第八章——单例模式

H1

> 概述：单例模式是最最简单的设计模式，它的出现是为了让系统中的一个类只有实例

一、定义

H2

> 确保一个类只有一个实例， 并提供一个全局访问点来访问这个唯一实例。

二、结构与实现

结构：

H2

> 对于Singleton(单例），在单例类的内部创建它的唯一实例， 并通过静态方法getlnstance()让客户端可以使用它的唯一实例； 为了防止在外部对单例类实例化， 将其构造函数的可见性设为private。在单例类内部定义一个Singleton类型的静态对象作为外部共享访问的唯一实例

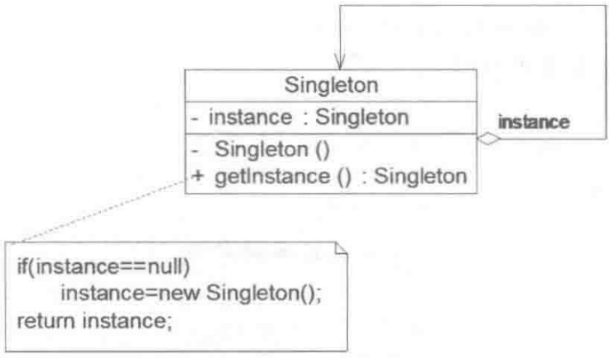


图 8-2　单例模式结构图

实现代码

```java
package com.IQIUM.Singletons.Normal;

/**
 * 使用 IoDH 方法实现单例模式
 */
public class Singleton {
    public Singleton() {
    }
    /**
     * 在 单例类 内部设置一个静态内，该类只有在 创建的时候会调用静态方法，所以它满足饿汉式
     * 单例模式，同时静态对象一旦创建就会放在堆空间中，所以它满足唯一性！！
     */
    public static class Holderclass {
        private final static Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return Holderclass.instance;
    }

    public static void main(String[] args) {
        Singleton s1 = getInstance();
        Singleton s2 = getInstance();
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

三、饿汉式和懒汉式单例

（一）饿汉式单例

结构图：

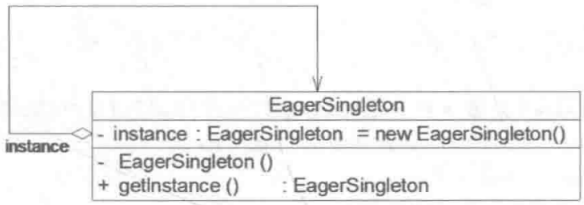


图 8-4 饿汉式单例类图

解释：定义静态变量的实例化对象（在java中，静态变量在类被加载的时候就会被创建！！！！）

（二）懒汉式单例

结构图：

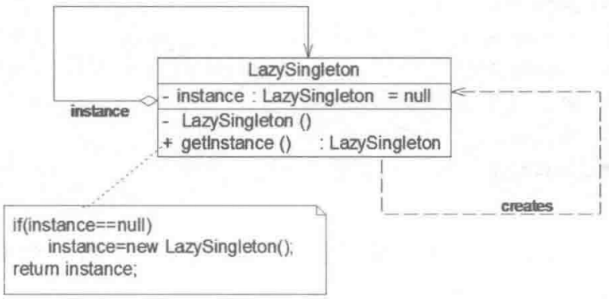


图 8-5 懒汉式单例类图

解释：懒汉式将单例创建的时间放在了第一次调用 `getInstance` 的时候，但这样可能会导致线程不安全的现象（了解，使用 `Java` 的加锁机制）

四、单例模式的优缺点

（一）优点

（l）单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例， 所以
它可以严格控制客户怎样以及何时访问它。
（２） 由于在系统内存中只存在一个对象， 因此可以节约系统资源， 对于一些需要频繁创
建和销毁的对象， 单例模式无疑可以提高系统的性能。

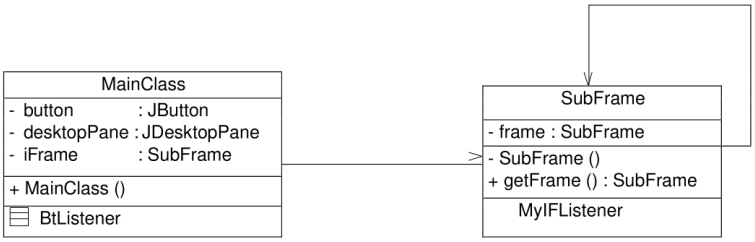
（3） 允许可变数目的实例。基于单例模式可以进行扩展， 使用与控制单例对象相似的
方法来获得指定个数的实例对象， 既节省系统资源， 又解决了由于单例对象共享过多有损性能的问题。

（二） 缺点

（l） 由于单例模式中没有抽象层， 因此单例类的扩展有很大的困难。
（2）单例类的职责过重， 在一定程度上违背了单一职责原则。因为单例类既提供了业
务方法， 又提供了创建对象的方法（工厂方法）， 将对象的创建和对象本身的功能耦合在
（3）现在很多面向对象语言（如Java 、C# ） 的运行环境都提供了自动垃圾回收技术， 因此如果实例化的共享对象长时间不被利用， 系统会认为它是垃圾， 会自动销毁并回收资源，下次利用时又将重新实例化， 这将导致共享的单例对象状态的丢失。

五、课后习题答案

1. B
2. B
3. B
4. 参见P111-P114， 可从延迟加载、 线程安全、 响应时间等角度进行分析与对比。 5. 参见P112-P113。
5. 双重检查锁定方式实现代码参见P113； IoDH 方式实现代码参见P114。
6. （略）
7. `SubFrame` 类充当单例类， 在其中定义了静态工厂方法 getFrame()。




```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

//子窗口： 单例类
class SubFrame extends JInternalFrame {
```

```
8         private static SubFrame frame;// 静态实例
9         // 私有构造函数
10
11        private SubFrame() {
12            super(" 子 窗 体 ", true, true, true, false);
13            this.setLocation(20, 20); // 设置内部窗体位置
14            this.setSize(200, 200); // 设置内部窗体大小
15            this.addInternalFrameListener(new MyIFListener());// 监听窗体事件
16            this.setVisible(true);
17        }
18
19        // 工厂方法, 返回窗体实例public
20        static SubFrame getFrame() {
21            // 如果窗体对象为空, 则创建窗体, 否则直接返回已有窗体
22            if (frame == null) {
23                frame = new SubFrame();
24            }
25            return frame;
26        }
27
28        // 事件监听器
29        class MyIFListener extends InternalFrameAdapter {
30            // 子窗体关闭时, 将窗体对象设为 null
31            public void internalFrameClosing(InternalFrameEvent e) {
32                if (frame != null) {
33                    frame = null;
34                }
35            }
36        }
37    }
38
39    // 客户端测试类
40    class MainClass extends JFrame {
41        private JButton button;
42        private JDesktopPane desktopPane;
43        private SubFrame iFrame = null;
44
45        public MainClass() {
46            super("主窗体");
47            Container c = this.getContentPane();
48            c.setLayout(new BorderLayout());
49            button = new JButton("点击创建一个内部窗体");
50            button.addActionListener(new BtListener());
51            c.add(button, BorderLayout.SOUTH);
52            desktopPane = new JDesktopPane(); // 创建DesktopPane
```

```
            c.add(desktopPane);
            this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            this.setLocationRelativeTo(null);
            this.setSize(400, 400);
            this.show();
        }

        // 事件监听器
        class BtListener implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                if (iFrame != null) {
                    desktopPane.remove(iFrame);
                }
                iFrame = SubFrame.getFrame();
                desktopPane.add(iFrame);
            }
        }

        public static void main(String[] args) {
            new MainClass();
        }
    }
```

`SubFrame` 类是 `JInternalFrame` 类的子类， 在 `SubFrame` 类中定义了一个静态的
`SubFrame` 类型的实例变量， 在静态工厂方法 `getFrame()` 中创建了 `SubFrame` 对象并将其返回。
在 `MainClass` 类中使用了该单例类， 确保子窗口在当前应用程序中只有唯一一个实例， 即只能弹出一个子窗口。

8. 多例模式(Multiton Pattern)是单例模式的一种扩展形式， 多例类可以有多个实例， 而且必须自行创建和管理实例， 并向外界提供自己的实例， 可以通过静态集合对象来存储这些实例。
   多例类Multiton 的代码如下所示：

```
import java.util.*;

public class Multiton {
    // 定义一个数组用于存储四个实例
    private static Multiton[] array = { new Multiton(), new
Multiton(), new Multiton(), new Multiton() };

    // 私有构造函数
    private Multiton() {
```

```
    }

    // 静态工厂方法，  随机返回数组中的一个实例
    public static Multiton getInstance() {
        return array[random()];
    }

    // 随机生成一个整数作为数组下标
    public static int random() {
        Date d = new Date();
        Random random = new Random();
        int value = Math.abs(random.nextInt());
        value = value % 4;
        return value;
    }

    public static void main(String args[]) {
        Multiton m1, m2, m3, m4;
        m1 = Multiton.getInstance();
        m2 = Multiton.getInstance();
        m3 = Multiton.getInstance();
        m4 = Multiton.getInstance();
        System.out.println(m1 == m2);
        System.out.println(m1 == m3);
        System.out.println(m1 == m4);
    }
}
```