# 设计模式之禅

# 迪米特法则

迪米特法则(Law of Demeter,LoD) 也称为最少知识原则(Least Knowledge Principle,LKP),

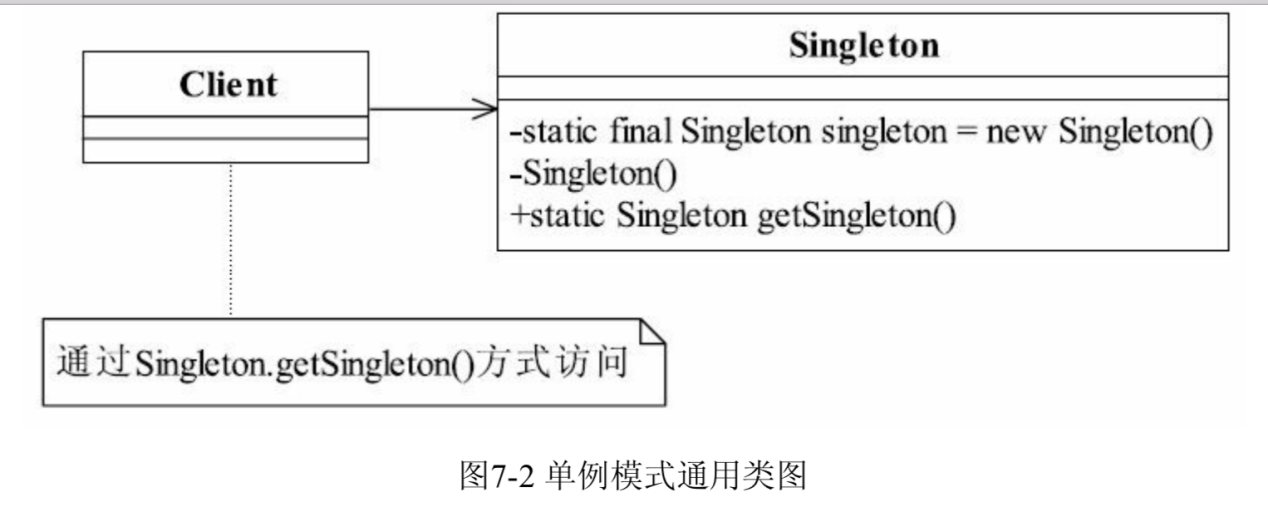
一个对象应该对其他对象有最少的了解。一个类应该对自己需要耦合的或调用的类知道的最少。

# 迪米特法则

# 单例模式

## 7.2 单例模式的定义

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例



## 7.3 单例模式的应用

### 7.3.1 单例模式的优点

1. 内存中只有一个实例，减少了内存开支，特别是一个对象需要频地创建、销毁时，而且创建或销毁时性能又无法优化，单例模式的优势就非常明显
2. 由于单例模式只生成一个实例，所以减少了系统的性能开销，当一个对象的产生需要比较多的资源时，如读取配置，产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。
3. 单例模式可以避免对资源的多重占用，例如一个写文件动作，由于只有一个实例存在内存中，避免对同一个资源文件的同时写操作。
4. 单例模式可以在系统设置全局的访问点，优化和共享资源访问，例如可以设计一个单例类，负责所有数据表的映射处理。

### 7.3.2 单例模式的缺点

1. 单例模式一般没有接口，扩展很困难，若要扩展，除了修改代码基本上没有第二种途径可以实现。
2. 单例模式对测试是不利的。
3. 单例模式与单一职责原则有冲突。一个类应该只实现一个逻辑，而不关心它是否是单例的，是不是要单例取决于环境。

## 7.4 单例模式的实现方式

### 7.4.1 饿汉式

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 饿汉式  \*/* public class SingletonObject1 {   private static final SingletonObject1 *instance* = new SingletonObject1();   private SingletonObject1() {  }   public static SingletonObject1 getInstance() {  return *instance*;  }  } |

### 7.4.2 懒汉式：线程不安全的

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:存在多线程安全问题  \*/* public class SingletonObject2 {   private static SingletonObject2 *instance*;   private SingletonObject2(){  }   public static SingletonObject2 getInstance(){  if(null == *instance*){  *instance* = new SingletonObject2();  }  return SingletonObject2.*instance*;  }  } |

### 7.4.3 懒汉式：同步方法

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:方法加同步，解决多线程问题  \*/* public class SingletonObject3 {   private static SingletonObject3 *instance*;   private SingletonObject3(){  }   public synchronized static SingletonObject3 getInstance(){  if(null == *instance*){  *instance* = new SingletonObject3();  }  return SingletonObject3.*instance*;  }   } |

### 7.4.4 懒汉式：double check 同步代码块

依旧存在线程安全问题

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式:double check方式解决多线程问题，实际上还是解决不了  \*/* public class SingletonObject4 {    private static SingletonObject4 *instance*;   private SingletonObject4() {   }   public static SingletonObject4 getInstance() {  if (null == *instance*) {  synchronized (SingletonObject4.class) {  if (null == *instance*) {  *instance* = new SingletonObject4();  }  }  }  return SingletonObject4.*instance*;  }  } |

### 7.4.5 懒汉式：double check 同步代码块 volatile关键字

|  |
| --- |
| package com.book.study.zen.chapter07;  */\*\*  \* 懒汉式：用volatile关键字解决double check问题  \*/* public class SingletonObject5 {   // 保证可见性、避免重排序  private static volatile SingletonObject5 *instance*;   private SingletonObject5() {   }   public static SingletonObject5 getInstance() {  if (null == *instance*) {  synchronized (SingletonObject5.class) {  if (null == *instance*) {  *instance* = new SingletonObject5();  }  }  }  return SingletonObject5.*instance*;  }   } |

### 7.4.6 静态内部类

|  |
| --- |
| package com.book.study.zen.chapter07;   */\*\*  \* 静态内部类的方式：当访问SingletonObject6的类时，会先初始化InstanceHolder，于是达到了访问的时候才初始化，而且由于是静态内部类实现的  \* 所以，  \*/* public class SingletonObject6 {   */\*\*  \* 1.类装载  \* 2.初始化  \*/* private SingletonObject6(){   }   private static class InstanceHolder{  private final static SingletonObject6 *intance* = new SingletonObject6();  }   public static SingletonObject6 getInstance(){  return InstanceHolder.*intance*;  }    } |

### 7.4.7 枚举1

|  |
| --- |
| package com.book.study.zen.chapter07;  public class SingletonObject7 {   private SingletonObject7(){   }   private enum Singleton{  *INSTANCE*,;   private final SingletonObject7 instance;   Singleton(){  instance = new SingletonObject7();  }   public SingletonObject7 getInstance(){  return instance;  }  }    public static SingletonObject7 getInstance(){  return Singleton.*INSTANCE*.getInstance();  }   } |

### 7.4.7 枚举2

|  |
| --- |
| package com.book.study.zen.chapter07;  public class DBConnection { }  package com.book.study.zen.chapter07;  */\*\*  \* 枚举实现单例模式  \*/* public enum SingletonObject8 {   *INSTANCE*,  ;   private final DBConnection instance;   private SingletonObject8() {  instance = new DBConnection();   }   public DBConnection getInstance() {  return instance;  }   } |

# 工厂方法模式

## 8.2 工厂方法模式的定义

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。工厂方法是一个类的实例化延迟到其子类。

|  |
| --- |
| package com.book.study.zen.chapter08;  public abstract class Product {   // 产品类的公共方法  public void method1(){  // 业务逻辑处理  }   // 抽象方法  public abstract void method2();  }  package com.book.study.zen.chapter08;  public class ConcreteProduct1 extends Product {    @Override  public void method2() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter08;  public class ConcreteProduct2 extends Product {   @Override  public void method2() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter08;  public abstract class Creator {   public abstract <T extends Product> T createProduct(Class<T> c);  }  package com.book.study.zen.chapter08;  public class ConcreteCreator extends Creator {   @Override  public <T extends Product> T createProduct(Class<T> c) {  Product product = null;  try {  product = (Product) Class.*forName*(c.getName()).newInstance();  }catch (Exception e){  // 异常处理   }  return (T) product;  } } |

## 8.3 工厂方法模式的应用

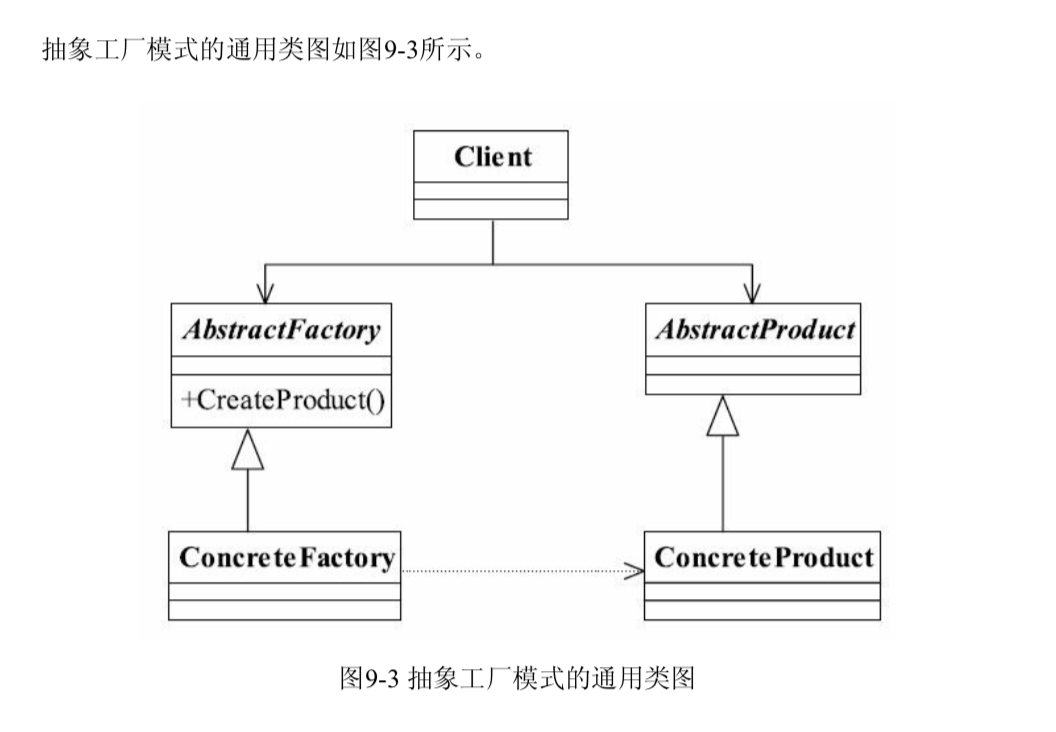
### 8.3.1 工厂方法模式的优点

1. 首先，良好的封装性，代码结构清晰。
2. 其次，工厂方法模式的扩展性非常优秀。
3. 再次，屏蔽产品类。
4. 工厂方法模式是典型的解耦框架。高层模块值不需要知道产品的抽象类，其他的实现类不用关心，复合迪米特法则，我不需要的就不要去交流。也符合依赖倒置原则。

# 抽象工厂

## 9.2 抽象工厂模式的定义

为创建一组相关或相互依赖的对象提供一个接口，而且无须指定他们的具体类



## 9.3 抽象工厂模式的应用

### 9.3.1 抽象工厂模式的优点

1. 封装性，每个产品的实现类不是高层模块要关心的。
2. 产品族类的约束为非公开状态。

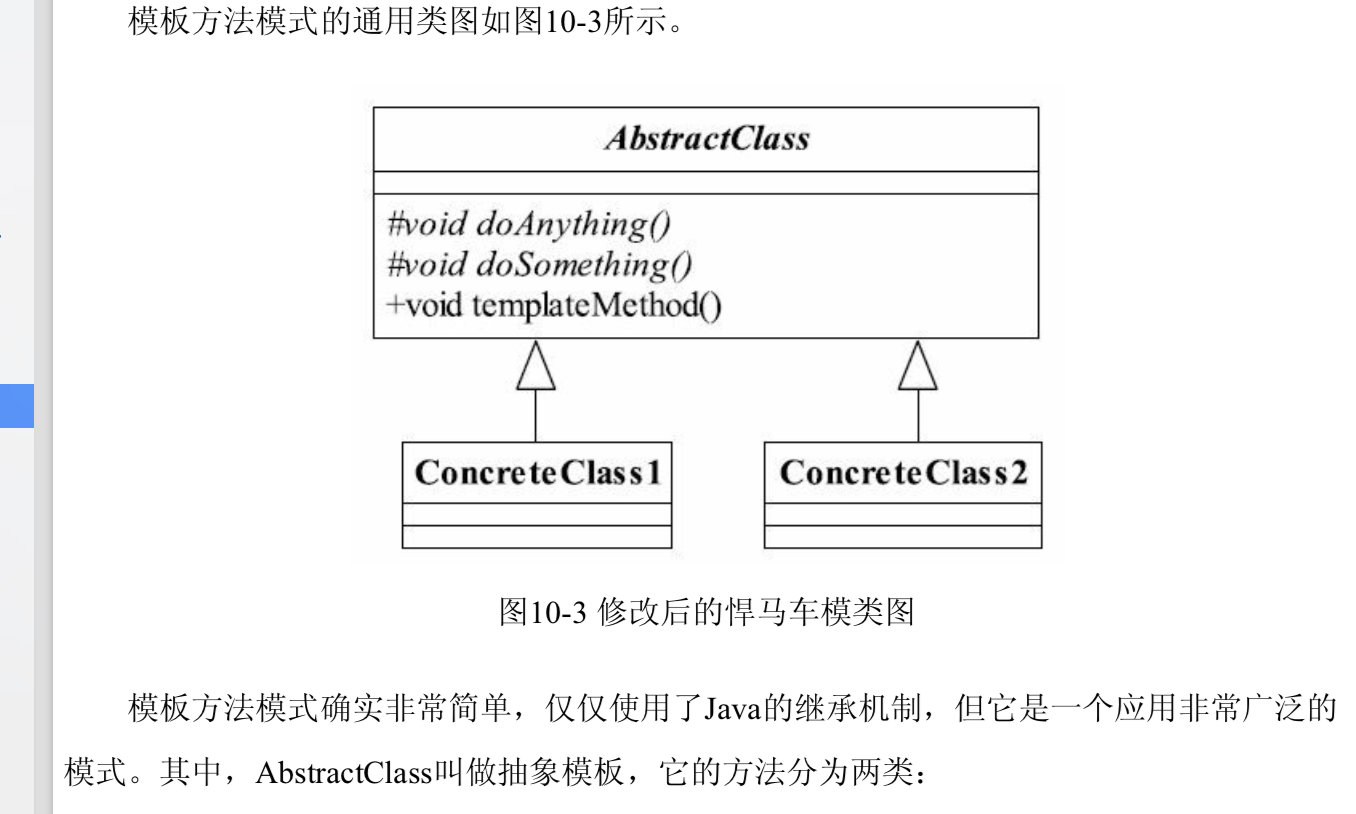
### 9.3.1 抽象工厂模式的缺点

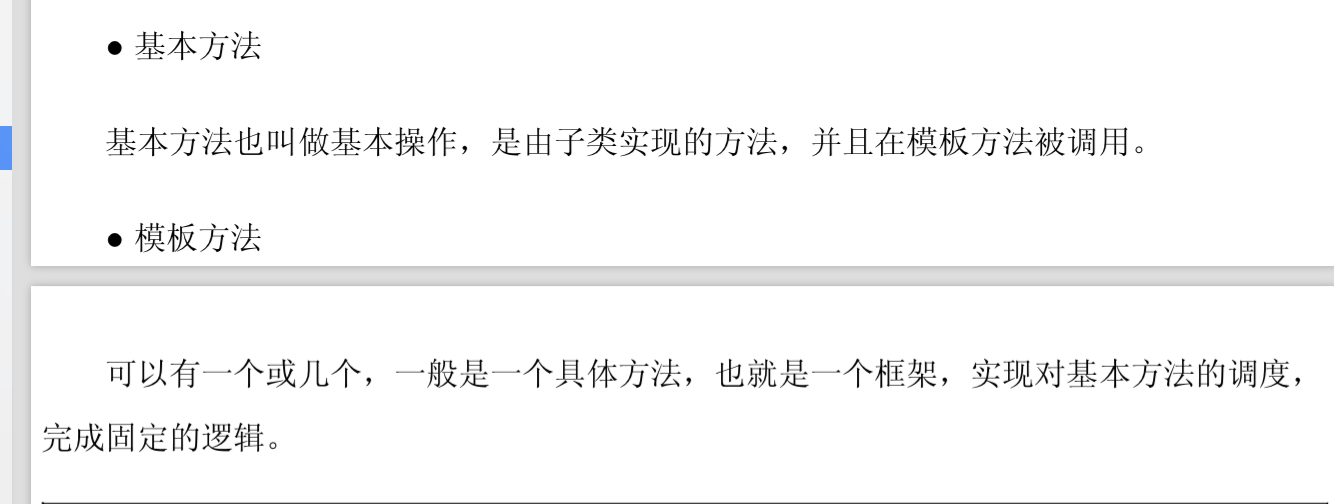
抽象工厂的模式的最大缺点就是产品族的扩展非常困难。

# 模板方法

## 10.2 模板方法的定义

定义一个操作中的算法的框架，而将一些步骤延迟到子类中。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





注意：

1. 为了防止恶意的操作，一般模板方法都加上final关键字，不允许被覆盖
2. 抽象模板中的基本方法尽量设计为protected类型，复合迪米特法则，不需要暴露属性或方法精良不要设置为protected类型。实现类若非必要，尽量不要扩大父类中的访问权限。

## 10.3 模板方法模式的应用

优点：

1. 封装不变不封，扩展可变部分
2. 提取公共部分代码，便于维护
3. 行为由父类控制，子类实现

缺点：

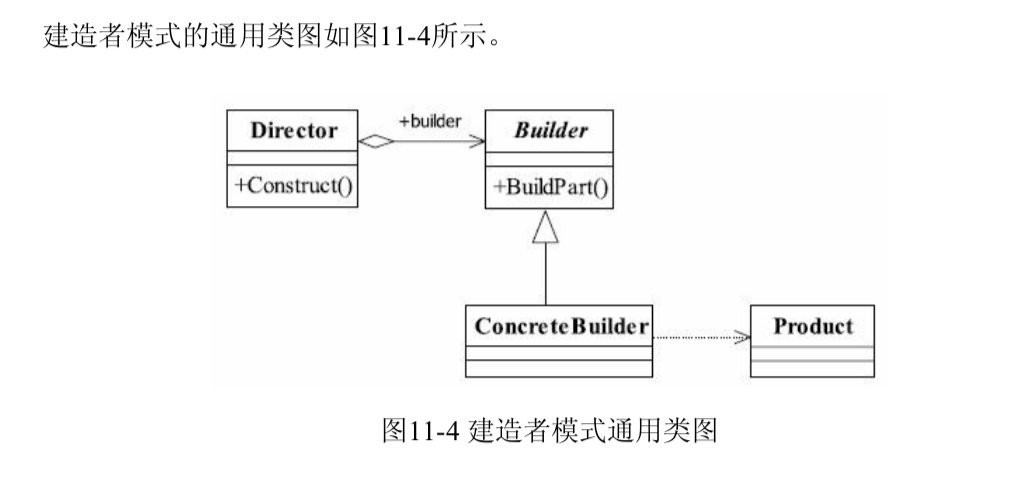
代码阅读起来有难度。

# 建造者模式

## 11.2 建造者模式的定义

建造者模式(Builder Pattern) 也叫做生成器模式，其定义如下：

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



建造者模式中，4个角色：

1. Product产品类:

通常是实现了模板方法模式，也就是有模板方法和基本方法，这个参考第10章的模板方法模式。

1. Builder抽象建造者

规范产品的组建，一般是由子类实现。例子中的CarBuilder就属于抽象建造者。

1. ConcreteBuilder具体建造者

实现抽象类定意思的所有方法，并且返回一个组建好的对象。例子中的BenzBuilder和BMWBuilder就叔叔于具体建造者。

1. Director导演类

负责安排已有模块的顺序，然后告诉Builder开始建造。

## 11.3 建造者模式的优点

### 11.3.1 建造者模式的优点

1. 封装性

使用建造者模式可以使客户端不必知道产品内部组成的细节，

1. 建造者独立，容易扩展

BenzBuilder和BMWBuilder是相互独立的，对系统的扩展非常有利

1. 便于控制细节风险

由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响

### 11.3.2 建造者模式的使用场景

1. 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效果，这个时候使用建造者模式非常合适
4. 在对象创建过程中会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不易得到时，也可以采用建造者模式封装该对象的创建过程。这种场景只能是一个补偿方法，因为一个对象不容易获得，而在设计阶段竟然没有发觉。而要通过创建者模式柔滑创建过程，本身已经违反了设计最初目标。

**建造者模式最主要的功能是基本方法的调用顺序安排。通俗的说就是零件的装配，顺序不同产生的对象也不同；而工厂方法侧重点事创建，创建零件是它的主要职责，组装顺序则不是它所关心的。**

## 11.4 建造者模式

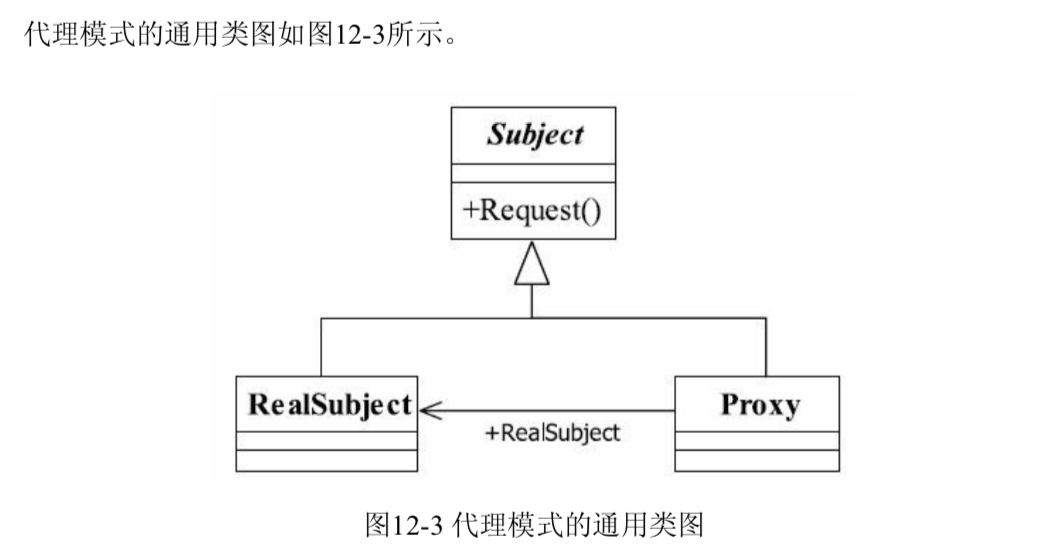
|  |
| --- |
| package com.book.study.zen.chapter11.demob;  public class Product {   public void doSomething(){  // 独立业务处理  }   }  package com.book.study.zen.chapter11.demob;  public abstract class Builder {   // 设置产品的不同部分，以获得不同的产品  public abstract void setPart();   // 建造产品  public abstract Product buildProduct();  }  package com.book.study.zen.chapter11.demob;  public class ConcreteProduct extends Builder {   private Product product = new Product();    @Override  public void setPart() {  */\*\*  \* 产品类内的逻辑处理  \*/* }   // 组建一个产品  @Override  public Product buildProduct() {  return product;  } }  package com.book.study.zen.chapter11.demob;  public class Director {   private Builder builder = new ConcreteProduct();   // 构建不同的产品  public Product getProduct() {  builder.setPart();   */\*\*  \* 设置不同的零件，产生不同的产品  \*/* return builder.buildProduct();  }   } |

# 代理模式

## 12.2 代理模式的定义

代理模式(Proxy pattern)是一个使用率非常高的模式，其定意思如下：

为其他对象提供一种代理以控制这个对象的访问。



1. Subject抽象主题角色

抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。

1. RealSubject具体主题角色

也叫做被委托角色，被代理角色。

1. Proxy代理主题角色
2. 也叫做委托类、代理类。它负责对这恩施角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题角色处理完毕前后做处理和善后工作

|  |
| --- |
| package com.book.study.zen.chapter12.demob;  public interface Subject {  // 定义一个方法  public void request(); }  package com.book.study.zen.chapter12.demob;  public class RealSubject implements Subject {   @Override  public void request() {  // 业务逻辑处理  } }  package com.book.study.zen.chapter12.demob;  public class Proxy implements Subject {   // 要代理哪个实现类  private Subject subject = null;   // 默认被代理者  public Proxy(){  this.subject = new Proxy();  }   // 通过构造函数传递代理者  @Override  public void request() {  this.before();  this.subject.request();  this.after();  }   // 预处理  private void after() {   }   // 善后处理  private void before() {   }   } |

## 12.3 代理模式的应用

### 12.3.1 代理模式的优点

1. 职责清晰

真实的角色就是实际的业务逻辑，不用关系其他非本质职责的事务，通过后期的代理完成一件事务，附带的结果就是编程简洁清晰

1. 高扩展性

具体主题角色是随时都会发生变化的，只要它实现了接口，

1. 智能化

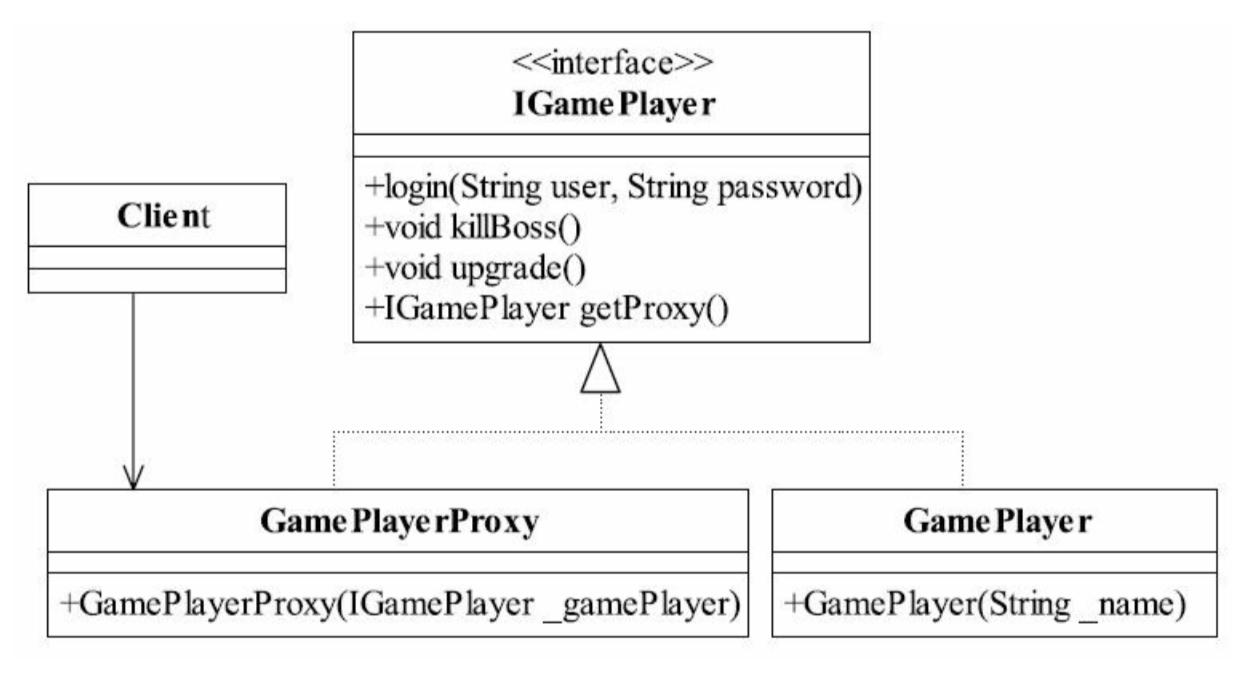
## 12.4 代理模式的扩展

### 12.4.1 普通代理

|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayer implements IGamePlayer {   private String name = "";    // 构造函数限制谁能创建对象，并同时传递姓名  public GamePlayer(IGamePlayer \_gamePlayer, String \_name) throws Exception {  if (\_gamePlayer == null) {  throw new Exception("不能创建真实角色");  } else {  this.name = \_name;  }  }    // 进游戏之前你肯定要登陆吧，这是一个必要条件  @Override  public void login(String user, String password) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");  }   // 打怪，最期望的就是杀老怪  @Override  public void killBoss() {  System.*out*.println(this.name + "在打怪");  }   // 升级，升级有很多方法，花钱买是一种，做任务也是一种  @Override  public void upgrade() {  System.*out*.println(this.name + "又升了一级");  } } |

|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayerProxy implements IGamePlayer {   private IGamePlayer gamePlayer = null;   // 通过构造函数传递要对谁进行代练   public GamePlayerProxy(String name) {  try {  this.gamePlayer = new GamePlayer(this,name);  }catch (Exception e){  // *TODO 异常处理* }   }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   // 通过构造函数传递要对谁进行代练  @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   // 通过构造函数传递要对谁进行代练  @Override  public void upgrade() {  this.gamePlayer.upgrade();  } } |

### 12.4.2 强制代理

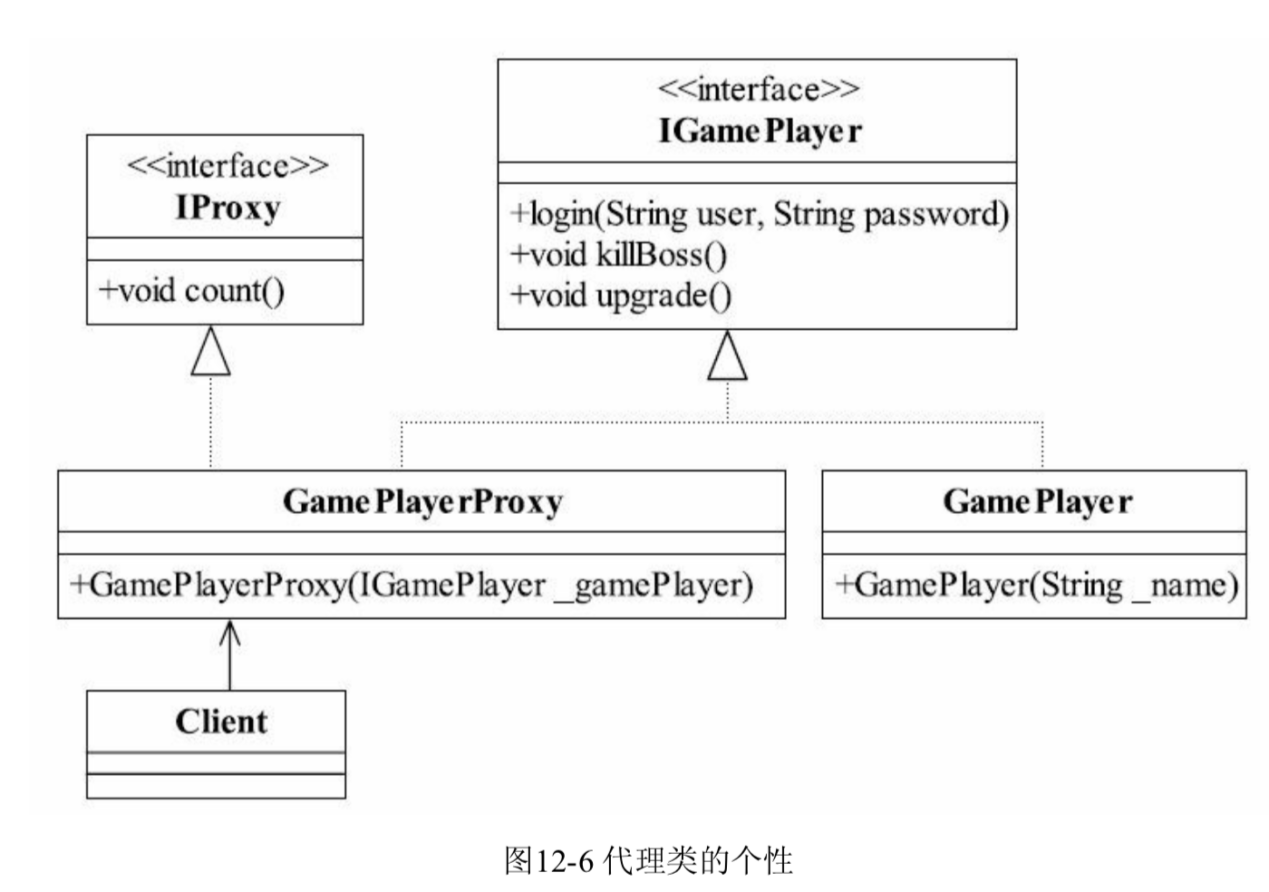


|  |
| --- |
| package com.book.study.zen.chapter12;  public interface INewGamePlayer {   // 登陆游戏  public void login(String user, String password);   // 杀怪，网络游戏的主要特色  public void killBoss();   // 升级  public void upgrade();    public INewGamePlayer getProxy();  } |

|  |
| --- |
| package com.book.study.zen.chapter12.demod;  import com.book.study.zen.chapter12.INewGamePlayer;  public class GamePlayer implements INewGamePlayer {   private String name = "";   private INewGamePlayer proxy = null;   public GamePlayer(String name) {  this.name = name;  }   @Override  public void login(String user, String password) {  if (this.isProxy()) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");   } else {  System.*out*.println("请使用指定的代理访问");  }  }   private boolean isProxy() {  return this.proxy != null;  }   @Override  public void killBoss() {  if (this.isProxy()) {  System.*out*.println(this.name + "在打怪");  } else {  System.*out*.println("请使用指定的代理访问");  }  }   @Override  public void upgrade() {  if (this.isProxy()) {  System.*out*.println(this.name + " 又升了一级");  } else {  System.*out*.println("请使用指定的代理访问");  }   }   @Override  public INewGamePlayer getProxy() {  this.proxy = new GamePlayerProxy(this);  return this.proxy;  } } |

|  |
| --- |
| package com.book.study.zen.chapter12.demod;   import com.book.study.zen.chapter12.INewGamePlayer;  public class GamePlayerProxy implements INewGamePlayer {    private INewGamePlayer gamePlayer = null;   public GamePlayerProxy(INewGamePlayer gamePlayer) {  this.gamePlayer = gamePlayer;  }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   @Override  public void upgrade() {  this.gamePlayer.upgrade();  }   @Override  public INewGamePlayer getProxy() {  return this;  } } |

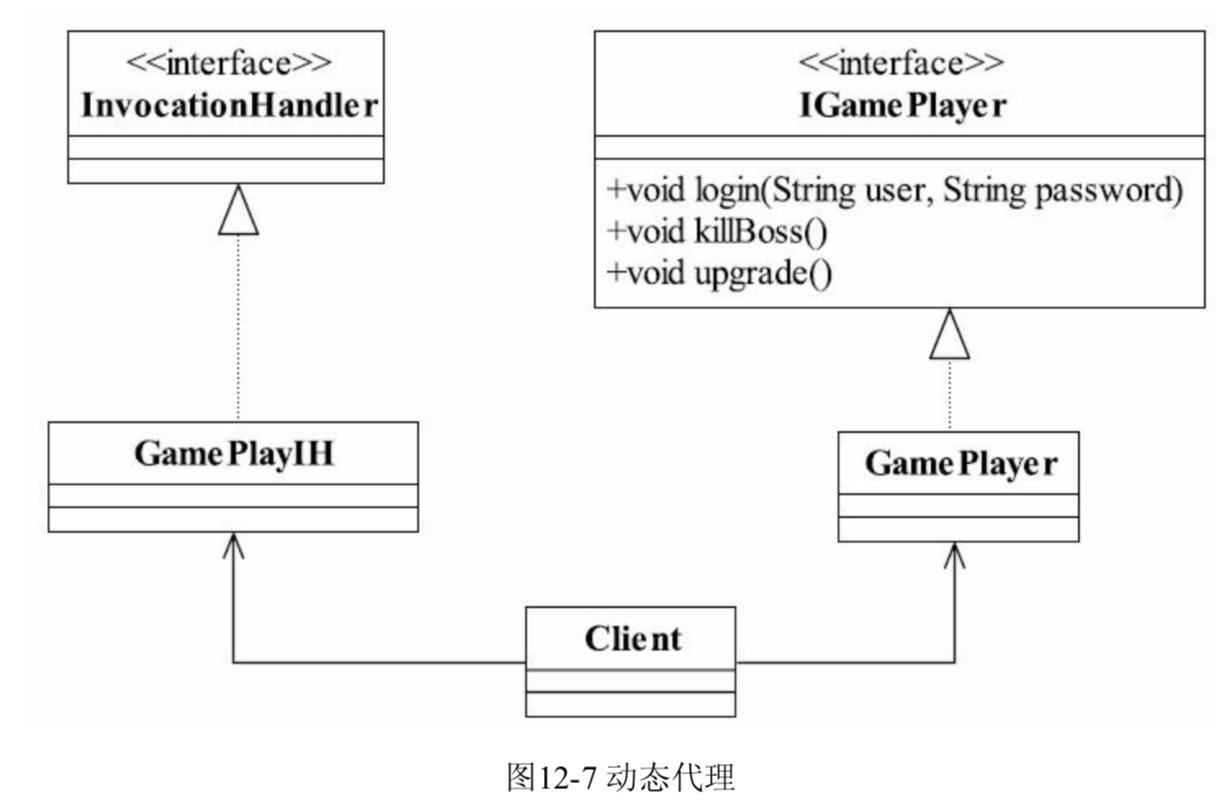
### 12.4.3 代理是有个性的



|  |
| --- |
| package com.book.study.zen.chapter12.demoe;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayer implements IGamePlayer {   private String name = "";    // 构造函数限制谁能创建对象，并同时传递姓名  public GamePlayer(String \_name) {  this.name = \_name;  }    // 进游戏之前你肯定要登陆吧，这是一个必要条件  @Override  public void login(String user, String password) {  System.*out*.println("登陆名为" + user + "的用户" + this.name + "登陆成功!");  }   // 打怪，最期望的就是杀老怪  @Override  public void killBoss() {  System.*out*.println(this.name + "在打怪");  }   // 升级，升级有很多方法，花钱买是一种，做任务也是一种  @Override  public void upgrade() {  System.*out*.println(this.name + "又升了一级");  } } |

|  |
| --- |
| package com.book.study.zen.chapter12.democ;  import com.book.study.zen.chapter12.IGamePlayer;  public class GamePlayerProxy implements IGamePlayer {   private IGamePlayer gamePlayer = null;   // 通过构造函数传递要对谁进行代练   public GamePlayerProxy(String name) {  try {  this.gamePlayer = new GamePlayer(this,name);  }catch (Exception e){  // *TODO 异常处理* }   }   @Override  public void login(String user, String password) {  this.gamePlayer.login(user,password);  }   // 通过构造函数传递要对谁进行代练  @Override  public void killBoss() {  this.gamePlayer.killBoss();  }   // 通过构造函数传递要对谁进行代练  @Override  public void upgrade() {  this.gamePlayer.upgrade();  } } |

### 12.4.3 动态代理

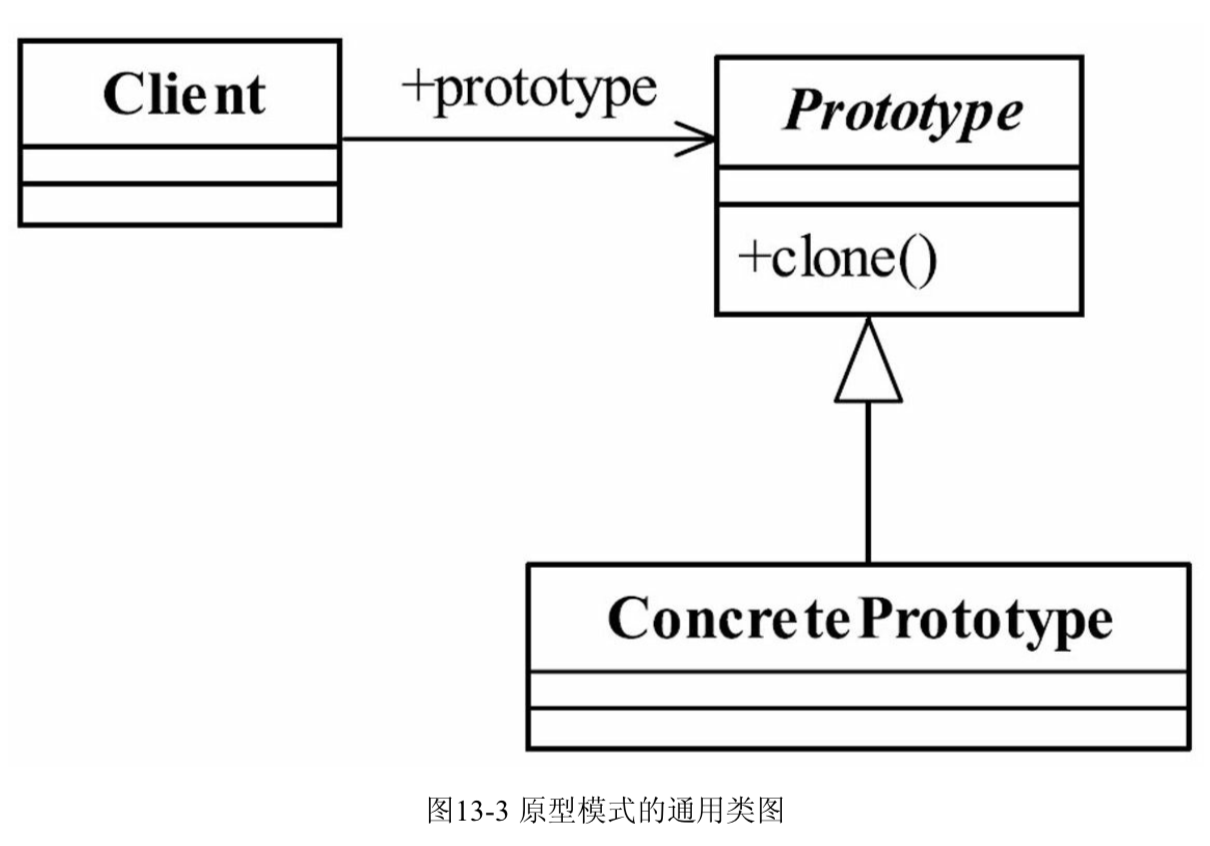


# 原型模式

## 13.2 原型模式的定义

原型模式（Prototype Pattern）的简单程度仅次于单例模式和迭代器模式。

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。



## 13.3 原型模式的应用

### 13.3.1 原型模式的优点

1.性能优良：原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。



1. 逃避构造函数的约束

这既是优点也是缺点，直接在内存中拷贝，构造函数式不会执行的。优点是减少了约束，缺点也是减少了约束。

### 13.3.2 原型模式的使用场景

1. 资源优化场景：类初始化需要消化非常多的资源，这个资源包括数据，硬件资源等
2. 性能和安全要求的场景：通过new产生一个对象需要非常繁琐的数据准备或方位权限，则可以使用原型模式。
3. 一个对象对各修改者的场景

一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。

## 13.4 原型模式的注意事项

### 13.4.1 构造函数不会被执行

Object类的clone方法的原理是从内存中（具体地说就是堆内存）以二进制流的方式进行拷贝，重新分配一个内存块，那构造函数没有被执行也是风场正常的。

### 13.4.2 浅拷贝和深拷贝

浅拷贝

|  |
| --- |
| package com.book.study.zen.chapter13.demod;  import java.util.ArrayList;  public class Thing implements Cloneable {   // 定义一个私有变量  private ArrayList<String> arrayList = new ArrayList<>();    @Override  protected Thing clone() {  Thing thing = null;  try {  thing = (Thing) super.clone();  }catch (CloneNotSupportedException e){  e.printStackTrace();  }  return thing;  }   // 设置HashMap的值  public void setValue(String value){  this.arrayList.add(value);  }   // 取得arrayList的值  public ArrayList<String> getValue(){  return this.arrayList;  } }  package com.book.study.zen.chapter13.demod;  public class Client {   public static void main(String[] args) {  // 产生一个对象  Thing thing = new Thing();  // 设置一个值  thing.setValue("张三");  // 拷贝一个对象  Thing cloneThing = thing.clone();  cloneThing.setValue("李四");  // [张三, 李四]  System.*out*.println(thing.getValue());   } }  输出可以看到，两者共用了arrayList属性。基本类型int，long，char和String类型是可以拷贝的，引用类型是不能拷贝的。这就是签拷贝 |

深拷贝

|  |
| --- |
| package com.book.study.zen.chapter13.demod;  import java.util.ArrayList;  public class Thing implements Cloneable {   // 定义一个私有变量  private ArrayList<String> arrayList = new ArrayList<>();    @Override  protected Thing clone() {  Thing thing = null;  try {  thing = (Thing) super.clone();  thing.arrayList = (ArrayList<String>) this.arrayList.clone();  }catch (CloneNotSupportedException e){  e.printStackTrace();  }  return thing;  }   // 设置HashMap的值  public void setValue(String value){  this.arrayList.add(value);  }   // 取得arrayList的值  public ArrayList<String> getValue(){  return this.arrayList;  } }  package com.book.study.zen.chapter13.demod;  public class Client {   public static void main(String[] args) {  // 产生一个对象  Thing thing = new Thing();  // 设置一个值  thing.setValue("张三");  // 拷贝一个对象  Thing cloneThing = thing.clone();  cloneThing.setValue("李四");   // [张三]  System.*out*.println(thing.getValue());  System.*out*.println(cloneThing.getValue());  // 1580066828  System.*out*.println(thing.hashCode());  // 491044090  System.*out*.println(cloneThing.hashCode());  } }  深拷贝就解决了浅拷贝的问题 |

### 13.4.3 clone与final

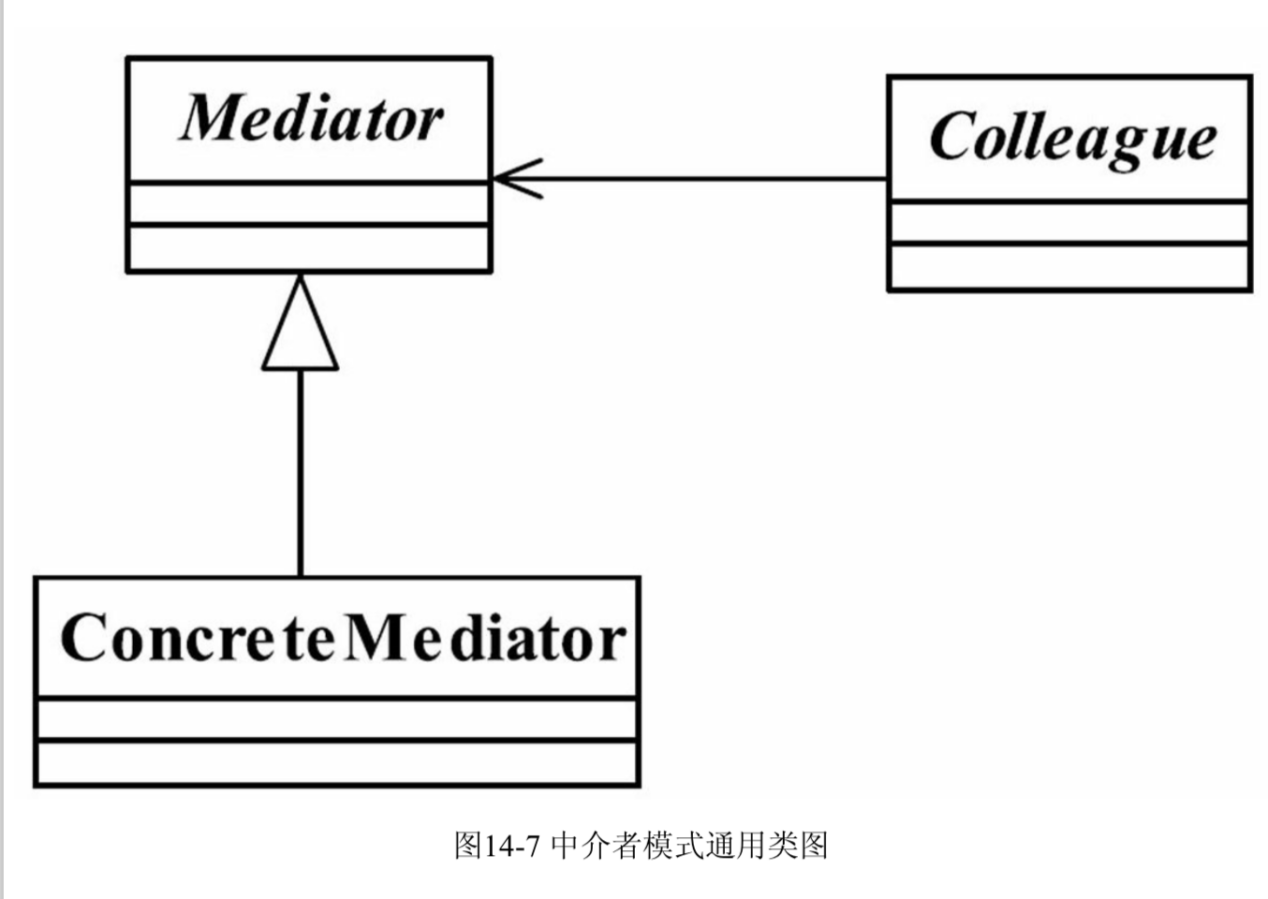
对象的clone与对象内的final关键字是由冲突的。

要使用clone方法，类的成员变量上不要增加final关键字

# 中介者模式

## 14.2 中介者模式的定义

中介者模式的定义：用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变他们之间的交互。



1.Mediator抽象中介者角色

抽象中介者角色定义统一的接口，用于各同事角色之间的通信

1. Concrete Mediator具体中介者角色

具体中介者角色通过协调个同事角色实现协作行为，因此它必须依赖于各个同时角色。

1. Colleague同时角色

每一个同事角色都知道中介者角色，而且与其他的同时角色通信的时候，一定要通过中介者协作。每个同事类的行为分为两种：一种是同事本身的行为，比如改变对本身的状态，处理自己的行为等，这种行为叫做自发行为(Self-Method)，与其他的同事类或中介者没有任何的依赖；第二种时必须依赖中介者才能完成的行为，叫做依赖方法(DepMethod)。

## 14.3 中介者模式的应用

### 14.3.1 中介者模式的优点

中介者模式的优点就是减少类间的依赖，把原有的一对多的依赖变成一对一依赖，同事类只依赖中介者，减少了依赖，当然同事也降低了类间的耦合。

### 14.3.2 中介者模式的缺点

中介者模式的缺点就是中介者会膨胀得很大，而且逻辑复杂，原本N个对象直接的相互依赖关系转换为中介者和同事类的依赖关系，同事类越多，中介者的逻辑就越复杂。

### 14.3.3 中介者模式的使用场景

# 中介者模式

## 15.2 命令模式的定义

命令模式是一个高内聚的模式：将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排对或者记录请求日志，可以提供命令的撤销和恢复功能。

## 15.3 命令模式的应用

### 15.3.1 命令模式的优点

1. 类间解耦：调用者角色与接收者角色没有任何依赖关系，调用者实现功能时只需调用Command抽象类的execute方法就可以，不需要了解到底是哪个接收者执行。
2. 可扩展性：Command的子类可以非常容易地扩展，而调用者Invoker和高层次的模块Client不产生严重的代码耦合
3. 命令模式结合其他模式会更优秀：命令模式可以结合责任链模式，实现命令族解析任务；结合模板方法模式，则可以减少Command子类的膨胀问题。

### 15.3.2 命令模式的缺点

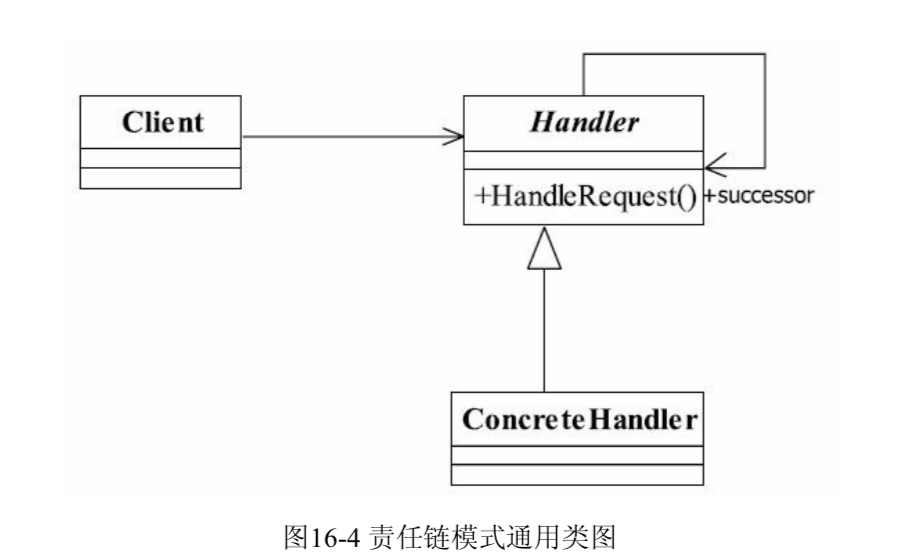
命令模式也是有缺点的，如果有N个命令，Command的子类就可不是几个，而是N个，这个类膨胀得非常大，这个就需要读者在项目中慎重考虑使用。

### 15.3.3 命令模式的使用场景

# 责任链模式

## 16.2 责任链模式的定义

使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象联成一条链，并沿着这条链传递该请求，知道有对象处理它为止。



## 16.3 责任链模式的应用

### 16.3.1 责任链模式的优点

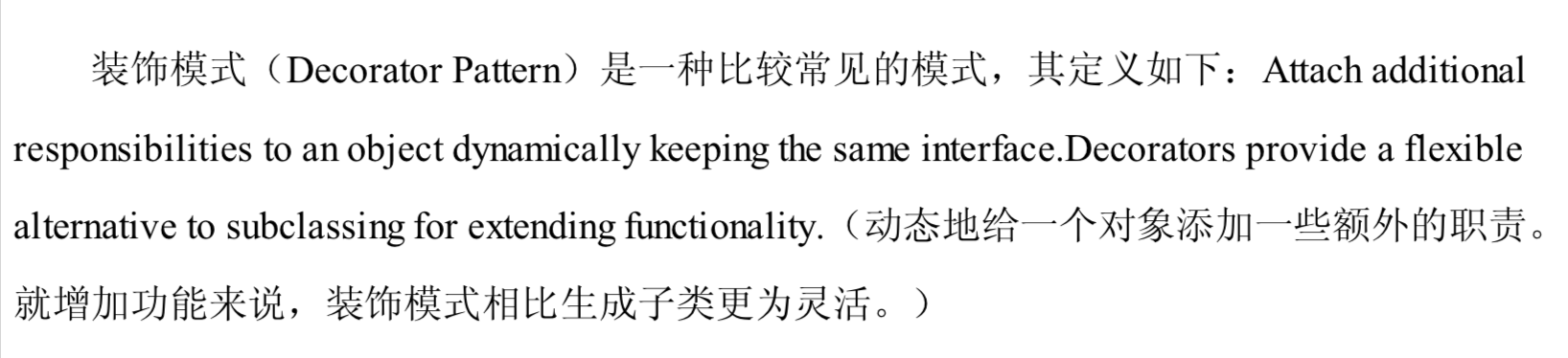
责任链模式非常显著的优点是将请求和处理分开。处理者可以不用知道请求的全貌。

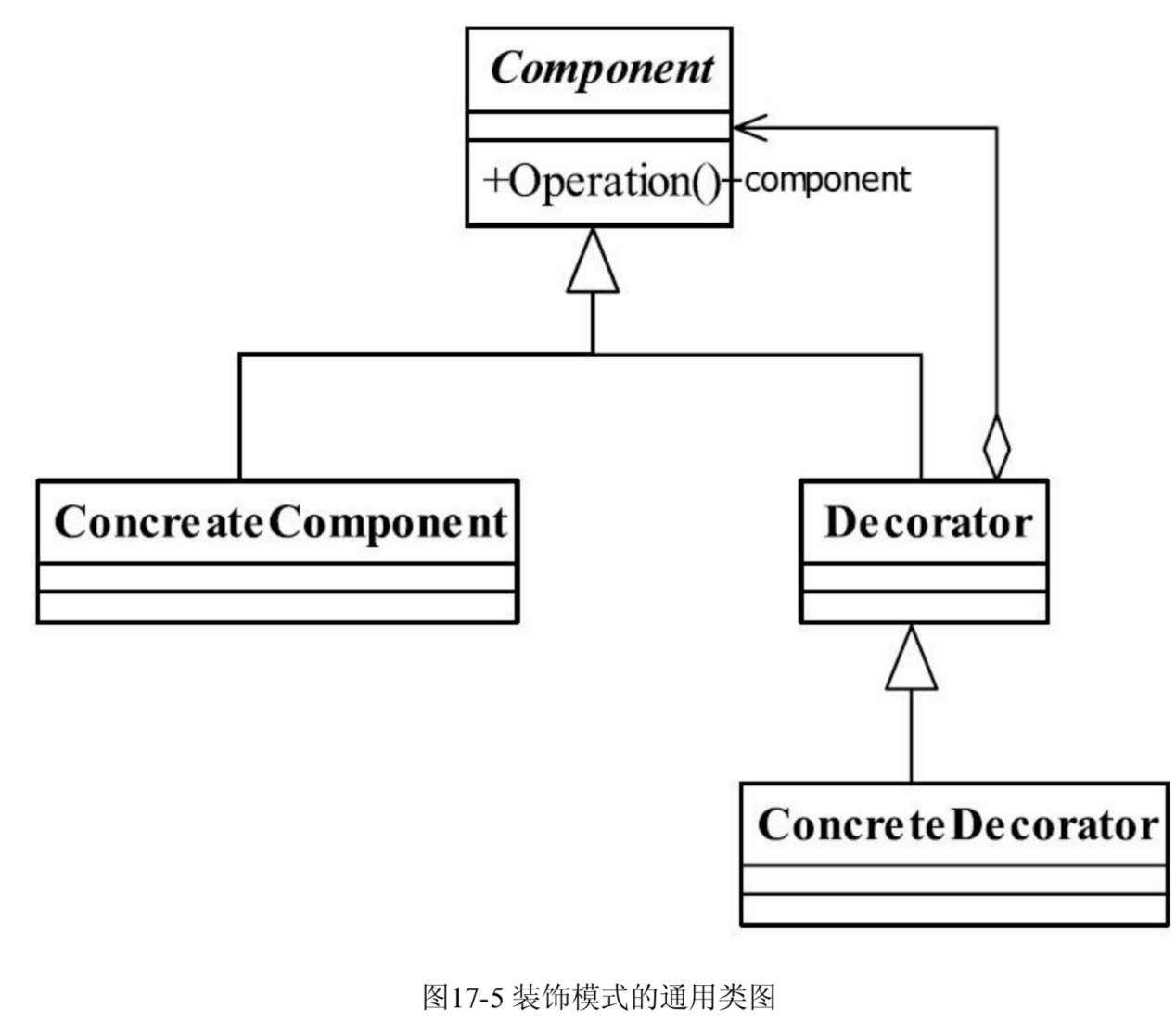
### 16.3.2 责任链模式的缺点

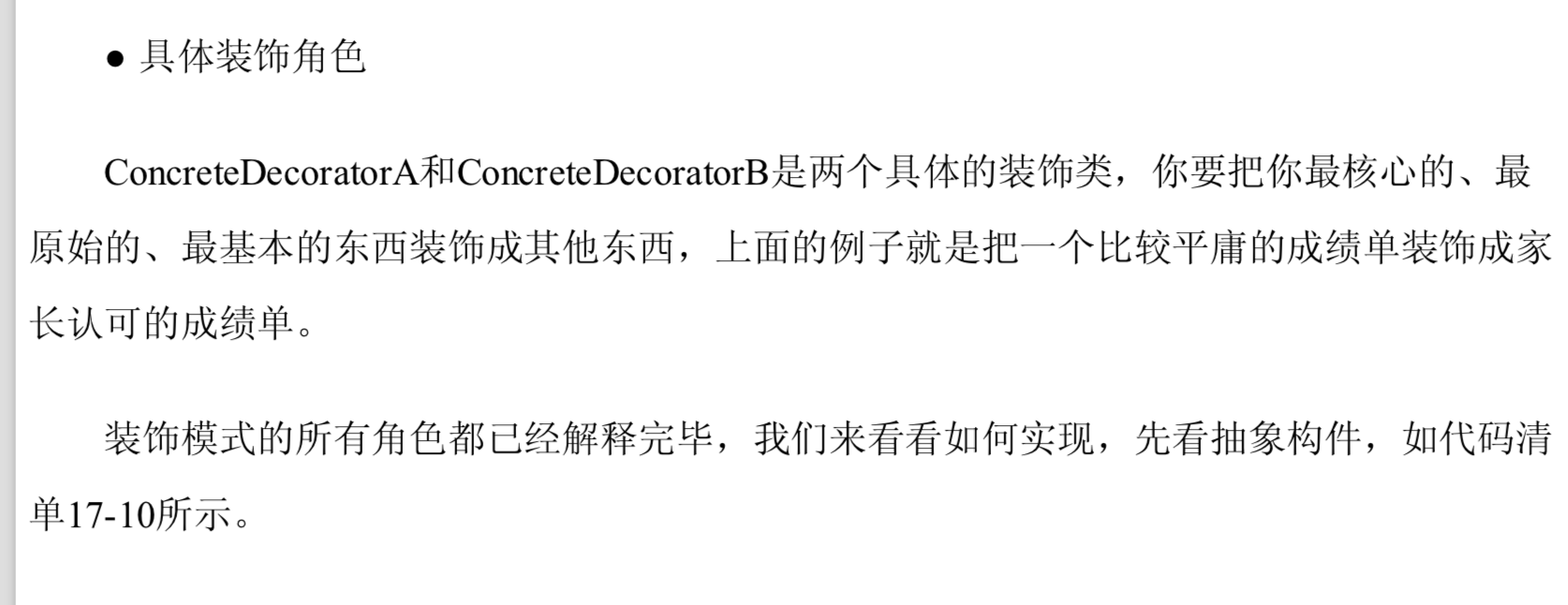
责任链两个显著的缺点：意识性能问题，每个请求都是从链头遍历到链尾。而是调试不很方便，特别是联调比较长，环节比较多的时候。

# 装饰者模式

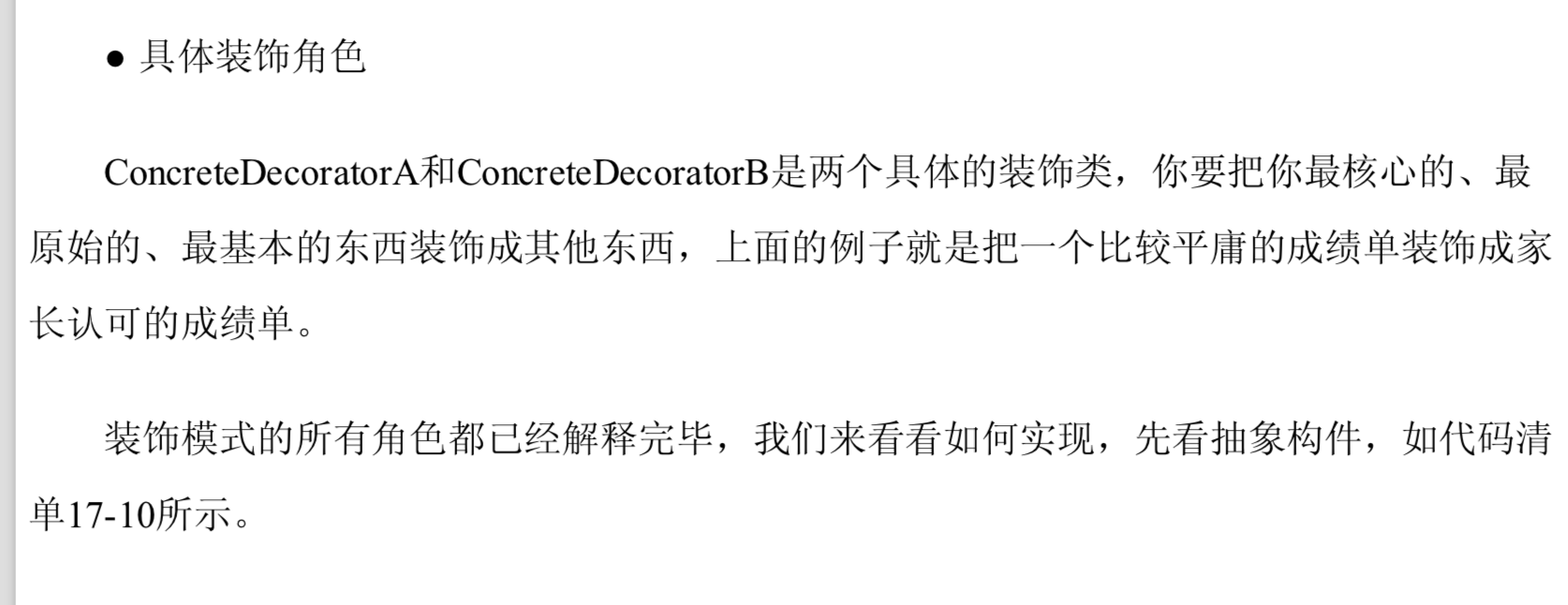
动态地给一个对象添加一些额外的职责

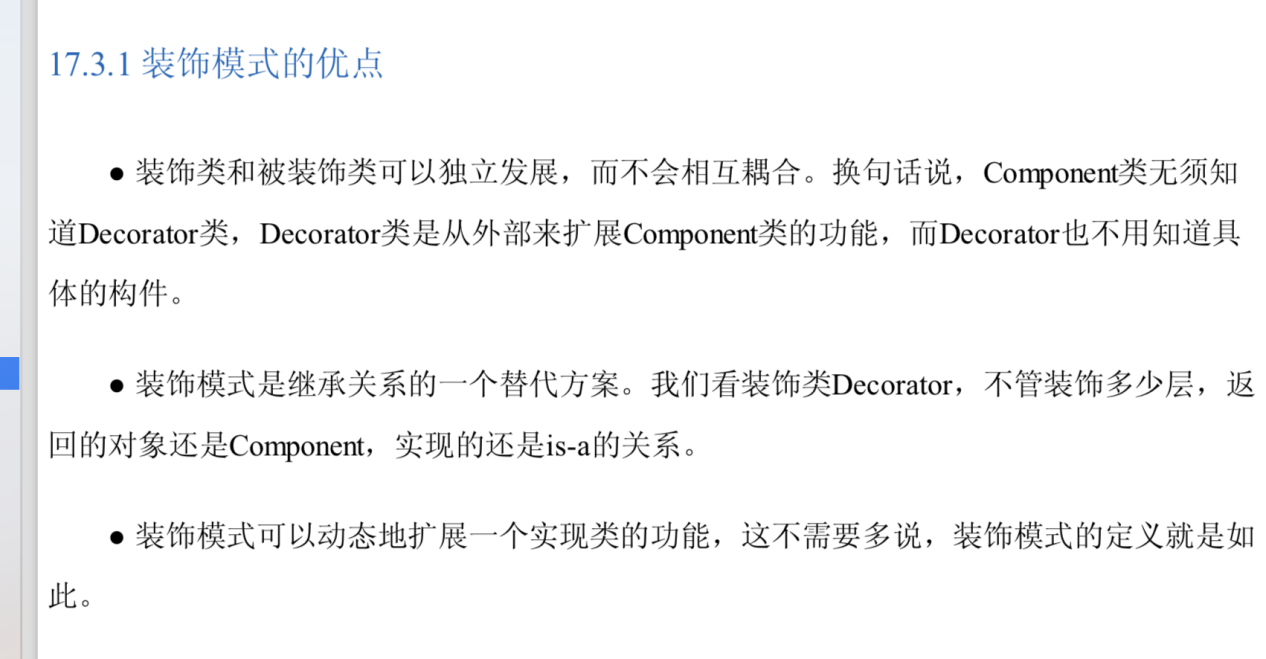




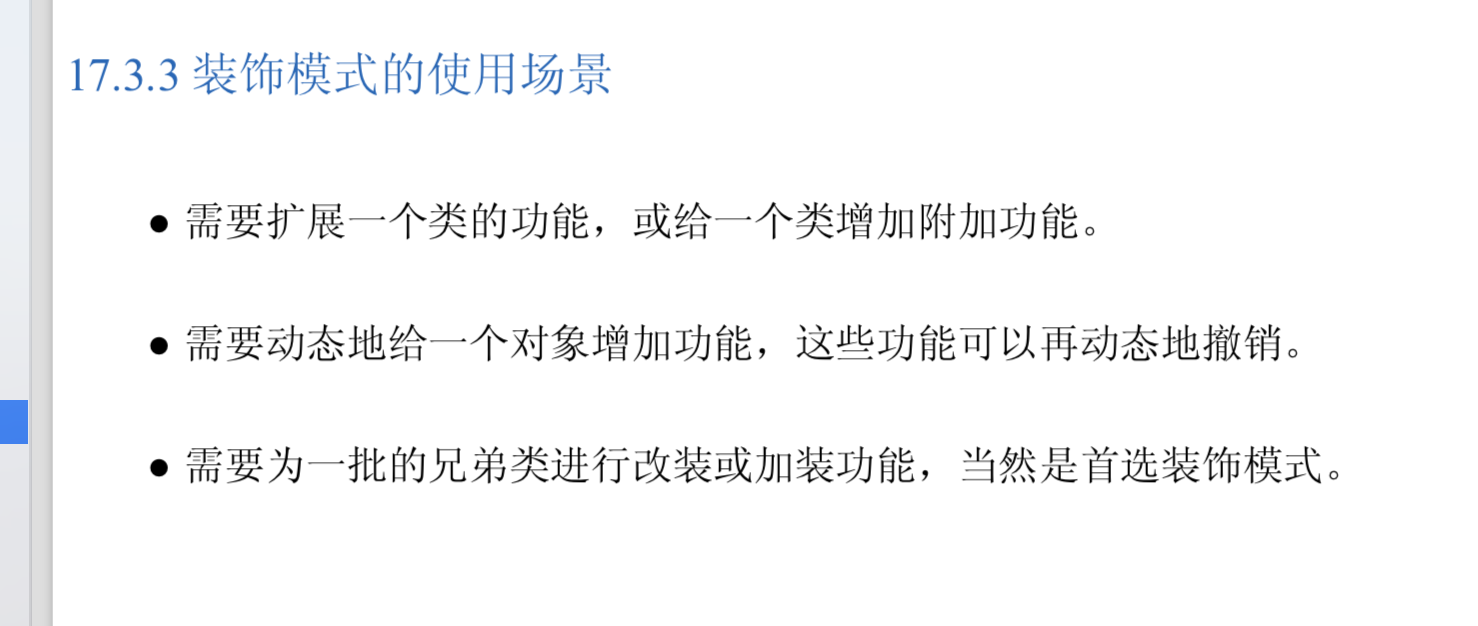








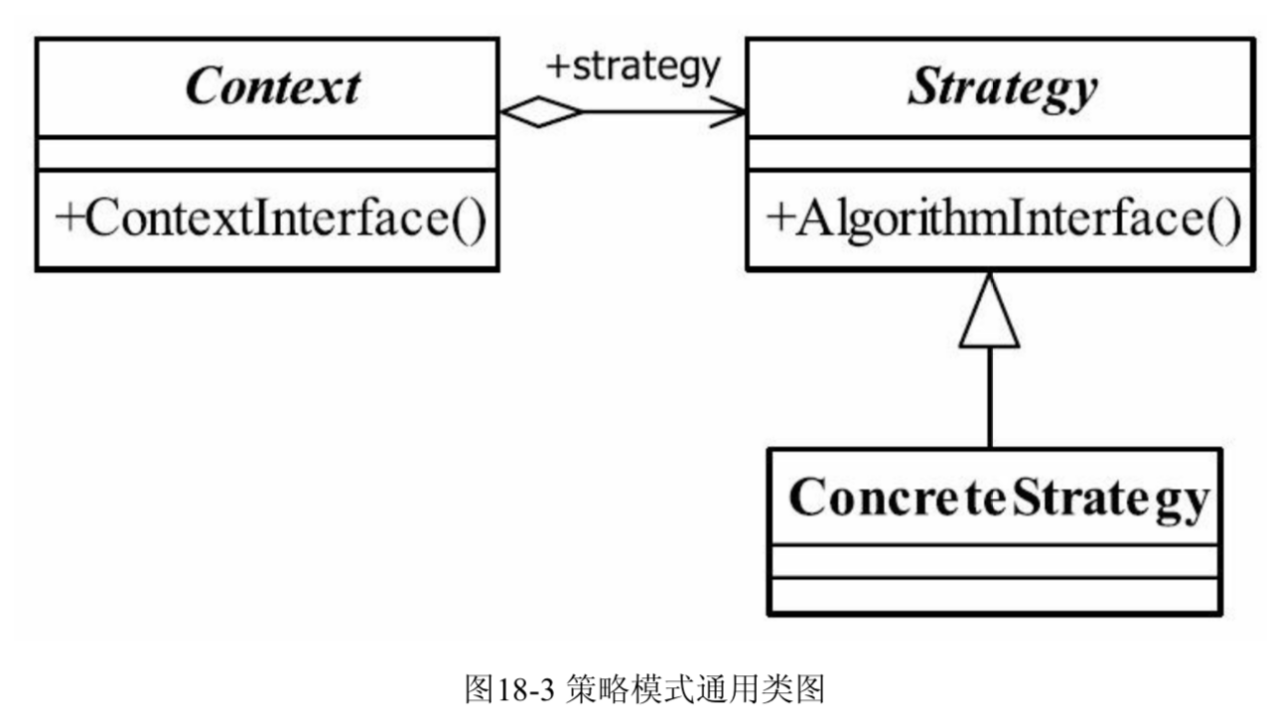




# 策略模式

## 18.2 策略模式的定义

策略模式是一种比较简单的模式，也叫做政策模式：定义一算法，将每个算法都封装起来，并且使他们之间可以互换。



1. Context
2. Strategy抽象策略角色
3. ConcreteStrategy具体策略角色：实现抽象策略中的操作，该类含有具体的算法

## 18.3 策略模式的应用

### 18.3.1 策略模式的优点

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

### 18.3.2 策略模式的缺点

1. 策略模式的缺点
2. 所有的策略类都需要对外暴露

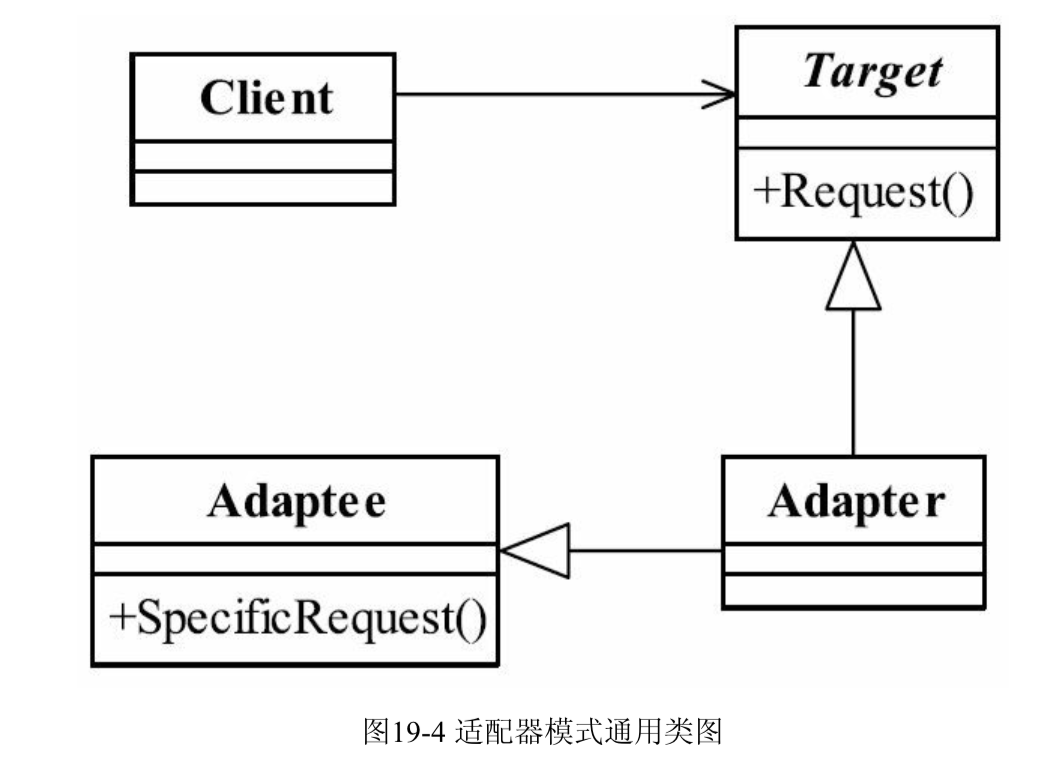
### 18.3.3 策略模式的使用场景

1. 多个类只有在算法或行为上稍有不同的场景
2. 算法需要自由切换的场景

# 适配器模式

## 19.2 适配器模式的定义

将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法再一起工作的两个类能够在一起工作



## 19.3 适配器模式的应用

### 19.3.1 适配器模式的优点

1. 适配器模式可以让两个没有任何关系的类在一起运行，只要适配器这个角色能够搞定他们就成。
2. 增加了类的透明性
3. 提高了类的复用度
4. 灵活性非常好

# 迭代器模式

## 20.2 迭代器模式的定义

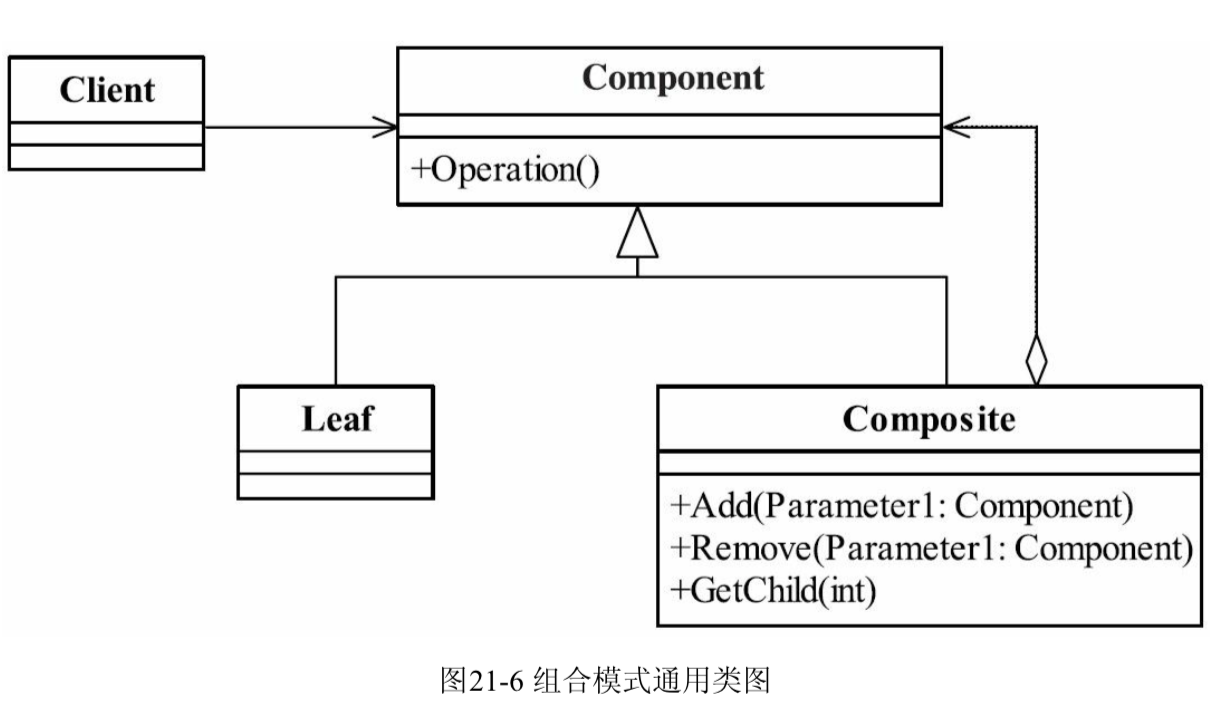
迭代器模式目前已经是一个没落的模式。它提供一种方法访问一个容器中各个元素，而又不需暴露该对象内部细节。

# 组合模式

## 21.2 组合模式的定义

组合模式(Composite Pattern)也叫合成模式，有时又叫部分-整体模式(Part-Whole)，主要是用来描述部分与整体的关系，其定义如下：

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



1.Component 抽象构件角色

定义参加组合对象的公有方法和属性，可以定义一些默认的行为或属性，比如我们例子中的getInfo就封装到了抽象类中。

1. Leaf叶子构件

叶子对象，其下再也没有其他的分支，也就是遍历的最小单位。

1. Composite树枝构件

树枝对象，它的作用是组合树枝节点和叶子节点形成一个属性结构

## 21.3 组合模式的应用

### 21.3.1 组合模式的优点

1. 高层模块的调用简单

一颗树形机构中的所有节点都是Component，局部和整体对调用者来说没有任何区别，也就是说，高层模块不必关系自己处理的是单个对象还是整个组合结构，简化了高层模块的代码。

1. 节点自由增加

使用了组合模式后，如果想增加一个树枝节点，树叶节点是不是都很容易，只要找到它的父节点就成，非常容易扩展，复合开闭原则，对以后的维护非常有利。

### 21.3.2 组合模式的缺点

组合模式有一个非常明显的缺点，在场景类中的定义，提到树叶和树枝使用时的定义了吗？直接使用了实现类！这在面向接口编程上时很不恰当的，与依赖倒置原则冲突，读者在使用的时候要考虑清湖，它限制了你接口的影响范围。

### 21.3.3 组合模式的使用场景

维护和展示部分-整体关系的场景，如属性菜单，文件和文件夹管理

从一个整体中能够独立出部分模块或功能的场景

### 21.3.4 组合模式的注意事项

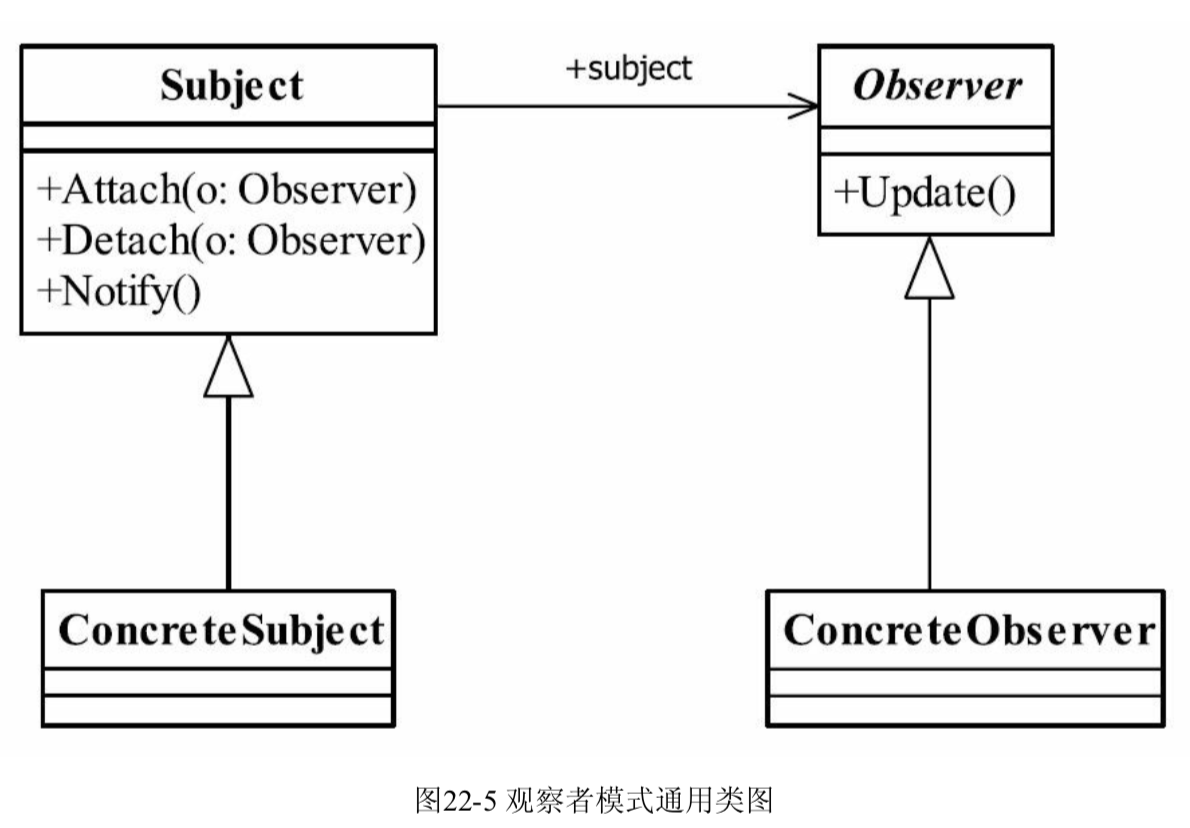
只要是属性结构，就要考虑使用组合模式，这个一定要记住，只要是要体现局部和整体的关系的时候，而且这种关系还可能比较深，考虑一下组合模式吧。

# 观察者模式

## 22.2 观察者模式的定义

观察者模式(Observer Pattern)也叫做发布订阅模式(Publish/subscribe)，它是一个在项目中经常使用的模式，其定义如下：

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新



1.Subject被观察者：

定义被观察者必须实现的职责，它必须能够动态地增加、取消观察者。它一般是抽象类或者实现类，仅仅完成作为被观察者必须实现的职责：管理观察者并通知观察者。

1. Observer 观察者

观察者接收到消息后，即进行update(更新方法)操作，对接收到的信息进行处理。

1. ConcreteSubject具体的被观察者

定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知

1. ConcreteObserver具体的观察者

每个观察在接收到消息后的处理反应是不同，各个观察者有自己的处理逻辑

## 22.3 观察者模式的应用

### 22.3.1 观察者模式的优点

1.观察者和被观察者之间是抽象耦合

2.建立一套触发机制

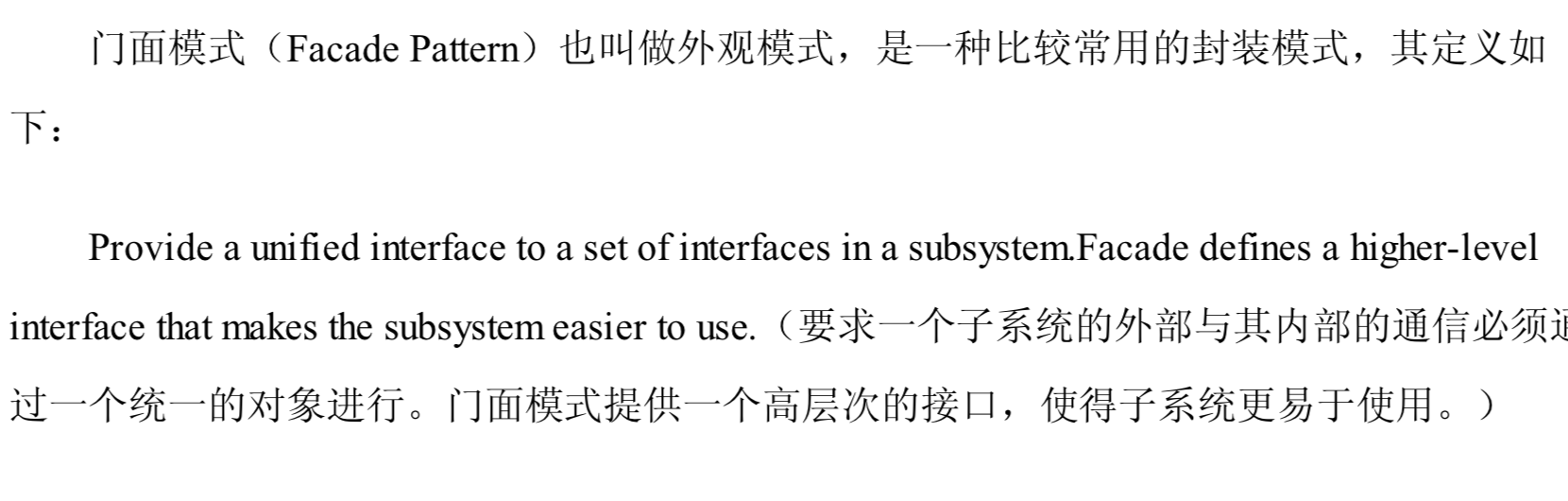
### 22.3.2 观察者模式的使用场景

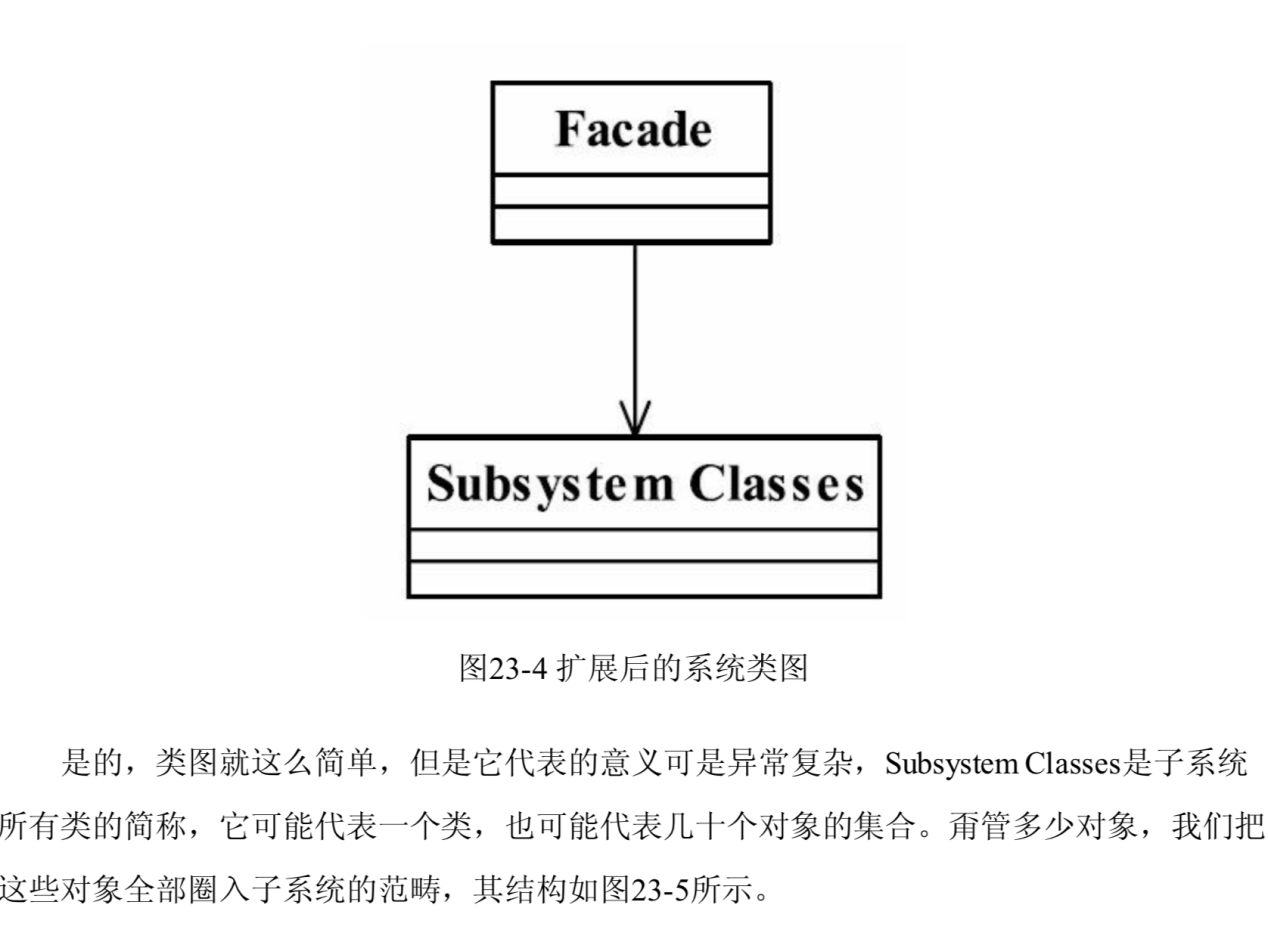
1. 关联行为场景。需要注意的是，关联行为是可查分的，而不是“组合”关系
2. 事件多级触发场景
3. 跨系统的消息交换场景，如消息对列的处理机制。

### 22.3.4 观察者模式的注意事项

1. 广播链的问题：在一个观察者模式中最多出现一个对象既是观察者也是被观察者。
2. 异步处理问题

# 门面模式

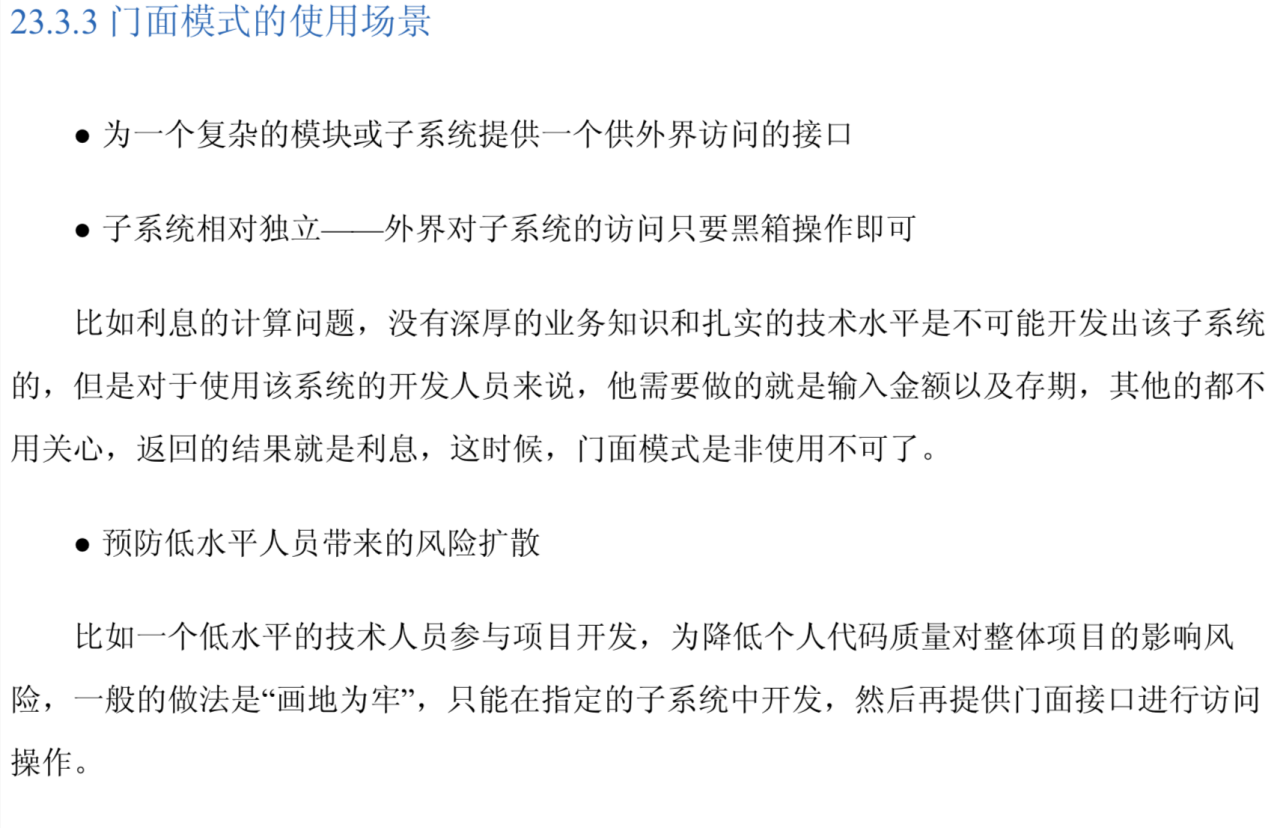




优点：

1. 减少系统的相互依赖
2. 提高了灵活性
3. 提高安全性

缺点：不符合开闭原则，对修改关闭，对扩展开放



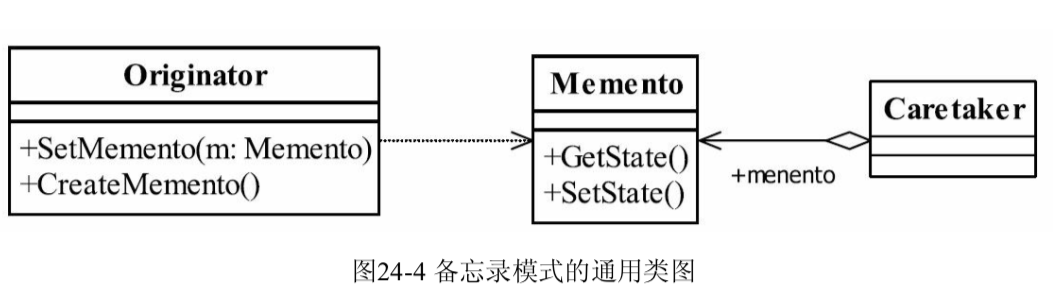
案例：tomcat的Request，Response就使用了门面模式，netty的ByteBuf对ByteBuffer等的封装，也是门面模式。其实传统的mvc架构中的。controller，service，dao，这三层也体现了门面模式，controller是对外暴露的服务，service组合dao层处理业务逻辑。

# 被忘录模式

## 24.2 备忘录模式的定义

备忘录模式(Memento Pattern)提供了一种弥补真实世界缺陷的方法，让”后悔药”在程序的世界中真实可行，其定义如下：

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，这样以后就可以将该对象恢复到原先保存的状态



1. Originator发起人角色

记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据。

1. Memento备忘录角色

负责存储Originator发起人对象的内部状态，在需要的时候提供发起人需要的内部状态。

1. Caretaker备忘录管理员角色

对备忘录进行管理、保存和提供备忘录

## 24.3 备忘录模式的应用

### 24.3.1 备忘录模式的使用场景

1. 需要保存和恢复数据的相关状态场景
2. 提供一个可回滚(rollback)的操作：比如Word中的CTRL+Z组合键，IE浏览器中的后退按钮，文件管理器上的backspace键等。
3. 数据库连接的事务管理就是用的备忘录模式，

### 24.3.2 备忘录模式的注意事项

1. 备忘录的生命周期

备忘录创建出来就要在最近的代码中使用，要注定管理它的生命周期，建立就要使用，不适用就要立刻删除其引用，等待垃圾回收器对它的回收处理

1. 备忘录的性能

不要在频繁建立备份的场景中使用备忘录模式(比如一个for循环中)，原因有二：一是控制不了备忘录建立的对象数量；而是大对象的建立是要消耗资源的，系统的新能需要考虑。

# 被忘录模式

# 被忘录模式

# 被忘录模式

# 享元模式

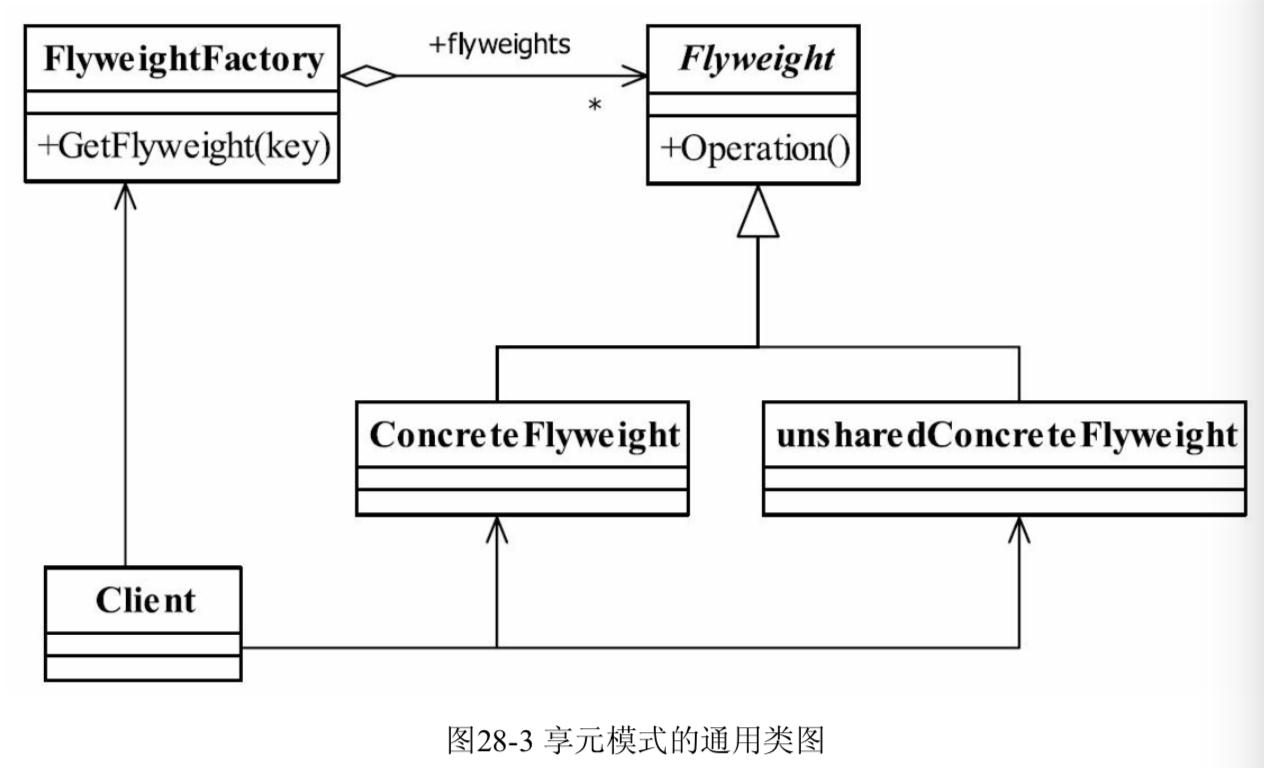
## 28.2 享元模式的定义

享元模式(Flyweight Pattern)是池技术的重要实现方式，其定义如下：(使用共享对象可有效地支持大量的细粒度的对象)

享元模式的定义为我们提出了两个要求：细粒度的对象和共享对象。

细粒度对象的信息分为两个部分：内部状态与外部状态。

1. 内部状态：内部状态是对象可共享出来的信息，存储在享元对象内部并且不会碎环境改变而改变。
2. 外部状态：外部状态是对象得以依赖的一个标记，是随环境改变而改变的、不可以共享的状态。



1. Flyweight——抽象享元角色

它简单地说就是一个产品的抽象类，同时定义出对象的外部状态和内部状态的接口或实现

1. ConcreteFlyweigth——具体享元角色

具体的一个产品类，实现抽象角色定义的业务。该角色中需要注意的是内部状态处理应该与环境无关。

1. unsharedConcreteFlyweigth——不可共享元角色
2. FlyweightFactory——享元工厂

就是构造一个池容器。

## 28.3 享元模式的应用

### 28.3.1 享元模式的应用

享元模式可以大大较少应用程序创建的对象，降低程序内存的占用，增强程序的性能，但它同时也提供了系统复杂性，需要分离出外部状态和内部状态，而且外部状态具有固化特性，不应该随内部状态改变而改变。

### 28.3.2 享元模式的使用场景

1. 系统中存在大量的相似对象。
2. 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份
3. 需要缓冲池的场景

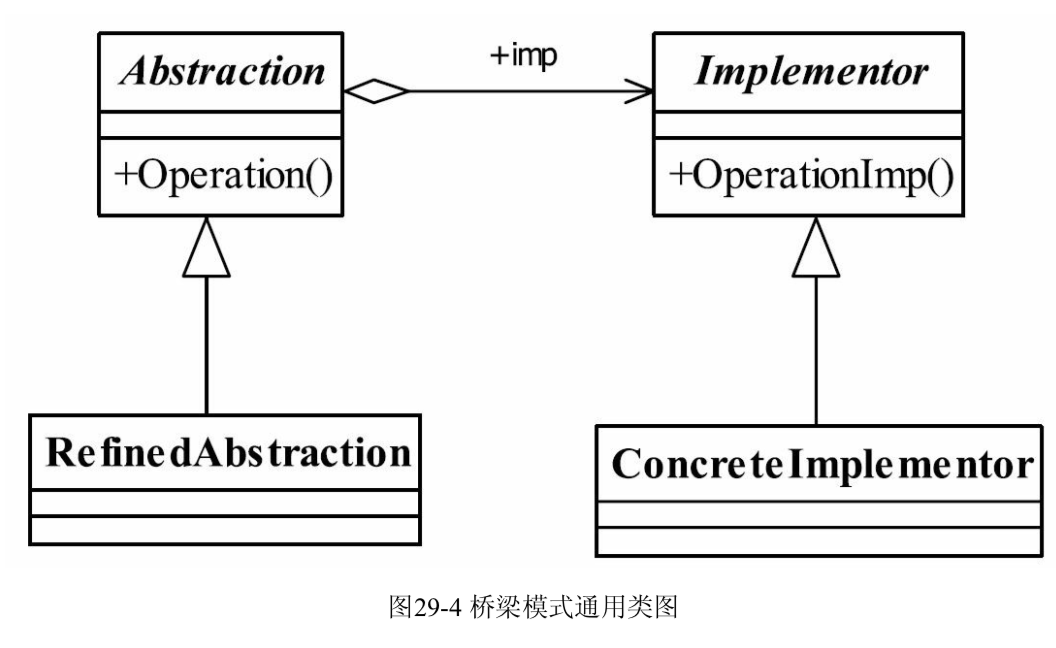
## 28.4 享元模式的扩展

# 桥梁模式

## 29.2 桥梁模式的定义

桥梁模式也叫桥接模式，是一个比较简单的模式，其定义如下：

将抽象和实现解耦，使得两者可以独立地变化。



1. Abstraction——抽象画角色

它的主要职责是定义出该角色的行为，同时保存一个对实现化角色的引用，该角色一般是抽象类

1. Implementor——实现化角色

它是接口或者抽象类，定义角色必须的行为和属性

1. RefinedAbstraction——修正抽象画角色

它引用实现化角色对抽象画角色进行修正

1. ConcreteImplementor——具体实现化角色

它实现接口或抽象类定义的方法和属性

## 29.3 桥梁模式的应用

### 29.3.1 优点

1. 抽象和实现分离
2. 优秀的扩充能力
3. 实现细节对客户透明