1. 设计模式

1. 引言

1.1. 什么是设计模式?

> 类似解题模式：遇到同类的问题，使用固定的模式来解决，如一元二次方程式

1.2. 设计模式的分类

1.2.1. 构造型模式

> 帮助我们更好地组织创建对象的代码。增强弹性，以应付在不同情况下创建和初始化对象的代码变化。

- 工厂方法模式
- 抽象工厂模式
- 单例模式 Singleton
- 建造者模式
- 原型模式

1.2.2. 结构型模式

> 增强代码重用，优化对象结构，使其职责分明、粒度合适，减少代码耦合。

- 适配器模式
- 装饰器模式

- 代理模式
- 外观模式
- 桥接模式
- 组合模式
- 享元模式

1.2.3. 行为型模式

> 更好地定义对象间的协作关系，使复杂的程序流程变得清晰。

- 策略模式
- 模板方法模式
- 观察者模式
- 迭代子模式
- 责任链模式
- 命令模式
- 备忘录模式
- 状态模式
- 访问者模式
- 中介者模式
- 解释器模式

---

2. 构造型模式

2.1. 单例模式 Singleton

2.1.4. 定义

> 唯一：确保一个类只有一个实例

2.1.5. 生活场景

- 一个就够了
- 多个反而会乱

投影仪：教室里只用一个投影仪就够了，如果装了多个投影仪到处投影会闪瞎你的眼

老师上课：同一个教室只有一个老师在上课，如果有多个老师同时在一个教室上课，学生如何听课？

2.1.6. 应用场景

代码中有一些对象正是如此：线程池、缓存、对话框、全局配置，以及Servlet、Spring的Bean等等，这类对象只需要一个实例，如果制造出多个实例

2.1.7. 模式的优点和缺点

- 节约系统资源
- 减少频繁创建与销毁对象
- 避免并发的场景下发生行为异常

2.1.8. 实现案例

2.1.8.1. 静态工厂式(最简单)

> 这是我最喜欢的一种方式来实现单例模式，是单例的最简单实现方式。

```
/**
 * XXX Pattern:Singleton:静态工厂式
 *
 * @author 张柏子
 *
 */
public class HungrySingleton {
    /**
     * 类加载时就初始化static、final的字段
     */
    private static final HungrySingleton INSTANCE = new HungrySingleton();

    /**
     * 私有的构造函数(表明这个类不可能通过外部调用创建实例)
     */
    private HungrySingleton() {
    }

    /**
     * 获取实例（只能通过此方法创建实例）
     */
    public static HungrySingleton getInstance() {
        return INSTANCE;
    }

    private final String[] names = { "张三", "李四", "王五" };

    public void printNames() {
        System.out.println("hello," + Arrays.toString(names));
    }

    public static void main(String[] args) {
        HungrySingleton.getInstance().printNames();
    }
}
```

> 静态工厂式的缺点

这种写法如果完美的话，就不会再BB后面那么多内容了，因为它有下面两个问题：

1. 它不是一种懒加载模式（lazy initialization）

单例会在加载类后一开始就被初始化，即使客户端没有调用 getInstance()方法。

在一些场景中将无法使用：譬如 Singleton 实例的创建是依赖参数或者配置文件的，在 getInstance() 之前必须调用某个方法设置参数给它，那样这种单例写法就无法使用了。

2. 当实现了Serializable接口后，反序列化时单例会被破坏

实现Serializable接口需要重写readResolve，才能保证其反序列化依旧是单例：

```
public class HungrySingleton implements Serializable {

    ......
    ......
    ......

    /**
     * 如果实现了Serializable, 必须重写这个方法
     */
    private Object readResolve() throws ObjectStreamException {
        return INSTANCE;
    }
}
```

2.1.8.2. 教科书式

当要实现懒加载的单例模式时，很多人的第一反应是写出如下的代码，包括教科书上也是这样教我们的。

```
/**
 * XXX Pattern:Singleton:教科书式（线程不安全）
 *
 * @author 张柏子
 *
 */
public class TextBookSingleton {
    private static TextBookSingleton INSTANCE = null;

    /**
     * 私有的构造函数(表明这个类不可能通过外部调用创建实例)
     */
    private TextBookSingleton() {
    }
```

```
    /**
     * 获取实例（只能通过此方法创建实例）
     */
    public static TextBookSingleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new TextBookSingleton();
        }
        return INSTANCE;
    }

    private final String[] names = { "张三", "李四", "王五" };

    public void printNames() {
        System.out.println("hello," + Arrays.toString(names));
    }

    public static void main(String[] args) {
        TextBookSingleton.getInstance().printNames();
    }
}
```

> 这段代码简单明了，而且使用了懒加载模式，但是在多个线程并行调用 getInstance() 的时候，就会创建多个实例，也就是说是线程不安全的。

2.1.8.3. 懒汉式

> 为了解决教科书式的问题，最简单的方法是将整个 getInstance() 方法设为synchronized（同步）。

```
/**
    * 获取实例（只能通过此方法创建实例）<br>
    * 将整个 getInstance() 方法设为synchronized（同步）
    */
public static synchronized LazySingleton getInstance() {
    if (INSTANCE == null) {
        INSTANCE = new LazySingleton();
    }
    return INSTANCE;
}
```

> 懒汉式简单粗暴的同步整个方法，导致同一时间内只有一个线程能够调用getInstance方法，这样又出现了性能问题。

2.1.8.4. 双检锁DCL（最复杂）