Java 设计模式

目录

1. 设计模式(超级详细)

内容简介

有感于设计模式在日常开发中的重要性，同时笔者也自觉对设计模式小有心得，故笔者*写二十三种设计模式的简单例子、
并整理二十三种设计模式的理论部分，综合汇总成这份 Java 设计模式（疯狂 J*va 联盟版），希望对大家有所帮助。

本份帮助文档主要是为了向读者介绍二十三种设计模式，包括模式的描述，适用性，模*的组成部分，并附带有简单的例
子和类*，目的是为了让读*了解二十三种*计模式，并能方便的查阅各种设计模*的用法及注意点。

所附的例子非常简单，慢慢的引导读者从浅到深了解设计模式，并能从中享受设计的乐趣。

由于每个人对设计*式的理解都不尽一致，因此，可能本文档的例子*有不恰当的地方，还望各位读者指出不恰当的地方。
欢迎登录疯狂 J*va 联盟进行技术交流，疯狂 Java 联盟的论坛宗旨是：

所有的技术发帖，均有回复。

疯狂 Java 联盟网址：http://www.crazyit.org

笔者简介

笔者曾师从李刚老师学习 Java，现居广州。对 Java 软件开发、各种 Java 开源技术都非常感兴趣，曾参与开发、主持*发过大
量 Java、Java EE 项目，对 Java、Java *E 项目有一定认识*见解。欢迎大家与笔者就 Java、Java EE 相*方面进行技术交流。

笔者现为疯狂 Jav*联盟的总版主（论坛 ID：杨恩雄），也希望通过该平台与大家分享 Java、Java EE 技术、*得。

本人邮箱：yangenxiong@163.com

声明

本文档编写、制作过程中得到了疯狂 Java 联盟、以及笔者学习工作过程大量朋友的支持，大家都抱着一个目的：为国内软件
软件开发事业作出绵薄贡献。

1.1 创建型模式

AbstractFactory (抽象工厂)
FactoryMethod (工厂方法)
Singleton (单态模式)
Builder (建造者模式)
Protot*pe * 原型模式)

1.1.1 工厂方法

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*延迟到其子类。

适用性

1. 当一个类不知道它所必须创建的对象的类的时候。

2. 当一个类希望由它的子类来指定它所创建的对象的时候。

3. 当*将创建对象的职责委托给多个帮助*类中的某一个，并且*希望将哪一个帮助子类是代理者这一信息局部化的时候。

参与者

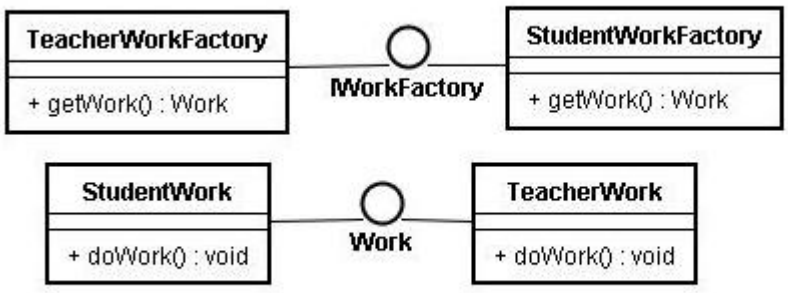
1. Product
   定义工厂方法所创建的对象的接口。

2. ConcreteProduct
   实现 Product 接口。

3. 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。

4. ConcreteCreator
   重定义工厂方法以返回一个 ConcreteProduct 实例。

类图



例子

***product**

```
public interface Work {

    void doWork();
}
```

ConcreteProduct

```
public class StudentWork implements Work {

    public void doWork() {
        System.out.println("学生做作业!");
    }
}

public class TeacherWork implements Work {

    public void doWork() {
        System.out.println("老师审批作业!");
    }
}
```

Creator

```
public interface IWorkFactory {

    Work getWork();
}
```

***ConcreteCreator**

```
public class StudentWorkFactory implements IWorkFactory {

    public Work getWork() {
       return new StudentWork();
    }
}
```

```
public class TeacherWorkFactory implements IWorkFactory {

    public Work getWork() {
        return new TeacherWork();
    }
}
```

Test

```
public class Test {

    public static void main(String[] args) {
        IWorkFactory studentWorkFactory = new StudentWorkFactory();
        studentWorkFactory.getWork().doWork();

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

result

```
学生做作业!
老师审批作业!
```

1.1.2 抽象工厂

提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

适用性

1.一个系统要独立于它的产品的创建、组合和表示时。

2.一个系统要由多个产品系列中的一个来配置时。

3.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。

4．当你提供一个产品类库，而只想显示它们*接口而不是实现时。

参与者

1.AbstractFactory
声明一个创建抽象产品对象的操作接口。

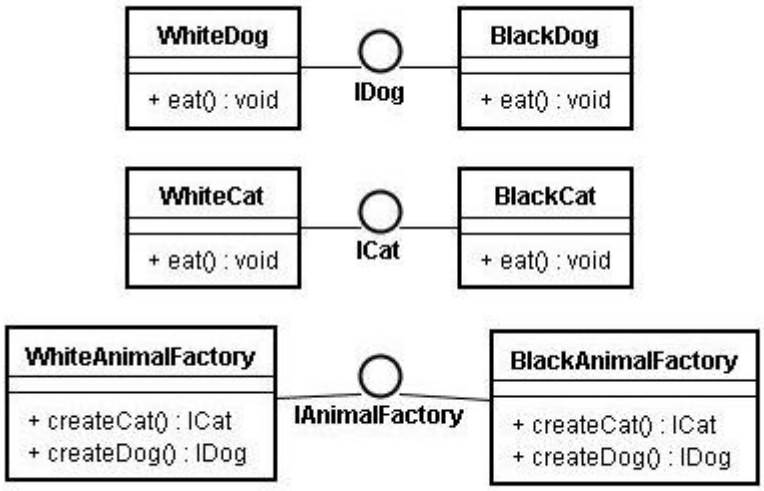
2.ConcreteFactory
实现创建具体产品对象的操作。

3.AbstractProduct
为一类产品对象声明一个接口。

4. 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。

5. Client
   仅使用由 AbstractFactory 和 AbstractProduc*类声明的接口

类图



例子

***bstractFactory**

```
public interface IAnimalFactory {

    ICat createCat();

    IDog createDog();
}
```

ConcreteFactory

```

public class BlackAnimalFactory implements IAnimalFactory
{

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

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

```
}

public class WhiteAnimalFactory implements IAnimalFactory
{
    public ICat createCat()
    {
        return new WhiteCat();
    }

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

AbstractProduct

```
public interface ICat {

    void eat();
}

public interface IDog {

    void eat();
}
```

Concreteproduct

```
public class Blackcat implements ICat {

    public void eat() {
        System.out.println("The black cat is eating!");
    }
}

public class WhiteCat implements ICat {

    public void eat() {
        Symtem.out.println("The write cat is eating! ");
    }
}

public class BlackDog implements IDog {

    public void eat() {
        System.out.println("The black dog is eating");
```

```
    }

}

public class WhiteDog implements IDog {

    public void eat() {
        System.out.println("The white dog is eating!");
    }

}
```

Client

```
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();
}
```

result

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

1.1.3 建造者模式

将一个复杂对象的构造与它的表示分离，使用同样的构建过程可以创建不同的表示。

适用性

1.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。

2.当构造过程必须允许被构造的对象有不同的表示时。

参与者

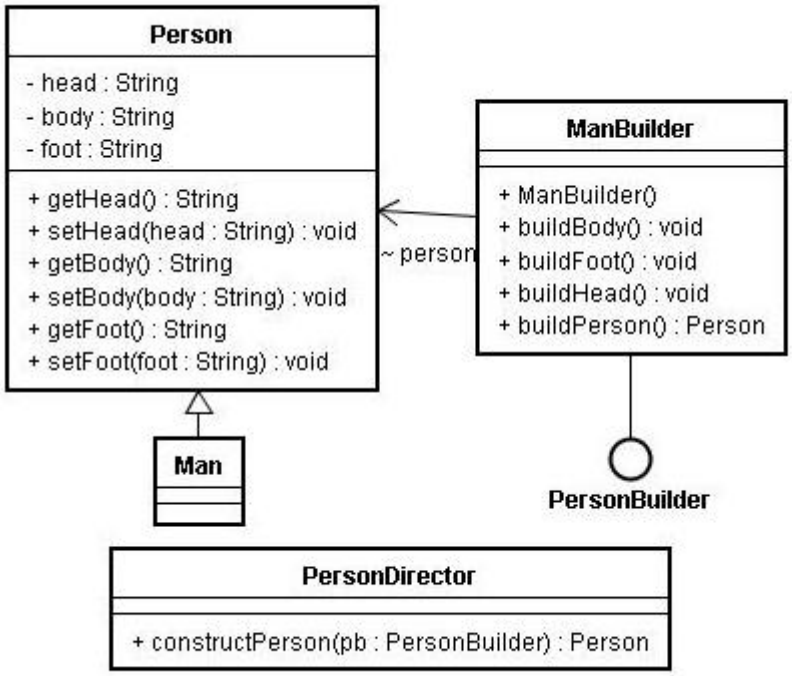
1. Builder
   为创建一个 Product 对象的各个部件指定抽象接口。

2. 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。

3. Director
   构造一个使用 Builder 接口的对象。

4. 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。

类图



例子

Builder

```
public interface PersonBuilder {

    void buildHead();

    void buildBody();
```

```
    void buildFoot();

    Person buildPerson();
}
```

ConcreteBuilder

```
public class ManBuilder implements PersonBuilder {

    Person person;

    public ManBuilder() {
        person = new Man();//下面有 Man 类
    }

    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 //构造一个使用 Builder 接口的对象。

```
public class PersonDirector {

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

Product

```
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;
    }
}

public class Man extends Person {

}
```

Test

```
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
```

1.1.4 单态模式

保证一个类仅有一个实例，仅提供一个访问它的全局访问点。

适用性

1.当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。

2.当这个唯一实例应该是通过**子类化**可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

参与者

Singleton

定义一个 Instance 操作，允许客户访问它的唯一实例。Instance 是一个类操作。

可能负*创建它自己的唯一实例。

类图

例子

Singleton

```
public class Singleton
{

    private static Singleton sing;

    private Singleton()
    {

    }
```

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

Test

```
public class Test {

    public static void main(string[] args) {
        Singleton sing = Singleton.getInstance();
        Singleton sing2 = Singleton. getInstance ();
        System.out.println(sing);
        System.out.println (sing2);
    }
}
```

result

```
singleton.Singleton@1c78e57
singleton.Singleton@1c78e57
```

1.1.5 原型模式

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。

适用性

1.当一个系统应该独立于它的产品创建、构成和表示时。

2.当要实例化的类是在运行时刻指定时，例如，通过动态装载。

3.为了避免创建一个与产品类层次平行的工厂*层次时。

4.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。

参与者

1. Prototype
   声明一个克隆自身的接口。

2. ConcretePrototype
   实现一个克隆自身的操作。

3. Client
   让一个原型克隆自身从而创建一个新的对象。

类图

例子

Prototype

```
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

```
publ*c class ConcretePrototype extend* Prototype {

    public ConcretePrototype(String name) {
        setName(name);
    }
}
```

Client

```
public clas* Test {

    public static void main(String[] args) {
        Prototype pro = new ConcretePrototy*e("prototype");
        Prototype pro2 = (Prototype)pro.clone();
        *ystem.out.println(pro.getName()*;
        System.out.println(pro2.getName());
    }
}
```

result

```
prototype
prototype
```

1.2 结构型模式

Adapter * 适配器模式 *

Bridge (桥接模*)

Composite (组合模式)

Decorator (装*模式)

Facade (外观模式)

Flyweight (享元模式)

Proxy (代理模式)

1.2.1 适配器模式

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*兼容而不能一起工作的那*类可以一起工作。

适用性

1.你想使*一个已经存在的类，而它的接口不符合你的需求。

2.你想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*接口可能不一定兼容的类）协同工作。

3.（仅适用于对象 Adapter）你想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。

参与者

1.Target
定义 Client 使用的与特定领域相关的接口。

2.Client
与符合 Target 接口的对象协同。

3.Adaptee
定义一个已经存在的接口，这个接口需要适配。

4.Adapter
对 Adaptee 的接口与 Target 接口进行适配

类图

例子

Target

```
public interface Target

    void adapteeMethod();

    void adapterMethod();
}
```

Adaptee

```

```

```
public class Adaptee {

    public void adapteeMethod() {
        Syste*.out.p*intln("Adaptee method!");
    }
}
```

Adapter

```
public clas* Adapter implement* Target {

    private Adap*ee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adapte* = adaptee;
    }

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

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

Client

```
public cla*s Test {

    public stati* void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        tar*et.adapteeMethod();

        target.adapterM*thod();
    }
}
```

result

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

1.2.2 桥接模式

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。

适用性

1.你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*选择或者切换。

2.类的抽象以及它的实现都应该可以通*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。

3.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。

4.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*一种类层次结构说明你必须将一个对象分解成两个部分。

5.*想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*这一点。

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。

*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

*现 Implementor 接口并定义它的具体实现。

类图

例子

Abstr*ction

```
public abstract class Person
{
    private Clothing clothing;

    private String type;

    public Clothing getClothing()
    {
        return clothing;
    }
    public void setClothing()
    {
        this.clothing = ClothingFactory.getClothing();
    }
    public void setType(String type)
    {
        t*is.type = type;
    }
    public String getType()
    {
        return this.ty*e;
    }
    public abstract void dress();
}
```

RefinedAbstraction

```
public class Man extends Person
```

```
{
    public Man()
    {
        setType("男人");
    }

    public void dress()
    {
        Clothing clothing = get*lothing();
        clothing.personDressCloth(this);
    }
}

public class Lady extends Person
{
    public Lady()
    {
        setTyp*("女人");
    }
    public void dress()
    {
        Cloth*ng clothing = getClothing();
        c*othing.personDressCloth(this);
    }
}
```

Implemento*

```
public abstract class Clothing
{
    public abstract void personDressC*oth(*erson person);
}
```

ConcreteImplemento*

```
public class jacket extends Clothing
{

    public void personDressCloth(Person person)
    {
        System.out.println(person.getType() + "穿马甲");
    }
}

public class Trouser extends Clothing {

    public void personDressCloth(Person person) {
```

```
        System.ou*.println(Person.getType() + "穿裤子");
    }
}
```

Test

```
public class Te*t {

    public static void main(String[] args) {

        Person man = new Man();

        Person lady = new Lady();

        Clothing jacket = new Ja*ket();

        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
```

1.2.3 组合模式

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*的使用具有一致性。"

适用性

1.你想表示对象的部分-整体层次结构。

2.你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。

参与者

1. 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*的情况下实现它。

2. 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。

3. Compoite
   定义有子部件的*些部件的行为。
   存储子部件。
   在 Component 接口中实现与子部件有*的操作。

4. Client
   通过 Component 接*操纵组合部件的对象。

类图

例子

Component

```
p*blic 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 employers;

    public void printInfo()
    {
        System.out.println(name);
    }
    *ublic List getE*ployers()
    {
        return this.employers;
    }
}
```

Leaf

```
public class Programmer extends Employer
{
    public Programmer(String name)
    {
        setNam*(name);
        employers = null;//程序员, 表示没有下属了
    }
    public v*id add(Employer employer)
    {

    }

    public void delete(Employer employer) {

    }
}

public class Pro*ectAssistant extends Employer {

    public ProjectAss*stant(String name) {
        setName(name);
        employers = *ull;//项目助理, 表示没有下属了
    }

    public void add(Employer employer) {
```

```
    }

    public void delet*(Employer employer) {

    }
}
```

Composite

```

public class ProjectManager extends Employer {

    public ProjectManager(String name) {
        setName(name);
        employers = new A*rayList();
    }

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

    public void delete(Emplo*er employer) {
        employers.remove(employer);
    }
}
```

Clie*t

```

publ*c class Test {

    public st*tic void main(String[] args) {
        Employer pm = new ProjectManager("项目经理");
        Emplo*er pa = new ProjectAssistant("项目助理");
        Employer progra*mer1 = 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);//*项目经理*加程序员

        List ems = pm.getEm*loyers();
        for (Employer em : ems) {
            System.out.println(em.getNam*());
        }
    *
}
```

result

```
项目助理
程序员二
```

1.2.4 装饰模式

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*相比生成子类更为*活。

适用性

1. 在不影响其他*象的情况下，以动态、透明的方式给单个对象添加职责。

2. 处理那些可以撤消的职责。

3. 当不能采用生成子类的方法进行扩充时。

参与者

1. Component
   定义一个对象接口，可以给这些对象动态地添加职责。

2. ConcreteComponent
   定义一个对象，可以给这个对象添加一些职责。

3. 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。

4. ConcreteDecorator
   向组件添加职责。

类图

例子

Component

```
public interface Person {

    void eat();
}
```

ConcreteComponent

```
public class Man implements Person {
```

```
        public void eat() {
                System.out.println("男人在吃");
        }
}
```

Decorator

```
public abstract class Decorator implements Person {

    protected Person person;

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

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

ConcreteDec*rator

```
public class ManDecoratorA extends Decorator {

    public void eat() {
        super.eat();
        reEat();
        Sy*tem.out.println("ManDecoratorA 类");
    }

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

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

Test

```
public class Test {
```

```
    public st*tic void main(Strin*[] args) {
        Man man = new Man();
        ManDecoratorA md1 = new ManDecoratorA();
        ManDecoratorB md2 = n*w ManDecoratorB();

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

result

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

1.2.5 外观模式

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*子系统更加容易使用。

适用性

1.当你要为一个*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*些不需要定制子系统的用户带来一些使用上的困难。
Fa*ade 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*，而那些需
要更多的可定制性的用户可以越过 facade 层。

2.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。

3.当你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。

参与者

1. 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。

2. 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*acade 的指针。

类图

例子

Facade

```
publi* class Facade {

    ServiceA s*;

    ServiceB sb;

    ServiceC sc;

    public Facade() {
        sa = new S*rviceAImpl();
        sb = new *erviceBImpl();
        sc = new ServiceCImpl();
    }

    public void methodA() {
```

```
        sa.methodA();
        sb.methodB();
    }

    publi* void methodB() {
        s*.methodB();
        sc.methodC();
    }

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

Subsystemclasse*

```
public *lass ServiceAImpl implements ServiceA {

    public void methodA() {
        System.out.println("这是服务 A");
    }
}

public class ServiceBImpl implements ServiceB {

    public void methodB() {
        System.out.println("这是服务 B");
    *
}

public class ServiceCImpl implements ServiceC {

    public void methodC() {
        System.out.println("这是服*C");
    }
}
```

Test

```
public class Test {

    public static voi* main(String[] args) {
       ServiceA sa = new ServiceAImpl();
       Ser*iceB sb = new ServiceBImpl();
```

```
        sa.metho*A();
        sb.methodB();

        System.out.println("========");
        //facade
        Facade facade = new Facade();
        facade.methodA();
        facade.methodB();
    }
}
```

resu*t

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

1.2.6 享元模式

运用共享技术有效地支持大量细粒度的对象。

适用性

当都具备下列情况时，使用 Flyweight 模式：

1.一个应用程序使用了大量的对象。

2.完全由于使用大量的对象，造成很大的存储开销。

3.对象的大多数状态都可变为外部状态。

4.如果删除对象的外部状态，那么可以由相对较少的共享对象取代很多组对象。

5.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。

参与者

1.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。

2.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。

3.UnsharedConcreteFlyweigh*

并非所有的 Flywe*ght 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*。

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。

4.Flyweigh*Factory

创建并管理 flywe*ght 对象。

确保合理地共享 flyweight。当用户请求一个 flywei*ht 时，Fl*weightFactory 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）。

例子

Flyweight

```
public interface Flyweight {

    void action(int arg);
}
```

ConcreteFlyweight

```
public class FlyweightImpl implements Flyweight {

    public void action(int arg) {
        // T*DO Auto-genera*ed method stub
        System.out.println(“参数值: “ + arg);
    }
}
```

FlyweightFactory

```
public class FlyweightFactory {

    private static Map flyweights = new HashMap();

    public FlyweightF*ctory(String arg) {
        flyweights.put(arg, new FlyweightImpl());
    }

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

    public static int g*tSize() {
        retu*n flyweights.size();
    }
}
```

Test

```
public class Test {

    public static v*id main(String[] args) {
        // TODO Auto-generated method stub
        Flyweight fly1 = Flyw*ightFact*ry.getFlyweight(*a");
        fly1.action(1);

        Flyweight fly* = FlyweightFactory.getF*yweight("a");
```

```
        System.out.println(fly1 == fly2);

        Flyweight fl*3 = FlyweightFactory.getFlywei*ht("b");
        fly3.action(2);

        Flyweight fly4 = Flyweigh*Factory.getF*yweight("c");
        fly4.action(3);

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

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

result

```
参数值: 1
true
参数值: 2
*数值: 3
参数值: 4
4
```

1.2.7 代理模式

为其他对象提供一种代理以控制对这个对象的访问。

适用性

1.远程代理（RemoteProxy）为一个对象在不同的地址空间提供局部代表。

2.虚*理（VirtualProxy）根据需*创建开销很大的对象。

3.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。

参与者

1.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。

*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*取，并可能负责创建和删除它。
其他功能依赖于*理的类型：

2.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。

3.VirtualProxy 可以缓存实体的附加信息，以便延迟对它的访问。

4.ProtectionProxy 检查调用者是*具有实现一个请求所必需的访问权限。

5.Subject
定义 RealSubject 和 Proxy 的共用接口，这样就在任何使用 RealSubject 的地方都*以使用 Proxy。

6.RealSubject
*义 Proxy 所代表的实体。

例子

Subject

```
public interface Object {

    void action();
}
```

Proxy

```
public class ProxyObject implements Object {

    Object obj;

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

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

RealSubject

```
public class ObjectImpl implements Object {

    public void action() {
        System.out.println("========");
        System.out.println("========");
        System.out.pr*ntln("这是被代理的类");
        System.out.println("========");
        System.out.println("========");
    }
}
```

Test

```
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
```

备注：静态方法、成员变量调用声明类型的方法，实例方法调用对象方法。

1.3 行为型模式

Chain of Responsibility (责任链模式)

Command (命令模式)

Interpreter (解释器模式)

Iterator (迭代器*式)

Mediator (中介者模式)

Memento (备忘录模式)

Observer (观察者模式)

State (状*模式)

Strategy (策略模式)

TemplateMethod (模板方法)

Visitor (访问者模式)

1.3.1 责任链模式

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*链，
并*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.

适用性

1.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。

2.你*在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

3.可处理一个请求的对象集合应被动态指定。

参与者

1.Handler
定义一个处理请求的接口。
（可选）实现后继链。

2.ConcreteHandler
处理它所负责的请*。
可访问它的后继者。
如果可处理该*求，就处理*；否则将该请求转发给它的后继者。

3.Client
向链上的具体处理者(ConcreteHandler)对象提交请求。

类图

例子

Handler

```
public interface RequestHandle {

    void handleRequest(R*quest request);
}
```

ConcreteHandler

```
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*");
    }
}

public class PMRequestHandle implements RequestHandle {

    Req*estHandle rh;

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

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

public class TLRequestHandle implements RequestHandle {

    RequestHandle rh;

    public TLRequestHandle(RequestHand*e rh) {
        this.rh = rh;
    }

    public void handleRequest(Request request) {
        if (request instanceof LeaveRe*uest) {
```

```
            System.ou*.println("要请假, 项目组长审批!");
        } else {
            rh.handleRequest(request);
        }
    }
}
```

Client

```
public *lass Test {


    public static v*id main(String[] args) {
        RequestHandle hr = new HRRequestHandle();
        Requ*stHandle pm = new PMRequestHandle(hr);
        RequestHandle tl = new TLRequestHandle(pm);

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

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

        System.out.println("========");
        //项目经理上理辞职请求
        requ*st = ne* Dimissio*Request();
        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
```

1.3.2 命令模式

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*作。

适用性

1. 抽象出待执行的动作以参数化某对象。

2. 在不同的时刻指定、排列和执行请求。

3. 支持取消操作。

4. 支持修改日志，这样当系统崩溃时，这*修改可以被重做一遍。

5. 用构建在原语操作上的高层操作构造一个系统。

参与者

1. Comma*d
   声明执行操作的接口。

2. 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。

3. Client
   创建一个具体命令对象并设定它的接收者。

4. Invoker
   要求该命令执行这个请求。

5. Recei*er
   知道如何实*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。

类图

例子

Command

```
public abstract class Command {

    protectet Receiver receiver;

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

```
    public abstract void execute();
}
```

ConcreteCommand

```
public class CommandImpl extends Command {

    public CommandImpl(Receiver receiver) {
        super(receiver);
    }

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

Invoker

```
public class Invoker {

    private Command command;

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

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

Receiver

```
public class Receiver {

    public void receiver() {
        System.out.println("This is Receive class!");
    }
}
```

Test

```
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!
```

1.3.3 解释器模式

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。

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：

1.该文法简单对于复杂的文法,文法的*层次变得庞大而无法管理。

2.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。

参与者

1.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。

2.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。

3.NonterminalExpression(非终结符表达式)
为文法中的非终结符实现解释(Interpret)操作。

4.Context（上下文）
包含解释器之外的一些全局信息。

5.Client（客户）
构建(或被给定)表示该文法定义的语言中*个特定的句子的抽象*法树。
该抽象语法树由 NonterminalExpression 和 TerminalExpression 的实例装配而成。
调用解*操作。

类图

例子

AbstractExpression

```
public abstract class Expression {

    abstract void interpret(Context ctx);
}
```

Expression

```
public class AdvanceExpression extends Expression {

    void interpret(Context ctx) {
        System.out.println("这是高级解析器!");
    }
}

public class SimpleExpression extends Expression {

    void interpret(context ctx) {
        System.out.pri*tln("这是普通解析器!");
    }
}
```

Context

```
public class Context {

    private String content;

    Private List list = new ArrayList();

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return this.content;
    }

    public void add(Expression eps) {
        list.add(eps);
    }

    public List getList() {
```

```
        return list;
    }
}
```

Test

```
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);
        }
    }
}
```

res*lt

```
*是普通解析器!
这是高级解析器!
*是普通解析器!
```

1.3.4 迭代器模式

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。

适用性

1.访问一个聚合对象的内容而无需暴露它的内部表示。

2.支持对聚合对象的多种遍历。

3.为遍历不同的聚合结构提供一*统一的接口(即,支持多态迭代)。

参与者

1.Iterator
迭代器定义访问和遍历元素的接口。

2.ConcreteIterator
具*迭代器实现迭代器接口。

对该聚合遍历时跟踪当前位置。

3. 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.

类图

备注：客户程序要先得到具体容器角色，然后再通过具体容器角色得到具体迭代器角色

例子

Iterator

```
public interface Iterator {

    Object next();

    void first();

    void last();

    boolean hasNext();
}
```

ConcreteIterator

```
public class IteratorImpl implements Iterator {

    private List list;//抽象容器

    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

```
public interface List {

    Iterator iterator();//抽迭代倒器

    Object get(int index);

    int getSize();

    void add(Object obj);
}
```

ConcreteAggregate

```
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 O*ject get(int index) {
        return list[index];
    }

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

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

Test

```
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()) {
            S*stem.out.println(it.next());
        }

        Syst*m.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
```

1.3.5 中介者模式

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。

适用性

1.一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。

2.一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*该对象。

3.想定制一个分布在多个类中的行为，*又不想生成太多的子类。

参与者

1.Mediator
中介者定义一个接口用于与各同事（Colleague）对象通信。

2.ConcreteMediator
具*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。

3.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

类图

例子

Mediator

```
public abstract class Mediator {

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

ConcreteMediator

```
public class ConcreteMediator extends Mediator {

    private ColleagueA ca;
```

```
    private ColleagueB cb;

    public ConcreteMediator() {
        ca = new ColleagueA();
        cb = new Col*eagueB();
    }

    public void no*ice(String content) {
        if (co*tent.equals("boss")) {
            //老板来了，通知员工 A
            Ca.action();
        }
        if (content.equals("client")) {
            //客户来了，*知前台 B
            cb.action();
        }
    }
}
```

Colleagueclass

```
public class ColleagueA extends Colleague {


    public void action(* {
        System.out.println("普通员工努力工作");
    }
}

public class ColleagueB extends Colleague {

    public void action() {
        System.out.println("前台注意了!");
    }
}
```

Test

```
public class Test {

    public static void main(String[] args) {
        Mediator med = new Concr*teMediator();
        */老板来了
        med.notice("boss");
```

```
        //客户来*
        med.n*tice("client");
    }
}
```

result

```
普通员工努力工作
前台注意了!
```

1.3.6 备忘录模式

在不破坏封装性*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。

适用性

1.必须*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。

2.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。

参与者

1.Memento
备忘录存储原发器对象的内部状态。

2.Originator
原发器创建一个备忘录,用以记录当前时刻*的内部状态。
使用备忘录恢复内部状态.

3.Caretaker
负责保存好备忘录。
不能对备忘录的内*进行操作或检查。

类图

例子

Memento

```
public class Memento {

    private String state;
```

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

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.stat* = s*ate;
    }
}
```

Or*ginator

```
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 meme*to) {
        state = memento.ge*State();
    }

    p*blic void showState(){
        System.out.println(state);
    }
}
```

Caretaker

```
public class Caretaker {

    p*ivate Memento memento;
```

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

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

Test

```
public class Test {

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

        C*retaker ctk = new Ca*etaker();
        ctk.setMemento(org.createMemento());//将数据封装在 Caretaker

        or*.setState("睡觉中");
        org.sh*wState();//显示

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

result

```
睡觉中
开会中
```

1.3.7 观察者模式

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。

适用性

1. 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。

2. 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*多少对象有待改变。

3.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。

参与者

1.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。

2.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。

3.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。

4.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*使自身状态与目标的状态保持一致

类*

例子

Subject

```
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);//抽象方法在子类实现

    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);
    }
}
```

oncreteSubjec

```
public class HuangPuCitizen extends Citizen {

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

    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);
        }
    }
}

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);
        }
    }
}
```

Observer

```
public interface Policeman {

    void action(Citizen ci);//在子类实现
}
```

ConcreteObserver

```
public clas* HuangPuPoliceman implements Pol*ceman {

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

public c*ass TianHePoliceman implements Policem*n {

    public void action(Citizen ci) {
        String help = *i.getHelp();
        if (help.equals("*ormal")) {
            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

```
public class Test{

    public st*tic void main(String[] args) {
        Policeman thPol = new TianHePoliceman();
        Pol*ceman hpPol = new HuangPuPoliceman();

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

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

        citizen = new TianHeCitizen(thPol);
        citizen.sendMessage("normal");
        citi*en.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!
```

1.3.8 状态模式

定义对象间*一种一对多的依赖关系,当一个对象的状态*生改变时,所*依赖于它的对象都得到通知并被自动更新。

适用性

1.一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*它的行为。

2.一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。

参与者

1.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。

2.State

定义一个接口以封装与 Context 的一个特定状态相关的行为。

3.ConcreteStatesubclasses

每一子类实现一个与 Context 的一个状态*关的行为。

类图

例子

***ontext**

```
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

```
public interface Weather {

    String getWeather();
}
```

Concrete*tatesubclasses

```
public class Rain implements Weather {

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

public class Sunshine implements Weather {

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

Test

```
public class Test{

    public static void main(String[] args) {
        Context ctx1 = new Context();
        ctx1.setWeather(new Sunshine());
        System.out.printlnctx1.weatherMessage());

        System.*ut.println("===============");

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

result

```
阳光
===============
下雨
```

1.3.9 策略模式

定义一系列的算法,把它们*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。

适用性

1.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。

2.需要使用一个算法的不同变体。

3.算法使用客户不应该知*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。

4*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*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。

参与者

1.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。

2. ConcreteStrategy
以 Strategy 接口实现某具体算法。

3. Context
用一个 ConcreteStrateg*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。

类图

例子

Strategy

```
public abstract class Strategy {

    pu*lic abstract void method();
}
```

ConcreteStrategy

```
public class *trategyImplA extends Strategy {

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

public class StrategyImplB extends Strategy {

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

public class StrategyImplC extends Strategy {

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

Context

```
pub*ic class Context {
```

```
    Strategy stra;

    public Cont*xt(Strategy stra) {
        this.stra = stra;
    }

    pub*ic void doMethod() {
        stra.method();
    }
}
```

Test

```
public class Test {

    public static void main(String[] ar*s) {
        Context ctx = new C*ntext(new StrategyImplA());
        ctx.doMethod();

        ctx = new Context(new *trategyImplB());
        ctx.doMethod();

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

result

```
这是第一个实现
这是第二个实现
这是第三个实现
```

1.3.10 模板方法

定义*个操作中的算法的骨架，*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。

适用性

1.一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。

2.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*码中的不同之处，并且将不同之处分离为新的操作。

最后，用*个调用这些新的操作的模板方法来替换这些不同的代码。

3.控制子类*展。

参与者

1.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。

2.ConcreteClass

实现*语操作以完成算法中与特定子类相关的步骤。

类图

例子

AbstractClass

```
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

```
public class TemplateConcrete extends Template {

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

Test

```
public class Test {

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

result

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

1.3.11 访问者模式

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。

适用性

1.一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*。

2.需要对一个对象结构中的对象进行很多不同的并且不相关的操作，*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。

3.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*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。

参与者

1. Visitor
   为该对象结构中 ConcreteEle*ent 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*isit 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*具体的类。
   这样访问者就可以通过该元素的特定接口直*访问它。

2. Concret*Visitor
   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。
   Concret*Visitor*该算法提供了上下文并存*它的局部状态。
   这一状态常常在遍历该结构的过程中累*结果。

3. Element
   定义一个 Accept 操作，它*一个访问者为参数。

4. ConcreteElement
   实现 Accept 操作，该操作以一个访问者为参数。

5. ObjectStru*ture
   能枚举它的元素。
   *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。

类图

例子

Visitor

```
public interface Visitor {

    public void visitString(StringElement stringE);

    public void visitFloat(FloatElement floatE);

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

ConcreteVisitor

```
public class C*ncreteVisitor implements Visitor {

    public void visitCollectio*(Collection colle*tion) {
        // TODO Auto-generated method stub
        Iterator iterator = collection.iterator();
        while (iterator.hasNext()) {
            Object o = iterato*.next();
```

```
            if (o in*tanceof 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

```
public interface Visitabl* {

    publ*c void accept(Visitor visitor);
}
```

ConcreteElement

```
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.*isitFloat(this);
    }
}

public class StringElement implements Visitable *

    private String se;

    public String*lement(String se) {
```

```
        this.se = se;
    }

    public String getS*() {
        return thi*.se;
    }

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

Test

```
public class Test {

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

        Fl*atElement fe = new FloatElement(n*w Float(1.5));
        fe.accept(visitor);
        S*stem.out.println("===========");
        List result = new ArrayList();
        result.add(new StringEle*ent("abc"));
        result.a*d(new StringElement("abc"));
        result.add(*ew 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
```