·单例模式（Singleton）

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

首先我们写一个简单的单例类：

[java] view plaincopy

public class Singleton {

/\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/

private static Singleton instance = null;

/\* 私有构造方法，防止被实例化 \*/

private Singleton() {

}

/\* 静态工程方法，创建实例 \*/

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

/\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/

public Object readResolve() {

return instance;

}

}

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对getInstance方法加synchronized关键字，如下：

[java] view plaincopy

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

[java] view plaincopy

public static Singleton getInstance() {

if (instance == null) {

synchronized (instance) {

if (instance == null) {

instance = new Singleton();

}

}

}

return instance;

}

似乎解决了之前提到的问题，将synchronized关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在instance为null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在Java指令中创建对象和赋值操作是分开进行的，也就是说instance = new Singleton();语句是分两步执行的。但是JVM并不保证这两个操作的先后顺序，也就是说有可能JVM会为新的Singleton实例分配空间，然后直接赋值给instance成员，然后再去初始化这个Singleton实例。这样就可能出错了，我们以A、B两个线程为例：

a>A、B线程同时进入了第一个if判断

b>A首先进入synchronized块，由于instance为null，所以它执行instance = new Singleton();

c>由于JVM内部的优化机制，JVM先画出了一些分配给Singleton实例的空白内存，并赋值给instance成员（注意此时JVM没有开始初始化这个实例），然后A离开了synchronized块。

d>B进入synchronized块，由于instance此时不是null，因此它马上离开了synchronized块并将结果返回给调用该方法的程序。

e>此时B线程打算使用Singleton实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。

-------------------------------------------------------------------------------------------------------

·观察者模式（Observer Pattern）

如果有更新，那么订阅了该话题/主题的客户端将被通知。

理解观察者模式的最简单方法是想象一个邮件列表，你可以在其中订阅任何主题，无论是开源、技术、名人、烹饪还是您感兴趣的任何其他内容。每个主题维护者一个它的订阅者列表，在观察者模式中它们相当于观察者。当某一个主题更新时，它所有的订阅者（观察者）都将被通知这次改变。并且订阅者总是能取消某一个主题的订阅。

让我们来看看观察者模式的代码示例，从主题/话题类开始：

package org.opensource.demo.observer;

public interface Topic {

public void addObserver([Observer][22] observer);

public void deleteObserver([Observer][22] observer);

public void notifyObservers();

}

这段代码描述了一个为不同的主题去实现已定义方法的接口。注意一个观察者如何被添加、移除和通知的。

这是一个主题的实现示例：

package org.opensource.demo.observer;

import java.util.List;

import java.util.ArrayList;

public class Conference implements Topic {

private List&lt;Observer&gt; listObservers;

private int totalAttendees;

private int totalSpeakers;

private [String][18] nameEvent;

public Conference() {

listObservers = new ArrayList&lt;Observer&gt;();

}

public void addObserver([Observer][22] observer) {

listObservers.add(observer);

}

public void deleteObserver([Observer][22] observer) {

int i = listObservers.indexOf(observer);

if (i &gt;= 0) {

listObservers.remove(i);

}

}

public void notifyObservers() {

for (int i=0, nObservers = listObservers.size(); i &lt; nObservers; ++ i) {

[Observer][22] observer = listObservers.get(i);

observer.update(totalAttendees,totalSpeakers,nameEvent);

}

}

public void setConferenceDetails(int totalAttendees, int totalSpeakers, [String][18] nameEvent) {

this.totalAttendees = totalAttendees;

this.totalSpeakers = totalSpeakers;

this.nameEvent = nameEvent;

notifyObservers();

}

}

这段代码定义了一个特定主题的实现。当发生改变时，这个实现调用它自己的方法。注意这将获取观察者的数量，它以列表方式存储，并且可以通知和维护观察者。

这是一个观察者类：

package org.opensource.demo.observer;

public interface [Observer][22] {

public void update(int totalAttendees, int totalSpeakers, [String][18] nameEvent);

}

这个类定义了一个接口，不同的观察者可以实现该接口以执行特定的操作。

例如，实现了该接口的观察者可以在会议上打印出与会者和发言人的数量：

package org.opensource.demo.observer;

public class MonitorConferenceAttendees implements [Observer][22] {

private int totalAttendees;

private int totalSpeakers;

private [String][18] nameEvent;

private Topic topic;

public MonitorConferenceAttendees(Topic topic) {

this.topic = topic;

topic.addObserver(this);

}

public void update(int totalAttendees, int totalSpeakers, [String][18] nameEvent) {

this.totalAttendees = totalAttendees;

this.totalSpeakers = totalSpeakers;

this.nameEvent = nameEvent;

printConferenceInfo();

}

public void printConferenceInfo() {

[System][19].out.println(this.nameEvent + " has " + totalSpeakers + " speakers and " + totalAttendees + " attendees");

}

}

-------------------------------------------------------------------------------------------------------

·工厂模式（Factory Pattern）

特点：封装对象的创建过程，通过工厂类来创建对象，使得客户端与具体类的实例化解耦。

示例：一个汽车制造工厂，工厂根据客户的需求（参数）生产不同类型的汽车，客户只需要告诉工厂要生产什么样的汽车，而不需要关心具体的制造过程。

public interface Car {

void drive();

}

public class SedanCar implements Car {

@Override

public void drive() {

System.out.println("Driving a sedan car");

}

}

public class SUVCar implements Car {

@Override

public void drive() {

System.out.println("Driving an SUV car");

}

}

public class CarFactory {

public Car createCar(String type) {

if (type.equalsIgnoreCase("sedan")) {

return new SedanCar();

} else if (type.equalsIgnoreCase("suv")) {

return new SUVCar();

} else {

throw new IllegalArgumentException("Invalid car type: " + type);

}

}

}

-------------------------------------------------------------------------------------------------------

装饰者模式（Decorator Pattern）：

特点：动态地给一个对象添加额外的功能，避免了使用继承扩展功能的问题。

示例：一个咖啡店的订单系统，有一个基础咖啡类，通过装饰者模式可以动态地为该咖啡添加额外的配料，如牛奶、糖浆等，而无需修改基础咖啡类的代码。

public interface Coffee {

String getDescription();

double getCost();

}

public class SimpleCoffee implements Coffee {

@Override

public String getDescription() {

return "Simple coffee";

}

@Override

public double getCost() {

return 1.0;

}

}

public abstract class CoffeeDecorator implements Coffee {

protected Coffee decoratedCoffee;

public CoffeeDecorator(Coffee decoratedCoffee) {

this.decoratedCoffee = decoratedCoffee;

}

@Override

public String getDescription() {

return decoratedCoffee.getDescription();

}

@Override

public double getCost() {

return decoratedCoffee.getCost();

}

}

public class MilkDecorator extends CoffeeDecorator {

public MilkDecorator(Coffee decoratedCoffee) {

super(decoratedCoffee);

}

@Override

public String getDescription() {

return super.getDescription() + ", with milk";

}

@Override

public double getCost() {

return super.getCost() + 0.5;

}

}

-------------------------------------------------------------------------------------------------------

总结：

这些设计模式都有各自的特点和适用场景，通过使用它们可以提高代码的可维护性、灵活性和可复用性。单例模式适用于需要全局访问一个实例的情况；观察者模式适用于对象间存在一对多的依赖关系，当一个对象改变时需要通知其他依赖者；工厂模式适用于需要封装对象创建过程的情况，客户端只需要与工厂接口交互；装饰者模式适用于动态地为对象添加额外功能的情况，避免了使用继承带来的静态限制。