccessMessage(String message)，该方法将参数message追加到accessMessage。LimitInstanceClass的printAccessMessage()方法输出accessMessage的内容。

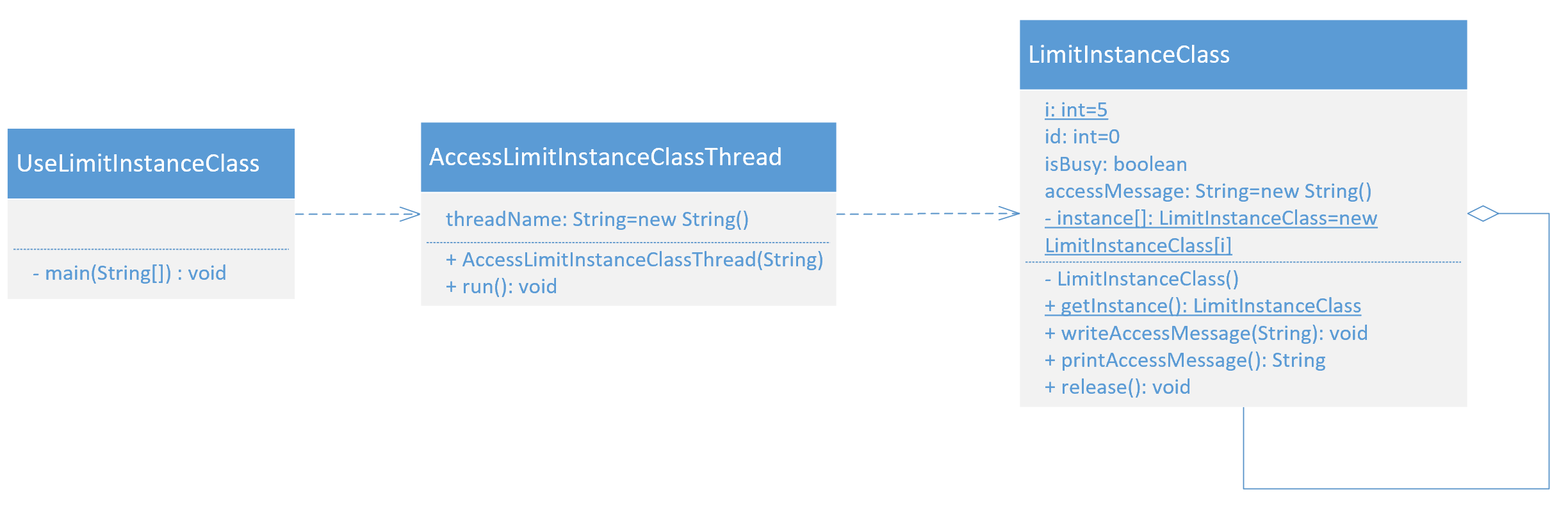
编写一个线程类AccessLimitInstanceClassThread，在其run()方法中获取一个LimitInstanceClass对象，调用获得的对象的writeAccessMessage(String message)将自己的线程名写入accessMessage，随机休眠0-5秒，再调用printAccessMessage()，最后调用release（）方法。

编写一个UseLimitInstanceClass类，在其main方法中实例化10个AccessLimitInstanceClassThread线程对象，并启动各个线程。设置InstanceLimit.cfg的内容为3。编写实验报告，要求画出相应的类图，提供java源程序。

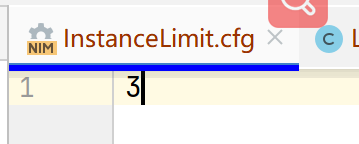
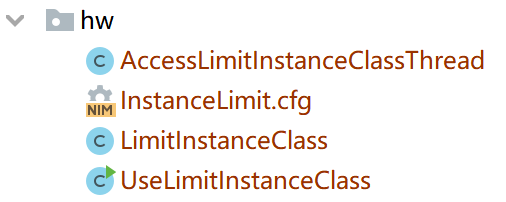
1. 类图

该系统涉及到指定个数实例，运用的是单例模式的延伸--多例模式。

UseLimitInstanceClass作为客户端模拟系统运作。



1. 源程序



①编写一个类LimitInstanceClass，该类最多可以实例化配置文件InstanceLimit.cfg中的指定个数实例

public class LimitInstanceClass {  
 static int i=5;  
 static int id=0;//对象编号  
 boolean isBusy; //如果该变量的值为true，表示该对象正在被使用，否则该对象空闲  
 String accessMessage=new String();  
   
 static {  
 try {  
 FileInputStream f=new FileInputStream("D:\\CodeProjects\\IdeaProjects\\design\_patterns\\src\\main\\java\\com\\pattern\\singleton\\hw\\InstanceLimit.cfg");  
 i=f.read()-48;  
 System.out.println("能产生"+i+"个对象");  
 f.close();  
 }  
 catch(IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 //饿汉式多例 类加载时就初始化  
 private static LimitInstanceClass instance[]=new LimitInstanceClass[i];  
 static {  
 for(int m=0;m<i;m++) {  
 instance[m]=new LimitInstanceClass();  
 }  
 }  
  
 //让构造函数为 private，这样该类就不会被实例化  
 private LimitInstanceClass() {  
 isBusy=false;  
 }  
 //获取空闲对象  
 public static LimitInstanceClass getInstance() {  
 int x;  
 for(x=0;x<i-1;x++) {  
 if(instance[x].isBusy==false) {  
 instance[x]=new LimitInstanceClass();  
 instance[x].isBusy=true;  
 break;  
 }  
 }  
 return instance[x];  
 }  
 //追加  
 public void writeAccessMessage(String message) {  
 accessMessage=message;  
 }  
 //输出  
 public String printAccessMessage(){  
 System.out.println(LocalTime.now().withNano(0) + " " + accessMessage );  
 return accessMessage;  
 }  
 //释放一个对象  
 public void release() {  
 isBusy=false;  
 }

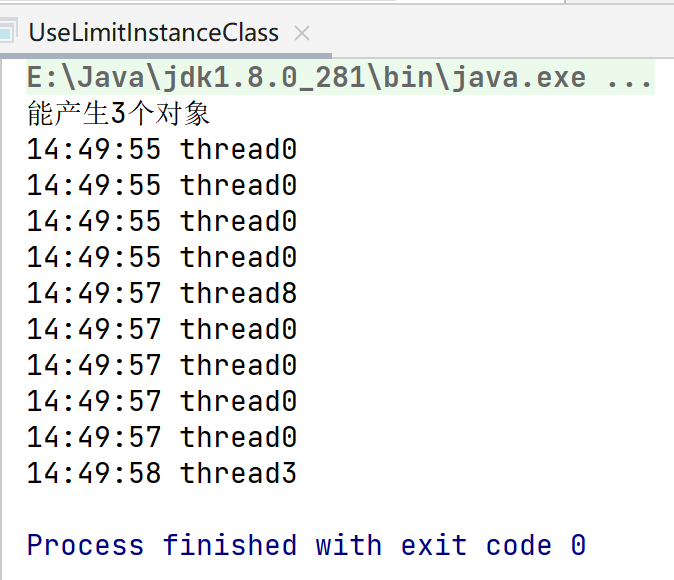
②编写一个线程类AccessLimitInstanceClassThread

public class AccessLimitInstanceClassThread extends Thread {  
 String threadName=new String();  
 public AccessLimitInstanceClassThread(String threadName) {  
 this.threadName=threadName;  
 }  
  
 public void run(){  
 //获取一个LimitInstanceClass对象  
 LimitInstanceClass limitInstance= LimitInstanceClass.getInstance();  
 //将自己的线程名写入accessMessage  
 limitInstance.writeAccessMessage(threadName);  
 try { //随机休眠0-5秒  
 Thread.sleep((int)(Math.random()\*5000));  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 limitInstance.printAccessMessage();  
 limitInstance.release();  
 }  
}

③编写一个UseLimitInstanceClass类，在其main方法中实例化10个AccessLimitInstanceClassThread线程对象，并启动各个线程。

public class UseLimitInstanceClass {  
 public static void main(String args[]) {  
 final int N=10;  
 AccessLimitInstanceClassThread t[]=new AccessLimitInstanceClassThread[N];  
 for(int i=0;i<N;i++){  
 t[i]=new AccessLimitInstanceClassThread("thread"+i);  
 t[i].start();  
 }  
 }  
}

1. 程序运行结果



可见，LimitInstanceClass只能实例化配置文件指定的3个对象。

1. 总结

对于单例和多例模式的使用场景总结。

1. 什么是单例多例:

所谓单例就是所有的请求都用一个对象来处理，比如常用的service和dao层的对象通常都是单例的，而多例则指每个请求用一个新的对象来处理，比如action;

1. 如何产生单例多例:

在通用的SSH中，单例在spring中是默认的，如果要产生多例，则在配置文件的 bean中添加scope="prototype";

1. 为什么用单例多例：

之所以用单例，是因为没必要每个请求都新建一个对象，这样子既浪费CPU又浪 费内存；

之所以用多例，是为了防止并发问题；即一个请求改变了对象的状态，此时对象又 处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处 理；

用单例和多例的标准只有一个：

当对象含有可改变的状态时（更精确的说就是在实际应用中该状态会改变），则多 例，否则单例；

1. 何时用单例？何时用多例？

对于struts2来说，action必须用多例，因为action本身含有请求参数的值， 即可改变的状态；

而对于STRUTS1来说，action则可用单例，因为请求参数的值是放在actionForm 中，而非action中的；

另外，并不是说service或dao一定是单例，标准同第3点所讲的，就曾见过有的 service中也包含了可改变的状态，同时执行方法也依赖该状态，但一样用的单例， 这样就会出现隐藏的BUG,而并发的BUG通常很难重现和查找；