单例模式

目 录

1. 概述

java 中单例模式是一种常见的设计模式，单例模式分三种：懒汉式单例、饿汉式单例、登记式单例三种。

单例模式有以下特点：

1、单例类只能有一个实例。

2、单例类必须自己创建自己的唯一实例。

3、单例类必须给所有其他对象提供这一实例。

单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个 Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。

首先看一个经典的单例实现，代码如示例 1 所示。

示例 1：

```
public class Singleton {
    private static Singleton uniqueInstance = null;

    private Singleton() {

    }

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

示例 1 中，Singleton 类通过将构造方法限定为 private 避免了类在外部被实例化，在同一个虚拟机范围内，Singleton 的唯一实例只能通过 getInstance()方法访问。（事实上，通过 Java 反射机制是能够实例化构造方法为 private 的类的，那基本上会使所有的 Java 单例实现失效。此问题在此处不做讨论。）

但是以上实现没有考虑线程安全问题。所谓线程安全是指：如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。或者说：一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。显然以上实现并不满足线程安全的要求，在并发环境下很可能出现多个 Singleton 实例。代码如示例 2 所示。

示例 2：

```
class TestStream  {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    //该类只能有一个实例
    private TestStream (){}     //私有无参构造方法
    //该类必须自行创建，有 2 种方式
    /*private static final TestStream ts=new TestStream();*/
    private static TestStream  ts1=null;
    //这个类必须自动向整个系统提供这个实例对象
    public static TestStream  getTest(){
        if(ts1==null){
            ts1=new TestStream ();
        }
        return ts1;
    }
    public void getInfo(){
        System.out.println("姓名："+name);
    }
}
public class TestMain {
    public static void main(String [] args){
       TestStream  s=TestStream.getTest();
        s.setName("光头强");
        System.out.println(s.getName());
        TestStream  s1=TestStream .getTest();
        s1.setName("光头强");
        System.out.println(s1.getName());
        s.getInfo();
        s1.getInfo();
        if(s==s1){
            System.out.println("创建的是同一个实例");
        }else if(s!=s1){
           System.out.println("创建的不是同一个实例");
        }else{
           System.out.println("application error");
        }
    }
```

```
}
```

示例 2 运行效果如图 1 所示。

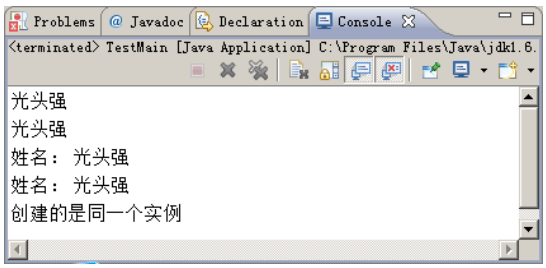

图 1 示例 2 运行效果

由示例运行效果可见：单例模式为一个面向对象的应用程序提供了对象惟一的访问点，不管它实现何种功能，整个应用程序都会同享一个实例对象。

2. 三种单例模式

1、懒汉式单例类，在第一次调用的时候实例化，代码如示例 3 所示

示例 3：

```
public class Singleton1 {
     //私有的默认构造方法
     private Singleton1() {}
     //注意，这里没有 final
     private static Singleton1 single=null;
     //静态工厂方法
     public synchronized static Singleton1 getInstance() {
          if (single == null) {
              single = new Singleton1();
          }
          return single;
     }
 }
```

2、饿汉式单例类，在类初始化时已经自行实例化，代码如示例 4 所示。

示例 4：

```
 public class Singleton2 {
    //私有的默认构造方法
     private Singleton2 () {}
     //已经自行实例化
     private static final Singleton2 single = new Singleton2();
     //静态工厂方法
     public static Singleton2 getInstance() {
         return single;
     }
 }
```

3、登记式单例类，将类名注册，下次从里面直接获取，代码如示例 5 所示。

示例 5：

```
import java.util.HashMap;
import java.util.Map;

public class Singleton3 {
   private static Map<String,Singleton3> map =
                                          new HashMap<String,Singleton3>();
   static{
       Singleton3 single = new Singleton3();
       map.put(single.getClass().getName(), single);
     }
     //保护的默认构造方法
     protected Singleton3(){}
     //静态工厂方法,返还此类惟一的实例
     public static Singleton3 getInstance(String name) {
         if(name == null) {
            name = Singleton3.class.getName();
             System.out.println("name == null"+"--->name="+name);
         }
         if(map.get(name) == null) {
            try {
                 map.put(name,(Singleton3)Class.forName(name)
                         .newInstance());
             } catch (InstantiationException e) {
                 e.printStackTrace();
             } catch (IllegalAccessException e) {
                 e.printStackTrace();
             } catch (ClassNotFoundException e) {
                 e.printStackTrace();
             }
         }
         return map.get(name);
     }
     //用于测试的方法
     public String about() {
         return "Hello, I am RegSingleton.";
     }
     public static void main(String[] args) {
         Singleton3 single3 = Singleton3.getInstance(null);
         System.out.println(single3.about());
     }
}
```