**总体概述**

**1 设计模式（Design pattern）**是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

**健壮性：功能准确前提下，是否考虑兼容空指针/数据异常/违法检查/边界值？**

**稳定性：**

**性能性：如果数据量很大是否能够支撑，并发性高是否考虑？多表连接的索引是否建立？是否可以改变方案使性能最大化？**

**安全性：接口是否有身份校验和数据加密？**

**可读性：**

**可扩展：如果需求改变了或者新增了是否好扩展，还是要全部推翻重来？**

**可伸缩：面向接口,**

**可复用：系统是否存在这样的工具类？会不会多个方法出现一样的代码？**

**可维护：核心代码注释是否齐全，模块之间是否松耦合？**

**可配置：是否能够动态配置，改变值可以不用修改代码？**

**高内聚低耦合：尽量使不同模块间少关联，即一个模块明确完成一个功能。但是一个模块内部又有许多子系统，子系统中的类之间不关联是不可能的，一个模块下的子系统要少用继承多用组合（使用组合时，就会使子系统的不同类之间产生关联）——总结一句话就是：模块之间要实现低耦合，模块下的类之间要多用组合少用继承。**

一：继承（is a）

继承是Is a 的关系，比如说Student继承Person,则说明Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

1：父类的内部细节对子类是可见的。

2：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

3：子类与父类是一种高耦合，违背了面向对象思想。

4 ：继承关系最大的弱点是打破了封装，子类能够访问父类的实现细节，子类与父类之间紧密耦合，子类缺乏独立性，从而影响了子类的可维护性。

5：不支持动态继承。在运行时，子类无法选择不同的父类。

二：组合（has a）

电脑和显卡，不能说电脑继承了显卡，而是说显卡是计算机的组成部分之一，这就是组合。

1：不破坏封装，整体类与局部类之间松耦合，彼此相对独立。

2：具有较好的可扩展性。

3：支持动态组合。在运行时，整体对象可以选择不同类型的局部对象。

**2 设计模式分为四大类，共33种。**

创建型模式共5种：

工厂方法模式（Factory Pattern）

抽象工厂模式（Abstract Factory Pattern）

单例模式（Singleton Pattern）

建造者模式（Builder Pattern）

原型模式（Prototype Pattern）

结构型模式共8种：

适配器模式（Adapter Pattern）

桥接模式（Bridge Pattern）

过滤器模式（Filter、Criteria Pattern）

组合模式（Composite Pattern）

装饰器模式（Decorator Pattern）

外观模式（Facade Pattern）

享元模式（Flyweight Pattern）

代理模式（Proxy Pattern）。

行为型模式共12种：

责任链模式（Chain of Responsibility Pattern）

命令模式（Command Pattern）

解释器模式（Interpreter Pattern）

迭代器模式（Iterator Pattern）

中介者模式（Mediator Pattern）

备忘录模式（Memento Pattern）

观察者模式（Observer Pattern）

状态模式（State Pattern）

空对象模式（Null Object Pattern）

策略模式（Strategy Pattern）

模板模式（Template Pattern）

访问者模式（Visitor Pattern）

JAVA EE型模式共8种：

MVC 模式（MVC Pattern）

业务代表模式（Business Delegate Pattern）

组合实体模式（Composite Entity Pattern）

数据访问对象模式（Data Access Object Pattern）

前端控制器模式（Front Controller Pattern）

拦截过滤器模式（Intercepting Filter Pattern）

服务定位器模式（Service Locator Pattern）

传输对象模式（Transfer Object Pattern）

**3 核心思想**

**AOP 分散关注，共享行为！**

**开闭原则，对扩展打开，对修改关闭。**

**高内聚，低耦合，迪米特原则**

**4 设计模式的六大原则**

（1）、开闭原则（Open Close Principle）

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

（2）、里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

（3）、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

（4）、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

（5）、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。高内聚低耦合。

（6）、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

**创建型模式共5种**

**1 工厂方法模式**

创建型模式，一个接口，几个实现类，一个接口工厂类，根据不同的参数创建不同的实例。但是增加子类需要修改工厂类。

应用实例： 1、您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。 2、Hibernate 换数据库只需换方言和驱动就可以。

**package** com.hiya.dp.creator.factoryMethod;

**public** **interface** IGame

{

String ***name*** = "Game";

**void** play();

}

**package** com.hiya.dp.creator.factoryMethod;

**public** **class** FightGame **implements** IGame

{

@Override

**public** **void** play()

{

System.***out***.println("This is FightGame。。。");

}

}

**package** com.hiya.dp.creator.factoryMethod;

**public** **class** GunGame **implements** IGame

{

@Override

**public** **void** play()

{

System.***out***.println("This is GunGame。。。");

}

}

**package** com.hiya.dp.creator.factoryMethod;

**public** **class** MotorGame **implements** IGame

{

@Override

**public** **void** play()

{

System.***out***.println("This is FightGame。。。");

}

}

**package** com.hiya.dp.creator.factoryMethod;

**public** **class** GameFactory

{

**public** IGame getGame(String gameType)

{

**if** (gameType == **null**)

{

**return** **null**;

}

**if** (gameType.equalsIgnoreCase("Fight"))

{

**return** **new** FightGame();

} **else** **if** (gameType.equalsIgnoreCase("Gun"))

{

**return** **new** GunGame();

} **else** **if** (gameType.equalsIgnoreCase("Motor"))

{

**return** **new** MotorGame();

}

**return** **null**;

}

}

**package** com.hiya.dp.creator.factoryMethod;

**public** **class** GameClient

{

**public** **static** **void** main(String[] args)

{

GameFactory shapeFactory = **new** GameFactory();

// 获取 FightGame 的对象，并调用它的 play 方法

IGame shape1 = shapeFactory.getGame("Fight");

shape1.play();

// 获取 GunGame 的对象，并调用它的 play 方法

IGame shape2 = shapeFactory.getGame("Gun");

shape2.play();

// 获取 MotorGame 的对象，并调用它的 play 方法

IGame shape3 = shapeFactory.getGame("Motor");

shape3.play();

}

}

**2 抽象工厂模式**

创建型模式，一个接口，几个实现类，多个接口工厂类，工厂类各自产生自己的实例，增加子类不需要修改工厂类，只需要扩展子类实现类和子类工厂类即可。做到了开闭原则。

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

package com.hiya.dp.creator.abstractFactory;

public class FightGame implements IGame

{

@Override

public void play()

{

System.out.println("This is FightGame。。。");

}

}

package com.hiya.dp.creator.abstractFactory;

public class FightGameFactory

{

public IGame getGame()

{

return new FightGame();

}

}

package com.hiya.dp.creator.abstractFactory;

public class GameClient

{

public static void main(String[] args)

{

//增加子类不需要修改工厂类，只需要扩展子类实现类和子类工厂类即可，做到了开闭原则

FightGameFactory fightGameFactory = new FightGameFactory();

GunGameFactory gunGameFactory = new GunGameFactory();

MotorGameFactory motorGameFactory = new MotorGameFactory();

// 获取 FightGame 的对象，并调用它的 play 方法

IGame fightGame = fightGameFactory.getGame();

fightGame.play();

// 获取 GunGame 的对象，并调用它的 play 方法

IGame gunGame = gunGameFactory.getGame();

gunGame.play();

// 获取 MotorGame 的对象，并调用它的 play 方法

IGame motorGame = motorGameFactory.getGame();

motorGame.play();

}

}

package com.hiya.dp.creator.abstractFactory;

public class GunGame implements IGame

{

@Override

public void play()

{

System.out.println("This is GunGame。。。");

}

}

package com.hiya.dp.creator.abstractFactory;

public class GunGameFactory

{

public IGame getGame()

{

return new FightGame();

}

}

package com.hiya.dp.creator.abstractFactory;

public interface IGame

{

String name = "Game";

void play();

}

package com.hiya.dp.creator.abstractFactory;

public class MotorGame implements IGame

{

@Override

public void play()

{

System.out.println("This is FightGame。。。");

}

}

package com.hiya.dp.creator.abstractFactory;

public class MotorGameFactory

{

public IGame getGame()

{

return new FightGame();

}

}

**3 单例模式**

（1）创建型模式，枚举创建单例模式线程安全的，一个jvm保证只有一个实例，对于一些大型的对象，这是一笔很大的系统开销，省去了new操作符，降低了系统内存的使用频率，减轻GC压力。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。

单例模式和多例模式：有些系统层级的实例保证一个就够了，业务数据需要多个。

意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

关键代码：构造函数是私有的。

应用实例： 1、一个党只能有一个主席。 2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行。 3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

优点： 1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。 2、避免对资源的多重占用（比如写文件操作）。

缺点：没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

使用场景： 1、要求生产唯一序列号。 2、WEB 中的计数器，不用每次刷新都在数据库里加一次，用单例先缓存起来。 3、创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接等。

注意事项：getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

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：

@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。

@省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

@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。

@提升性能，不用反复创建和销毁实例，多线程情况下注意线程安全。

@（懒汉方式：类加载时不初始化

饿汉方式：在类加载时就完成了初始化，所以类加载较慢，但获取对象的速度快）

（2）readResolve的说明

下面我们先简要地回顾下对象的序列化. 一般来说, 一个类实现了 Serializable接口, 我们就可以把它往内存地写再从内存里读出而"组装"成一个跟原来一模一样的对象. 不过当序列化遇到单例时,这里边就有了个问题: 从内存读出而组装的对象破坏了单例的规则. 单例是要求一个JVM中只有一个类对象的, 而现在通过反序列化,一个新的对象克隆了出来.

public final class MySingleton implements Serializable

{

private MySingleton() { }

private static final MySingleton INSTANCE = new MySingleton();

public static MySingleton getInstance() { return INSTANCE; }

}

　当把 MySingleton对象(通过getInstance方法获得的那个单例对象)序列化后再从内存中读出时, 就有一个全新但跟原来一样的MySingleton对象存在了. 那怎么来维护单例模式呢?这就要用到readResolve方法了. 如下所示:

public final class MySingleton implements Serializable

{

private MySingleton() { }

private static final MySingleton INSTANCE = new MySingleton();

public static MySingleton getInstance() { return INSTANCE; }

private Object readResolve() throws ObjectStreamException

{

return INSTANCE;

}

}

这样当JVM从内存中反序列化地"组装"一个新对象时,就会自动调用这个 readResolve方法来返回我们指定好的对象了, 单例规则也就得到了保证.

## （3）实现方式

**1、懒汉式，线程不安全**

是否 Lazy 初始化：是

是否多线程安全：否

实现难度：易

描述：这种方式是最基本的实现方式，这种实现最大的问题就是不支持多线程。因为没有加锁 synchronized，所以严格意义上它并不算单例模式。

这种方式 lazy loading 很明显，不要求线程安全，在多线程不能正常工作。

代码实例：

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

接下来介绍的几种实现方式都支持多线程，但是在性能上有所差异。

**2、懒汉式，线程安全**

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：易

描述：这种方式具备很好的 lazy loading，能够在多线程中很好的工作，但是，效率很低，99% 情况下不需要同步。

优点：第一次调用才初始化，避免内存浪费。

缺点：必须加锁 synchronized 才能保证单例，但加锁会影响效率。

getInstance() 的性能对应用程序不是很关键（该方法使用不太频繁）。

代码实例：

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

**3、饿汉式**

是否 Lazy 初始化：否

是否多线程安全：是

实现难度：易

描述：这种方式比较常用，但容易产生垃圾对象。

优点：没有加锁，执行效率会提高。

缺点：类加载时就初始化，浪费内存。

它基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。

代码实例：

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

**4、双检锁/双重校验锁（DCL，即 double-checked locking）**

JDK 版本：JDK1.5 起

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：较复杂

描述：这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

getInstance() 的性能对应用程序很关键。

代码实例：

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

**5、登记式/静态内部类**

是否 Lazy 初始化：是

是否多线程安全：是

实现难度：一般

描述：这种方式能达到双检锁方式一样的功效，但实现更简单。对静态域使用延迟初始化，应使用这种方式而不是双检锁方式。这种方式只适用于静态域的情况，双检锁方式可在实例域需要延迟初始化时使用。

这种方式同样利用了 classloder 机制来保证初始化 instance 时只有一个线程，它跟第 3 种方式不同的是：第 3 种方式只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有显示通过调用 getInstance 方法时，才会显示装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，所以想让它延迟加载，另外一方面，又不希望在 Singleton 类加载时就实例化，因为不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第 3 种方式就显得很合理。

代码实例：

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

**6 通过enum关键字来实现枚举，在枚举中需要注意的有：**

1. 序列化后可以保持单例

2. 枚举中可以和java类一样定义方法

3. 枚举中的构造方法必须是私有的

4. 枚举默认是线程安全的

**package** com.hiya.dp.creator.singleton;

**public** **class** EnumSingleton

{

**public** **static** EnumSingleton getSigleInstance()

{

**return** SigleInstance.***INSTANCE***.instance;

}

**public** **enum** SigleInstance

{

***INSTANCE***;

**private** EnumSingleton instance;

SigleInstance()

{

instance = **new** EnumSingleton();

}

**public** EnumSingleton getInstance()

{

**return** instance;

}

}

}

上面的类Resource是我们要应用单例模式的资源，具体可以表现为网络连接，数据库连接，线程池等等。

获取资源的方式很简单，只要 SomeThing.INSTANCE.getInstance() 即可获得所要实例。下面我们来看看单例是如何被保证的：

**4 建造者模式（Builder）**

建造者模式：是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。

应用实例：

1、台式机，笔记本，平板各自有构建主机，构建输入输出，构建连接线的步骤产生对象。

2、JAVA 中的 StringBuilder。

使用场景：

1、需要生成的对象具有复杂的内部结构。

2、需要生成的对象内部属性本身相互依赖。

与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

package com.hiya.dp.creator.builder;

/\*\*

\* 笔记本

\* @author zjq

\*

\*/

public class BookComputerBuilder implements IComputerBuilder

{

Computer computer;

public BookComputerBuilder() {

computer = new Computer();

}

@Override

public void buildHostMachine()

{

computer.setHostMachine("笔记本组装主机");

System.out.println("笔记本组装主机。。。");

}

@Override

public void buildIoDevice()

{

computer.setIoDevice("笔记本组装IO设备");

System.out.println("笔记本组装IO设备。。。");

}

@Override

public void buildConnLine()

{

computer.setIoDevice("笔记本组装连接线");

System.out.println("笔记本组装连接线。。。");

}

@Override

public Computer buildComputer()

{

return computer;

}

}

package com.hiya.dp.creator.builder;

public class Computer

{

//主机

private String hostMachine;

//io设备

private String ioDevice;

//连接线

private String connLine;

public String getHostMachine()

{

return hostMachine;

}

public void setHostMachine(String hostMachine)

{

this.hostMachine = hostMachine;

}

public String getIoDevice()

{

return ioDevice;

}

public void setIoDevice(String ioDevice)

{

this.ioDevice = ioDevice;

}

public String getConnLine()

{

return connLine;

}

public void setConnLine(String connLine)

{

this.connLine = connLine;

}

}

package com.hiya.dp.creator.builder;

public class ComputerClient

{

public static void main(String[] args)

{

ComputerDirector pd = new ComputerDirector();

Computer padComputer = pd.constructComputer(new PadComputerBuilder());

System.out.println(padComputer.getHostMachine());

System.out.println(padComputer.getIoDevice());

System.out.println(padComputer.getConnLine());

Computer tableComputer = pd.constructComputer(new TableComputerBuilder());

System.out.println(tableComputer.getHostMachine());

System.out.println(tableComputer.getIoDevice());

System.out.println(tableComputer.getConnLine());

Computer bookComputer = pd.constructComputer(new BookComputerBuilder());

System.out.println(bookComputer.getHostMachine());

System.out.println(bookComputer.getIoDevice());

System.out.println(bookComputer.getConnLine());

}

}

package com.hiya.dp.creator.builder;

/\*\*

\* 调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建

\* @author zjq

\*

\*/

public class ComputerDirector

{

public Computer constructComputer(IComputerBuilder pb) {

//按照 主机--->输入输出--->连接线 的顺序创建电脑

pb.buildHostMachine();

pb.buildIoDevice();

pb.buildConnLine();

return pb.buildComputer();

}

}

package com.hiya.dp.creator.builder;

/\*\*

\* 给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建

\* @author zjq

\*

\*/

public interface IComputerBuilder

{

void buildHostMachine();

void buildIoDevice();

void buildConnLine();

Computer buildComputer();

}

package com.hiya.dp.creator.builder;

/\*\*

\* 平板机

\* @author zjq

\*实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例

\*/

public class PadComputerBuilder implements IComputerBuilder

{

Computer computer;

public PadComputerBuilder() {

computer = new Computer();

}

@Override

public void buildHostMachine()

{

computer.setHostMachine("平板机组装主机");

System.out.println("平板机组装主机。。。");

}

@Override

public void buildIoDevice()

{

computer.setIoDevice("平板机组装IO设备");

System.out.println("平板机组装IO设备。。。");

}

@Override

public void buildConnLine()

{

computer.setIoDevice("平板机组装连接线");

System.out.println("平板机组装连接线。。。");

}

@Override

public Computer buildComputer()

{

return computer;

}

}

package com.hiya.dp.creator.builder;

/\*\*

\* 台式机

\* @author zjq

\*

\*/

public class TableComputerBuilder implements IComputerBuilder

{

Computer computer;

public TableComputerBuilder() {

computer = new Computer();

}

@Override

public void buildHostMachine()

{

computer.setHostMachine("台式机组装主机");

System.out.println("台式机组装主机。。。");

}

@Override

public void buildIoDevice()

{

computer.setIoDevice("台式机组装IO设备");

System.out.println("台式机组装IO设备。。。");

}

@Override

public void buildConnLine()

{

computer.setIoDevice("台式机组装连接线");

System.out.println("台式机组装连接线。。。");

}

@Override

public Computer buildComputer()

{

return computer;

}

}

**5 原型模式（Prototype）**

在Java中原型模式变成clone()方法的使用，由于Java的纯洁的面向对象特性，使得在Java中使用设计模式变得很自然，两者已经几乎是浑然一体了。

（1）为了获取对象的一份拷贝，我们可以利用Object类的clone()方法。

（2）在派生类中覆盖基类的clone()方法，并声明为public。

（3）在派生类的clone()方法中，调用super.clone()。

（4）在派生类中实现Cloneable接口。

在Java中，clone()方法是浅复制。

**浅复制（浅克隆） ：**被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

**深复制（深克隆）：**被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。可以利用串行化来做深复制，所谓对象序列化就是将对象的状态转换成字节流，以后可以通过这些值再生成相同状态的对象

package com.hiya.dp.creator.prototype;

import java.io.Serializable;

public class PersonalInfo implements Serializable

{

private static final long serialVersionUID = 546543645808271L;

private String name;

private String sex;

private Integer age;

public String getName()

{

return name;

}

public void setName(String name)

{

this.name = name;

}

public String getSex()

{

return sex;

}

public void setSex(String sex)

{

this.sex = sex;

}

public Integer getAge()

{

return age;

}

public void setAge(Integer age)

{

this.age = age;

}

}

package com.hiya.dp.creator.prototype;

import java.io.IOException;

public class ProtoTypeClient

{

public static void main(String[] args) throws ClassNotFoundException, IOException

{

Resume a = new Resume();

a.setPersonalInfo("大鸟", "男", 29);

a.setWorkExperience("1998-2000", "xx公司");

// 浅复制

Resume b = (Resume) a.clone();

b.setWorkExperience("1998-2006", "yy公司");

b.setPersonalInfo("大鸟2", "男", 25);

/\*

\* 大鸟2 男 25

工作经历：1998-2006 yy公司

大鸟2 男 25

工作经历：1998-2006 yy公司

\*/

a.display();

b.display();

Resume c = new Resume();

c.setPersonalInfo("大鸟3", "男", 29);

c.setWorkExperience("1998-2000", "xx公司");

// 深复制

Resume d = (Resume) c.deepClone();

d.setWorkExperience("1998-2006", "yy公司");

d.setPersonalInfo("大鸟4", "男", 25);

/\*

\* 大鸟3 男 29

工作经历：1998-2000 xx公司

大鸟4 男 25

工作经历：1998-2006 yy公司

\*/

c.display();

d.display();

}

}

package com.hiya.dp.creator.prototype;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class Resume implements Serializable, Cloneable

{

private static final long serialVersionUID = 8614140644886700330L;

private PersonalInfo personalInfo;

private WorkExperience workExperience;

public Resume()

{

personalInfo = new PersonalInfo();

workExperience = new WorkExperience();

}

// 设置个人信息

public void setPersonalInfo(String name,String sex, Integer age)

{

personalInfo.setName(name);

personalInfo.setSex(sex);

personalInfo.setAge(age);

}

// 设置工作经历

public void setWorkExperience(String workDate, String company)

{

workExperience.setWorkDate(workDate);

workExperience.setCompany(company);

}

// 显示

public void display()

{

System.out.println(String.format("%s %s %s", personalInfo.getName(), personalInfo.getSex(), personalInfo.getAge()));

System.out.println(String.format("工作经历：%s %s", workExperience.getWorkDate(), workExperience.getCompany()));

}

// 浅复制

@Override

public Object clone()

{

Object obj = null;

try

{

obj = super.clone();

} catch (CloneNotSupportedException e)

{

e.printStackTrace();

}

return obj;

}

// 深复制

public Object deepClone() throws IOException, ClassNotFoundException

{

// 将对象写到流里

ByteArrayOutputStream bo = new ByteArrayOutputStream();

ObjectOutputStream oo = new ObjectOutputStream(bo);

oo.writeObject(this);

// 从流里读出来

ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());

ObjectInputStream oi = new ObjectInputStream(bi);

return (oi.readObject());

}

}

package com.hiya.dp.creator.prototype;

import java.io.Serializable;

/\*\*

\* 工作经历类

\*

\*/

public class WorkExperience implements Serializable

{

private static final long serialVersionUID = -5461562344310808271L;

private String workDate;

private String company;

public String getWorkDate()

{

return workDate;

}

public void setWorkDate(String workDate)

{

this.workDate = workDate;

}

public String getCompany()

{

return company;

}

public void setCompany(String company)

{

this.company = company;

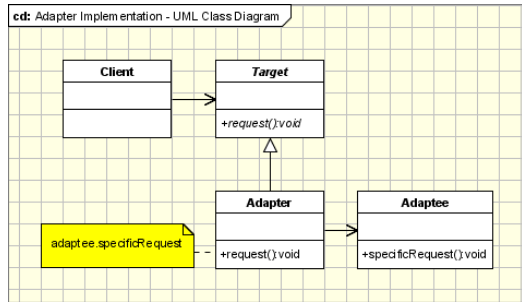
}

}

**结构型模式**

**6 适配器模式**

结构型模式, 在不修改原来两个模块代码的情况下,将两个不兼容的类融合在一起,有点像粘合剂,通过转换使得他们能够协作起来, 符合了开闭原则.将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。



**类适配器通过继承**,是静态的定义方式，当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可

**对象适配器通过代理,是动态组合**的方式。当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

**接口配器，当不希望实现一个接口中所有的方法**时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

package com.hiya.dp.structrue.adapter;

public abstract class AbstractVoltInterfaceAdapter implements IOriginalVolt{

@Override

public void SSH(){};

@Override

public void NET(){};

@Override

public void Tomcat(){};

@Override

public void Mysql(){};

@Override

public void Oracle(){};

@Override

public void FTP(){};

}

package com.hiya.dp.structrue.adapter;

public class AdapterClient

{

public static void main(String[] args)

{

//类适配器：通过继承,是静态的定义方式，当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口

VoltClassAdapter voltClassAdapter = new VoltClassAdapter();

System.out.println(voltClassAdapter.getVolt110());

System.out.println(voltClassAdapter.getVolt220());

//对象适配器：通过代理,是动态组合的方式。当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法

VoltObjectAdapter voltObjectAdapter = new VoltObjectAdapter(new VoltChina());

System.out.println(voltObjectAdapter.getVolt110());

System.out.println(voltObjectAdapter.getVolt220());

//接口适配器：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可

VoltInterfaceTargetAdapter voltInterfaceTargetAdapter = new VoltInterfaceTargetAdapter();

voltInterfaceTargetAdapter.FTP();

}

}

package com.hiya.dp.structrue.adapter;

public interface IOriginalVolt

{

// 远程SSH端口22

public void SSH();

// 网络端口80

public void NET();

// Tomcat容器端口8080

public void Tomcat();

// Mysql数据库端口3306

public void Mysql();

// Oracle数据库端口1521

public void Oracle();

// 文件传输FTP端口21

public void FTP();

}

package com.hiya.dp.structrue.adapter;

public interface IVoltUsa

{

public int getVolt110();

}

package com.hiya.dp.structrue.adapter;

public class VoltChina

{

public int getVolt220()

{

return 220;

}

}

package com.hiya.dp.structrue.adapter;

public class VoltClassAdapter extends VoltChina implements IVoltUsa

{

@Override

public int getVolt110()

{

return 110;

}

}

package com.hiya.dp.structrue.adapter;

public class VoltInterfaceTargetAdapter extends AbstractVoltInterfaceAdapter

{

@Override

public void FTP()

{

System.out.println("Hello world!");

};

}

package com.hiya.dp.structrue.adapter;

public class VoltObjectAdapter implements IVoltUsa

{

//持有原类的一个实例，在Wrapper类的方法中，调用实例的方法

VoltChina voltChina;

public VoltObjectAdapter(VoltChina voltChina) {

this.voltChina = voltChina;

}

@Override

public int getVolt110()

{

return 110;

}

public int getVolt220()

{

return voltChina.getVolt220();

}

}

**7 桥接模式**

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。两个维度变化。

使用场景：

1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。

2、对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

3、一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

注意事项：**对于两个独立变化的维度，使用桥接模式再适合不过了**。

**当一个维度平等关系用接口（邮件/短信/系统消息），另外一个维度递进后者依赖的用抽象类（普通/紧急/特急）**

package com.hiya.dp.structrue.bridge;

public class AbstractMessage

{

/\*\*

\* 持有一个实现部分的对象

\*/

IMessage message;

/\*\* \*

\* 构造方法，传入实现部分的对象 \*

\* @param implementor 实现部分的对象

\* \*/

public AbstractMessage(IMessage message)

{

this.message = message;

}

/\*\*

\* \* 发送消息，委派给实现部分的方法 \*

\* @param message 要发送的消息 \*

\* @param toUser 接收人

\* \*/

public void sendMessage(String message, String toUser)

{

this.message.send(message, toUser);

}

}

package com.hiya.dp.structrue.bridge;

public class BridgeClient

{

public static void main(String[] args)

{

//两个维度，其中抽象类持有接口的对象 ，动态

IMessage smsMessage = new SmsMessage();

AbstractMessage abstractMessage = new CommonMessage(smsMessage);

abstractMessage.sendMessage("短信普通审批", "陈总");

smsMessage = new SmsMessage();

abstractMessage = new UrgencyMessage(smsMessage);

abstractMessage.sendMessage("短信加急审批", "陈总");

smsMessage = new SmsMessage();

SpecialUrgencyMessage specialUrgencyMessage = new SpecialUrgencyMessage(smsMessage);

specialUrgencyMessage.sendMessage("短信特急审批", "陈总");

specialUrgencyMessage.watch("455");

IMessage emailMessage = new EmailMessage();

specialUrgencyMessage = new SpecialUrgencyMessage(emailMessage);

specialUrgencyMessage.sendMessage("邮件特急审批", "陈总");

specialUrgencyMessage.watch("455");

}

}

package com.hiya.dp.structrue.bridge;

public class CommonMessage extends AbstractMessage

{

public CommonMessage(IMessage message)

{

super(message);

}

}

package com.hiya.dp.structrue.bridge;

public class EmailMessage implements IMessage

{

@Override

public void send(String message, String toUser)

{

System.out.println(String.format("使用邮件的方法，发送消息 %s 给 %s", message, toUser));

}

}

package com.hiya.dp.structrue.bridge;

public interface IMessage

{

/\*\*

\* 发送消息

\* @param message 发送消息的内容

\* @param toUser 接收人

\*/

void send (String message, String toUser);

}

package com.hiya.dp.structrue.bridge;

public class SmsMessage implements IMessage

{

@Override

public void send(String message, String toUser)

{

System.out.println(String.format("使用短信的方法，发送消息 %s 给 %s", message, toUser));

}

}

package com.hiya.dp.structrue.bridge;

public class SpecialUrgencyMessage extends AbstractMessage

{

public SpecialUrgencyMessage(IMessage message)

{

super(message);

}

@Override

public void sendMessage(String message, String toUser)

{

message = "特急：" + message;

super.sendMessage(message, toUser);

}

/\*\*

\* 扩展它自己的功能，监控某个消息的处理状态

\* @param messageId

\* @return

\*/

public Object watch(String messageId)

{

//根据给出的消息编码查询消息的处理状态，组织成监控的处理状态，然后返回。 return null;

return null;

}

/\*\*

\* 扩展它自己的功能，监控某个消息的处理状态2

\* @param messageId

\* @return

\*/

public Object watch2(String messageId)

{

return null;

}

}

package com.hiya.dp.structrue.bridge;

public class SysNoticeMessage implements IMessage

{

@Override

public void send(String message, String toUser)

{

System.out.println(String.format("使用系统短消息的方法，发送消息 %s 给 %s", message, toUser));

}

}

package com.hiya.dp.structrue.bridge;

public class UrgencyMessage extends AbstractMessage

{

public UrgencyMessage(IMessage message)

{

super(message);

}

@Override

public void sendMessage(String message, String toUser)

{

message = "加急：" + message;

super.sendMessage(message, toUser);

}

/\*\*

\* 扩展它自己的功能，监控某个消息的处理状态

\* @param messageId

\* @return

\*/

public Object watch(String messageId)

{

//根据给出的消息编码查询消息的处理状态，组织成监控的处理状态，然后返回。 return null;

return null;

}

}

# 8过滤器模式

过滤器模式（Filter Pattern）或标准模式（Criteria Pattern）是一种设计模式，这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。**这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准**

**package** com.hiya.dp.structrue.filter;

**public** **class** Person

{

**private** String name;

**private** String gender;

**private** String maritalStatus;

**public** Person(String name, String gender, String maritalStatus)

{

**this**.name = name;

**this**.gender = gender;

**this**.maritalStatus = maritalStatus;

}

**public** String getName()

{

**return** name;

}

**public** String getGender()

{

**return** gender;

}

**public** String getMaritalStatus()

{

**return** maritalStatus;

}

}

**package** com.hiya.dp.structrue.filter;

**import** java.util.List;

**public** **interface** IFilter

{

**public** List<Person> filterPerson(List<Person> persons);

}

**package** com.hiya.dp.structrue.filter;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FilterMale **implements** IFilter

{

@Override

**public** List<Person> filterPerson(List<Person> persons)

{

List<Person> malePersons = **new** ArrayList<Person>();

**for** (Person person : persons)

{

**if** (person.getGender().equalsIgnoreCase("MALE"))

{

malePersons.add(person);

}

}

**return** malePersons;

}

}

package com.hiya.dp.structrue.filter;

import java.util.ArrayList;

import java.util.List;

public class FilterSingle implements IFilter

{

@Override

public List<Person> filterPerson(List<Person> persons)

{

List<Person> malePersons = new ArrayList<Person>();

for (Person person : persons)

{

if(person.getMaritalStatus().equalsIgnoreCase("SINGLE"))

{

malePersons.add(person);

}

}

return malePersons;

}

}

package com.hiya.dp.structrue.filter;

import java.util.ArrayList;

import java.util.List;

public class FilterFemale implements IFilter

{

@Override

public List<Person> filterPerson(List<Person> persons)

{

List<Person> malePersons = new ArrayList<Person>();

for (Person person : persons)

{

if (person.getGender().equalsIgnoreCase("FEMALE"))

{

malePersons.add(person);

}

}

return malePersons;

}

}

**package** com.hiya.dp.structrue.filter;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** FilterClient

{

**public** **static** **void** main(String[] args)

{

List<Person> persons = **new** ArrayList<Person>();

persons.add(**new** Person("Robert", "Male", "Single"));

persons.add(**new** Person("John", "Male", "Married"));

persons.add(**new** Person("Laura", "Female", "Married"));

persons.add(**new** Person("Diana", "Female", "Single"));

persons.add(**new** Person("Mike", "Male", "Single"));

persons.add(**new** Person("Bobby", "Male", "Single"));

IFilter male = **new** FilterMale();

IFilter female = **new** FilterFemale();

IFilter single = **new** FilterSingle();

System.***out***.println("Males: ");

*printPersons*(male.filterPerson(persons));

System.***out***.println("\nFemales: ");

*printPersons*(female.filterPerson(persons));

System.***out***.println("\nSingle Males: ");

*printPersons*(single.filterPerson(persons));

}

**public** **static** **void** printPersons(List<Person> persons)

{

**for** (Person person : persons)

{

System.***out***.println("Person : [ Name : " + person.getName() + ", Gender : " + person.getGender()+ ", Marital Status : " + person.getMaritalStatus() + " ]");

}

}

}

**9组合模式**

组合模式（Composite Pattern），又叫部分整体模式，**是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次**。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

关键代码：树枝内部组合该接口，并且含有内部属性 List，里面放 Component。

步骤 1

创建 Employee 类，该类带有 Employee 对象的列表。

Employee.java

import java.util.ArrayList;

import java.util.List;

public class Employee {

private String name;

private String dept;

private int salary;

private List<Employee> subordinates;

//构造函数

public Employee(String name,String dept, int sal) {

this.name = name;

this.dept = dept;

this.salary = sal;

subordinates = new ArrayList<Employee>();

}

public void add(Employee e) {

subordinates.add(e);

}

public void remove(Employee e) {

subordinates.remove(e);

}

public List<Employee> getSubordinates(){

return subordinates;

}

public String toString(){

return ("Employee :[ Name : "+ name

+", dept : "+ dept + ", salary :"

+ salary+" ]");

}

}

步骤 2

使用 Employee 类来创建和打印员工的层次结构。

CompositePatternDemo.java

public class CompositePatternDemo {

public static void main(String[] args) {

Employee CEO = new Employee("John","CEO", 30000);

Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);

Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);

Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);

CEO.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

//打印该组织的所有员工

System.out.println(CEO);

for (Employee headEmployee : CEO.getSubordinates()) {

System.out.println(headEmployee);

for (Employee employee : headEmployee.getSubordinates()) {

System.out.println(employee);

}

}

}

}

步骤 3

验证输出。

Employee :[ Name : John, dept : CEO, salary :30000 ]

Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]

Employee :[ Name : Richard, dept : Sales, salary :10000 ]

Employee :[ Name : Rob, dept : Sales, salary :10000 ]

Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]

Employee :[ Name : Laura, dept : Marketing, salary :10000 ]

Employee :[ Name : Bob, dept : Marketing, salary :10000 ]

**10装饰器模式**

相似的模式，拦截方法增加前后的逻辑

装饰模式就是给一个对象增加一些新的功能，而且是动态。

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

使用场景： 1、扩展一个类的功能。 2、动态增加功能，动态撤销。

注意事项：可代替继承。

package com.hiya.dp.structrue.decorator;

public class DecoratorClient

{

public static void main(String[] args)

{

IGame game = new FightGame();

GameShengdaDecorator gameShengdaDecorator = new GameShengdaDecorator(game);

gameShengdaDecorator.play();

GameTencentDecorator gameTencentDecorator = new GameTencentDecorator(gameShengdaDecorator);

gameTencentDecorator.play();

}

}

package com.hiya.dp.structrue.decorator;

public class FightGame implements IGame

{

@Override

public void play()

{

System.out.println("This is FightGame。。。");

}

}

package com.hiya.dp.structrue.decorator;

public class GameShengdaDecorator implements IGame

{

private IGame game;

public GameShengdaDecorator(IGame game)

{

super();

this.game=game;

}

@Override

public void play()

{

System.out.println("欢迎盛大游戏。。。");

game.play();

System.out.println("欢迎再次光临盛大游戏。。。");

}

}

package com.hiya.dp.structrue.decorator;

public class GameTencentDecorator implements IGame

{

private IGame game;

public GameTencentDecorator(IGame game)

{

super();

this.game=game;

}

@Override

public void play()

{

System.out.println("欢迎腾讯游戏。。。");

game.play();

System.out.println("欢迎再次光临腾讯游戏。。。");

}

}

package com.hiya.dp.structrue.decorator;

public class GameWangyiDecorator implements IGame

{

private IGame game;

public GameWangyiDecorator(IGame game)

{

super();

this.game=game;

}

@Override

public void play()

{

System.out.println("欢迎网易游戏。。。");

game.play();

System.out.println("欢迎再次光临网易游戏。。。");

}

}

package com.hiya.dp.structrue.decorator;

public interface IGame

{

String name = "Game";

void play();

}

# 11外观模式

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。

**这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。**

主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。**客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可**。

应用实例： 1、去医院看病，可能要去挂号、门诊、划价、取药，让患者或患者家属觉得很复杂，如果有提供接待人员，只让接待人员来处理，就很方便。

优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。

缺点：不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

使用场景： 1、为复杂的模块或子系统提供外界访问的模块。 2、子系统相对独立。 3、预防低水平人员带来的风险。

package com.hiya.dp.structrue.facade;

public class FacadeClient

{

public static void main(String[] args)

{

GameFacade gameFacade = new GameFacade();

gameFacade.playFightGame();

gameFacade.playGunGame();

gameFacade.playMotorGame();

}

}

package com.hiya.dp.structrue.facade;

public class FightGame implements IGame

{

@Override

public void play()

{

System.out.println("This is FightGame。。。");

}

}

package com.hiya.dp.structrue.facade;

public class GameFacade

{

private IGame fightGame;

private IGame gunGame;

private IGame motorGame;

public GameFacade()

{

this.fightGame = new FightGame();

this.gunGame = new GunGame();

this.motorGame = new MotorGame();

}

public void playFightGame()

{

fightGame.play();

}

public void playGunGame()

{

gunGame.play();

}

public void playMotorGame()

{

motorGame.play();

}

}

package com.hiya.dp.structrue.facade;

public class GunGame implements IGame

{

@Override

public void play()

{

System.out.println("This is GunGame。。。");

}

}

package com.hiya.dp.structrue.facade;

public interface IGame

{

String name = "Game";

void play();

}

package com.hiya.dp.structrue.facade;

public class MotorGame implements IGame

{

@Override

public void play()

{

System.out.println("This is FightGame。。。");

}

}

# 12享元模式

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。

享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。我们将通过创建 5 个对象来画出 20 个分布于不同位置的圆来演示这种模式。由于只有 5 种可用的颜色，所以 color 属性被用来检查现有的 Circle 对象。

何时使用： 1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。

如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。

关键代码：用 HashMap 存储这些对象。

应用实例： 1、JAVA 中的 String，如果有则返回，如果没有则创建一个字符串保存在字符串缓存池里面。 2、数据库的数据池。3、int和Integer

优点：大大减少对象的创建，降低系统的内存，使效率提高。

缺点：提高了系统的复杂度，需要分离出外部状态和内部状态，而且外部状态具有固有化的性质，不应该随着内部状态的变化而变化，否则会造成系统的混乱。

package com.hiya.dp.structrue.flyweight;

public class Circle

{

private String color;

private int x;

private int y;

private int radius;

public Circle(String color)

{

this.color = color;

}

public void setX(int x)

{

this.x = x;

}

public void setY(int y)

{

this.y = y;

}

public void setRadius(int radius)

{

this.radius = radius;

}

public void draw()

{

System.out.println("Circle: Draw() [Color : " + color + ", x : " + x + ", y :" + y + ", radius :" + radius);

}

}

package com.hiya.dp.structrue.flyweight;

import java.util.HashMap;

public class CircleFlyweightFactory

{

private static final HashMap<String, Circle> circleMap = new HashMap<String, Circle>();

public static Circle getCircle(String color)

{

Circle circle = (Circle) circleMap.get(color);

if (circle == null)

{

circle = new Circle(color);

circleMap.put(color, circle);

System.out.println("Creating circle of color : " + color);

}

return circle;

}

}

package com.hiya.dp.structrue.flyweight;

public class FlyweightClient

{

private static final String colors[] = { "Red", "Green", "Blue", "White", "Black" };

public static void main(String[] args)

{

for (int i = 0; i < 20; ++i)

{

Circle circle = (Circle) CircleFlyweightFactory.getCircle(getRandomColor());

circle.setX(getRandom());

circle.setY(getRandom());

circle.setRadius(100);

circle.draw();

}

}

private static String getRandomColor()

{

return colors[(int) (Math.random() \* colors.length)];

}

private static int getRandom()

{

return (int) (Math.random() \* 100);

}

}

# 13代理模式

1. **静态代理**

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

在代理模式中，我们创建具有现有对象的对象，以便向外界提供功能接口。

意图：为其他对象提供一种代理以控制对这个对象的访问。

主要解决：在直接访问对象时带来的问题，比如说：要访问的对象在远程的机器上。在面向对象系统中，有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。

应用实例： 1、Windows 里面的快捷方式。 2、猪八戒去找高翠兰结果是孙悟空变的，可以这样理解：把高翠兰的外貌抽象出来，高翠兰本人和孙悟空都实现了这个接口，猪八戒访问高翠兰的时候看不出来这个是孙悟空，所以说孙悟空是高翠兰代理类。 3、买火车票不一定在火车站买，也可以去代售点。 4、一张支票或银行存单是账户中资金的代理。支票在市场交易中用来代替现金，并提供对签发人账号上资金的控制。 5、spring aop。

**和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制或者增强。**

**（2）静态代理和动态代理**

静态代理通常用于对原有业务逻辑的扩充。比如持有二方包的某个类，并调用了其中的某些方法。然后出于某种原因，比如记录日志、打印方法执行时间，但是又不好将这些逻辑写入二方包的方法里。

所以可以创建一个代理类实现和二方方法相同的方法，通过让代理类持有真实对象，然后在原代码中调用代理类方法，来达到添加我们需要业务逻辑的目的。

这其实也就是代理模式的一种实现，通过对真实对象的封装，来实现扩展性。

public interface Action {

public void doSomething();

}

真实对象

public class RealObject implements Action{

public void doSomething() {

System.out.println("do something");

}

}

代理对象

public class Proxy implements Action {

private Action realObject;

public Proxy(Action realObject) {

this.realObject = realObject;

}

public void doSomething() {

System.out.println("proxy do");

realObject.doSomething();

}

}

运行代码

Proxy proxy = new Proxy(new RealObject());

proxy.doSomething();

simple\_proxy.png

这种代理模式也最为简单，就是通过proxy持有realObject的引用，并进行一层封装。

先看看代理模式的优点： 扩展原功能，不侵入原代码。再看看这种代理模式的缺点：

假如有这样一个需求，有十个不同的RealObject，同时我们要去代理的方法是不同的，比要代理方法：doSomething、doAnotherThing、doTwoAnotherThing，添加代理前，原代码可能是这样的：

realObject.doSomething();

realObject1.doAnotherThing();

realObject2.doTwoAnother();

为了解决这个问题，我们有方案一：

为这些方法创建不同的代理类，代理后的代码是这样的：

proxy.doSomething();

proxy1.doAnotherThing();

proxy2.doTwoAnother();

当然，也有方案二：

通过创建一个proxy，持有不同的realObject，实现Action1、Action2、Action3接口，来让代码变成这样：

proxy.doSomething();

proxy.doAnotherThing();

proxy.doTwoAnother();

毫无疑问，仅仅为了扩展同样的功能，在方案一种，我们会重复创建多个逻辑相同，仅仅RealObject引用不同的Proxy。

而在方案二中，会导致proxy的膨胀，而且这种膨胀往往是无意义的。此外，假如方法签名是相同的，更需要在调用的时候引入额外的判断逻辑。

搞清楚静态代理的缺点十分重要，因为动态代理的目的就是为了解决静态代理的缺点。通过使用动态代理，我们可以通过在运行时，动态生成一个持有RealObject、并实现代理接口的Proxy，

同时注入我们相同的扩展逻辑。哪怕你要代理的RealObject是不同的对象，甚至代理不同的方法，都可以动过动态代理，来扩展功能。

public class DynamicProxyHandler implements InvocationHandler

{

private Object realObject;

public DynamicProxyHandler(Object realObject) {

this.realObject = realObject;

}

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

//代理扩展逻辑

System.out.println("proxy do");

return method.invoke(realObject, args);

}

}

这个Handler中的invoke方法中实现了代理类要扩展的公共功能。

到这里，需要先看一下这个handler的用法：

public static void main(String[] args) {

RealObject realObject = new RealObject();

**Action proxy =**

**(Action) Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(), new Class[]{Action.class}, new DynamicProxyHandler(realObject));**

proxy.doSomething();

}

Proxy.newProxyInstance 传入的是一个ClassLoader， 一个代理接口，和我们定义的handler，返回的是一个Proxy的实例。

仔细体会这个过程，其实有点类似我们在静态代理中提到的方案一，生成了一个包含我们扩展功能，持有RealObject引用，实现Action接口的代理实例Proxy。**只不过这个Proxy不是我们自己写的，而是java帮我们生成的**，有没有一点动态的味道。

**行为型模式**

**14 责任链模式**

顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

关键代码：Handler 里面聚合它自己，在 HanleRequest 里判断是否合适，如果没达到条件则向下传递，向谁传递之前 set 进去。

应用实例： activiti的下一步

使用场景： 1、有多个对象可以处理同一个请求，具体哪个对象处理该请求由运行时刻自动确定。 2、在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。 3、可动态指定一组对象处理请求。

注意事项：在 JAVA WEB 中遇到很多应用。

package com.hiya.dp.behavior.chain;

public abstract class AbstractLogger

{

public static int INFO = 1;

public static int DEBUG = 2;

public static int ERROR = 3;

protected int level;

// 责任链中的下一个元素

protected AbstractLogger nextLogger;

public void setNextLogger(AbstractLogger nextLogger)

{

this.nextLogger = nextLogger;

}

public void logMessage(int level, String message)

{

if (this.level <= level)

{

write(message);

}

if (nextLogger != null)

{

nextLogger.logMessage(level, message);

}

}

abstract protected void write(String message);

}

package com.hiya.dp.behavior.chain;

public class ChainClient

{

public static void main(String[] args)

{

AbstractLogger loggerChain = ChainFactory.getChainOfLoggers();

loggerChain.logMessage(AbstractLogger.INFO, "This is an information.");

loggerChain.logMessage(AbstractLogger.DEBUG, "This is an debug level information.");

loggerChain.logMessage(AbstractLogger.ERROR, "This is an error information.");

}

}

package com.hiya.dp.behavior.chain;

public class ChainFactory

{

public static AbstractLogger getChainOfLoggers()

{

AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);

AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);

AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);

errorLogger.setNextLogger(fileLogger);

fileLogger.setNextLogger(consoleLogger);

return errorLogger;

}

}

package com.hiya.dp.behavior.chain;

public class ConsoleLogger extends AbstractLogger

{

public ConsoleLogger(int level)

{

this.level = level;

}

@Override

protected void write(String message)

{

System.out.println("Standard Console::Logger: " + message);

}

}

package com.hiya.dp.behavior.chain;

public class ErrorLogger extends AbstractLogger

{

public ErrorLogger(int level)

{

this.level = level;

}

@Override

protected void write(String message)

{

System.out.println("Error Console::Logger: " + message);

}

}

package com.hiya.dp.behavior.chain;

public class FileLogger extends AbstractLogger

{

public FileLogger(int level)

{

this.level = level;

}

@Override

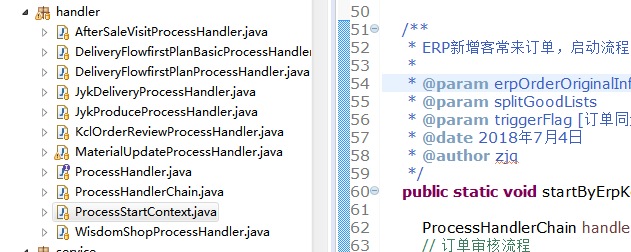
protected void write(String message)

{

System.out.println("File::Logger: " + message);

}

}



**public** **interface** ProcessHandler

{

String ***FIRST*** = "isFist";

**void** start(List<SplitGoodForm> request, ErpOrderOriginalInfo erpOrderOriginalInfo, Map<String, Object> response, ProcessHandlerChain chain);

**public** **static** ProcessHandler create(**final** Supplier<ProcessHandler> supplier)

{

**return** supplier.get();

}

}

public class ProcessHandlerChain implements ProcessHandler {

List<ProcessHandler> filters = new ArrayList<ProcessHandler>();

int index = 0;

public ProcessHandlerChain addHandler(ProcessHandler f) {

filters.add(f);

return this;

}

public void start(List<SplitGoodForm> request, ErpOrderOriginalInfo erpOrderOriginalInfo, Map<String, Object> response,

ProcessHandlerChain chain) {

if (index == filters.size()) {

return;

}

ProcessHandler f = filters.get(index);

index++;

f.start(request, erpOrderOriginalInfo, response, chain);

}

}

/\*\*

\* 流程启动上下文

\*

\* **@author** Ejon

\* **@date** 2018年7月4日

\*/

**public** **class** ProcessStartContext {

/\*\*

\* 贝虎订单同步/Erp新增Jyk订单，启动流程

\*

\* **@param** erpOrderOriginalInfo

\* **@param** splitGoodLists

\* **@param** triggerFlag [订单同步触发 true / ERP新增 false]

\* **@date** 2018年7月4日

\* **@author** zjq

\*/

**public** **static** **void** startByBeihuOrErpJykOrder(ErpOrderOriginalInfo erpOrderOriginalInfo, List<SplitGoodForm> splitGoodLists, **boolean** triggerFlag) {

ProcessHandlerChain handlerChain = **new** ProcessHandlerChain();

// 聚引客生产

handlerChain.addHandler(ProcessHandler.*create*(JykProduceProcessHandler::**new**))

// 聚引客交付

.addHandler(ProcessHandler.*create*(JykDeliveryProcessHandler::**new**))

// 首次营销策划

.addHandler(ProcessHandler.*create*(DeliveryFlowfirstPlanProcessHandler::**new**))

// 首次营销策划(基础版)

.addHandler(ProcessHandler.*create*(DeliveryFlowfirstPlanBasicProcessHandler::**new**))

// 售后上门培训付费

.addHandler(ProcessHandler.*create*(AfterSaleVisitProcessHandler::**new**))

// 智慧餐厅

.addHandler(ProcessHandler.*create*(WisdomShopProcessHandler::**new**));

Map<String, Object> vars = Maps.*newHashMap*();

vars.put(ProcessHandler.***FIRST***, triggerFlag);

handlerChain.start(splitGoodLists, erpOrderOriginalInfo, vars, handlerChain);

}

/\*\*

\* ERP新增客常来订单，启动流程

\*

\* **@param** erpOrderOriginalInfo

\* **@param** splitGoodLists

\* **@param** triggerFlag [订单同步触发 true / ERP新增 false]

\* **@date** 2018年7月4日

\* **@author** zjq

\*/

**public** **static** **void** startByErpKclOrder(ErpOrderOriginalInfo erpOrderOriginalInfo) {

ProcessHandlerChain handlerChain = **new** ProcessHandlerChain();

// 订单审核流程

handlerChain.addHandler(ProcessHandler.*create*(KclOrderReviewProcessHandler::**new**));

handlerChain.start(**null**, erpOrderOriginalInfo, **null**, handlerChain);

}

}

**public** **class** AfterSaleVisitProcessHandler **implements** ProcessHandler

**public** **class** DeliveryFlowfirstPlanBasicProcessHandler **implements** ProcessHandler

**15 命令模式**

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

主要解决：在软件系统中，行为请求者与行为实现者通常是一种紧耦合的关系，但某些场合，比如需要对行为进行记录、撤销或重做、事务等处理时，这种无法抵御变化的紧耦合的设计就不太合适。

何时使用：在某些场合，比如要对行为进行"记录、撤销/重做、事务"等处理，这种无法抵御变化的紧耦合是不合适的。在这种情况下，**如何将"行为请求者"与"行为实现者"解耦**？将一组行为抽象为对象，可以实现二者之间的松耦合。

**适用情况**

1.系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。

2.系统需要在不同的时间指定请求、将请求排队和执行请求。

3.系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作。

4.系统需要将一组操作组合在一起，即支持宏命令。

角色

Command

定义命令的接口，声明执行的方法。

ConcreteCommand

命令接口实现对象，是“虚”的实现；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。

Receiver

接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

Invoker

要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。

Client

创建具体的命令对象，并且设置命令对象的接收者。注意这个不是我们常规意义上的客户端，而是在组装命令对象和接收者，或许，把这个Client称为装配者会更好理解，因为真正使用命令的客户端是从Invoker来触发执行。

package com.hiya.dp.behavior.command;

/\*\*

\* 命令接口

\* @author zjq

\*

\*/

public interface Command<T>

{

T execute(CommandContext commandContext);

}

package com.hiya.dp.behavior.command;

/\*\*

\* 启动流程命令，持有能干事的 对象，调用其中的方法

\* @author zjq

\*

\*/

public class CommandChangeChanel implements Command<String>

{

private int channel;

public CommandChangeChanel(int channel)

{

this.channel = channel;

}

public String execute(CommandContext commandContext)

{

commandContext.changeChannel(channel);

return null;

}

}

package com.hiya.dp.behavior.command;

public class CommandClient

{

/\*\*

\* 模拟activiti的命令模式

\* @param args

\*/

public static void main(String[] args)

{

//初始化命令执行器 ，注入命令上下文

ICommandExecutor CommandExecutor = new CommandExecutorImpl(new CommandContext());

//传参不同的 命令

CommandExecutor.excute(new CommandOn());

CommandExecutor.excute(new CommandOff());

CommandExecutor.excute(new CommandChangeChanel(3));

}

}

package com.hiya.dp.behavior.command;

/\*\*

\* 接受者命令上下文，能够干事的，相当于runtimeService的 启动流程，终止，暂停 ，删除流程操作

\* @author zjq

\*

\*/

public class CommandContext

{

public int currentChannel = 0;

public void turnOn()

{

System.out.println("The televisino is on.");

}

public void turnOff()

{

System.out.println("The television is off.");

}

public void changeChannel(int channel)

{

this.currentChannel = channel;

System.out.println("Now TV channel is " + channel);

}

}

package com.hiya.dp.behavior.command;

public class CommandExecutorImpl implements ICommandExecutor

{

private CommandContext commandContext;

public CommandExecutorImpl(CommandContext commandContext)

{

this.commandContext = commandContext;

}

@Override

public void excute(Command<?> command)

{

command.execute( commandContext);

}

}

package com.hiya.dp.behavior.command;

/\*\*

\* 启动流程命令，持有能干事的 对象，调用其中的方法

\* @author zjq

\*

\*/

public class CommandOff implements Command<String>

{

public String execute(CommandContext commandContext)

{

commandContext.turnOff();

return null;

}

}

package com.hiya.dp.behavior.command;

/\*\*

\* 启动流程命令，持有能干事的 对象，调用其中的方法

\* @author zjq

\*

\*/

public class CommandOn implements Command<String>

{

public String execute(CommandContext commandContext)

{

commandContext.turnOn();

return null;

}

}

package com.hiya.dp.behavior.command;

public interface ICommandExecutor

{

public void excute(Command<?> command);

}

**16解释器模式**

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

意图：给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

主要解决：对于一些固定文法构建一个解释句子的解释器。

何时使用：如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

如何解决：构件语法树，定义终结符与非终结符。

关键代码：构件环境类，包含解释器之外的一些全局信息，一般是 HashMap。

应用实例：编译器、运算表达式计算。

优点： 1、可扩展性比较好，灵活。 2、增加了新的解释表达式的方式。 3、易于实现简单文法。

缺点： 1、可利用场景比较少。 2、对于复杂的文法比较难维护。 3、解释器模式会引起类膨胀。 4、解释器模式采用递归调用方法。

使用场景： 1、可以将一个需要解释执行的语言中的句子表示为一个抽象语法树。 2、一些重复出现的问题可以用一种简单的语言来进行表达。 3、一个简单语法需要解释的场景。

注意事项：可利用场景比较少，JAVA 中如果碰到可以用 expression4J 代替。

package com.hiya.dp.behavior.interpret;

public class And implements Expression

{

private Expression left, right;

public And(Expression left, Expression right)

{

this.left = left;

this.right = right;

}

@Override

public boolean interpret(Context ctx)

{

return left.interpret(ctx) && right.interpret(ctx);

}

@Override

public String toString()

{

return "(" + left.toString() + " AND " + right.toString() + ")";

}

}

package com.hiya.dp.behavior.interpret;

public class Constant implements Expression

{

private boolean value;

public Constant(boolean value)

{

this.value = value;

}

@Override

public boolean interpret(Context ctx)

{

return value;

}

}

package com.hiya.dp.behavior.interpret;

import java.util.HashMap;

import java.util.Map;

public class Context

{

private Map<Variable, Boolean> map = new HashMap<Variable, Boolean>();

public void assign(Variable var, boolean value)

{

map.put(var, new Boolean(value));

}

public boolean lookup(Variable var) throws IllegalArgumentException

{

Boolean value = map.get(var);

if (value == null)

{

throw new IllegalArgumentException();

}

return value.booleanValue();

}

}

package com.hiya.dp.behavior.interpret;

public interface Expression

{

boolean interpret(Context ctx);

}

package com.hiya.dp.behavior.interpret;

public class InterpretClient

{

public static void main(String[] args)

{

Context ctx = new Context();

Variable x = new Variable("x");

Variable y = new Variable("y");

Constant c = new Constant(true);

ctx.assign(x, false);

ctx.assign(y, true);

Expression exp = new Or(new And(c,x) , new And(y,new Not(x)));

System.out.println("x=" + x.interpret(ctx));

System.out.println("y=" + y.interpret(ctx));

System.out.println(exp.toString() + "=" + exp.interpret(ctx));

}

}

package com.hiya.dp.behavior.interpret;

public class Not implements Expression

{

private Expression exp;

public Not(Expression exp)

{

this.exp = exp;

}

@Override

public boolean interpret(Context ctx)

{

return !exp.interpret(ctx);

}

@Override

public String toString()

{

return "(Not " + exp.toString() + ")";

}

}

package com.hiya.dp.behavior.interpret;

public class Or implements Expression

{

private Expression left, right;

public Or(Expression left, Expression right)

{

this.left = left;

this.right = right;

}

@Override

public boolean interpret(Context ctx)

{

return left.interpret(ctx) || right.interpret(ctx);

}

@Override

public String toString()

{

return "(" + left.toString() + " OR " + right.toString() + ")";

}

}

package com.hiya.dp.behavior.interpret;

public class Variable implements Expression

{

private String name;

public Variable(String name)

{

this.name = name;

}

@Override

public boolean equals(Object obj)

{

if (obj != null && obj instanceof Variable)

{

return this.name.equals(((Variable) obj).name);

}

return false;

}

@Override

public int hashCode()

{

return this.toString().hashCode();

}

@Override

public String toString()

{

return name;

}

@Override

public boolean interpret(Context ctx)

{

return ctx.lookup(this);

}

}

17 **迭代器模式**

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

迭代器模式属于行为型模式。

意图：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

如何解决：把在元素之间游走的责任交给迭代器，而不是聚合对象。

关键代码：定义接口：hasNext, next。

应用实例：JAVA 中的 iterator。

注意事项：**迭代器模式就是分离了集合对象的遍历行为，抽象出一个迭代器类来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。**

package com.hiya.dp.behavior.iterator;

public class HiyaIterator implements Iterator

{

private List list = null;

private int index;

public HiyaIterator(List list)

{

super();

this.list = list;

}

@Override

public boolean hasNext()

{

if (index >= list.getSize())

{

return false;

} else

{

return true;

}

}

@Override

public Object next()

{

Object object = list.get(index);

index++;

return object;

}

}

package com.hiya.dp.behavior.iterator;

public class HiyaList implements List

{

private Object[] list;

private int size = 0;

private int index = 0;

public HiyaList()

{

index = 0;

size = 0;

list = new Object[100];

}

@Override

public void add(Object obj)

{

list[index++] = obj;

size++;

}

@Override

public Iterator iterator()

{

return new HiyaIterator(this);

}

@Override

public Object get(int index)

{

return list[index];

}

@Override

public int getSize()

{

return size;

}

}

package com.hiya.dp.behavior.iterator;

public interface Iterator

{

public boolean hasNext();

public Object next();

}

package com.hiya.dp.behavior.iterator;

public class IteratorClient

{

public static void main(String[] args)

{

List list = new HiyaList();

list.add("a");

list.add("b");

list.add("c");

list.add("d");

Iterator it = list.iterator();

while (it.hasNext())

{

System.out.println(it.next());

}

}

}

package com.hiya.dp.behavior.iterator;

public interface List

{

public void add(Object obj);

public Object get(int index);

public Iterator iterator();

public int getSize();

}

18 **中介者模式**

中介者模式（Mediator Pattern）**是用来降低多个对象和类之间的通信复杂性**。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

主要解决：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

应用实例： 1、中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。 2、机场调度系统。 3、MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。

优点： 1、降低了类的复杂度，将一对多转化成了一对一。 2、各个类之间的解耦。 3、符合迪米特原则。

中介者就是一个处于众多对象中间，并恰当地处理众多对象之间相互之间的联系的角色。以上代码中只有两个参与者类，但是这些我们都可以根据中介者模式的宗旨进行适当地扩展，即增加参与者类，然后中介者就得担负更加重的任务了，我们看到上面具体中介者类Mediator中的方法比较多而且有点乱。 所以，在解耦参与者类之间的联系的同时，中介者自身也不免任务过重，因为几乎所有的业务逻辑都交代到中介者身上了，可谓是“万众期待”的一个角色了。这就是中介者模式的不足之处了。此外，上面这个代码例子的参与者的属性和方法都是一样的，我们可以抽取一个抽象类出来，减少代码，但是有时候我们根本抽取不了多个“参与者”之间的共性来形成一个抽象类，这也大大增加了中介者模式的使用难度。

package com.hiya.dp.behavior.mediator;

public class AbstractColleague

{

protected String name;

protected AbstractMediator mediator;

public AbstractColleague(String name, AbstractMediator mediator)

{

this.name = name;

this.mediator = mediator;

}

}

package com.hiya.dp.behavior.mediator;

public abstract class AbstractMediator

{

public abstract void contact(String content,AbstractColleague coll);

}

package com.hiya.dp.behavior.mediator;

public class ColleagueA extends AbstractColleague

{

// 具体同事类继承自Colleague,此刻就可以与中介者mediator进行通信了

public ColleagueA(String name, AbstractMediator mediator)

{

super(name, mediator);

}

public void getMessage(String message)

{

System.out.println("同事A" + name + "获得信息" + message);

}

// 同事A与中介者通信

public void contact(String message)

{

mediator.contact(message, this);

}

}

package com.hiya.dp.behavior.mediator;

public class ColleagueB extends AbstractColleague

{

// 具体同事类继承自Colleague,此刻就可以与中介者mediator进行通信了

public ColleagueB(String name, AbstractMediator mediator)

{

super(name, mediator);

}

public void getMessage(String message)

{

System.out.println("同事B" + name + "获得信息" + message);

}

// 同事A与中介者通信

public void contact(String message)

{

mediator.contact(message, this);

}

}

package com.hiya.dp.behavior.mediator;

public class MediatorClient

{

public static void main(String[] args)

{

// 定义中介者

MediatorImpl mediator = new MediatorImpl();

// 定义具体同事类

ColleagueA colleagueA = new ColleagueA("张三", mediator);

ColleagueB colleagueB = new ColleagueB("李四", mediator);

// 中介者知晓每一个具体的Colleague类

mediator.setCollA(colleagueA);

mediator.setCollB(colleagueB);

colleagueA.contact("我是A，我要和同事B说说工作的事情");

colleagueB.contact("我是B,我下午有时间,下午商量吧");

}

}

package com.hiya.dp.behavior.mediator;

public class MediatorImpl extends AbstractMediator

{

ColleagueA collA;

ColleagueB collB;

public ColleagueA getCollA()

{

return collA;

}

public void setCollA(ColleagueA collA)

{

this.collA = collA;

}

public ColleagueB getCollB()

{

return collB;

}

public void setCollB(ColleagueB collB)

{

this.collB = collB;

}

@Override

public void contact(String content, AbstractColleague coll)

{

if (coll == collA)

{

collB.getMessage(content);

} else

{

collA.getMessage(content);

}

}

}

19 **备忘录模式**

备忘录模式（Memento Pattern）保存一个对象的某个状态，以便在适当的时候恢复对象。备忘录模式属于行为型模式。

主要解决：**所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态**。

何时使用：很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态，使得他有"后悔药"可吃。

如何解决：通过一个备忘录类专门存储对象状态。

关键代码：客户不与备忘录类耦合，与备忘录管理类耦合。

应用实例： 1、后悔药。 2、打游戏时的存档。 3、Windows 里的 ctri + z。 4、IE 中的后退。 4、数据库的事务管理。

优点： 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。 2、实现了信息的封装，使得用户不需要关心状态的保存细节。

缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

使用场景： 1、需要保存/恢复数据的相关状态场景。 2、提供一个可回滚的操作。

注意事项： 1、为了符合迪米特原则，还要增加一个管理备忘录的类。 2、为了节约内存，可使用原型模式+备忘录模式。

package com.hiya.dp.behavior.memento;

public class Memento

{

private String state;

public Memento(String state)

{

this.state = state;

}

public String getState()

{

return state;

}

public void setState(String state)

{

this.state = state;

}

}

package com.hiya.dp.behavior.memento;

/\*\*

\* @author zjq

\*

\*/

public class MementoClient

{

public static void main(String[] args)

{

Starter o = new Starter();

Monitor c = new Monitor();

// 改变负责人对象的状态

o.setState("On");

// 创建备忘录对象，并将发起人对象的状态储存起来

c.saveMemento(o.createMemento());

// 修改发起人的状态

o.setState("Off");

// 恢复发起人对象的状态

o.restoreMemento(c.retrieveMemento());

System.out.println(o.getState());

}

}

package com.hiya.dp.behavior.memento;

/\*\*

\* 监控管理类，负责保存状态

\* @author zjq

\*

\*/

public class Monitor

{

private Memento memento;

/\*\*

\* 备忘录的取值方法

\*/

public Memento retrieveMemento()

{

return this.memento;

}

/\*\*

\* 备忘录的赋值方法

\*/

public void saveMemento(Memento memento)

{

this.memento = memento;

}

}

package com.hiya.dp.behavior.memento;

public class Starter

{

private String state;

/\*\*

\* 工厂方法，返回一个新的备忘录对象

\*/

public Memento createMemento()

{

return new Memento(state);

}

/\*\*

\* 将发起人恢复到备忘录对象所记载的状态

\*/

public void restoreMemento(Memento memento)

{

this.state = memento.getState();

}

public String getState()

{

return state;

}

public void setState(String state)

{

this.state = state;

System.out.println("当前状态：" + this.state);

}

}

20 **观察者模式**

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，**当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式**。

意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

应用实例： 1、拍卖的时候，拍卖师观察最高标价，然后通知给其他竞价者竞价。 2、西游记里面悟空请求菩萨降服红孩儿，菩萨洒了一地水招来一个老乌龟，这个乌龟就是观察者，他观察菩萨洒水这个动作。

使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。

注意事项： 1、JAVA 中已经有了对观察者模式的支持类。 2、避免循环引用。 3、如果顺序执行，某一观察者错误会导致系统卡壳，一般采用异步方式。

package com.hiya.dp.behavior.observer;

/\*\*\*

\* 抽象观察者

\* 定义了一个update()方法，当被观察者调用notifyObservers()方法时，观察者的update()方法会被回调。

\* @author jstao

\*

\*/

public interface Observer

{

public void update(String message);

}

package com.hiya.dp.behavior.observer;

/\*\*\*

\* 抽象被观察者接口

\* 声明了添加、删除、通知观察者方法

\* @author jstao

\*/

public interface Observerable

{

public void registerObserver(Observer o);

public void removeObserver(Observer o);

public void notifyObserver();

}

package com.hiya.dp.behavior.observer;

/\*\*

\*

\*/

public class ObserverClient

{

public static void main(String[] args)

{

WechatObserverableImpl server = new WechatObserverableImpl();

//3个观察者，一般是多个用户

Observer userZhang = new UserObserverImpl("ZhangSan");

Observer userLi = new UserObserverImpl("LiSi");

Observer userWang = new UserObserverImpl("WangWu");

server.registerObserver(userZhang);

server.registerObserver(userLi);

server.registerObserver(userWang);

server.setInfomation("PHP是世界上最好用的语言！");

System.out.println("----------------------------------------------");

server.removeObserver(userZhang);

server.setInfomation("JAVA是世界上最好用的语言！");

}

}

package com.hiya.dp.behavior.observer;

/\*\*

\* 观察者

\* 实现了update方法

\* @author jstao

\*/

public class UserObserverImpl implements Observer

{

private String name;

private String message;

public UserObserverImpl(String name) {

this.name = name;

}

@Override

public void update(String message) {

this.message = message;

read();

}

public void read() {

System.out.println(name + " 收到推送消息： " + message);

}

}

package com.hiya.dp.behavior.observer;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 被观察者，也就是微信公众号服务 实现了Observerable接口，对Observerable接口的三个方法进行了具体实现

\*

\* @author jstao

\*

\*/

public class WechatObserverableImpl implements Observerable

{

// 注意到这个List集合的泛型参数为Observer接口，设计原则：面向接口编程而不是面向实现编程

private List<Observer> list;

private String message;

public WechatObserverableImpl()

{

list = new ArrayList<Observer>();

}

@Override

public void registerObserver(Observer o)

{

list.add(o);

}

@Override

public void removeObserver(Observer o)

{

if (!list.isEmpty())

list.remove(o);

}

// 遍历

@Override

public void notifyObserver()

{

for (int i = 0; i < list.size(); i++)

{

Observer oserver = list.get(i);

oserver.update(message);

}

}

public void setInfomation(String s)

{

this.message = s;

System.out.println("微信服务更新消息： " + s);

// 消息更新，通知所有观察者

notifyObserver();

}

}

21 **状态模式**

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。**在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。**

意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

何时使用：代码中包含大量与对象状态有关的条件语句。

如何解决：将各种具体的状态类抽象出来。

关键代码：通常命令模式的接口中只有一个方法。而状态模式的接口中有一个或者多个方法。而且，状态模式的实现类的方法，一般返回值，或者是改变实例变量的值。也就是说，状态模式一般和对象的状态有关。实现类的方法有不同的功能，覆盖接口中的方法。状态模式和命令模式一样，也可以用于消除 if...else 等条件选择语句。

应用实例： 1、打篮球的时候运动员可以有正常状态、不正常状态和超常状态。 2、曾侯乙编钟中，'钟是抽象接口','钟A'等是具体状态，'曾侯乙编钟'是具体环境（Context）。

package com.hiya.dp.behavior.state;

/\*\*

\* @author zjq

\*

\*/

public class CandyMachine

{

State stateInitImpl;

State stateOnReadyImpl;

State stateReadyImpl;

State stateSoldOutImpl;

State stateWinnerImpl;

private State state;

private int count = 0;

public CandyMachine(int count)

{

this.count = count;

stateInitImpl = new StateInitImpl(this);

stateOnReadyImpl = new StateOnReadyImpl(this);

stateReadyImpl = new StateReadyImpl(this);

stateSoldOutImpl = new StateSoldOutImpl(this);

stateWinnerImpl = new StateWinnerImpl(this);

if (count > 0)

{

state = stateOnReadyImpl;

} else

{

state = stateInitImpl;

}

}

public void setState(State state)

{

this.state = state;

}

public void insertCoin()

{

state.insertCoin();

}

public void returnCoin()

{

state.returnCoin();

}

public void turnCrank()

{

state.turnCrank();

state.dispense();

}

void releaseCandy()

{

if (count > 0)

{

count = count - 1;

System.out.println("a candy rolling out!");

}

}

public int getCount()

{

return count;

}

public void printstate()

{

state.printstate();

}

}

有一个糖果机项目，那么我们知道正常一般糖果机提供给用户的行为有这么几种：投入硬币、转动曲柄、退出硬币几种行为；

那么糖果机呢一般有这几中状态，待机状态、持有硬币的准备状态、运行状态即正在售出状态和初始状态 这么几种正常状态。

我们发现处于不同状态的时候，持有的行为是不一样的

package com.hiya.dp.behavior.state;

public interface State

{

public void insertCoin();

public void returnCoin();

public void turnCrank();

public void dispense();

public void printstate();

}

package com.hiya.dp.behavior.state;

public class StateClient

{

public static void main(String[] args)

{

CandyMachine mCandyMachine = new CandyMachine(6);

mCandyMachine.printstate();

mCandyMachine.insertCoin();

mCandyMachine.printstate();

mCandyMachine.turnCrank();

mCandyMachine.printstate();

mCandyMachine.insertCoin();

mCandyMachine.printstate();

mCandyMachine.turnCrank();

mCandyMachine.printstate();

}

}

package com.hiya.dp.behavior.state;

import java.util.Random;

public class StateInitImpl implements State

{

private CandyMachine mCandyMachine;

public StateInitImpl(CandyMachine mCandyMachine)

{

this.mCandyMachine = mCandyMachine;

}

@Override

public void insertCoin()

{

System.out.println("you can't insert another coin!");

}

@Override

public void returnCoin()

{

System.out.println("coin return!");

mCandyMachine.setState(mCandyMachine.stateOnReadyImpl);

}

@Override

public void turnCrank()

{

System.out.println("crank turn...!");

Random ranwinner = new Random();

int winner = ranwinner.nextInt(10);

if (winner == 0)

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

} else

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

}

}

@Override

public void dispense()

{

}

@Override

public void printstate()

{

System.out.println("\*\*\*HasCoin\*\*\*");

}

}

package com.hiya.dp.behavior.state;

import java.util.Random;

public class StateOnReadyImpl implements State

{

private CandyMachine mCandyMachine;

public StateOnReadyImpl(CandyMachine mCandyMachine)

{

this.mCandyMachine = mCandyMachine;

}

@Override

public void insertCoin()

{

System.out.println("you can't insert another coin!");

}

@Override

public void returnCoin()

{

System.out.println("coin return!");

mCandyMachine.setState(mCandyMachine.stateOnReadyImpl);

}

@Override

public void turnCrank()

{

System.out.println("crank turn...!");

Random ranwinner = new Random();

int winner = ranwinner.nextInt(10);

if (winner == 0)

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

} else

{

mCandyMachine.setState(mCandyMachine.stateSoldOutImpl);

}

}

@Override

public void dispense()

{

}

@Override

public void printstate()

{

System.out.println("\*\*\*HasCoin\*\*\*");

}

}

package com.hiya.dp.behavior.state;

import java.util.Random;

public class StateReadyImpl implements State

{

private CandyMachine mCandyMachine;

public StateReadyImpl(CandyMachine mCandyMachine)

{

this.mCandyMachine = mCandyMachine;

}

@Override

public void insertCoin()

{

System.out.println("you can't insert another coin!");

}

@Override

public void returnCoin()

{

System.out.println("coin return!");

mCandyMachine.setState(mCandyMachine.stateOnReadyImpl);

}

@Override

public void turnCrank()

{

System.out.println("crank turn...!");

Random ranwinner = new Random();

int winner = ranwinner.nextInt(10);

if (winner == 0)

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

} else

{

mCandyMachine.setState(mCandyMachine.stateInitImpl);

}

}

@Override

public void dispense()

{

}

@Override

public void printstate()

{

System.out.println("\*\*\*HasCoin\*\*\*");

}

}

package com.hiya.dp.behavior.state;

import java.util.Random;

public class StateSoldOutImpl implements State

{

private CandyMachine mCandyMachine;

public StateSoldOutImpl(CandyMachine mCandyMachine)

{

this.mCandyMachine = mCandyMachine;

}

@Override

public void insertCoin()

{

System.out.println("you can't insert another coin!");

}

@Override

public void returnCoin()

{

System.out.println("coin return!");

mCandyMachine.setState(mCandyMachine.stateOnReadyImpl);

}

@Override

public void turnCrank()

{

System.out.println("crank turn...!");

Random ranwinner = new Random();

int winner = ranwinner.nextInt(10);

if (winner == 0)

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

} else

{

mCandyMachine.setState(mCandyMachine.stateSoldOutImpl);

}

}

@Override

public void dispense()

{

}

@Override

public void printstate()

{

System.out.println("\*\*\*HasCoin\*\*\*");

}

}

package com.hiya.dp.behavior.state;

import java.util.Random;

public class StateWinnerImpl implements State

{

private CandyMachine mCandyMachine;

public StateWinnerImpl(CandyMachine mCandyMachine)

{

this.mCandyMachine = mCandyMachine;

}

@Override

public void insertCoin()

{

System.out.println("you can't insert another coin!");

}

@Override

public void returnCoin()

{

System.out.println("coin return!");

mCandyMachine.setState(mCandyMachine.stateOnReadyImpl);

}

@Override

public void turnCrank()

{

System.out.println("crank turn...!");

Random ranwinner = new Random();

int winner = ranwinner.nextInt(10);

if (winner == 0)

{

mCandyMachine.setState(mCandyMachine.stateWinnerImpl);

} else

{

mCandyMachine.setState(mCandyMachine.stateInitImpl);

}

}

@Override

public void dispense()

{

}

@Override

public void printstate()

{

System.out.println("\*\*\*HasCoin\*\*\*");

}

}

22 **空对象模式**

在空对象模式（Null Object Pattern）中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。**在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方**。

步骤 1

创建一个抽象类。

AbstractCustomer.java

public abstract class AbstractCustomer {

protected String name;

public abstract boolean isNil();

public abstract String getName();

}

步骤 2

创建扩展了上述类的实体类。

RealCustomer.java

public class RealCustomer extends AbstractCustomer {

public RealCustomer(String name) {

this.name = name;

}

@Override

public String getName() {

return name;

}

@Override

public boolean isNil() {

return false;

}

}

NullCustomer.java

public class NullCustomer extends AbstractCustomer {

@Override

public String getName() {

return "Not Available in Customer Database";

}

@Override

public boolean isNil() {

return true;

}

}

步骤 3

创建 CustomerFactory 类。

CustomerFactory.java

public class CustomerFactory {

public static final String[] names = {"Rob", "Joe", "Julie"};

public static AbstractCustomer getCustomer(String name){

for (int i = 0; i < names.length; i++) {

if (names[i].equalsIgnoreCase(name)){

return new RealCustomer(name);

}

}

return new NullCustomer();

}

}

步骤 4

使用 CustomerFactory，基于客户传递的名字，来获取 RealCustomer 或 NullCustomer 对象。

NullPatternDemo.java

public class NullPatternDemo {

public static void main(String[] args) {

AbstractCustomer customer1 = CustomerFactory.getCustomer("Rob");

AbstractCustomer customer2 = CustomerFactory.getCustomer("Bob");

AbstractCustomer customer3 = CustomerFactory.getCustomer("Julie");

AbstractCustomer customer4 = CustomerFactory.getCustomer("Laura");

System.out.println("Customers");

System.out.println(customer1.getName());

System.out.println(customer2.getName());

System.out.println(customer3.getName());

System.out.println(customer4.getName());

}

}

步骤 5

验证输出。

Customers

Rob

Not Available in Customer Database

Julie

Not Available in Customer Database

23 **策略模式**

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

意图：**定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。**

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

**策略模式是对算法的包装，是把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是：“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。**

　　●　环境(Context)角色：持有一个Strategy的引用。

　　●　抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

　　●　具体策略(ConcreteStrategy)角色：包装了相关的算法或行为

package com.hiya.dp.behavior.strategy;

public class BusStrategy implements ICalculateStrategy

{

@Override

public int calculatePrice(int km)

{

if (km <= 6)

{

return 3;

} else if (km > 6 && km < 12)

{

return 4;

} else if (km < 22 && km > 12)

{

return 5;

} else if (km < 32 && km > 22)

{

return 6;

}

return 7;

}

}

package com.hiya.dp.behavior.strategy;

public class Context

{

ICalculateStrategy mStrategy;

public void setStrategy(ICalculateStrategy mStrategy)

{

this.mStrategy = mStrategy;

}

public int calculatePrice(int km)

{

return mStrategy.calculatePrice(km);

}

}

package com.hiya.dp.behavior.strategy;

public interface ICalculateStrategy

{

int calculatePrice(int km);

}

package com.hiya.dp.behavior.strategy;

/\*\*

\* 在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 if...else 所带来的复杂和难以维护。

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是：“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。

　　●　环境(Context)角色：持有一个Strategy的引用。

　　●　抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

　　●　具体策略(ConcreteStrategy)角色：包装了相关的算法或行为

\*

\*/

public class StrategyClient

{

public static void main(String[] args)

{

Context calculator = new Context();

//设置计算策略

calculator.setStrategy(new BusStrategy());

//计算价格

System.out.println("公交车乘10公里的价格：" + calculator.calculatePrice(10));

}

}

package com.hiya.dp.behavior.strategy;

public class SubwayStrategy implements ICalculateStrategy

{

@Override

public int calculatePrice(int km)

{

int extraTotal = km - 10;

int extraFactor = extraTotal / 5;

int fraction = extraTotal % 5;

int price = 1 + extraFactor \* 1;

return fraction > 0 ? ++price : price;

}

}

package com.hiya.dp.behavior.strategy;

public class TaxStrategy implements ICalculateStrategy

{

@Override

public int calculatePrice(int km)

{

int extraTotal = km - 10;

int extraFactor = extraTotal / 4;

int fraction = extraTotal % 6;

int price = 1 + extraFactor \* 1;

return fraction > 0 ? ++price : price;

}

}

24 **模板模式**

在模板模式（Template Pattern）中，**一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。**这种类型的设计模式属于行为型模式。

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

应用实例： 1、在造房子的时候，地基、走线、水管都一样，只有在建筑的后期才有加壁橱加栅栏等差异。 2、西游记里面菩萨定好的 81 难，这就是一个顶层的逻辑骨架。 3、spring 中对 Hibernate 的支持，将一些已经定好的方法封装起来，比如开启事务、获取 Session、关闭 Session 等，程序员不重复写那些已经规范好的代码，直接丢一个实体就可以保存。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

步骤 1

创建一个抽象类，它的模板方法被设置为 final。

Game.java

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

//模板

public final void play(){

//初始化游戏

initialize();

//开始游戏

startPlay();

//结束游戏

endPlay();

}

}

步骤 2

创建扩展了上述类的实体类。

Cricket.java

public class Cricket extends Game {

@Override

void endPlay() {

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

Football.java

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

步骤 3

使用 Game 的模板方法 play() 来演示游戏的定义方式。

TemplatePatternDemo.java

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

System.out.println();

game = new Football();

game.play();

}

}

步骤 4

验证输出。

Cricket Game Initialized! Start playing.

Cricket Game Started. Enjoy the game!

Cricket Game Finished!

Football Game Initialized! Start playing.

Football Game Started. Enjoy the game!

Football Game Finished!

25 **访问者模式**

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

意图：主要将数据结构与数据操作分离。

主要解决：稳定的数据结构和易变的操作耦合问题。

何时使用：需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。

如何解决：在被访问的类里面加一个对外提供接待访问者的接口。

关键代码：在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。

应用实例：您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。

优点： 1、符合单一职责原则。 2、优秀的扩展性。 3、灵活性。

缺点： 1、具体元素对访问者公布细节，违反了迪米特原则。 2、具体元素变更比较困难。 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。

使用场景： 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

注意事项：访问者可以对功能进行统一，可以做报表、UI、拦截器与过滤器。

步骤 1

定义一个表示元素的接口。

ComputerPart.java

public interface ComputerPart {

public void accept(ComputerPartVisitor computerPartVisitor);

}

步骤 2

创建扩展了上述类的实体类。

Keyboard.java

public class Keyboard implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Monitor.java

public class Monitor implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Mouse.java

public class Mouse implements ComputerPart {

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

computerPartVisitor.visit(this);

}

}

Computer.java

public class Computer implements ComputerPart {

ComputerPart[] parts;

public Computer(){

parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};

}

@Override

public void accept(ComputerPartVisitor computerPartVisitor) {

for (int i = 0; i < parts.length; i++) {

parts[i].accept(computerPartVisitor);

}

computerPartVisitor.visit(this);

}

}

步骤 3

定义一个表示访问者的接口。

ComputerPartVisitor.java

public interface ComputerPartVisitor {

public void visit(Computer computer);

public void visit(Mouse mouse);

public void visit(Keyboard keyboard);

public void visit(Monitor monitor);

}

步骤 4

创建实现了上述类的实体访问者。

ComputerPartDisplayVisitor.java

public class ComputerPartDisplayVisitor implements ComputerPartVisitor {

@Override

public void visit(Computer computer) {

System.out.println("Displaying Computer.");

}

@Override

public void visit(Mouse mouse) {

System.out.println("Displaying Mouse.");

}

@Override

public void visit(Keyboard keyboard) {

System.out.println("Displaying Keyboard.");

}

@Override

public void visit(Monitor monitor) {

System.out.println("Displaying Monitor.");

}

}

步骤 5

使用 ComputerPartDisplayVisitor 来显示 Computer 的组成部分。

VisitorPatternDemo.java

public class VisitorPatternDemo {

public static void main(String[] args) {

ComputerPart computer = new Computer();

computer.accept(new ComputerPartDisplayVisitor());

}

}

步骤 6

验证输出。

Displaying Mouse.

Displaying Keyboard.

Displaying Monitor.

Displaying Computer.

**JAVA EE模式**

**26 MVC模式**

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

步骤 1

创建模型。

Student.java

public class Student {

private String rollNo;

private String name;

public String getRollNo() {

return rollNo;

}

public void setRollNo(String rollNo) {

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

步骤 2

创建视图。

StudentView.java

public class StudentView {

public void printStudentDetails(String studentName, String studentRollNo){

System.out.println("Student: ");

System.out.println("Name: " + studentName);

System.out.println("Roll No: " + studentRollNo);

}

}

步骤 3

创建控制器。

StudentController.java

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view){

this.model = model;

this.view = view;

}

public void setStudentName(String name){

model.setName(name);

}

public String getStudentName(){

return model.getName();

}

public void setStudentRollNo(String rollNo){

model.setRollNo(rollNo);

}

public String getStudentRollNo(){

return model.getRollNo();

}

public void updateView(){

view.printStudentDetails(model.getName(), model.getRollNo());

}

}

步骤 4

使用 StudentController 方法来演示 MVC 设计模式的用法。

MVCPatternDemo.java

public class MVCPatternDemo {

public static void main(String[] args) {

//从数据可获取学生记录

Student model = retriveStudentFromDatabase();

//创建一个视图：把学生详细信息输出到控制台

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

//更新模型数据

controller.setStudentName("John");

controller.updateView();

}

private static Student retriveStudentFromDatabase(){

Student student = new Student();

student.setName("Robert");

student.setRollNo("10");

return student;

}

}

步骤 5

验证输出。

Student:

Name: Robert

Roll No: 10

Student:

Name: John

Roll No: 10

**27 业务代表模式**

业务代表模式（Business Delegate Pattern）用于对表示层和业务层解耦。它基本上是用来减少通信或对表示层代码中的业务层代码的远程查询功能。在业务层中我们有以下实体。

客户端（Client） - 表示层代码可以是 JSP、servlet 或 UI java 代码。

业务代表（Business Delegate） - 一个为客户端实体提供的入口类，它提供了对业务服务方法的访问。

查询服务（LookUp Service） - 查找服务对象负责获取相关的业务实现，并提供业务对象对业务代表对象的访问。

业务服务（Business Service） - 业务服务接口。实现了该业务服务的实体类，提供了实际的业务实现逻辑。

步骤 1

创建 BusinessService 接口。

BusinessService.java

public interface BusinessService {

public void doProcessing();

}

步骤 2

创建实体服务类。

EJBService.java

public class EJBService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking EJB Service");

}

}

JMSService.java

public class JMSService implements BusinessService {

@Override

public void doProcessing() {

System.out.println("Processing task by invoking JMS Service");

}

}

步骤 3

创建业务查询服务。

BusinessLookUp.java

public class BusinessLookUp {

public BusinessService getBusinessService(String serviceType){

if(serviceType.equalsIgnoreCase("EJB")){

return new EJBService();

}else {

return new JMSService();

}

}

}

步骤 4

创建业务代表。

BusinessDelegate.java

public class BusinessDelegate {

private BusinessLookUp lookupService = new BusinessLookUp();

private BusinessService businessService;

private String serviceType;

public void setServiceType(String serviceType){

this.serviceType = serviceType;

}

public void doTask(){

businessService = lookupService.getBusinessService(serviceType);

businessService.doProcessing();

}

}

步骤 5

创建客户端。

Client.java

public class Client {

BusinessDelegate businessService;

public Client(BusinessDelegate businessService){

this.businessService = businessService;

}

public void doTask(){

businessService.doTask();

}

}

步骤 6

使用 BusinessDelegate 和 Client 类来演示业务代表模式。

BusinessDelegatePatternDemo.java

public class BusinessDelegatePatternDemo {

public static void main(String[] args) {

BusinessDelegate businessDelegate = new BusinessDelegate();

businessDelegate.setServiceType("EJB");

Client client = new Client(businessDelegate);

client.doTask();

businessDelegate.setServiceType("JMS");

client.doTask();

}

}

步骤 7

验证输出。

Processing task by invoking EJB Service

Processing task by invoking JMS Service

**28 组合实体模式**

组合实体模式（Composite Entity Pattern）用在 EJB 持久化机制中。一个组合实体是一个 EJB 实体 bean，代表了对象的图解。当更新一个组合实体时，内部依赖对象 beans 会自动更新，因为它们是由 EJB 实体 bean 管理的。以下是组合实体 bean 的参与者。

组合实体（Composite Entity） - 它是主要的实体 bean。它可以是粗粒的，或者可以包含一个粗粒度对象，用于持续生命周期。

粗粒度对象（Coarse-Grained Object） - 该对象包含依赖对象。它有自己的生命周期，也能管理依赖对象的生命周期。

依赖对象（Dependent Object） - 依赖对象是一个持续生命周期依赖于粗粒度对象的对象。

策略（Strategies） - 策略表示如何实现组合实体。

步骤 1

创建依赖对象。

DependentObject1.java

public class DependentObject1 {

private String data;

public void setData(String data){

this.data = data;

}

public String getData(){

return data;

}

}

DependentObject2.java

public class DependentObject2 {

private String data;

public void setData(String data){

this.data = data;

}

public String getData(){

return data;

}

}

步骤 2

创建粗粒度对象。

CoarseGrainedObject.java

public class CoarseGrainedObject {

DependentObject1 do1 = new DependentObject1();

DependentObject2 do2 = new DependentObject2();

public void setData(String data1, String data2){

do1.setData(data1);

do2.setData(data2);

}

public String[] getData(){

return new String[] {do1.getData(),do2.getData()};

}

}

步骤 3

创建组合实体。

CompositeEntity.java

public class CompositeEntity {

private CoarseGrainedObject cgo = new CoarseGrainedObject();

public void setData(String data1, String data2){

cgo.setData(data1, data2);

}

public String[] getData(){

return cgo.getData();

}

}

步骤 4

创建使用组合实体的客户端类。

Client.java

public class Client {

private CompositeEntity compositeEntity = new CompositeEntity();

public void printData(){

for (int i = 0; i < compositeEntity.getData().length; i++) {

System.out.println("Data: " + compositeEntity.getData()[i]);

}

}

public void setData(String data1, String data2){

compositeEntity.setData(data1, data2);

}

}

步骤 5

使用 Client 来演示组合实体设计模式的用法。

CompositeEntityPatternDemo.java

public class CompositeEntityPatternDemo {

public static void main(String[] args) {

Client client = new Client();

client.setData("Test", "Data");

client.printData();

client.setData("Second Test", "Data1");

client.printData();

}

}

步骤 6

验证输出。

Data: Test

Data: Data

Data: Second Test

Data: Data1

**29 数据访问对象模式**

数据访问对象模式（Data Access Object Pattern）或 DAO 模式用于把低级的数据访问 API 或操作从高级的业务服务中分离出来。以下是数据访问对象模式的参与者。

数据访问对象接口（Data Access Object Interface） - 该接口定义了在一个模型对象上要执行的标准操作。

数据访问对象实体类（Data Access Object concrete class） - 该类实现了上述的接口。该类负责从数据源获取数据，数据源可以是数据库，也可以是 xml，或者是其他的存储机制。

模型对象/数值对象（Model Object/Value Object） - 该对象是简单的 POJO，包含了 get/set 方法来存储通过使用 DAO 类检索到的数据。

步骤 1

创建数值对象。

Student.java

public class Student {

private String name;

private int rollNo;

Student(String name, int rollNo){

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

}

步骤 2

创建数据访问对象接口。

StudentDao.java

import java.util.List;

public interface StudentDao {

public List<Student> getAllStudents();

public Student getStudent(int rollNo);

public void updateStudent(Student student);

public void deleteStudent(Student student);

}

步骤 3

创建实现了上述接口的实体类。

StudentDaoImpl.java

import java.util.ArrayList;

import java.util.List;

public class StudentDaoImpl implements StudentDao {

//列表是当作一个数据库

List<Student> students;

public StudentDaoImpl(){

students = new ArrayList<Student>();

Student student1 = new Student("Robert",0);

Student student2 = new Student("John",1);

students.add(student1);

students.add(student2);

}

@Override

public void deleteStudent(Student student) {

students.remove(student.getRollNo());

System.out.println("Student: Roll No " + student.getRollNo()

+", deleted from database");

}

//从数据库中检索学生名单

@Override

public List<Student> getAllStudents() {

return students;

}

@Override

public Student getStudent(int rollNo) {

return students.get(rollNo);

}

@Override

public void updateStudent(Student student) {

students.get(student.getRollNo()).setName(student.getName());

System.out.println("Student: Roll No " + student.getRollNo()

+", updated in the database");

}

}

步骤 4

使用 StudentDao 来演示数据访问对象模式的用法。

CompositeEntityPatternDemo.java

public class DaoPatternDemo {

public static void main(String[] args) {

StudentDao studentDao = new StudentDaoImpl();

//输出所有的学生

for (Student student : studentDao.getAllStudents()) {

System.out.println("Student: [RollNo : "

+student.getRollNo()+", Name : "+student.getName()+" ]");

}

//更新学生

Student student =studentDao.getAllStudents().get(0);

student.setName("Michael");

studentDao.updateStudent(student);

//获取学生

studentDao.getStudent(0);

System.out.println("Student: [RollNo : "

+student.getRollNo()+", Name : "+student.getName()+" ]");

}

}

步骤 5

验证输出。

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]

**30 前端控制器模式**

前端控制器模式（Front Controller Pattern）是用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

前端控制器（Front Controller） - 处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 web 的应用程序，也可以是基于桌面的应用程序。

调度器（Dispatcher） - 前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序。

视图（View） - 视图是为请求而创建的对象。

步骤 1

创建视图。

HomeView.java

public class HomeView {

public void show(){

System.out.println("Displaying Home Page");

}

}

StudentView.java

public class StudentView {

public void show(){

System.out.println("Displaying Student Page");

}

}

步骤 2

创建调度器 Dispatcher。

Dispatcher.java

public class Dispatcher {

private StudentView studentView;

private HomeView homeView;

public Dispatcher(){

studentView = new StudentView();

homeView = new HomeView();

}

public void dispatch(String request){

if(request.equalsIgnoreCase("STUDENT")){

studentView.show();

}else{

homeView.show();

}

}

}

步骤 3

创建前端控制器 FrontController。

Context.java

public class FrontController {

private Dispatcher dispatcher;

public FrontController(){

dispatcher = new Dispatcher();

}

private boolean isAuthenticUser(){

System.out.println("User is authenticated successfully.");

return true;

}

private void trackRequest(String request){

System.out.println("Page requested: " + request);

}

public void dispatchRequest(String request){

//记录每一个请求

trackRequest(request);

//对用户进行身份验证

if(isAuthenticUser()){

dispatcher.dispatch(request);

}

}

}

步骤 4

使用 FrontController 来演示前端控制器设计模式。

FrontControllerPatternDemo.java

public class FrontControllerPatternDemo {

public static void main(String[] args) {

FrontController frontController = new FrontController();

frontController.dispatchRequest("HOME");

frontController.dispatchRequest("STUDENT");

}

}

步骤 5

验证输出。

Page requested: HOME

User is authenticated successfully.

Displaying Home Page

Page requested: STUDENT

User is authenticated successfully.

Displaying Student Page

**31 拦截过滤器模式**

拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

过滤器（Filter） - 过滤器在请求处理程序执行请求之前或之后，执行某些任务。

过滤器链（Filter Chain） - 过滤器链带有多个过滤器，并在 Target 上按照定义的顺序执行这些过滤器。

Target - Target 对象是请求处理程序。

过滤管理器（Filter Manager） - 过滤管理器管理过滤器和过滤器链。

客户端（Client） - Client 是向 Target 对象发送请求的对象。

步骤 1

创建过滤器接口 Filter。

Filter.java

public interface Filter {

public void execute(String request);

}

步骤 2

创建实体过滤器。

AuthenticationFilter.java

public class AuthenticationFilter implements Filter {

public void execute(String request){

System.out.println("Authenticating request: " + request);

}

}

DebugFilter.java

public class DebugFilter implements Filter {

public void execute(String request){

System.out.println("request log: " + request);

}

}

步骤 3

创建 Target。

Target.java

public class Target {

public void execute(String request){

System.out.println("Executing request: " + request);

}

}

步骤 4

创建过滤器链。

FilterChain.java

import java.util.ArrayList;

import java.util.List;

public class FilterChain {

private List<Filter> filters = new ArrayList<Filter>();

private Target target;

public void addFilter(Filter filter){

filters.add(filter);

}

public void execute(String request){

for (Filter filter : filters) {

filter.execute(request);

}

target.execute(request);

}

public void setTarget(Target target){

this.target = target;

}

}

步骤 5

创建过滤管理器。

FilterManager.java

public class FilterManager {

FilterChain filterChain;

public FilterManager(Target target){

filterChain = new FilterChain();

filterChain.setTarget(target);

}

public void setFilter(Filter filter){

filterChain.addFilter(filter);

}

public void filterRequest(String request){

filterChain.execute(request);

}

}

步骤 6

创建客户端 Client。

Client.java

public class Client {

FilterManager filterManager;

public void setFilterManager(FilterManager filterManager){

this.filterManager = filterManager;

}

public void sendRequest(String request){

filterManager.filterRequest(request);

}

}

步骤 7

使用 Client 来演示拦截过滤器设计模式。

FrontControllerPatternDemo.java

public class InterceptingFilterDemo {

public static void main(String[] args) {

FilterManager filterManager = new FilterManager(new Target());

filterManager.setFilter(new AuthenticationFilter());

filterManager.setFilter(new DebugFilter());

Client client = new Client();

client.setFilterManager(filterManager);

client.sendRequest("HOME");

}

}

步骤 8

验证输出。

Authenticating request: HOME

request log: HOME

Executing request: HOME

**32 服务器定位模式**

服务定位器模式（Service Locator Pattern）用在我们想使用 JNDI 查询定位各种服务的时候。考虑到为某个服务查找 JNDI 的代价很高，服务定位器模式充分利用了缓存技术。在首次请求某个服务时，服务定位器在 JNDI 中查找服务，并缓存该服务对象。当再次请求相同的服务时，服务定位器会在它的缓存中查找，这样可以在很大程度上提高应用程序的性能。以下是这种设计模式的实体。

服务（Service） - 实际处理请求的服务。对这种服务的引用可以在 JNDI 服务器中查找到。

Context / 初始的 Context - JNDI Context 带有对要查找的服务的引用。

服务定位器（Service Locator） - 服务定位器是通过 JNDI 查找和缓存服务来获取服务的单点接触。

缓存（Cache） - 缓存存储服务的引用，以便复用它们。

客户端（Client） - Client 是通过 ServiceLocator 调用服务的对象。

步骤 1

创建服务接口 Service。

Service.java

public interface Service {

public String getName();

public void execute();

}

步骤 2

创建实体服务。

Service1.java

public class Service1 implements Service {

public void execute(){

System.out.println("Executing Service1");

}

@Override

public String getName() {

return "Service1";

}

}

Service2.java

public class Service2 implements Service {

public void execute(){

System.out.println("Executing Service2");

}

@Override

public String getName() {

return "Service2";

}

}

步骤 3

为 JNDI 查询创建 InitialContext。

InitialContext.java

public class InitialContext {

public Object lookup(String jndiName){

if(jndiName.equalsIgnoreCase("SERVICE1")){

System.out.println("Looking up and creating a new Service1 object");

return new Service1();

}else if (jndiName.equalsIgnoreCase("SERVICE2")){

System.out.println("Looking up and creating a new Service2 object");

return new Service2();

}

return null;

}

}

步骤 4

创建缓存 Cache。

Cache.java

import java.util.ArrayList;

import java.util.List;

public class Cache {

private List<Service> services;

public Cache(){

services = new ArrayList<Service>();

}

public Service getService(String serviceName){

for (Service service : services) {

if(service.getName().equalsIgnoreCase(serviceName)){

System.out.println("Returning cached "+serviceName+" object");

return service;

}

}

return null;

}

public void addService(Service newService){

boolean exists = false;

for (Service service : services) {

if(service.getName().equalsIgnoreCase(newService.getName())){

exists = true;

}

}

if(!exists){

services.add(newService);

}

}

}

步骤 5

创建服务定位器。

ServiceLocator.java

public class ServiceLocator {

private static Cache cache;

static {

cache = new Cache();

}

public static Service getService(String jndiName){

Service service = cache.getService(jndiName);

if(service != null){

return service;

}

InitialContext context = new InitialContext();

Service service1 = (Service)context.lookup(jndiName);

cache.addService(service1);

return service1;

}

}

步骤 6

使用 ServiceLocator 来演示服务定位器设计模式。

ServiceLocatorPatternDemo.java

public class ServiceLocatorPatternDemo {

public static void main(String[] args) {

Service service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

service = ServiceLocator.getService("Service1");

service.execute();

service = ServiceLocator.getService("Service2");

service.execute();

}

}

步骤 7

验证输出。

Looking up and creating a new Service1 object

Executing Service1

Looking up and creating a new Service2 object

Executing Service2

Returning cached Service1 object

Executing Service1

Returning cached Service2 object

Executing Service2

**33 传输对象模式**

传输对象模式（Transfer Object Pattern）用于从客户端向服务器一次性传递带有多个属性的数据。传输对象也被称为数值对象。传输对象是一个具有 getter/setter 方法的简单的 POJO 类，它是可序列化的，所以它可以通过网络传输。它没有任何的行为。服务器端的业务类通常从数据库读取数据，然后填充 POJO，并把它发送到客户端或按值传递它。对于客户端，传输对象是只读的。客户端可以创建自己的传输对象，并把它传递给服务器，以便一次性更新数据库中的数值。以下是这种设计模式的实体。

业务对象（Business Object） - 为传输对象填充数据的业务服务。

传输对象（Transfer Object） - 简单的 POJO，只有设置/获取属性的方法。

客户端（Client） - 客户端可以发送请求或者发送传输对象到业务对象。

步骤 1

创建传输对象。

StudentVO.java

public class StudentVO {

private String name;

private int rollNo;

StudentVO(String name, int rollNo){

this.name = name;

this.rollNo = rollNo;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getRollNo() {

return rollNo;

}

public void setRollNo(int rollNo) {

this.rollNo = rollNo;

}

}

步骤 2

创建业务对象。

StudentBO.java

import java.util.ArrayList;

import java.util.List;

public class StudentBO {

//列表是当作一个数据库

List<StudentVO> students;

public StudentBO(){

students = new ArrayList<StudentVO>();

StudentVO student1 = new StudentVO("Robert",0);

StudentVO student2 = new StudentVO("John",1);

students.add(student1);

students.add(student2);

}

public void deleteStudent(StudentVO student) {

students.remove(student.getRollNo());

System.out.println("Student: Roll No "

+ student.getRollNo() +", deleted from database");

}

//从数据库中检索学生名单

public List<StudentVO> getAllStudents() {

return students;

}

public StudentVO getStudent(int rollNo) {

return students.get(rollNo);

}

public void updateStudent(StudentVO student) {

students.get(student.getRollNo()).setName(student.getName());

System.out.println("Student: Roll No "

+ student.getRollNo() +", updated in the database");

}

}

步骤 3

使用 StudentBO 来演示传输对象设计模式。

TransferObjectPatternDemo.java

public class TransferObjectPatternDemo {

public static void main(String[] args) {

StudentBO studentBusinessObject = new StudentBO();

//输出所有的学生

for (StudentVO student : studentBusinessObject.getAllStudents()) {

System.out.println("Student: [RollNo : "

+student.getRollNo()+", Name : "+student.getName()+" ]");

}

//更新学生

StudentVO student =studentBusinessObject.getAllStudents().get(0);

student.setName("Michael");

studentBusinessObject.updateStudent(student);

//获取学生

studentBusinessObject.getStudent(0);

System.out.println("Student: [RollNo : "

+student.getRollNo()+", Name : "+student.getName()+" ]");

}

}

步骤 4

验证输出。

Student: [RollNo : 0, Name : Robert ]

Student: [RollNo : 1, Name : John ]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael ]