设计模式
- 分类
- 设计模式的六大原则
- 创建型模式
  - 单例模式
    - 特点
    - 懒汉模式
    - 饿汉模式
    - 双重锁模式
    - 静态内部类单例模式
    - 枚举单例模式
    - 总结
  - 工厂模式
    - 简单工厂模式
    - 工厂方法模式
    - 抽象工厂模式
    - 总结
    - 参考
  - 建造者模式
    - 简介
      - 适用场景
      - 主要作用
      - 解决的问题
    - 模式结构
    - 示例代码
    - 参考
  - 原型模式
    - 举例
    - 参考
- 行为型模式
  - 责任链模式
    - 优点
    - 缺点
    - 应用场景
    - 参考
  - 命令模式
    - 实现
    - 参考
  - 解释器模式
    - 实现
    - 参考
  - 迭代器模式
    - 实现
    - 参考
  - 观察者模式
    - 介绍
    - 实现
    - 参考
  - 中介者模式
    - 中介者模式角色组成
    - 实现
      - 程序分析
    - 参考
  - 备忘录模式
    - 实现

        参考
    状态模式
        实现
        参考
    策略模式
        实现
        参考
    模板模式
        实现
        参考
    空对象模式
        实现
    适配器模式
        实现
        参考
    代理模式
        为什么要用代理模式?
        有哪几种代理模式?
            静态代理
            动态代理
            CGLIB代理
        参考
    装饰器模式
        实现
        参考
    桥接模式
        实现
        参考
    组合模式
        实现
        参考
    外观模式
        实现
        参考
    享元模式
        实现
        参考
J2EE 模式
    MVC 模式
        实现
        参考
    业务代表模式
        实现
        参考
    组合实体模式
        实现
        参考
    数据访问对象模式
        实现
        参考
    前端控制器模式
        实现
        参考
    拦截过滤器模式
        实现
        参考
    服务定位器模式
        实现

参考
传输对象模式
实现
参考

设计模式

源码地址：https://github.com/KuroChan1998/Design-Patterns-Demo

博客地址：
http://blog.kurochan.top/2019/12/15/%E8%AE%BE%E8%AE%A1%E6%A8%A1%E5%BC%8F%E5%A4%A7%E5%85%A8/

分类

序号	模式 & 描述	包括
1	**创建型模式**：这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。	工厂模式（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）
2	**结构型模式**：这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。	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）
3	**行为型模式**：这些设计模式特别关注对象之间的通信。	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）
4	**J2EE 模式**：这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。	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）

设计模式的六大原则

总原则 - 开闭原则

对扩展开放，对修改封闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，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等，后面的具体设计中我们会提到这点。

1、单一职责原则

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，否则就应该把类拆分。

2、里氏替换原则（Liskov Substitution Principle）

任何基类可以出现的地方，子类一定可以出现。里氏替换原则是继承复用的基石，只有当衍生类可以替换基类，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。里氏替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

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

面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

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

每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

5、迪米特法则（最少知道原则）（Demeter Principle）

一个类对自己依赖的类知道的越少越好。无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

6、合成复用原则（Composite Reuse Principle）

尽量首先使用合成/聚合的方式，而不是使用继承。

创建型模式

单例模式

所谓单例，就是整个程序有且仅有一个实例。该类负责创建自己的对象，同时确保只有一个对象被创建。

特点

- 类构造器私有
- 持有自己类型的属性
- 对外提供获取实例的静态方法

懒汉模式

线程不安全，延迟初始化，懒加载

```
public class Singleton {
    private static Singleton instance;
    private Singleton (){}

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

饿汉模式

线程安全，比较常用，但容易产生垃圾，因为**一开始就初始化**

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton (){}
    public static Singleton getInstance() {
        return instance;
    }
}
```

双重锁模式

线程安全，延迟初始化。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。

```
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;
    }
}
```

双重检查模式，进行了两次的判断，第一次是为了避免不要的实例，第二次是为了进行同步，避免多线程问题。由于`singleton=new Singleton()`对象的创建在JVM中可能会进行重排序，在多线程访问下存在风险，使用`volatile`修饰`signleton`实例变量有效，解决该问题。

静态内部类单例模式

```
public class Singleton {
    private Singleton(){
    }
    public static Singleton getInstance(){
        return Inner.instance;
    }
    private static class Inner {
        private static final Singleton instance = new Singleton();
    }
}
```

只有第一次调用getInstance方法时，虚拟机才加载 Inner 并初始化instance ，只有一个线程可以获得对象的初始化锁，其他线程无法进行初始化，保证对象的唯一性。线程安全。

枚举单例模式

默认枚举实例的创建是线程安全的，并且在任何情况下都是单例。实际上

- 枚举类隐藏了私有的构造器。
- 枚举类的域 是相应类型的一个实例对

```
public enum Singleton  {
    INSTANCE;

    //doSomething 该实例支持的行为

    //可以省略此方法，通过Singleton.INSTANCE进行操作
    public static Singleton getInstance() {
        return Singleton.INSTANCE;
    }
}
```

枚举实例在日常开发是很少使用的，就是很简单以导致可读性较差。

总结

在以上所有的单例模式中，推荐静态内部类单例模式。主要是非常直观，**即保证线程安全又保证唯一性。**

工厂模式

简单工厂模式

又叫做**静态工厂方法模式。**是由一个工厂对象决定创建出哪一种产品类的实例。实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。

作用：将"类实例化的操作"与"使用对象的操作"分开，让使用者不用知道具体参数就可以实例化出所需要的"产品"类，从而避免了在客户端代码中显式指定，实现了解耦。

> **主要角色 工厂**：负责实现创建所有实例的内部逻辑，并提供一个外界调用的方法，创建所需的产品对象。 **抽象产品**：负责描述产品的公共接口 **具体产品**：描述生产的具体产品。 **举个简单易懂的例子**：“假设”有一台饮料机（工厂），可以调出各种口味的饮料（抽象产品），有三个按钮（参数）对应这三种饮料（具体产品）。这时候你可以根据点击按钮来选择你喜欢的饮料。

```
/**
 *  @ Product.java
```

```
 *  抽象产品
 *  描述产品的公共接口
 */
abstract  class Product {
    //产品介绍
    abstract void intro();
}

/**
 * @ AProduct.java
 * 具体产品A
 * （可以看成是一种饮料：可乐）
 */
public class AProduct extends Product{
    @Override
    void intro() {
        System.out.println("可乐");
    }
}

/**
 * @ BProduct.java
 * @具体产品B
 * @（可以看成是一种饮料：奶茶）
 */
public class BProduct extends Product{
    @Override
    void intro() {
        System.out.println("奶茶");
    }
}

/**
 * @ CProduct.java
 * 具体产品C
 * （可以看成是一种饮料：咖啡）
 */
public class CProduct extends Product{
    @Override
    void intro() {
        System.out.println("咖啡");
    }
}
```

```
/**
 * 工厂
 * 负责实现创建所有实例的内部逻辑，并提供一个外界调用的方法，创建所需的产品对象。
 */
public class Factory {
    /**
     * 供外界调用的方法
     * （可以看成是对外提供的三种按钮）
     * @param type
     * @return 产品实例
     */
    public static Product getProduct(String type) {
        switch (type) {
```

```
            case "A":
                return new AProduct();
            case "B":
                return new BProduct();
            case "C":
                return new CProduct();
            default:
                return null;
        }
    }
}
```

根据例子可以描述为：**一个抽象产品类，可以派生出多个具体产品类。一个具体工厂类，通过往此工厂的static方法中传入不同参数，产出不同的具体产品类实例。**

> 优点：将创建使用工作分开，不必关心类对象如何创建，实现了解耦； 缺点：违背“开放 - 关闭原则”，一旦添加新产品就不得不修改工厂类的逻辑，这样就会造成工厂逻辑过于复杂。

工厂方法模式

又称**工厂模式、多态工厂模式**和**虚拟构造器模式**，通过**定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象**。一种常用的对象创建型设计模式,此模式的核心精神是封装 类中不变的部分。

作用：将类的实例化（具体产品的创建）延迟到**工厂类的子类**（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

> **主要角色 抽象工厂**：描述具体工厂的公共接口 **具体工厂**：描述具体工厂，创建产品的实例，供外界调用 **抽象产品**：负责描述产品的公共接口 **具体产品**：描述生产的具体产品

> **举个简单易懂的例子**：“假设”有各类的饮料机（抽象工厂），可以调出各种的饮料（抽象产品）。但是一类饮料机（具体工厂），只能生产一种饮料（具体产品）。如果你需要喝可乐，就需要买可乐饮料机。

```
/**
 * @ Product.java
 *   抽象产品
 */
abstract class Product {
    //产品介绍
    abstract void intro();
}

/**
 * @ ProductA.java
 * 具体产品A
 */
public class ProductA extends Product{
    @Override
    void intro() {
        System.out.println("饮料A");
    }
}

/**
 * @ ProductB.java
 * 具体产品B
 */
```

```
public class ProductB extends Product{
    @Override
    void intro() {
        System.out.println("饮料B");
    }
}
```

```
/**
 *  @ Factory.java
 *    抽象工厂
 */
abstract class Factory {
    //生产产品
    abstract Product getProduct();
}

/**
 * @ FactoryA.java
 * 具体工厂A
 * 负责具体的产品A生产
 */
public class FactoryA extends Factory{
    @Override
    Product getProduct() {
        return new ProductA();
    }
}

/**
 * @ FactoryB.java
 * @具体工厂B
 * 负责具体的产品B生产
 */
public class FactoryB extends Factory{
    @Override
    Product getProduct() {
        return new ProductB();
    }
}
```

根据例子可以描述为：**一个抽象产品类，可以派生出多个具体产品类。一个抽象工厂类，可以派生出多个具体工厂类。每个具体工厂类只能创建一个具体产品类的实例。**

> 优点：
>
> 1. 符合开-闭原则：新增一种产品时，只需要增加相应的具体产品类和相应的工厂子类即可
> 2. 符合单一职责原则：每个具体工厂类只负责创建对应的产品
>
> 缺点：
>
> 1. 增加了系统的复杂度：类的个数将成对增加
> 2. 增加了系统的抽象性和理解难度
> 3. 一个具体工厂只能创建一种具体产品

抽象工厂模式

定义：**提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类；具体的工厂负责实现具体的产品实例。**

抽象工厂模式与工厂方法模式最大的区别：**抽象工厂中每个工厂可以创建多种类的产品；而工厂方法每个工厂只能创建一类**

> **主要对象 抽象工厂**：描述具体工厂的公共接口 **具体工厂**：描述具体工厂，创建产品的实例，供外界调用 **抽象产品族**：描述抽象产品的公共接口 **抽象产品**：描述具体产品的公共接口 **具体产品**：具体产品

> **举个简单易懂的例子：（找了个不怎么好的比喻，看不懂得可以看相关推荐链接）** “假设”有各类的自动售卖机（抽象工厂），可以出售各类食品（抽象产品族）。 有饮料、零食（抽象产品），比如常见的零食售卖机（具体工厂），出售矿泉水与面包（具体产品）。

> **产品：Product、ProductA、ProductB、ProductAa、ProductBb**

```
/**
 * @ Product.java
 * 抽象产品族 (食品)
 */
abstract class Product {
    //产品介绍
    abstract void intro();
}

/**
 * @ ProductA.java
 * 抽象产品  (饮料)
 */
abstract class ProductA extends Product{
    @Override
    abstract void intro();
}

/**
 * @ ProductB.java
 * 抽象产品  (零食)
 */
abstract class ProductB extends Product{
    @Override
    abstract void intro();
}

/**
 * @ ProductAa.java
 * 具体产品  (矿泉水)
 */
public  class ProductAa extends ProductA{
    @Override
    void intro() {
        System.out.println("矿泉水");
    }
}

/**
 * @ ProductBb.java
 * 抽象产品  (面包)
 */
```

```
public class ProductBb extends ProductB{
    @Override
    void intro() {
        System.out.println("面包");
    }
}
```

```
/**
 * @ Factory.java
 * 抽象工厂
 */
abstract class Factory {
    //生产饮料
    abstract Product getProductA();
    //生产零食
    abstract Product getProductB();
}

/**
 * @ FactoryA.java
 * 具体工厂A
 * 负责具体的A类产品生产
 */
public class FactoryA extends Factory{
    @Override
    Product getProductA() {
        //生产矿泉水
        return new ProductAa();
    }
    @Override
    Product getProductB() {
        //生产面包
        return new ProductBb();
    }
}
```

根据实例可以描述为： **多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。一个抽象工厂类，可以派生出多个具体工厂类。 每个具体工厂类可以创建多个具体产品类的实例。**.

> 优点：
>
> 1. 降低耦合
> 2. 符合开-闭原则
> 3. 符合单一职责原则
> 4. 不使用静态工厂方法，可以形成基于继承的等级结构。
>
> 缺点：难以扩展新种类产品

总结

角色不同：

> 1. 简单工厂：具体工厂、抽象产品、具体产品

2. 工厂方法：**抽象工厂**、具体工厂、抽象产品、具体产品
3. 抽象工厂：抽象工厂、具体工厂、**抽象产品族**、抽象产品、具体产品

定义：

1. 简单工厂：**由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（继承自一个父类或接口）的实例。**
2. 工厂方法：**定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象**
3. 抽象工厂：**提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类；具体的工厂负责实现具体的产品实例。**

对比：

1. **工厂方法模式**解决了**简单工厂模式**的"开放 - 关闭原则
2. 抽象工厂模式解决了**工厂方法模式**一个具体工厂只能创建一类产品

参考

https://www.jianshu.com/p/d951ac56136e

建造者模式

简介

创建者模式又叫建造者模式，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。创建者模式隐藏了复杂对象的创建过程，它把复杂对象的创建过程加以抽象，通过子类继承或者重载的方式，动态的创建具有复合属性的对象。

适用场景

- 隔离复杂对象的创建和使用，相同的方法，不同执行顺序，产生不同事件结果
- 多个部件都可以装配到一个对象中，但产生的运行结果不相同
- 产品类非常复杂或者产品类因为调用顺序不同而产生不同作用
- 初始化一个对象时，参数过多，或者很多参数具有默认值
- Builder模式不适合创建差异性很大的产品类 产品内部变化复杂，会导致需要定义很多具体建造者类实现变化，增加项目中类的数量，增加系统的理解难度和运行成本
- 需要生成的产品对象有复杂的内部结构，这些产品对象具备共性；

主要作用

> 在用户不知道对象的建造过程和细节的情况下就可以直接创建复杂的对象。

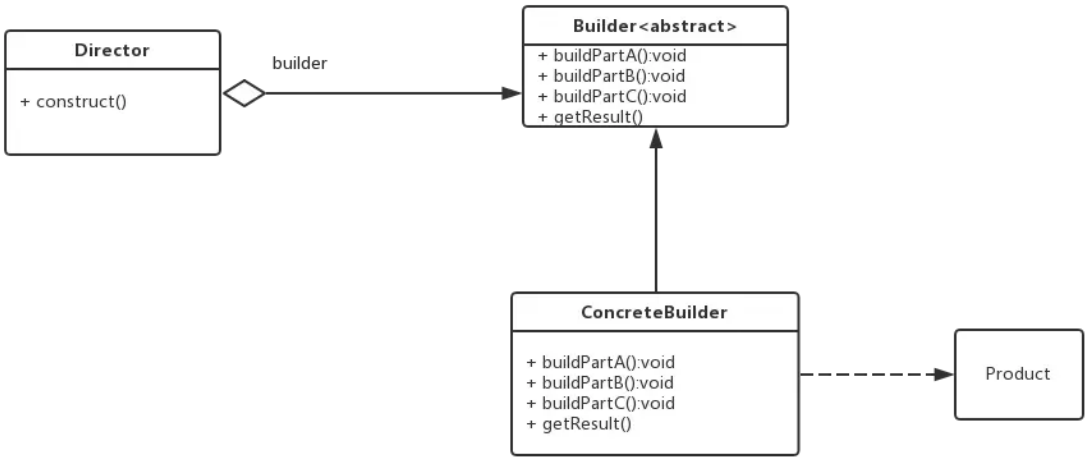
- 用户只需要给出指定复杂对象的类型和内容；
- 建造者模式负责按顺序创建复杂对象（把内部的建造过程和细节隐藏起来)

解决的问题

- 方便用户创建复杂的对象（不需要知道实现过程）
- 代码复用性 & 封装性（将对象构建过程和细节进行封装 & 复用）

> 例子：造汽车 & 买汽车。
>
> 1. 工厂（建造者模式）：负责制造汽车（组装过>程和细节在工厂内）
> 2. 汽车购买者（用户）：你只需要说出你需要的>型号（对象的类型和内容），然后直接购买就可>>以使用了 （不需要知道汽车是怎么组装的（车轮、车门、>发动机、方向盘等等））

模式结构



建造者模式主要有四个角色

- Product:要创建的产品类对象
- Builder：建造者的抽象类，规范产品对象的各个组成部分的建造，一般由子类实现具体的建造过程。
- ConcreteBuilder:具体的Builder类，根据不同的业务逻辑，具体化对象的各个组成部分的创建。
- Director：调用具体的建造者，来创建对象的各个部分，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。

示例代码

产品类：

```
/*
 * 产品类，Product角色
 * @author Jackson
 * @version 1.0.0
 * since 2018 12 24
 */
public class MacBook {

    private String mBoard;
    private String mDisplay;
    private String mOs;

    public String getBoard() {
        return mBoard;
    }

    public void setBoard(String board) {
        mBoard = board;
    }

    public String getDisplay() {
        return mDisplay;
    }

    public void setDisplay(String display) {
        mDisplay = display;
    }
```

```
    public String getOs() {
        return mOs;
    }

    public void setOs(String os) {
        mOs = os;
    }

    @Override
    public String toString() {
        return "MacBook:" + "mBoard=" + mBoard + ",mDisplay=" + mDisplay +
",mOs=" + mOs;
    }

}
```

抽象的Builder类:

```
/*
 * 抽象Builder类
 * @author Jackson
 * @version 1.0.0
 * since 2018 12 24
 */
public abstract class Builder {
    // 设置主板
    public abstract void builderBoard(String board);

    // 设置显示器
    public abstract void builderDisplay(String display);

    // 设置操作系统
    public abstract void builderOs(String os);

    // 创建Computer
    public abstract MacBook getComputer();
}
```

具体的Builder类

```
/*
 * 具体的Builder类
 * @author Jackson
 * @version 1.0.0
 * since 2018 12 24
 */
public class MacbookBulder extends Builder{


    private MacBook mMacBook=new MacBook();
```

```
    @Override
    public void builderBoard(String board) {
        mMacBook.setBoard(board);

    }

    @Override
    public void builderDisplay(String display) {
        mMacBook.setDisplay(display);

    }

    @Override
    public void builderOs(String os) {
        mMacBook.setOs(os);

    }

    @Override
    public MacBook getComputer() {
        return mMacBook;
    }
}
```

Director类:

Director类的主要作用是调用具体的builder，来构建对象的各个部分，Director类起到封装作用，避免高层模块深入到建造者内部的实现类。

```
public class Director {

    Builder mBuilder=null;

    public Director(Builder builder){
        this.mBuilder=builder;
    }

    public void constract(String board,String display,String os){
        mBuilder.builderBoard(board);
        mBuilder.builderDisplay(display);
        mBuilder.builderOs(os);
    }
}
```

客户端代码：

```
  Builder builder=new MacbookBulder();
  Director director=new Director(builder);
  director.constract("英特尔主板","Retina显示器","Mac OS X系统");
  System.out.println(builder.getComputer().toString());
```

实际在开发中，**Director常常被忽略**，而是直接通过Builder的方式**链式组装**，同是，builder是一个静态内部类。

```
public class ThinkPad {

    private String mBoard;
    private String mDisplay;
    private String mOs;

    private ThinkPad(Builder builder){
        this.mBoard=builder.mBoard;
        this.mDisplay=builder.mDisplay;
        this.mOs=builder.mOs;
    }

    public String getBoard() {
        return mBoard;
    }

    public String getDisplay() {
        return mDisplay;
    }


    public String getOs() {
        return mOs;
    }


    static class Builder{

        private String mBoard;
        private String mDisplay;
        private String mOs;

        public Builder setBoard(String board){
            this.mBoard=board;
            return this;
        }

        public Builder setDisplay(String display){
            this.mDisplay=display;
            return this;
        }

        public Builder setOs(String os){
            this.mOs=os;
            return this;
        }

        public ThinkPad builder(){
            return new ThinkPad(this);
        }
    }

    @Override
    public String toString() {
```

```
        return "ThinkPad{" +
                "mBoard='" + mBoard + '\'' +
                ", mDisplay='" + mDisplay + '\'' +
                ", mOs='" + mOs + '\'' +
                '}';
    }

    public static void main(String[] args) {
        ThinkPad thinkPad=
                new
ThinkPad.Builder().setBoard("Intel").setDisplay("JDI").setOs("Windows
10").builder();
        System.out.println(thinkPad);
    }
}
```

参考

https://www.jianshu.com/p/3d1c9ffb0a28

https://www.jianshu.com/p/bc798e5e63b1

原型模式

- 作用：用原型实例指定创建对象的种类，并且通过复制（克隆）这些原型创建新的对象。
- 使用频率：不常用，使用场景非常少。
- 特点：通过克隆的方式创建对象。
- 境界:写代码的最高境界就是crtl+c/v，建立自己的技术平台，复用代码，修改代码。

关键点：

- 抽象原型类：它是声明克隆方法的接口，是所有具体原型类的公共父类，可以是抽象类也可以是接口，甚至可以是具体实现类。
- 具体原型类：它实现抽象原型类中声明的方法，在克隆方法中返回自己的一个克隆对象。
- 客户类：具体原型对象克隆自身从而创建一个全新的对象。

使用场景：构造函数复杂，这样通过new来创建对象需要非常繁琐的数据准备。

举例

抽象原型类：

```
public abstract class Prototype implements Cloneable{

    private String id;

    public Prototype(String id) {
        this.id = id;
    }

    public String getId() {
        return this.id;
    }

    public Prototype clone() throws CloneNotSupportedException {
```

```
        return (Prototype) super.clone();

    }

}
```

具体原型类:

```
public class CreatePrototype extends Prototype{

    public CreatePrototype(String id) {
        super(id);
    }
}
```

客户端代码:

```
public class Client {
    public static void main(String[] args) throws CloneNotSupportedException {

        Prototype p1 = new CreatePrototype("1");
        Prototype p2 = p1.clone();
        System.out.println(p1 == p2);

    }
}
```

参考

https://www.jianshu.com/p/ed6a4552517c

https://blog.csdn.net/qq_29726359/article/details/92406312

行为型模式

责任链模式

角色:

抽象处理者（Handler）角色：该角色对请求进行抽象，并定义一个方法来设定和返回对下一个处理者的引用。

具体处理者（Concrete Handler）：该角色接到请求后，可以选择将请求处理掉，或者将请求传给下一个处理者。由于具体处理者持有对下一个处理者的引用，因此，如果需要，处理者可以访问下一个处理者。

优点

- 责任链模式将请求和处理分开，请求者不知道是谁处理的，处理者可以不用知道请求的全貌。
- 提高系统的灵活性。

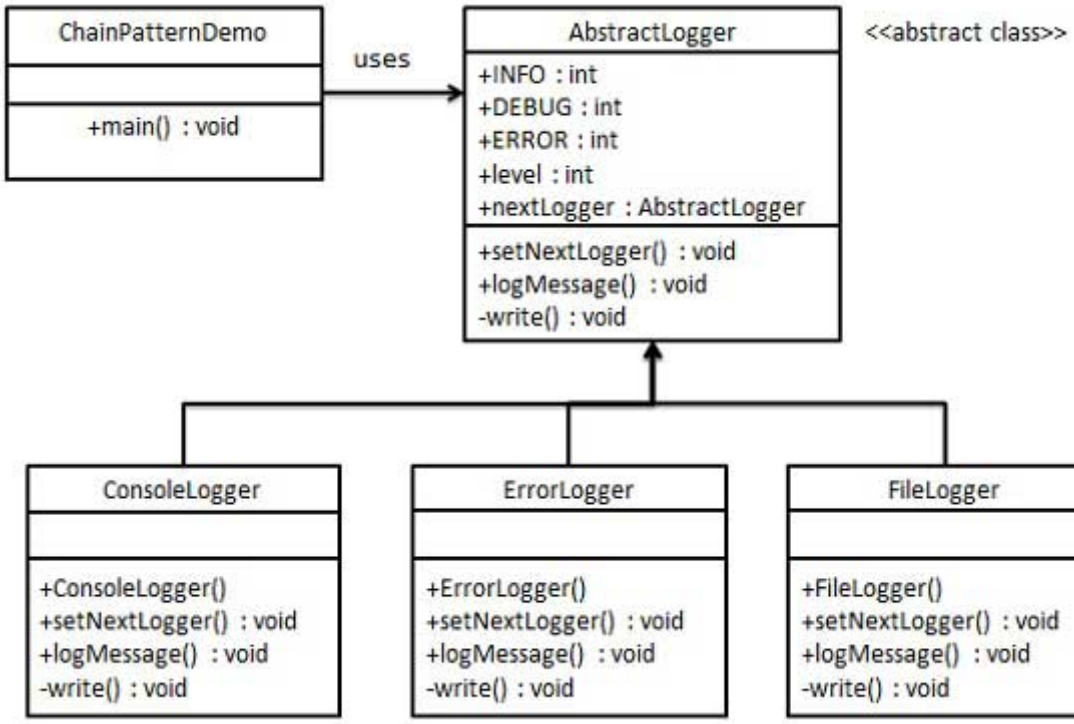
缺点

- 降低程序的性能。每个请求都是从链头遍历到链尾，当链比较长的时候，性能会大幅下降。
- 不易于调试。由于该模式采用了类似递归的方式，调试的时候逻辑比较复杂。

应用场景

责任链模式是一种常见的模式，Struts2的核心控件FilterDispatcher是一个Servlet过滤器，该控件就是采用责任链模式，可以对用户请求进行层层过滤处理。责任链模式在实际项目中的使用比较多，其典型的应用场景如下：

- 一个请求需要一系列的处理工作。
- 业务流的处理，例如文件审批。
- 对系统进行扩展补充。




```
/**
 * 抽象处理器.
 */
public abstract class AbstractLogger {
    public static final int INFO = 1;    //一级日志
    public static final int DEBUG = 2;   //二级日志包括一级
    public static final int ERROR = 3;   //三级包括前两个

    protected int level;
    //责任链下一个元素
    protected AbstractLogger nextLogger ;
    public void setNextLogger(AbstractLogger nextLogger){
        this.nextLogger = nextLogger;
    }

    //不同级别的记录方法不一样,这里给一个抽象的记录方法
    abstract protected void write(String message);

    //调用责任链处理器的记录方法.并且判断下一个责任链元素是否存在,若存在,则执行下一个方法.
    public void logMessage(int level,String message){
        if (this.level <= level){    //根据传进来的日志等级,判断哪些责任链元素要去记录
```

```
            write(message);
        }
        if (nextLogger != null){
            nextLogger.logMessage(level,message);    //进行下一个责任链元素处理
        }
    }
}

/**
 * 控制台处理器.
 */
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);
    }
}

/**
 * 文件处理器.
 */
public class FileLogger extends AbstractLogger {
    public FileLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("File Console::Logger"+message);
    }
}

/**
 * error日志处理器.
 */
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);
    }
}

/**
 * 处理链.
 */
public class ChainPatternDemo {

    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;
    }
}

public class Main {
    public static void main(String[] args) {
        AbstractLogger logger = ChainPatternDemo.getChainOfLoggers();
        logger.logMessage(1,"一级日志记录");
        System.out.println("--------------------------------");
        logger.logMessage(2,"二级日志记录");
        System.out.println("--------------------------------");
        logger.logMessage(3,"三级日志记录");
    }
}
```

参考

https://www.cnblogs.com/aeolian/p/8888958.html

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。

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。

如何解决：通过调用者调用接受者执行命令，顺序：调用者→接受者→命令。

关键代码：定义三个角色：1、received 真正的命令执行对象 2、Command 3、invoker 使用命令对象的入口

应用实例：struts 1 中的 action 核心控制器 ActionServlet 只有一个，相当于 Invoker，而模型层的类会随着不同的应用有不同的模型类，相当于具体的 Command。

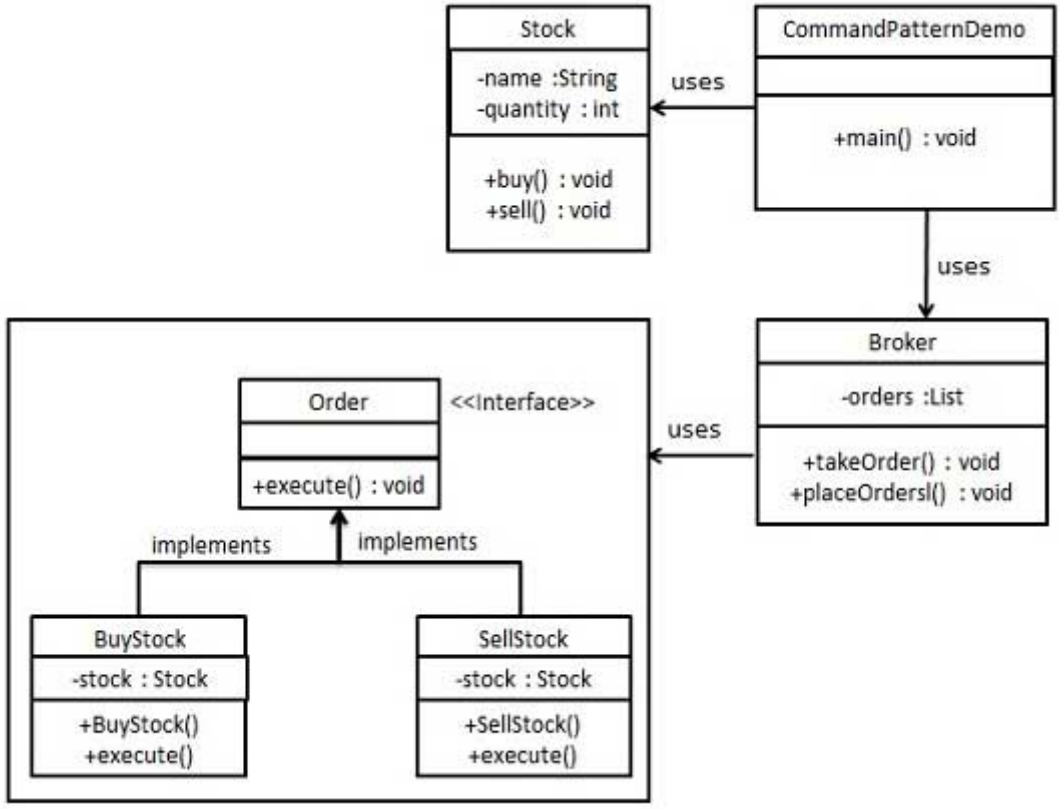
优点： 1、降低了系统耦合度。 2、新的命令可以很容易添加到系统中去。

缺点：使用命令模式可能会导致某些系统有过多的具体命令类。

使用场景：认为是命令的地方都可以使用命令模式，比如： 1、GUI 中每一个按钮都是一条命令。 2、模拟 CMD。

注意事项：系统需要支持命令的撤销(Undo)操作和恢复(Redo)操作，也可以考虑使用命令模式，见命令模式的扩展。

实现



我们首先创建作为命令的接口 *Order*

```
public interface Order {
    void execute();
}
```

然后创建作为请求的 *Stock* 类。

```
public class Stock {

    private String name = "ABC";
    private int quantity = 10;

    public void buy(){
        System.out.println("Stock [ Name: "+name+", Quantity: " + quantity +" ]
bought");
    }
    public void sell(){
        System.out.println("Stock [ Name: "+name+", Quantity: " + quantity +" ]
sold");
    }
}
```

实体命令类 *BuyStock* 和 *SellStock*，实现了 *Order* 接口，将执行实际的命令处理。

```
public class BuyStock implements Order {
   private Stock abcStock;
```

```
   public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.buy();
   }
}

public class SellStock implements Order {
   private Stock abcStock;

   public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }

   public void execute() {
      abcStock.sell();
   }
}
```

创建作为调用对象的类 *Broker*，它接受订单并能下订单。

```
public class Broker {
   private List<Order> orderList = new ArrayList<Order>();

   public void takeOrder(Order order){
      orderList.add(order);
   }

   public void placeOrders(){
      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```

Broker 对象使用命令模式，基于命令的类型确定哪个对象执行哪个命令。

```
    public static void main(String[] args) {
        Stock abcStock = new Stock();

        BuyStock buyStockOrder = new BuyStock(abcStock);
        SellStock sellStockOrder = new SellStock(abcStock);

        Broker broker = new Broker();
        broker.takeOrder(buyStockOrder);
        broker.takeOrder(sellStockOrder);

        broker.placeOrders();
    }
```

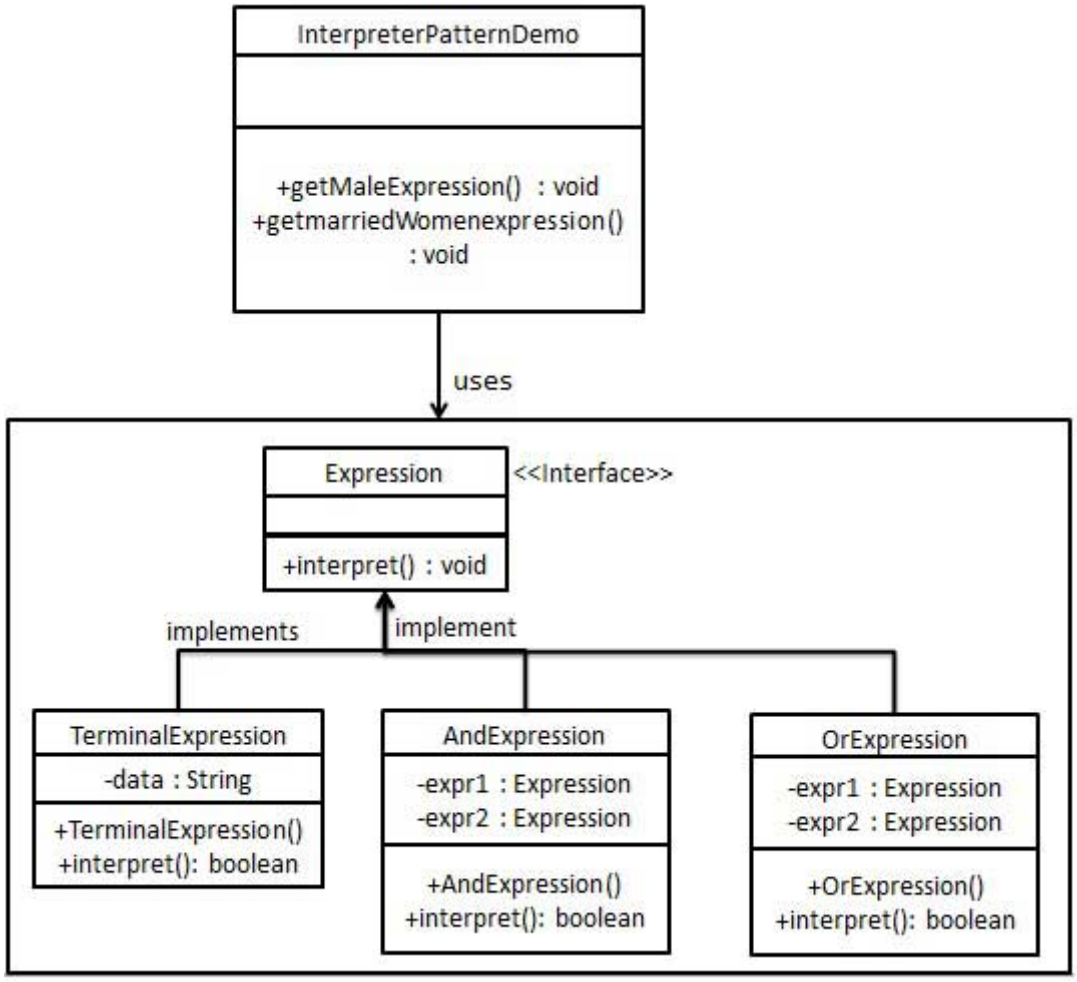
参考

https://www.runoob.com/design-pattern/command-pattern.html

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。

实现



我们将创建一个接口 *Expression* 和实现了 *Expression* 接口的实体类。定义作为上下文中主要解释器的 *TerminalExpression* 类。其他的类 *OrExpression*、*AndExpression* 用于创建组合式表达式。

```
public interface Expression {
    public boolean interpret(String context);
}

public class TerminalExpression implements Expression {

   private String data;

   public TerminalExpression(String data){
      this.data = data;
   }
```

```
   @Override
   public boolean interpret(String context) {
      if(context.contains(data)){
         return true;
      }
      return false;
   }
}

public class OrExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public OrExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) || expr2.interpret(context);
   }
}

public class AndExpression implements Expression {

   private Expression expr1 = null;
   private Expression expr2 = null;

   public AndExpression(Expression expr1, Expression expr2) {
      this.expr1 = expr1;
      this.expr2 = expr2;
   }

   @Override
   public boolean interpret(String context) {
      return expr1.interpret(context) && expr2.interpret(context);
   }
}
```

InterpreterPatternDemo，我们的演示类使用 *Expression* 类创建规则和演示表达式的解析。

```
    //规则：Robert 和 John 是男性
    public static Expression getMaleExpression(){
        Expression robert = new TerminalExpression("Robert");
        Expression john = new TerminalExpression("John");
        return new OrExpression(robert, john);
    }

    //规则：Julie 是一个已婚的女性
    public static Expression getMarriedWomanExpression(){
        Expression julie = new TerminalExpression("Julie");
        Expression married = new TerminalExpression("Married");
```

```
        return new AndExpression(julie, married);
    }

    public static void main(String[] args) {
        Expression isMale = getMaleExpression();
        Expression isMarriedWoman = getMarriedWomanExpression();

        System.out.println("John is male? " + isMale.interpret("John"));
        System.out.println("Julie is a married women? "
                + isMarriedWoman.interpret("Married Julie"));
    }
```

参考

https://www.runoob.com/design-pattern/interpreter-pattern.html

迭代器模式

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。

主要解决：不同的方式来遍历整个整合对象。

何时使用：遍历一个聚合对象。

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。

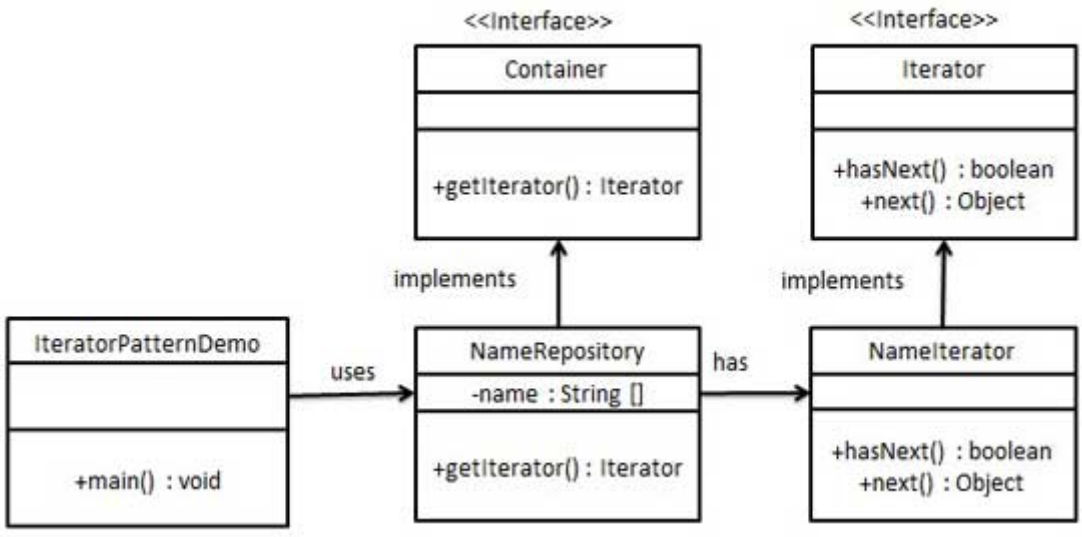
优点： 1、它支持以不同的方式遍历一个聚合对象。 2、迭代器简化了聚合类。 3、在同一个聚合上可以有多个遍历。 4、在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

缺点：由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

使用场景： 1、访问一个聚合对象的内容而无须暴露它的内部表示。 2、需要为聚合对象提供多种遍历方式。 3、为遍历不同的聚合结构提供一个统一的接口。

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。

实现



我们将创建一个叙述导航方法的 *Iterator* 接口和一个返回迭代器的 *Container* 接口。

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
}

public interface Container {
    public Iterator getIterator();
}
```

实现了 *Container* 接口的实体类将负责实现 *Iterator* 接口。该类有实现了 *Iterator* 接口的**内部类** *NameIterator*。

```
public class NameRepository implements Container {
    public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    private class NameIterator implements Iterator {

        int index;

        @Override
        public boolean hasNext() {
            if(index < names.length){
                return true;
            }
            return false;
        }

        @Override
        public Object next() {
            if(this.hasNext()){
                return names[index++];
            }
```

```
            return null;
        }
    }
}
```

IteratorPatternDemo，我们的演示类使用实体类 *NamesRepository* 来打印 *NamesRepository* 中存储为集合的 *Names*。

```
    public static void main(String[] args) {
        NameRepository namesRepository = new NameRepository();

        for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
            String name = (String)iter.next();
            System.out.println("Name : " + name);
        }
    }
```

参考

https://www.runoob.com/design-pattern/iterator-pattern.html

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。

介绍

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。

何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。

如何解决：使用面向对象技术，可以将这种依赖关系弱化。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

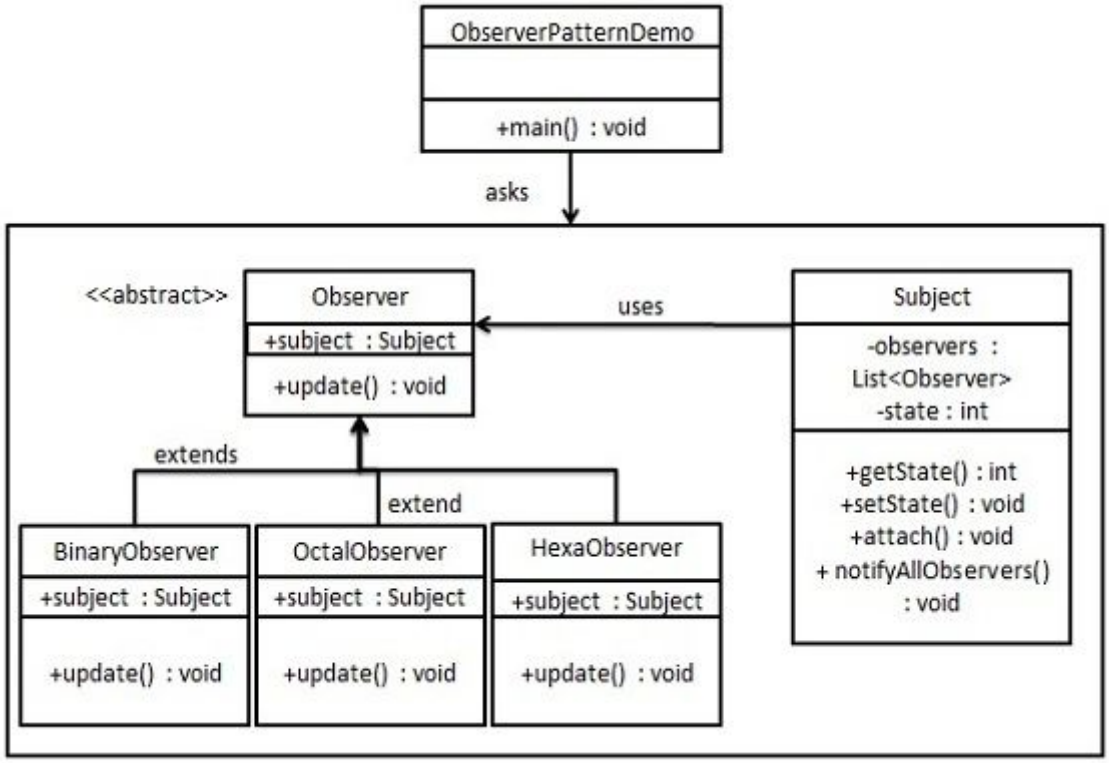
使用场景：

- 一个抽象模型有两个方面，其中一个方面依赖于另一个方面。将这些方面封装在独立的对象中使它们可以各自独立地改变和复用。
- 一个对象的改变将导致其他一个或多个对象也发生改变，而不知道具体有多少对象将发生改变，可以降低对象之间的耦合度。
- 一个对象必须通知其他对象，而并不知道这些对象是谁。

- 需要在系统中创建一个触发链，A对象的行为将影响B对象，B对象的行为将影响C对象……，可以使用观察者模式创建一种链式触发机制。

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。

实现



观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 *Subject* 类、*Observer* 抽象类和扩展了抽象类 *Observer* 的实体类。

```
public abstract class Observer {
    protected Subject subject;
    public abstract void update();
}

public class Subject {

    private List<Observer> observers
            = new ArrayList<Observer>();
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        notifyAllObservers();
    }

    public void attach(Observer observer){
        observers.add(observer);
    }
```

```
    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

public class OctalObserver extends Observer{

    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Octal String: "
                + Integer.toOctalString( subject.getState() ) );
    }
}

public class HexaObserver extends Observer{

    public HexaObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Hex String: "
                + Integer.toHexString( subject.getState() ).toUpperCase() );
    }
}

public class BinaryObserver extends Observer{

    public BinaryObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println( "Binary String: "
                + Integer.toBinaryString( subject.getState() ) );
    }
}
```

ObserverPatternDemo，我们的演示类使用 *Subject* 和实体类对象来演示观察者模式。

```
    public static void main(String[] args) {
        Subject subject = new Subject();
```

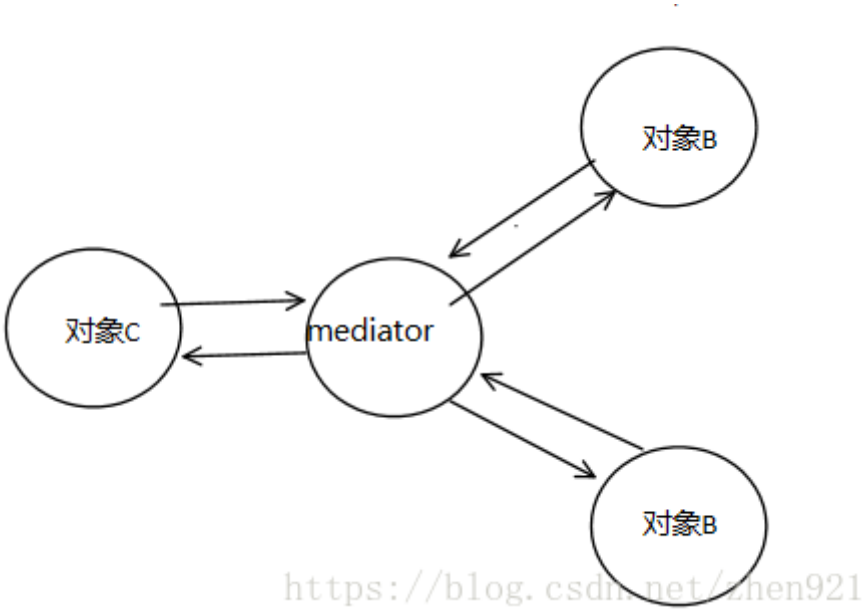
```
        new HexaObserver(subject);
        new OctalObserver(subject);
        new BinaryObserver(subject);

        System.out.println("First state change: 15");
        subject.setState(15);
        System.out.println("Second state change: 10");
        subject.setState(10);
    }
```

参考

https://www.runoob.com/design-pattern/observer-pattern.html

中介者模式

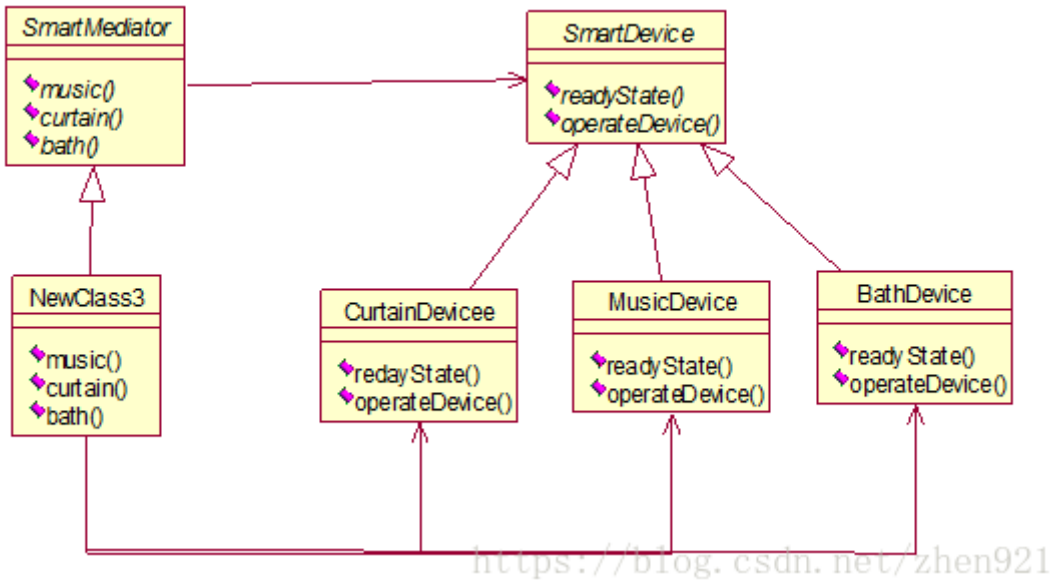
中介者模式的引入则极大的弥补了上述程序的两个缺陷，对象交互如下图



中介者模式角色组成

角色	作用
抽象中介者（mediator）	定义一个接口用于和对象通信（SmartDevice）
具体中介者（concretemediator）	协调各同事对象实现协作，了解维护各个同事()
抽象同事角色（colleague）	规定了同事的基本类型
具体同事角色（concreteColleague）	每个同事都知道中介者对象，要与同事通信则把通信告诉中介者

实现



抽象同事类(智能设备)

```
public abstract class SmartDevice {
    //相关设备打开之后 使其进入准备状态
    public abstract void readyState(String instruction);
    //操作该设备
    public abstract void operateDevice(String instruction, SmartMediator
mediator);
}
```

具体同事类1（窗帘设备）、具体同事类2（音响设备）、具体同事类3（洗浴设备）

```
public class CurtainDevice extends SmartDevice{

    public void operateDevice(String instruction,SmartMediator mediator) {
        System.out.println("窗帘已"+instruction);//通过传入指令，打开或关闭窗帘
        mediator.curtain(instruction);//窗帘通过中介者唤醒音乐设备和洗浴设备
    }

    public void readyState(String instruction) {
        //如果其他设备开启则调用此方法，唤醒窗帘
        System.out.println("窗帘设备准备"+instruction);
    }

}

public class MusicDevice extends SmartDevice{

    public void operateDevice(String instruction,SmartMediator mediator) {
        System.out.println("音乐设备已"+instruction);
        mediator.music(instruction);
    }

    public void readyState(String instruction) {
        System.out.println("音乐设备准备"+instruction);
    }
```

```
}

public class ConcreteMediator extends SmartMediator{

    public ConcreteMediator(SmartDevice bd, SmartDevice cd, SmartDevice md) {
        super(bd, cd, md);
    }

    public void music(String instruction) {//音乐被唤醒后，使其他设备进入准备状态
        cd.readyState(instruction);//调用窗帘的准备方法
        bd.readyState(instruction);//调用洗浴设备的准备方法
    }

    public void curtain(String instruction) {
        md.readyState(instruction);
        bd.readyState(instruction);
    }

    public void bath(String instruction) {
        cd.readyState(instruction);
        md.readyState(instruction);
    }

}
```

抽象中介者（中介设备）

```
public abstract class SmartMediator {
    //保留所有设备的引用是为了当接收指令时可以唤醒其他设备的操作
    SmartDevice bd;
    SmartDevice md;
    SmartDevice cd;
    public SmartMediator(SmartDevice bd, SmartDevice md, SmartDevice cd) {
        super();
        this.bd = bd;
        this.md = md;
        this.cd = cd;
    }
    public abstract void music(String instruction);
    public abstract void curtain(String instruction);
    public abstract void bath(String instruction);
}
```

具体中介者

```
public class ConcreteMediator extends SmartMediator{

    public ConcreteMediator(SmartDevice bd, SmartDevice cd, SmartDevice md) {
        super(bd, cd, md);
    }
```

```
    public void music(String instruction) {//音乐被唤醒后，使其他设备进入准备状态
        cd.readyState(instruction);//调用窗帘的准备方法
        bd.readyState(instruction);//调用洗浴设备的准备方法
    }

    public void curtain(String instruction) {
        md.readyState(instruction);
        bd.readyState(instruction);
    }

    public void bath(String instruction) {
        cd.readyState(instruction);
        md.readyState(instruction);
    }

}
```

客户端

```
    public static void main(String[] args) {
        SmartDevice bd=new BathDevice();
        SmartDevice cd=new CurtainDevice();
        SmartDevice md=new MusicDevice();
        SmartMediator sm=new ConcreteMediator(bd, cd, md);//把设备引用都保存在调停者中
        cd.operateDevice("open",sm); //开启窗帘
        md.operateDevice("close",sm);//关闭音乐
    }
```

程序分析

首先我们要说这个程序解决了我上面说的两个缺陷，现在我们不管是要新增设备还是替换设备，既然我们的设备的关联者只有中介者了，那么修改一下中介者就好了。之前我们是在自己设备中就能联系其他设备，现在我们把这种联系统统交给中介者去做。不过可以看出现在的程序变的并不简单，由于我们引入中介者，反而看起来程序变的更加复杂了 。

既然认识了中介者，现在我们在了解一下中介者的特点

- 作用：中介者对象封装了一系列的对象交互，中介者使各对象不需要彼此联系来相互作用，从而使耦合松散，而且可以独立的改变他们之间的交互。
- 应用场景：当有多个对象彼此间相互交互的时候，自然就会想到对象间的耦合度过高，解决办法就是封装对象间的交互行为，因此就能想到中介者模式就是干这行的。
- 应用到的设计原则：高内聚，低耦合，使用中介者明显降低了对象之间的耦合
- 中介者模式优点：
    - 通过让对象彼此解耦，增加对象的复用性
    - 通过将控制逻辑集中，可以简化系统维护
    - 通过中介者使一对所变成了一堆一，便于理解
- 缺点
    - 如果涉及不好，引入中介者会使程序变的复杂
    - 中介者承担过多责任，维护不好会出大事

参考

https://blog.csdn.net/zhen921/article/details/82316707

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。

意图：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

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。

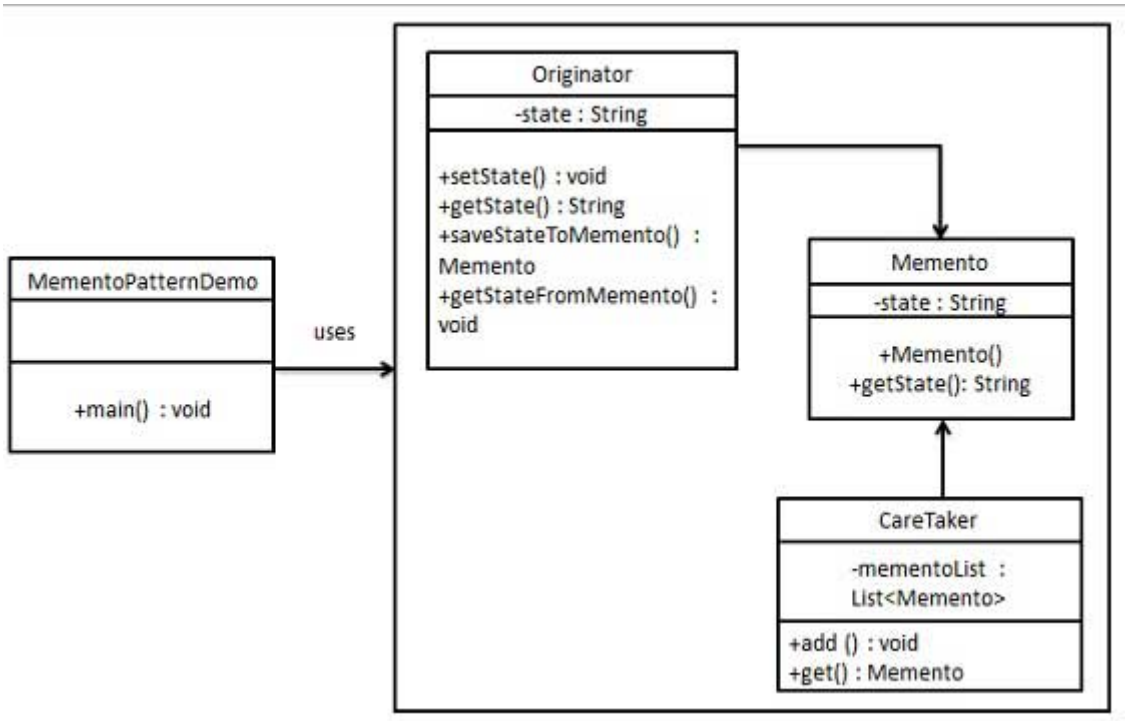
优点： 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。2、实现了信息的封装，使得用户不需要关心状态的保存细节。

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。

实现



备忘录模式使用三个类 *Memento*、*Originator* 和 *CareTaker*。Memento 包含了要被恢复的对象的状态。

```
public class Memento {
    private String state;

    public Memento(String state){
        this.state = state;
    }

    public String getState(){
        return state;
```

```
    }
}

public class Originator {
   private String state;

   public void setState(String state){
      this.state = state;
   }

   public String getState(){
      return state;
   }

   public Memento saveStateToMemento(){
      return new Memento(state);
   }

   public void getStateFromMemento(Memento Memento){
      state = Memento.getState();
   }
}

public class CareTaker {
   private List<Memento> mementoList = new ArrayList<Memento>();

   public void add(Memento state){
      mementoList.add(state);
   }

   public Memento get(int index){
      return mementoList.get(index);
   }
}
```

Originator 创建并在 Memento 对象中存储状态。Caretaker 对象负责从 Memento 中恢复对象的状态。

MementoPatternDemo，我们的演示类使用 *CareTaker* 和 *Originator* 对象来显示对象的状态恢复。

```
public static void main(String[] args) {
      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();
      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());
      originator.setState("State #4");

      System.out.println("Current State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
```

```
    }
```

参考

https://www.runoob.com/design-pattern/memento-pattern.html

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。

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）。

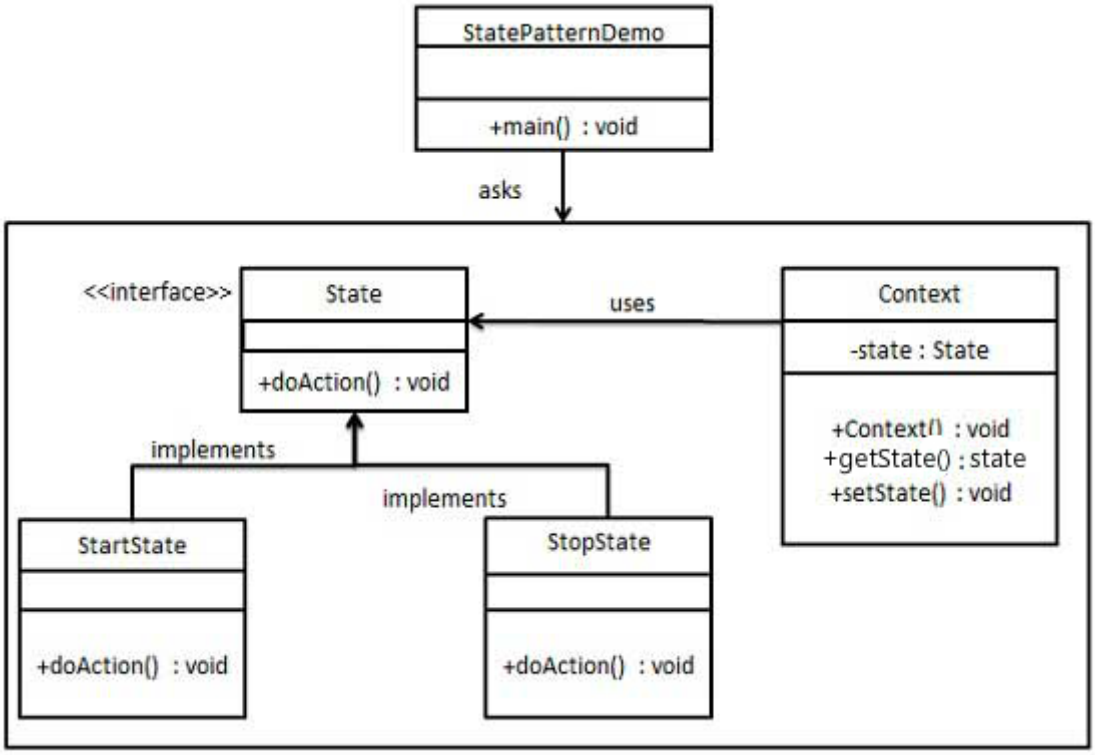
优点： 1、封装了转换规则。 2、枚举可能的状态，在枚举状态之前需要确定状态种类。 3、将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。 4、允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。 5、可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数。

缺点： 1、状态模式的使用必然会增加系统类和对象的个数。 2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。 3、状态模式对"开闭原则"的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需修改对应类的源代码。

使用场景： 1、行为随状态改变而改变的场景。 2、条件、分支语句的代替者。

注意事项：在行为受状态约束的时候使用状态模式，而且状态不超过 5 个。

实现



我们将创建一个 *State* 接口和实现了 *State* 接口的实体状态类。

```
public interface State {
    public void doAction(Context context);
}

public class StartState implements State {

    public void doAction(Context context) {
        System.out.println("Player is in start state");
        context.setState(this);
    }

    public String toString(){
        return "Start State";
    }
}

public class StopState implements State {

   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString(){
      return "Stop State";
   }
}
```

Context 是一个带有某个状态的类。

```
public class Context {
   private State state;

   public Context(){
      state = null;
   }

   public void setState(State state){
      this.state = state;
   }

   public State getState(){
      return state;
   }
}
```

StatePatternDemo，我们的演示类使用 *Context* 和状态对象来演示 Context 在状态改变时的行为变化。

```
    public static void main(String[] args) {
        Context context = new Context();

        StartState startState = new StartState();
        startState.doAction(context);

        System.out.println(context.getState().toString());

        StopState stopState = new StopState();
        stopState.doAction(context);

        System.out.println(context.getState().toString());
    }
```

参考

https://www.runoob.com/design-pattern/state-pattern.html

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。

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。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

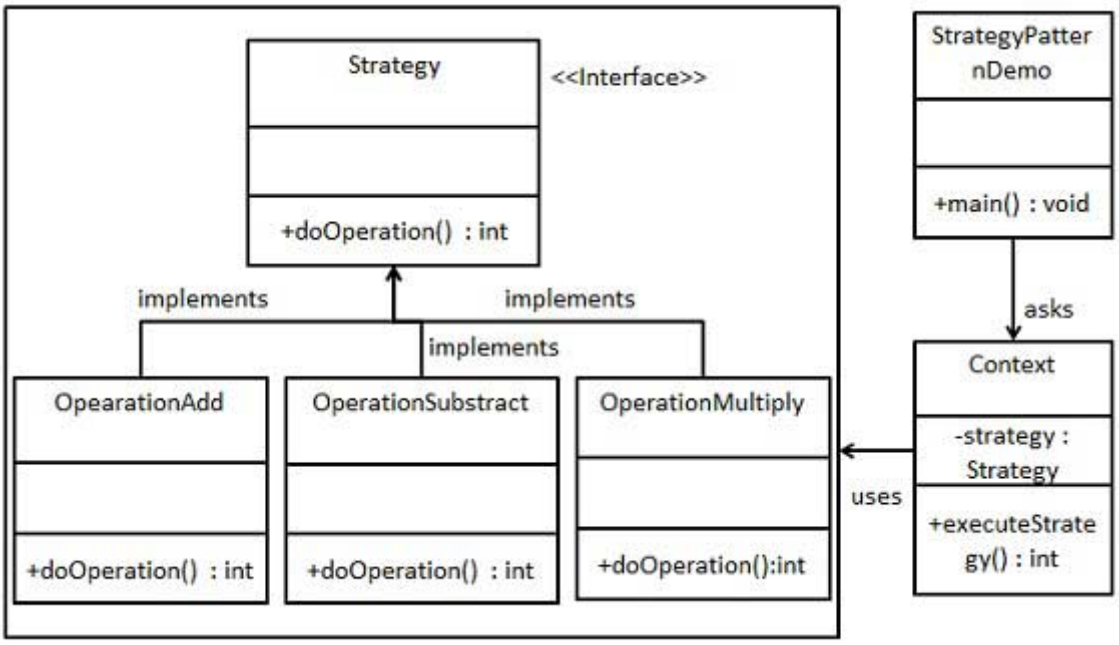
优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

实现



我们将创建一个定义活动的 *Strategy* 接口和实现了 *Strategy* 接口的实体策略类。

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}

public class OperationSubstract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}

public class OperationMultiply implements Strategy{
   @Override
   public int doOperation(int num1, int num2) {
      return num1 * num2;
   }
}
```

Context 是一个使用了某种策略的类。

```
public class Context {
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

StrategyPatternDemo，我们的演示类使用 *Context* 和策略对象来演示 Context 在它所配置或使用的策略改变时的行为变化。

```
public class Demo {
    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationSubstract());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

参考

https://www.runoob.com/design-pattern/strategy-pattern.html

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。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。

何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

关键代码：在抽象类实现，其他步骤在子类实现。

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。

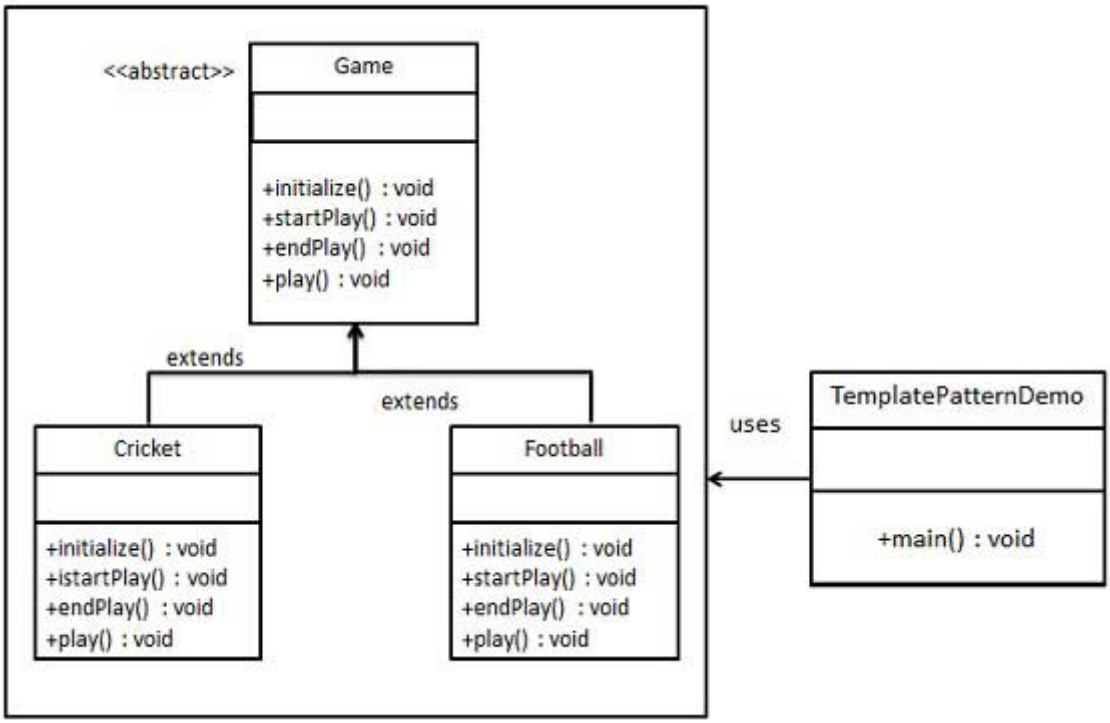
优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

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：为防止恶意操作，一般模板方法都加上 final 关键词。

实现



我们将创建一个定义操作的 *Game* 抽象类，其中，模板方法设置为 final，这样它就不会被重写。

```
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();
   }
}
```

Cricket 和 *Football* 是扩展了 *Game* 的实体类，它们重写了抽象类的方法。

```
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!");
   }
}

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!");
   }
}
```

TemplatePatternDemo，我们的演示类使用 *Game* 来演示模板模式的用法。

```
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
```

参考

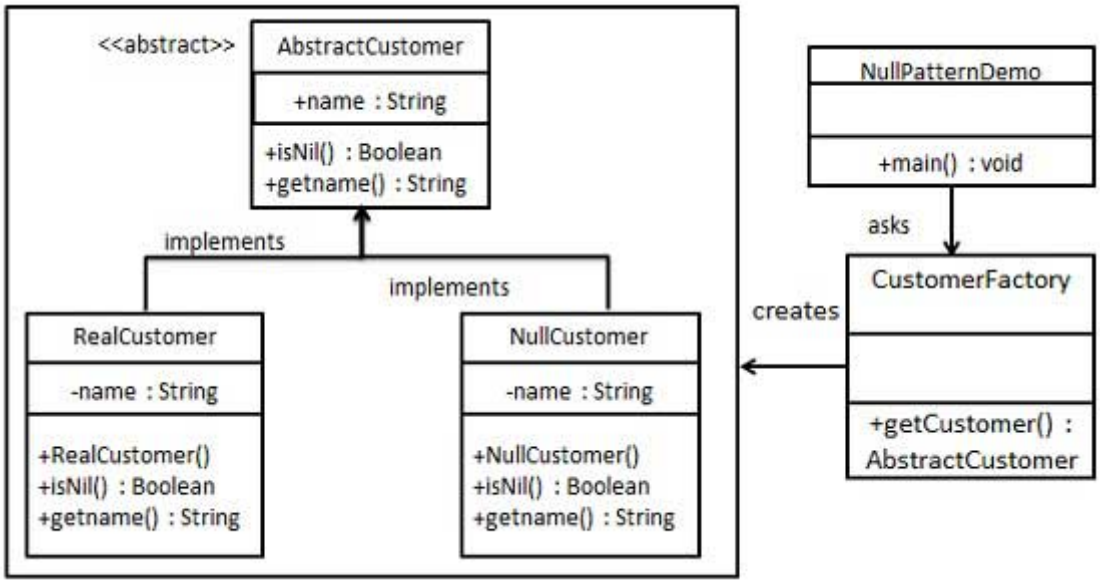
https://www.runoob.com/design-pattern/template-pattern.html

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。

实现



我们将创建一个定义操作（在这里，是客户的名称）的 *AbstractCustomer* 抽象类，和扩展了 *AbstractCustomer* 类的实体类。

```
public abstract class AbstractCustomer {
    protected String name;
    public abstract boolean isNil();
    public abstract String getName();
}

public class NullCustomer extends AbstractCustomer {

    @Override
    public String getName() {
        return "Not Available in Customer Database";
    }

    @Override
    public boolean isNil() {
        return true;
    }
}

public class RealCustomer extends AbstractCustomer {

    public RealCustomer(String name) {
```

```
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public boolean isNil() {
        return false;
    }
}
```

工厂类 *CustomerFactory* 基于客户传递的名字来返回 *RealCustomer* 或 *NullCustomer* 对象。

```
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();
    }
}
```

NullPatternDemo，我们的演示类使用 *CustomerFactory* 来演示空对象模式的用法。

```
   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());
   }
```

适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

我们通过下面的实例来演示适配器模式的使用。其中，音频播放器设备只能播放 mp3 文件，通过使用一个更高级的音频播放器来播放 vlc 和 mp4 文件。

意图：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

主要解决：主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

何时使用： 1、系统需要使用现有的类，而此类的接口不符合系统的需要。 2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。 3、通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

如何解决：继承或依赖（推荐）。

关键代码：适配器继承或依赖已有的对象，实现想要的目标接口。

应用实例： 1、美国电器 110V，中国 220V，就要有一个适配器将 110V 转化为 220V。 2、JAVA JDK 1.1 提供了 Enumeration 接口，而在 1.2 中提供了 Iterator 接口，想要使用 1.2 的 JDK，则要将以前系统的 Enumeration 接口转化为 Iterator 接口，这时就需要适配器模式。 3、在 LINUX 上运行 WINDOWS 程序。 4、JAVA 中的 jdbc。

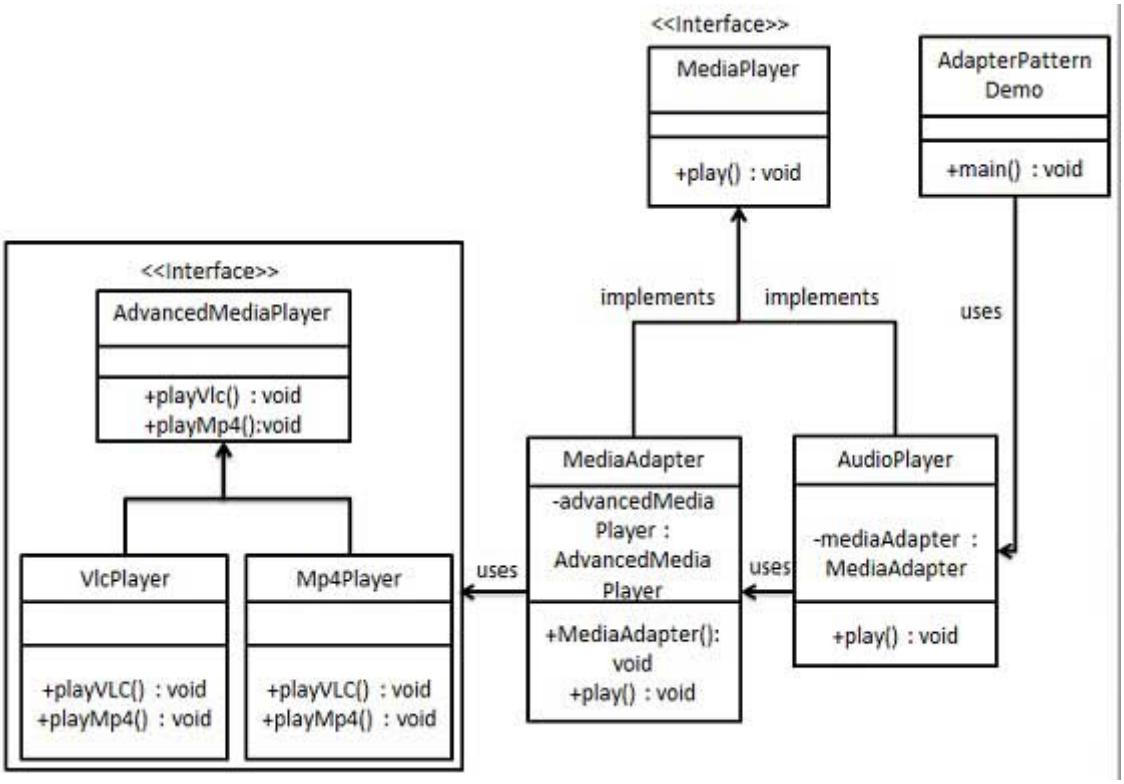
优点： 1、可以让任何两个没有关联的类一起运行。 2、提高了类的复用。 3、增加了类的透明度。 4、灵活性好。

缺点： 1、过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。 2.由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

使用场景：有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。

注意事项：适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。

实现



我们有一个 *MediaPlayer* 接口和一个实现了 *MediaPlayer* 接口的实体类 *AudioPlayer*。默认情况下，*AudioPlayer* 可以播放 mp3 格式的音频文件。

我们还有另一个接口 *AdvancedMediaPlayer* 和实现了 *AdvancedMediaPlayer* 接口的实体类。该类可以播放 vlc 和 mp4 格式的文件。

```
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}

public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}

public class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: "+ fileName);
    }

    @Override
    public void playMp4(String fileName) {
        //什么也不做
    }
}

public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //什么也不做
   }

   @Override
```

```
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: "+ fileName);
    }
}
```

我们想要让 *AudioPlayer* 播放其他格式的音频文件。为了实现这个功能，我们需要创建一个实现了 *MediaPlayer* 接口的适配器类 *MediaAdapter*，并使用 *AdvancedMediaPlayer* 对象来播放所需的格式。

```
public class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

    public MediaAdapter(String audioType){
        if(audioType.equalsIgnoreCase("vlc") ){
            advancedMusicPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("vlc")){
            advancedMusicPlayer.playVlc(fileName);
        }else if(audioType.equalsIgnoreCase("mp4")){
            advancedMusicPlayer.playMp4(fileName);
        }
    }
}
```

AudioPlayer 使用适配器类 *MediaAdapter* 传递所需的音频类型，不需要知道能播放所需格式音频的实际类。

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {

        //播放 mp3 音乐文件的内置支持
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: "+ fileName);
        }
        //mediaAdapter 提供了播放其他文件格式的支持
        else if(audioType.equalsIgnoreCase("vlc")
            || audioType.equalsIgnoreCase("mp4")){
            mediaAdapter = new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }
        else{
            System.out.println("Invalid media. "+
                audioType + " format not supported");
```

```
        }
    }
}
```

AdapterPatternDemo，我们的演示类使用 *AudioPlayer* 类来播放各种格式。

```
    public static void main(String[] args) {
        AudioPlayer audioPlayer = new AudioPlayer();

        audioPlayer.play("mp3", "beyond the horizon.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc");
        audioPlayer.play("avi", "mind me.avi");
    }
```

参考

https://www.runoob.com/design-pattern/adapter-pattern.html

代理模式

代理模式的定义：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。通俗的来讲代理模式就是我们生活中常见的中介。

为什么要用代理模式?

- **中介隔离作用**：在某些情况下，一个客户类不想或者不能直接引用一个委托对象，而代理类对象可以在客户类和委托对象之间起到中介的作用，其特征是代理类和委托类实现相同的接口。
- **开闭原则，增加功能**：代理类除了是客户类和委托类的中介之外，我们还可以通过给代理类增加额外的功能来扩展委托类的功能，这样做我们只需要修改代理类而不需要再修改委托类，符合代码设计的开闭原则。代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后对返回结果的处理等。代理类本身并不真正实现服务，而是同过调用委托类的相关方法，来提供特定的服务。真正的业务功能还是由委托类来实现，但是可以在业务功能执行的前后加入一些公共的服务。例如我们想给项目加入缓存、日志这些功能，我们就可以使用代理类来完成，而没必要打开已经封装好的委托类。

有哪几种代理模式?

我们有多种不同的方式来实现代理。如果按照代理创建的时期来进行分类的话， 可以分为两种：静态代理、动态代理。静态代理是由程序员创建或特定工具自动生成源代码，在对其编译。在程序员运行之前，代理类.class文件就已经被创建了。动态代理是在程序运行时通过反射机制动态创建的。

静态代理

1. 服务类接口及实现

```
/**
 * @Auther: dan gao
 * @Description:
 * @Date: 22:40 2018/1/9 0009
 */
public interface BuyHouse {
```

```
    void buyHouse();
}

/**
 * @Auther: dan gao
 * @Description:
 * @Date: 22:42 2018/1/9 0009
 */
public class BuyHouseImpl implements BuyHouse {

    @Override
    public void buyHouse() {
        System.out.println("我要买房");
    }
}
```

2. 代理类

```
import main.java.proxy.BuyHouse;

/**
 * @Auther: dan gao
 * @Description:
 * @Date: 22:43 2018/1/9 0009
 */
public class BuyHouseProxy implements BuyHouse {

    private BuyHouse buyHouse;

    public BuyHouseProxy(final BuyHouse buyHouse) {
        this.buyHouse = buyHouse;
    }

    @Override
    public void buyHouse() {
        System.out.println("买房前准备");
        buyHouse.buyHouse();
        System.out.println("买房后装修");

    }
}
```

3. 测试

```
    public static void main(String[] args) {
        BuyHouse buyHouse = new BuyHouseImpl();
        buyHouse.buyHouse();
        BuyHouseProxy buyHouseProxy = new BuyHouseProxy(buyHouse);
        buyHouseProxy.buyHouse();
    }
```

静态代理总结：

- 优点：可以做到在符合开闭原则的情况下对目标对象进行功能扩展。
- 缺点：我们得为每一个服务都得创建代理类，工作量太大，不易管理。同时接口一旦发生改变，代理类也得相应修改。

动态代理

在动态代理中我们不再需要再手动的创建代理类，我们只需要编写一个动态处理器就可以了。真正的代理对象由JDK再运行时为我们动态的来创建。

1. 第一步：编写动态处理器

```
/**
 * @Auther: dan gao
 * @Description:
 * @Date: 20:34 2018/1/12 0012
 */
public class DynamicProxyHandler implements InvocationHandler {

    private Object object;

    public DynamicProxyHandler(final Object object) {
        this.object = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        System.out.println("买房前准备");
        Object result = method.invoke(object, args);
        System.out.println("买房后装修");
        return result;
    }
}
```

2. 测试

```
    public static void main(String[] args) {
        BuyHouse buyHouse = new BuyHouseImpl();
        BuyHouse proxyBuyHouse = (BuyHouse)
Proxy.newProxyInstance(BuyHouse.class.getClassLoader(), new
                Class[]{BuyHouse.class}, new DynamicProxyHandler(buyHouse));
        proxyBuyHouse.buyHouse();
    }
```

注意*Proxy.newProxyInstance()*方法接受三个参数：

- `ClassLoader loader`:指定当前目标对象使用的类加载器,获取加载器的方法是固定的
- `Class[] interfaces`:指定目标对象实现的接口的类型,使用泛型方式确认类型
- `InvocationHandler`:指定动态处理器，执行目标对象的方法时,会触发事件处理器的方法

动态代理总结：虽然相对于静态代理，动态代理大大减少了我们的开发任务，同时减少了对业务接口的依赖，降低了耦合度。但是还是有一点点小小的遗憾之处，那就是它始终无法摆脱仅支持interface代理的桎梏，因为它的设计注定了这个遗憾。回想一下那些动态生成的代理类的继承关系图，它们已经注定有一个共同的父类叫Proxy。Java的继承机制注定了这些动态代理类们无法实现对class的动态代理，原因是多继承在Java中本质上就行不通。有很多条理由，人们可以否定对 class代理的必要性，但是同样有一些理由，相信支持class动态代理会更美好。接口和类的划分，本就不是很明显，只是到了Java中才变得如此的细化。如果只从方法的声明及是否被定义来考量，有一种两者的混合体，它的名字叫抽象类。实现对抽象类的动态代理，相信也有其内在的价值。此外，还有一些历史遗留的类，它们将因为没有实现任何接口而从此与动态代理永世无缘。如此种种，不得不说是一个小小的遗憾。但是，不完美并不等于不伟大，伟大是一种本质，Java动态代理就是佐例。

CGLIB代理

JDK实现动态代理需要实现类通过接口定义业务方法，对于没有接口的类，如何实现动态代理呢，这就需要CGLib了。CGLib采用了非常底层的字节码技术，其原理是通过字节码技术**为一个类创建子类**，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。但因为采用的是**继承**，所以**不能对final修饰的类进行代理**。JDK动态代理与CGLib动态代理均是实现Spring AOP的基础。

1. CGLIB代理类

```
/**
 * @Auther: dan gao
 * @Description:
 * @Date: 20:38 2018/1/16 0016
 */
public class CglibProxy implements MethodInterceptor {
    private Object target;
    public Object getInstance(final Object target) {
        this.target = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(this.target.getClass());
        enhancer.setCallback(this);
        return enhancer.create();
    }

    public Object intercept(Object object, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
        System.out.println("买房前准备");
        Object result = methodProxy.invokeSuper(object, args);
        System.out.println("买房后装修");
        return result;
    }
}
```

2. 测试

```
    public static void main(String[] args){
        BuyHouse buyHouse = new BuyHouseImpl();
        CglibProxy cglibProxy = new CglibProxy();
        BuyHouse buyHouseCglibProxy = (BuyHouse)
cglibProxy.getInstance(buyHouse);
        buyHouseCglibProxy.buyHouse();
    }
```

CGLIB代理总结： CGLIB创建的动态代理对象比JDK创建的动态代理对象的性能更高，但是CGLIB创建代理对象时所花费的时间却比JDK多得多。所以对于单例的对象，因为无需频繁创建对象，用CGLIB合适，反之使用JDK方式要更为合适一些。同时由于CGLib由于是采用动态创建子类的方法，对于final修饰的方法无法进行代理。

参考

https://www.cnblogs.com/daniels/p/8242592.html

装饰器模式

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。

意图：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

主要解决：一般的，我们为了扩展一个类经常使用继承方式实现，由于继承为类引入静态特征，并且随着扩展功能的增多，子类会很膨胀。

何时使用：在不想增加很多子类的情况下扩展类。

如何解决：将具体功能职责划分，同时继承装饰者模式。

关键代码： 1、Component 类充当抽象角色，不应该具体实现。 2、修饰类引用和继承 Component 类，具体扩展类重写父类方法。

应用实例： 1、孙悟空有 72 变，当他变成"庙宇"后，他的根本还是一只猴子，但是他又有了庙宇的功能。 2、不论一幅画有没有画框都可以挂在墙上，但是通常都是有画框的，并且实际上是画框被挂在墙上。在挂在墙上之前，画可以被蒙上玻璃，装到框子里；这时画、玻璃和画框形成了一个物体。

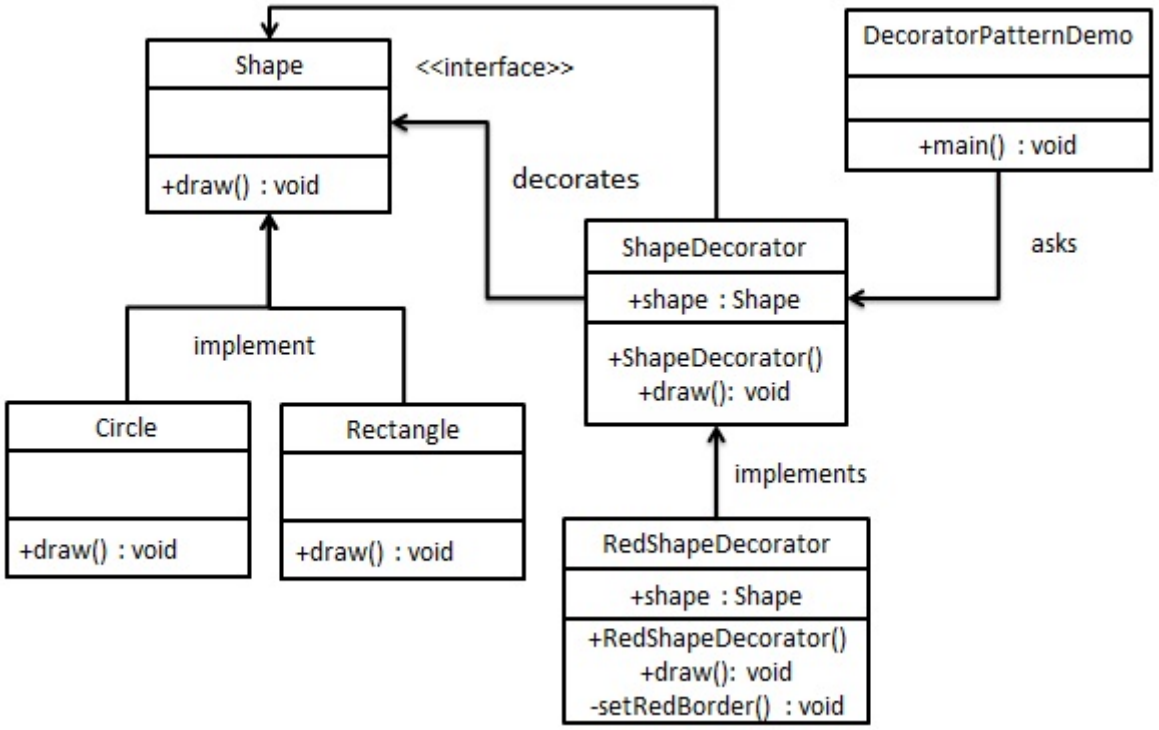
优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

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。

实现



我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。

```
public interface Shape {
    void draw();
}

public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Circle");
    }
}

public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}
```

然后我们创建一个实现了 *Shape* 接口的抽象装饰类 *ShapeDecorator*，并把 *Shape* 对象作为它的实例变量。

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

RedShapeDecorator 是实现了 *ShapeDecorator* 的实体类。

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

DecoratorPatternDemo，我们的演示类使用 *RedShapeDecorator* 来装饰 *Shape* 对象。

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

参考

https://www.runoob.com/design-pattern/decorator-pattern.html

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。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

意图：将抽象部分与实现部分分离，使它们都可以独立的变化。

主要解决：在有多种可能会变化的情况下，用继承会造成类爆炸问题，扩展起来不灵活。

何时使用：实现系统可能有多个角度分类，每一种角度都可能变化。

如何解决：把这种多角度分类分离出来，让它们独立变化，减少它们之间耦合。

关键代码：抽象类依赖实现类。

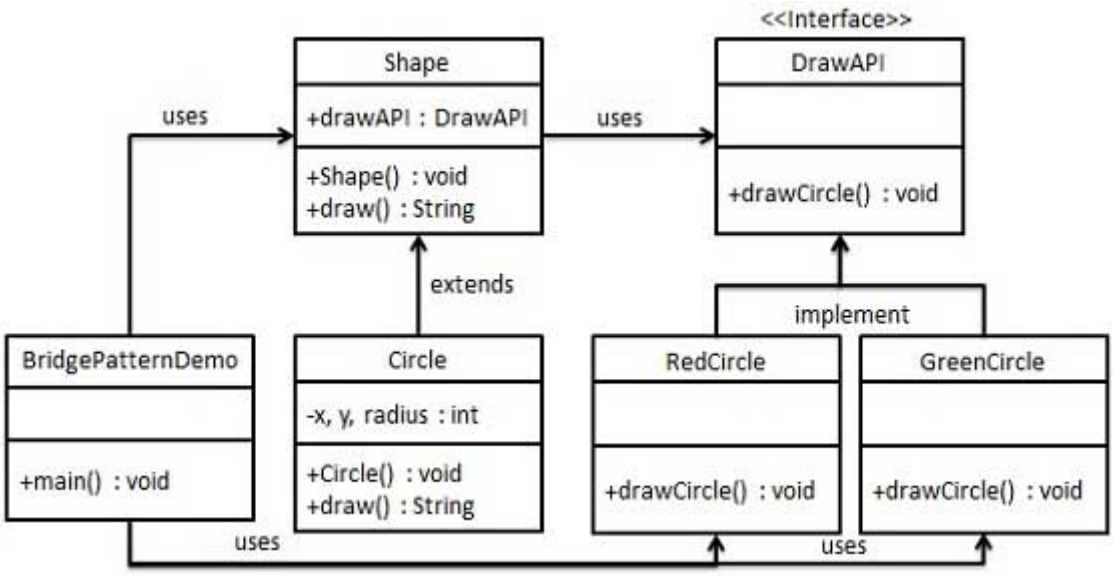
优点： 1、抽象和实现的分离。 2、优秀的扩展能力。 3、实现细节对客户透明。

缺点：桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程。

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。

实现



我们有一个作为桥接实现的 *DrawAPI* 接口和实现了 *DrawAPI* 接口的实体类 *RedCircle*、*GreenCircle*。

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}

public class RedCircle implements DrawAPI {
```

```
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}

public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}
```

Shape 是一个抽象类，将使用 *DrawAPI* 的对象。

```
public abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}

public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

BridgePatternDemo，我们的演示类使用 *Shape* 类来画出不同颜色的圆。

```
    public static void main(String[] args) {
        Shape redCircle = new Circle(100, 100, 10, new RedCircle());
        Shape greenCircle = new Circle(100, 100, 10, new GreenCircle());

        redCircle.draw();
        greenCircle.draw();
    }
```

参考

https://www.runoob.com/design-pattern/bridge-pattern.html

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。

这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

意图：将对象组合成树形结构以表示"部分-整体"的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

主要解决：它在我们树型结构的问题中，模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

何时使用： 1、您想表示对象的部分-整体层次结构（树形结构）。 2、您希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

如何解决：树枝和叶子实现统一接口，树枝内部组合该接口。

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。

应用实例： 1、算术表达式包括操作数、操作符和另一个操作数，其中，另一个操作符也可以是操作数、操作符和另一个操作数。 2、在 JAVA AWT 和 SWING 中，对于 Button 和 Checkbox 是树叶，Container 是树枝。

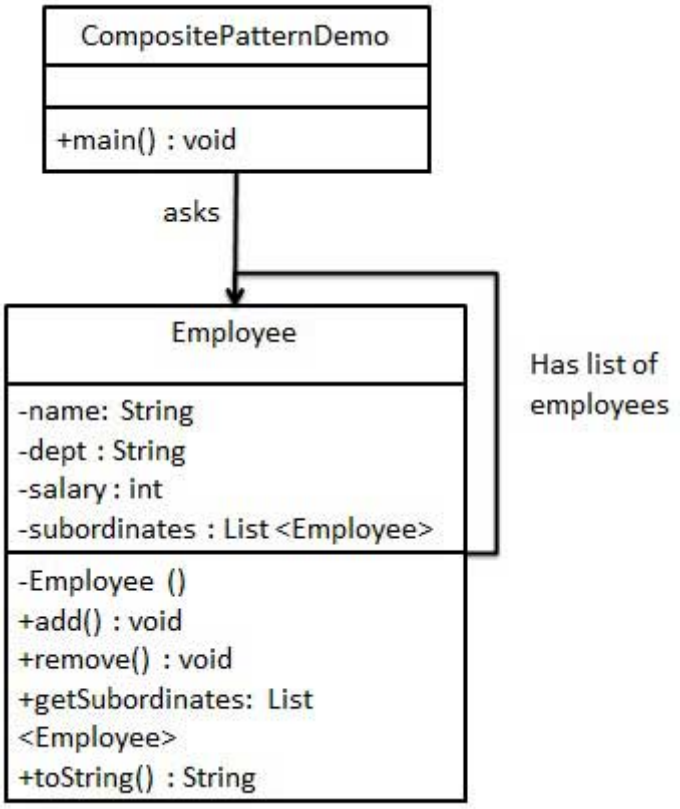
优点： 1、高层模块调用简单。 2、节点自由增加。

缺点：在使用组合模式时，其叶子和树枝的声明都是实现类，而不是接口，违反了依赖倒置原则。

使用场景：部分、整体场景，如树形菜单，文件、文件夹的管理。

注意事项：定义时为具体类。

实现



我们有一个类 *Employee*，该类被当作组合模型类。

```
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+" ]");
   }
```

```
}
```

CompositePatternDemo，我们的演示类使用 *Employee* 类来添加部门层次结构，并打印所有员工。

```
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);
       }
    }
 }
```

参考

https://www.runoob.com/design-pattern/composite-pattern.html

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。

意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

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。

何时使用： 1、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。 2、定义系统的入口。

如何解决：客户端不与系统耦合，外观类与系统耦合。

关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。

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。 2、JAVA 的三层开发模式。

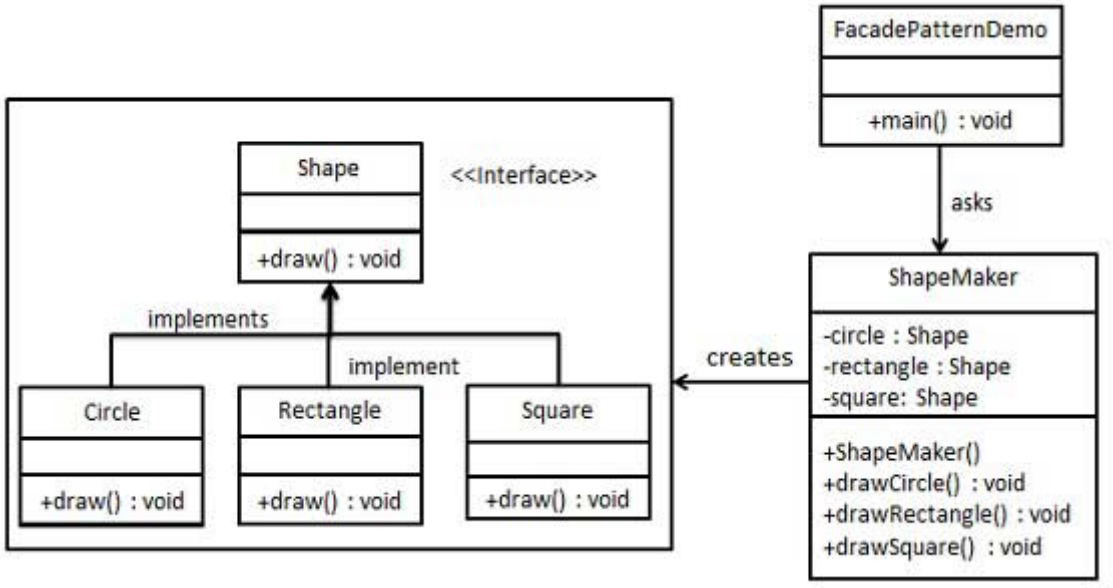
优点： 1、减少系统相互依赖。 2、提高灵活性。 3、提高了安全性。

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。

注意事项：在层次化结构中，可以使用外观模式定义系统中每一层的入口。

实现



我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类。

```
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Rectangle::draw()");
   }
}

public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Square::draw()");
   }
}

public class Circle implements Shape {
```

```
   @Override
   public void draw() {
      System.out.println("Circle::draw()");
   }
}
```

下一步是定义一个外观类 *ShapeMaker*。*ShapeMaker* 类使用实体类来代表用户对这些类的调用。

```
public class ShapeMaker {
   private Shape circle;
   private Shape rectangle;
   private Shape square;

   public ShapeMaker() {
      circle = new Circle();
      rectangle = new Rectangle();
      square = new Square();
   }

   public void drawCircle(){
      circle.draw();
   }
   public void drawRectangle(){
      rectangle.draw();
   }
   public void drawSquare(){
      square.draw();
   }
}
```

FacadePatternDemo，我们的演示类使用 *ShapeMaker* 类来显示结果。

```
   public static void main(String[] args) {
      ShapeMaker shapeMaker = new ShapeMaker();

      shapeMaker.drawCircle();
      shapeMaker.drawRectangle();
      shapeMaker.drawSquare();
   }
```

参考

https://www.runoob.com/design-pattern/facade-pattern.html

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。

意图：运用共享技术有效地支持大量细粒度的对象。

主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。

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。

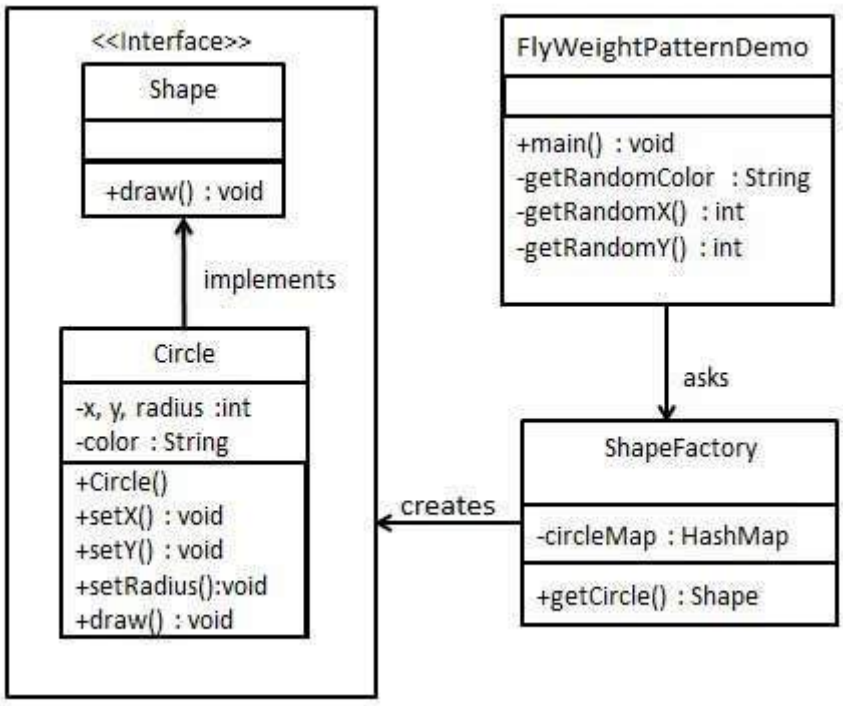
优点：大大减少对象的创建，降低系统的内存，使效率提高。

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。

使用场景： 1、系统有大量相似对象。 2、需要缓冲池的场景。

注意事项： 1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。 2、这些类必须有一个工厂对象加以控制。

实现



我们将创建一个 *Shape* 接口和实现了 *Shape* 接口的实体类 *Circle*。

```
public interface Shape {
   void draw();
}

public class Circle implements Shape {
```

```
   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;
   }

   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color
         +", x : " + x +", y :" + y +", radius :" + radius);
   }
}
```

下一步是定义工厂类 *ShapeFactory*。

ShapeFactory 有一个 *Circle* 的 *HashMap*，其中键名为 *Circle* 对象的颜色。无论何时接收到请求，都会创建一个特定颜色的圆。*ShapeFactory* 检查它的 *HashMap* 中的 circle 对象，如果找到 *Circle* 对象，则返回该对象，否则将创建一个存储在 hashmap 中以备后续使用的新对象，并把该对象返回到客户端。

```
public class ShapeFactory {
   private static final HashMap<String, Shape> circleMap = new HashMap<>();

   public static Shape 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;
   }
}
```

FlyWeightPatternDemo，我们的演示类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*red / green / blue/ black / white*），以便获取它所需对象的颜色。

```
  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)ShapeFactory.getCircle(getRandomColor());
          circle.setX(getRandomX());
          circle.setY(getRandomY());
          circle.setRadius(100);
          circle.draw();
       }
    }
    private static String getRandomColor() {
       return colors[(int)(Math.random()*colors.length)];
    }
    private static int getRandomX() {
       return (int)(Math.random()*100 );
    }
    private static int getRandomY() {
       return (int)(Math.random()*100);
    }
```

参考

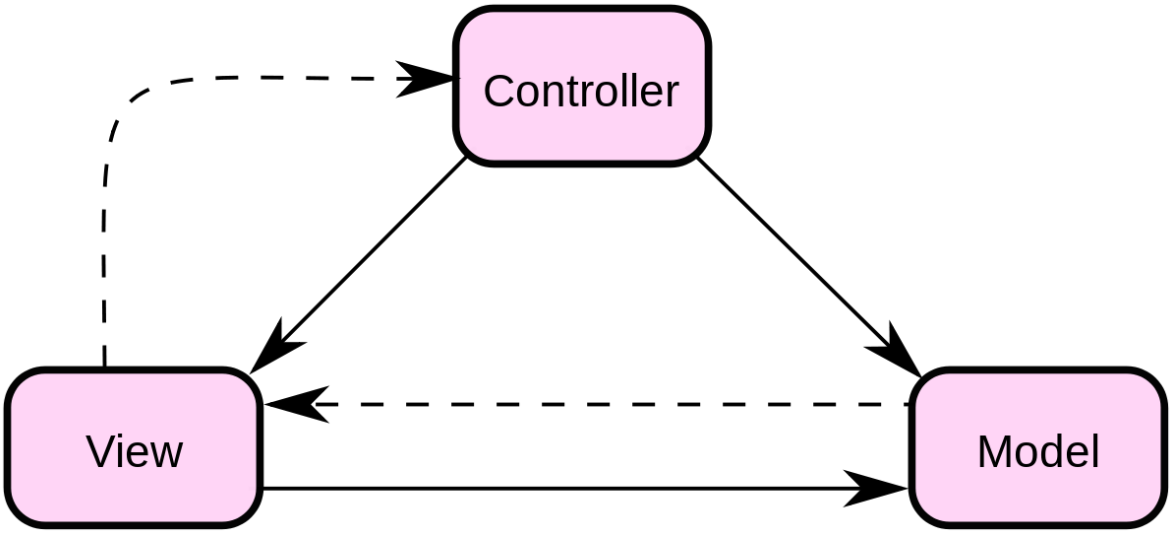
https://www.runoob.com/design-pattern/flyweight-pattern.html

J2EE 模式

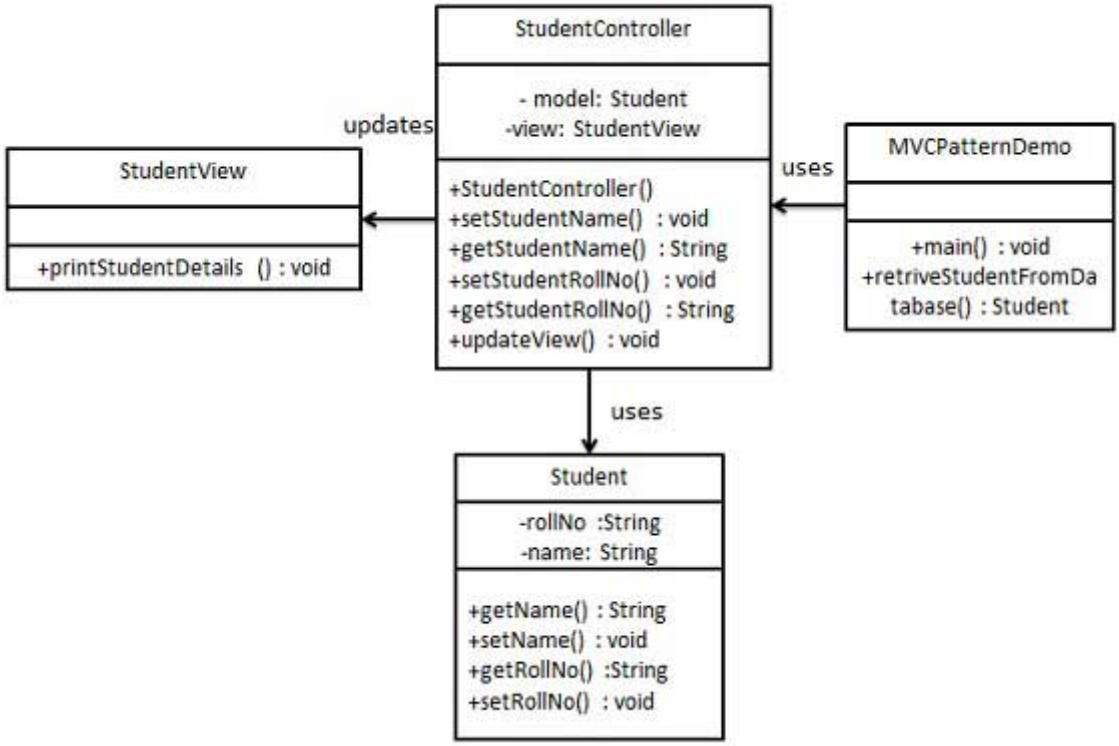
MVC 模式

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。



实现



我们将创建一个作为模型的 *Student* 对象。

```
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;
   }
}
```

StudentView 是一个把学生详细信息输出到控制台的视图类，*StudentController* 是负责存储数据到 *Student* 对象中的控制器类，并相应地更新视图 *StudentView*。

```
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);
    }
}

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());
   }
}
```

MVCPatternDemo，我们的演示类使用 *StudentController* 来演示 MVC 模式的用法。

```
public static void main(String[] args) {

      //从数据库获取学生记录
      Student model  = retrieveStudentFromDatabase();

      //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 retrieveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
```

参考

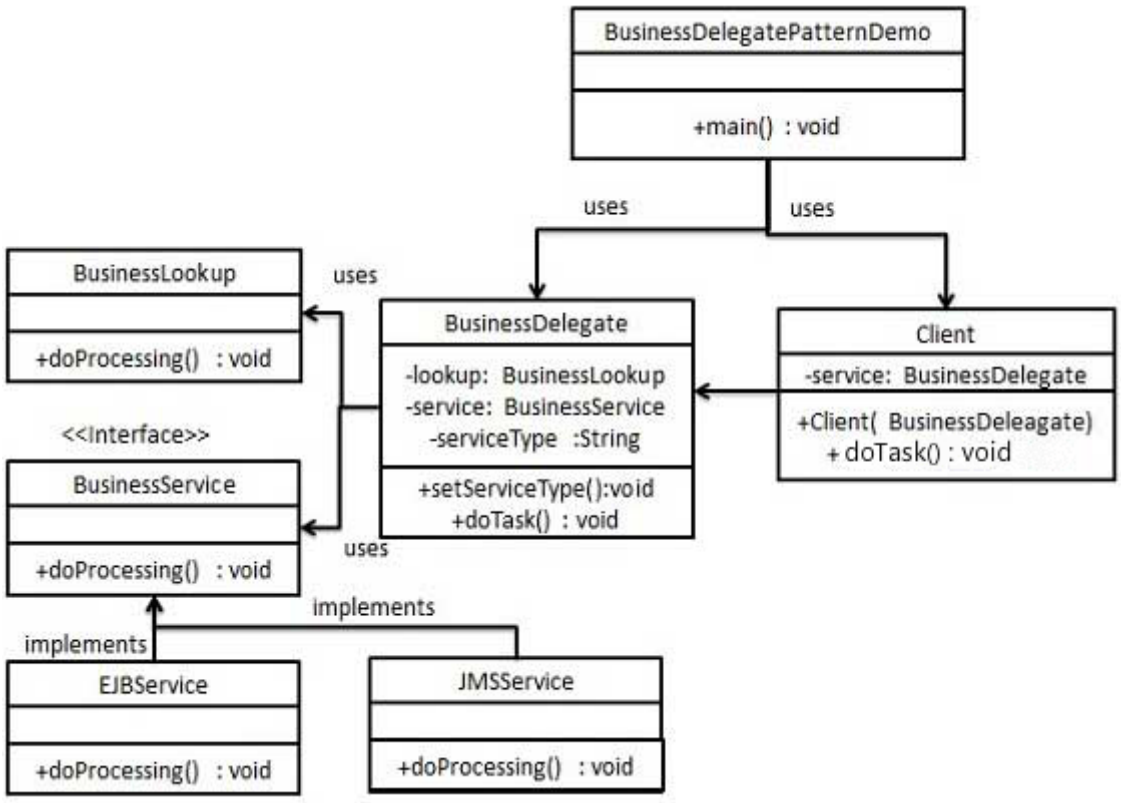
https://www.runoob.com/design-pattern/mvc-pattern.html

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。

实现



我们将创建 *Client*、*BusinessDelegate*、*BusinessService*、*LookUpService*、*JMSService* 和 *EJBService* 来表示业务代表模式中的各种实体。

```
public interface BusinessService {
    public void doProcessing();
}

public class EJBService implements BusinessService {

   @Override
   public void doProcessing() {
      System.out.println("Processing task by invoking EJB Service");
   }
}

public class JMSService implements BusinessService {
```

```
   @Override
   public void doProcessing() {
      System.out.println("Processing task by invoking JMS Service");
   }
}

public class BusinessLookUp {
   public BusinessService getBusinessService(String serviceType){
      if(serviceType.equalsIgnoreCase("EJB")){
         return new EJBService();
      }else {
         return new JMSService();
      }
   }
}

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();
   }
}

public class Client {

   BusinessDelegate businessService;

   public Client(BusinessDelegate businessService){
      this.businessService  = businessService;
   }

   public void doTask(){
      businessService.doTask();
   }
}
```

BusinessDelegatePatternDemo，我们的演示类使用 *BusinessDelegate* 和 *Client* 来演示业务代表模式的用法。

```
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();
   }
```

参考

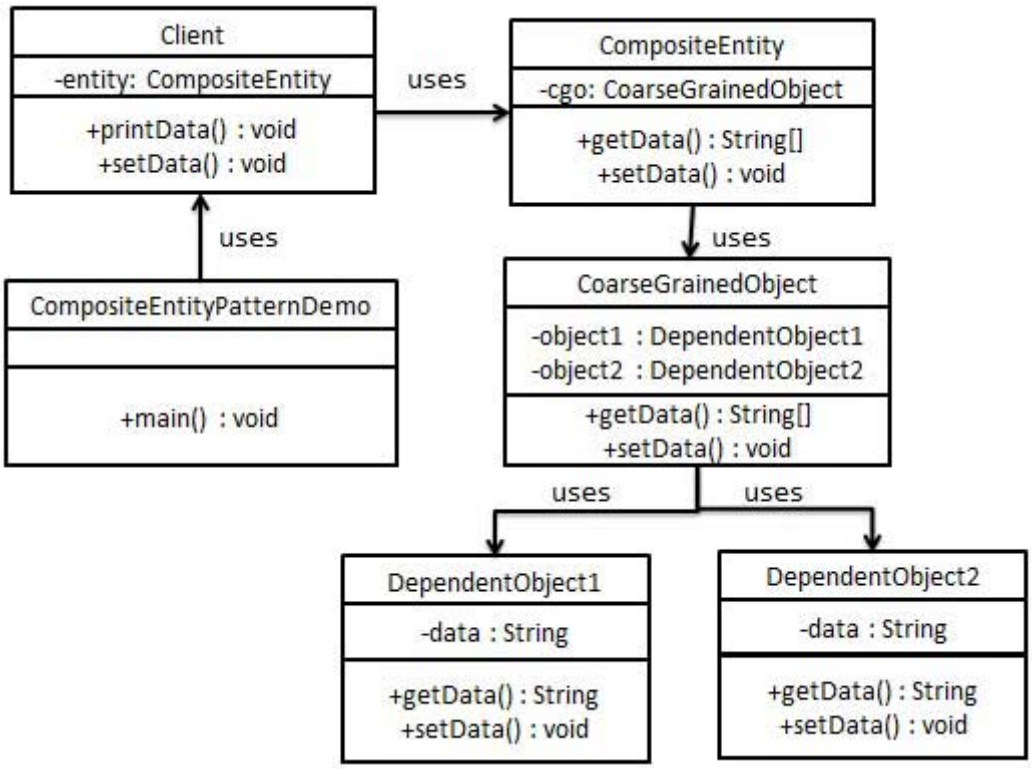
https://www.runoob.com/design-pattern/business-delegate-pattern.html

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。

实现



我们将创建作为组合实体的 *CompositeEntity* 对象。*CoarseGrainedObject* 是一个包含依赖对象的类。

创建依赖对象。

```
public class DependentObject1 {

    private String data;

    public void setData(String data){
        this.data = data;
    }

    public String getData(){
        return data;
    }
}

public class DependentObject2 {

   private String data;

   public void setData(String data){
      this.data = data;
   }

   public String getData(){
      return data;
   }
}
```

创建粗粒度对象。

```
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()};
   }
}
```

创建组合实体。

```
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();
    }
}
```

创建使用组合实体的客户端类。

```
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);
    }
}
```

CompositeEntityPatternDemo，我们的演示类使用 *Client* 类来演示组合实体模式的用法。

```
    public static void main(String[] args) {
        Client client = new Client();
        client.setData("Test", "Data");
        client.printData();
        client.setData("Second Test", "Data1");
        client.printData();
    }
```

参考

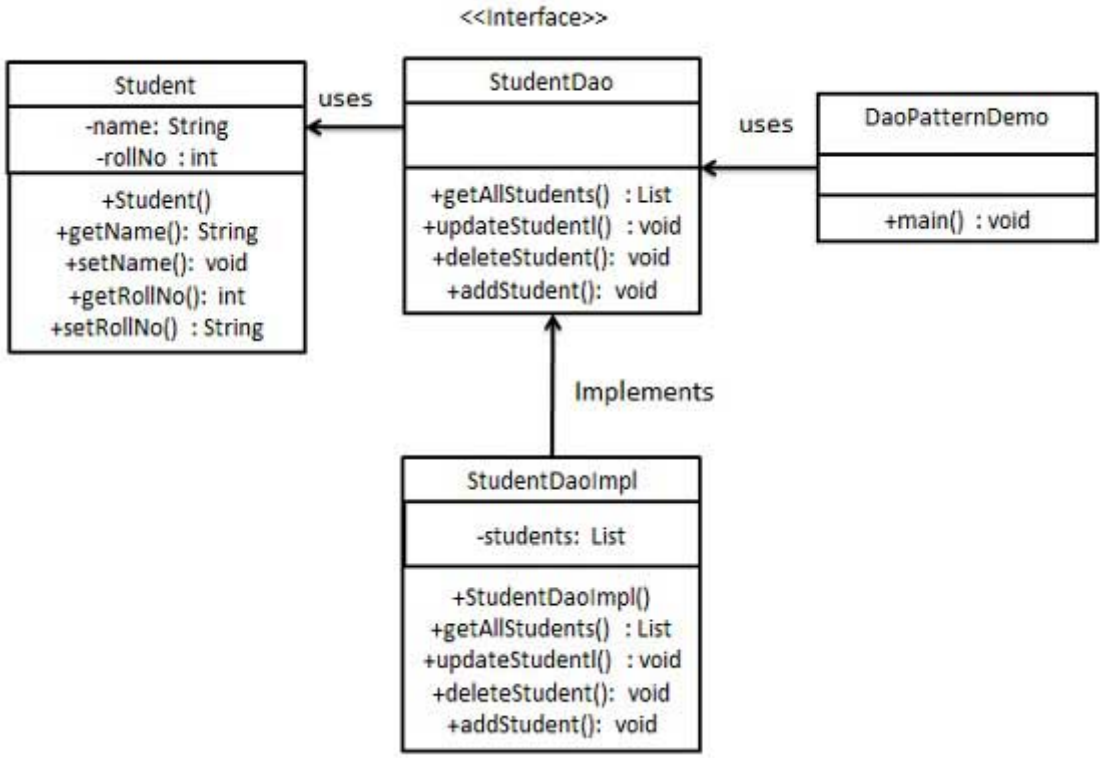
https://www.runoob.com/design-pattern/composite-entity-pattern.html

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。

实现



我们将创建一个作为模型对象或数值对象的 *Student* 对象。

```
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;
    }
}
```

StudentDao 是数据访问对象接口。*StudentDaoImpl* 是实现了数据访问对象接口的实体类。

```
public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
```

```
}


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");
   }
}
```

DaoPatternDemo，我们的演示类使用 *StudentDao* 来演示数据访问对象模式的用法。

```
   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()+" ]");
   }
```

参考

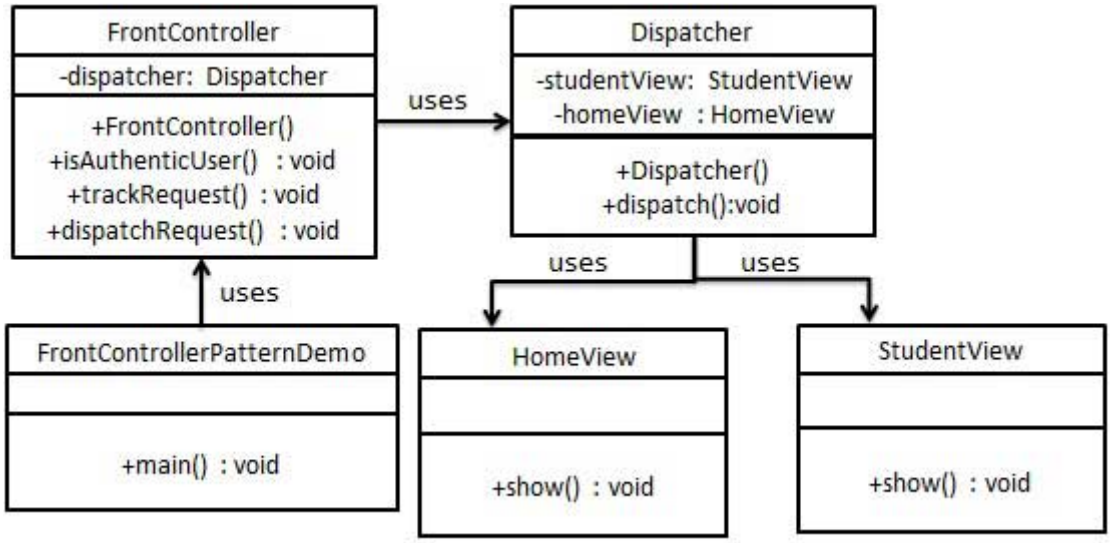
https://www.runoob.com/design-pattern/data-access-object-pattern.html

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。

实现



HomeView 和 *StudentView* 表示各种为前端控制器接收到的请求而创建的视图。

```
public class HomeView {
   public void show(){
      System.out.println("Displaying Home Page");
   }
}

public class StudentView {
   public void show(){
      System.out.println("Displaying Student Page");
   }
}
```

我们将创建 *FrontController*、*Dispatcher* 分别当作前端控制器和调度器。

```
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();
      }
   }
}

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);
        }
    }
}
```

FrontControllerPatternDemo，我们的演示类使用 *FrontController* 来演示前端控制器设计模式。

```
   public static void main(String[] args) {
      FrontController frontController = new FrontController();
      frontController.dispatchRequest("HOME");
      frontController.dispatchRequest("STUDENT");
   }
```

参考

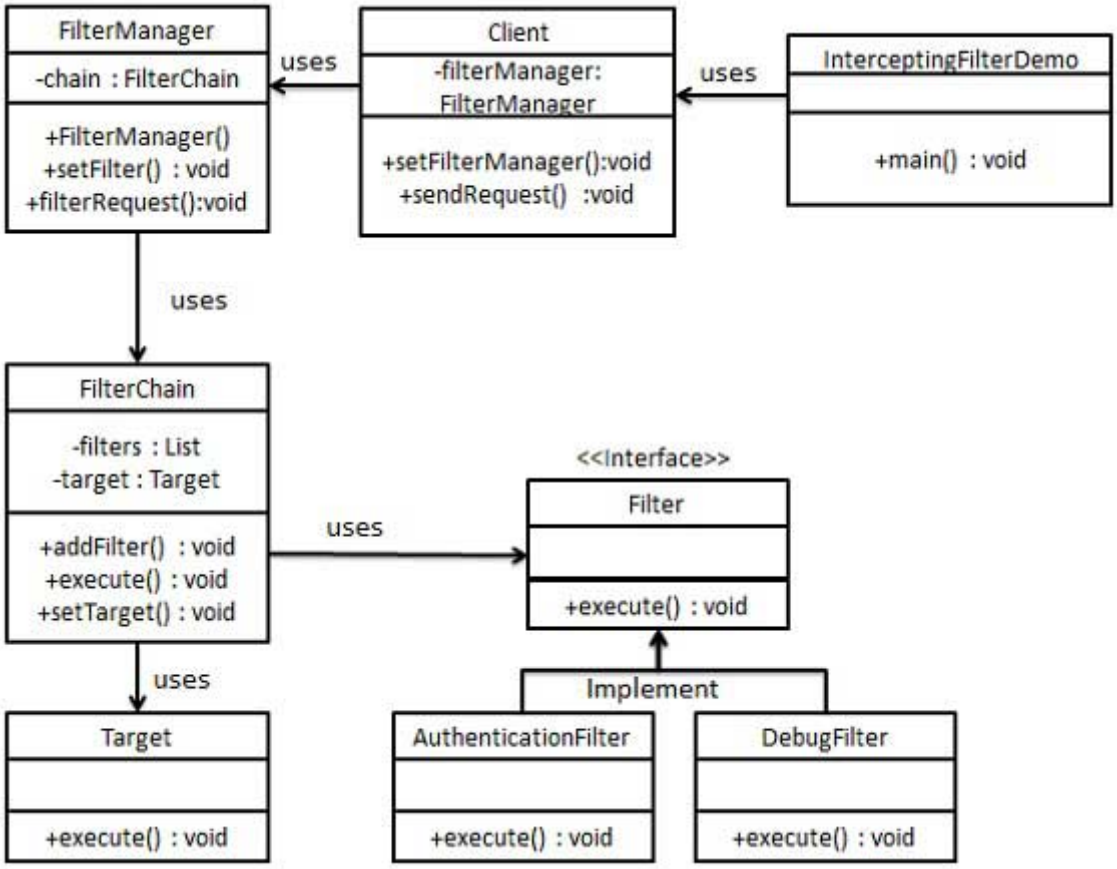
https://www.runoob.com/design-pattern/front-controller-pattern.html

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。

实现



我们将创建 *FilterChain*、*FilterManager*、*Target*、*Client* 作为表示实体的各种对象。*AuthenticationFilter* 和 *DebugFilter* 表示实体过滤器。

创建过滤器接口 Filter。创建实体过滤器。

```
public interface Filter {
    public void execute(String request);
}

public class AuthenticationFilter implements Filter {
   public void execute(String request){
      System.out.println("Authenticating request: " + request);
   }
}

public class DebugFilter implements Filter {
   public void execute(String request){
```

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

创建 Target。

```
public class Target {
   public void execute(String request){
      System.out.println("Executing request: " + request);
   }
}
```

创建过滤器链。

```
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;
   }
}
```

创建过滤管理器。

```
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);
   }
}
```

创建客户端 Client。

```
public class Client {
   FilterManager filterManager;

   public void setFilterManager(FilterManager filterManager){
      this.filterManager = filterManager;
   }

   public void sendRequest(String request){
      filterManager.filterRequest(request);
   }
}
```

InterceptingFilterDemo，我们的演示类使用 *Client* 来演示拦截过滤器设计模式。

```
   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;
   }
```

参考

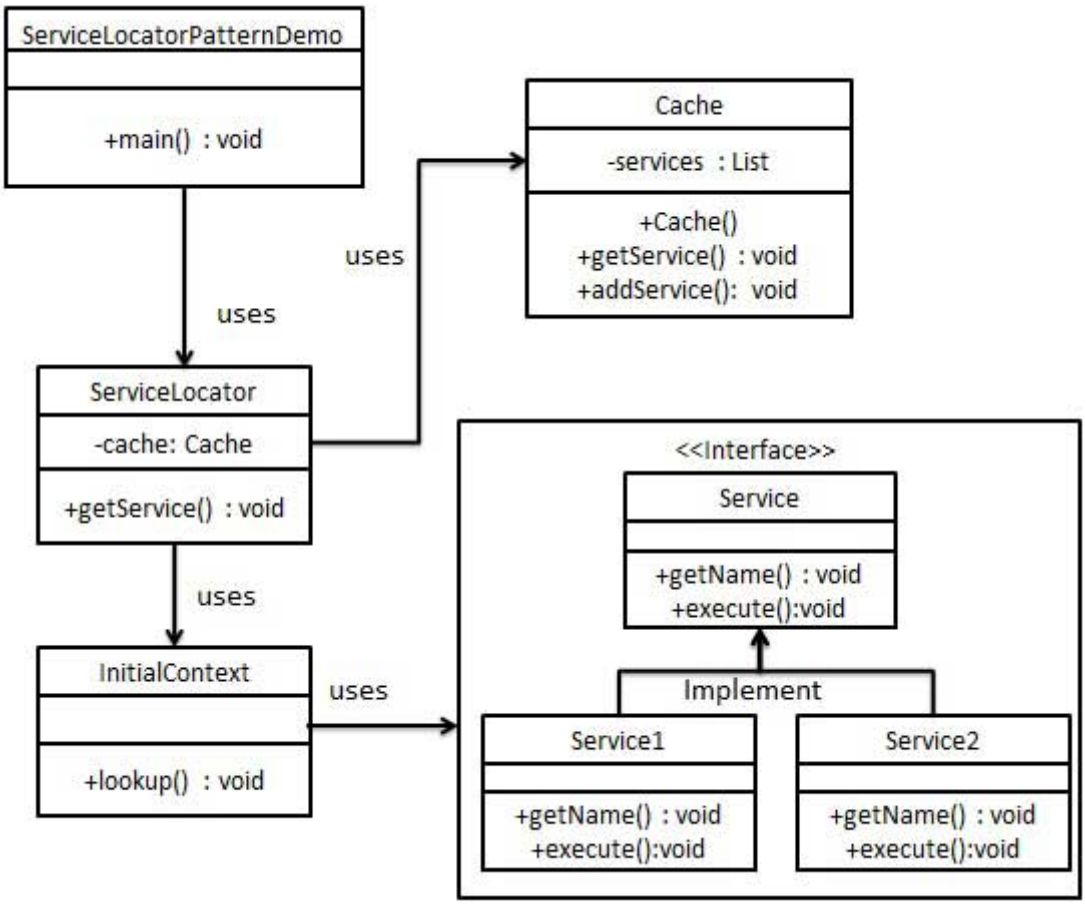
https://www.runoob.com/design-pattern/intercepting-filter-pattern.html

服务定位器模式

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。

实现



我们将创建 *ServiceLocator*、*InitialContext*、*Cache*、*Service* 作为表示实体的各种对象。*Service1* 和 *Service2* 表示实体服务。

```
public interface Service {
   public String getName();
   public void execute();
}

public class Service1 implements Service {
   public void execute(){
      System.out.println("Executing Service1");
   }

   @Override
   public String getName() {
      return "Service1";
   }
}

public class Service2 implements Service {
   public void execute(){
      System.out.println("Executing Service2");
   }

   @Override
   public String getName() {
      return "Service2";
   }
}

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;
   }
}



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);
      }
   }
}

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;
    }
}
```

ServiceLocatorPatternDemo，我们的演示类在这里是作为一个客户端，将使用 *ServiceLocator* 来演示服务定位器设计模式。

```
   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();
    }
```

参考

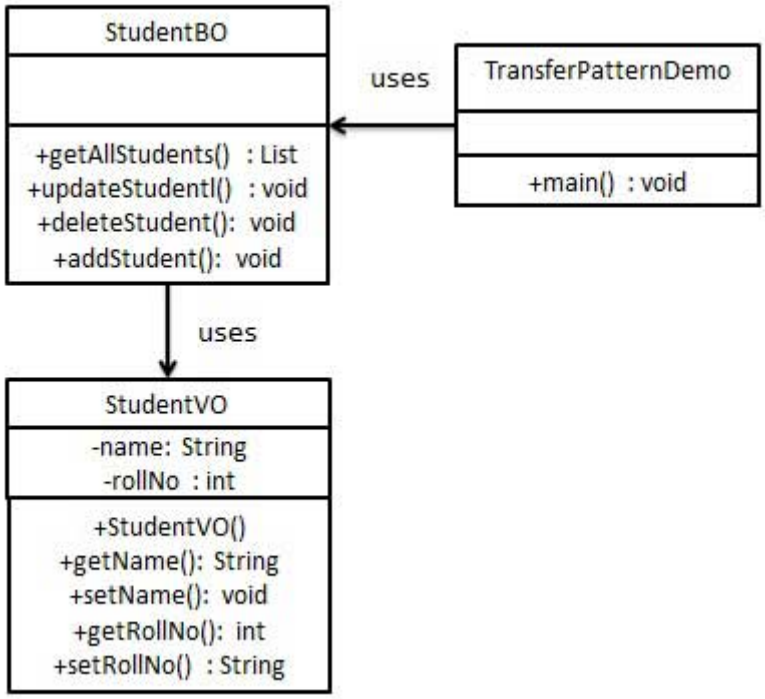
https://www.runoob.com/design-pattern/service-locator-pattern.html

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。

实现



我们将创建一个作为业务对象的 *StudentBO* 和作为传输对象的 *StudentVO*，它们都代表了我们的实体。

```
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;
   }
}

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");
    }
}
```

TransferObjectPatternDemo，我们的演示类在这里是作为一个客户端，将使用 *StudentBO* 和 *Student* 来演示传输对象设计模式。

```
   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()+" ]");
    }
```

参考

https://www.runoob.com/design-pattern/transfer-object-pattern.html