深入浅出Java23种设计模式

写在前面

> 在「**冰河技术**」微信公众号中【设计模式专题】更新完毕已有一段时间了。不少小伙伴在我微信上留言说：冰河，你能不能把【设计模式专题】的文章汇总成PDF文档呢？一直没有时间整理，最近在公众号后台发现很多读者会在公众号回复“设计模式”关键字，但是公众号回复的内容却不是大家想要的，今天，终于抽空将「**冰河技术**」微信公众号中【设计模式专题】发布的文章，整理成《深入浅出Java 23种设计模式》PDF文档，并开放下载链接，免费供小伙伴们下载学习。

关于作者

冰河，大数据架构师，编程专家，Mykit系列开源框架作者，多年来致力于分布式系统架构、微服务、分布式数据库、分布式事务与大数据技术的研究，曾主导过众多分布式系统、微服务及大数据项目的架构设计、研发和实施落地。在高并发、高可用、高可扩展性、高可维护性和大数据等领域拥有丰富的架构经验。对Hadoop，Storm，Spark，Flink等大数据框架源码进行过深度分析，并具有丰富的实战经验。目前在研究云原生。公众号【冰河技术】作者，《海量数据处理与大数据技术实战》、《MySQL开发、优化与运维实战》作者，基于最终消息可靠性的开源分布式事务框架mykit-transaction-message作者。

- 个人微信：sun_shine_lyz
- 公众号：冰河技术
- github地址：https://github.com/sunshinelyz

如何持续提升自身能力

如何获取PDF

关注「 **冰河技术** 」微信公众号，后台回复 “设计模式” 关键字领取《深入浅出Java 23种设计模式》PDF文档。

关于设计模式

在软件开发的历程中，正是无数前辈们本着追求完美的架构设计和代码设计的初心。经过无数前辈们的探索和努力，逐渐形成了一系列的设计原则和设计模式。

对于Java语言来说，通常包含有6大设计原则和23种设计模式，这些都是前辈们对于开发思想的结晶。我们学习和理解这些设计原则和设计模式，深入掌握其实现原理和使用场景，能够更好的设计我们的系统架构。编写出具有高性能、高并发、高可用、高可扩展性和高可维护性的代码。

在Java的常见的23种设计模式中，大体上可以分为创建型模式、结构型模式和行为型模式三大类。本文就对Java中的23种涉及模式进行汇总，方便记录和查阅。

创建型模式

单例模式

看几个单例对象的示例代码，其中有些代码是线程安全的，有些则不是线程安全的，需要大家细细品味，这些代码也是冰河本人在高并发环境下测试验证过的。

- 代码一：SingletonExample1

  这个类是懒汉模式，并且是线程不安全的

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 懒汉模式，单例实例在第一次使用的时候进行创建，这个类是线程不安全的
 */
public class SingletonExample1 {

    private SingletonExample1(){}

    private static SingletonExample1 instance = null;

    public static SingletonExample1 getInstance(){
        //多个线程同时调用，可能会创建多个对象
        if (instance == null){
```

```
            instance = new SingletonExample1();
        }
        return instance;
    }
}
```

- 代码二：SingletonExample2

饿汉模式，单例实例在类装载的时候进行创建，是线程安全的

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 饿汉模式，单例实例在类装载的时候进行创建，是线程安全的
 */
public class SingletonExample2 {

    private SingletonExample2(){}

    private static SingletonExample2 instance = new SingletonExample2();

    public static SingletonExample2 getInstance(){
        return instance;
    }
}
```

- 代码三：SingletonExample3

懒汉模式，单例实例在第一次使用的时候进行创建，这个类是线程安全的，但是这个写法不推荐

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 懒汉模式，单例实例在第一次使用的时候进行创建，这个类是线程安全的，但是这个写法不推荐
 */
public class SingletonExample3 {

    private SingletonExample3(){}

    private static SingletonExample3 instance = null;

    public static synchronized SingletonExample3 getInstance(){
        if (instance == null){
            instance = new SingletonExample3();
        }
        return instance;
    }
}
```

- 代码四：SingletonExample4

懒汉模式（双重锁同步锁单例模式），单例实例在第一次使用的时候进行创建，但是，这个类不是线程安全的！！！！！

```java
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 懒汉模式（双重锁同步锁单例模式）
 *              单例实例在第一次使用的时候进行创建，这个类不是线程安全的
 */
public class SingletonExample4 {

    private SingletonExample4(){}

    private static SingletonExample4 instance = null;

    //线程不安全
    //当执行instance = new SingletonExample4();这行代码时，CPU会执行如下指令：
    //1.memory = allocate() 分配对象的内存空间
    //2.ctorInstance() 初始化对象
    //3.instance = memory 设置instance指向刚分配的内存
    //单纯执行以上三步没啥问题，但是在多线程情况下，可能会发生指令重排序。
    // 指令重排序对单线程没有影响，单线程下CPU可以按照顺序执行以上三个步骤，但是在多线程下，如果发生了指令重排序，则会打乱上面的三个步骤。

    //如果发生了JVM和CPU优化，发生重排序时，可能会按照下面的顺序执行：
    //1.memory = allocate() 分配对象的内存空间
    //3.instance = memory 设置instance指向刚分配的内存
    //2.ctorInstance() 初始化对象


    //假设目前有两个线程A和B同时执行getInstance()方法，A线程执行到instance = new SingletonExample4(); B线程刚执行到第一个 if (instance == null){处，
    //如果按照1.3.2的顺序，假设线程A执行到3.instance = memory 设置instance指向刚分配的内存，此时，线程B判断instance已经有值，就会直接return instance;
    //而实际上，线程A还未执行2.ctorInstance() 初始化对象，也就是说线程B拿到的instance对象还未进行初始化，这个未初始化的instance对象一旦被线程B使用，就会出现问题。


    public static SingletonExample4 getInstance(){
        if (instance == null){
            synchronized (SingletonExample4.class){
                if(instance == null){
                    instance = new SingletonExample4();
                }
            }
        }
        return instance;
    }
}
```

线程不安全分析如下：
当执行instance = new SingletonExample4();这行代码时，CPU会执行如下指令：
1.memory = allocate() 分配对象的内存空间
2.ctorInstance() 初始化对象
3.instance = memory 设置instance指向刚分配的内存
单纯执行以上三步没啥问题，但是在多线程情况下，可能会发生指令重排序。
指令重排序对单线程没有影响，单线程下CPU可以按照顺序执行以上三个步骤，但是在多线程下，如果发生了指令重排序，则会打乱上面的三个步骤。

如果发生了JVM和CPU优化，发生重排序时，可能会按照下面的顺序执行：
1.memory = allocate() 分配对象的内存空间
3.instance = memory 设置instance指向刚分配的内存
2.ctorInstance() 初始化对象

假设目前有两个线程A和B同时执行getInstance()方法，A线程执行到instance = new SingletonExample4(); B线程刚执行到第一个 if (instance == null){处，如果按照1.3.2的顺序，假设线程A执行到3.instance = memory 设置instance指向刚分配的内存，此时，线程B判断instance已经有值，就会直接return instance;而实际上，线程A还未执行2.ctorInstance() 初始化对象，也就是说线程B拿到的instance对象还未进行初始化，这个未初始化的instance对象一旦被线程B使用，就会出现问题。

- 代码五:SingletonExample5

懒汉模式（双重锁同步锁单例模式）单例实例在第一次使用的时候进行创建，这个类是线程安全的，使用的是 volatile + 双重检测机制来禁止指令重排达到线程安全

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 懒汉模式（双重锁同步锁单例模式）
 *              单例实例在第一次使用的时候进行创建，这个类是线程安全的
 */
public class SingletonExample5 {

    private SingletonExample5(){}

    //单例对象  volatile + 双重检测机制来禁止指令重排
    private volatile static SingletonExample5 instance = null;

    public static SingletonExample5 getInstance(){
        if (instance == null){
            synchronized (SingletonExample5.class){
                if(instance == null){
                    instance = new SingletonExample5();
                }
            }
        }
        return instance;
    }
}
```

- 代码六：SingletonExample6

饿汉模式，单例实例在类装载的时候（使用静态代码块）进行创建，是线程安全的

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 饿汉模式，单例实例在类装载的时候进行创建，是线程安全的
 */
public class SingletonExample6 {

    private SingletonExample6(){}

    private static SingletonExample6 instance = null;
```

```
    static {
        instance = new SingletonExample6();
    }

    public static SingletonExample6 getInstance(){
        return instance;
    }
}
```

- 代码七：SingletonExample7

枚举方式进行实例化，是线程安全的，此种方式也是线程最安全的

```
package io.binghe.concurrency.example.singleton;
/**
 * @author binghe
 * @version 1.0.0
 * @description 枚举方式进行实例化，是线程安全的，此种方式也是线程最安全的
 */
public class SingletonExample7 {

    private SingletonExample7(){}

    public static SingletonExample7 getInstance(){
        return Singleton.INSTANCE.getInstance();
    }

    private enum Singleton{
        INSTANCE;
        private SingletonExample7 singleton;

        //JVM保证这个方法绝对只调用一次
        Singleton(){
            singleton = new SingletonExample7();
        }
        public SingletonExample7 getInstance(){
            return singleton;
        }
    }
}
```

抽象工厂模式

一、概述：

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

二、为何使用

工厂模式是我们最常用的模式了,著名的Jive论坛 ,就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。

为什么工厂模式是如此常用？因为工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑实用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量

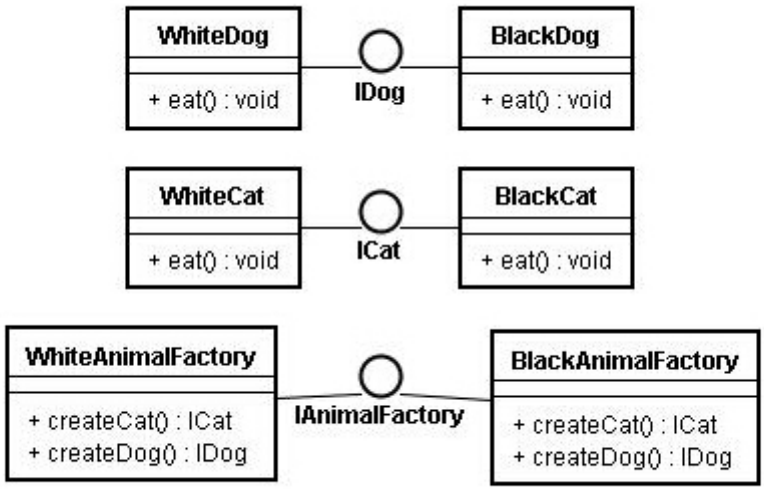
三、实用性

- 一个系统要独立于它的产品的创建、组合和表示时。
- 一个系统要由多个产品系列中的一个来配置时。
- 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
- 当你提供一个产品类库，而只想显示它们的接口而不是实现时。

四、参与者

AbstractFactory 声明一个创建抽象产品对象的操作接口。
ConcreteFactory 实现创建具体产品对象的操作。
AbstractProduct 为一类产品对象声明一个接口。
ConcreteProduct 定义一个将被相应的具体工厂创建的产品对象。 实现AbstractProduct接口。
Client 仅使用由AbstractFactory和AbstractProduct类声明的接口

五、类图



六、示例

AbstractFactory:定义抽象工程类IAnimalFactory

```
package com.binghe.design.abstractfactory;
/**
 * 这个接口就是类图中标识的
 * AbstractFactory抽象工厂
 * @author binghe
 *
 */
public interface IAnimalFactory {
    /**
     * 定义创建Icat接口实例的方法
     * @return
     */
    ICat createCat();
    /**
     * 定义创建IDog接口实例的方法
     * @return
     */
    IDog createDog();
}
```

ConcreteFactory：创建抽象工厂类的两个实现类，WhiteAnimalFactory和BlackAnimalFactory

```
package com.binghe.design.abstractfactory;
/**
 * IAnimalFactory抽象工厂的实现类
 * @author binghe
 *
 */
public class WhiteAnimalFactory implements IAnimalFactory {

    public ICat createCat() {
        return new WhiteCat();
    }

    public IDog createDog() {
        return new WhiteDog();
    }

}
```

```
package com.binghe.design.abstractfactory;
/**
 * IAnimalFactory抽象工厂的实现类
 * @author binghe
 */
public class BlackAnimalFactory implements IAnimalFactory {
    @Override
    public ICat createCat() {
        return new BlackCat();
    }

    public IDog createDog() {
        return new BlackDog();
    }

}
```

AbstractProduct：定义抽象工厂中要生产的抽象产品接口ICat和IDog

```
package com.binghe.design.abstractfactory;
/**
 * 类图中定义的AbstractProduct
 * 指定工厂生产的产品
 * @author binghe
 *
 */
public interface ICat {
    /**
     * 定义方法
     */
    void eat();
}
```

```
package com.binghe.design.abstractfactory;
/**
```

```
 * 类图中定义的AbstractProduct
 * 指定工厂生产的产品
 * @author binghe
 *
 */
public interface IDog {

     /**
      * 定义方法
      */
     void eat();
}
```

ConcreteProduct：创建产品的实现类BlackCat、BlackDog、WhiteCat、WhiteDog

```
package com.binghe.design.abstractfactory;
/**
 * ICat接口的实现类
 * @author binghe
 *
 */
public class BlackCat implements ICat {
    @Override
    public void eat() {
        System.out.println("The black cat is eating!");
    }
}
```

```
package com.binghe.design.abstractfactory;

/**
 * IDog的实现类
 * @author binghe
 */
public class BlackDog implements IDog {
    @Override
    public void eat() {
        System.out.println("The black dog is eating");
    }
}
```

```
package com.binghe.design.abstractfactory;
/**
 * ICat的实现类
 * @author binghe
 *
 */
public class WhiteCat implements ICat {
    @Override
    public void eat() {
        System.out.println("The white cat is eating!");
    }
}
```

```
package com.binghe.design.abstractfactory;
/**
 * IDog的实现类
 * @author binghe
 *
 */
public class WhiteDog implements IDog {
    @Override
    public void eat() {
        System.out.println("The white dog is eating!");
    }

}
```

Client：定义一个测试类Test

```
package com.binghe.design.abstractfactory;

/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        IAnimalFactory blackAnimalFactory = new BlackAnimalFactory();
        ICat blackCat = blackAnimalFactory.createCat();
        blackCat.eat();
        IDog blackDog = blackAnimalFactory.createDog();
        blackDog.eat();

        IAnimalFactory whiteAnimalFactory = new WhiteAnimalFactory();
        ICat whiteCat = whiteAnimalFactory.createCat();
        whiteCat.eat();
        IDog whiteDog = whiteAnimalFactory.createDog();
        whiteDog.eat();
    }
}
```

输出结果

```
The black cat is eating!
The black dog is eating
The white cat is eating!
The white dog is eating!
```

七、总结

由此可见，工厂方法确实为系统结构提供了非常灵活强大的动态扩展机制，只要我们更换一下具体的工厂方法，系统其他地方无需一点变换，就有可能将系统功能进行改头换面的变化

工厂方法

一、概述

定义一个用于创建对象的接口，让子类决定实例化哪一个类。FactoryMethod使一个类的实例化延迟到其子类。

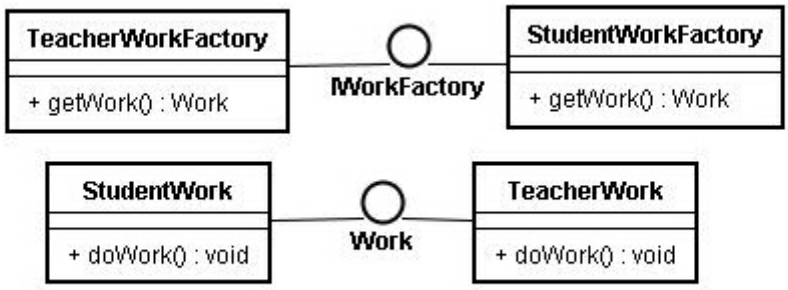
二、适用性

- 当一个类不知道它所必须创建的对象的类的时候。
- 当一个类希望由它的子类来指定它所创建的对象的时候。
- 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。

三、参与者

- Product 定义工厂方法所创建的对象的接口。
- ConcreteProduct 实现Product接口。
- Creator 声明工厂方法，该方法返回一个Product类型的对象。 Creator也可以定义一个工厂方法的缺省实现，它返回一个缺省的ConcreteProduct对象。 可以调用工厂方法以创建一个Product对象。
- ConcreteCreator 重定义工厂方法以返回一个ConcreteProduct实例。

四、类图



五、示例

Product

```
package com.binghe.design.factorymethod;

/**
 * 定义一个接口work
 * @author binghe
 *
 */
public interface work {
    /**
     * 定义方法
     */
    void dowork();
}
```

ConcreteProduct

```java
package com.binghe.design.factorymethod;
/**
 *  Work接口的具体实现类
 * @author binghe
 *
 */
public class TeacherWork implements Work {
    public void doWork() {
        System.out.println("老师审批作业!");
    }
}
```

```java
package com.binghe.design.factorymethod;
/**
 * Work接口的具体实现类
 * @author binghe
 */
public class StudentWork implements Work {
    @Override
    public void doWork() {
        System.out.println("学生做作业!");
    }
}
```

Creator

```java
package com.binghe.design.factorymethod;
/**
 * 抽象工厂接口
 * @author binghe
 *
 */
public interface IWorkFactory {
    /**
     * 定义获取Work实例对象的方法
     * @return
     */
    Work getWork();
}
```

ConcreteCreator

```java
package com.binghe.design.factorymethod;
/**
 * IWorkFactory工厂的实现类
 * @author binghe
 *
 */
public class StudentWorkFactory implements IWorkFactory {
    @Override
    public Work getWork() {
        return new StudentWork();
    }
}
```

```
package com.binghe.design.factorymethod;
/**
 * IWorkFactory工厂的实现类
 * @author binghe
 *
 */
public class TeacherWorkFactory implements IWorkFactory {
    @Override
    public Work getWork() {
        return new TeacherWork();
    }
}
```

Test

```
package com.binghe.design.factorymethod;
/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        IWorkFactory studentWorkFactory = new StudentWorkFactory();
        studentWorkFactory.getWork().doWork();

        IWorkFactory teacherWorkFactory = new TeacherWorkFactory();
        teacherWorkFactory.getWork().doWork();
    }
}
```

Resut

```
学生做作业!
老师审批作业!
```

建造者模式

一、概述

将一个复杂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。

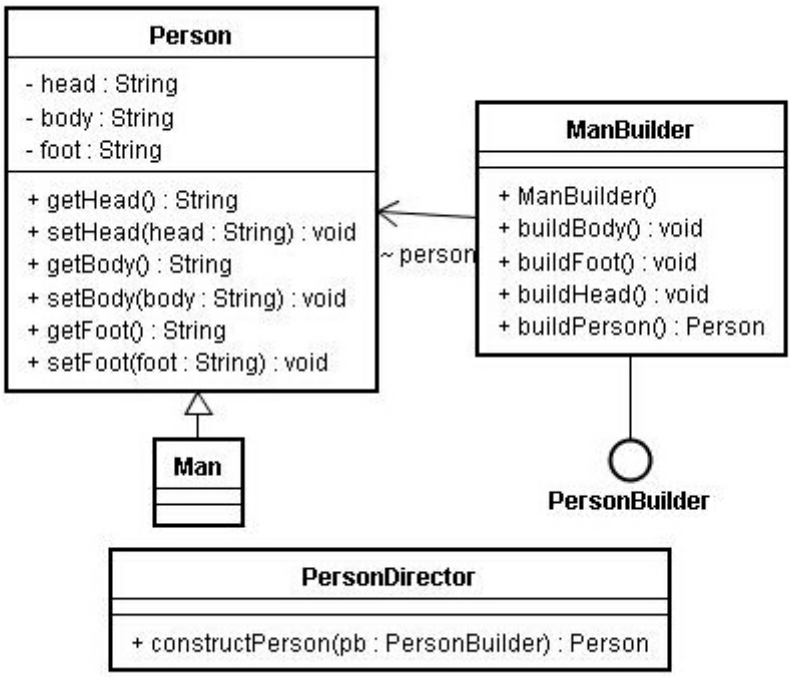
二、适用性

- 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。
- 当构造过程必须允许被构造的对象有不同的表示时。

三、参与者

- Builder 为创建一个Product对象的各个部件指定抽象接口。
- ConcreteBuilder 实现Builder的接口以构造和装配该产品的各个部件。 定义并明确它所创建的表示。 提供一个检索产品的接口。
- Director 构造一个使用Builder接口的对象。
- Product 表示被构造的复杂对象。ConcreteBuilder创建该产品的内部表示并定义它的装配过程。包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

四、类图



五、示例

Builder

```
package com.binghe.design.builder;
/**
 * Person对象的构造接口
 * @author binghe
 *
 */
public interface PersonBuilder {

    void buildHead();

    void buildBody();

    void buildFoot();

    Person buildPerson();
}
```

ConcreteBuilder

```
package com.binghe.design.builder;
/**
 * Person对象的构造器
 * @author binghe
 *
 */
public class ManBuilder implements PersonBuilder {
    Person person;
    public ManBuilder() {
        person = new Man();
```

```
    }
    public void buildBody() {
        person.setBody("建造男人的身体");
    }

    public void buildFoot() {
        person.setFoot("建造男人的脚");
    }

    public void buildHead() {
        person.setHead("建造男人的头");
    }

    public Person buildPerson() {
        return person;
    }
}
```

Director

```
package com.binghe.design.builder;
/**
 * Person对象的整体构造器
 * @author binghe
 *
 */
public class PersonDirector {

    public Person constructPerson(PersonBuilder pb) {
        pb.buildHead();
        pb.buildBody();
        pb.buildFoot();
        return pb.buildPerson();
    }
}
```

Product

```
package com.binghe.design.builder;

/**
 * Person对象
 * @author binghe
 *
 */
public class Person {
    private String head;
    private String body;
    private String foot;

    public String getHead() {
        return head;
    }

    public void setHead(String head) {
        this.head = head;
```

```
    }

    public String getBody() {
        return body;
    }

    public void setBody(String body) {
        this.body = body;
    }

    public String getFoot() {
        return foot;
    }

    public void setFoot(String foot) {
        this.foot = foot;
    }
}
```

```
package com.binghe.design.builder;
/**
 * 创建一个男人类继承Person
 * @author binghe
 *
 */
public class Man extends Person {

}
```

Test

```
package com.binghe.design.builder;

/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        PersonDirector pd = new PersonDirector();
        Person person = pd.constructPerson(new ManBuilder());
        System.out.println(person.getBody());
        System.out.println(person.getFoot());
        System.out.println(person.getHead());
    }
}
```

Result

```
建造男人的身体
建造男人的脚
建造男人的头
```

原型模式

一、概述

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

二、适用性

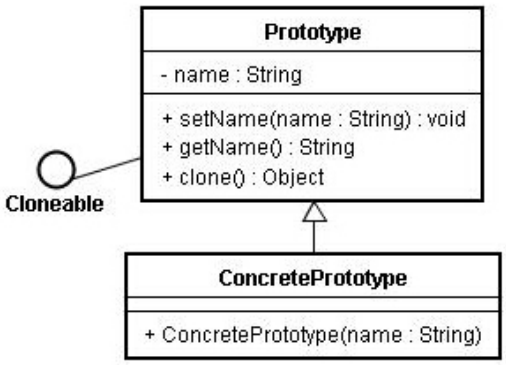
- 当一个系统应该独立于它的产品创建、构成和表示时。
- 当要实例化的类是在运行时刻指定时，例如，通过动态装载。
- 为了避免创建一个与产品类层次平行的工厂类层次时。
- 当一个类的实例只能有几个不同状态组合中的一种时。

建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

三、参与者

- Prototype 声明一个克隆自身的接口。
- ConcretePrototype 实现一个克隆自身的操作。
- Client 让一个原型克隆自身从而创建一个新的对象。

四、类图



五、示例

Prototype

```
package com.binghe.design.prototype;
/**
 * 原型类，实现Cloneable接口
 * @author binghe
 *
 */
public class Prototype implements Cloneable {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public Object clone(){
        try {
            return super.clone();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
```

```
        }
    }
}
```

ConcretePrototype

```
package com.binghe.design.prototype;
/**
 * 原型类的子类，用于构建原型
 * @author binghe
 *
 */
public class ConcretePrototype extends Prototype {
    public ConcretePrototype(String name) {
        setName(name);
    }
}
```

Client

```
package com.binghe.design.prototype;
/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Prototype pro = new ConcretePrototype("prototype");
        Prototype pro2 = (Prototype)pro.clone();
        System.out.println(pro.getName());
        System.out.println(pro2.getName());
    }
}
```

Result

```
prototype
prototype
```

结构型模式

适配器模式

一、概述

将一个类的接口转换成客户希望的另外一个接口。Adapter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。

二、适用性

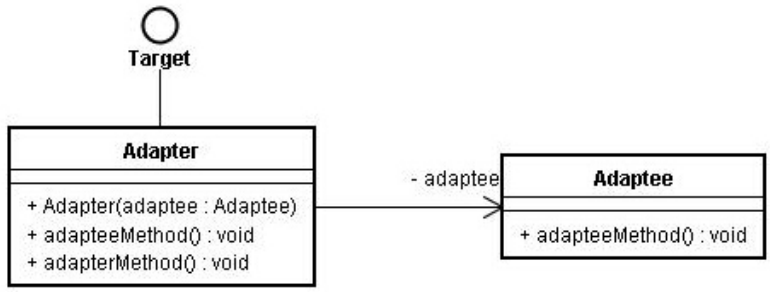
- 使用一个已经存在的类，而它的接口不符合你的需求。
- 创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口 可能不一定兼容的类）协同工作。

- （仅适用于对象Adapter）使用一些已经存在的子类，但是不可能对每一个都进行 子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

三、参与者

- Target 定义Client使用的与特定领域相关的接口。
- Client 与符合Target接口的对象协同。
- Adaptee 定义一个已经存在的接口，这个接口需要适配。
- Adapter 对Adaptee的接口与Target接口进行适配

四、类图



五、示例

Target

```
package com.binghe.design.adapter;
/**
 * 定义Target接口
 * @author binghe
 *
 */
public interface Target {
    void adapteeMethod();
    void adapterMethod();
}
```

Adaptee

```
package com.binghe.design.adapter;
/**
 * 适配器类
 * @author binghe
 *
 */
public class Adaptee {
    public void adapteeMethod() {
        System.out.println("Adaptee method!");
    }
}
```

Adapter

```
package com.binghe.design.adapter;
/**
```

```
 * Target的实现类
 * @author binghe
 *
 */
public class Adapter implements Target {
    private Adaptee adaptee;
    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    public void adapteeMethod() {
        adaptee.adapteeMethod();
    }

    public void adapterMethod() {
        System.out.println("Adapter method!");
    }
}
```

Client

```
package com.binghe.design.adapter;

/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.adapteeMethod();
        target.adapterMethod();
    }
}
```

Result

```
Adaptee method!
Adapter method!
```

装饰模式

一、概述

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

二、适用性

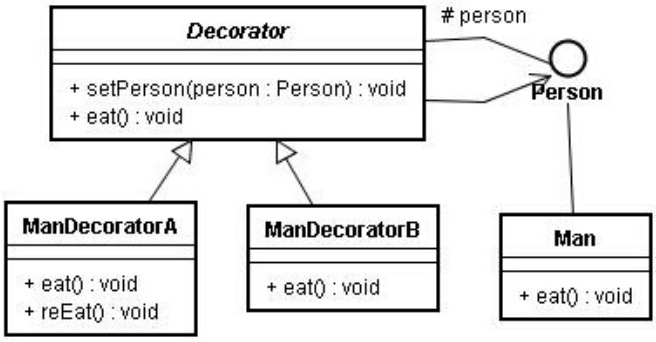
- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。

三、参与者

- Component 定义一个对象接口，可以给这些对象动态地添加职责。

- ConcreteComponent 定义一个对象，可以给这个对象添加一些职责。
- Decorator 维持一个指向Component对象的指针，并定义一个与Component接口一致的接口。
- ConcreteDecorator 向组件添加职责。

四、类图



五、示例

Component

```
package com.binghe.design.decorator;
/**
 * 定义Component接口Person
 * @author binghe
 *
 */
public interface Person {
    void eat();
}
```

ConcreteComponent

```
package com.binghe.design.decorator;
/**
 * Person接口的实现类Man
 * @author binghe
 *
 */
public class Man implements Person {
    public void eat() {
        System.out.println("男人在吃");
    }
}
```

Decorator

```
package com.binghe.design.decorator;
/**
 * Decorator抽象类实现Person接口
 * @author binghe
 *
 */
public abstract class Decorator implements Person {
    protected Person person;
```

```
    public void setPerson(Person person) {
        this.person = person;
    }

    public void eat() {
        person.eat();
    }
}
```

ConcreteDecorator

```
package com.binghe.design.decorator;
/**
 * Decorator的子类
 * @author binghe
 *
 */
public class ManDecoratorA extends Decorator {
    public void eat() {
        super.eat();
        reEat();
        System.out.println("ManDecoratorA类");
    }

    public void reEat() {
        System.out.println("再吃一顿饭");
    }
}
```

```
package com.binghe.design.decorator;
/**
 * Decorator的子类
 * @author binghe
 *
 */
public class ManDecoratorB extends Decorator {

    public void eat() {
        super.eat();
        System.out.println("===============");
        System.out.println("ManDecoratorB类");
    }
}
```

Test

```
package com.binghe.design.decorator;
/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Man man = new Man();
```

```
        ManDecoratorA md1 = new ManDecoratorA();
        ManDecoratorB md2 = new ManDecoratorB();

        md1.setPerson(man);
        md2.setPerson(md1);
        md2.eat();
    }
}
```

Result

```
男人在吃
再吃一顿饭
ManDecoratorA类
===============
ManDecoratorB类
```

外观模式

一、概述

为子系统中的一组接口提供一个一致的界面，Facade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。

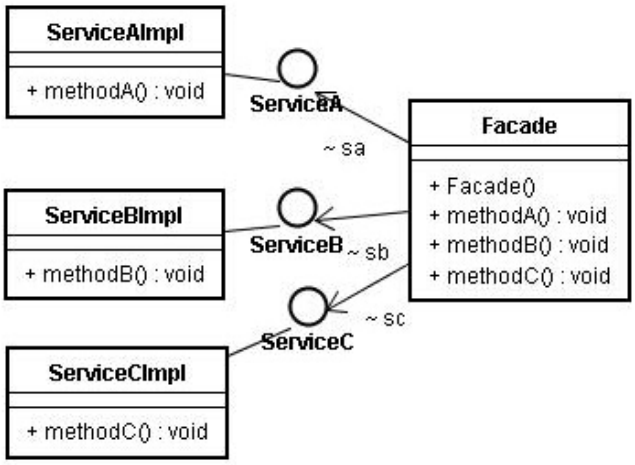
二、适用性

- 当要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越 复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容 易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。 Facade可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需 要更多的可定制性的用户可以越过facade层。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入facade将这个子系统与客 户以及其他的子系统分离，可以提高子系统的独立性和可移植性。
- 当需要构建一个层次结构的子系统时，使用facade模式定义子系统中每层的入口点。 如果子系统之间是相互依赖的，你可以让它们仅通过facade进行通讯，从而简化了它们 之间的依赖关系。

三、参与者

- Facade 知道哪些子系统类负责处理请求。 将客户的请求代理给适当的子系统对象。
- Subsystemclasses 实现子系统的功能。 处理由Facade对象指派的任务。 没有facade的任何相关信息；即没有指向facade的指针。

四、类图



五、示例

Facade

```
package com.binghe.design.facade;
/**
 * Facade
 * @author binghe
 *
 */
public class Facade {
    ServiceA sa;
    ServiceB sb;
    ServiceC sc;
    public Facade() {
        sa = new ServiceAImpl();
        sb = new ServiceBImpl();
        sc = new ServiceCImpl();
    }

    public void methodA() {
        sa.methodA();
        sb.methodB();
    }

    public void methodB() {
        sb.methodB();
        sc.methodC();
    }

    public void methodC() {
        sc.methodC();
        sa.methodA();
    }
}
```

Subsystemclasses

```java
package com.binghe.design.facade;
/**
 * Subsystemclasses
 * @author binghe
 *
 */
public class ServiceAImpl implements ServiceA {
    public void methodA() {
        System.out.println("这是服务A");
    }
}
```

```java
package com.binghe.design.facade;
/**
 * Subsystemclasses
 * @author binghe
 *
 */
public class ServiceBImpl implements ServiceB {
    public void methodB() {
        System.out.println("这是服务B");
    }
}
```

```java
package com.binghe.design.facade;
/**
 * Subsystemclasses
 * @author binghe
 *
 */
public class ServiceCImpl implements ServiceC {
    public void methodC() {
        System.out.println("这是服务C");
    }
}
```

Test

```java
package com.binghe.design.facade;
/**
 * Test
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        ServiceA sa = new ServiceAImpl();
        ServiceB sb = new ServiceBImpl();
        sa.methodA();
        sb.methodB();
        System.out.println("========");
        //facade
        Facade facade = new Facade();
        facade.methodA();
        facade.methodB();
```

```
    }
}
```

Result

```
这是服务A
这是服务B
========
这是服务A
这是服务B
这是服务B
这是服务C
```

代理模式

一、概述

为其他对象提供一种代理以控制对这个对象的访问。

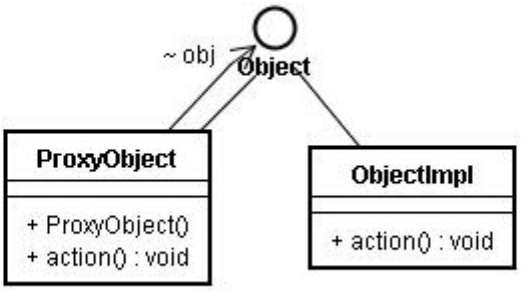
二、适用性

- 远程代理（RemoteProxy）为一个对象在不同的地址空间提供局部代表。
- 虚代理（VirtualProxy）根据需要创建开销很大的对象。
- 保护代理（ProtectionProxy）控制对原始对象的访问。 4.智能指引（SmartReference）取代了简单的指针，它在访问对象时执行一些附加操作。

三、参与者

- Proxy 保存一个引用使得代理可以访问实体。若RealSubject和Subject的接口相同，Proxy会引用Subject。 提供一个与Subject的接口相同的接口，这样代理就可以用来替代实体。 控制对实体的存取，并可能负责创建和删除它。 其他功能依赖于代理的类型：
- RemoteProxy负责对请求及其参数进行编码，并向不同地址空间中的实体发送已编码的请求。
- VirtualProxy可以缓存实体的附加信息，以便延迟对它的访问。
- ProtectionProxy检查调用者是否具有实现一个请求所必需的访问权限。
- Subject 定义RealSubject和Proxy的共用接口，这样就在任何使用RealSubject的地方都可以使用Proxy。
- RealSubject 定义Proxy所代表的实体

四、类图



五、示例

Proxy

```java
package com.binghe.design.proxy;
/**
 * Proxy
 * @author binghe
 *
 */
public class ProxyObject implements Object {
    Object obj;

    public ProxyObject() {
        System.out.println("这是代理类");
        obj = new ObjectImpl();
    }

    public void action() {
        System.out.println("代理开始");
        obj.action();
        System.out.println("代理结束");
    }
}
```

Subject

```java
package com.binghe.design.proxy;
/**
 * Subject
 * @author binghe
 *
 */
public interface Object {
    void action();
}
```

RealSubject

```java
package com.binghe.design.proxy;
/**
 * RealSubject
 * @author binghe
 *
 */
public class ObjectImpl implements Object {
    public void action() {
        System.out.println("========");
        System.out.println("========");
        System.out.println("这是被代理的类");
        System.out.println("========");
        System.out.println("========");
    }
}
```

Test

```
package com.binghe.design.proxy;
/**
 * Test
 * @author binghe
 *
 */
public class Test {
    public static void main() {
        Object obj = new ProxyObject();
        obj.action();
    }
}
```

Result

```
这是代理类
代理开始
========
========
这是被代理的类
========
========
代理结束
```

桥接模式

一、概述

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

二、适用性

- 不希望在抽象和它的实现部分之间有一个固定的绑定关系。 例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。
- 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。 这时Bridge模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。
- 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
- 正如在意图一节的第一个类图中所示的那样，有许多类要生成。 这样一种类层次结构说明你必须将一个对象分解成两个部分。
- 在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

三、参与者

1.Abstraction

定义抽象类的接口。 维护一个指向Implementor类型对象的指针。

2.RefinedAbstraction

扩充由Abstraction定义的接口。

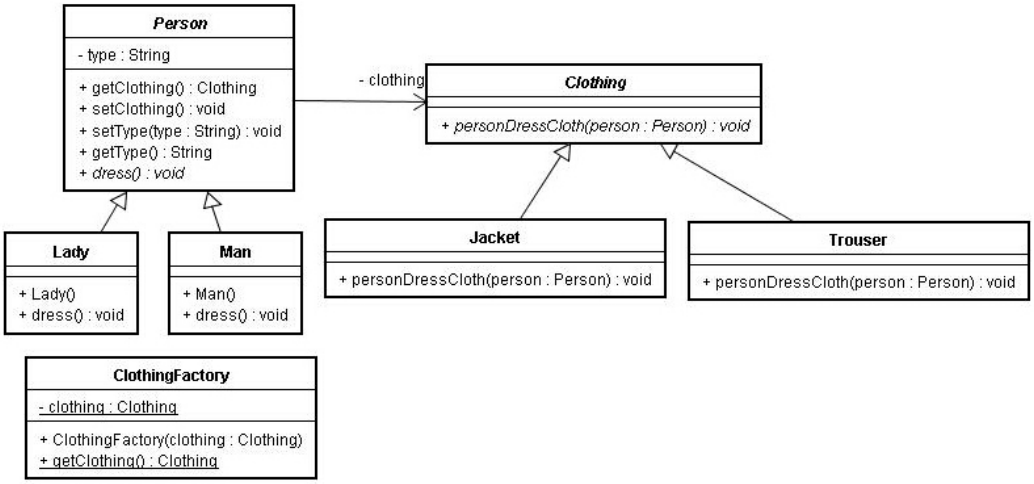
3.Implementor

定义实现类的接口，该接口不一定要与Abstraction的接口完全一致。 事实上这两个接口可以完全不同。 一般来讲，Implementor接口仅提供基本操作，而Abstraction则定义了基于这些基本操作的较高层次的操作。

4.ConcreteImplementor

实现Implementor接口并定义它的具体实现。

四、类图



五、示例

Abstraction

```
package com.binghe.design.bridge;

/**
 * 定义Abstraction Person类
 * @author binghe
 *
 */
public abstract class Person {

    private Clothing clothing;
    private String type;

    public Clothing getClothing() {
        return clothing;
    }

    public void setClothing(Clothing clothing) {
        this.clothing = clothing;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getType() {
        return this.type;
    }

    public abstract void dress();
}
```

RefinedAbstraction

```java
package com.binghe.design.bridge;
/**
 * 定义RefinedAbstraction类Man
 * @author binghe
 *
 */
public class Man extends Person {
    public Man() {
        setType("男人");
    }

    public void dress() {
        Clothing clothing = getClothing();
        clothing.personDressCloth(this);
    }
}
```

```java
package com.binghe.design.bridge;
/**
 * 定义RefinedAbstraction类Lady
 * @author binghe
 *
 */
public class Lady extends Person {
    public Lady() {
        setType("女人");
    }

    public void dress() {
        Clothing clothing = getClothing();
        clothing.personDressCloth(this);
    }
}
```

Implementor

```java
package com.binghe.design.bridge;
/**
 * 定义Implementor 类Clothing
 * @author binghe
 *
 */
public abstract class Clothing {
    public abstract void personDressCloth(Person person);
}
```

ConcreteImplementor

```
package com.binghe.design.bridge;
/**
 * 定义ConcreteImplementor类Jacket
 * @author binghe
 *
 */
public class Jacket extends Clothing {
    public void personDressCloth(Person person) {
        System.out.println(person.getType() + "穿马甲");
    }
}
```

```
package com.binghe.design.bridge;
/**
 * 定义ConcreteImplementor类 Trouser
 * @author binghe
 *
 */
public class Trouser extends Clothing {
    public void personDressCloth(Person person) {
        System.out.println(person.getType() + "穿裤子");
    }
}
```

Test

```
package com.binghe.design.bridge;

/**
 * 测试类
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {

        Person man = new Man();

        Person lady = new Lady();

        Clothing jacket = new Jacket();

        Clothing trouser = new Trouser();

        jacket.personDressCloth(man);
        trouser.personDressCloth(man);

        jacket.personDressCloth(lady);
        trouser.personDressCloth(lady);
    }
}
```

Result

```
男人穿马甲
男人穿裤子
女人穿马甲
女人穿裤子
```

组合模式

一、概述

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

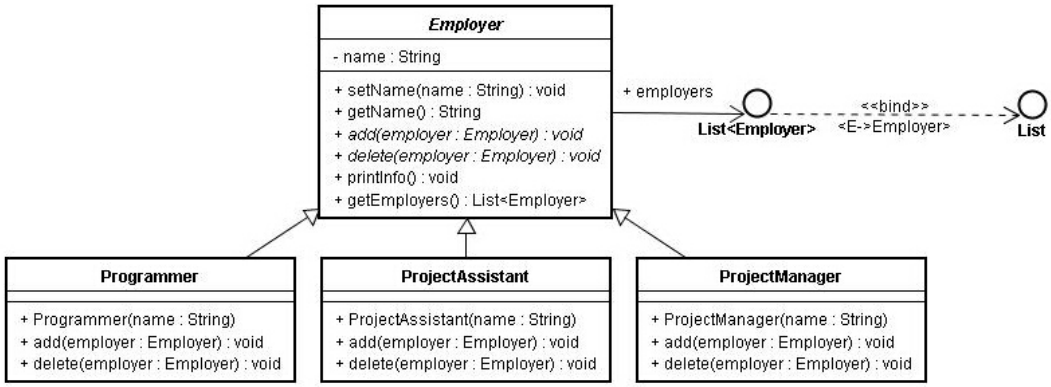
二、适用性

- 表示对象的部分-整体层次结构。
- 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。

三、参与者

- Component 为组合中的对象声明接口。 在适当的情况下，实现所有类共有接口的缺省行为。 声明一个接口用于访问和管理Component的子组件。 (可选)在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它。
- Leaf 在组合中表示叶节点对象，叶节点没有子节点。 在组合中定义节点对象的行为。
- Composite 定义有子部件的那些部件的行为。 存储子部件。 在Component接口中实现与子部件有关的操作。
- Client 通过Component接口操纵组合部件的对象。

四、类图



五、示例

Component

```
package com.binghe.design.composite;

import java.util.List;

/**
 * 定义Component 类Employer
 * @author binghe
 *
 */
public abstract class Employer {
    private String name;
```

```
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public abstract void add(Employer employer);

    public abstract void delete(Employer employer);

    public List<Employer> employers;

    public void printInfo() {
        System.out.println(name);
    }

    public List<Employer> getEmployers() {
        return this.employers;
    }
}
```

Leaf

```
package com.binghe.design.composite;
/**
 * 定义Leaf类Programmer
 * @author binghe
 *
 */
public class Programmer extends Employer {
    public Programmer(String name) {
        setName(name);
        employers = null;//程序员，表示没有下属了
    }
    public void add(Employer employer) {
    }

    public void delete(Employer employer) {
    }
}
```

```
package com.binghe.design.composite;
/**
 * 定义Leaf类ProjectAssistant
 * @author binghe
 *
 */
public class ProjectAssistant extends Employer {
    public ProjectAssistant(String name) {
        setName(name);
        employers = null;//项目助理，表示没有下属了
    }
    public void add(Employer employer) {

    }
```

```
    public void delete(Employer employer) {

    }
}
```

Composite

```
package com.binghe.design.composite;

import java.util.ArrayList;

/**
 * 定义Composite类ProjectManager类
 * @author binghe
 *
 */
public class ProjectManager extends Employer {
    public ProjectManager(String name) {
        setName(name);
        employers = new ArrayList<Employer>();
    }

    public void add(Employer employer) {
        employers.add(employer);
    }

    public void delete(Employer employer) {
        employers.remove(employer);
    }
}
```

Client

```
package com.binghe.design.composite;

import java.util.List;

/**
 * 测试类
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Employer pm = new ProjectManager("项目经理");
        Employer pa = new ProjectAssistant("项目助理");
        Employer programmer1 = new Programmer("程序员一");
        Employer programmer2 = new Programmer("程序员二");

        pm.add(pa);//为项目经理添加项目助理
        pm.add(programmer2);//为项目经理添加程序员

        List<Employer> ems = pm.getEmployers();
        for (Employer em : ems) {
            System.out.println(em.getName());
        }
```

```
    }
}
```

Result

```
项目助理
程序员二
```

享元模式

一、概述

运用共享技术有效地支持大量细粒度的对象。

二、适用性

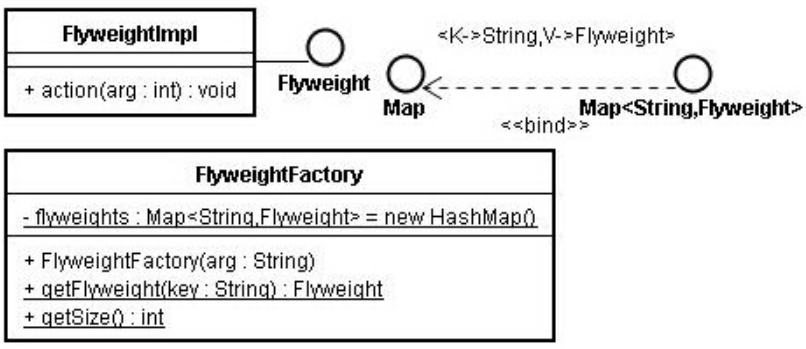
当都具备下列情况时，使用Flyweight模式：

- 一个应用程序使用了大量的对象。
- 完全由于使用大量的对象，造成很大的存储开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖于对象标识。由于Flyweight对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

三、参与者

- Flyweight 描述一个接口，通过这个接口flyweight可以接受并作用于外部状态。
- ConcreteFlyweight 实现Flyweight接口，并为内部状态（如果有的话）增加存储空间。ConcreteFlyweight对象必须是可共享的。它所存储的状态必须是内部的；即，它必须独立于ConcreteFlyweight对象的场景。
- UnsharedConcreteFlyweight 并非所有的Flyweight子类都需要被共享。Flyweight接口使共享成为可能，但它并不强制共享。 在Flyweight对象结构的某些层次，UnsharedConcreteFlyweight对象通常将ConcreteFlyweight对象作为子节点。
- FlyweightFactory 创建并管理flyweight对象。 确保合理地共享flyweight。当用户请求一个flyweight时，FlyweightFactory对象提供一个已创建的实例或者创建一个（如果不存在的话）。

四、类图



五、示例

Flyweight

```java
package com.binghe.design.flyweight;
/**
 * Flyweight
 * @author binghe
 *
 */
public interface Flyweight {
    void action(int arg);
}
```

ConcreteFlyweight

```java
package com.binghe.design.flyweight;
/**
 * ConcreteFlyweight
 * @author binghe
 *
 */
public class FlyweightImpl implements Flyweight {

    public void action(int arg) {
        // TODO Auto-generated method stub
        System.out.println("参数值: " + arg);
    }
}
```

FlyweightFactory

```java
package com.binghe.design.flyweight;

import java.util.HashMap;
import java.util.Map;

/**
 * FlyweightFactory
 * @author binghe
 *
 */
public class FlyweightFactory {

    private static Map flyweights = new HashMap();

    public FlyweightFactory(String arg) {
        flyweights.put(arg, new FlyweightImpl());
    }

    public static Flyweight getFlyweight(String key) {
        if (flyweights.get(key) == null) {
            flyweights.put(key, new FlyweightImpl());
        }
        return (Flyweight) flyweights.get(key);
    }

    public static int getSize() {
        return flyweights.size();
    }
```

```
}
```

Test

```
package com.binghe.design.flyweight;
/**
 * Test
 * @author binghe
 */
public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Flyweight fly1 = FlyweightFactory.getFlyweight("a");
        fly1.action(1);

        Flyweight fly2 = FlyweightFactory.getFlyweight("a");
        System.out.println(fly1 == fly2);

        Flyweight fly3 = FlyweightFactory.getFlyweight("b");
        fly3.action(2);

        Flyweight fly4 = FlyweightFactory.getFlyweight("c");
        fly4.action(3);

        Flyweight fly5 = FlyweightFactory.getFlyweight("d");
        fly4.action(4);

        System.out.println(FlyweightFactory.getSize());
    }
}
```

行为模式

责任链模式

一、概述

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链， 并沿着这条链传递该请求，直到有一个对象处理它为止。 这一模式的想法是，给多个对象处理一个请求的机会，从而解耦发送者和接受者。

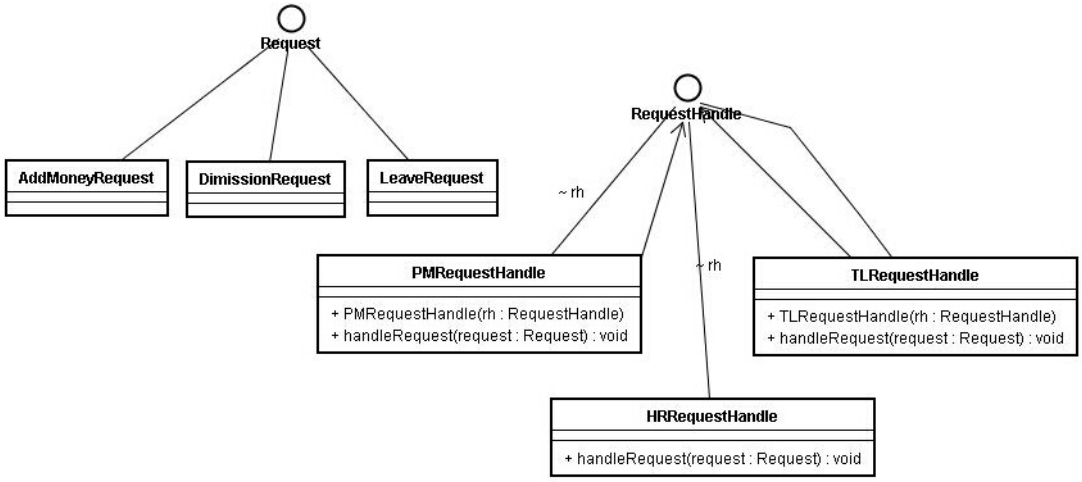
二、适用性

- 有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 可处理一个请求的对象集合应被动态指定。

三、参与者

- Handler 定义一个处理请求的接口。 （可选）实现后继链。
- ConcreteHandler 处理它所负责的请求。 可访问它的后继者。 如果可处理该请求，就处理之；否则将该请求转发给它的后继者。
- Client 向链上的具体处理者(ConcreteHandler)对象提交请求。

四、类图



五、示例

Handler

```
package com.binghe.design.cor;

import org.omg.CORBA.Request;

/**
 * Handler
 * @author binghe
 *
 */
public interface RequestHandle {
    void handleRequest(Request request);
}
```

ConcreteHandler

```
package com.binghe.design.cor;

import org.omg.CORBA.Request;

/**
 * ConcreteHandler
 * @author binghe
 *
 */
public class HRRequestHandle implements RequestHandle {

    public void handleRequest(Request request) {
        if (request instanceof DimissionRequest) {
            System.out.println("要离职, 人事审批!");
        }

        System.out.println("请求完毕");
    }
}
```

```
package com.binghe.design.cor;
/**
```

```
 * ConcreteHandler
 * @author binghe
 *
 */
public class PMRequestHandle implements RequestHandle {

    RequestHandle rh;

    public PMRequestHandle(RequestHandle rh) {
        this.rh = rh;
    }

    public void handleRequest(Request request) {
        if (request instanceof AddMoneyRequest) {
            System.out.println("要加薪, 项目经理审批!");
        } else {
            rh.handleRequest(request);
        }
    }
}
```

```
package com.binghe.design.cor;
/**
 * ConcreteHandler
 * @author binghe
 *
 */
public class TLRequestHandle implements RequestHandle {
    RequestHandle rh;
    public TLRequestHandle(RequestHandle rh) {
        this.rh = rh;
    }

    public void handleRequest(Request request) {
        if (request instanceof LeaveRequest) {
            System.out.println("要请假, 项目组长审批!");
        } else {
            rh.handleRequest(request);
        }
    }
}
```

Test

```
package com.binghe.design.cor;

import org.omg.CORBA.Request;

/**
 * Test
 * @author binghe
 *
 */
public class Test {

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

```
        RequestHandle hr = new HRRequestHandle();
        RequestHandle pm = new PMRequestHandle(hr);
        RequestHandle tl = new TLRequestHandle(pm);

        //team leader处理离职请求
        Request request = new Request();
        tl.handleRequest(request);

        System.out.println("===========");
        //team leader处理加薪请求
        request = new AddMoneyRequest();
        tl.handleRequest(request);

        System.out.println("========");
        //项目经理上理辞职请求
        request = new DimissionRequest();
        pm.handleRequest(request);
    }
}
```

Result

```
要离职，人事审批!
请求完毕
===========
要加薪，项目经理审批!
========
要离职，人事审批!
请求完毕
```

命令模式

一、概述

将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

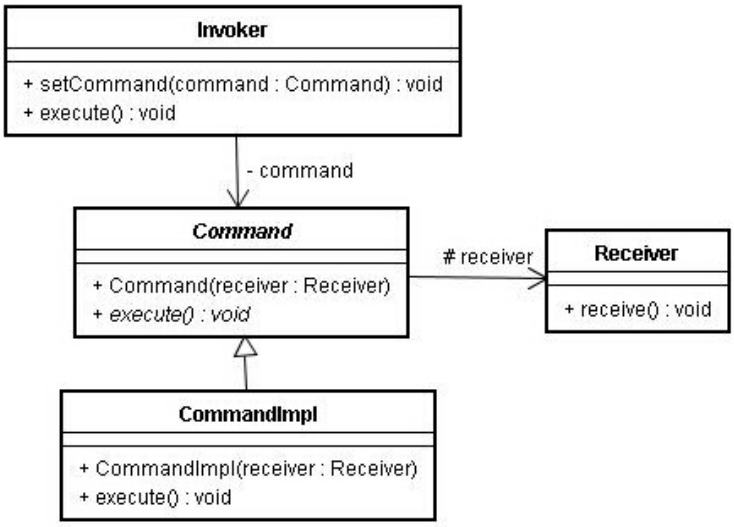
二、适用性

- 抽象出待执行的动作以参数化某对象。
- 在不同的时刻指定、排列和执行请求。
- 支持取消操作。
- 支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。
- 用构建在原语操作上的高层操作构造一个系统。

三、参与者

- Command 声明执行操作的接口。
- ConcreteCommand 将一个接收者对象绑定于一个动作。 调用接收者相应的操作，以实现 Execute。
- Client 创建一个具体命令对象并设定它的接收者。
- Invoker 要求该命令执行这个请求。
- Receiver 知道如何实施与执行一个请求相关的操作。任何类都可能作为一个接收者。

四、类图



五、示例

Command

```
package com.binghe.design.command;
/**
 * Command
 * @author binghe
 *
 */
public abstract class Command {
    protected Receiver receiver;

    public Command(Receiver receiver) {
        this.receiver = receiver;
    }

    public abstract void execute();
}
```

ConcreteCommand

```
package com.binghe.design.command;
/**
 * ConcreteCommand
 * @author binghe
 *
 */
public class CommandImpl extends Command {
    public CommandImpl(Receiver receiver) {
        super(receiver);
    }

    public void execute() {
        receiver.request();
    }
}
```

Invoker

```
package com.binghe.design.command;
/**
 * Invoker
 * @author binghe
 *
 */
public class Invoker {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void execute() {
        command.execute();
    }
}
```

Receiver

```
package com.binghe.design.command;
/**
 * Receiver
 * @author binghe
 *
 */
public class Receiver {
    public void receive() {
        System.out.println("This is Receive class!");
    }
}
```

Test

```
package com.binghe.design.command;
/**
 * Test
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Receiver rec = new Receiver();
        Command cmd = new CommandImpl(rec);
        Invoker i = new Invoker();
        i.setCommand(cmd);
        i.execute();
    }
}
```

Result

```
This is Receive class!
```

解析器模式

一、概述

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

二、适用性

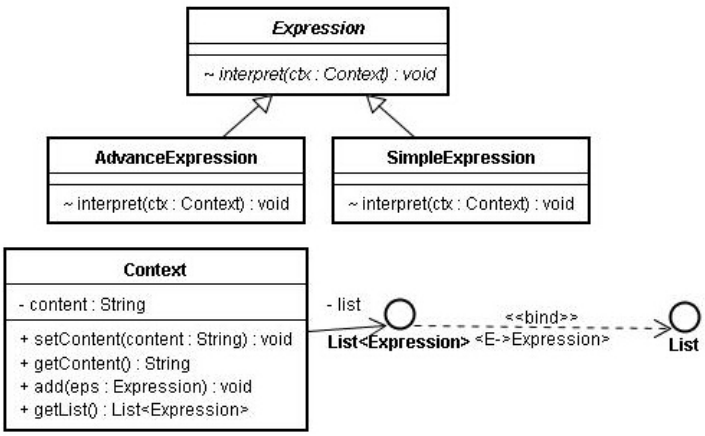
当有一个语言需要解释执行,并且你可将该语言中的句子表示为一个抽象语法树时，可使 用解释器模式。而当存在以下情况时该模式效果最好：

- 该文法简单对于复杂的文法,文法的类层次变得庞大而无法管理。
- 效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的,而是首先将它们转换成另一种形式。

三、参与者

- AbstractExpression(抽象表达式) 声明一个抽象的解释操作，这个接口为抽象语法树中所有的节点所共享。
- TerminalExpression(终结符表达式) 实现与文法中的终结符相关联的解释操作。 一个句子中的每个终结符需要该类的一个实例。
- NonterminalExpression(非终结符表达式) 为文法中的非终结符实现解释(Interpret)操作。
- Context（上下文） 包含解释器之外的一些全局信息。
- Client（客户） 构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。 该抽象语法树由NonterminalExpression和TerminalExpression的实例装配而成。 调用解释操作。

四、类图



五、示例

AbstractExpression

```
package com.binghe.design.interpreter;
/**
 * AbstractExpression
 * @author binghe
 *
 */
public abstract class Expression {
    abstract void interpret(Context ctx);
}
```

Expression

```
package com.binghe.design.interpreter;
/**
 * Expression
 * @author binghe
 *
 */
public class AdvanceExpression extends Expression {
    void interpret(Context ctx) {
        System.out.println("这是高级解析器!");
    }
}
```

```
package com.binghe.design.interpreter;
/**
 * Expression
 * @author binghe
 *
 */
public class SimpleExpression extends Expression {
    void interpret(Context ctx) {
        System.out.println("这是普通解析器!");
    }
}
```

Context

```
package com.binghe.design.interpreter;

import java.util.ArrayList;
import java.util.List;

/**
 * Context
 * @author binghe
 *
 */
public class Context {
    private String content;
    private List<Expression> list = new ArrayList<Expression>();
    public void setContent(String content) {
        this.content = content;
    }
    public String getContent() {
        return this.content;
    }
    public void add(Expression eps) {
        list.add(eps);
    }
    public List<Expression> getList() {
        return list;
    }
}
```

Test

```
package com.binghe.design.interpreter;
/**
 * Text
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Context ctx = new Context();
        ctx.add(new SimpleExpression());
        ctx.add(new AdvanceExpression());
        ctx.add(new SimpleExpression());
        for (Expression eps : ctx.getList()) {
            eps.interpret(ctx);
        }
    }
}
```

Result

```
这是普通解析器!
这是高级解析器!
这是普通解析器!
```

迭代器模式

一、概述

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

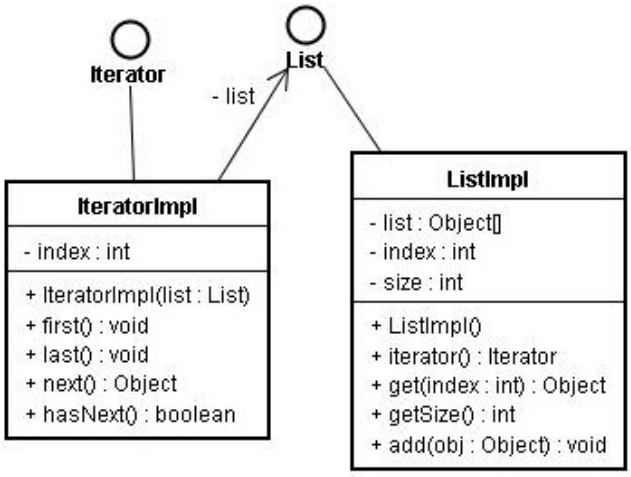
二、适用性

- 访问一个聚合对象的内容而无需暴露它的内部表示。
- 支持对聚合对象的多种遍历。
- 为遍历不同的聚合结构提供一个统一的接口(即,支持多态迭代)。

三、参与者

- Iterator 迭代器定义访问和遍历元素的接口。
- ConcreteIterator 具体迭代器实现迭代器接口。 对该聚合遍历时跟踪当前位置。
- Aggregate 聚合定义创建相应迭代器对象的接口。 4.ConcreteAggregate 具体聚合实现创建相应迭代器的接口，该操作返回ConcreteIterator的一个适当的实例.

四、类图



五、示例

Iterator

```
package com.binghe.design.iterator;
/**
 * Iterator
 * @author binghe
 *
 */
public interface Iterator {

    Object next();

    void first();

    void last();

    boolean hasNext();
}
```

ConcreteIterator

```
package com.binghe.design.iterator;
/**
 * ConcreteIterator
 * @author binghe
 *
 */
public class IteratorImpl implements Iterator {

    private List list;

    private int index;

    public IteratorImpl(List list) {
        index = 0;
        this.list = list;
    }

    public void first() {
```

```
        index = 0;
    }

    public void last() {
        index = list.getSize();
    }

    public Object next() {
        Object obj = list.get(index);
        index++;
        return obj;
    }

    public boolean hasNext() {
        return index < list.getSize();
    }
}
```

Aggregate

```
package com.binghe.design.iterator;
/**
 * Aggregate
 * @author binghe
 *
 */
public interface List {

    Iterator iterator();

    Object get(int index);

    int getSize();

    void add(Object obj);
}
```

ConcreteAggregate

```
package com.binghe.design.iterator;
/**
 * ConcreteAggregate
 * @author binghe
 *
 */
public class ListImpl implements List {

    private Object[] list;

    private int index;

    private int size;

    public ListImpl() {
        index = 0;
        size = 0;
```

```
        list = new Object[100];
    }

    public Iterator iterator() {
        return new IteratorImpl(this);
    }

    public Object get(int index) {
        return list[index];
    }

    public int getSize() {
        return this.size;
    }

    public void add(Object obj) {
        list[index++] = obj;
        size++;
    }
}
```

Test

```
package com.binghe.design.iterator;
/**
 * Test
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {
        List list = new ListImpl();
        list.add("a");
        list.add("b");
        list.add("c");
        //第一种迭代方式
        Iterator it = list.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

        System.out.println("=====");
        //第二种迭代方式
        for (int i = 0; i < list.getSize(); i++) {
            System.out.println(list.get(i));
        }
    }
}
```

Result

```
a
b
c
=====
a
b
c
```

中介者模式

一、概述

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

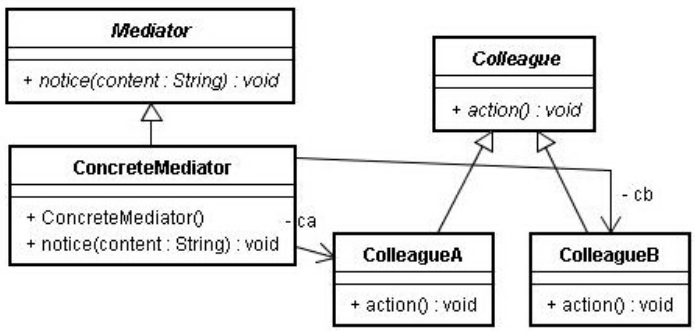
二、适用性

- 一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。
- 一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。
- 定制一个分布在多个类中的行为，而又不想生成太多的子类。

三、参与者

- Mediator 中介者定义一个接口用于与各同事（Colleague）对象通信。
- ConcreteMediator 具体中介者通过协调各同事对象实现协作行为。 了解并维护它的各个同事。
- Colleagueclass 每一个同事类都知道它的中介者对象。 每一个同事对象在需与其他的同事通信的时候，与它的中介者通信

四、类图



五、示例

Mediator

```
package com.binghe.design.mediator;
/**
 * Mediator
 * @author binghe
 *
 */
public abstract class Mediator {

    public abstract void notice(String content);
}
```

ConcreteMediator

```
package com.binghe.design.mediator;
/**
 * ConcreteMediator
 * @author binghe
 *
 */
public class ConcreteMediator extends Mediator {

    private ColleagueA ca;

    private ColleagueB cb;

    public ConcreteMediator() {
        ca = new ColleagueA();
        cb = new ColleagueB();
    }

    public void notice(String content) {
        if (content.equals("boss")) {
            //老板来了，通知员工A
            ca.action();
        }
        if (content.equals("client")) {
            //客户来了，通知前台B
            cb.action();
        }
    }
}
```

Colleagueclass

```
package com.binghe.design.mediator;
/**
 * Colleagueclass
 * @author binghe
 *
 */
public class ColleagueA extends Colleague {
    public void action() {
        System.out.println("普通员工努力工作");
    }
}
```

```java
package com.binghe.design.mediator;
/**
 * Colleagueclass
 * @author binghe
 *
 */
public class ColleagueB extends Colleague {
    public void action() {
        System.out.println("前台注意了!");
    }
}
```

Test

```java
package com.binghe.design.mediator;
/**
 * Test
 * @author binghe
 *
 */
public class Test {
    public static void main(String[] args) {
        Mediator med = new ConcreteMediator();
        //老板来了
        med.notice("boss");

        //客户来了
        med.notice("client");
    }
}
```

Result

```
普通员工努力工作
前台注意了!
```

备忘录模式

一、概述

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。这样以后就可将该对象恢复到原先保存的状态。

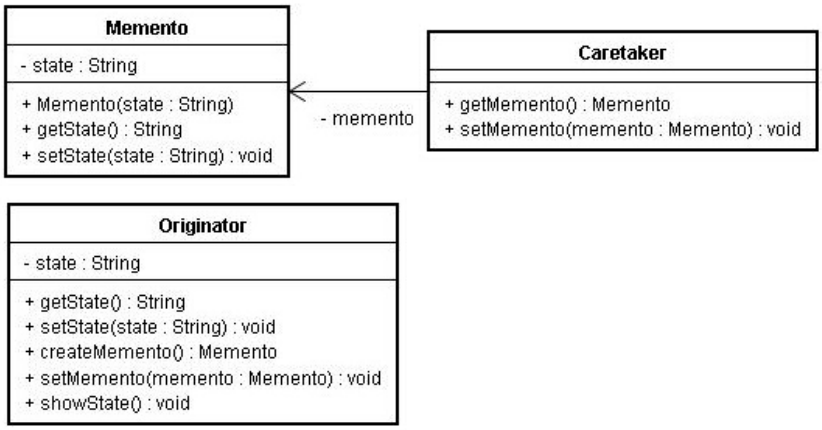
二、适用性

- 必须保存一个对象在某一个时刻的(部分)状态,这样以后需要时它才能恢复到先前的状态。
- 如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

三、参与者

- Memento 备忘录存储原发器对象的内部状态。
- Originator 原发器创建一个备忘录,用以记录当前时刻它的内部状态。 使用备忘录恢复内部状态.
- Caretaker 负责保存好备忘录。 不能对备忘录的内容进行操作或检查。

四、类图



五、示例

Memento

```
package com.binghe.design.memento;
/**
 * Memento
 * @author binghe
 *
 */
public class Caretaker {

    private Memento memento;

    public Memento getMemento(){
        return this.memento;
    }

    public void setMemento(Memento memento){
        this.memento = memento;
    }
}
```

Originator

```
package com.binghe.design.memento;
/**
 * Originator
 * @author binghe
 *
 */
public class Memento {

    private String state;

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

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

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

Caretaker

```
package com.binghe.design.memento;
/**
 * Caretaker
 * @author binghe
 *
 */
public class Originator {

    private String state;

    public String getState() {
        return state;
    }

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

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

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

    public void showState(){
        System.out.println(state);
    }
}
```

Test

```
package com.binghe.design.memento;
/**
 * Test
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {
        Originator org = new Originator();
        org.setState("开会中");

        Caretaker ctk = new Caretaker();
        ctk.setMemento(org.createMemento());//将数据封装在Caretaker

        org.setState("睡觉中");
```

```
        org.showState();//显示

        org.setMemento(ctk.getMemento());//将数据重新导入
        org.showState();
    }
}
```

Result

```
睡觉中
开会中
```

观察者模式

一、概述

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。

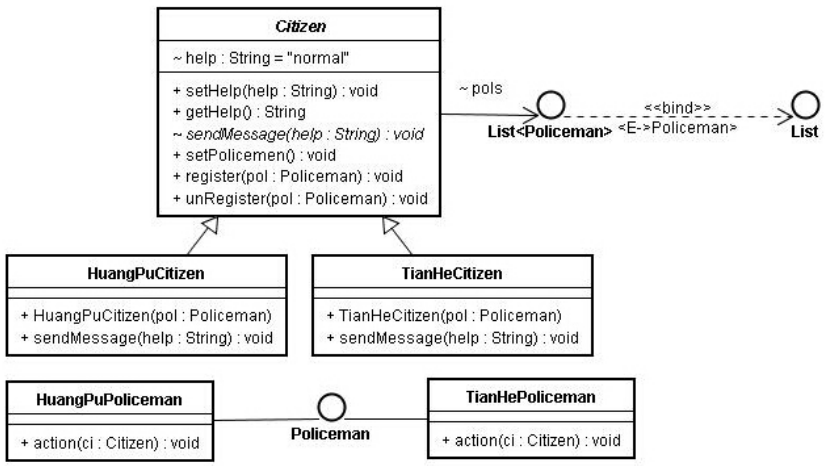
二、适用性

- 当一个抽象模型有两个方面,其中一个方面依赖于另一方面。 将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
- 当对一个对象的改变需要同时改变其它对象,而不知道具体有多少对象有待改变。
- 当一个对象必须通知其它对象，而它又不能假定其它对象是谁。

三、参与者

- Subject（目标） 目标知道它的观察者。可以有任意多个观察者观察同一个目标。 提供注册和删除观察者对象的接口。
- Observer（观察者） 为那些在目标发生改变时需获得通知的对象定义一个更新接口。
- ConcreteSubject（具体目标） 将有关状态存入各ConcreteObserver对象。 当它的状态发生改变时,向它的各个观察者发出通知。
- ConcreteObserver（具体观察者） 维护一个指向ConcreteSubject对象的引用。 存储有关状态，这些状态应与目标的状态保持一致。 实现Observer的更新接口以使自身状态与目标的状态保持一致

四、类图



五、示例

Subject

```
package com.binghe.design.observer;

import java.util.ArrayList;
import java.util.List;

/**
 * Subject
 * @author binghe
 *
 */
public abstract class Citizen {

    List pols;

    String help = "normal";

    public void setHelp(String help) {
        this.help = help;
    }

    public String getHelp() {
        return this.help;
    }

    abstract void sendMessage(String help);

    public void setPolicemen() {
        this.pols = new ArrayList();
    }

    public void register(Policeman pol) {
        this.pols.add(pol);
    }

    public void unRegister(Policeman pol) {
        this.pols.remove(pol);
    }
}
```

Observer

```
package com.binghe.design.observer;
/**
 * Observer
 * @author binghe
 *
 */
public interface Policeman {

    void action(Citizen ci);
}
```

ConcreteSubject

```java
package com.binghe.design.observer;
/**
 * ConcreteSubject
 * @author binghe
 *
 */
public class HuangPuCitizen extends Citizen {

    public HuangPuCitizen(Policeman pol) {
        setPolicemen();
        register(pol);
    }

    public void sendMessage(String help) {
        setHelp(help);
        for(int i = 0; i < pols.size(); i++) {
            Policeman pol = pols.get(i);
            //通知警察行动
            pol.action(this);
        }
    }
}
```

```java
package com.binghe.design.observer;
/**
 * ConcreteSubject
 * @author binghe
 *
 */
public class TianHeCitizen extends Citizen {

    public TianHeCitizen(Policeman pol) {
        setPolicemen();
        register(pol);
    }

    public void sendMessage(String help) {
        setHelp(help);
        for (int i = 0; i < pols.size(); i++) {
            Policeman pol = pols.get(i);
            //通知警察行动
            pol.action(this);
        }
    }
}
```

ConcreteObserver

```java
package com.binghe.design.observer;
/**
 * ConcreteObserver
 * @author binghe
 *
 */
public class HuangPuPoliceman implements Policeman {
```

```
    public void action(Citizen ci) {
        String help = ci.getHelp();
        if (help.equals("normal")) {
            System.out.println("一切正常, 不用出动");
        }
        if (help.equals("unnormal")) {
            System.out.println("有犯罪行为, 黄埔警察出动!");
        }
    }
}
```

```
package com.binghe.design.observer;
/**
 * ConcreteObserver
 * @author binghe
 *
 */
public class TianHePoliceman implements Policeman {

    public void action(Citizen ci) {
        String help = ci.getHelp();
        if (help.equals("normal")) {
            System.out.println("一切正常, 不用出动");
        }
        if (help.equals("unnormal")) {
            System.out.println("有犯罪行为, 天河警察出动!");
        }
    }
}
```

Test

```
package com.binghe.design.observer;
/**
 * Test
 * @author binghe
 *
 */
public class Test{

    public static void main(String[] args) {
        Policeman thPol = new TianHePoliceman();
        Policeman hpPol = new HuangPuPoliceman();

        Citizen citizen = new HuangPuCitizen(hpPol);
        citizen.sendMessage("unnormal");
        citizen.sendMessage("normal");

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

        citizen = new TianHeCitizen(thPol);
        citizen.sendMessage("normal");
        citizen.sendMessage("unnormal");
    }
}
```

Result

```
有犯罪行为，黄埔警察出动!
一切正常，不用出动
==========
一切正常，不用出动
有犯罪行为，天河警察出动!
```

状态模式

一、概述

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。

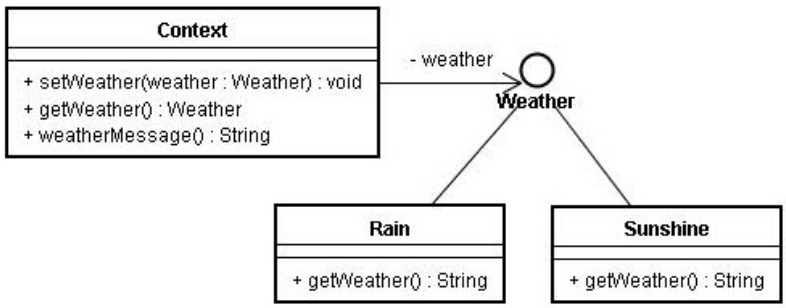
二、适用性

- 一个对象的行为取决于它的状态,并且它必须在运行时刻根据状态改变它的行为。
- 一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。 这个状态通常用一个或多个枚举常量表示。 通常,有多个操作包含这一相同的条件结构。 State模式将每一个条件分支放入一个独立的类中。 这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

三、适用性

- Context 定义客户感兴趣的接口。 维护一个ConcreteState子类的实例，这个实例定义当前状态。
- State 定义一个接口以封装与Context的一个特定状态相关的行为。
- ConcreteStatesubclasses 每一子类实现一个与Context的一个状态相关的行为。

四、类图



五、示例

Context

```
package com.binghe.design.state;
/**
 * Context
 * @author binghe
 *
 */
public class Context {

    private Weather weather;
```

```
    public void setWeather(Weather weather) {
        this.weather = weather;
    }

    public Weather getWeather() {
        return this.weather;
    }

    public String weatherMessage() {
        return weather.getWeather();
    }
}
```

State

```
package com.binghe.design.state;
/**
 * State
 * @author binghe
 *
 */
public interface Weather {

    String getWeather();
}
```

ConcreteStatesubclasses

```
package com.binghe.design.state;
/**
 * ConcreteStatesubclasses
 * @author binghe
 *
 */
public class Rain implements Weather {

    public String getWeather() {
        return "下雨";
    }
}
```

```
package com.binghe.design.state;
/**
 * ConcreteStatesubclasses
 * @author binghe
 *
 */
public class Sunshine implements Weather {

    public String getWeather() {
        return "阳光";
    }
}
```

Test

```
package com.binghe.design.state;
/**
 * Test
 * @author binghe
 *
 */
public class Test{

    public static void main(String[] args) {
        Context ctx1 = new Context();
        ctx1.setWeather(new Sunshine());
        System.out.println(ctx1.weatherMessage());

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

        Context ctx2 = new Context();
        ctx2.setWeather(new Rain());
        System.out.println(ctx2.weatherMessage());
    }
}
```

Result

```
阳光
===============
下雨
```

策略模式

一、概述

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。本模式使得算法可独立于使用它的客户而变化。

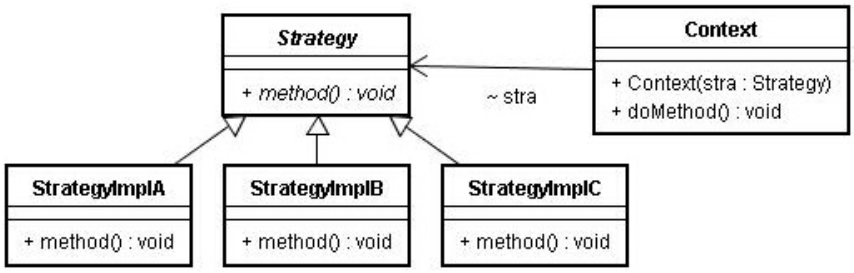
二、适用性

- 许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。
- 需要使用一个算法的不同变体。
- 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
- 一个类定义了多种行为,并且这些行为在这个类的操作中以多个条件语句的形式出现。 将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

三、参与者

- Strategy 定义所有支持的算法的公共接口。Context使用这个接口来调用某ConcreteStrategy定义的算法。
- ConcreteStrategy 以Strategy接口实现某具体算法。
- Context 用一个ConcreteStrategy对象来配置。 维护一个对Strategy对象的引用。 可定义一个接口来让Stategy访问它的数据。

四、类图



五、示例

Strategy

```
package com.binghe.design.strategy;
/**
 * Strategy
 * @author binghe
 *
 */
public abstract class Strategy {

    public abstract void method();
}
```

ConcreteStrategy

```
package com.binghe.design.strategy;
/**
 * ConcreteStrategy
 * @author binghe
 *
 */
public class StrategyImplA extends Strategy {

    public void method() {
        System.out.println("这是第一个实现");
    }
}
```

```
package com.binghe.design.strategy;
/**
 * ConcreteStrategy
 * @author binghe
 *
 */
public class StrategyImplB extends Strategy {

    public void method() {
        System.out.println("这是第二个实现");
    }
}
```

```java
package com.binghe.design.strategy;
/**
 * ConcreteStrategy
 * @author binghe
 *
 */
public class StrategyImplC extends Strategy {

    public void method() {
        System.out.println("这是第三个实现");
    }
}
```

Context

```java
package com.binghe.design.strategy;
/**
 * Context
 * @author binghe
 *
 */
public class Context {

    Strategy stra;

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

    public void doMethod() {
        stra.method();
    }
}
```

Test

```java
package com.binghe.design.strategy;
/**
 * Test
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {
        Context ctx = new Context(new StrategyImplA());
        ctx.doMethod();

        ctx = new Context(new StrategyImplB());
        ctx.doMethod();

        ctx = new Context(new StrategyImplC());
        ctx.doMethod();
    }
}
```

Resut

```
这是第一个实现
这是第二个实现
这是第三个实现
```

模板方法

一、概述

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。 TemplateMethod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。

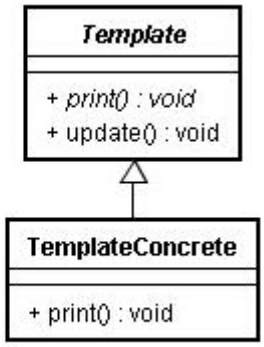
二、适用性

- 一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。
- 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。 首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。 最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 控制子类扩展。

三、参与者

- AbstractClass 定义抽象的原语操作（primitiveoperation），具体的子类将重定义它们以实现一个算法的各步骤。 实现一个模板方法,定义一个算法的骨架。 该模板方法不仅调用原语操作，也调用定义在AbstractClass或其他对象中的操作。
- ConcreteClass 实现原语操作以完成算法中与特定子类相关的步骤。

四、类图



五、示例

AbstractClass

```
package com.binghe.design.template;
/**
 * AbstractClass
 * @author binghe
 *
 */
public abstract class Template {

    public abstract void print();
```

```
    public void update() {
        System.out.println("开始打印");
        for (int i = 0; i < 10; i++) {
            print();
        }
    }
}
```

ConcreteClass

```
package com.binghe.design.template;
/**
 * ConcreteClass
 * @author binghe
 *
 */
public class TemplateConcrete extends Template {

    @Override
    public void print() {
        System.out.println("这是子类的实现");
    }
}
```

Test

```
package com.binghe.design.template;
/**
 * Test
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {
        Template temp = new TemplateConcrete();
        temp.update();
    }
}
```

Result

```
开始打印
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
这是子类的实现
```

访问者模式

一、概述

表示一个作用于某对象结构中的各元素的操作。 它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

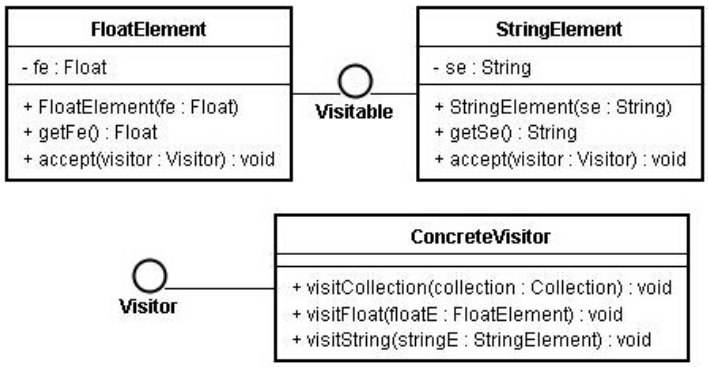
二、适用性

- 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作"污染"这些对象的类。 Visitor使得你可以将相关的操作集中起来定义在一个类中。 当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。
- 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。 改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。 如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

三、参与者

- Visitor 为该对象结构中ConcreteElement的每一个类声明一个Visit操作。 该操作的名字和特征标识了发送Visit请求给该访问者的那个类。 这使得访问者可以确定正被访问元素的具体的类。 这样访问者就可以通过该元素的特定接口直接访问它。
- ConcreteVisitor 实现每个由Visitor声明的操作。 每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。 ConcreteVisitor为该算法提供了上下文并存储它的局部状态。 这一状态常常在遍历该结构的过程中累积结果。
- Element 定义一个Accept操作，它以一个访问者为参数。
- ConcreteElement 实现Accept操作，该操作以一个访问者为参数。
- ObjectStructure 能枚举它的元素。 可以提供一个高层的接口以允许该访问者访问它的元素。 可以是一个复合或是一个集合，如一个列表或一个无序集合。

四、类图



五、示例

Visitor

```
package com.binghe.design.visitor;

import java.util.Collection;

/**
 * Visitor
 * @author binghe
 *
```

```
 */
public interface Visitor {

    public void visitString(StringElement stringE);

    public void visitFloat(FloatElement floatE);

    public void visitCollection(Collection collection);
}
```

ConcreteVisitor

```
package com.binghe.design.visitor;

import java.util.Collection;
import java.util.Iterator;

/**
 * ConcreteVisitor
 * @author binghe
 *
 */
public class ConcreteVisitor implements Visitor {

    public void visitCollection(Collection collection) {
        // TODO Auto-generated method stub
        Iterator iterator = collection.iterator();
        while (iterator.hasNext()) {
            Object o = iterator.next();
            if (o instanceof Visitable) {
                ((Visitable)o).accept(this);
            }
        }
    }

    public void visitFloat(FloatElement floatE) {
        System.out.println(floatE.getFe());
    }

    public void visitString(StringElement stringE) {
        System.out.println(stringE.getSe());
    }
}
```

Element

```
package com.binghe.design.visitor;
/**
 * Element
 * @author binghe
 *
 */
public interface Visitable {

    public void accept(Visitor visitor);
}
```

ConcreteElement

```
package com.binghe.design.visitor;
/**
 * ConcreteElement
 * @author binghe
 *
 */
public class FloatElement implements Visitable {

    private Float fe;

    public FloatElement(Float fe) {
        this.fe = fe;
    }

    public Float getFe() {
        return this.fe;
    }

    public void accept(Visitor visitor) {
        visitor.visitFloat(this);
    }
}
```

```
package com.binghe.design.visitor;
/**
 * ConcreteElement
 * @author binghe
 *
 */
public class StringElement implements Visitable {

    private String se;

    public StringElement(String se) {
        this.se = se;
    }

    public String getSe() {
        return this.se;
    }

    public void accept(Visitor visitor) {
        visitor.visitString(this);
    }
}
```

Test

```
package com.binghe.design.visitor;

import java.util.ArrayList;
import java.util.List;

/**
```

```
 * Test
 * @author binghe
 *
 */
public class Test {

    public static void main(String[] args) {
        Visitor visitor = new ConcreteVisitor();
        StringElement se = new StringElement("abc");
        se.accept(visitor);

        FloatElement fe = new FloatElement(new Float(1.5));
        fe.accept(visitor);
        System.out.println("===========");
        List result = new ArrayList();
        result.add(new StringElement("abc"));
        result.add(new StringElement("abc"));
        result.add(new StringElement("abc"));
        result.add(new FloatElement(new Float(1.5)));
        result.add(new FloatElement(new Float(1.5)));
        result.add(new FloatElement(new Float(1.5)));
        visitor.visitCollection(result);
    }
}
```

Result

```
abc
1.5
===========
abc
abc
abc
1.5
1.5
1.5
```

写在最后

> 如果你觉得冰河写的还不错，请微信搜索并关注「**冰河技术**」微信公众号，跟冰河学习高并发、分布式、微服务、大数据、互联网和云原生技术，「**冰河技术**」微信公众号更新了大量技术专题，每一篇技术文章干货满满！不少读者已经通过阅读「**冰河技术**」微信公众号文章，吊打面试官，成功跳槽到大厂；也有不少读者实现了技术上的飞跃，成为公司的技术骨干！如果你也想像他们一样提升自己的能力，实现技术能力的飞跃，进大厂，升职加薪，那就关注「**冰河技术**」微信公众号吧，每天更新超硬核技术干货，让你对如何提升技术能力不再迷茫！

冰河技术微信公众号

微信公众号：冰河技术

微信：sun_shine_lyz

扫码关注！