

# 1, 设计模式概述

---

## 1.1 软件设计模式的产生背景

---

"设计模式"最初并不是出现在软件设计中,而是被用于建筑领域的设计中。

1977年美国著名建筑大师、加利福尼亚大学伯克利分校环境结构中心主任 克里斯托夫·亚历山大 (Christopher Alexander) 在他的著作《建筑模式语言: 城镇、建筑、构造》中描述了一些常见的建筑设计问题,并提出了 253 种关于对城镇、邻里、住宅、花园和房间等进行设计的基本模式。

1990年软件工程界开始研讨设计模式的话题,后来召开了多次关于设计模式的研讨会。直到1995年,艾瑞克·伽马 (Erich Gamma)、理查德·海尔姆 (Richard Helm)、拉尔夫·约翰森 (Ralph Johnson)、约翰·威利斯迪斯 (John Vlissides) 等 4 位作者合作出版了《设计模式: 可复用面向对象软件的基础》一书,在此书中收录了 23 个设计模式,这是设计模式领域里程碑的事件,导致了软件设计模式的突破。这 4 位作者在软件开发领域里也以他们的“四人组”(Gang of Four, GoF) 著称。

## 1.2 软件设计模式的概念

---

软件设计模式 (Software Design Pattern), 又称设计模式,是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题,以及该问题的解决方案。也就是说,它是解决特定问题的一系列套路,是前辈们的代码设计经验的总结,具有一定的普遍性,可以反复使用。

## 1.3 学习设计模式的必要性

---

设计模式的本质是面向对象设计原则的实际运用,是对类的封装性、继承性和多态性以及类的关联关系和组合关系的充分理解。

正确使用设计模式具有以下优点。

- 可以提高程序员的思维能力、编程能力和设计能力。
- 使程序设计更加标准化、代码编制更加工程化,使软件开发效率大大提高,从而缩短软件的开发周期。
- 使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强。

## 1.4 设计模式分类

---

### • 创建型模式

用于描述“怎样创建对象”,它的主要特点是“将对象的创建与使用分离”。GoF (四人组) 书中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

### • 结构型模式

用于描述如何将类或对象按某种布局组成更大的结构,GoF (四人组) 书中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

### • 行为型模式

用于描述类或对象之间怎样相互协作共同完成单个对象无法单独完成的任务,以及怎样分配职责。GoF (四人组) 书中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

## 2, UML图

统一建模语言 (Unified Modeling Language, UML) 是用来设计软件的可视化建模语言。它的特点是简单、统一、图形化、能表达软件设计中的动态与静态信息。

UML 从目标系统的不同角度出发, 定义了用例图、类图、对象图、状态图、活动图、时序图、协作图、构件图、部署图等 9 种图。

### 2.1 类图概述

类图 (Class diagram) 是显示了模型的静态结构, 特别是模型中存在的类、类的内部结构以及它们与其他类的关系等。类图不显示暂时性的信息。类图是面向对象建模的主要组成部分。

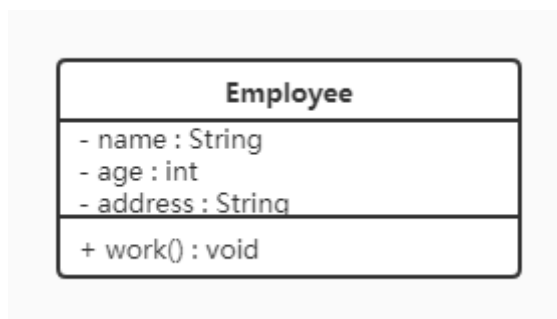
### 2.2 类图的作用

- 在软件工程中, 类图是一种静态的结构图, 描述了系统的类的集合, 类的属性和类之间的关系, 可以简化了人们对系统的理解;
- 类图是系统分析和设计阶段的重要产物, 是系统编码和测试的重要模型。

### 2.3 类图表示法

#### 2.3.1 类的表示方式

在UML类图中, 类使用包含类名、属性 (field) 和方法 (method) 且带有分割线的矩形来表示, 如下图所示表示一个Employee类, 它包含name, age和address这3个属性, 以及work() 方法。



属性/方法名称前加的加号和减号表示了这个属性/方法的可见性, UML类图中表示可见性的符号有三种:

- +: 表示public
- -: 表示private
- #: 表示protected

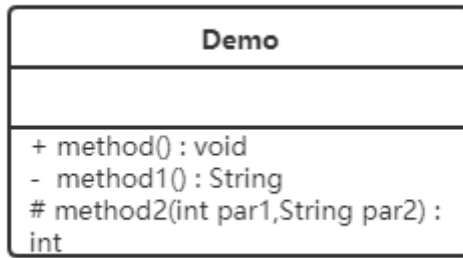
属性的完整表示方式是: **可见性 名称 : 类型 [ = 缺省值]**

方法的完整表示方式是: **可见性 名称 (参数列表) [ : 返回类型]**

注意:

- 1, 中括号中的内容表示是可选的
- 2, 也有将类型放在变量名前面, 返回值类型放在方法名前面

举个例子:



上图Demo类定义了三个方法:

- method()方法: 修饰符为public, 没有参数, 没有返回值。
- method1()方法: 修饰符为private, 没有参数, 返回值类型为String。
- method2()方法: 修饰符为protected, 接收两个参数, 第一个参数类型为int, 第二个参数类型为String, 返回值类型是int。

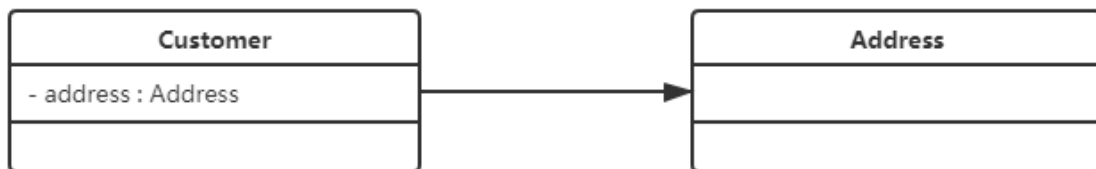
## 2.3.2 类与类之间关系的表示方式

### 2.3.2.1 关联关系

关联关系是对象之间的一种引用关系, 用于表示一类对象与另一类对象之间的联系, 如老师和学生、师傅和徒弟、丈夫和妻子等。关联关系是类与类之间最常用的一种关系, 分为一般关联关系、聚合关系和组合关系。我们先介绍一般关联。

关联又可以分为单向关联, 双向关联, 自关联。

#### 1, 单向关联



在UML类图中单向关联用一个带箭头的实线表示。上图表示每个顾客都有一个地址, 这通过让Customer类持有一个类型为Address的成员变量类实现。

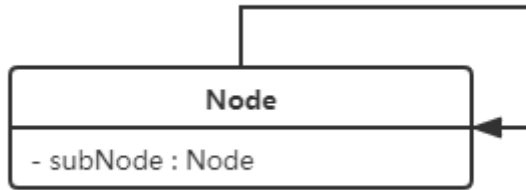
#### 2, 双向关联



从上图中我们很容易看出, 所谓的双向关联就是双方各自持有对方类型的成员变量。

在UML类图中, 双向关联用一个不带箭头的直线表示。上图中在Customer类中维护一个List<Product>, 表示一个顾客可以购买多个商品; 在Product类中维护一个Customer类型的成员变量表示这个产品被哪个顾客所购买。

#### 3, 自关联



自关联在UML类图中用一个带有箭头且指向自身的线表示。上图的意思就是Node类包含类型为Node的成员变量，也就是“自己包含自己”。

### 2.3.2.2 聚合关系

聚合关系是关联关系的一种，是强关联关系，是整体和部分之间的关系。

聚合关系也是通过成员对象来实现的，其中成员对象是整体对象的一部分，但是成员对象可以脱离整体对象而独立存在。例如，学校与老师的关系，学校包含老师，但如果学校停办了，老师依然存在。

在 UML 类图中，聚合关系可以用带空心菱形的实线来表示，菱形指向整体。下图所示是大学和教师的关系图：

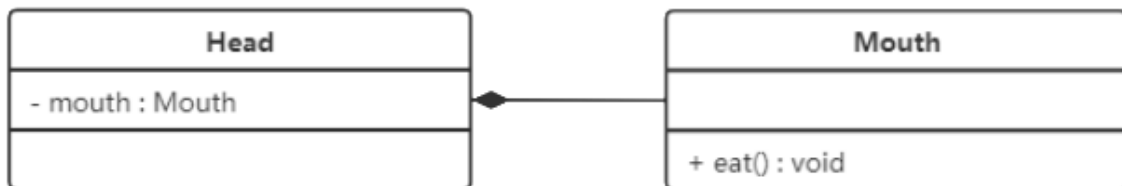


### 2.3.2.3 组合关系

组合表示类之间的整体与部分的关系，但它是一种更强烈的聚合关系。

在组合关系中，整体对象可以控制部分对象的生命周期，一旦整体对象不存在，部分对象也将不存在，部分对象不能脱离整体对象而存在。例如，头和嘴的关系，没有了头，嘴也就不存在了。

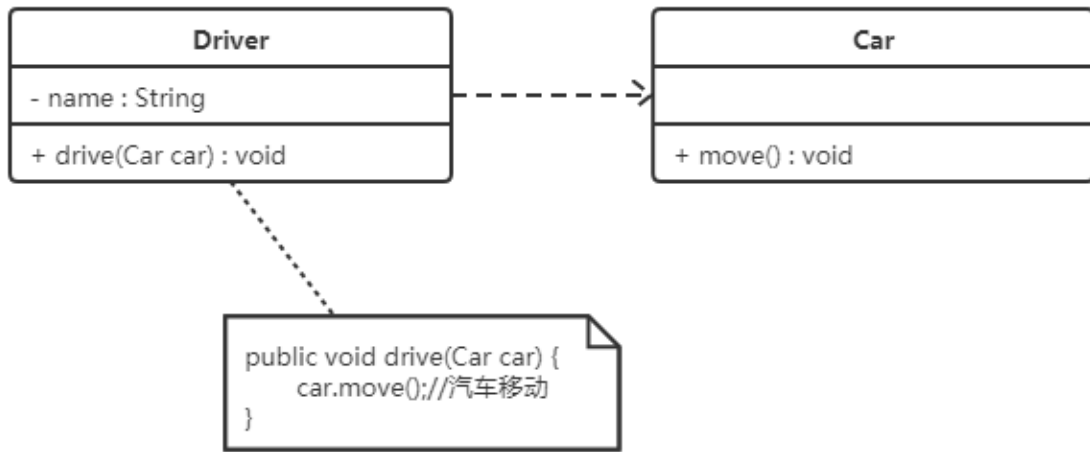
在 UML 类图中，组合关系用带实心菱形的实线来表示，菱形指向整体。下图所示是头和嘴的关系图：



### 2.3.2.4 依赖关系

依赖关系是一种使用关系，它是对象之间耦合度最弱的一种关联方式，是临时性的关联。在代码中，某个类的方法通过局部变量、方法的参数或者对静态方法的调用来访问另一个类（被依赖类）中的某些方法来完成一些职责。

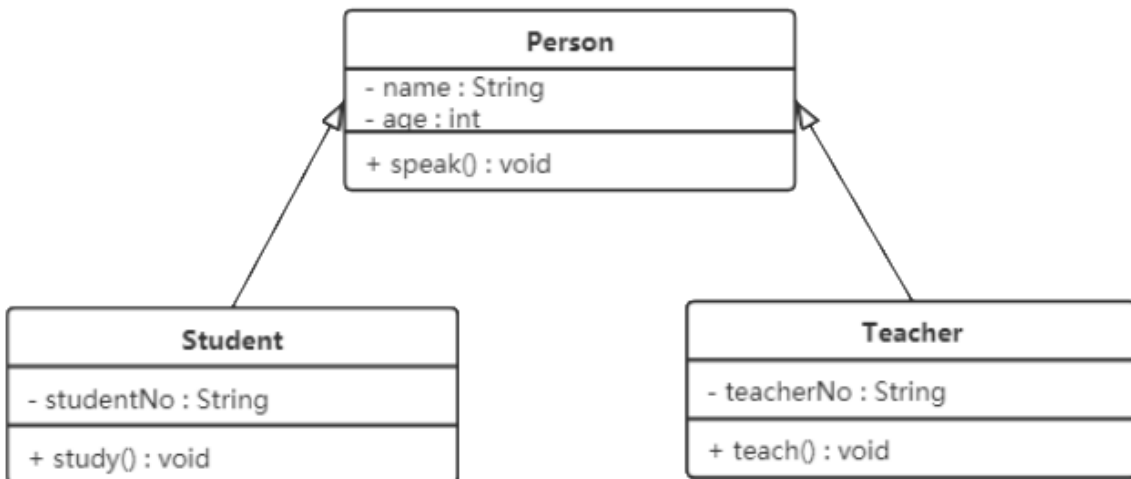
在 UML 类图中，依赖关系使用带箭头的虚线来表示，箭头从使用类指向被依赖的类。下图所示是司机和汽车的关系图，司机驾驶汽车：



### 2.3.2.5 继承关系

继承关系是对象之间耦合度最大的一种关系，表示一般与特殊的关系，是父类与子类之间的关系，是一种继承关系。

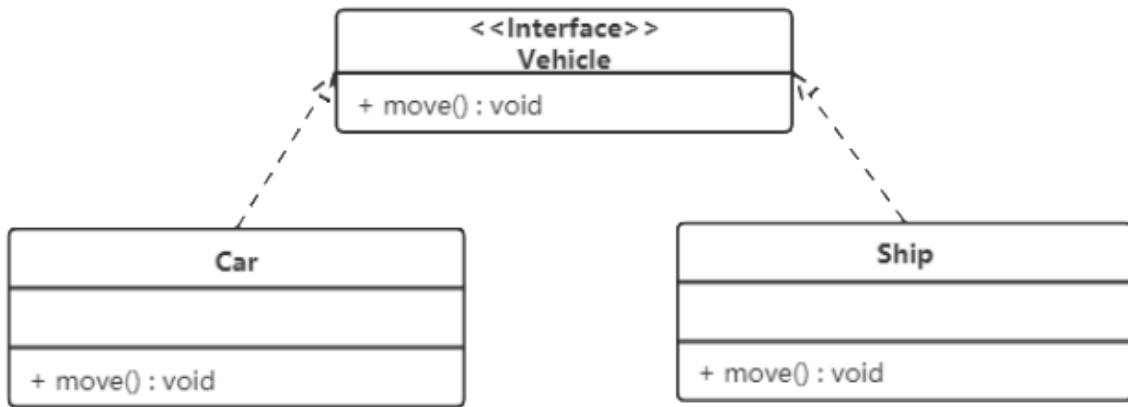
在 UML 类图中，泛化关系用带空心三角箭头的实线来表示，箭头从子类指向父类。在代码实现时，使用面向对象的继承机制来实现泛化关系。例如，Student 类和 Teacher 类都是 Person 类的子类，其类图如下图所示：



### 2.3.2.6 实现关系

实现关系是接口与实现类之间的关系。在这种关系中，类实现了接口，类中的操作实现了接口中所声明的所有的抽象操作。

在 UML 类图中，实现关系使用带空心三角箭头的虚线来表示，箭头从实现类指向接口。例如，汽车和船实现了交通工具，其类图如图 9 所示。



## 3， 软件设计原则

在软件开发中，为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，程序员要尽量根据6条原则来开发程序，从而提高软件开发效率、节约软件开发成本和维护成本。

### 3.1 开闭原则

**对扩展开放，对修改关闭。**在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。

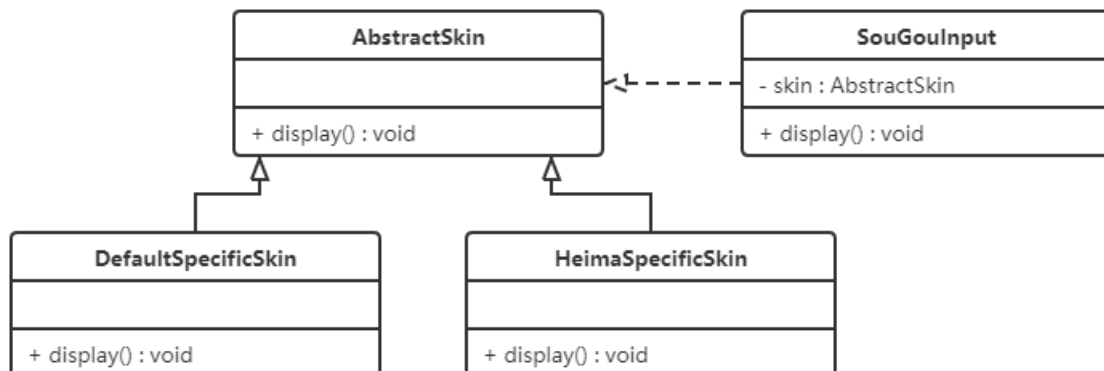
想要达到这样的效果，我们需要使用接口和抽象类。

因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节可以从抽象派生来的实现类来进行扩展，当软件需要发生变化时，只需要根据需求重新派生一个实现类来扩展就可以了。

下面以 **搜狗输入法** 的皮肤为例介绍开闭原则的应用。

**【例】** 搜狗输入法的皮肤设计。

分析：**搜狗输入法** 的皮肤是输入法背景图片、窗口颜色和声音等元素的组合。用户可以根据自己的喜爱更换自己的输入法的皮肤，也可以从网上下载新的皮肤。这些皮肤有共同的特点，可以为它定义一个抽象类 (`AbstractSkin`)，而每个具体的皮肤 (`DefaultSpecificSkin` 和 `HeimaSpecificSkin`) 是其子类。用户窗体可以根据需要选择或者增加新的主题，而不需要修改原代码，所以它是满足开闭原则的。



## 3.2 里氏代换原则

里氏代换原则是面向对象设计的基本原则之一。

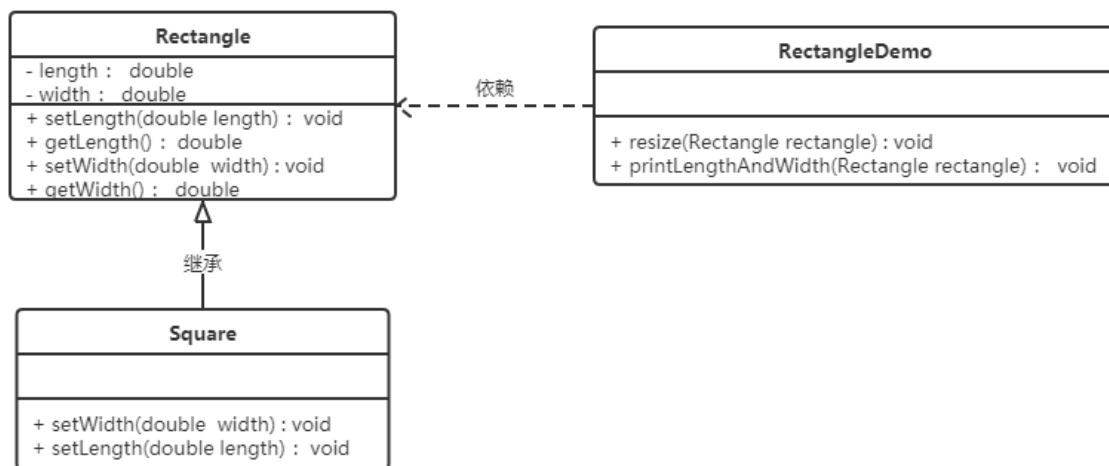
里氏代换原则：任何基类可以出现的地方，子类一定可以出现。通俗理解：子类可以扩展父类的功能，但不能改变父类原有的功能。换句话说，子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。

如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

下面看一个里氏替换原则中经典的一个例子

【例】正方形不是长方形。

在数学领域里，正方形毫无疑问是长方形，它是一个长宽相等的长方形。所以，我们开发的一个与几何图形相关的软件系统，就可以顺理成章的让正方形继承自长方形。



代码如下：

**长方形类 (Rectangle) :**

```
1 public class Rectangle {
2     private double length;
3     private double width;
4
5     public double getLength() {
6         return length;
7     }
8
9     public void setLength(double length) {
10        this.length = length;
11    }
12
13    public double getWidth() {
14        return width;
15    }
16
17    public void setWidth(double width) {
18        this.width = width;
19    }
20 }
```

## 正方形 (Square) :

由于正方形的长和宽相同，所以在方法`setLength`和`setWidth`中，对长度和宽度都需要赋相同值。

```
1 public class Square extends Rectangle {
2
3     public void setWidth(double width) {
4         super.setLength(width);
5         super.setWidth(width);
6     }
7
8     public void setLength(double length) {
9         super.setLength(length);
10        super.setWidth(length);
11    }
12 }
```

类`RectangleDemo`是我们的软件系统中的一个组件，它有一个`resize`方法依赖基类`Rectangle`，`resize`方法是`RectangleDemo`类中的一个方法，用来实现宽度逐渐增长的效果。

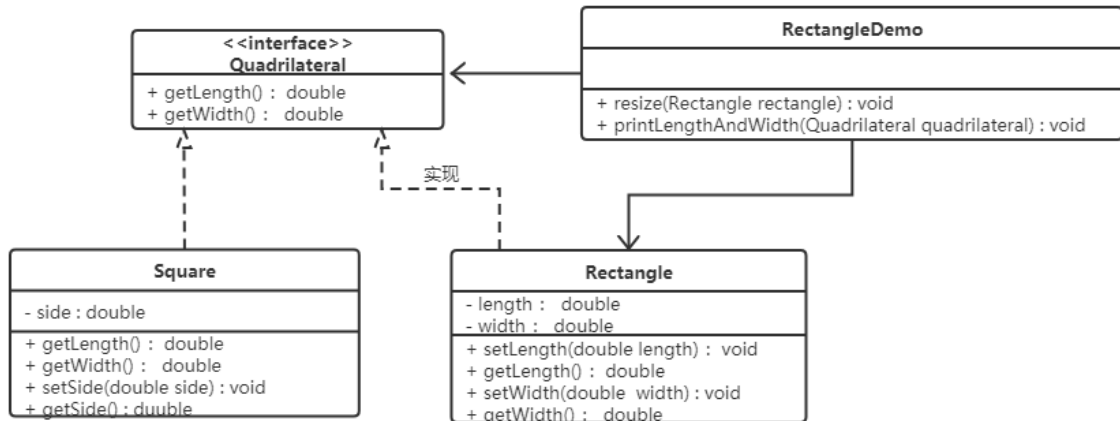
```
1 public class RectangleDemo {
2
3     public static void resize(Rectangle rectangle) {
4         while (rectangle.getWidth() <= rectangle.getLength()) {
5             rectangle.setWidth(rectangle.getWidth() + 1);
6         }
7     }
8
9     //打印长方形的长和宽
10    public static void printLengthAndWidth(Rectangle rectangle) {
11        System.out.println(rectangle.getLength());
12        System.out.println(rectangle.getWidth());
13    }
14
15    public static void main(String[] args) {
16        Rectangle rectangle = new Rectangle();
17        rectangle.setLength(20);
18        rectangle.setWidth(10);
19        resize(rectangle);
20        printLengthAndWidth(rectangle);
21
22        System.out.println("=====");
23
24        Rectangle rectangle1 = new Square();
25        rectangle1.setLength(10);
26        resize(rectangle1);
27        printLengthAndWidth(rectangle1);
28    }
29 }
```

我们运行一下这段代码就会发现，假如我们把一个普通长方形作为参数传入`resize`方法，就会看到长方形宽度逐渐增长的效果，当宽度大于长度，代码就会停止，这种行为的结果符合我们的预期；假如我们再把一个正方形作为参数传入`resize`方法后，就会看到正方形的宽度和长度都在不断增长，代码会一直运行下去，直至系统产生溢出错误。所以，普通的长方形是适合这段代码的，正方形不适合。我们得出结论：在`resize`方法中，`Rectangle`类型的参数是不能被`Square`类型的参数所代替，如果进



行了替换就得不到预期结果。因此，Square类和Rectangle类之间的继承关系违反了里氏代换原则，它们之间的继承关系不成立，正方形不是长方形。

如何改进呢？此时我们需要重新设计他们之间的关系。抽象出来一个四边形接口(Quadrilateral)，让Rectangle类和Square类实现Quadrilateral接口



### 3.3 依赖倒转原则

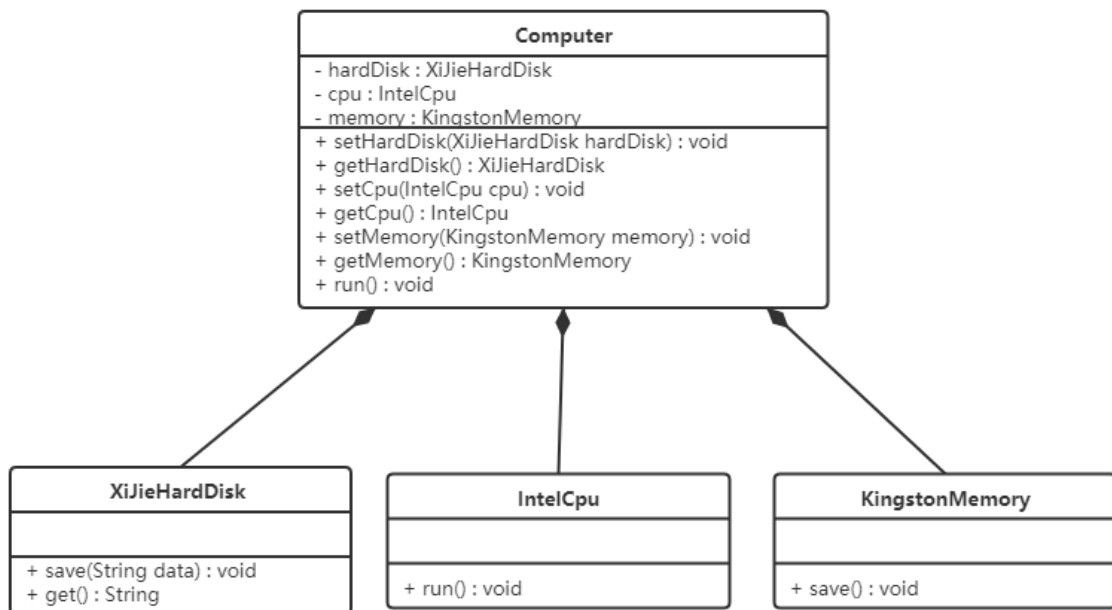
高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

下面看一个例子来理解依赖倒转原则

#### 【例】组装电脑

现要组装一台电脑，需要配件cpu，硬盘，内存条。只有这些配置都有了，计算机才能正常的运行。选择cpu有很多选择，如Intel，AMD等，硬盘可以选择希捷，西数等，内存条可以选择金士顿，海盗船等。

类图如下：



代码如下：

希捷硬盘类 (XiJieHardDisk) :

```
1 public class XiJieHardDisk implements HardDisk {
2
3     public void save(String data) {
4         System.out.println("使用希捷硬盘存储数据" + data);
5     }
6
7     public String get() {
8         System.out.println("使用希捷希捷硬盘取数据");
9         return "数据";
10    }
11 }
```

Intel处理器 (IntelCpu) :

```
1 public class IntelCpu implements Cpu {
2
3     public void run() {
4         System.out.println("使用Intel处理器");
5     }
6 }
```

金士顿内存条 (KingstonMemory) :

```
1 public class KingstonMemory implements Memory {
2
3     public void save() {
4         System.out.println("使用金士顿作为内存条");
5     }
6 }
```

电脑 (Computer) :

```
1 public class Computer {
2
3     private XiJieHardDisk hardDisk;
4     private IntelCpu cpu;
5     private KingstonMemory memory;
6
7     public IntelCpu getCpu() {
8         return cpu;
9     }
10
11    public void setCpu(IntelCpu cpu) {
12        this.cpu = cpu;
13    }
14
15    public KingstonMemory getMemory() {
16        return memory;
17    }
18
19    public void setMemory(KingstonMemory memory) {
20        this.memory = memory;
21    }
}
```

```

22
23     public XijieHardDisk getHardDisk() {
24         return hardDisk;
25     }
26
27     public void setHardDisk(XijieHardDisk hardDisk) {
28         this.hardDisk = hardDisk;
29     }
30
31     public void run() {
32         System.out.println("计算机工作");
33         cpu.run();
34         memory.save();
35         String data = hardDisk.get();
36         System.out.println("从硬盘中获取的数据为: " + data);
37     }
38 }

```

### 测试类 (TestComputer) :

测试类用来组装电脑。

```

1  public class TestComputer {
2      public static void main(String[] args) {
3          Computer computer = new Computer();
4          computer.setHardDisk(new XijieHardDisk());
5          computer.setCpu(new IntelCpu());
6          computer.setMemory(new KingstonMemory());
7
8          computer.run();
9      }
10 }

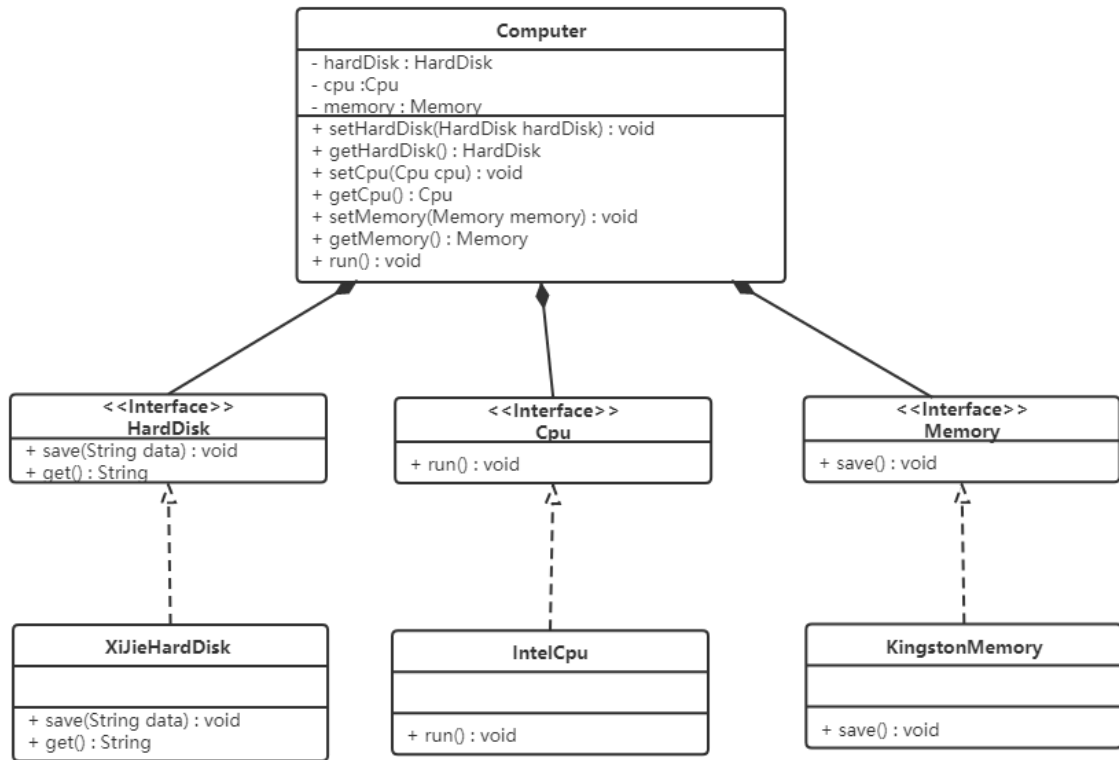
```

上面代码可以看到已经组装了一台电脑，但是似乎组装的电脑的cpu只能是Intel的，内存条只能是金士顿的，硬盘只能是希捷的，这对用户肯定是不友好的，用户有了机箱肯定是想按照自己的喜好，选择自己喜欢的配件。

根据依赖倒转原则进行改进：

代码我们只需要修改Computer类，让Computer类依赖抽象（各个配件的接口），而不是依赖于各个组件具体的实现类。

类图如下：



电脑 (Computer) :

```

1  public class Computer {
2
3      private HardDisk hardDisk;
4      private Cpu cpu;
5      private Memory memory;
6
7      public HardDisk getHardDisk() {
8          return hardDisk;
9      }
10
11     public void setHardDisk(HardDisk hardDisk) {
12         this.hardDisk = hardDisk;
13     }
14
15     public Cpu getCpu() {
16         return cpu;
17     }
18
19     public void setCpu(Cpu cpu) {
20         this.cpu = cpu;
21     }
22
23     public Memory getMemory() {
24         return memory;
25     }
26
27     public void setMemory(Memory memory) {
28         this.memory = memory;
29     }
30
31     public void run() {
32         system.out.println("计算机工作");
  
```

```
33     }
34 }
```

面向对象的开发很好的解决了这个问题，一般情况下抽象的变化概率很小，让用户程序依赖于抽象，实现的细节也依赖于抽象。即使实现细节不断变动，只要抽象不变，客户程序就不需要变化。这大大降低了客户程序与实现细节的耦合度。

### 3.4 接口隔离原则

客户端不应该被迫依赖于它不使用的方法；一个类对另一个类的依赖应该建立在最小的接口上。

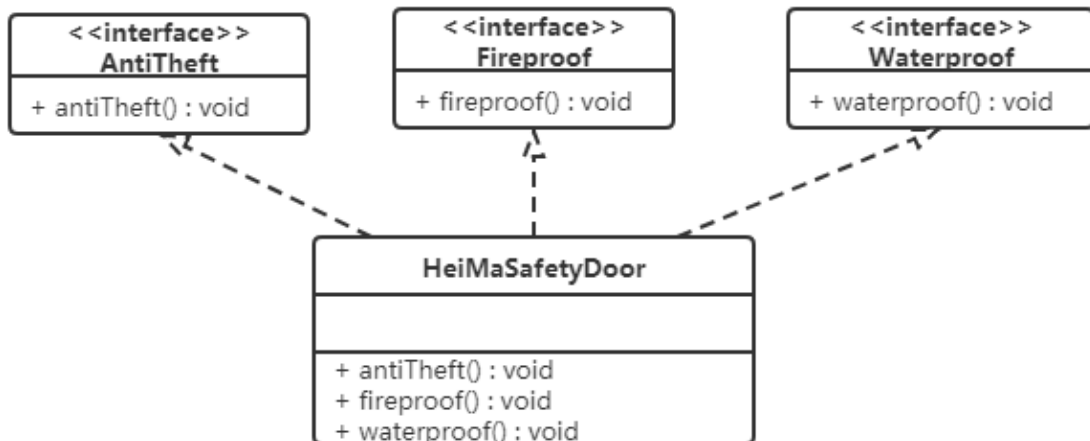
下面看一个例子来理解接口隔离原则

#### 【例】安全门案例

我们需要创建一个 **黑马** 品牌的安全门，该安全门具有防火、防水、防盗的功能。可以将防火，防水，防盗功能提取成一个接口，形成一套规范。类图如下：



上面的设计我们发现了它存在的问题，黑马品牌的安全门具有防盗，防水，防火的功能。现在如果我们还需要再创建一个传智品牌的安全门，而该安全门只具有防盗、防水功能呢？很显然如果实现 `SafetyDoor` 接口就违背了接口隔离原则，那么我们如何进行修改呢？看如下类图：



代码如下：

**AntiTheft (接口) :**

```
1 public interface AntiTheft {
2     void antiTheft();
3 }
```

**Fireproof (接口) :**

```
1 public interface Fireproof {
2     void fireproof();
3 }
```

Waterproof (接口) :

```
1 public interface Waterproof {
2     void waterproof();
3 }
```

HeiMaSafetyDoor (类) :

```
1 public class HeiMaSafetyDoor implements AntiTheft, Fireproof, Waterproof {
2     public void antiTheft() {
3         System.out.println("防盗");
4     }
5
6     public void fireproof() {
7         System.out.println("防火");
8     }
9
10
11     public void waterproof() {
12         System.out.println("防水");
13     }
14 }
```

ItcastSafetyDoor (类) :

```
1 public class ItcastSafetyDoor implements AntiTheft, Fireproof {
2     public void antiTheft() {
3         System.out.println("防盗");
4     }
5
6     public void fireproof() {
7         System.out.println("防火");
8     }
9 }
```

## 3.5 迪米特法则

迪米特法则又叫最少知识原则。

只和你的直接朋友交谈，不跟“陌生人”说话 (Talk only to your immediate friends and not to strangers)。

其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

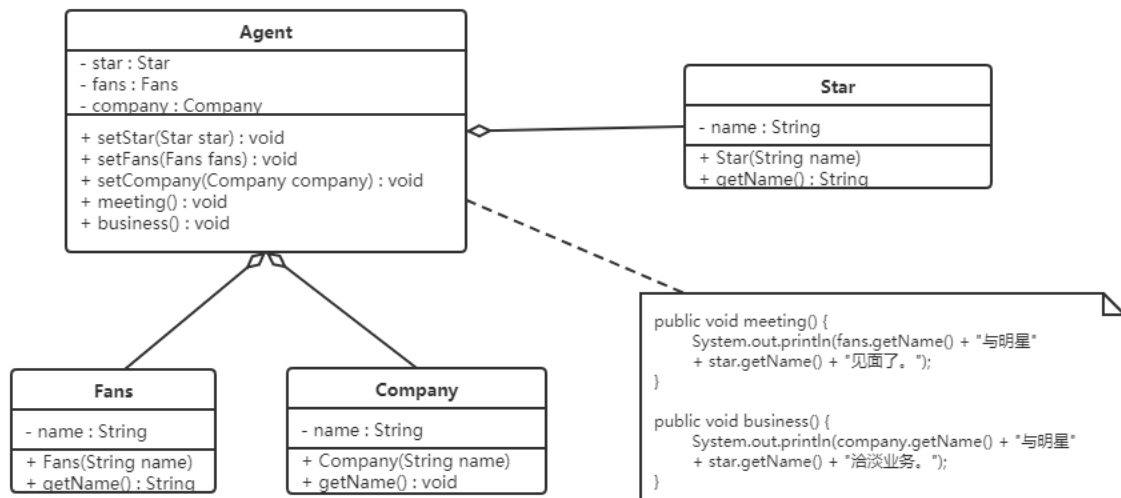
迪米特法则中的“朋友”是指：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。

下面看一个例子来理解迪米特法则

## 【例】明星与经纪人的关系实例

明星由于全身心投入艺术，所以许多日常事务由经纪人负责处理，如和粉丝的见面会，和媒体公司的业务洽谈等。这里的经纪人是明星的朋友，而粉丝和媒体公司是陌生人，所以适合使用迪米特法则。

类图如下：



代码如下：

### 明星类 (Star)

```
1 public class Star {
2     private String name;
3
4     public Star(String name) {
5         this.name=name;
6     }
7
8     public String getName() {
9         return name;
10    }
11 }
```

### 粉丝类 (Fans)

```
1 public class Fans {
2     private String name;
3
4     public Fans(String name) {
5         this.name=name;
6     }
7
8     public String getName() {
9         return name;
10    }
11 }
```

### 媒体公司类 (Company)

```

1 public class Company {
2     private String name;
3
4     public Company(String name) {
5         this.name=name;
6     }
7
8     public String getName() {
9         return name;
10    }
11 }

```

### 经纪人类 (Agent)

```

1 public class Agent {
2     private Star star;
3     private Fans fans;
4     private Company company;
5
6     public void setStar(Star star) {
7         this.star = star;
8     }
9
10    public void setFans(Fans fans) {
11        this.fans = fans;
12    }
13
14    public void setCompany(Company company) {
15        this.company = company;
16    }
17
18    public void meeting() {
19        System.out.println(fans.getName() + "与明星" + star.getName() + "见面
20    了。");
21    }
22
23    public void business() {
24        System.out.println(company.getName() + "与明星" + star.getName() +
25    "洽谈业务。");
26    }
27 }

```

## 3.6 合成复用原则

合成复用原则是指：尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。

通常类的复用分为继承复用和合成复用两种。

继承复用虽然有简单和易实现的优点，但它也存在以下缺点：

1. 继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。
2. 子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。



3. 它限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

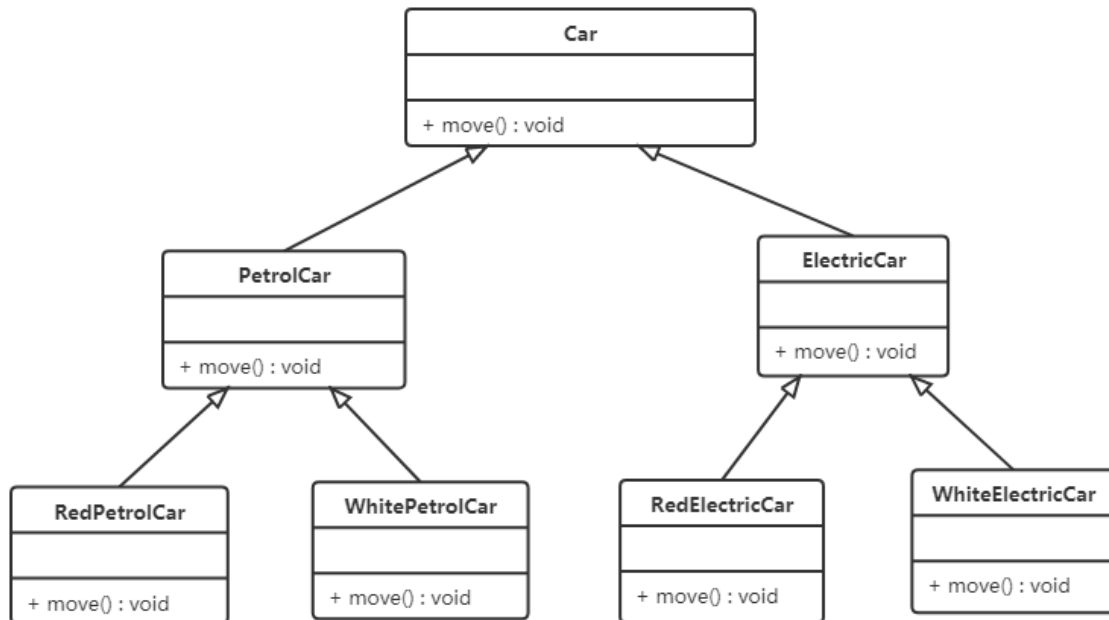
采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点：

1. 它维持了类的封装性。因为成分对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。
2. 对象间的耦合度低。可以在类的成员位置声明抽象。
3. 复用的灵活性高。这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

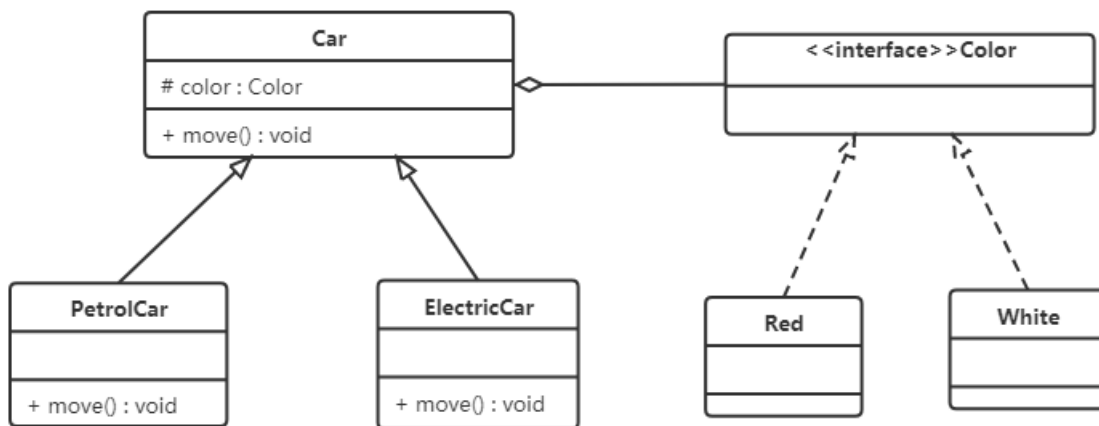
下面看一个例子来理解合成复用原则

#### 【例】汽车分类管理程序

汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。如果同时考虑这两种分类，其组合就很多。类图如下：



从上面类图我们可以看到使用继承复用产生了很多子类，如果现在又有新的动力源或者新的颜色的话，就需要再定义新的类。我们试着将继承复用改为聚合复用看一下。



## 4，创建者模式

创建型模式的主要关注点是“怎样创建对象？”，它的主要特点是“将对象的创建与使用分离”。

这样可以降低系统的耦合度，使用者不需要关注对象的创建细节。

创建型模式分为：

- 单例模式
- 工厂方法模式
- 抽象工程模式
- 原型模式
- 建造者模式

### 4.1 单例设计模式

单例模式 (Singleton Pattern) 是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

#### 4.1.1 单例模式的结构

单例模式的主要有以下角色：

- 单例类。只能创建一个实例的类
- 访问类。使用单例类

#### 4.1.2 单例模式的实现

单例设计模式分类两种：

饿汉式：类加载就会导致该单实例对象被创建

懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

##### 1. 饿汉式-方式1（静态变量方式）

```
1  /**
2   * 饿汉式
3   *    静态变量创建类的对象
4   */
5  public class Singleton {
6      //私有构造方法
7      private Singleton() {}
8
9      //在成员位置创建该类的对象
10     private static Singleton instance = new Singleton();
11
12     //对外提供静态方法获取该对象
13     public static Singleton getInstance() {
14         return instance;
15     }
16 }
```

说明：

该方式在成员位置声明Singleton类型的静态变量，并创建Singleton类的对象instance。instance对象是随着类的加载而创建的。如果该对象足够大的话，而一直没有使用就会造成内存的浪费。

## 2. 饿汉式-方式2 (静态代码块方式)

```
1  /**
2   * 饿汉式
3   *      在静态代码块中创建该类对象
4   */
5  public class Singleton {
6
7      //私有构造方法
8      private Singleton() {}
9
10     //在成员位置创建该类的对象
11     private static Singleton instance;
12
13     static {
14         instance = new Singleton();
15     }
16
17     //对外提供静态方法获取该对象
18     public static Singleton getInstance() {
19         return instance;
20     }
21 }
```

### 说明:

该方式在成员位置声明Singleton类型的静态变量，而对象的创建是在静态代码块中，也是对着类的加载而创建。所以和饿汉式的方式1基本上一样，当然该方式也存在内存浪费问题。

## 3. 懒汉式-方式1 (线程不安全)

```
1  /**
2   * 懒汉式
3   *      线程不安全
4   */
5  public class Singleton {
6      //私有构造方法
7      private Singleton() {}
8
9      //在成员位置创建该类的对象
10     private static Singleton instance;
11
12     //对外提供静态方法获取该对象
13     public static Singleton getInstance() {
14
15         if(instance == null) {
16             instance = new Singleton();
17         }
18         return instance;
19     }
20 }
```

#### 说明:

从上面代码我们可以看出该方式在成员位置声明Singleton类型的静态变量，并没有进行对象的赋值操作，那么什么时候赋值的呢？当调用getInstance()方法获取Singleton类的对象的时候才创建Singleton类的对象，这样就实现了懒加载的效果。但是，如果是多线程环境，会出现线程安全问题。

#### 4. 懒汉式-方式2 (线程安全)

```
1  /**
2   * 懒汉式
3   * 线程安全
4   */
5  public class Singleton {
6      //私有构造方法
7      private Singleton() {}
8
9      //在成员位置创建该类的对象
10     private static Singleton instance;
11
12     //对外提供静态方法获取该对象
13     public static synchronized Singleton getInstance() {
14
15         if(instance == null) {
16             instance = new Singleton();
17         }
18         return instance;
19     }
20 }
```

#### 说明:

该方式也实现了懒加载效果，同时又解决了线程安全问题。但是在getInstance()方法上添加了synchronized关键字，导致该方法的执行效果特别低。从上面代码我们可以看出，其实就是在初始化instance的时候才会出现线程安全问题，一旦初始化完成就不存在了。

#### 5. 懒汉式-方式3 (双重检查锁)

再来讨论一下懒汉模式中加锁的问题，对于getInstance()方法来说，绝大部分的操作都是读操作，读操作是线程安全的，所以我们不必让每个线程必须持有锁才能调用该方法，我们需要调整加锁的时机。由此也产生了一种新的实现模式：双重检查锁模式

```
1  /**
2   * 双重检查方式
3   */
4  public class Singleton {
5
6      //私有构造方法
7      private Singleton() {}
8
9      private static Singleton instance;
10
11     //对外提供静态方法获取该对象
12     public static Singleton getInstance() {
13         //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例
```

```

14     if(instance == null) {
15         synchronized (Singleton.class) {
16             //抢到锁之后再次判断是否为null
17             if(instance == null) {
18                 instance = new Singleton();
19             }
20         }
21     }
22     return instance;
23 }
24 }

```

双重检查锁模式是一种非常好的单例实现模式，解决了单例、性能、线程安全问题，上面的双重检测锁模式看上去完美无缺，其实是存在问题，在多线程的情况下，可能会出现空指针问题，出现问题的原因是JVM在实例化对象的时候会进行优化和指令重排序操作。

要解决双重检查锁模式带来空指针异常的问题，只需要使用 `volatile` 关键字，`volatile` 关键字可以保证可见性和有序性。

```

1  /**
2   * 双重检查方式
3   */
4  public class Singleton {
5
6      //私有构造方法
7      private Singleton() {}
8
9      private static volatile Singleton instance;
10
11     //对外提供静态方法获取该对象
12     public static Singleton getInstance() {
13         //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
14         if(instance == null) {
15             synchronized (Singleton.class) {
16                 //抢到锁之后再次判断是否为空
17                 if(instance == null) {
18                     instance = new Singleton();
19                 }
20             }
21         }
22         return instance;
23     }
24 }

```

### 小结:

添加 `volatile` 关键字之后的双重检查锁模式是一种比较好的单例实现模式，能够保证在多线程的情况下线程安全也不会有性能问题。

## 6. 懒汉式-方式4 (静态内部类方式)

静态内部类单例模式中实例由内部类创建，由于 JVM 在加载外部类的过程中，是不会加载静态内部类的，只有内部类的属性/方法被调用时才会被加载，并初始化其静态属性。静态属性由于被 `static` 修饰，保证只被实例化一次，并且严格保证实例化顺序。

```

1  /**

```

```

2  * 静态内部类方式
3  */
4  public class Singleton {
5
6      //私有构造方法
7      private Singleton() {}
8
9      private static class SingletonHolder {
10         private static final Singleton INSTANCE = new Singleton();
11     }
12
13     //对外提供静态方法获取该对象
14     public static Singleton getInstance() {
15         return SingletonHolder.INSTANCE;
16     }
17 }

```

#### 说明:

第一次加载Singleton类时不会去初始化INSTANCE，只有第一次调用getInstance，虚拟机加载SingletonHolder

并初始化INSTANCE，这样不仅能确保线程安全，也能保证 Singleton 类的唯一性。

#### 小结:

静态内部类单例模式是一种优秀的单例模式，是开源项目中比较常用的一种单例模式。在没有加任何锁的情况下，保证了多线程下的安全，并且没有任何性能影响和空间的浪费。

## 7. 枚举方式

枚举类实现单例模式是极力推荐的单例实现模式，因为枚举类型是线程安全的，并且只会装载一次，设计者充分的利用了枚举的这个特性来实现单例模式，枚举的写法非常简单，而且枚举类型是所用单例实现中唯一一种不会被破坏的单例实现模式。

```

1  /**
2   * 枚举方式
3   */
4  public enum Singleton {
5      INSTANCE;
6  }

```

#### 说明:

枚举方式属于恶汉式方式。

## 4.1.3 存在的问题

### 4.1.3.1 问题演示

破坏单例模式:

使上面定义的单例类 (Singleton) 可以创建多个对象，枚举方式除外。有两种方式，分别是序列化和反射。

- 序列化反序列化

Singleton类:

```

1 public class Singleton implements Serializable {
2
3     //私有构造方法
4     private Singleton() {}
5
6     private static class SingletonHolder {
7         private static final Singleton INSTANCE = new Singleton();
8     }
9
10    //对外提供静态方法获取该对象
11    public static Singleton getInstance() {
12        return SingletonHolder.INSTANCE;
13    }
14 }

```

**Test类:**

```

1 public class Test {
2     public static void main(String[] args) throws Exception {
3         //往文件中写对象
4         //writeObject2File();
5         //从文件中读取对象
6         Singleton s1 = readObjectFromFile();
7         Singleton s2 = readObjectFromFile();
8
9         //判断两个反序列化后的对象是否是同一个对象
10        System.out.println(s1 == s2);
11    }
12
13    private static Singleton readObjectFromFile() throws Exception {
14        //创建对象输入流对象
15        ObjectInputStream ois = new ObjectInputStream(new
16        FileInputStream("C:\\Users\\Think\\Desktop\\a.txt"));
17        //第一个读取Singleton对象
18        Singleton instance = (Singleton) ois.readObject();
19
20        return instance;
21    }
22
23    public static void writeObject2File() throws Exception {
24        //获取Singleton类的对象
25        Singleton instance = Singleton.getInstance();
26        //创建对象输出流
27        ObjectOutputStream oos = new ObjectOutputStream(new
28        FileOutputStream("C:\\Users\\Think\\Desktop\\a.txt"));
29        //将instance对象写出到文件中
30        oos.writeObject(instance);
31    }
32 }

```

上面代码运行结果是 `false`，表明序列化和反序列化已经破坏了单例设计模式。

- 反射

**Singleton类:**

```

1 public class Singleton {

```

```

2
3 //私有构造方法
4 private Singleton() {}
5
6 private static volatile Singleton instance;
7
8 //对外提供静态方法获取该对象
9 public static Singleton getInstance() {
10
11     if(instance != null) {
12         return instance;
13     }
14
15     synchronized (Singleton.class) {
16         if(instance != null) {
17             return instance;
18         }
19         instance = new Singleton();
20         return instance;
21     }
22 }
23 }

```

**Test类:**

```

1 public class Test {
2     public static void main(String[] args) throws Exception {
3         //获取Singleton类的字节码对象
4         Class clazz = Singleton.class;
5         //获取Singleton类的私有无参构造方法对象
6         Constructor constructor = clazz.getDeclaredConstructor();
7         //取消访问检查
8         constructor.setAccessible(true);
9
10        //创建Singleton类的对象s1
11        Singleton s1 = (Singleton) constructor.newInstance();
12        //创建Singleton类的对象s2
13        Singleton s2 = (Singleton) constructor.newInstance();
14
15        //判断通过反射创建的两个Singleton对象是否是同一个对象
16        System.out.println(s1 == s2);
17    }
18 }

```

上面代码运行结果是 `false`，表明序列化和反序列化已经破坏了单例设计模式

**注意:** 枚举方式不会出现这两个问题。

#### 4.1.3.2 问题的解决

- 序列化、反序列方式破坏单例模式的解决方法

在Singleton类中添加 `readResolve()` 方法，在反序列化时被反射调用，如果定义了这个方法，就返回这个方法的值，如果没有定义，则返回新new出来的对象。

**Singleton类:**



```

1 public class Singleton implements Serializable {
2
3     //私有构造方法
4     private Singleton() {}
5
6     private static class SingletonHolder {
7         private static final Singleton INSTANCE = new Singleton();
8     }
9
10    //对外提供静态方法获取该对象
11    public static Singleton getInstance() {
12        return SingletonHolder.INSTANCE;
13    }
14
15    /**
16     * 下面是为了解决序列化反序列化破解单例模式
17     */
18    private Object readResolve() {
19        return SingletonHolder.INSTANCE;
20    }
21 }

```

### 源码解析:

ObjectInputStream类

```

1 public final Object readObject() throws IOException,
  ClassNotFoundException{
2     ...
3     // if nested read, passHandle contains handle of enclosing object
4     int outerHandle = passHandle;
5     try {
6         Object obj = readObject0(false); //重点查看readObject0方法
7         .....
8     }
9
10    private Object readObject0(boolean unshared) throws IOException {
11        ...
12        try {
13            switch (tc) {
14                ...
15                case TC_OBJECT:
16                    return checkResolve(readOrdinaryObject(unshared)); //重点
  查看readOrdinaryObject方法
17                ...
18            }
19        } finally {
20            depth--;
21            bin.setBlockDataMode(oldMode);
22        }
23    }
24
25    private Object readOrdinaryObject(boolean unshared) throws IOException
  {
26        ...
27        //isInstantiable 返回true, 执行 desc.newInstance(), 通过反射创建新的单例
  类,

```

```

28     obj = desc.isInstantiable() ? desc.newInstance() : null;
29     ...
30     // 在Singleton类中添加 readResolve 方法后 desc.hasReadResolveMethod()
方法执行结果为true
31     if (obj != null && handles.lookupException(passHandle) == null &&
desc.hasReadResolveMethod()) {
32         // 通过反射调用 Singleton 类中的 readResolve 方法，将返回值赋值给rep变
量
33         // 这样多次调用ObjectInputStream类中的readObject方法，继而就会调用我们
定义的readResolve方法，所以返回的是同一个对象。
34         Object rep = desc.invokeReadResolve(obj);
35         ...
36     }
37     return obj;
38 }

```

- 反射方式破解单例的解决方法

```

1  public class Singleton {
2
3      //私有构造方法
4      private Singleton() {
5          /*
6              反射破解单例模式需要添加的代码
7          */
8          if(instance != null) {
9              throw new RuntimeException();
10         }
11     }
12
13     private static volatile Singleton instance;
14
15     //对外提供静态方法获取该对象
16     public static Singleton getInstance() {
17
18         if(instance != null) {
19             return instance;
20         }
21
22         synchronized (Singleton.class) {
23             if(instance != null) {
24                 return instance;
25             }
26             instance = new Singleton();
27             return instance;
28         }
29     }
30 }

```

#### 说明:

这种方式比较好理解。当通过反射方式调用构造方法进行创建创建时，直接抛异常。不运行此中操作。

## 4.1.4 JDK源码解析-Runtime类

Runtime类就是使用的单例设计模式。

## 1. 通过源代码查看使用的是哪一种单例模式

```
1 public class Runtime {
2     private static Runtime currentRuntime = new Runtime();
3
4     /**
5      * Returns the runtime object associated with the current Java
6      application.
7      * Most of the methods of class <code>Runtime</code> are instance
8      * methods and must be invoked with respect to the current runtime
9      object.
10     *
11     * @return the <code>Runtime</code> object associated with the
12     current
13     *         Java application.
14     */
15     public static Runtime getRuntime() {
16         return currentRuntime;
17     }
18
19     /** Don't let anyone else instantiate this class */
20     private Runtime() {}
21     ...
22 }
```

从上面源代码中可以看出Runtime类使用的是饿汉式（静态属性）方式来实现单例模式的。

## 2. 使用Runtime类中的方法

```
1 public class RuntimeDemo {
2     public static void main(String[] args) throws IOException {
3         //获取Runtime类对象
4         Runtime runtime = Runtime.getRuntime();
5
6         //返回 Java 虚拟机中的内存总量。
7         System.out.println(runtime.totalMemory());
8         //返回 Java 虚拟机试图使用的最大内存量。
9         System.out.println(runtime.maxMemory());
10
11         //创建一个新的进程执行指定的字符串命令，返回进程对象
12         Process process = runtime.exec("ipconfig");
13         //获取命令执行后的结果，通过输入流获取
14         InputStream inputStream = process.getInputStream();
15         byte[] arr = new byte[1024 * 1024 * 100];
16         int b = inputStream.read(arr);
17         System.out.println(new String(arr,0,b,"gbk"));
18     }
19 }
```

# 4, 创建型模式

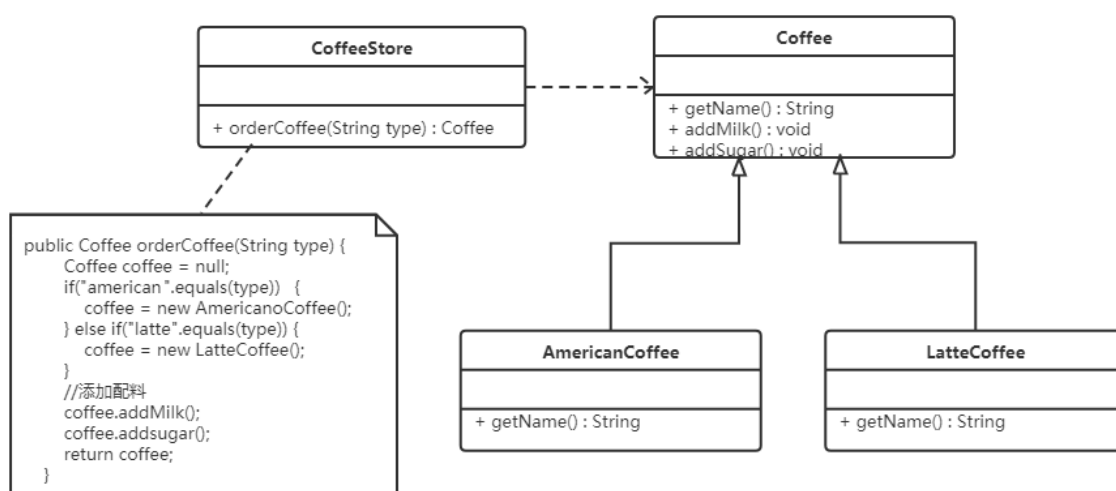
## 4.2 工厂模式

### 4.2.1 概述

需求：设计一个咖啡店点餐系统。

设计一个咖啡类 (Coffee) , 并定义其两个子类 (美式咖啡【AmericanoCoffee】和拿铁咖啡【LatteCoffee】) ; 再设计一个咖啡店类 (CoffeeStore) , 咖啡店具有点咖啡的功能。

具体类的设计如下:



在java中, 万物皆对象, 这些对象都需要创建, 如果创建的时候直接new该对象, 就会对该对象耦合严重, 假如我们要更换对象, 所有new对象的地方都需要修改一遍, 这显然违背了软件设计的开闭原则。如果我们使用工厂来生产对象, 我们就只和工厂打交道就可以了, 彻底和对象解耦, 如果要更换对象, 直接在工厂里更换该对象即可, 达到了与对象解耦的目的; 所以说, 工厂模式最大的优点就是: **解耦**。

在本教程中会介绍三种工厂的使用

- 简单工厂模式 (不属于GOF的23种经典设计模式)
- 工厂方法模式
- 抽象工厂模式

### 4.2.2 简单工厂模式

简单工厂不是一种设计模式, 反而比较像是一种编程习惯。

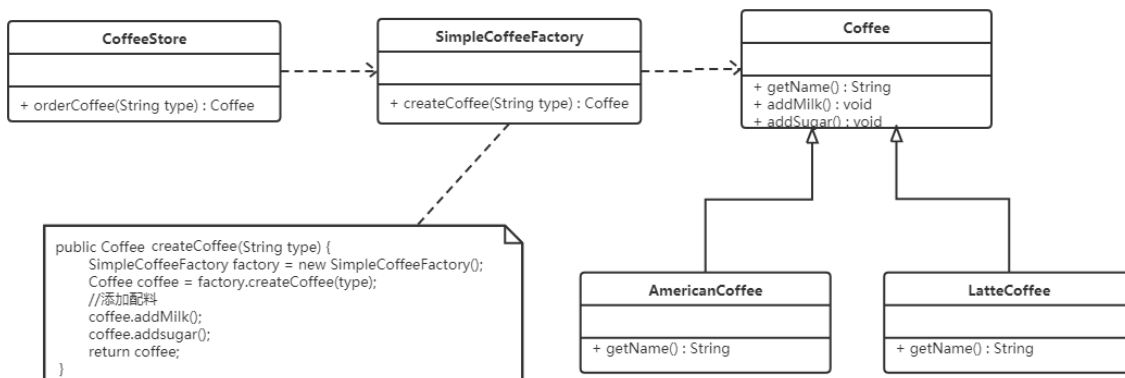
#### 4.2.2.1 结构

简单工厂包含如下角色:

- 抽象产品 : 定义了产品的规范, 描述了产品的主要特性和功能。
- 具体产品 : 实现或者继承抽象产品的子类
- 具体工厂 : 提供了创建产品的方法, 调用者通过该方法来获取产品。

#### 4.2.2.2 实现

现在使用简单工厂对上面案例进行改进，类图如下：



工厂类代码如下：

```
1 public class SimpleCoffeeFactory {
2
3     public Coffee createCoffee(String type) {
4         Coffee coffee = null;
5         if("americano".equals(type)) {
6             coffee = new AmericanoCoffee();
7         } else if("latte".equals(type)) {
8             coffee = new LatteCoffee();
9         }
10        return coffee;
11    }
12 }
```

工厂 (factory) 处理创建对象的细节，一旦有了 `SimpleCoffeeFactory`, `CoffeeStore` 类中的 `orderCoffee()` 就变成此对象的客户，后期如果需要 `Coffee` 对象直接从工厂中获取即可。这样也就解除了和 `Coffee` 实现类的耦合，同时又产生了新的耦合，`CoffeeStore` 对象和 `SimpleCoffeeFactory` 工厂对象的耦合，工厂对象和商品对象的耦合。

后期如果再加新品种的咖啡，我们势必要需求修改 `SimpleCoffeeFactory` 的代码，违反了开闭原则。工厂类的客户端可能有很多，比如创建美团外卖等，这样只需要修改工厂类的代码，省去其他的修改操作。

#### 4.2.2.4 优缺点

**优点：**

封装了创建对象的过程，可以通过参数直接获取对象。把对象的创建和业务逻辑层分开，这样以后就避免了修改客户代码，如果要实现新产品直接修改工厂类，而不需要在原代码中修改，这样就降低了客户代码修改的可能性，更加容易扩展。

**缺点：**

增加新产品时还是需要修改工厂类的代码，违背了“开闭原则”。

#### 4.2.2.3 扩展

**静态工厂**

在开发中也有一部分人将工厂类中的创建对象的功能定义为静态的，这个就是静态工厂模式，它也不是 23 种设计模式中的。代码如下：

```

1 public class SimpleCoffeeFactory {
2
3     public static Coffee createCoffee(String type) {
4         Coffee coffee = null;
5         if("americano".equals(type)) {
6             coffee = new AmericanoCoffee();
7         } else if("latte".equals(type)) {
8             coffee = new LatteCoffee();
9         }
10        return coffee;
11    }
12 }

```

### 4.2.3 工厂方法模式

针对上例中的缺点，使用工厂方法模式就可以完美的解决，完全遵循开闭原则。

#### 4.2.3.1 概念

定义一个用于创建对象的接口，让子类决定实例化哪个产品类对象。工厂方法使一个产品类的实例化延迟到其工厂的子类。

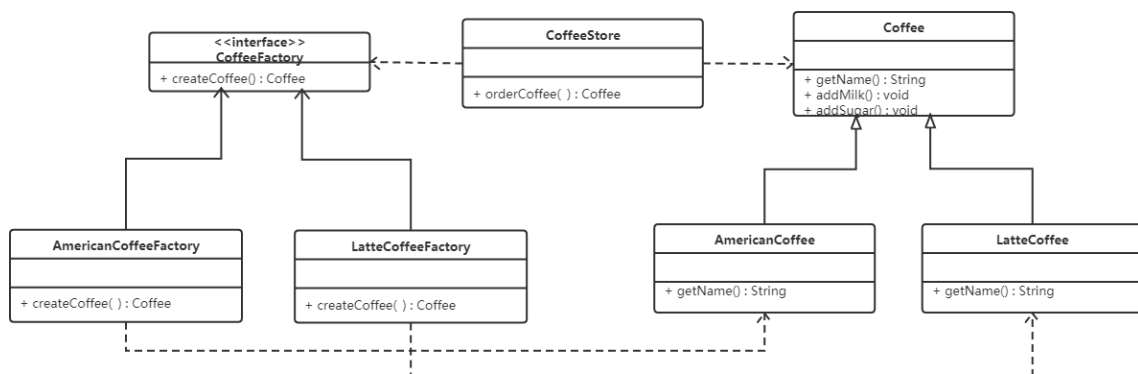
#### 4.2.3.2 结构

工厂方法模式的主要角色：

- 抽象工厂 (Abstract Factory)：提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法来创建产品。
- 具体工厂 (Concrete Factory)：主要是实现抽象工厂中的抽象方法，完成具体产品的创建。
- 抽象产品 (Product)：定义了产品的规范，描述了产品的主要特性和功能。
- 具体产品 (Concrete Product)：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

#### 4.2.3.3 实现

使用工厂方法模式对上例进行改进，类图如下：



代码如下：

抽象工厂：

```

1 public interface CoffeeFactory {
2
3     Coffee createCoffee();
4 }

```

具体工厂:

```

1 public class LatteCoffeeFactory implements CoffeeFactory {
2
3     public Coffee createCoffee() {
4         return new LatteCoffee();
5     }
6 }
7
8 public class AmericanCoffeeFactory implements CoffeeFactory {
9
10    public Coffee createCoffee() {
11        return new AmericanCoffee();
12    }
13 }

```

咖啡店类:

```

1 public class CoffeeStore {
2
3     private CoffeeFactory factory;
4
5     public CoffeeStore(CoffeeFactory factory) {
6         this.factory = factory;
7     }
8
9     public Coffee orderCoffee(String type) {
10        Coffee coffee = factory.createCoffee();
11        coffee.addMilk();
12        coffee.addsugar();
13        return coffee;
14    }
15 }

```

从以上的编写的代码可以看到，要增加产品类时也要相应地增加工厂类，不需要修改工厂类的代码了，这样就解决了简单工厂模式的缺点。

工厂方法模式是简单工厂模式的进一步抽象。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。

#### 4.2.3.4 优缺点

**优点:**

- 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

**缺点:**

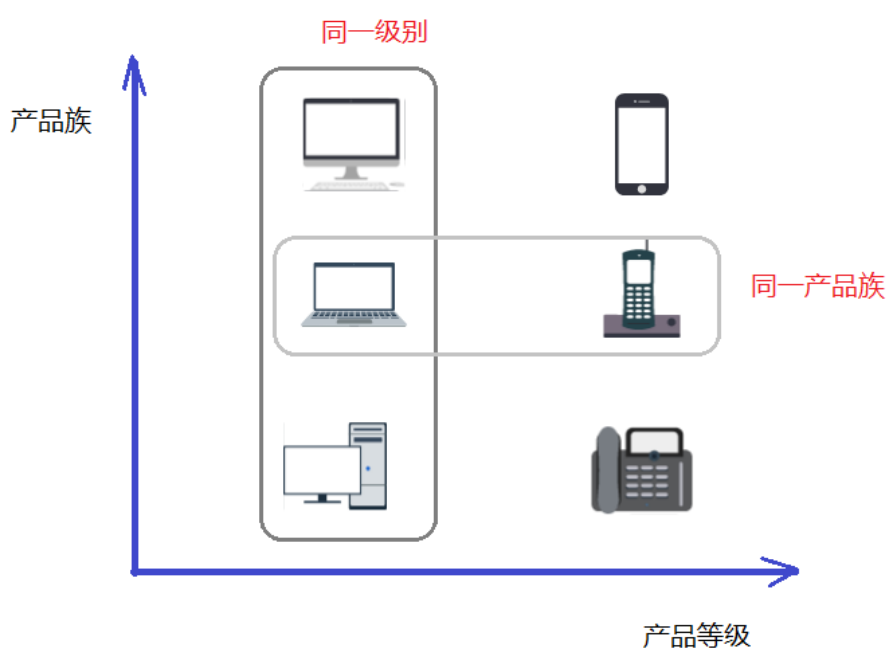
- 每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

## 4.2.4 抽象工厂模式

前面介绍的工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机、传智播客只培养计算机软件专业的学生等。

这些工厂只生产同种类产品，同种类产品称为同等级产品，也就是说：工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类）的产品，如电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

本节要介绍的抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族，下图所示横轴是产品等级，也就是同一类产品；纵轴是产品族，也就是同一品牌的产品，同一品牌的产品产自同一个工厂。



同一产品族



同一产品族



### 4.2.4.1 概念

是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。



抽象工厂模式是工厂方法模式的升级版，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

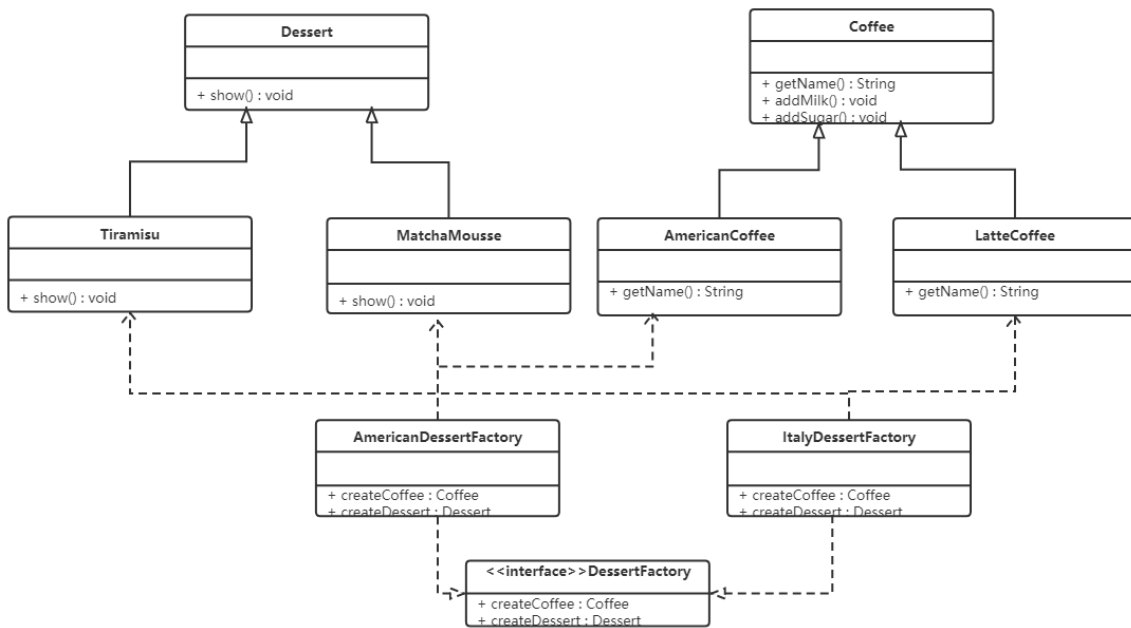
#### 4.2.4.2 结构

抽象工厂模式的主要角色如下：

- 抽象工厂 (Abstract Factory)：提供了创建产品的接口，它包含多个创建产品的方法，可以创建多个不同等级的产品。
- 具体工厂 (Concrete Factory)：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。
- 抽象产品 (Product)：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。
- 具体产品 (ConcreteProduct)：实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间是多对一的关系。

#### 4.2.4.2 实现

现咖啡店业务发生改变，不仅要生产咖啡还要生产甜点，如提拉米苏、抹茶慕斯等，要是按照工厂方法模式，需要定义提拉米苏类、抹茶慕斯类、提拉米苏工厂、抹茶慕斯工厂、甜点工厂类，很容易发生类爆炸情况。其中拿铁咖啡、美式咖啡是一个产品等级，都是咖啡；提拉米苏、抹茶慕斯也是一个产品等级；拿铁咖啡和提拉米苏是同一产品族（也就是都属于意大利风味），美式咖啡和抹茶慕斯是同一产品族（也就是都属于美式风味）。所以这个案例可以使用抽象工厂模式实现。类图如下：



代码如下：

抽象工厂：

```
1 public interface DessertFactory {
2
3     coffee createCoffee();
4
5     Dessert createDessert();
6 }
```

具体工厂：

```
1
```

```

1 //美式甜点工厂
2 public class AmericanDessertFactory implements DessertFactory {
3
4     public Coffee createCoffee() {
5         return new AmericanCoffee();
6     }
7
8     public Dessert createDessert() {
9         return new MatchaMousse();
10    }
11 }
12 //意大利风味甜点工厂
13 public class ItalyDessertFactory implements DessertFactory {
14
15     public Coffee createCoffee() {
16         return new LatteCoffee();
17     }
18
19     public Dessert createDessert() {
20         return new Tiramisu();
21     }
22 }

```

如果要加同一个产品族的话，只需要再加一个对应的工厂类即可，不需要修改其他的类。

#### 4.2.4.3 优缺点

##### 优点:

当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象。

##### 缺点:

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

#### 4.2.4.4 使用场景

- 当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。
- 系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。
- 系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

如：输入法换皮肤，一整套一起换。生成不同操作系统的程序。

#### 4.2.5 模式扩展

##### 简单工厂+配置文件解除耦合

可以通过工厂模式+配置文件的方式解除工厂对象和产品对象的耦合。在工厂类中加载配置文件中的全类名，并创建对象进行存储，客户端如果需要对象，直接进行获取即可。

第一步：定义配置文件

为了演示方便，我们使用properties文件作为配置文件，名称为bean.properties

```
1 | american=com.itheima.pattern.factory.config_factory.AmericanCoffee
2 | latte=com.itheima.pattern.factory.config_factory.LatteCoffee
```

第二步：改进工厂类

```
1 | public class CoffeeFactory {
2 |
3 |     private static Map<String,Coffee> map = new HashMap();
4 |
5 |     static {
6 |         Properties p = new Properties();
7 |         InputStream is =
CoffeeFactory.class.getClassLoader().getResourceAsStream("bean.properties")
8 |         ;
9 |         try {
10 |             p.load(is);
11 |             //遍历Properties集合对象
12 |             Set<Object> keys = p.keySet();
13 |             for (Object key : keys) {
14 |                 //根据键获取值（全类名）
15 |                 String className = p.getProperty((String) key);
16 |                 //获取字节码对象
17 |                 Class clazz = Class.forName(className);
18 |                 Coffee obj = (Coffee) clazz.newInstance();
19 |                 map.put((String)key,obj);
20 |             }
21 |         } catch (Exception e) {
22 |             e.printStackTrace();
23 |         }
24 |
25 |     public static Coffee createCoffee(String name) {
26 |
27 |         return map.get(name);
28 |     }
29 | }
```

静态成员变量用来存储创建的对象（键存储的是名称，值存储的是对应的对象），而读取配置文件以及创建对象写在静态代码块中，目的就是只需要执行一次。

#### 4.2.6 JDK源码解析-Collection.iterator方法

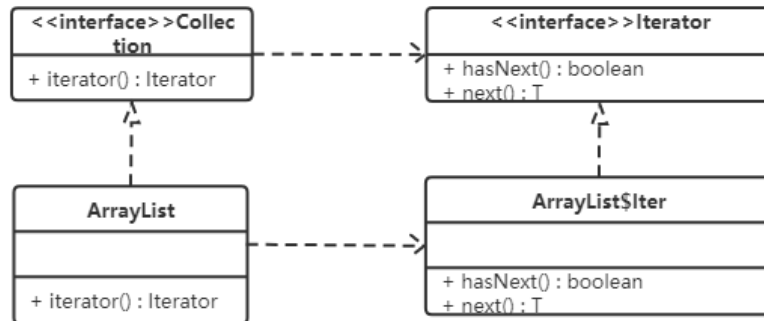
```
1 | public class Demo {
2 |     public static void main(String[] args) {
3 |         List<String> list = new ArrayList<>();
4 |         list.add("令狐冲");
5 |         list.add("风清扬");
6 |         list.add("任我行");
7 |
8 |         //获取迭代器对象
9 |         Iterator<String> it = list.iterator();
10 |        //使用迭代器遍历
11 |        while(it.hasNext()) {
12 |            String ele = it.next();
```

```

13     System.out.println(ele);
14     }
15 }
16 }

```

对上面的代码大家应该很熟，使用迭代器遍历集合，获取集合中的元素。而单列集合获取迭代器的方法就使用到了工厂方法模式。我们看通过类图看看结构：



Collection接口是抽象工厂类，ArrayList是具体的工厂类；Iterator接口是抽象商品类，ArrayList类中的Iter内部类是具体的商品类。在具体的工厂类中iterator()方法创建具体的商品类的对象。

另：

- 1, DateFormat类中的getInstance()方法使用的是工厂模式；
- 2, Calendar类中的getInstance()方法使用的是工厂模式；

## 4.3 原型模式

### 4.3.1 概述

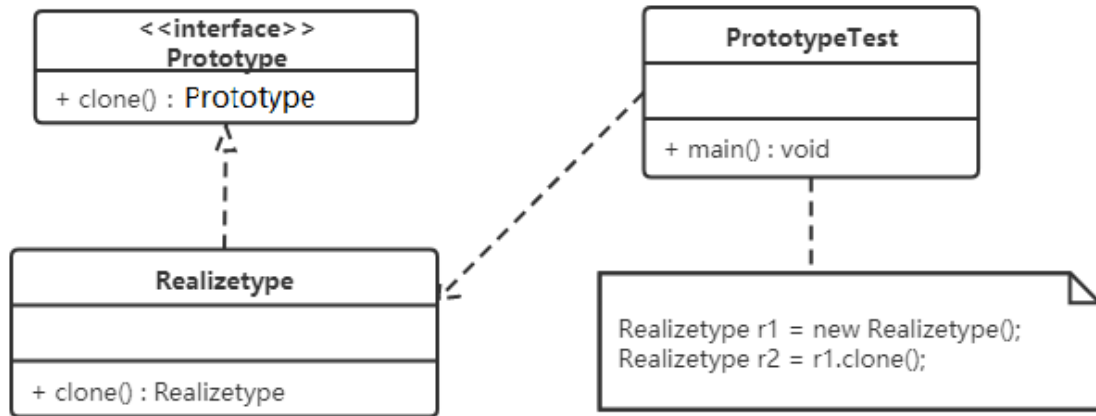
用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型对象相同的新对象。

### 4.3.2 结构

原型模式包含如下角色：

- 抽象原型类：规定了具体原型对象必须实现的的 clone() 方法。
- 具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象。
- 访问类：使用具体原型类中的 clone() 方法来复制新的对象。

接口类图如下：



### 4.3.3 实现

原型模式的克隆分为浅克隆和深克隆。

浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。

深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。

Java中的Object类中提供了 `clone()` 方法来实现浅克隆。 `Cloneable` 接口是上面的类图中的抽象原型类，而实现了 `Cloneable` 接口的子实现类就是具体的原型类。代码如下：

**Realizetype (具体的原型类) :**

```

1 public class Realizetype implements Cloneable {
2
3     public Realizetype() {
4         System.out.println("具体的原型对象创建完成! ");
5     }
6
7     @Override
8     protected Realizetype clone() throws CloneNotSupportedException {
9         System.out.println("具体原型复制成功! ");
10        return (Realizetype) super.clone();
11    }
12 }

```

**PrototypeTest (测试访问类) :**

```

1 public class PrototypeTest {
2     public static void main(String[] args) throws CloneNotSupportedException
3     {
4         Realizetype r1 = new Realizetype();
5         Realizetype r2 = r1.clone();
6
7         System.out.println("对象r1和r2是同一个对象? " + (r1 == r2));
8     }
9 }

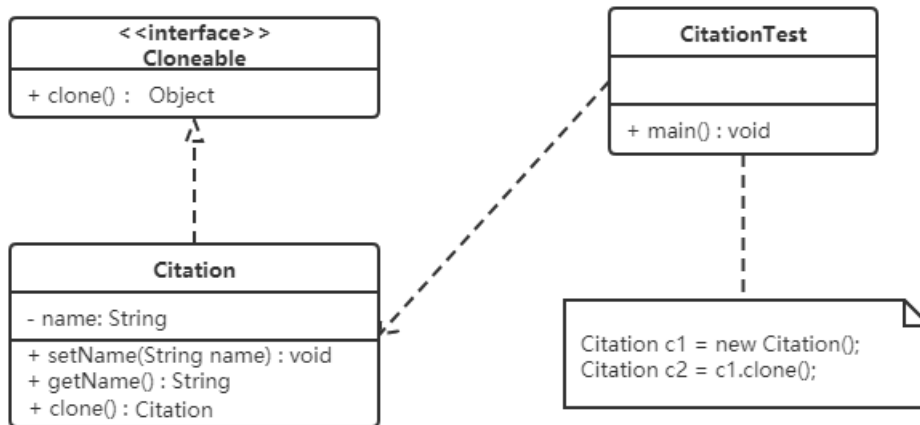
```

### 4.3.4 案例

用原型模式生成“三好学生”奖状

同一学校的“三好学生”奖状除了获奖人姓名不同，其他都相同，可以使用原型模式复制多个“三好学生”奖状出来，然后在修改奖状上的名字即可。

类图如下：



代码如下：

```
1 //奖状类
2 public class Citation implements Cloneable {
3     private String name;
4
5     public void setName(String name) {
6         this.name = name;
7     }
8
9     public String getName() {
10        return (this.name);
11    }
12
13    public void show() {
14        System.out.println(name + "同学：在2020学年第一学期中表现优秀，被评为三好学
15        生。特发此状！");
16    }
17
18    @Override
19    public Citation clone() throws CloneNotSupportedException {
20        return (Citation) super.clone();
21    }
22
23 //测试访问类
24 public class CitationTest {
25     public static void main(String[] args) throws
26     CloneNotSupportedException {
27         Citation c1 = new Citation();
28         c1.setName("张三");
29
30         //复制奖状
31         Citation c2 = c1.clone();
32         //将奖状的名字修改李四
33         c2.setName("李四");
34
35         c1.show();
36     }
37 }
```

```
35     c2.show();
36     }
37 }
```

### 4.3.5 使用场景

- 对象的创建非常复杂，可以使用原型模式快捷的创建对象。
- 性能和安全要求比较高。

### 4.3.6 扩展（深克隆）

将上面的“三好学生”奖章的案例中Citation类的name属性修改为Student类型的属性。代码如下：

```
1 //奖状类
2 public class Citation implements Cloneable {
3     private Student stu;
4
5     public Student getStu() {
6         return stu;
7     }
8
9     public void setStu(Student stu) {
10        this.stu = stu;
11    }
12
13    void show() {
14        System.out.println(stu.getName() + "同学：在2020学年第一学期中表现优秀，
15        被评为三好学生。特发此状！");
16    }
17
18    @Override
19    public Citation clone() throws CloneNotSupportedException {
20        return (Citation) super.clone();
21    }
22
23 //学生类
24 public class Student {
25     private String name;
26     private String address;
27
28     public Student(String name, String address) {
29         this.name = name;
30         this.address = address;
31     }
32
33     public Student() {
34     }
35
36     public String getName() {
37         return name;
38     }
39
40     public void setName(String name) {
41         this.name = name;
42     }
43 }
```

```

44     public String getAddress() {
45         return address;
46     }
47
48     public void setAddress(String address) {
49         this.address = address;
50     }
51 }
52
53 //测试类
54 public class CitationTest {
55     public static void main(String[] args) throws
CloneNotSupportedException {
56
57         Citation c1 = new Citation();
58         Student stu = new Student("张三", "西安");
59         c1.setStu(stu);
60
61         //复制奖状
62         Citation c2 = c1.clone();
63         //获取c2奖状所属学生对象
64         Student stu1 = c2.getStu();
65         stu1.setName("李四");
66
67         //判断stu对象和stu1对象是否是同一个对象
68         System.out.println("stu和stu1是同一个对象? " + (stu == stu1));
69
70         c1.show();
71         c2.show();
72     }
73 }

```

运行结果为:

```
D:\develop\java\jdk1.8\bin\java.exe ...
```

stu和stu1是同一个对象? true

李四同学: 在2020学年第一学期中表现优秀, 被评为三好学生。特发此状!

李四同学: 在2020学年第一学期中表现优秀, 被评为三好学生。特发此状!

说明:

stu对象和stu1对象是同一个对象, 就会产生将stu1对象中name属性值改为“李四”, 两个Citation (奖状) 对象中显示的都是李四。这就是浅克隆的效果, 对具体原型类 (Citation) 中的引用类型的属性进行引用的复制。这种情况需要使用深克隆, 而进行深克隆需要使用对象流。代码如下:

```

1 public class CitationTest1 {
2     public static void main(String[] args) throws Exception {
3         Citation c1 = new Citation();
4         Student stu = new Student("张三", "西安");
5         c1.setStu(stu);
6
7         //创建对象输出流对象
8         ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("C:\\Users\\Think\\Desktop\\b.txt"));
9         //将c1对象写出到文件中
10        oos.writeObject(c1);

```



```

11         oos.close();
12
13         //创建对象出入流对象
14         ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("c:\\Users\\Think\\Desktop\\b.txt"));
15         //读取对象
16         Citation c2 = (Citation) ois.readObject();
17         //获取c2奖状所属学生对象
18         Student stu1 = c2.getStu();
19         stu1.setName("李四");
20
21         //判断stu对象和stu1对象是否是同一个对象
22         System.out.println("stu和stu1是同一个对象? " + (stu == stu1));
23
24         c1.show();
25         c2.show();
26     }
27 }

```

运行结果为:

```
D:\develop\java\jdk1.8\bin\java.exe ...
```

stu和stu1是同一个对象? false

张三同学: 在2020学年第一学期中表现优秀, 被评为三好学生。特发此状!

李四同学: 在2020学年第一学期中表现优秀, 被评为三好学生。特发此状!

注意: Citation类和Student类必须实现Serializable接口, 否则会抛NotSerializableException异常。

## 4.5 建造者模式

### 4.4.1 概述

将一个复杂对象的构建与表示分离, 使得同样的构建过程可以创建不同的表示。



- 分离了部件的构造(由Builder来负责)和装配(由Director负责)。从而可以构造出复杂的对象。这个模式适用于: 某个对象的构建过程复杂的情况。
- 由于实现了构建和装配的解耦。不同的构建器, 相同的装配, 也可以做出不同的对象; 相同的构建器, 不同的装配顺序也可以做出不同的对象。也就是实现了构建算法、装配算法的解耦, 实现了更好的复用。

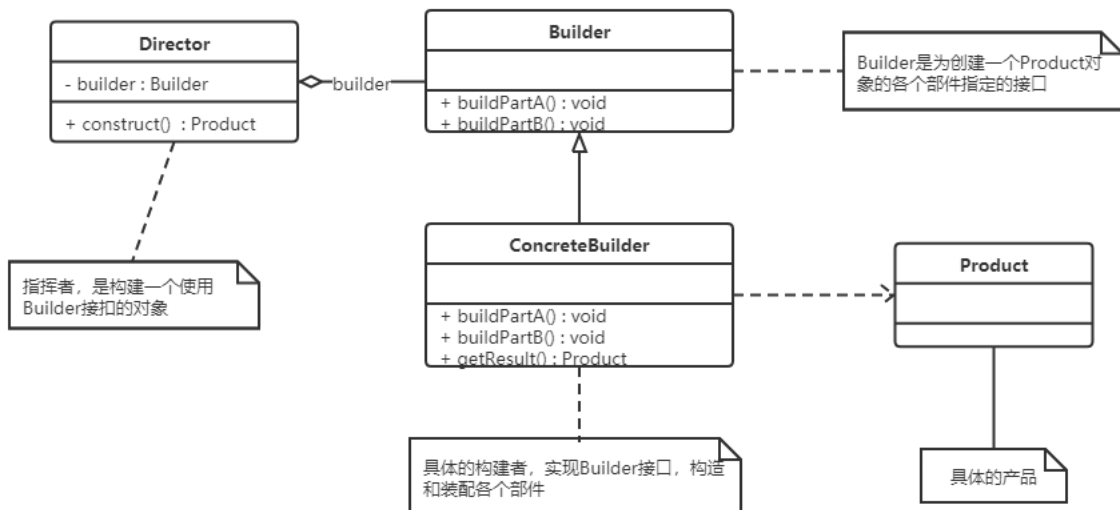
- 建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。

## 4.4.2 结构

建造者 (Builder) 模式包含如下角色：

- 抽象建造者类 (Builder)：这个接口规定要实现复杂对象的那些部分的创建，并不涉及具体的部件对象的创建。
- 具体建造者类 (ConcreteBuilder)：实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。在构造过程完成后，提供产品的实例。
- 产品类 (Product)：要创建的复杂对象。
- 指挥者类 (Director)：调用具体建造者来创建复杂对象的各个部分，在指挥者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

类图如下：

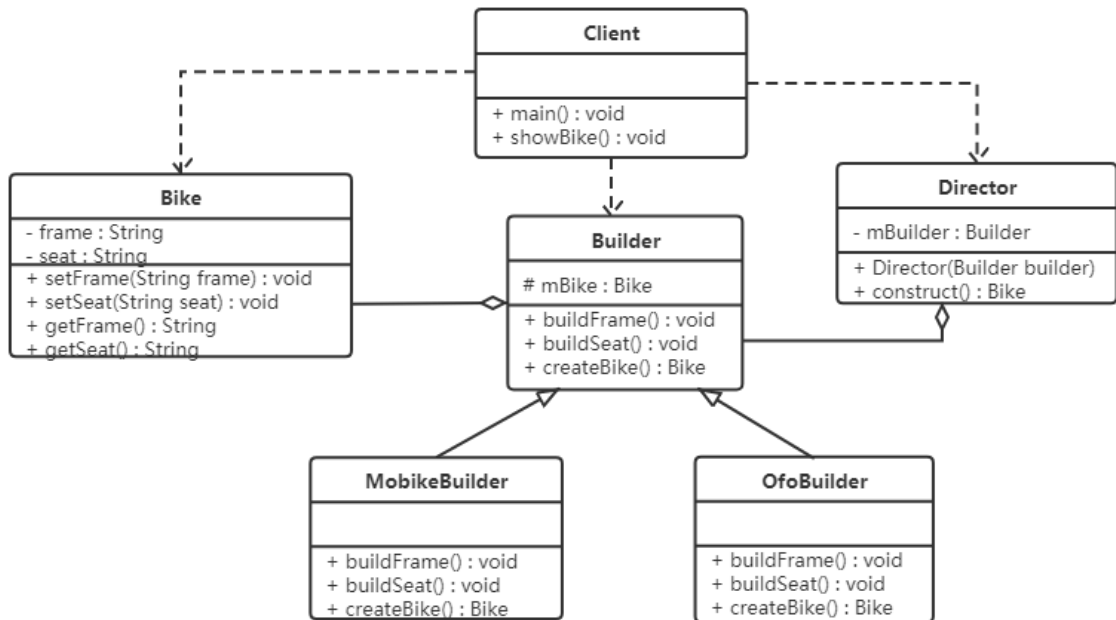


## 4.4.3 实例

### 创建共享单车

生产自行车是一个复杂的过程，它包含了车架，车座等组件的生产。而车架又有碳纤维，铝合金等材质的，车座有橡胶，真皮等材质。对于自行车的生产就可以使用建造者模式。

这里Bike是产品，包含车架，车座等组件；Builder是抽象建造者，MobikeBuilder和OfoBuilder是具体的建造者；Director是指挥者。类图如下：



具体的代码如下：

```

1 //自行车类
2 public class Bike {
3     private String frame;
4     private String seat;
5
6     public String getFrame() {
7         return frame;
8     }
9
10    public void setFrame(String frame) {
11        this.frame = frame;
12    }
13
14    public String getSeat() {
15        return seat;
16    }
17
18    public void setSeat(String seat) {
19        this.seat = seat;
20    }
21 }
22
23 // 抽象 builder 类
24 public abstract class Builder {
25
26     protected Bike mBike = new Bike();
27
28     public abstract void buildFrame();
29     public abstract void buildSeat();
30     public abstract Bike createBike();
31 }
32
33 // 摩拜单车Builder类
34 public class MobikeBuilder extends Builder {
35
36     @Override

```

```

37     public void buildFrame() {
38         mBike.setFrame("铝合金车架");
39     }
40
41     @Override
42     public void buildSeat() {
43         mBike.setSeat("真皮车座");
44     }
45
46     @Override
47     public Bike createBike() {
48         return mBike;
49     }
50 }
51
52 //ofo单车Builder类
53 public class OfoBuilder extends Builder {
54
55     @Override
56     public void buildFrame() {
57         mBike.setFrame("碳纤维车架");
58     }
59
60     @Override
61     public void buildSeat() {
62         mBike.setSeat("橡胶车座");
63     }
64
65     @Override
66     public Bike createBike() {
67         return mBike;
68     }
69 }
70
71 //指挥者类
72 public class Director {
73     private Builder mBuilder;
74
75     public Director(Builder builder) {
76         mBuilder = builder;
77     }
78
79     public Bike construct() {
80         mBuilder.buildFrame();
81         mBuilder.buildSeat();
82         return mBuilder.createBike();
83     }
84 }
85
86 //测试类
87 public class Client {
88     public static void main(String[] args) {
89         showBike(new OfoBuilder());
90         showBike(new MobikeBuilder());
91     }
92     private static void showBike(Builder builder) {
93         Director director = new Director(builder);
94         Bike bike = director.construct();

```

```
95     System.out.println(bike.getFrame());
96     System.out.println(bike.getSeat());
97 }
98 }
```

#### 注意:

上面示例是 Builder 模式的常规用法，指挥者类 Director 在建造者模式中具有很重要的作用，它用于指导具体构建者如何构建产品，控制调用先后次序，并向调用者返回完整的产品类，但是有些情况下需要简化系统结构，可以把指挥者类和抽象建造者进行结合

```
1 // 抽象 builder 类
2 public abstract class Builder {
3
4     protected Bike mBike = new Bike();
5
6     public abstract void buildFrame();
7     public abstract void buildSeat();
8     public abstract Bike createBike();
9
10    public Bike construct() {
11        this.buildFrame();
12        this.BuildSeat();
13        return this.createBike();
14    }
15 }
```

#### 说明:

这样做确实简化了系统结构，但同时也加重了抽象建造者类的职责，也不是太符合单一职责原则，如果 construct() 过于复杂，建议还是封装到 Director 中。

### 4.4.4 优缺点

#### 优点:

- 建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在指挥者类中对整体而言可以取得比较好的稳定性。
- 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。
- 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。
- 建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。符合开闭原则。

#### 缺点:

造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，则不适合使用建造者模式，因此其使用范围受到一定的限制。

### 4.4.5 使用场景

建造者 (Builder) 模式创建的是复杂对象，其产品的各个部分经常面临着剧烈的变化，但将它们组合在一起的算法却相对稳定，所以它通常在以下场合使用。

- 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。
- 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

#### 4.4.6 模式扩展

建造者模式除了上面的用途外，在开发中还有一个常用的使用方式，就是当一个类构造器需要传入很多参数时，如果创建这个类的实例，代码可读性会非常差，而且很容易引入错误，此时就可以利用建造者模式进行重构。

重构前代码如下：

```
1 public class Phone {
2     private String cpu;
3     private String screen;
4     private String memory;
5     private String mainboard;
6
7     public Phone(String cpu, String screen, String memory, String
mainboard) {
8         this.cpu = cpu;
9         this.screen = screen;
10        this.memory = memory;
11        this.mainboard = mainboard;
12    }
13
14    public String getCpu() {
15        return cpu;
16    }
17
18    public void setCpu(String cpu) {
19        this.cpu = cpu;
20    }
21
22    public String getScreen() {
23        return screen;
24    }
25
26    public void setScreen(String screen) {
27        this.screen = screen;
28    }
29
30    public String getMemory() {
31        return memory;
32    }
33
34    public void setMemory(String memory) {
35        this.memory = memory;
36    }
37
38    public String getMainboard() {
39        return mainboard;
40    }
41
42    public void setMainboard(String mainboard) {
```

```

43     this.mainboard = mainboard;
44 }
45
46 @Override
47 public String toString() {
48     return "Phone{" +
49         "cpu='" + cpu + '\'' +
50         ", screen='" + screen + '\'' +
51         ", memory='" + memory + '\'' +
52         ", mainboard='" + mainboard + '\'' +
53         '}';
54 }
55 }
56
57 public class Client {
58     public static void main(String[] args) {
59         //构建Phone对象
60         Phone phone = new Phone("intel", "三星屏幕", "金士顿", "华硕");
61         System.out.println(phone);
62     }
63 }

```

上面在客户端代码中构建Phone对象，传递了四个参数，如果参数更多呢？代码的可读性及使用的成本就是比较高。

重构后代码：

```

1  public class Phone {
2
3     private String cpu;
4     private String screen;
5     private String memory;
6     private String mainboard;
7
8     private Phone(Builder builder) {
9         cpu = builder.cpu;
10        screen = builder.screen;
11        memory = builder.memory;
12        mainboard = builder.mainboard;
13    }
14
15    public static final class Builder {
16        private String cpu;
17        private String screen;
18        private String memory;
19        private String mainboard;
20
21        public Builder() {}
22
23        public Builder cpu(String val) {
24            cpu = val;
25            return this;
26        }
27        public Builder screen(String val) {
28            screen = val;
29            return this;
30        }

```

```

31     public Builder memory(String val) {
32         memory = val;
33         return this;
34     }
35     public Builder mainboard(String val) {
36         mainboard = val;
37         return this;
38     }
39     public Phone build() {
40         return new Phone(this);}
41     }
42     @Override
43     public String toString() {
44         return "Phone{" +
45             "cpu='" + cpu + '\'' +
46             ", screen='" + screen + '\'' +
47             ", memory='" + memory + '\'' +
48             ", mainboard='" + mainboard + '\'' +
49             '}';
50     }
51 }
52
53 public class Client {
54     public static void main(String[] args) {
55         Phone phone = new Phone.Builder()
56             .cpu("intel")
57             .mainboard("华硕")
58             .memory("金士顿")
59             .screen("三星")
60             .build();
61         System.out.println(phone);
62     }
63 }

```

重构后的代码在使用起来更方便，某种程度上也可以提高开发效率。从软件设计上，对程序员的要求比较高。

## 4.6 创建者模式对比

### 4.6.1 工厂方法模式vs建造者模式

工厂方法模式注重的是整体对象的创建方式；而建造者模式注重的是部件构建的过程，意在通过一步一步地精确构造创建一个复杂的对象。

我们举个简单例子来说明两者的差异，如要制造一个超人，如果使用工厂方法模式，直接产生出来的就是一个力大无穷、能够飞翔、内裤外穿的超人；而如果使用建造者模式，则需要组装手、头、脚、躯干等部分，然后再把内裤外穿，于是一个超人就诞生了。

### 4.6.2 抽象工厂模式vs建造者模式

抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品：具有不同分类维度的产品组合，采用抽象工厂模式则是不需要关心构建过程，只关心什么产品由什么工厂生产即可。

建造者模式则是要求按照指定的蓝图建造产品，它的主要目的是通过组装零配件而产生一个新产品。



如果将抽象工厂模式看成汽车配件生产工厂，生产一个产品族的产品，那么建造者模式就是一个汽车组装工厂，通过对部件的组装可以返回一辆完整的汽车。

# 5, 结构型模式

---

结构型模式描述如何将类或对象按某种布局组成更大的结构。它分为类结构型模式和对象结构型模式，前者采用继承机制来组织接口和类，后者采用组合或聚合来组合对象。

由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象结构型模式比类结构型模式具有更大的灵活性。

结构型模式分为以下 7 种：

- 代理模式
- 适配器模式
- 装饰者模式
- 桥接模式
- 外观模式
- 组合模式
- 享元模式

## 5.1 代理模式

---

### 5.1.1 概述

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

Java中的代理按照代理类生成时机不同又分为静态代理和动态代理。静态代理代理类在编译期就生成，而动态代理代理类则是在Java运行时动态生成。动态代理又有JDK代理和CGLib代理两种。

### 5.1.2 结构

代理 (Proxy) 模式分为三种角色：

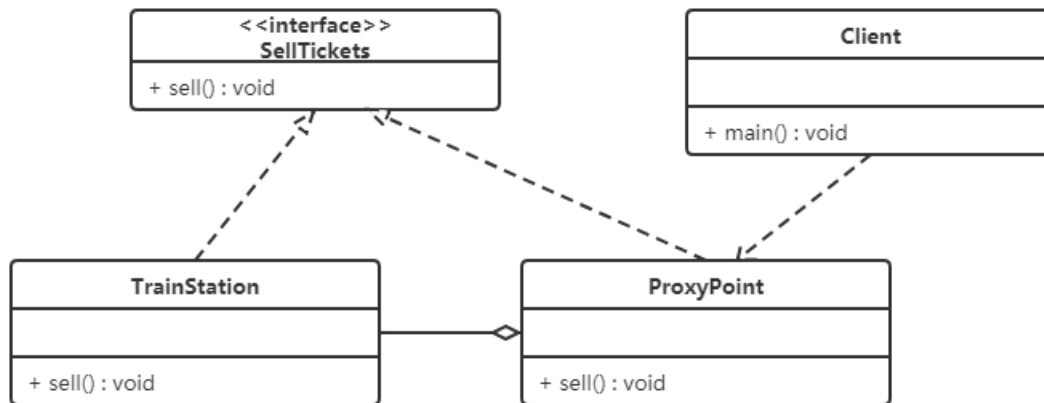
- 抽象主题 (Subject) 类： 通过接口或抽象类声明真实主题和代理对象实现的业务方法。
- 真实主题 (Real Subject) 类： 实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。
- 代理 (Proxy) 类： 提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

### 5.1.3 静态代理

我们通过案例来感受一下静态代理。

#### 【例】火车站卖票

如果要买火车票的话，需要去火车站买票，坐车到火车站，排队等一系列的操作，显然比较麻烦。而火车站在多个地方都有代售点，我们去代售点买票就方便很多了。这个例子其实就是典型的代理模式，火车站是目标对象，代售点是代理对象。类图如下：



代码如下:

```

1 //卖票接口
2 public interface SellTickets {
3     void sell();
4 }
5
6 //火车站 火车站具有卖票功能, 所以需要实现SellTickets接口
7 public class TrainStation implements SellTickets {
8
9     public void sell() {
10        System.out.println("火车站卖票");
11    }
12 }
13
14 //代售点
15 public class ProxyPoint implements SellTickets {
16
17     private TrainStation station = new TrainStation();
18
19     public void sell() {
20        System.out.println("代理点收取一些服务费用");
21        station.sell();
22    }
23 }
24
25 //测试类
26 public class Client {
27     public static void main(String[] args) {
28        ProxyPoint pp = new ProxyPoint();
29        pp.sell();
30    }
31 }
  
```

从上面代码中可以看出测试类直接访问的是ProxyPoint类对象, 也就是说ProxyPoint作为访问对象和目标对象的中介。同时也对sell方法进行了增强(代理点收取一些服务费用)。

#### 5.1.4 JDK动态代理

接下来我们使用动态代理实现上面案例, 先说说JDK提供的动态代理。Java中提供了一个动态代理类Proxy, Proxy并不是我们上述所说的代理对象的类, 而是提供了一个创建代理对象的静态方法(newProxyInstance方法)来获取代理对象。

代码如下:

```
1 //卖票接口
2 public interface SellTickets {
3     void sell();
4 }
5
6 //火车站 火车站具有卖票功能,所以需要实现SellTickets接口
7 public class TrainStation implements SellTickets {
8
9     public void sell() {
10         System.out.println("火车站卖票");
11     }
12 }
13
14 //代理工厂,用来创建代理对象
15 public class ProxyFactory {
16
17     private TrainStation station = new TrainStation();
18
19     public SellTickets getProxyObject() {
20         //使用Proxy获取代理对象
21         /*
22             newProxyInstance()方法参数说明:
23             ClassLoader loader : 类加载器,用于加载代理类,使用真实对象的类加载
器即可
24             Class<?>[] interfaces : 真实对象所实现的接口,代理模式真实对象和代
理对象实现相同的接口
25             InvocationHandler h : 代理对象的调用处理程序
26         */
27         SellTickets sellTickets = (SellTickets)
Proxy.newProxyInstance(station.getClass().getClassLoader(),
28             station.getClass().getInterfaces(),
29             new InvocationHandler() {
30                 /*
31                     InvocationHandler中invoke方法参数说明:
32                     proxy : 代理对象
33                     method : 对应于在代理对象上调用的接口方法的 Method 实
例
34                     args : 代理对象调用接口方法时传递的实际参数
35                 */
36                 public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
37
38                     System.out.println("代理点收取一些服务费用(JDK动态代理方
式)");
39
40                     //执行真实对象
41                     Object result = method.invoke(station, args);
42                     return result;
43                 }
44             });
45         return sellTickets;
46     }
47
48 //测试类
49 public class Client {
```

```

50     public static void main(String[] args) {
51         //获取代理对象
52         ProxyFactory factory = new ProxyFactory();
53
54         SellTickets proxyObject = factory.getProxyObject();
55         proxyObject.sell();
56     }
57 }

```

使用了动态代理，我们思考下面问题：

- ProxyFactory是代理类吗？

ProxyFactory不是代理模式中所说的代理类，而代理类是程序在运行过程中动态的在内存中生成的类。通过阿里巴巴开源的 Java 诊断工具 (Arthas【阿尔萨斯】) 查看代理类的结构：

```

1  package com.sun.proxy;
2
3  import com.itheima.proxy.dynamic.jdk.SellTickets;
4  import java.lang.reflect.InvocationHandler;
5  import java.lang.reflect.Method;
6  import java.lang.reflect.Proxy;
7  import java.lang.reflect.UndeclaredThrowableException;
8
9  public final class $Proxy0 extends Proxy implements SellTickets {
10     private static Method m1;
11     private static Method m2;
12     private static Method m3;
13     private static Method m0;
14
15     public $Proxy0(InvocationHandler invocationHandler) {
16         super(invocationHandler);
17     }
18
19     static {
20         try {
21             m1 = Class.forName("java.lang.Object").getMethod("equals",
22             Class.forName("java.lang.Object"));
23             m2 =
24             Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
25             m3 =
26             Class.forName("com.itheima.proxy.dynamic.jdk.SellTickets").getMethod("s
27             ell", new Class[0]);
28             m0 =
29             Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
30             return;
31         }
32         catch (NoSuchMethodException noSuchMethodException) {
33             throw new
34             NoSuchMethodError(noSuchMethodException.getMessage());
35         }
36         catch (ClassNotFoundException classNotFoundException) {
37             throw new
38             NoClassDefFoundError(classNotFoundException.getMessage());
39         }
40     }
41
42     public final boolean equals(Object object) {

```

```

36     try {
37         return (Boolean)this.h.invoke(this, m1, new Object[]
{object});
38     }
39     catch (Error | RuntimeException throwable) {
40         throw throwable;
41     }
42     catch (Throwable throwable) {
43         throw new UndeclaredThrowableException(throwable);
44     }
45 }
46
47 public final String toString() {
48     try {
49         return (String)this.h.invoke(this, m2, null);
50     }
51     catch (Error | RuntimeException throwable) {
52         throw throwable;
53     }
54     catch (Throwable throwable) {
55         throw new UndeclaredThrowableException(throwable);
56     }
57 }
58
59 public final int hashCode() {
60     try {
61         return (Integer)this.h.invoke(this, m0, null);
62     }
63     catch (Error | RuntimeException throwable) {
64         throw throwable;
65     }
66     catch (Throwable throwable) {
67         throw new UndeclaredThrowableException(throwable);
68     }
69 }
70
71 public final void sell() {
72     try {
73         this.h.invoke(this, m3, null);
74         return;
75     }
76     catch (Error | RuntimeException throwable) {
77         throw throwable;
78     }
79     catch (Throwable throwable) {
80         throw new UndeclaredThrowableException(throwable);
81     }
82 }
83 }

```

从上面的类中，我们可以看到以下几个信息：

- 代理类 (\$Proxy0) 实现了 SellTickets。这也就印证了我们之前说的真实类和代理类实现同样的接口。
  - 代理类 (\$Proxy0) 将我们提供的匿名内部类对象传递给了父类。
- 动态代理的执行流程是什么样？

下面是摘取的重点代码：

```

1 //程序运行过程中动态生成的代理类
2 public final class $Proxy0 extends Proxy implements SellTickets {
3     private static Method m3;
4
5     public $Proxy0(InvocationHandler invocationHandler) {
6         super(invocationHandler);
7     }
8
9     static {
10         m3 =
11         Class.forName("com.itheima.proxy.dynamic.jdk.SellTickets").getMethod("s
12         ell", new Class[0]);
13     }
14
15     public final void sell() {
16         this.h.invoke(this, m3, null);
17     }
18 }
19 //Java提供的动态代理相关类
20 public class Proxy implements java.io.Serializable {
21     protected InvocationHandler h;
22
23     protected Proxy(InvocationHandler h) {
24         this.h = h;
25     }
26 }
27 //代理工厂类
28 public class ProxyFactory {
29
30     private TrainStation station = new TrainStation();
31
32     public SellTickets getProxyObject() {
33         SellTickets sellTickets = (SellTickets)
34         Proxy.newProxyInstance(station.getClass().getClassLoader(),
35         station.getClass().getInterfaces(),
36         new InvocationHandler() {
37
38             public Object invoke(Object proxy, Method method,
39             Object[] args) throws Throwable {
40
41                 System.out.println("代理点收取一些服务费用(JDK动态代
42                 理方式)");
43
44                 Object result = method.invoke(station, args);
45                 return result;
46             }
47         });
48     }
49     return sellTickets;
50 }
51 }
52 //测试访问类
53 public class Client {
54     public static void main(String[] args) {
55         //获取代理对象

```

```

53     ProxyFactory factory = new ProxyFactory();
54     SellTickets proxyObject = factory.getProxyObject();
55     proxyObject.sell();
56     }
57 }

```

执行流程如下：

1. 在测试类中通过代理对象调用sell()方法
2. 根据多态的特性，执行的是代理类 (\$Proxy0) 中的sell()方法
3. 代理类 (\$Proxy0) 中的sell()方法中又调用了InvocationHandler接口的子实现类对象的invoke方法
4. invoke方法通过反射执行了真实对象所属类(TrainStation)中的sell()方法

### 5.1.5 CGLIB动态代理

同样是上面的案例，我们再次使用CGLIB代理实现。

如果没有定义SellTickets接口，只定义了TrainStation(火车站类)。很显然JDK代理是无法使用了，因为JDK动态代理要求必须定义接口，对接口进行代理。

CGLIB是一个功能强大，高性能的代码生成包。它为没有实现接口的类提供代理，为JDK的动态代理提供了很好的补充。

CGLIB是第三方提供的包，所以需要引入jar包的坐标：

```

1 <dependency>
2   <groupId>cglib</groupId>
3   <artifactId>cglib</artifactId>
4   <version>2.2.2</version>
5 </dependency>

```

代码如下：

```

1 //火车站
2 public class TrainStation {
3
4     public void sell() {
5         System.out.println("火车站卖票");
6     }
7 }
8
9 //代理工厂
10 public class ProxyFactory implements MethodInterceptor {
11
12     private TrainStation target = new TrainStation();
13
14     public TrainStation getProxyObject() {
15         //创建Enhancer对象，类似于JDK动态代理的Proxy类，下一步就是设置几个参数
16         Enhancer enhancer = new Enhancer();
17         //设置父类的字节码对象
18         enhancer.setSuperclass(target.getClass());
19         //设置回调函数
20         enhancer.setCallback(this);
21         //创建代理对象
22         TrainStation obj = (TrainStation) enhancer.create();
23         return obj;

```



```

24     }
25
26     /*
27         intercept方法参数说明:
28         o : 代理对象
29         method : 真实对象中的方法的Method实例
30         args : 实际参数
31         methodProxy : 代理对象中的方法的method实例
32     */
33     public TrainStation intercept(Object o, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
34         System.out.println("代理点收取一些服务费用(CGLIB动态代理方式)");
35         TrainStation result = (TrainStation) methodProxy.invokeSuper(o,
args);
36         return result;
37     }
38 }
39
40 //测试类
41 public class Client {
42     public static void main(String[] args) {
43         //创建代理工厂对象
44         ProxyFactory factory = new ProxyFactory();
45         //获取代理对象
46         TrainStation proxyObject = factory.getProxyObject();
47
48         proxyObject.sell();
49     }
50 }

```

### 5.1.6 三种代理的对比

- jdk代理和CGLIB代理

使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，在JDK1.6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的类或者方法进行代理，因为CGLib原理是动态生成被代理类的子类。

在JDK1.6、JDK1.7、JDK1.8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLib代理效率，只有当进行大量调用的时候，JDK1.6和JDK1.7比CGLib代理效率低一点，但是到JDK1.8的时候，JDK代理效率高于CGLib代理。所以如果有接口使用JDK动态代理，如果没有接口使用CGLIB代理。

- 动态代理和静态代理

动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口方法数量比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样每一个方法进行中转。

如果接口增加一个方法，静态代理模式除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。而动态代理不会出现该问题

### 5.1.7 优缺点

#### 优点:

- 代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用;

- 代理对象可以扩展目标对象的功能;
- 代理模式能将客户端与目标对象分离,在一定程度上降低了系统的耦合度;

缺点:

- 增加了系统的复杂度;

## 5.1.8 使用场景

- 远程 (Remote) 代理

本地服务通过网络请求远程服务。为了实现本地到远程的通信,我们需要实现网络通信,处理其中可能的异常。为良好的代码设计和可维护性,我们将网络通信部分隐藏起来,只暴露给本地服务一个接口,通过该接口即可访问远程服务提供的功能,而不必过多关心通信部分的细节。

- 防火墙 (Firewall) 代理

当你将浏览器配置成使用代理功能时,防火墙就将你的浏览器的请求转给互联网;当互联网返回响应时,代理服务器再把它转给你的浏览器。

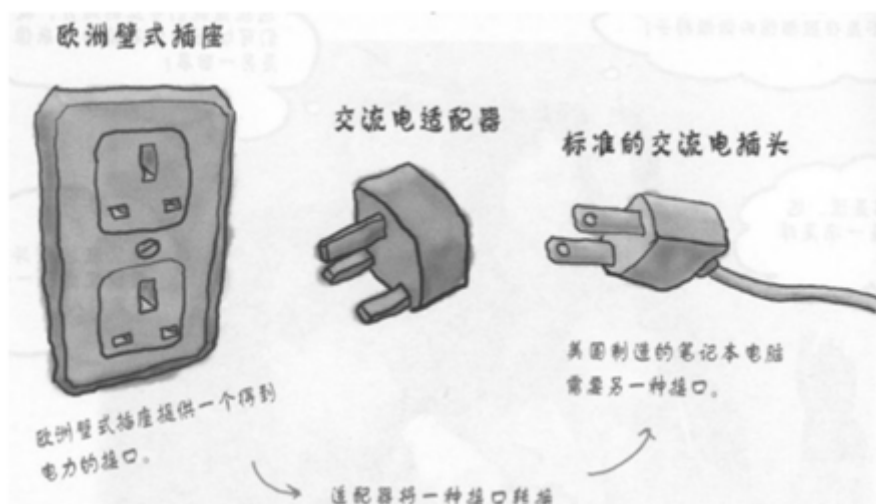
- 保护 (Protect or Access) 代理

控制对一个对象的访问,如果需要,可以给不同的用户提供不同级别的使用权限。

## 5.2 适配器模式

### 5.2.1 概述

如果去欧洲国家去旅游的话,他们的插座如下图最左边,是欧洲标准。而我们使用的插头如下图最右边的。因此我们的笔记本电脑,手机在当地不能直接充电。所以就需要一个插座转换器,转换器第1面插入当地的插座,第2面供我们充电,这样使得我们的插头在当地能使用。生活中这样的例子很多,手机充电器(将220v转换为5v的电压),读卡器等,其实就是使用到了适配器模式。



定义:

将一个类的接口转换成客户希望的另外一个接口,使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

适配器模式分为类适配器模式和对象适配器模式,前者类之间的耦合度比后者高,且要求程序员了解现有组件库中的相关组件的内部结构,所以应用相对较少些。

### 5.2.2 结构

适配器模式 (Adapter) 包含以下主要角色:

- 目标 (Target) 接口: 当前系统业务所期待的接口, 它可以是抽象类或接口。
- 适配者 (Adaptee) 类: 它是被访问和适配的现存组件库中的组件接口。
- 适配器 (Adapter) 类: 它是一个转换器, 通过继承或引用适配者的对象, 把适配者接口转换成目标接口, 让客户按目标接口的格式访问适配者。

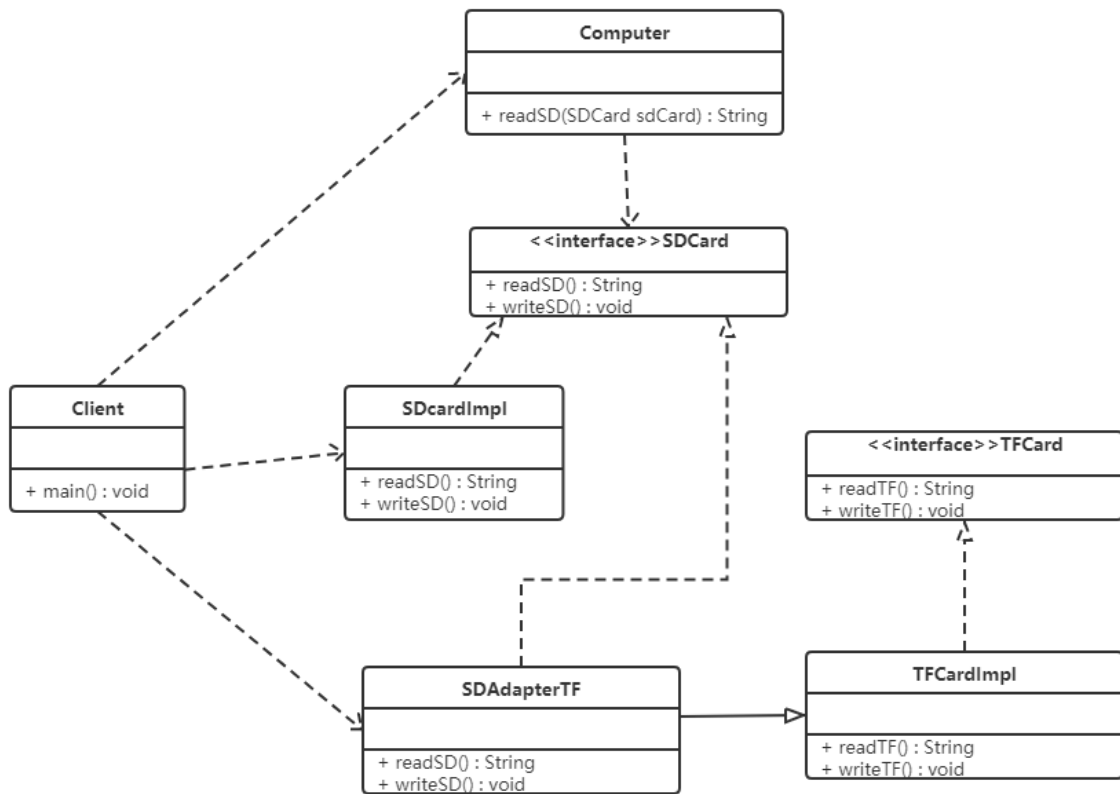
### 5.2.3 类适配器模式

实现方式: 定义一个适配器类来实现当前系统的业务接口, 同时又继承现有组件库中已经存在的组件。

#### 【例】读卡器

现有一台电脑只能读取SD卡, 而要读取TF卡中的内容的话就需要使用到适配器模式。创建一个读卡器, 将TF卡中的内容读取出来。

类图如下:



代码如下:

```
1 //SD卡的接口
2 public interface SDCard {
3     //读取SD卡方法
4     String readSD();
5     //写入SD卡功能
6     void writeSD(String msg);
7 }
8
9 //SD卡实现类
10 public class SDCardImpl implements SDCard {
11     public String readSD() {
12         String msg = "sd card read a msg :hello word SD";
13         return msg;
14     }
15 }
```

```

15
16     public void writeSD(String msg) {
17         System.out.println("sd card write msg : " + msg);
18     }
19 }
20
21 //电脑类
22 public class Computer {
23
24     public String readSD(SDCard sdCard) {
25         if(sdCard == null) {
26             throw new NullPointerException("sd card null");
27         }
28         return sdCard.readSD();
29     }
30 }
31
32 //TF卡接口
33 public interface TFCard {
34     //读取TF卡方法
35     String readTF();
36     //写入TF卡功能
37     void writeTF(String msg);
38 }
39
40 //TF卡实现类
41 public class TFCardImpl implements TFCard {
42
43     public String readTF() {
44         String msg ="tf card read msg : hello word tf card";
45         return msg;
46     }
47
48     public void writeTF(String msg) {
49         System.out.println("tf card write a msg : " + msg);
50     }
51 }
52
53 //定义适配器类 (SD兼容TF)
54 public class SDAdapterTF extends TFCardImpl implements SDCard {
55
56     public String readSD() {
57         System.out.println("adapter read tf card ");
58         return readTF();
59     }
60
61     public void writeSD(String msg) {
62         System.out.println("adapter write tf card");
63         writeTF(msg);
64     }
65 }
66
67 //测试类
68 public class Client {
69     public static void main(String[] args) {
70         Computer computer = new Computer();
71         SDCard sdCard = new SDCardImpl();
72         System.out.println(computer.readSD(sdCard));

```

```

73
74     System.out.println("-----");
75
76     SDAdapterTF adapter = new SDAdapterTF();
77     System.out.println(computer.readSD(adapter));
78 }
79 }

```

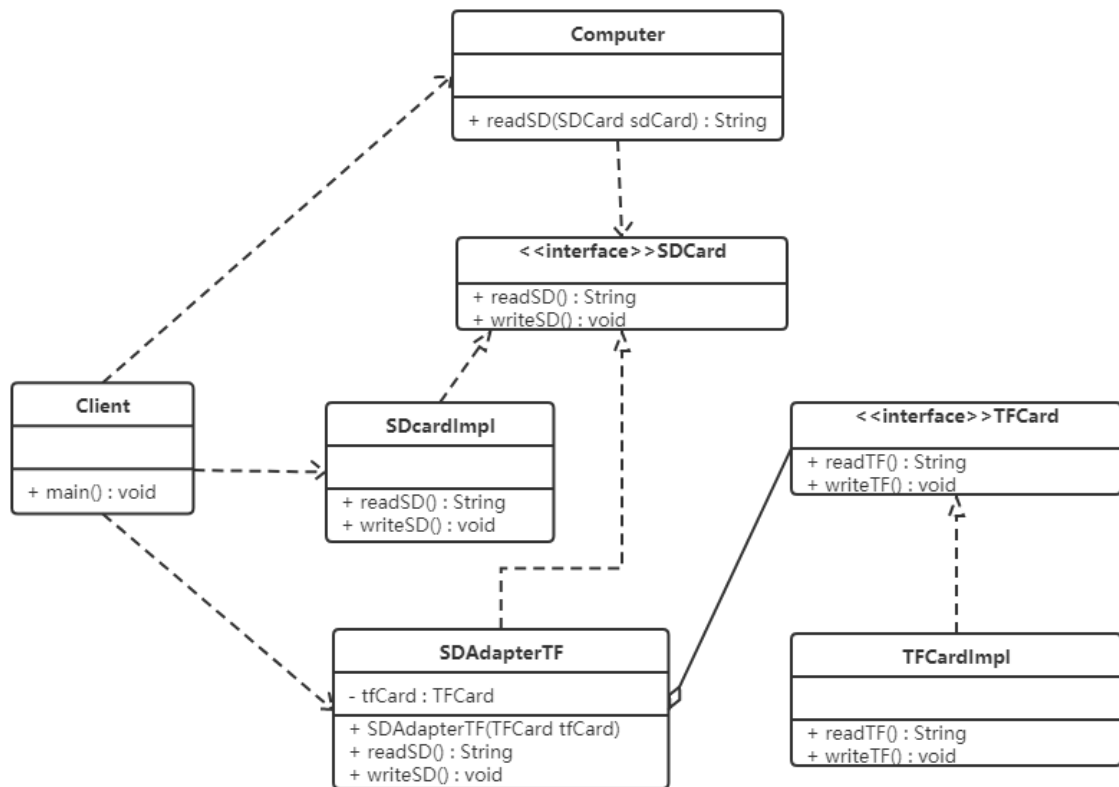
类适配器模式违背了合成复用原则。类适配器是客户类有一个接口规范的情况下可用，反之不可用。

## 5.2.4 对象适配器模式

实现方式：对象适配器模式可采用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。

【例】读卡器

我们使用对象适配器模式将读卡器的案例进行改写。类图如下：



代码如下：

类适配器模式的代码，我们只需要修改适配器类（`SDAdapterTF`）和测试类。

```

1 //创建适配器对象（SD兼容TF）
2 public class SDAdapterTF implements SDCard {
3
4     private TFCard tfCard;
5
6     public SDAdapterTF(TFCard tfCard) {
7         this.tfCard = tfCard;
8     }
9

```

```

10     public String readSD() {
11         System.out.println("adapter read tf card ");
12         return tfCard.readTF();
13     }
14
15     public void writeSD(String msg) {
16         System.out.println("adapter write tf card");
17         tfCard.writeTF(msg);
18     }
19 }
20
21 //测试类
22 public class Client {
23     public static void main(String[] args) {
24         Computer computer = new Computer();
25         SDCard sdCard = new SDCardImpl();
26         System.out.println(computer.readSD(sdCard));
27
28         System.out.println("-----");
29
30         TFCard tfCard = new TFCardImpl();
31         SDAdapterTF adapter = new SDAdapterTF(tfCard);
32         System.out.println(computer.readSD(adapter));
33     }
34 }

```

注意：还有一个适配器模式是接口适配器模式。当不希望实现一个接口中所有的方法时，可以创建一个抽象类`Adapter`，实现所有方法。而此时我们只需要继承该抽象类即可。

### 5.2.5 应用场景

- 以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。
- 使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

### 5.2.6 JDK源码解析

`Reader`（字符流）、`InputStream`（字节流）的适配使用的是`InputStreamReader`。

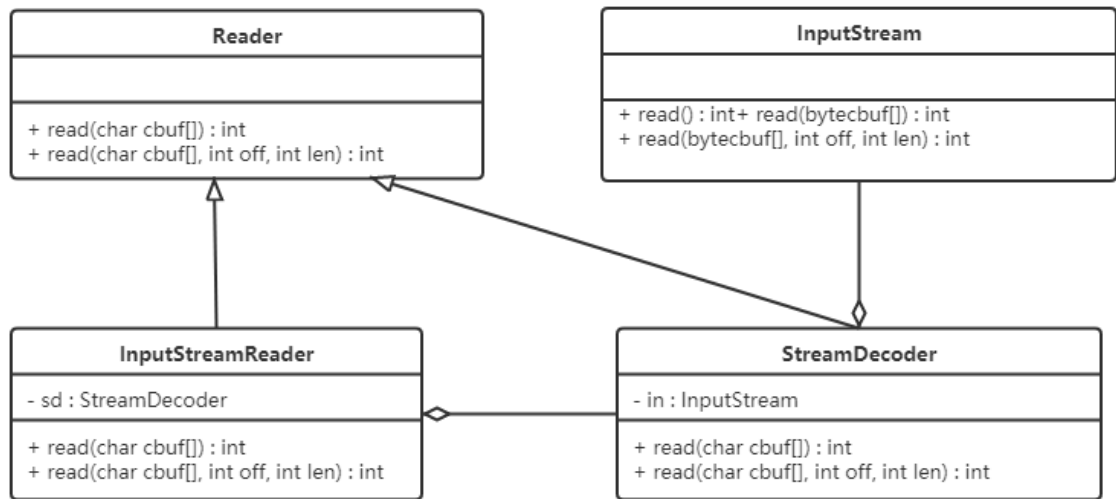
`InputStreamReader`继承自`java.io`包中的`Reader`，对他中的抽象的未实现的方法给出实现。如：

```

1     public int read() throws IOException {
2         return sd.read();
3     }
4
5     public int read(char cbuf[], int offset, int length) throws IOException {
6         return sd.read(cbuf, offset, length);
7     }

```

如上代码中的`sd`（`StreamDecoder`类对象），在Sun的JDK实现中，实际的方法实现是对`sun.nio.cs.StreamDecoder`类的同名方法的调用封装。类结构图如下：



从上图可以看出：

- `InputStreamReader`是对同样实现了`Reader`的`StreamDecoder`的封装。
- `StreamDecoder`不是Java SE API中的内容，是Sun JDK给出的自身实现。但我们知道他们对构造方法中的字节流类（`InputStream`）进行封装，并通过该类进行了字节流和字符流之间的解码转换。

**结论：**

从表层来看，`InputStreamReader`做了`InputStream`字节流类到`Reader`字符流之间的转换。而从如上Sun JDK中的实现类关系结构中可以看出，是`StreamDecoder`的设计实现在实际上采用了适配器模式。

## 5.3 装饰者模式

### 5.3.1 概述

我们先来看一个快餐店的例子。

快餐店有炒面、炒饭这些快餐，可以额外附加鸡蛋、火腿、培根这些配菜，当然加配菜需要额外加钱，每个配菜的价钱通常不太一样，那么计算总价就会显得比较麻烦。

使用继承的方式存在的问题：

- 扩展性不好  
如果要再加一种配料（火腿肠），我们就会发现需要给`FriedRice`和`FriedNoodles`分别定义一个子类。如果要新增一个快餐品类（炒河粉）的话，就需要定义更多的子类。
- 产生过多的子类

**定义：**

指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式。

### 5.3.2 结构

装饰（Decorator）模式中的角色：

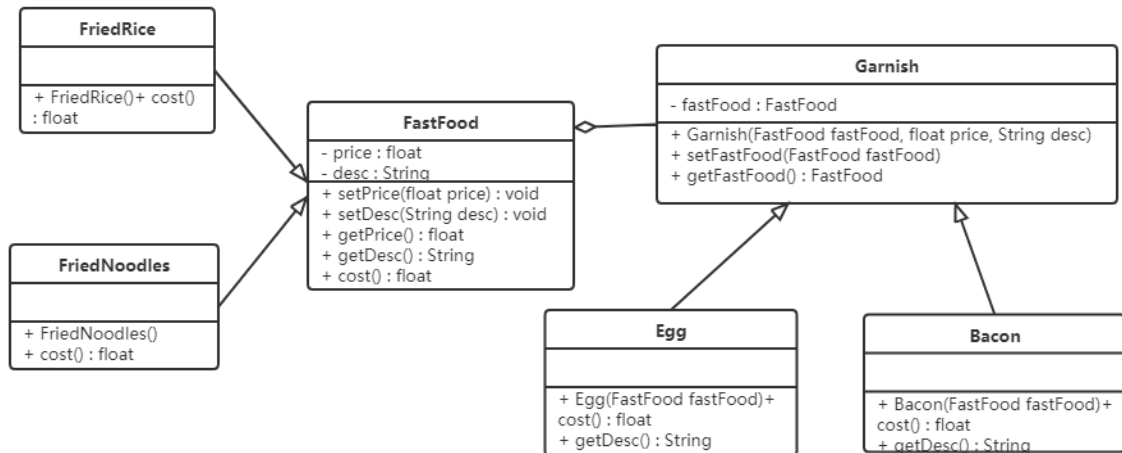
- 抽象构件（Component）角色：定义一个抽象接口以规范准备接收附加责任的对象。
- 具体构件（Concrete Component）角色：实现抽象构件，通过装饰角色为其添加一些职责。

- 抽象装饰 (Decorator) 角色 : 继承或实现抽象构件, 并包含具体构件的实例, 可以通过其子类扩展具体构件的功能。
- 具体装饰 (ConcreteDecorator) 角色 : 实现抽象装饰的相关方法, 并给具体构件对象添加附加的责任。

### 5.3.3 案例

我们使用装饰者模式对快餐店案例进行改进, 体会装饰者模式的精髓。

类图如下:



代码如下:

```

1 //快餐接口
2 public abstract class FastFood {
3     private float price;
4     private String desc;
5
6     public FastFood() {
7     }
8
9     public FastFood(float price, String desc) {
10        this.price = price;
11        this.desc = desc;
12    }
13
14    public void setPrice(float price) {
15        this.price = price;
16    }
17
18    public float getPrice() {
19        return price;
20    }
21
22    public String getDesc() {
23        return desc;
24    }
25
26    public void setDesc(String desc) {
27        this.desc = desc;
28    }
  
```



```
29
30     public abstract float cost(); //获取价格
31 }
32
33 //炒饭
34 public class FriedRice extends FastFood {
35
36     public FriedRice() {
37         super(10, "炒饭");
38     }
39
40     public float cost() {
41         return getPrice();
42     }
43 }
44
45 //炒面
46 public class FriedNoodles extends FastFood {
47
48     public FriedNoodles() {
49         super(12, "炒面");
50     }
51
52     public float cost() {
53         return getPrice();
54     }
55 }
56
57 //配料类
58 public abstract class Garnish extends FastFood {
59
60     private FastFood fastFood;
61
62     public FastFood getFastFood() {
63         return fastFood;
64     }
65
66     public void setFastFood(FastFood fastFood) {
67         this.fastFood = fastFood;
68     }
69
70     public Garnish(FastFood fastFood, float price, String desc) {
71         super(price, desc);
72         this.fastFood = fastFood;
73     }
74 }
75
76 //鸡蛋配料
77 public class Egg extends Garnish {
78
79     public Egg(FastFood fastFood) {
80         super(fastFood, 1, "鸡蛋");
81     }
82
83     public float cost() {
84         return getPrice() + getFastFood().getPrice();
85     }
86 }
```

```

87     @Override
88     public String getDesc() {
89         return super.getDesc() + getFastFood().getDesc();
90     }
91 }
92
93 //培根配料
94 public class Bacon extends Garnish {
95
96     public Bacon(FastFood fastFood) {
97
98         super(fastFood,2,"培根");
99     }
100
101     @Override
102     public float cost() {
103         return getPrice() + getFastFood().getPrice();
104     }
105
106     @Override
107     public String getDesc() {
108         return super.getDesc() + getFastFood().getDesc();
109     }
110 }
111
112 //测试类
113 public class Client {
114     public static void main(String[] args) {
115         //点一份炒饭
116         FastFood food = new FriedRice();
117         //花费的价格
118         System.out.println(food.getDesc() + " " + food.cost() + "元");
119
120         System.out.println("=====");
121         //点一份加鸡蛋的炒饭
122         FastFood food1 = new FriedRice();
123
124         food1 = new Egg(food1);
125         //花费的价格
126         System.out.println(food1.getDesc() + " " + food1.cost() + "元");
127
128         System.out.println("=====");
129         //点一份加培根的炒面
130         FastFood food2 = new FriedNoodles();
131         food2 = new Bacon(food2);
132         //花费的价格
133         System.out.println(food2.getDesc() + " " + food2.cost() + "元");
134     }
135 }

```

### 好处:

- 饰者模式可以带来比继承更加灵活性的扩展功能，使用更加方便，可以通过组合不同的装饰者对象来获取具有不同行为状态的多样化的结果。装饰者模式比继承更具良好的扩展性，完美的遵循开闭原则，继承是静态的附加责任，装饰者则是动态的附加责任。
- 装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

### 5.3.4 使用场景

- 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。

不能采用继承的情况主要有两类：

- 第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；
- 第二类是因为类定义不能继承（如final类）
- 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。
- 当对象的功能要求可以动态地添加，也可以再动态地撤销时。

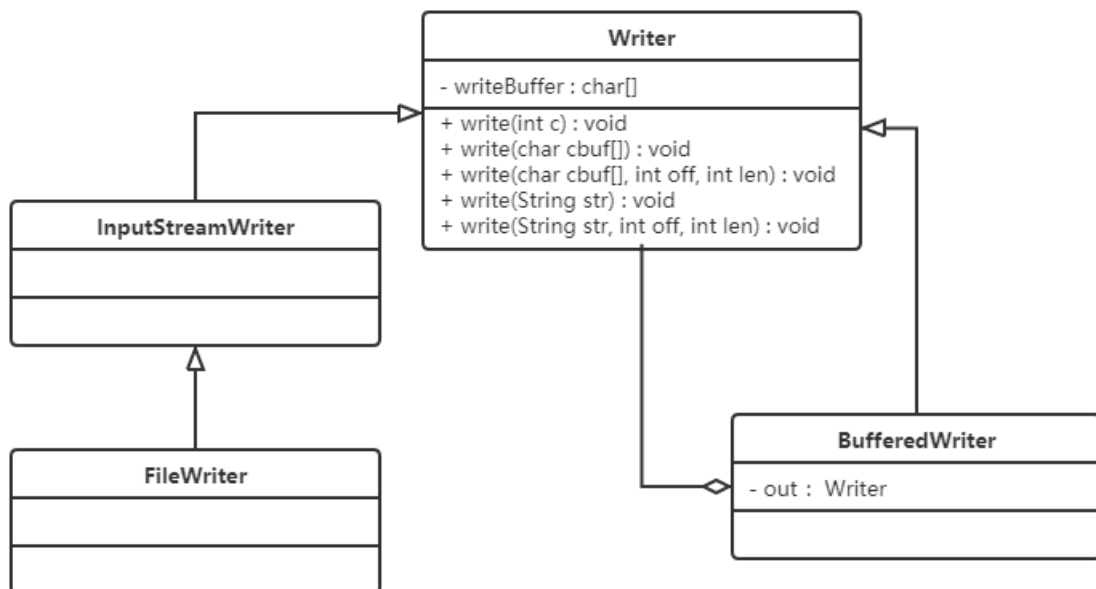
### 5.3.5 JDK源码解析

IO流中的包装类使用到了装饰者模式。BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter。

我们以BufferedWriter举例来说明，先看看如何使用BufferedWriter

```
1 public class Demo {
2     public static void main(String[] args) throws Exception{
3         //创建Bufferedwriter对象
4         //创建Filewriter对象
5         FileWriter fw = new FileWriter("C:\\Users\\Think\\Desktop\\a.txt");
6         BufferedWriter bw = new BufferedWriter(fw);
7
8         //写数据
9         bw.write("hello Buffered");
10
11        bw.close();
12    }
13 }
```

使用起来感觉确实像是装饰者模式，接下来看它们的结构：



## 小结:

BufferedWriter使用装饰者模式对Writer子实现类进行了增强，添加了缓冲区，提高了写数据的效率。

### 5.3.6 代理和装饰者的区别

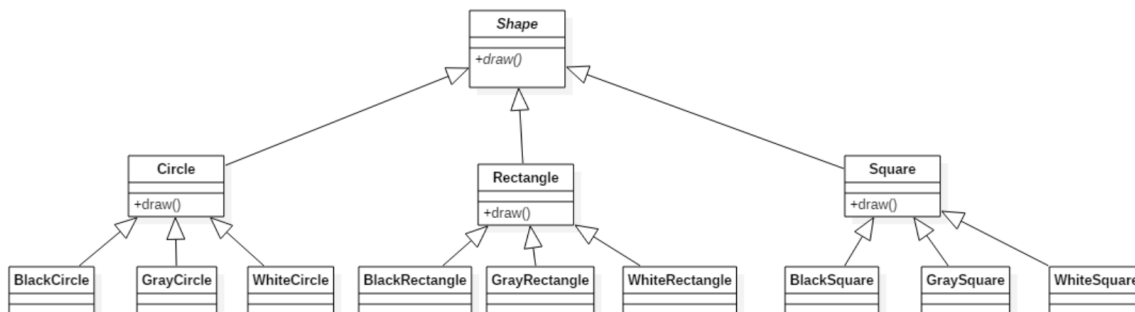
静态代理和装饰者模式的区别:

- 相同点:
  - 都要实现与目标类相同的业务接口
  - 在两个类中都要声明目标对象
  - 都可以在不修改目标类的前提下增强目标方法
- 不同点:
  - 目的不同 装饰者是为了增强目标对象 静态代理是为了保护和隐藏目标对象
  - 获取目标对象构建的地方不同 装饰者是由外界传递进来，可以通过构造方法传递 静态代理是在代理类内部创建，以此来隐藏目标对象

## 5.4 桥接模式

### 5.4.1 概述

现在有一个需求，需要创建不同的图形，并且每个图形都有可能会有不同的颜色。我们可以利用继承的方式来设计类的关系:



我们可以发现有很多类，假如我们再增加一个形状或再增加一种颜色，就需要创建更多的类。

试想，在一个有多种可能会变化的维度的系统中，用继承方式会造成类爆炸，扩展起来不灵活。每次在一个维度上新增一个具体实现都要增加多个子类。为了更加灵活的设计系统，我们此时可以考虑使用桥接模式。

#### 定义:

将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

### 5.4.2 结构

桥接 (Bridge) 模式包含以下主要角色:

- 抽象化 (Abstraction) 角色 : 定义抽象类，并包含一个对实现化对象的引用。

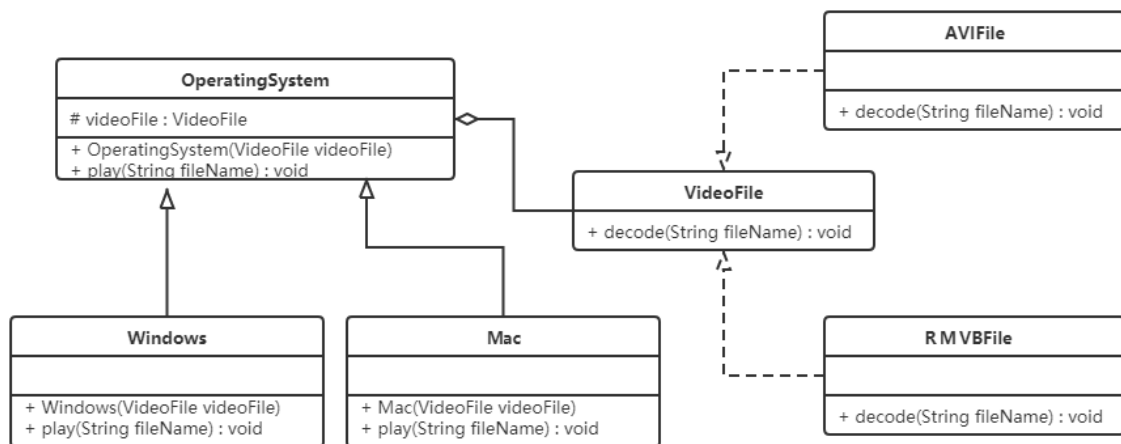
- 扩展抽象化 (Refined Abstraction) 角色：是抽象化角色的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法。
- 实现化 (Implementor) 角色：定义实现化角色的接口，供扩展抽象化角色调用。
- 具体实现化 (Concrete Implementor) 角色：给出实现化角色接口的具体实现。

### 5.4.3 案例

#### 【例】视频播放器

需要开发一个跨平台视频播放器，可以在不同操作系统平台（如Windows、Mac、Linux等）上播放多种格式的视频文件，常见的视频格式包括RMVB、AVI、WMV等。该播放器包含了两个维度，适合使用桥接模式。

类图如下：



代码如下：

```

1 //视频文件
2 public interface VideoFile {
3     void decode(String fileName);
4 }
5
6 //avi文件
7 public class AVIFile implements VideoFile {
8     public void decode(String fileName) {
9         System.out.println("avi视频文件: "+ fileName);
10    }
11 }
12
13 //rmvb文件
14 public class REVBFile implements VideoFile {
15
16     public void decode(String fileName) {
17         System.out.println("rmvb文件: " + fileName);
18     }
19 }
20
21 //操作系统版本
22 public abstract class OperatingSystemVersion {
23
24     protected VideoFile videoFile;
  
```

```

25
26     public OperatingSystemVersion(VideoFile videoFile) {
27         this.videoFile = videoFile;
28     }
29
30     public abstract void play(String fileName);
31 }
32
33 //windows版本
34 public class Windows extends OperatingSystem {
35
36     public windows(VideoFile videoFile) {
37         super(videoFile);
38     }
39
40     public void play(String fileName) {
41         videoFile.decode(fileName);
42     }
43 }
44
45 //mac版本
46 public class Mac extends OperatingSystemVersion {
47
48     public Mac(VideoFile videoFile) {
49         super(videoFile);
50     }
51
52     public void play(String fileName) {
53         videoFile.decode(fileName);
54     }
55 }
56
57 //测试类
58 public class Client {
59     public static void main(String[] args) {
60         OperatingSystem os = new Windows(new AVIFile());
61         os.play("战狼3");
62     }
63 }

```

#### 好处:

- 桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。  
如：如果现在还有一种视频文件类型wmv，我们只需要再定义一个类实现VideoFile接口即可，其他类不需要发生变化。
- 实现细节对客户透明

#### 5.4.4 使用场景

- 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时。
- 当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时。
- 当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时。避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。

## 5.5 外观模式

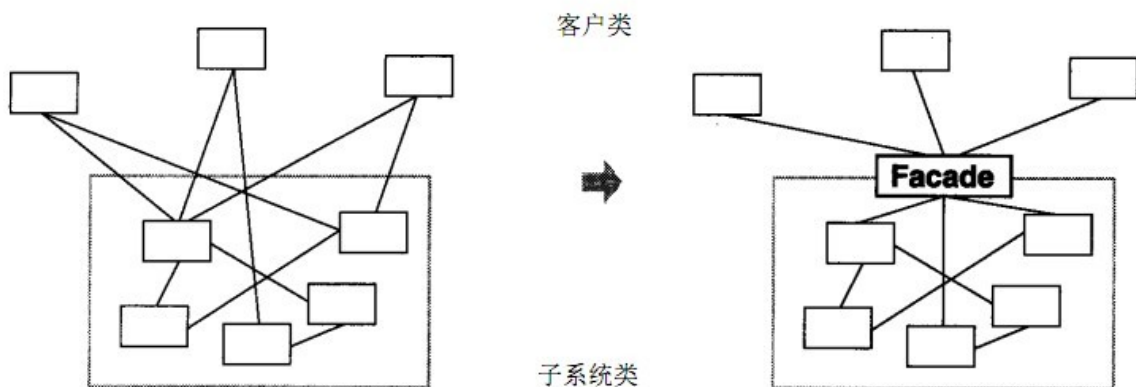
### 5.5.1 概述

有些人可能炒过股票，但其实大部分人都不太懂，这种没有足够了解证券知识的情况下做股票是很容易亏钱的，刚开始炒股肯定都会想，如果有个懂行的帮帮手就好，其实基金就是个好帮手，支付宝里就有许多的基金，它将投资者分散的资金集中起来，交由专业的经理人进行管理，投资于股票、债券、外汇等领域，而基金投资的收益归持有者所有，管理机构收取一定比例的托管管理费用。

**定义：**

又名门面模式，是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体的细节，这样会大大降低应用程序的复杂度，提高了程序的可维护性。

外观 (Facade) 模式是“迪米特法则”的典型应用



### 5.5.2 结构

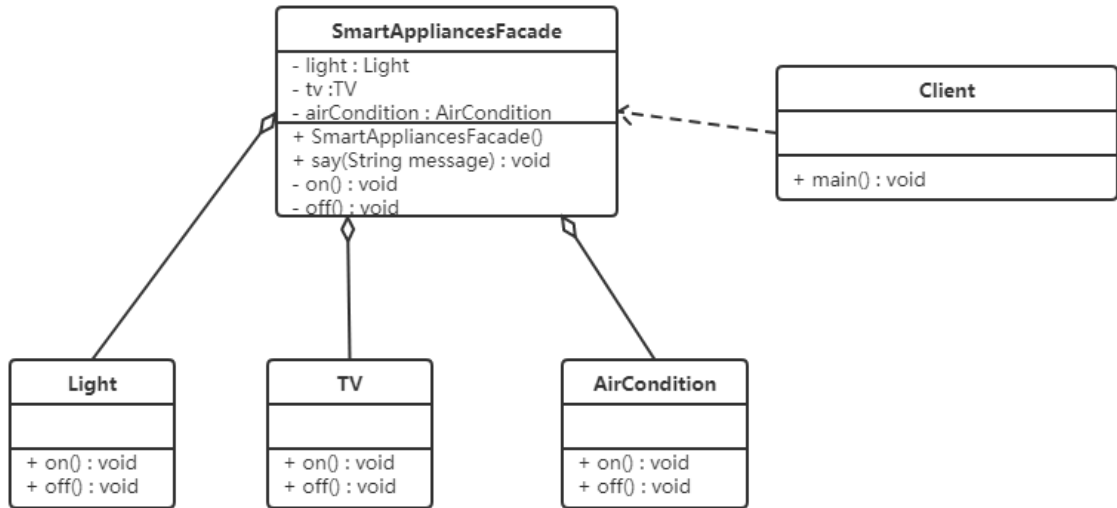
外观 (Facade) 模式包含以下主要角色：

- 外观 (Facade) 角色：为多个子系统对外提供一个共同的接口。
- 子系统 (Sub System) 角色：实现系统的部分功能，客户可以通过外观角色访问它。

### 5.5.3 案例

**【例】智能家电控制**

小明的爷爷已经60岁了，一个人在家生活：每次都需要打开灯、打开电视、打开空调；睡觉时关闭灯、关闭电视、关闭空调；操作起来都比较麻烦。所以小明给爷爷买了智能音箱，可以通过语音直接控制这些智能家电的开启和关闭。类图如下：



代码如下:

```

1 //灯类
2 public class Light {
3     public void on() {
4         System.out.println("打开了灯....");
5     }
6
7     public void off() {
8         System.out.println("关闭了灯....");
9     }
10 }
11
12 //电视类
13 public class TV {
14     public void on() {
15         System.out.println("打开了电视....");
16     }
17
18     public void off() {
19         System.out.println("关闭了电视....");
20     }
21 }
22
23 //控制类
24 public class AirCondition {
25     public void on() {
26         System.out.println("打开了空调....");
27     }
28
29     public void off() {
30         System.out.println("关闭了空调....");
31     }
32 }
33
34 //智能音箱
35 public class SmartAppliancesFacade {
36
37     private Light light;
38     private TV tv;
39     private AirCondition airCondition;

```



```

40
41     public SmartAppliancesFacade() {
42         light = new Light();
43         tv = new TV();
44         airCondition = new AirCondition();
45     }
46
47     public void say(String message) {
48         if(message.contains("打开")) {
49             on();
50         } else if(message.contains("关闭")) {
51             off();
52         } else {
53             System.out.println("我还听不懂你说的!!!");
54         }
55     }
56
57     //起床后一键开电器
58     private void on() {
59         System.out.println("起床了");
60         light.on();
61         tv.on();
62         airCondition.on();
63     }
64
65     //睡觉一键关电器
66     private void off() {
67         System.out.println("睡觉了");
68         light.off();
69         tv.off();
70         airCondition.off();
71     }
72 }
73
74 //测试类
75 public class Client {
76     public static void main(String[] args) {
77         //创建外观对象
78         SmartAppliancesFacade facade = new SmartAppliancesFacade();
79         //客户端直接与外观对象进行交互
80         facade.say("打开家电");
81         facade.say("关闭家电");
82     }
83 }

```

#### 好处:

- 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。
- 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。

#### 缺点:

- 不符合开闭原则，修改很麻烦

### 5.5.4 使用场景

- 对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系。
- 当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问。
- 当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

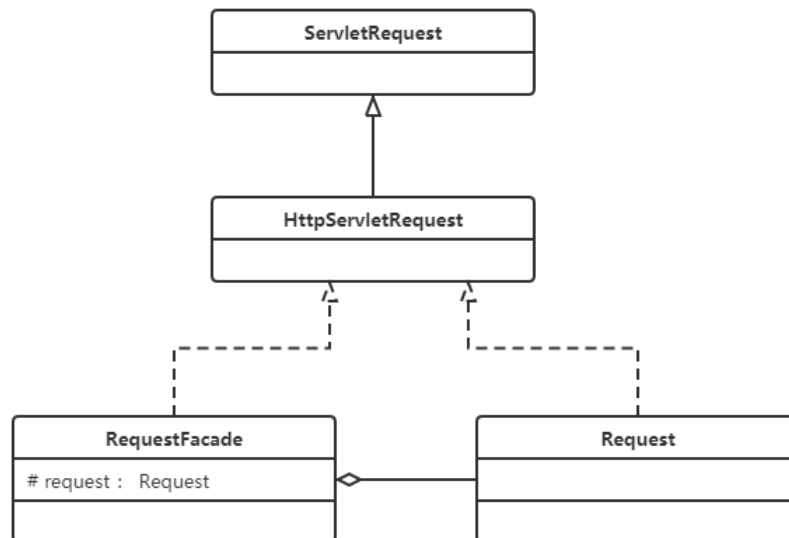
### 5.5.5 源码解析

使用tomcat作为web容器时，接收浏览器发送过来的请求，tomcat会将请求信息封装成ServletRequest对象，如下图①处对象。但是大家想想ServletRequest是一个接口，它还有一个子接口HttpServletRequest，而我们知道该request对象肯定是一个HttpServletRequest对象的子实现类对象，到底是哪个类的对象呢？可以通过输出request对象，我们就会发现是一个名为RequestFacade的类的对象。

```
public class ServletDemo extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        this.doPost(request, response);
    }
}
```

RequestFacade类就使用了外观模式。先看结构图：



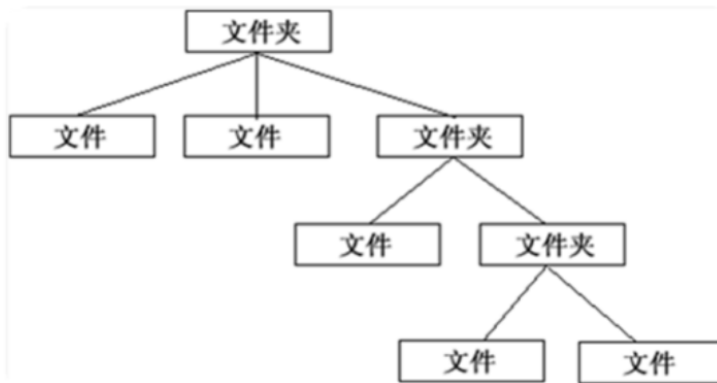
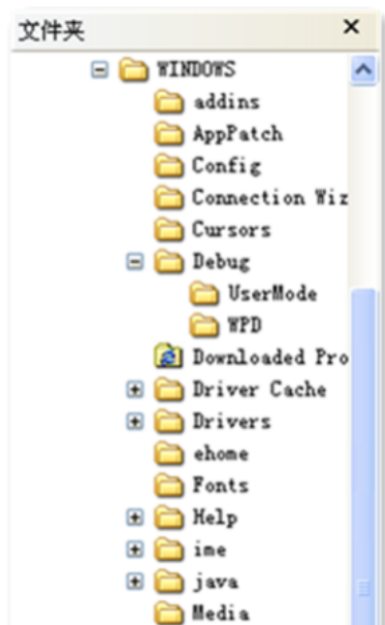
#### 为什么在此处使用外观模式呢？

定义 RequestFacade 类，分别实现 ServletRequest ，同时定义私有成员变量 Request ，并且方法的实现调用 Request 的实现。然后，将 RequestFacade上转为 ServletRequest 传给 servlet 的 service 方法，这样即使在 servlet 中被下转为 RequestFacade ，也不能访问私有成员变量对象中的方法。既用了 Request ，又能防止其中方法被不合理的访问。

# 5, 结构型模式

## 5.6 组合模式

### 5.6.1 概述



对于这个图片肯定会非常熟悉，上图我们可以看做是一个文件系统，对于这样的结构我们称之为树形结构。在树形结构中可以通过调用某个方法来遍历整个树，当我们找到某个叶子节点后，就可以对叶子节点进行相关的操作。可以将这颗树理解成一个大的容器，容器里面包含很多的成员对象，这些成员对象即可是容器对象也可以是叶子对象。但是由于容器对象和叶子对象在功能上面的区别，使得我们在使用的过程中必须要区分容器对象和叶子对象，但是这样就会给客户带来不必要的麻烦，作为客户而已，它始终希望能够一致的对对待容器对象和叶子对象。

#### 定义：

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

### 5.6.2 结构

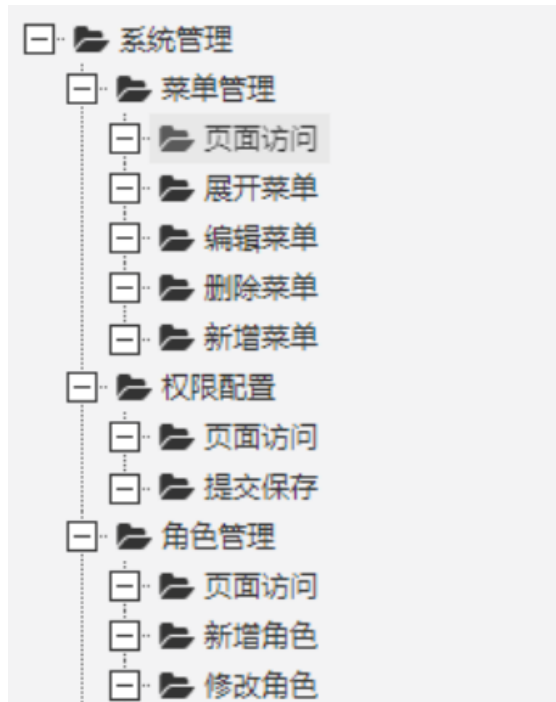
组合模式主要包含三种角色：

- 抽象根节点 (Component)：定义系统各层次对象的共有方法和属性，可以预先定义一些默认行为和属性。
- 树枝节点 (Composite)：定义树枝节点的行为，存储子节点，组合树枝节点和叶子节点形成一个树形结构。
- 叶子节点 (Leaf)：叶子节点对象，其下再无分支，是系统层次遍历的最小单位。

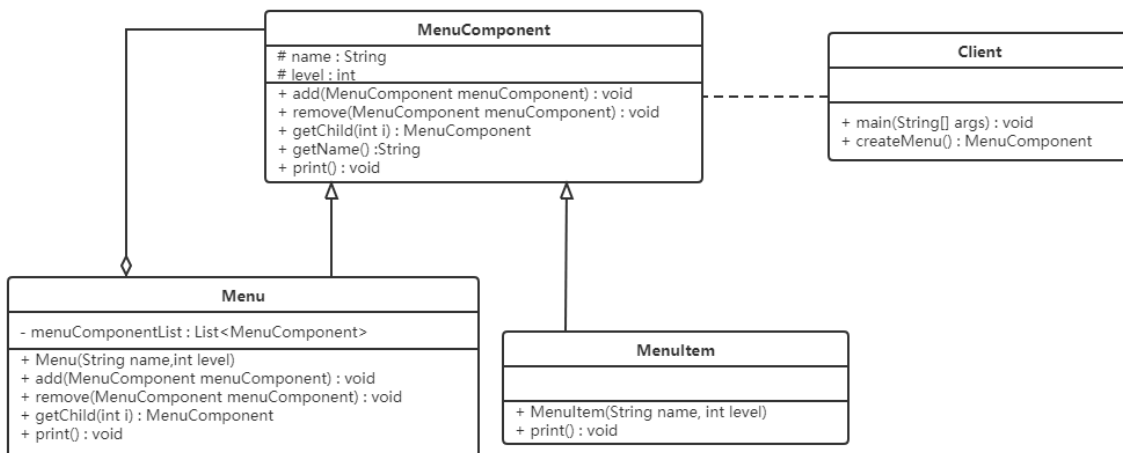
### 5.6.3 案例实现

【例】软件菜单

如下图，我们在访问别的一些管理系统时，经常可以看到类似的菜单。一个菜单可以包含菜单项（菜单项是指不再包含其他内容的菜单条目），也可以包含带有其他菜单项的菜单，因此使用组合模式描述菜单就很恰当，我们的需求是针对一个菜单，打印出其包含的所有菜单以及菜单项的名称。



要实现该案例，我们先画出类图：



### 代码实现：

不管是菜单还是菜单项，都应该继承自统一的接口，这里姑且将这个统一的接口称为菜单组件。

```

1 //菜单组件 不管是菜单还是菜单项，都应该继承该类
2 public abstract class MenuComponent {
3
4     protected String name;
5     protected int level;
6
7     //添加菜单
8     public void add(MenuComponent menuComponent){
9         throw new UnsupportedOperationException();
10    }
11
12    //移除菜单
13    public void remove(MenuComponent menuComponent){

```

```

14         throw new UnsupportedOperationException();
15     }
16
17     //获取指定的子菜单
18     public MenuComponent getChild(int i){
19         throw new UnsupportedOperationException();
20     }
21
22     //获取菜单名称
23     public String getName(){
24         return name;
25     }
26
27     public void print(){
28         throw new UnsupportedOperationException();
29     }
30 }

```

这里的MenuComponent定义为抽象类，因为有一些共有的属性和行为要在该类中实现，Menu和MenuItem类就可以只覆盖自己感兴趣的方法，而不用搭理不需要或者不感兴趣的方法，举例来说，Menu类可以包含子菜单，因此需要覆盖add()、remove()、getChild()方法，但是MenuItem就不应该有这些方法。这里给出的默认实现是抛出异常，你也可以根据自己的需要改写默认实现。

```

1 public class Menu extends MenuComponent {
2
3     private List<MenuComponent> menuComponentList;
4
5     public Menu(String name,int level){
6         this.level = level;
7         this.name = name;
8         menuComponentList = new ArrayList<MenuComponent>();
9     }
10
11     @Override
12     public void add(MenuComponent menuComponent) {
13         menuComponentList.add(menuComponent);
14     }
15
16     @Override
17     public void remove(MenuComponent menuComponent) {
18         menuComponentList.remove(menuComponent);
19     }
20
21     @Override
22     public MenuComponent getChild(int i) {
23         return menuComponentList.get(i);
24     }
25
26     @Override
27     public void print() {
28
29         for (int i = 1; i < level; i++) {
30             System.out.print("--");
31         }
32         System.out.println(name);
33         for (MenuComponent menuComponent : menuComponentList) {
34             menuComponent.print();

```

```
35     }
36 }
37 }
```

Menu类已经实现了除了getName方法的其他所有方法，因为Menu类具有添加菜单，移除菜单和获取子菜单的功能。

```
1 public class MenuItem extends MenuComponent {
2
3     public MenuItem(String name,int level) {
4         this.name = name;
5         this.level = level;
6     }
7
8     @Override
9     public void print() {
10        for (int i = 1; i < level; i++) {
11            System.out.print("--");
12        }
13        System.out.println(name);
14    }
15 }
```

MenuItem是菜单项，不能再有子菜单，所以添加菜单，移除菜单和获取子菜单的功能并不能实现。

## 5.6.4 组合模式的分类

在使用组合模式时，根据抽象构件类的定义形式，我们可将组合模式分为透明组合模式和安全组合模式两种形式。

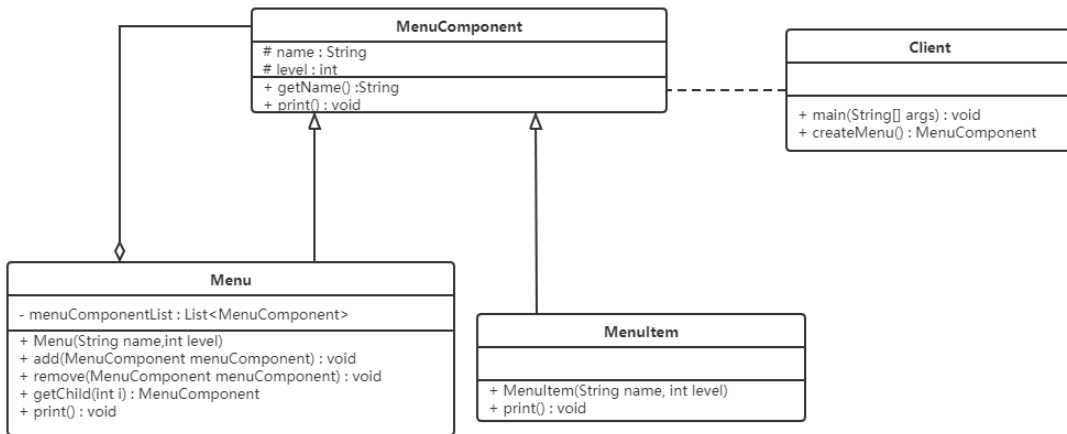
- 透明组合模式

透明组合模式中，抽象根节点角色中声明了所有用于管理成员对象的方法，比如在示例中MenuComponent 声明了 add、remove 、getChild 方法，这样做的好处是确保所有的构件类都有相同的接口。透明组合模式也是组合模式的标准形式。

透明组合模式的缺点是不够安全，因为叶子对象和容器对象在本质上是有所区别的，叶子对象不可能有下一个层次的对象，即不可能包含成员对象，因此为其提供 add()、remove() 等方法是没有意义的，这在编译阶段不会出错，但在运行阶段如果调用这些方法可能会出错（如果没有提供相应的错误处理代码）

- 安全组合模式

在安全组合模式中，在抽象构件角色中没有声明任何用于管理成员对象的方法，而是在树枝节点Menu 类中声明并实现这些方法。安全组合模式的缺点是不够透明，因为叶子构件和容器构件具有不同的方法，且容器构件中那些用于管理成员对象的方法没有在抽象构件类中定义，因此客户端不能完全针对抽象编程，必须有区别地对待叶子构件和容器构件。



### 5.6.5 优点

- 组合模式可以清楚地定义分层次的复杂对象，表示对象的全部或部分层次，它让客户端忽略了层次的差异，方便对整个层次结构进行控制。
- 客户端可以一致地使用一个组合结构或其中单个对象，不必关心处理的是单个对象还是整个组合结构，简化了客户端代码。
- 在组合模式中增加新的树枝节点和叶子节点都很方便，无须对现有类库进行任何修改，符合“开闭原则”。
- 组合模式为树形结构的面向对象实现提供了一种灵活的解决方案，通过叶子节点和树枝节点的递归组合，可以形成复杂的树形结构，但对树形结构的控制却非常简单。

### 5.6.6 使用场景

组合模式正是应树形结构而生，所以组合模式的使用场景就是出现树形结构的地方。比如：文件目录显示，多级目录呈现等树形结构数据的操作。

## 5.7 享元模式

### 5.7.1 概述

**定义：**

运用共享技术来有效地支持大量细粒度对象的复用。它通过共享已经存在的对象来大幅度减少需要创建的对象数量、避免大量相似对象的开销，从而提高系统资源的利用率。

### 5.7.2 结构

享元 (Flyweight ) 模式中存在以下两种状态：

1. 内部状态，即不会随着环境的改变而改变的可共享部分。
2. 外部状态，指随环境改变而改变的不可以共享的部分。享元模式的实现要领就是区分应用中的这两种状态，并将外部状态外部化。

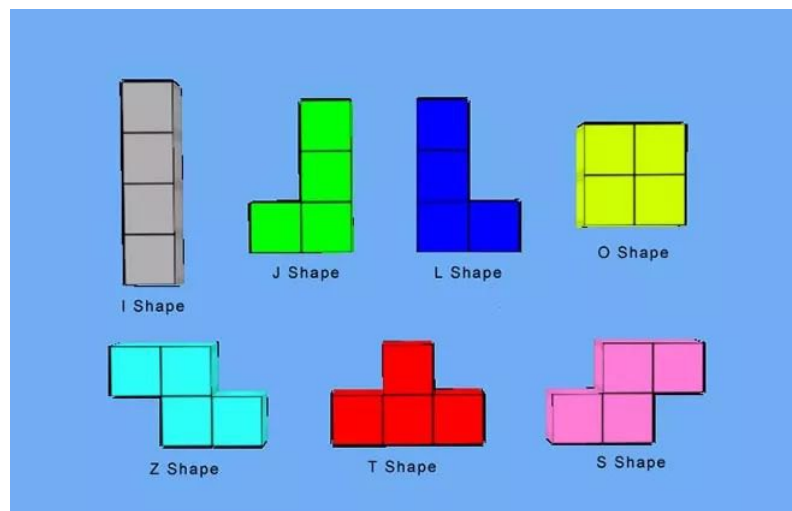
享元模式的主要有以下角色：

- 抽象享元角色 (Flyweight) : 通常是一个接口或抽象类, 在抽象享元类中声明了具体享元类公共的方法, 这些方法可以向外界提供享元对象的内部数据 (内部状态), 同时也可以通过这些方法来设置外部数据 (外部状态)。
- 具体享元 (Concrete Flyweight) 角色 : 它实现了抽象享元类, 称为享元对象; 在具体享元类中为内部状态提供了存储空间。通常我们可以结合单例模式来设计具体享元类, 为每一个具体享元类提供唯一的享元对象。
- 非享元 (Unsharable Flyweight) 角色 : 并不是所有的抽象享元类的子类都需要被共享, 不能被共享的子类可设计为非共享具体享元类; 当需要一个非共享具体享元类的对象时可以直接通过实例化创建。
- 享元工厂 (Flyweight Factory) 角色 : 负责创建和管理享元角色。当客户对象请求一个享元对象时, 享元工厂检查系统中是否存在符合要求的享元对象, 如果存在则提供给客户; 如果不存在的话, 则创建一个新的享元对象。

### 5.7.3 案例实现

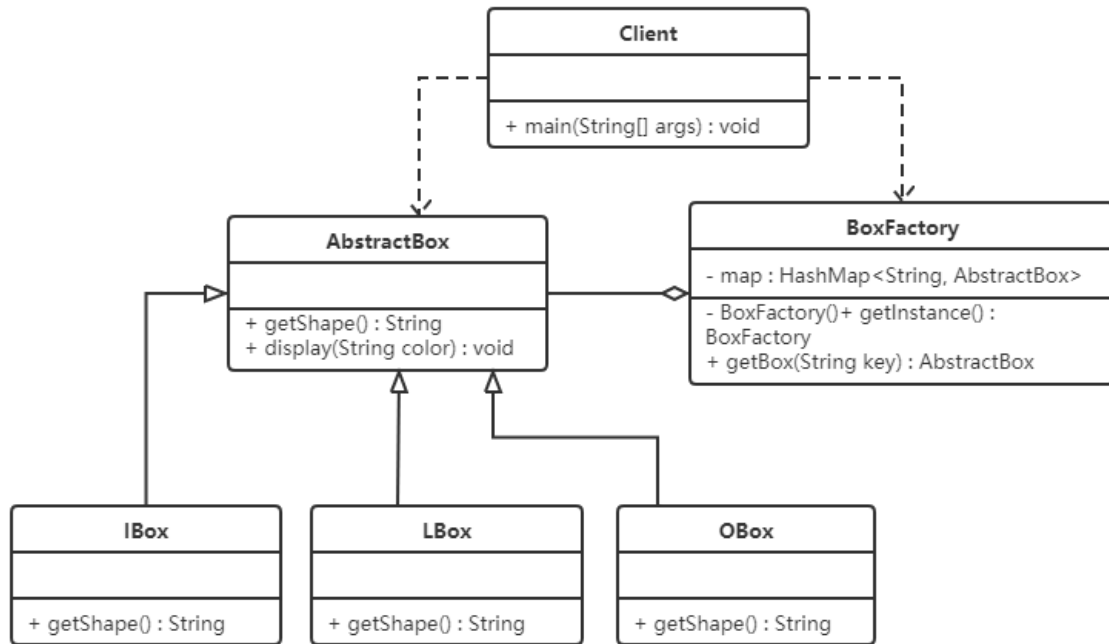
#### 【例】俄罗斯方块

下面的图片是众所周知的俄罗斯方块中的一个方块, 如果在俄罗斯方块这个游戏中, 每个不同的方块都是一个实例对象, 这些对象就要占用很多的内存空间, 下面利用享元模式进行实现。



先来看类图:





代码如下:

俄罗斯方块有不同的形状，我们可以对这些形状向上抽取出AbstractBox，用来定义共性的属性和行为。

```

1 public abstract class AbstractBox {
2     public abstract String getShape();
3
4     public void display(String color) {
5         system.out.println("方块形状: " + this.getShape() + " 颜色: " + color);
6     }
7 }
  
```

接下来就是定义不同的形状了，IBox类、LBox类、OBox类等。

```

1 public class IBox extends AbstractBox {
2
3     @Override
4     public String getShape() {
5         return "I";
6     }
7 }
8
9 public class LBox extends AbstractBox {
10
11     @Override
12     public String getShape() {
13         return "L";
14     }
15 }
16
17 public class OBox extends AbstractBox {
18
19     @Override
20     public String getShape() {
21         return "O";
  
```

```
22     }
23 }
```

提供了一个工厂类 (BoxFactory)，用来管理享元对象 (也就是AbstractBox子类对象)，该工厂类对象只需要一个，所以可以使用单例模式。并给工厂类提供一个获取形状的方法。

```
1  public class BoxFactory {
2
3      private static HashMap<String, AbstractBox> map;
4
5      private BoxFactory() {
6          map = new HashMap<String, AbstractBox>();
7          AbstractBox iBox = new IBox();
8          AbstractBox lBox = new LBox();
9          AbstractBox oBox = new OBox();
10         map.put("I", iBox);
11         map.put("L", lBox);
12         map.put("O", oBox);
13     }
14
15     public static final BoxFactory getInstance() {
16         return SingletonHolder.INSTANCE;
17     }
18
19     private static class SingletonHolder {
20         private static final BoxFactory INSTANCE = new BoxFactory();
21     }
22
23     public AbstractBox getBox(String key) {
24         return map.get(key);
25     }
26 }
```

## 5.7.5 优缺点和使用场景

### 1, 优点

- 极大减少内存中相似或相同对象数量，节约系统资源，提供系统性能
- 享元模式中的外部状态相对独立，且不影响内部状态

### 2, 缺点:

为了使对象可以共享，需要将享元对象的部分状态外部化，分离内部状态和外部状态，使程序逻辑复杂

### 3, 使用场景:

- 一个系统有大量相同或者相似的对象，造成内存的大量耗费。
- 对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。
- 在使用享元模式时需要维护一个存储享元对象的享元池，而这需要耗费一定的系统资源，因此，应当在需要多次重复使用享元对象时才值得使用享元模式。

## 5.7.6 JDK源码解析

Integer类使用了享元模式。我们先看下面的例子：

```

1 public class Demo {
2     public static void main(String[] args) {
3         Integer i1 = 127;
4         Integer i2 = 127;
5
6         System.out.println("i1和i2对象是否是同一个对象? " + (i1 == i2));
7
8         Integer i3 = 128;
9         Integer i4 = 128;
10
11        System.out.println("i3和i4对象是否是同一个对象? " + (i3 == i4));
12    }
13 }

```

运行上面代码，结果如下：

```

D:\develop\java\jdk1.8\bin\java.exe ...
i1和i2对象是否是同一个对象? true
i3和i4对象是否是同一个对象? false

```

为什么第一个输出语句输出的是true，第二个输出语句输出的是false？通过反编译软件进行反编译，代码如下：

```

1 public class Demo {
2     public static void main(String[] args) {
3         Integer i1 = Integer.valueOf((int)127);
4         Integer i2 = Integer.valueOf((int)127);
5         System.out.println((String)new
6     StringBuilder().append((String)"i1\u548ci2\u5bf9\u8c61\u662f\u5426\u662f\u5
7     40c\u4e00\u4e2a\u5bf9\u8c61\u5417").append((boolean)(i1 ==
8     i2)).toString());
9         Integer i3 = Integer.valueOf((int)128);
10        Integer i4 = Integer.valueOf((int)128);
11        System.out.println((String)new
12    StringBuilder().append((String)"i3\u548ci4\u5bf9\u8c61\u662f\u5426\u662f\u5
13    40c\u4e00\u4e2a\u5bf9\u8c61\u5417").append((boolean)(i3 ==
14    i4)).toString());
15    }
16 }

```

上面代码可以看到，直接给Integer类型的变量赋值基本数据类型数据的操作底层使用的是valueOf()，所以只需要看该方法即可

```

1 public final class Integer extends Number implements Comparable<Integer> {
2
3     public static Integer valueOf(int i) {
4         if (i >= IntegerCache.low && i <= IntegerCache.high)
5             return IntegerCache.cache[i + (-IntegerCache.low)];
6         return new Integer(i);
7     }
8
9     private static class IntegerCache {
10        static final int low = -128;
11        static final int high;
12        static final Integer cache[];

```

```

13
14     static {
15         int h = 127;
16         String integerCacheHighPropValue =
17
18         sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
19         if (integerCacheHighPropValue != null) {
20             try {
21                 int i = parseInt(integerCacheHighPropValue);
22                 i = Math.max(i, 127);
23                 // Maximum array size is Integer.MAX_VALUE
24                 h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
25             } catch (NumberFormatException nfe) {
26             }
27         }
28         high = h;
29         cache = new Integer[(high - low) + 1];
30         int j = low;
31         for(int k = 0; k < cache.length; k++)
32             cache[k] = new Integer(j++);
33         // range [-128, 127] must be interned (JLS7 5.1.7)
34         assert IntegerCache.high >= 127;
35     }
36     private IntegerCache() {}
37 }
38 }

```

可以看到 `Integer` 默认先创建并缓存 `-128 ~ 127` 之间数的 `Integer` 对象，当调用 `valueOf` 时如果参数在 `-128 ~ 127` 之间则计算下标并从缓存中返回，否则创建一个新的 `Integer` 对象。

## 6, 行为型模式

行为型模式用于描述程序在运行时复杂的流程控制，即描述多个类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，它涉及算法与对象间职责的分配。

行为型模式分为类行为模式和对象行为模式，前者采用继承机制来在类间分派行为，后者采用组合或聚合在对象间分配行为。由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象行为模式比类行为模式具有更大的灵活性。

行为型模式分为：

- 模板方法模式
- 策略模式
- 命令模式
- 职责链模式
- 状态模式
- 观察者模式
- 中介者模式
- 迭代器模式
- 访问者模式
- 备忘录模式

- 解释器模式

以上 11 种行为型模式，除了模板方法模式和解释器模式是类行为型模式，其他的全部属于对象行为型模式。

## 6.1 模板方法模式

### 6.1.1 概述

在面向对象程序设计过程中，程序员常常会遇到这种情况：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。

例如，去银行办理业务一般要经过以下4个流程：取号、排队、办理具体业务、对银行工作人员进行评分等，其中取号、排队和对银行工作人员进行评分的业务对客户是一样的，可以在父类中实现，但是办理具体业务却因人而异，它可能是存款、取款或者转账等，可以延迟到子类中实现。

**定义：**

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

### 6.1.2 结构

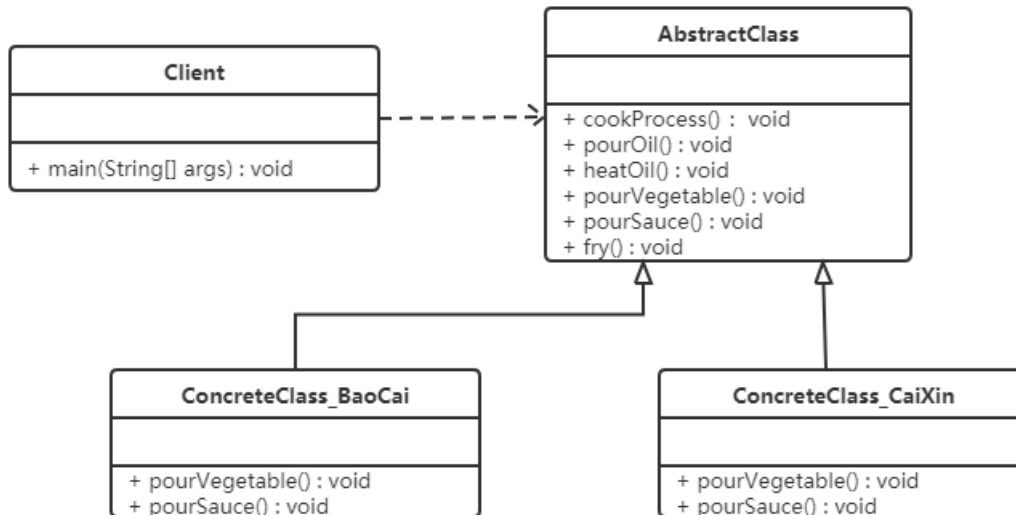
模板方法 (Template Method) 模式包含以下主要角色：

- 抽象类 (Abstract Class) : 负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。
  - 模板方法: 定义了算法的骨架，按某种顺序调用其包含的基本方法。
  - 基本方法: 是实现算法各个步骤的方法，是模板方法的组成部分。基本方法又可以分为三种：
    - 抽象方法 (Abstract Method) : 一个抽象方法由抽象类声明、由其具体子类实现。
    - 具体方法 (Concrete Method) : 一个具体方法由一个抽象类或具体类声明并实现，其子类可以进行覆盖也可以直接继承。
    - 钩子方法 (Hook Method) : 在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。  
一般钩子方法是用于判断的逻辑方法，这类方法名一般为 `isXxx`，返回值类型为 `boolean` 类型。
- 具体子类 (Concrete Class) : 实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的组成步骤。

### 6.1.3 案例实现

**【例】炒菜**

炒菜的步骤是固定的，分为倒油、热油、倒蔬菜、倒调料品、翻炒等步骤。现通过模板方法模式来用代码模拟。类图如下：



代码如下:

```

1 public abstract class AbstractClass {
2
3     public final void cookProcess() {
4         //第一步: 倒油
5         this.pourOil();
6         //第二步: 热油
7         this.heatOil();
8         //第三步: 倒蔬菜
9         this.pourVegetable();
10        //第四步: 倒调味料
11        this.pourSauce();
12        //第五步: 翻炒
13        this.fry();
14    }
15
16    public void pourOil() {
17        System.out.println("倒油");
18    }
19
20    //第二步: 热油是一样的, 所以直接实现
21    public void heatOil() {
22        System.out.println("热油");
23    }
24
25    //第三步: 倒蔬菜是不一样的 (一个下包菜, 一个是下菜心)
26    public abstract void pourVegetable();
27
28    //第四步: 倒调味料是不一样
29    public abstract void pourSauce();
30
31
32    //第五步: 翻炒是一样的, 所以直接实现
33    public void fry(){
34        System.out.println("炒啊炒啊炒到熟啊");
35    }
36 }
37
38 public class ConcreteClass_BaoCai extends AbstractClass {

```

```

39
40     @Override
41     public void pourVegetable() {
42         System.out.println("下锅的蔬菜是包菜");
43     }
44
45     @Override
46     public void pourSauce() {
47         System.out.println("下锅的酱料是辣椒");
48     }
49 }
50
51 public class ConcreteClass_CaiXin extends AbstractClass {
52     @Override
53     public void pourVegetable() {
54         System.out.println("下锅的蔬菜是菜心");
55     }
56
57     @Override
58     public void pourSauce() {
59         System.out.println("下锅的酱料是蒜蓉");
60     }
61 }
62
63 public class Client {
64     public static void main(String[] args) {
65         //炒手撕包菜
66         ConcreteClass_BaoCai baoCai = new ConcreteClass_BaoCai();
67         baoCai.cookProcess();
68
69         //炒蒜蓉菜心
70         ConcreteClass_CaiXin caiXin = new ConcreteClass_CaiXin();
71         caiXin.cookProcess();
72     }
73 }

```

注意：为防止恶意操作，一般模板方法都加上 `final` 关键词。

### 6.1.3 优缺点

#### 优点：

- 提高代码复用性  
将相同部分的代码放在抽象的父类中，而将不同的代码放入不同的子类中。
- 实现了反向控制  
通过一个父类调用其子类的操作，通过对子类的具体实现扩展不同的行为，实现了反向控制，并符合“开闭原则”。

#### 缺点：

- 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象。
- 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。

## 6.1.4 适用场景

- 算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。
- 需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制。

## 6.1.5 JDK源码解析

InputStream类就使用了模板方法模式。在InputStream类中定义了多个 `read()` 方法，如下：

```
1 public abstract class InputStream implements Closeable {
2     //抽象方法，要求子类必须重写
3     public abstract int read() throws IOException;
4
5     public int read(byte b[]) throws IOException {
6         return read(b, 0, b.length);
7     }
8
9     public int read(byte b[], int off, int len) throws IOException {
10        if (b == null) {
11            throw new NullPointerException();
12        } else if (off < 0 || len < 0 || len > b.length - off) {
13            throw new IndexOutOfBoundsException();
14        } else if (len == 0) {
15            return 0;
16        }
17
18        int c = read(); //调用了无参的read方法，该方法是每次读取一个字节数据
19        if (c == -1) {
20            return -1;
21        }
22        b[off] = (byte)c;
23
24        int i = 1;
25        try {
26            for (; i < len ; i++) {
27                c = read();
28                if (c == -1) {
29                    break;
30                }
31                b[off + i] = (byte)c;
32            }
33        } catch (IOException ee) {
34        }
35        return i;
36    }
37 }
```

从上面代码可以看到，无参的 `read()` 方法是抽象方法，要求子类必须实现。而 `read(byte b[])` 方法调用了 `read(byte b[], int off, int len)` 方法，所以在此处重点看的方法是带三个参数的方法。

在该方法中第18行、27行，可以看到调用了无参的抽象的 `read()` 方法。

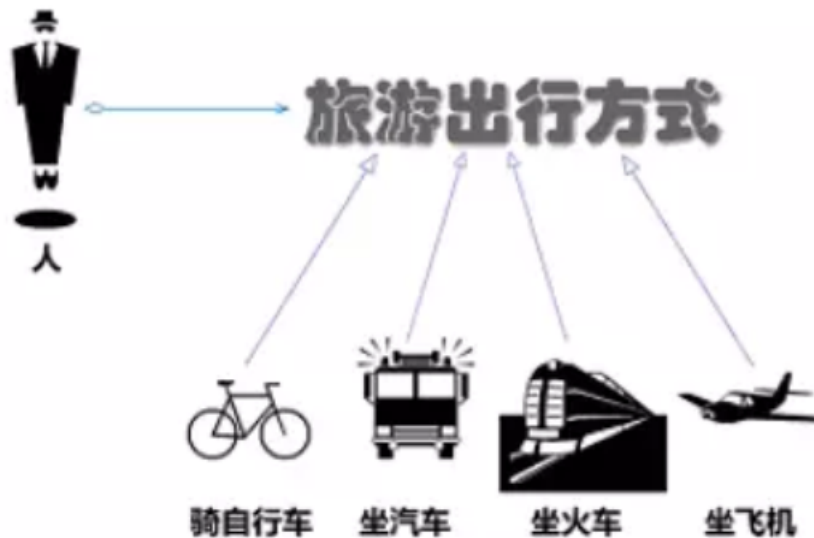


总结如下：在InputStream父类中已经定义好了读取一个字节数组数据的方法是每次读取一个字节，并将其存储到数组的第一个索引位置，读取len个字节数据。具体如何读取一个字节数据呢？由子类实现。

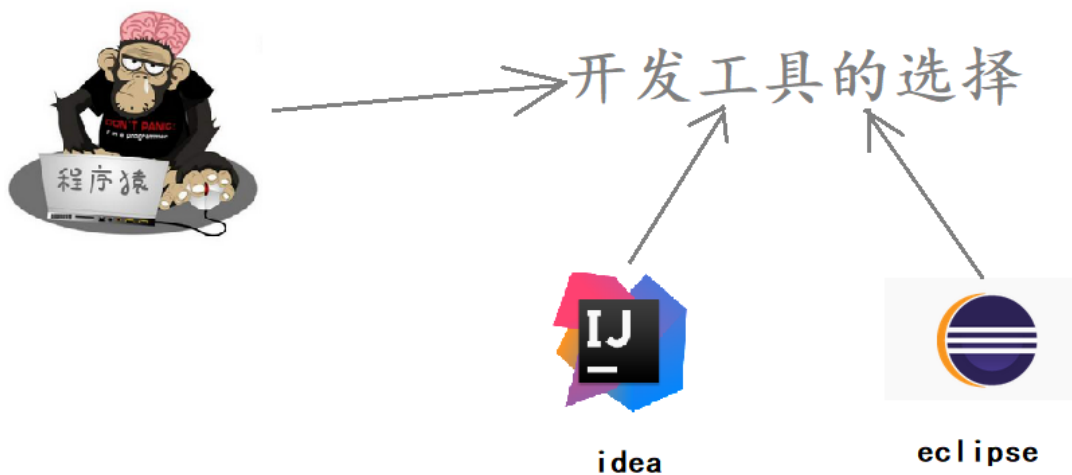
## 6.2 策略模式

### 6.2.1 概述

先看下面的图片，我们去旅游选择出行模式有很多种，可以骑自行车、可以坐汽车、可以坐火车、可以坐飞机。



作为一个程序猿，开发需要选择一款开发工具，当然可以进行代码开发的工具有很多，可以选择Idea进行开发，也可以使用eclipse进行开发，也可以使用其他的一些开发工具。



**定义：**

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

### 6.2.2 结构

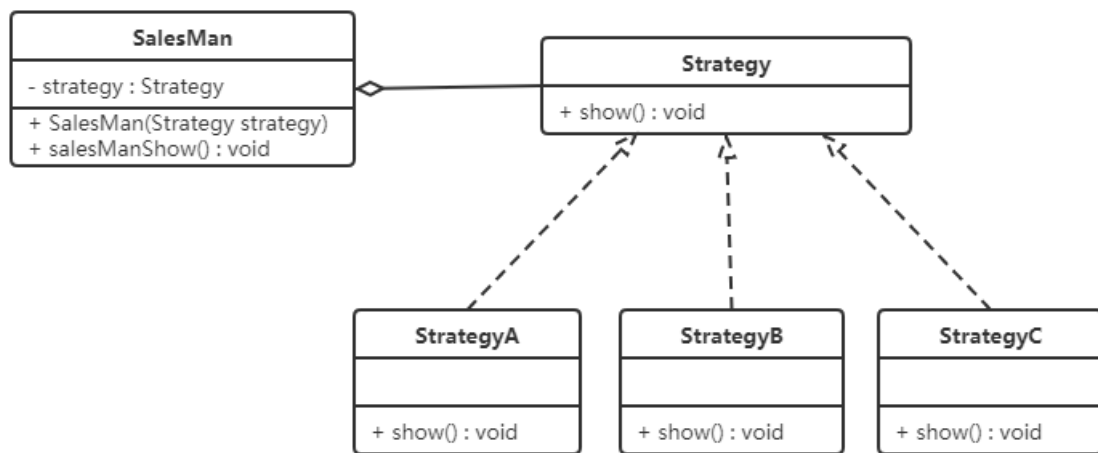
策略模式的主要角色如下：

- 抽象策略 (Strategy) 类：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。
- 具体策略 (Concrete Strategy) 类：实现了抽象策略定义的接口，提供具体的算法实现或行为。
- 环境 (Context) 类：持有一个策略类的引用，最终给客户端调用。

### 6.2.3 案例实现

【例】促销活动

一家百货公司在定年度的促销活动。针对不同的节日（春节、中秋节、圣诞节）推出不同的促销活动，由促销员将促销活动展示给客户。类图如下：



代码如下：

定义百货公司所有促销活动的共同接口

```
1 public interface Strategy {
2     void show();
3 }
```

定义具体策略角色 (Concrete Strategy)：每个节日具体的促销活动

```
1 //为春节准备的促销活动A
2 public class StrategyA implements Strategy {
3
4     public void show() {
5         System.out.println("买一送一");
6     }
7 }
8
9 //为中秋准备的促销活动B
10 public class StrategyB implements Strategy {
11
12     public void show() {
13         System.out.println("满200元减50元");
14     }
15 }
```

```

16
17 //为圣诞准备的促销活动C
18 public class StrategyC implements Strategy {
19
20     public void show() {
21         System.out.println("满1000元加一元换购任意200元以下商品");
22     }
23 }

```

定义环境角色 (Context)：用于连接上下文，即把促销活动推销给客户，这里可以理解为销售员

```

1 public class SalesMan {
2     //持有抽象策略角色的引用
3     private Strategy strategy;
4
5     public SalesMan(Strategy strategy) {
6         this.strategy = strategy;
7     }
8
9     //向客户展示促销活动
10    public void salesManShow(){
11        strategy.show();
12    }
13 }

```

## 6.2.4 优缺点

### 1, 优点:

- 策略类之间可以自由切换  
由于策略类都实现同一个接口，所以使它们之间可以自由切换。
- 易于扩展  
增加一个新的策略只需要添加一个具体的策略类即可，基本不需要改变原有的代码，符合“开闭原则”
- 避免使用多重条件选择语句 (if else) ，充分体现面向对象设计思想。

### 2, 缺点:

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。
- 策略模式将造成产生很多策略类，可以通过使用享元模式在一定程度上减少对象的数量。

## 6.2.5 使用场景

- 一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中。
- 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，可将每个条件分支移入它们各自的策略类中以代替这些条件语句。
- 系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时。
- 系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构。
- 多个类只区别在表现行为不同，可以使用策略模式，在运行时动态选择具体要执行的行为。

## 6.2.6 JDK源码解析

`Comparator` 中的策略模式。在`Arrays`类中有一个 `sort()` 方法，如下：

```
1 public class Arrays{
2     public static <T> void sort(T[] a, Comparator<? super T> c) {
3         if (c == null) {
4             sort(a);
5         } else {
6             if (LegacyMergeSort.userRequested)
7                 LegacyMergeSort(a, c);
8             else
9                 TimSort.sort(a, 0, a.length, c, null, 0, 0);
10        }
11    }
12 }
```

`Arrays`就是一个环境角色类，这个`sort`方法可以传一个新策略让`Arrays`根据这个策略来进行排序。就比如下面的测试类。

```
1 public class demo {
2     public static void main(String[] args) {
3
4         Integer[] data = {12, 2, 3, 2, 4, 5, 1};
5         // 实现降序排序
6         Arrays.sort(data, new Comparator<Integer>() {
7             public int compare(Integer o1, Integer o2) {
8                 return o2 - o1;
9             }
10        });
11        System.out.println(Arrays.toString(data)); //[12, 5, 4, 3, 2, 2, 1]
12    }
13 }
```

这里我们在调用`Arrays`的`sort`方法时，第二个参数传递的是`Comparator`接口的子实现类对象。所以`Comparator`充当的是抽象策略角色，而具体的子实现类充当的是具体策略角色。环境角色类（`Arrays`）应该持有抽象策略的引用来调用。那么，`Arrays`类的`sort`方法到底有没有使用`Comparator`子实现类中的 `compare()` 方法吗？让我们继续查看`TimSort`类的 `sort()` 方法，代码如下：

```
1 class TimSort<T> {
2     static <T> void sort(T[] a, int lo, int hi, Comparator<? super T> c,
3         T[] work, int workBase, int workLen) {
4         assert c != null && a != null && lo >= 0 && lo <= hi && hi <=
5         a.length;
6
7         int nRemaining = hi - lo;
8         if (nRemaining < 2)
9             return; // Arrays of size 0 and 1 are always sorted
10
11        // If array is small, do a "mini-TimSort" with no merges
12        if (nRemaining < MIN_MERGE) {
13            int initRunLen = countRunAndMakeAscending(a, lo, hi, c);
```

```

13         binarySort(a, lo, hi, lo + initRunLen, c);
14         return;
15     }
16     ...
17 }
18
19     private static <T> int countRunAndMakeAscending(T[] a, int lo, int
hi, Comparator<? super T> c) {
20         assert lo < hi;
21         int runHi = lo + 1;
22         if (runHi == hi)
23             return 1;
24
25         // Find end of run, and reverse range if descending
26         if (c.compare(a[runHi++], a[lo]) < 0) { // Descending
27             while (runHi < hi && c.compare(a[runHi], a[runHi - 1]) < 0)
28                 runHi++;
29             reverseRange(a, lo, runHi);
30         } else { // Ascending
31             while (runHi < hi && c.compare(a[runHi], a[runHi - 1]) >= 0)
32                 runHi++;
33         }
34
35         return runHi - lo;
36     }
37 }

```

上面的代码中最终会跑到 `countRunAndMakeAscending()` 这个方法中。我们可以看见，只用了 `compare` 方法，所以在调用 `Arrays.sort` 方法只传具体 `compare` 重写方法的类对象就行，这也是 `Comparator` 接口中必须要子类实现的一个方法。

## 6.3 命令模式

### 6.3.1 概述

日常生活中，我们出去吃饭都会遇到下面的场景。



### 定义：

将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行存储、传递、调用、增加与管理。

## 6.3.2 结构

命令模式包含以下主要角色：

- 抽象命令类 (Command) 角色： 定义命令的接口，声明执行的方法。
- 具体命令 (Concrete Command) 角色： 具体的命令，实现命令接口；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。
- 实现者/接收者 (Receiver) 角色： 接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。
- 调用者/请求者 (Invoker) 角色： 要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。

## 6.3.3 案例实现

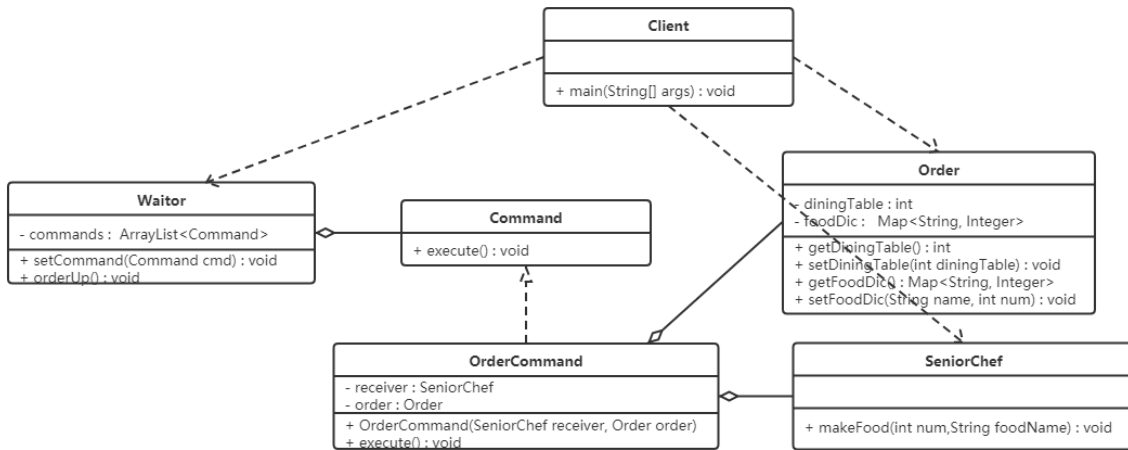
将上面的案例用代码实现，那我们就需要分析命令模式的角色在该案例中由谁来充当。

服务员： 就是调用者角色，由她来发起命令。

资深大厨： 就是接收者角色，真正命令执行的对象。

订单： 命令中包含订单。

类图如下：



代码如下:

```

1  public interface Command {
2      void execute(); //只需要定义一个统一的执行方法
3  }
4
5  public class OrderCommand implements Command {
6
7      //持有接受者对象
8      private SeniorChef receiver;
9      private Order order;
10
11     public OrderCommand(SeniorChef receiver, Order order){
12         this.receiver = receiver;
13         this.order = order;
14     }
15
16     public void execute() {
17         System.out.println(order.getDiningTable() + "桌的订单: ");
18         Set<String> keys = order.getFoodDic().keySet();
19         for (String key : keys) {
20             receiver.makeFood(order.getFoodDic().get(key), key);
21         }
22
23         try {
24             Thread.sleep(100); //停顿一下 模拟做饭的过程
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28
29         System.out.println(order.getDiningTable() + "桌的饭弄好了");
30     }
31 }
32 }
33
34 public class Order {
35     // 餐桌号码
36     private int diningTable;
37
38     // 用来存储餐名并记录份数
39     private Map<String, Integer> foodDic = new HashMap<String, Integer>();
40
41     public int getDiningTable() {

```

```

42     return diningTable;
43 }
44
45 public void setDiningTable(int diningTable) {
46     this.diningTable = diningTable;
47 }
48
49 public Map<String, Integer> getFoodDic() {
50     return foodDic;
51 }
52
53 public void setFoodDic(String name, int num) {
54     foodDic.put(name,num);
55 }
56 }
57
58 // 资深大厨类 是命令的Receiver
59 public class SeniorChef {
60
61     public void makeFood(int num,String foodName) {
62         System.out.println(num + "份" + foodName);
63     }
64 }
65
66 public class Waitor {
67
68     private ArrayList<Command> commands;//可以持有很多的命令对象
69
70     public Waitor() {
71         commands = new ArrayList();
72     }
73
74     public void setCommand(Command cmd){
75         commands.add(cmd);
76     }
77
78     // 发出命令 喊 订单来了, 厨师开始执行
79     public void orderUp() {
80         System.out.println("美女服务员: 叮咚, 大厨, 新订单来了.....");
81         for (int i = 0; i < commands.size(); i++) {
82             Command cmd = commands.get(i);
83             if (cmd != null) {
84                 cmd.execute();
85             }
86         }
87     }
88 }
89
90 public class Client {
91     public static void main(String[] args) {
92         //创建2个order
93         Order order1 = new Order();
94         order1.setDiningTable(1);
95         order1.getFoodDic().put("西红柿鸡蛋面",1);
96         order1.getFoodDic().put("小杯可乐",2);
97
98         Order order2 = new Order();
99         order2.setDiningTable(3);

```



```

100     order2.getFoodDic().put("尖椒肉丝盖饭",1);
101     order2.getFoodDic().put("小杯雪碧",1);
102
103     //创建接收者
104     SeniorChef receiver=new SeniorChef();
105     //将订单和接收者封装成命令对象
106     OrderCommand cmd1 = new OrderCommand(receiver, order1);
107     OrderCommand cmd2 = new OrderCommand(receiver, order2);
108     //创建调用者 waiter
109     Waitor invoker = new Waitor();
110     invoker.setCommand(cmd1);
111     invoker.setCommand(cmd2);
112
113     //将订单带到柜台 并向厨师喊 订单来了
114     invoker.orderUp();
115 }
116 }

```

### 6.3.4 优缺点

#### 1, 优点:

- 降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦。
- 增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活。
- 可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令。
- 方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复。

#### 2, 缺点:

- 使用命令模式可能会导致某些系统有过多的具体命令类。
- 系统结构更加复杂。

### 6.3.5 使用场景

- 系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互。
- 系统需要在不同的时间指定请求、将请求排队和执行请求。
- 系统需要支持命令的撤销 (Undo) 操作和恢复 (Redo) 操作。

### 6.3.6 JDK源码解析

Runnable是一个典型命令模式，Runnable担当命令的角色，Thread充当的是调用者，start方法就是其执行方法

```

1 //命令接口(抽象命令角色)
2 public interface Runnable {
3     public abstract void run();
4 }
5
6 //调用者

```

```

7 public class Thread implements Runnable {
8     private Runnable target;
9
10    public synchronized void start() {
11        if (threadStatus != 0)
12            throw new IllegalThreadStateException();
13
14        group.add(this);
15
16        boolean started = false;
17        try {
18            start0();
19            started = true;
20        } finally {
21            try {
22                if (!started) {
23                    group.threadStartFailed(this);
24                }
25            } catch (Throwable ignore) {
26            }
27        }
28    }
29
30    private native void start0();
31 }

```

会调用一个native方法start0(),调用系统方法,开启一个线程。而接收者是对程序员开放的,可以自己定义接收者。

```

1 /**
2  * jdk Runnable 命令模式
3  * TurnOffThread : 属于具体
4  */
5 public class TurnOffThread implements Runnable{
6     private Receiver receiver;
7
8     public TurnOffThread(Receiver receiver) {
9         this.receiver = receiver;
10    }
11    public void run() {
12        receiver.turnOFF();
13    }
14 }

```

```

1 /**
2  * 测试类
3  */
4 public class Demo {
5     public static void main(String[] args) {
6         Receiver receiver = new Receiver();
7         TurnOffThread turnOffThread = new TurnOffThread(receiver);
8         Thread thread = new Thread(turnOffThread);
9         thread.start();
10    }
11 }

```

## 6.4 责任链模式

### 6.4.1 概述

在现实生活中，常常会出现这样的事例：一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同，员工必须根据自己要请假的天数去找不同的领导签名，也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。这样的例子还有很多，如找领导出差报销、生活中的“击鼓传花”游戏等。

#### 定义：

又名职责链模式，为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

### 6.4.2 结构

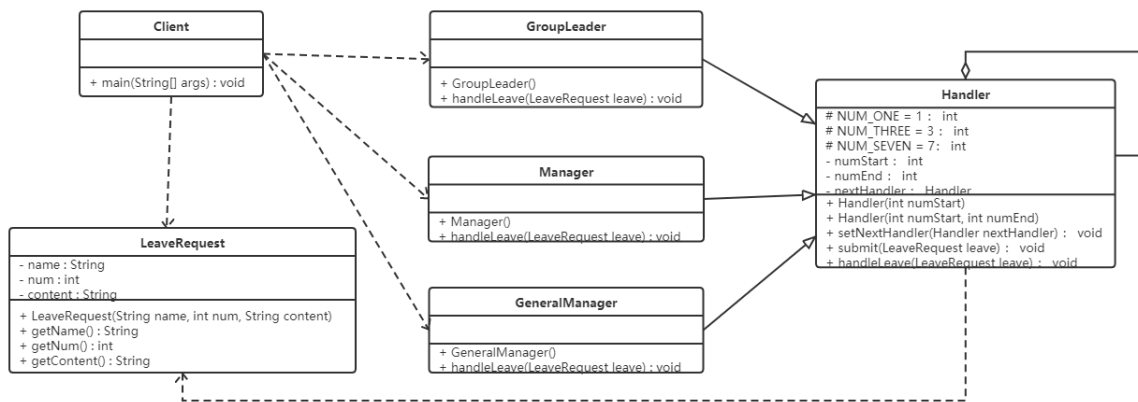
职责链模式主要包含以下角色：

- 抽象处理者 (Handler) 角色：定义一个处理请求的接口，包含抽象处理方法和一个后继连接。
- 具体处理者 (Concrete Handler) 角色：实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。
- 客户类 (Client) 角色：创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

### 6.4.3 案例实现

现需要开发一个请假流程控制系统。请假一天以下的假只需要小组长同意即可；请假1天到3天的假还需要部门经理同意；请求3天到7天还需要总经理同意才行。

类图如下：



代码如下：

```
1 //请假条
2 public class LeaveRequest {
3     private String name;//姓名
4     private int num;//请假天数
```

```

5     private String content; //请假内容
6
7     public LeaveRequest(String name, int num, String content) {
8         this.name = name;
9         this.num = num;
10        this.content = content;
11    }
12
13    public String getName() {
14        return name;
15    }
16
17    public int getNum() {
18        return num;
19    }
20
21    public String getContent() {
22        return content;
23    }
24 }
25
26 //处理者抽象类
27 public abstract class Handler {
28     protected final static int NUM_ONE = 1;
29     protected final static int NUM_THREE = 3;
30     protected final static int NUM_SEVEN = 7;
31
32     //该领导处理的请假天数区间
33     private int numStart;
34     private int numEnd;
35
36     //领导上面还有领导
37     private Handler nextHandler;
38
39     //设置请假天数范围 上不封顶
40     public Handler(int numStart) {
41         this.numStart = numStart;
42     }
43
44     //设置请假天数范围
45     public Handler(int numStart, int numEnd) {
46         this.numStart = numStart;
47         this.numEnd = numEnd;
48     }
49
50     //设置上级领导
51     public void setNextHandler(Handler nextHandler){
52         this.nextHandler = nextHandler;
53     }
54
55     //提交请假条
56     public final void submit(LeaveRequest leave){
57         if(0 == this.numStart){
58             return;
59         }
60
61         //如果请假天数达到该领导者的处理要求
62         if(leave.getNum() >= this.numStart){

```

```

63         this.handleLeave(leave);
64
65         //如果还有上级 并且请假天数超过了当前领导的处理范围
66         if(null != this.nextHandler && leave.getNum() > numEnd){
67             this.nextHandler.submit(leave); //继续提交
68         } else {
69             System.out.println("流程结束");
70         }
71     }
72 }
73
74 //各级领导处理请假条方法
75 protected abstract void handleLeave(LeaveRequest leave);
76 }
77
78 //小组长
79 public class GroupLeader extends Handler {
80     public GroupLeader() {
81         //小组长处理1-3天的请假
82         super(Handler.NUM_ONE, Handler.NUM_THREE);
83     }
84
85     @Override
86     protected void handleLeave(LeaveRequest leave) {
87         System.out.println(leave.getName() + "请假" + leave.getNum() +
88 "天," + leave.getContent() + "。");
89         System.out.println("小组长审批: 同意。");
90     }
91 }
92 //部门经理
93 public class Manager extends Handler {
94     public Manager() {
95         //部门经理处理3-7天的请假
96         super(Handler.NUM_THREE, Handler.NUM_SEVEN);
97     }
98
99     @Override
100    protected void handleLeave(LeaveRequest leave) {
101        System.out.println(leave.getName() + "请假" + leave.getNum() +
102 "天," + leave.getContent() + "。");
103        System.out.println("部门经理审批: 同意。");
104    }
105 }
106 //总经理
107 public class GeneralManager extends Handler {
108     public GeneralManager() {
109         //部门经理处理7天以上的请假
110         super(Handler.NUM_SEVEN);
111     }
112
113     @Override
114     protected void handleLeave(LeaveRequest leave) {
115         System.out.println(leave.getName() + "请假" + leave.getNum() +
116 "天," + leave.getContent() + "。");
117         System.out.println("总经理审批: 同意。");

```

```

118 }
119
120 //测试类
121 public class Client {
122     public static void main(String[] args) {
123         //请假条来一张
124         LeaveRequest leave = new LeaveRequest("小花",5,"身体不适");
125
126         //各位领导
127         GroupLeader groupLeader = new GroupLeader();
128         Manager manager = new Manager();
129         GeneralManager generalManager = new GeneralManager();
130
131         groupLeader.setNextHandler(manager); //小组长的领导是部门经理
132         manager.setNextHandler(generalManager); //部门经理的领导是总经理
133         //之所以在这里设置上级领导，是因为可以根据实际需求来更改设置，如果实战中上级领
           导人都是固定的，则可以移到领导实现类中。
134
135         //提交申请
136         groupLeader.submit(leave);
137     }
138 }

```

## 6.4.4 优缺点

### 1, 优点:

- 降低了对象之间的耦合度  
该模式降低了请求发送者和接收者的耦合度。
- 增强了系统的可扩展性  
可以根据需要增加新的请求处理类，满足开闭原则。
- 增强了给对象指派职责的灵活性  
当工作流程发生变化，可以动态地改变链内的成员或者修改它们的次序，也可动态地新增或者删除责任。
- 责任链简化了对象之间的连接  
一个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if...else 语句。
- 责任分担  
每个类只需要处理自己该处理的工作，不能处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

### 2, 缺点:

- 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
- 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
- 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

## 6.4.5 源码解析

在javaWeb应用开发中, FilterChain是职责链(过滤器)模式的典型应用, 以下是Filter的模拟实现分析:

- 模拟web请求Request以及web响应Response

```
1 public interface Request{
2
3 }
4
5 public interface Response{
6
7 }
```

- 模拟web过滤器Filter

```
1 public interface Filter {
2     public void doFilter(Request req,Response res,FilterChain c);
3 }
```

- 模拟实现具体过滤器

```
1 public class FirstFilter implements Filter {
2     @Override
3     public void doFilter(Request request, Response response,
4         FilterChain chain) {
5
6         System.out.println("过滤器1 前置处理");
7
8         // 先执行所有request再倒序执行所有response
9         chain.doFilter(request, response);
10
11        System.out.println("过滤器1 后置处理");
12    }
13
14    public class SecondFilter implements Filter {
15        @Override
16        public void doFilter(Request request, Response response,
17            FilterChain chain) {
18
19            System.out.println("过滤器2 前置处理");
20
21            // 先执行所有request再倒序执行所有response
22            chain.doFilter(request, response);
23
24            System.out.println("过滤器2 后置处理");
25        }
26    }
27 }
```

- 模拟实现过滤器链FilterChain

```
1 public class FilterChain {
2
```

```

3     private List<Filter> filters = new ArrayList<Filter>();
4
5     private int index = 0;
6
7     // 链式调用
8     public FilterChain addFilter(Filter filter) {
9         this.filters.add(filter);
10        return this;
11    }
12
13    public void doFilter(Request request, Response response) {
14        if (index == filters.size()) {
15            return;
16        }
17        Filter filter = filters.get(index);
18        index++;
19        filter.doFilter(request, response, this);
20    }
21 }

```

- 测试类

```

1     public class Client {
2         public static void main(String[] args) {
3             Request req = null;
4             Response res = null ;
5
6             FilterChain filterChain = new FilterChain();
7             filterChain.addFilter(new FirstFilter()).addFilter(new
SecondFilter());
8             filterChain.doFilter(req, res);
9         }
10    }

```



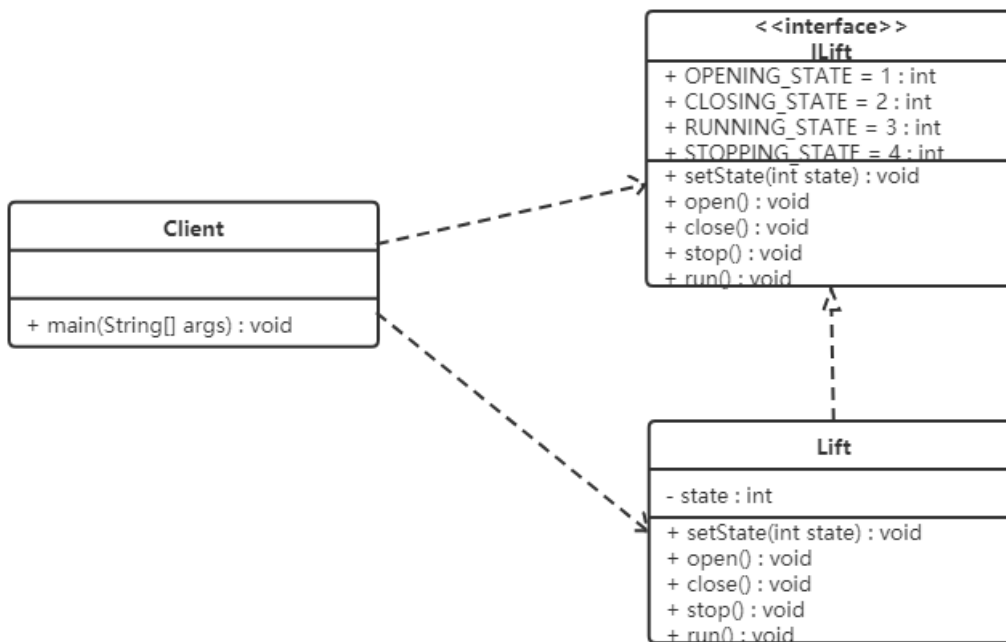
# 6, 行为型模式

## 6.5 状态模式

### 6.5.1 概述

【例】通过按钮来控制一个电梯的状态，一个电梯有开门状态，关门状态，停止状态，运行状态。每一种状态改变，都有可能要根据其他状态来更新处理。例如，如果电梯门现在处于运行时状态，就不能进行开门操作，而如果电梯门是停止状态，就可以执行开门操作。

类图如下：



代码如下：

```
1 public interface ILift {
2     //电梯的4个状态
3     //开门状态
4     public final static int OPENING_STATE = 1;
5     //关门状态
6     public final static int CLOSING_STATE = 2;
7     //运行状态
8     public final static int RUNNING_STATE = 3;
9     //停止状态
10    public final static int STOPPING_STATE = 4;
11
12    //设置电梯的状态
13    public void setState(int state);
14
15    //电梯的动作
16    public void open();
17    public void close();
18    public void run();
19    public void stop();
```

```

20 }
21
22 public class Lift implements ILift {
23     private int state;
24
25     @Override
26     public void setState(int state) {
27         this.state = state;
28     }
29
30     //执行关门动作
31     @Override
32     public void close() {
33         switch (this.state) {
34             case OPENING_STATE:
35                 System.out.println("电梯关门了。。。"); //只有开门状态可以关闭电梯
门，可以对应电梯状态表来看
36                 this.setState(CLOSING_STATE); //关门之后电梯就是关闭状态了
37                 break;
38             case CLOSING_STATE:
39                 //do nothing //已经是关门状态，不能关门
40                 break;
41             case RUNNING_STATE:
42                 //do nothing //运行时电梯门是关着的，不能关门
43                 break;
44             case STOPPING_STATE:
45                 //do nothing //停止时电梯也是关着的，不能关门
46                 break;
47         }
48     }
49
50     //执行开门动作
51     @Override
52     public void open() {
53         switch (this.state) {
54             case OPENING_STATE: //门已经开了，不能再开门了
55                 //do nothing
56                 break;
57             case CLOSING_STATE: //关门状态，门打开:
58                 System.out.println("电梯门打开了。。。");
59                 this.setState(OPENING_STATE);
60                 break;
61             case RUNNING_STATE:
62                 //do nothing 运行时电梯不能开门
63                 break;
64             case STOPPING_STATE:
65                 System.out.println("电梯门开了。。。"); //电梯停了，可以开门了
66                 this.setState(OPENING_STATE);
67                 break;
68         }
69     }
70
71     //执行运行动作
72     @Override
73     public void run() {
74         switch (this.state) {
75             case OPENING_STATE: //电梯不能开着门就走
76                 //do nothing

```

```

77         break;
78     case CLOSING_STATE://门关了, 可以运行了
79         System.out.println("电梯开始运行了。。。");
80         this.setState(RUNNING_STATE);//现在是运行状态
81         break;
82     case RUNNING_STATE:
83         //do nothing 已经是运行状态了
84         break;
85     case STOPPING_STATE:
86         System.out.println("电梯开始运行了。。。");
87         this.setState(RUNNING_STATE);
88         break;
89     }
90 }
91
92 //执行停止动作
93 @Override
94 public void stop() {
95     switch (this.state) {
96         case OPENING_STATE: //开门的电梯已经是是停止的了(正常情况下)
97             //do nothing
98             break;
99         case CLOSING_STATE://关门时才可以停止
100            System.out.println("电梯停止了。。。");
101            this.setState(STOPPING_STATE);
102            break;
103         case RUNNING_STATE://运行时当然可以停止了
104            System.out.println("电梯停止了。。。");
105            this.setState(STOPPING_STATE);
106            break;
107         case STOPPING_STATE:
108            //do nothing
109            break;
110     }
111 }
112 }
113
114 public class Client {
115     public static void main(String[] args) {
116         Lift lift = new Lift();
117         lift.setState(ILift.STOPPING_STATE);//电梯是停止的
118         lift.open();//开门
119         lift.close();//关门
120         lift.run();//运行
121         lift.stop();//停止
122     }
123 }

```

问题分析:

- 使用了大量的switch...case这样的判断 (if...else也是一样), 使程序的可阅读性变差。
- 扩展性很差。如果新加了断电的状态, 我们需要修改上面判断逻辑

定义:

对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。

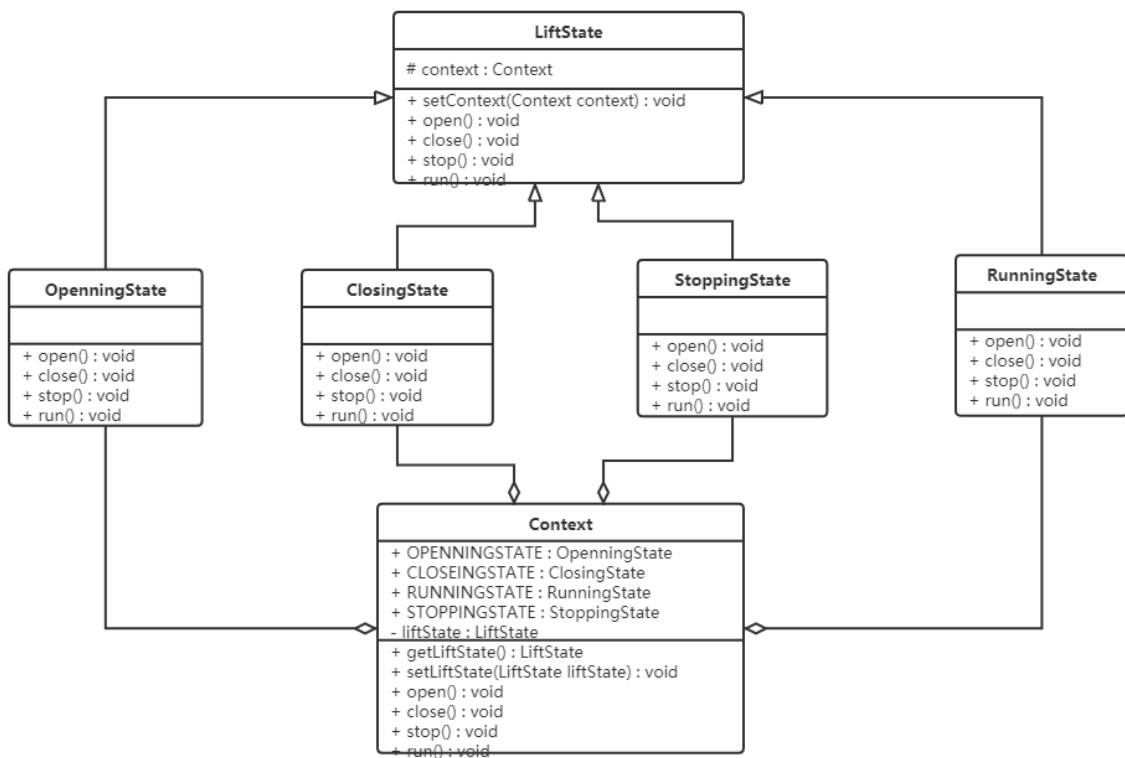
## 6.5.2 结构

状态模式包含以下主要角色。

- 环境 (Context) 角色：也称为上下文，它定义了客户程序需要的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理。
- 抽象状态 (State) 角色：定义一个接口，用以封装环境对象中的特定状态所对应的行为。
- 具体状态 (Concrete State) 角色：实现抽象状态所对应的行为。

## 6.5.3 案例实现

对上述电梯的案例使用状态模式进行改进。类图如下：



代码如下：

```
1 //抽象状态类
2 public abstract class LiftState {
3     //定义一个环境角色，也就是封装状态的变化引起的功能变化
4     protected Context context;
5
6     public void setContext(Context context) {
7         this.context = context;
8     }
9
10    //电梯开门动作
11    public abstract void open();
12
13    //电梯关门动作
```

```

14     public abstract void close();
15
16     //电梯运行动作
17     public abstract void run();
18
19     //电梯停止动作
20     public abstract void stop();
21 }
22
23 //开启状态
24 public class OpeningState extends LiftState {
25
26     //开启当然可以关闭了，我就想测试一下电梯门开关功能
27     @Override
28     public void open() {
29         System.out.println("电梯门开启...");
30     }
31
32     @Override
33     public void close() {
34         //状态修改
35         super.context.setLiftState(Context.closeingState);
36         //动作委托为CloseState来执行，也就是委托给了ClosingState子类执行这个动作
37         super.context.getLiftState().close();
38     }
39
40     //电梯门不能开着就跑，这里什么也不做
41     @Override
42     public void run() {
43         //do nothing
44     }
45
46     //开门状态已经是停止的了
47     @Override
48     public void stop() {
49         //do nothing
50     }
51 }
52
53 //运行状态
54 public class RunningState extends LiftState {
55
56     //运行的时候开电梯门？你疯了！电梯不会给你开的
57     @Override
58     public void open() {
59         //do nothing
60     }
61
62     //电梯门关闭？这是肯定了
63     @Override
64     public void close() { //虽然可以关门，但这个动作不归我执行
65         //do nothing
66     }
67
68     //这是在运行状态下要实现的方法
69     @Override
70     public void run() {
71         System.out.println("电梯正在运行...");

```

```

72     }
73
74     //这个事绝对是合理的，光运行不停止还有谁敢做这个电梯?! 估计只有上帝了
75     @Override
76     public void stop() {
77         super.context.setLiftState(Context.stoppingState);
78         super.context.stop();
79     }
80 }
81
82 //停止状态
83 public class StoppingState extends LiftState {
84
85     //停止状态，开门，那是要的!
86     @Override
87     public void open() {
88         //状态修改
89         super.context.setLiftState(Context.openningState);
90         //动作委托为CloseState来执行，也就是委托给了ClosingState子类执行这个动作
91         super.context.getLiftState().open();
92     }
93
94     @Override
95     public void close() { //虽然可以关门，但这个动作不归我执行
96         //状态修改
97         super.context.setLiftState(Context.closeingState);
98         //动作委托为CloseState来执行，也就是委托给了ClosingState子类执行这个动作
99         super.context.getLiftState().close();
100    }
101
102    //停止状态再跑起来，正常的很
103    @Override
104    public void run() {
105        //状态修改
106        super.context.setLiftState(Context.runningState);
107        //动作委托为CloseState来执行，也就是委托给了ClosingState子类执行这个动作
108        super.context.getLiftState().run();
109    }
110
111    //停止状态是怎么发生的呢? 当然是停止方法执行了
112    @Override
113    public void stop() {
114        system.out.println("电梯停止了...");
115    }
116 }
117
118 //关闭状态
119 public class ClosingState extends LiftState {
120
121     @Override
122     //电梯门关闭，这是关闭状态要实现的动作
123     public void close() {
124         system.out.println("电梯门关闭...");
125     }
126
127     //电梯门关了再打开，逗你玩呢，那这个允许呀
128     @Override
129     public void open() {

```

```

130         super.context.setLiftState(Context.openningState);
131         super.context.open();
132     }
133
134
135     //电梯门关了就跑，这是再正常不过了
136     @Override
137     public void run() {
138         super.context.setLiftState(Context.runningState);
139         super.context.run();
140     }
141
142     //电梯门关着，我就不按楼层
143     @Override
144     public void stop() {
145         super.context.setLiftState(Context.stoppingState);
146         super.context.stop();
147     }
148 }
149
150 //环境角色
151 public class Context {
152     //定义出所有的电梯状态
153     public final static OpenningState openningState = new
OpenningState();//开门状态，这时候电梯只能关闭
154     public final static ClosingState closeingState = new ClosingState();//
关闭状态，这时候电梯可以运行、停止和开门
155     public final static RunningState runningState = new RunningState();//
运行状态，这时候电梯只能停止
156     public final static StoppingState stoppingState = new
StoppingState();//停止状态，这时候电梯可以开门、运行
157
158
159     //定义一个当前电梯状态
160     private LiftState liftState;
161
162     public LiftState getLiftState() {
163         return this.liftState;
164     }
165
166     public void setLiftState(LiftState liftState) {
167         //当前环境改变
168         this.liftState = liftState;
169         //把当前的环境通知到各个实现类中
170         this.liftState.setContext(this);
171     }
172
173     public void open() {
174         this.liftState.open();
175     }
176
177     public void close() {
178         this.liftState.close();
179     }
180
181     public void run() {
182         this.liftState.run();
183     }

```

```

184
185     public void stop() {
186         this.liftState.stop();
187     }
188 }
189
190 //测试类
191 public class Client {
192     public static void main(String[] args) {
193         Context context = new Context();
194         context.setLiftState(new ClosingState());
195
196         context.open();
197         context.close();
198         context.run();
199         context.stop();
200     }
201 }

```

## 6.5.4 优缺点

### 1, 优点:

- 将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。
- 允许状态转换逻辑与状态对象合成一体，而不是某一个巨大的条件语句块。

### 2, 缺点:

- 状态模式的使用必然会增加系统类和对象的个数。
- 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。
- 状态模式对"开闭原则"的支持并不太好。

## 6.5.5 使用场景

- 当一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为时，就可以考虑使用状态模式。
- 一个操作中含有庞大的分支结构，并且这些分支决定于对象的状态时。

## 6.6 观察者模式

### 6.6.1 概述

#### 定义:

又被称为发布-订阅 (Publish/Subscribe) 模式，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

### 6.6.2 结构



在观察者模式中有如下角色：

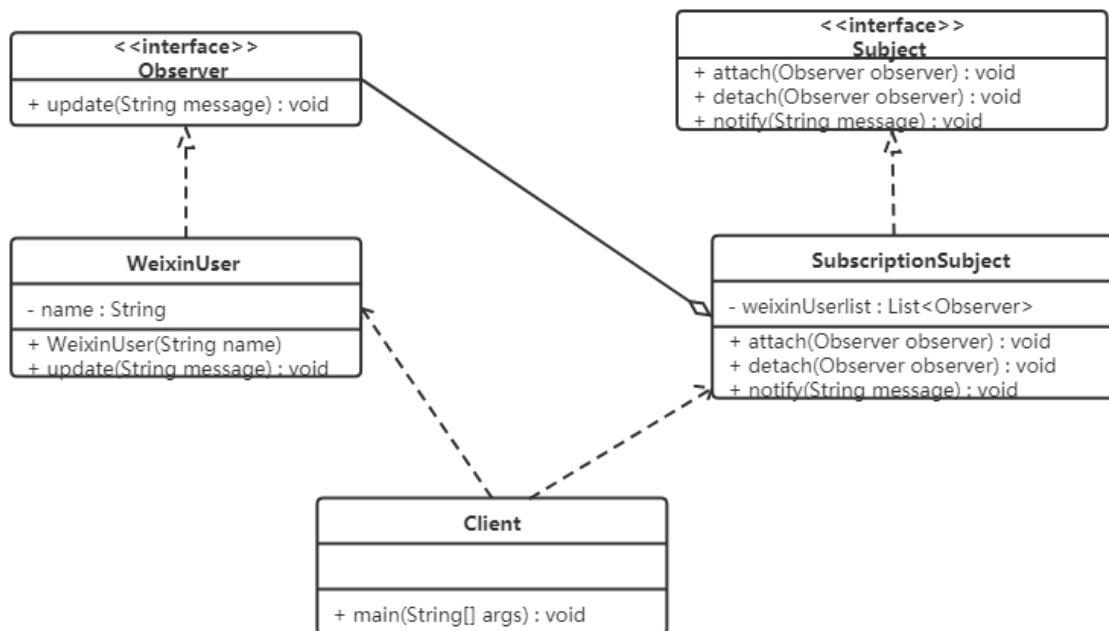
- Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。
- ConcreteSubject：具体主题（具体被观察者），该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。
- Observer：抽象观察者，是观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。
- ConcreteObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

### 6.6.3 案例实现

【例】微信公众号

在使用微信公众号时，大家都会有这样的体验，当你关注的公众号中有新内容更新的话，它就会推送给关注公众号的微信用户端。我们使用观察者模式来模拟这样的场景，微信用户就是观察者，微信公众号是被观察者，有多个的微信用户关注了程序猿这个公众号。

类图如下：



代码如下：

定义抽象观察者类，里面定义一个更新的方法

```
1 public interface Observer {
2     void update(String message);
3 }
```

定义具体观察者类，微信用户是观察者，里面实现了更新的方法

```

1 public class WeixinUser implements Observer {
2     // 微信用户名
3     private String name;
4
5     public WeixinUser(String name) {
6         this.name = name;
7     }
8     @Override
9     public void update(String message) {
10        System.out.println(name + "-" + message);
11    }
12 }

```

定义抽象主题类，提供了attach、detach、notify三个方法

```

1 public interface Subject {
2     //增加订阅者
3     public void attach(Observer observer);
4
5     //删除订阅者
6     public void detach(Observer observer);
7
8     //通知订阅者更新消息
9     public void notify(String message);
10 }
11

```

微信公众号是具体主题（具体被观察者），里面存储了订阅该公众号的微信用户，并实现了抽象主题中的方法

```

1 public class SubscriptionSubject implements Subject {
2     //储存订阅公众号的微信用户
3     private List<Observer> weixinUserlist = new ArrayList<Observer>();
4
5     @Override
6     public void attach(Observer observer) {
7         weixinUserlist.add(observer);
8     }
9
10    @Override
11    public void detach(Observer observer) {
12        weixinUserlist.remove(observer);
13    }
14
15    @Override
16    public void notify(String message) {
17        for (Observer observer : weixinUserlist) {
18            observer.update(message);
19        }
20    }
21 }

```

客户端程序

```

1 public class Client {

```

```

2     public static void main(String[] args) {
3         SubscriptionSubject mSubscriptionSubject=new SubscriptionSubject();
4         //创建微信用户
5         WeixinUser user1=new WeixinUser("孙悟空");
6         WeixinUser user2=new WeixinUser("猪悟能");
7         WeixinUser user3=new WeixinUser("沙悟净");
8         //订阅公众号
9         mSubscriptionSubject.attach(user1);
10        mSubscriptionSubject.attach(user2);
11        mSubscriptionSubject.attach(user3);
12        //公众号更新发出消息给订阅的微信用户
13        mSubscriptionSubject.notify("传智黑马的专栏更新了");
14    }
15 }
16

```

## 6.6.4 优缺点

### 1, 优点:

- 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
- 被观察者发送通知，所有注册的观察者都会收到信息【可以实现广播机制】

### 2, 缺点:

- 如果观察者非常多的话，那么所有的观察者收到被观察者发送的通知会耗时
- 如果被观察者有循环依赖的话，那么被观察者发送通知会使观察者循环调用，会导致系统崩溃

## 6.6.5 使用场景

- 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
- 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时。

## 6.6.6 JDK中提供的实现

在 Java 中, 通过 `java.util.Observable` 类和 `java.util.Observer` 接口定义了观察者模式, 只要实现它们的子类就可以编写观察者模式实例。

### 1, Observable类

`Observable` 类是抽象目标类 (被观察者), 它有一个 `Vector` 集合成员变量, 用于保存所有要通知的观察者对象, 下面来介绍它最重要的 3 个方法。

- `void addObserver(Observer o)` 方法: 用于将新的观察者对象添加到集合中。
- `void notifyObservers(Object arg)` 方法: 调用集合中的所有观察者对象的 `update` 方法, 通知它们数据发生改变。通常越晚加入集合的观察者越先得到通知。
- `void setChange()` 方法: 用来设置一个 `boolean` 类型的内部标志, 注明目标对象发生了变化。当它为`true`时, `notifyObservers()` 才会通知观察者。

### 2, Observer 接口

`Observer` 接口是抽象观察者, 它监视目标对象的变化, 当目标对象发生变化时, 观察者得到通知, 并调用 `update` 方法, 进行相应的工作。

## 【例】警察抓小偷

警察抓小偷也可以使用观察者模式来实现，警察是观察者，小偷是被观察者。代码如下：

小偷是一个被观察者，所以需要继承Observable类

```
1 public class Thief extends Observable {
2
3     private String name;
4
5     public Thief(String name) {
6         this.name = name;
7     }
8
9     public void setName(String name) {
10        this.name = name;
11    }
12
13    public String getName() {
14        return name;
15    }
16
17    public void steal() {
18        System.out.println("小偷：我偷东西了，有没有人来抓我!!!");
19        super.setChanged(); //changed = true
20        super.notifyObservers();
21    }
22 }
23
```

警察是一个观察者，所以需要让其实现Observer接口

```
1 public class Policemen implements Observer {
2
3     private String name;
4
5     public Policemen(String name) {
6         this.name = name;
7     }
8     public void setName(String name) {
9         this.name = name;
10    }
11
12    public String getName() {
13        return name;
14    }
15
16    @Override
17    public void update(Observable o, Object arg) {
18        System.out.println("警察：" + ((Thief) o).getName() + "，我已经盯你很久了，你可以保持沉默，但你所说的将成为呈堂证供!!!");
19    }
20 }
```

客户端代码

```

1 public class Client {
2     public static void main(String[] args) {
3         //创建小偷对象
4         Thief t = new Thief("隔壁老王");
5         //创建警察对象
6         Policemen p = new Policemen("小李");
7         //让警察盯着小偷
8         t.addObserver(p);
9         //小偷偷东西
10        t.steal();
11    }
12 }

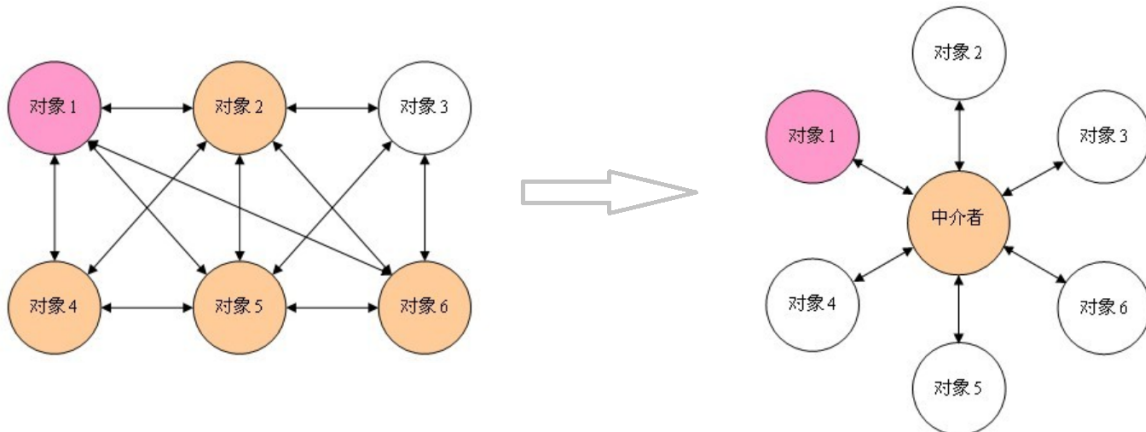
```

## 6.7 中介者模式

### 6.7.1 概述

一般来说，同事类之间的关系是比较复杂的，多个同事类之间互相关联时，他们之间的关系会呈现为复杂的网状结构，这是一种过度耦合的架构，即不利于类的复用，也不稳定。例如在下左图中，有六个同事类对象，假如对象1发生变化，那么将会有4个对象受到影响。如果对象2发生变化，那么将会有5个对象受到影响。也就是说，同事类之间直接关联的设计是不好的。

如果引入中介者模式，那么同事类之间的关系将变为星型结构，从下右图中可以看到，任何一个类的变动，只会影响的类本身，以及中介者，这样就减小了系统的耦合。一个好的设计，必定不会把所有的对象关系处理逻辑封装在本类中，而是使用一个专门的类来管理那些不属于自己的行为。



**定义：**

又叫调停模式，定义一个中介角色来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。

### 6.7.2 结构

中介者模式包含以下主要角色：

- 抽象中介者 (Mediator) 角色：它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。
- 具体中介者 (ConcreteMediator) 角色：实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。

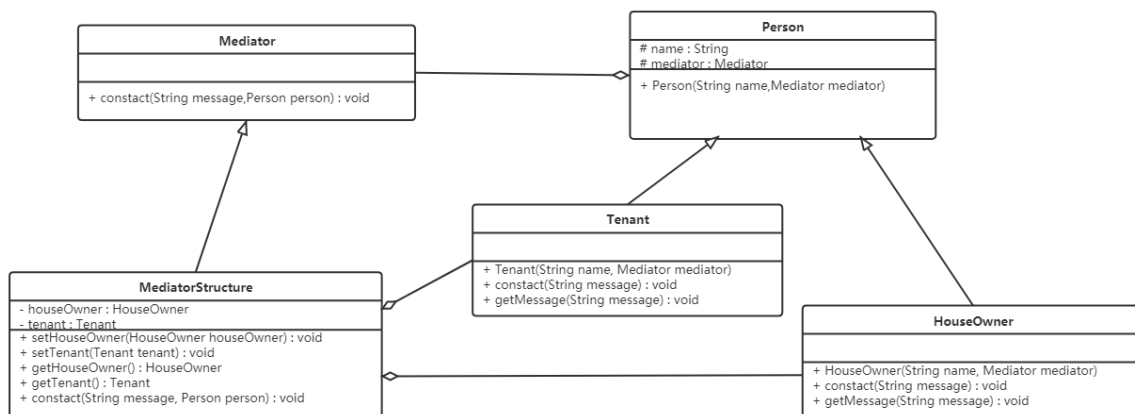
- 抽象同事类 (Colleague) 角色：定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。
- 具体同事类 (Concrete Colleague) 角色：是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。

### 6.7.3 案例实现

#### 【例】租房

现在租房基本都是通过房屋中介，房主将房屋托管给房屋中介，而租房者从房屋中介获取房屋信息。房屋中介充当租房者与房屋所有者之间的中介者。

类图如下：



代码如下：

```

1 //抽象中介者
2 public abstract class Mediator {
3     //申明一个联络方法
4     public abstract void constact(String message, Person person);
5 }
6
7 //抽象同事类
8 public abstract class Person {
9     protected String name;
10    protected Mediator mediator;
11
12    public Person(String name, Mediator mediator){
13        this.name = name;
14        this.mediator = mediator;
15    }
16 }
17
18 //具体同事类 房屋拥有者
19 public class HouseOwner extends Person {
20
21    public HouseOwner(String name, Mediator mediator) {
22        super(name, mediator);
23    }
24
25    //与中介者联系
26    public void constact(String message){
27        mediator.constact(message, this);
28    }
29 }
  
```

```

28     }
29
30     //获取信息
31     public void getMessage(String message){
32         System.out.println("房主" + name +"获取到的信息: " + message);
33     }
34 }
35
36 //具体同事类 承租人
37 public class Tenant extends Person {
38     public Tenant(String name, Mediator mediator) {
39         super(name, mediator);
40     }
41
42     //与中介者联系
43     public void constact(String message){
44         mediator.constact(message, this);
45     }
46
47     //获取信息
48     public void getMessage(String message){
49         System.out.println("租房者" + name +"获取到的信息: " + message);
50     }
51 }
52
53 //中介机构
54 public class MediatorStructure extends Mediator {
55     //首先中介结构必须知道所有房主和租房者的信息
56     private HouseOwner houseOwner;
57     private Tenant tenant;
58
59     public HouseOwner getHouseOwner() {
60         return houseOwner;
61     }
62
63     public void setHouseOwner(HouseOwner houseOwner) {
64         this.houseOwner = houseOwner;
65     }
66
67     public Tenant getTenant() {
68         return tenant;
69     }
70
71     public void setTenant(Tenant tenant) {
72         this.tenant = tenant;
73     }
74
75     public void constact(String message, Person person) {
76         if (person == houseOwner) {           //如果是房主，则租房者获得信息
77             tenant.getMessage(message);
78         } else {           //反正则是房主获得信息
79             houseOwner.getMessage(message);
80         }
81     }
82 }
83
84 //测试类
85 public class Client {

```

```

86     public static void main(String[] args) {
87         //一个房主、一个租房者、一个中介机构
88         MediatorStructure mediator = new MediatorStructure();
89
90         //房主和租房者只需要知道中介机构即可
91         HouseOwner houseOwner = new HouseOwner("张三", mediator);
92         Tenant tenant = new Tenant("李四", mediator);
93
94         //中介结构要知道房主和租房者
95         mediator.setHouseOwner(houseOwner);
96         mediator.setTenant(tenant);
97
98         tenant.constact("需要租三室的房子");
99         houseOwner.constact("我这三室的房子，你需要租吗？");
100     }
101 }

```

## 6.7.4 优缺点

### 1, 优点:

- 松散耦合

中介者模式通过把多个同事对象之间的交互封装到中介者对象里面，从而使得同事对象之间松散耦合，基本上可以做到互补依赖。这样一来，同事对象就可以独立地变化和复用，而不再像以前那样“牵一处而动全身”了。

- 集中控制交互

多个同事对象的交互，被封装在中介者对象里面集中管理，使得这些交互行为发生变化的时候，只需要修改中介者对象就可以了，当然如果是已经做好的系统，那么就扩展中介者对象，而各个同事类不需要做修改。

- 一对多关联转变为一对一的关联

没有使用中介者模式的时候，同事对象之间的关系通常是一对多的，引入中介者对象以后，中介者对象和同事对象的关系通常变成双向的一对一，这会让对象的关系更容易理解和实现。

### 2, 缺点:

当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

## 6.7.5 使用场景

- 系统中对象之间存在复杂的引用关系，系统结构混乱且难以理解。
- 当想创建一个运行于多个类之间的对象，又不想生成新的子类时。

## 6.8 迭代器模式

### 6.8.1 概述

定义:



提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

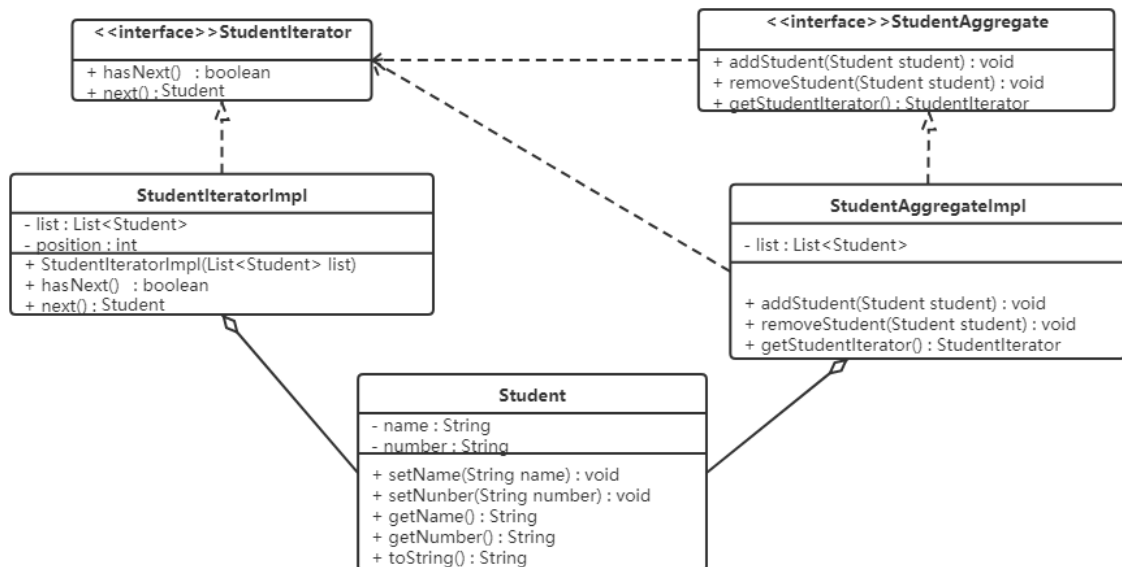
## 6.8.2 结构

迭代器模式主要包含以下角色：

- 抽象聚合 (Aggregate) 角色：定义存储、添加、删除聚合元素以及创建迭代器对象的接口。
- 具体聚合 (ConcreteAggregate) 角色：实现抽象聚合类，返回一个具体迭代器的实例。
- 抽象迭代器 (Iterator) 角色：定义访问和遍历聚合元素的接口，通常包含 `hasNext()`、`next()` 等方法。
- 具体迭代器 (ConcreteIterator) 角色：实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历，记录遍历的当前位置。

## 6.8.3 案例实现

【例】定义一个可以存储学生对象的容器对象，将遍历该容器的功能交由迭代器实现，涉及到的类如下：



代码如下：

定义迭代器接口，声明`hasNext`、`next`方法

```
1 public interface StudentIterator {
2     boolean hasNext();
3     Student next();
4 }
```

定义具体的迭代器类，重写所有的抽象方法

```
1 public class StudentIteratorImpl implements StudentIterator {
2     private List<Student> list;
3     private int position = 0;
4
5     public StudentIteratorImpl(List<Student> list) {
```

```

6         this.list = list;
7     }
8
9     @Override
10    public boolean hasNext() {
11        return position < list.size();
12    }
13
14    @Override
15    public Student next() {
16        Student currentStudent = list.get(position);
17        position ++;
18        return currentStudent;
19    }
20 }

```

定义抽象容器类，包含添加元素，删除元素，获取迭代器对象的方法

```

1 public interface StudentAggregate {
2     void addStudent(Student student);
3
4     void removeStudent(Student student);
5
6     StudentIterator getStudentIterator();
7 }

```

定义具体的容器类，重写所有的方法

```

1 public class StudentAggregateImpl implements StudentAggregate {
2
3     private List<Student> list = new ArrayList<Student>(); // 学生列表
4
5     @Override
6     public void addStudent(Student student) {
7         this.list.add(student);
8     }
9
10    @Override
11    public void removeStudent(Student student) {
12        this.list.remove(student);
13    }
14
15    @Override
16    public StudentIterator getStudentIterator() {
17        return new StudentIteratorImpl(list);
18    }
19 }

```

## 6.8.4 优缺点

### 1, 优点:

- 它支持以不同的方式遍历一个聚合对象，在同一个聚合对象上可以定义多种遍历方式。在迭代器模式中只需要用一个不同的迭代器来替换原有迭代器即可改变遍历算法，我们也可以自己定义迭代器

的子类以支持新的遍历方式。

- 迭代器简化了聚合类。由于引入了迭代器，在原有的聚合对象中不需要再自行提供数据遍历等方法，这样可以简化聚合类的设计。
- 在迭代器模式中，由于引入了抽象层，增加新的聚合类和迭代器类都很方便，无须修改原有代码，满足“开闭原则”的要求。

## 2, 缺点:

增加了类的个数，这在一定程度上增加了系统的复杂性。

## 6.8.5 使用场景

- 当需要为聚合对象提供多种遍历方式时。
- 当需要为遍历不同的聚合结构提供一个统一的接口时。
- 当访问一个聚合对象的内容而无须暴露其内部细节的表示时。

## 6.8.6 JDK源码解析

迭代器模式在JAVA的很多集合类中被广泛应用，接下来看看JAVA源码中是如何使用迭代器模式的。

```
1 List<String> list = new ArrayList<>();
2 Iterator<String> iterator = list.iterator(); //list.iterator()方法返回的肯定是
  Iterator接口的子实现类对象
3 while (iterator.hasNext()) {
4     System.out.println(iterator.next());
5 }
```

看完这段代码是不是很熟悉，与我们上面代码基本类似。单列集合都使用到了迭代器，我们以ArrayList举例来说明

- List: 抽象聚合类
- ArrayList: 具体的聚合类
- Iterator: 抽象迭代器
- list.iterator(): 返回的是实现了 `Iterator` 接口的具体迭代器对象

具体的来看看 ArrayList的代码实现

```
1 public class ArrayList<E> extends AbstractList<E>
2     implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
3
4     public Iterator<E> iterator() {
5         return new Itr();
6     }
7
8     private class Itr implements Iterator<E> {
9         int cursor;           // 下一个要返回元素的索引
10        int lastRet = -1; // 上一个返回元素的索引
11        int expectedModCount = modCount;
12
13        Itr() {}
14
15        //判断是否还有元素
16        public boolean hasNext() {
```

```

17         return cursor != size;
18     }
19
20     //获取下一个元素
21     public E next() {
22         checkForComodification();
23         int i = cursor;
24         if (i >= size)
25             throw new NoSuchElementException();
26         Object[] elementData = ArrayList.this.elementData;
27         if (i >= elementData.length)
28             throw new ConcurrentModificationException();
29         cursor = i + 1;
30         return (E) elementData[lastRet = i];
31     }
32     ...
33 }

```

这部分代码还是比较简单，大致就是在 `iterator` 方法中返回了一个实例化的 `Iterator` 对象。`Itr`是一个内部类，它实现了 `Iterator` 接口并重写了其中的抽象方法。

注意：

当我们在使用JAVA开发的时候，想使用迭代器模式的话，只要让我们自己定义的容器类实现 `java.util.Iterable` 并实现其中的 `iterator()` 方法使其返回一个 `java.util.Iterator` 的实现类就可以了。

## 6.9 访问者模式

### 6.9.1 概述

定义：

封装一些作用于某种数据结构中的各元素的操作，它可以在不改变这个数据结构的前提下定义作用于这些元素的新的操作。

### 6.9.2 结构

访问者模式包含以下主要角色：

- 抽象访问者 (Visitor) 角色：定义了对每一个元素 (Element) 访问的行为，它的参数就是可以访问的元素，它的方法个数理论上讲与元素类个数 (Element的实现类个数) 是一样的，从这点不难看出，访问者模式要求元素类的个数不能改变。
- 具体访问者 (ConcreteVisitor) 角色：给出对每一个元素类访问时所产生的具体行为。
- 抽象元素 (Element) 角色：定义了一个接受访问者的方法 (accept)，其意义是指，每一个元素都要可以被访问者访问。
- 具体元素 (ConcreteElement) 角色：提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法。
- 对象结构 (Object Structure) 角色：定义当中所提到的对象结构，对象结构是一个抽象表述，具体点可以理解为一个具有容器性质或者复合对象特性的类，它会含有一组元素 (Element)，并且可以迭代这些元素，供访问者访问。

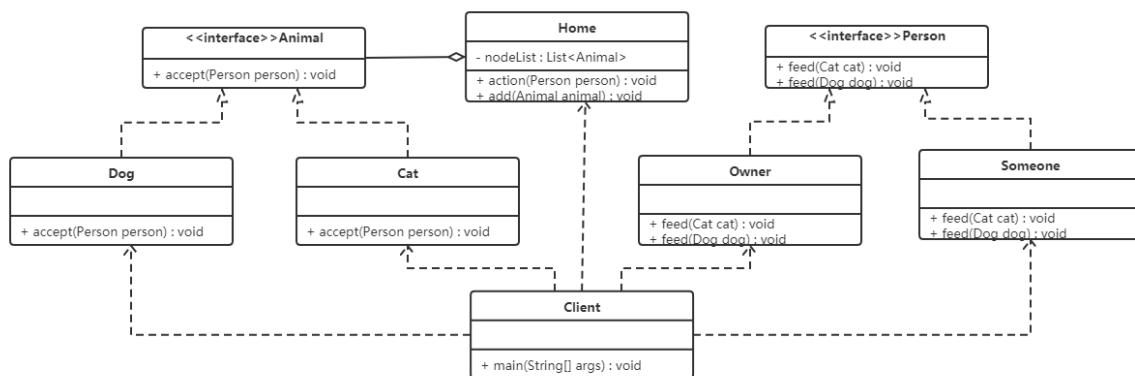
### 6.9.3 案例实现

#### 【例】给宠物喂食

现在养宠物的人特别多，我们就以这个为例，当然宠物还分为狗，猫等，要给宠物喂食的话，主人可以喂，其他人也可以喂食。

- 访问者角色：给宠物喂食的人
- 具体访问者角色：主人、其他人
- 抽象元素角色：动物抽象类
- 具体元素角色：宠物狗、宠物猫
- 结构对象角色：主人家

类图如下：



代码如下：

创建抽象访问者接口

```
1 public interface Person {
2     void feed(Cat cat);
3
4     void feed(Dog dog);
5 }
```

创建不同的具体访问者角色（主人和其他人），都需要实现 Person 接口

```
1 public class Owner implements Person {
2
3     @Override
4     public void feed(Cat cat) {
5         System.out.println("主人喂食猫");
6     }
7
8     @Override
9     public void feed(Dog dog) {
10        System.out.println("主人喂食狗");
11    }
12 }
13
14 public class Someone implements Person {
15     @Override
16     public void feed(Cat cat) {
```

```

17     System.out.println("其他人喂食猫");
18     }
19
20     @Override
21     public void feed(Dog dog) {
22         System.out.println("其他人喂食狗");
23     }
24 }

```

定义抽象节点 -- 宠物

```

1 public interface Animal {
2     void accept(Person person);
3 }

```

定义实现 Animal 接口的 具体节点 (元素)

```

1 public class Dog implements Animal {
2
3     @Override
4     public void accept(Person person) {
5         person.feed(this);
6         System.out.println("好好吃, 汪汪汪!!!");
7     }
8 }
9
10 public class Cat implements Animal {
11
12     @Override
13     public void accept(Person person) {
14         person.feed(this);
15         System.out.println("好好吃, 喵喵喵!!!");
16     }
17 }

```

定义对象结构, 此案例中就是主人的家

```

1 public class Home {
2     private List<Animal> nodeList = new ArrayList<Animal>();
3
4     public void action(Person person) {
5         for (Animal node : nodeList) {
6             node.accept(person);
7         }
8     }
9
10    //添加操作
11    public void add(Animal animal) {
12        nodeList.add(animal);
13    }
14 }
15

```

测试类

```

1 public class Client {
2     public static void main(String[] args) {
3         Home home = new Home();
4         home.add(new Dog());
5         home.add(new Cat());
6
7         Owner owner = new Owner();
8         home.action(owner);
9
10        Someone someone = new Someone();
11        home.action(someone);
12    }
13 }

```

## 6.9.4 优缺点

### 1, 优点:

- 扩展性好  
在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。
- 复用性好  
通过访问者来定义整个对象结构通用的功能，从而提高复用程度。
- 分离无关行为  
通过访问者来分离无关的行为，把相关的行为封装在一起，构成一个访问者，这样每一个访问者的功能都比较单一。

### 2, 缺点:

- 对象结构变化很困难  
在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”。
- 违反了依赖倒置原则  
访问者模式依赖了具体类，而没有依赖抽象类。

## 6.9.5 使用场景

- 对象结构相对稳定，但其操作算法经常变化的程序。
- 对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构。

## 6.9.6 扩展

访问者模式用到了一种双分派的技术。

### 1, 分派:

变量被声明时的类型叫做变量的静态类型，有些人又把静态类型叫做明显类型；而变量所引用的对象的真实类型又叫做变量的实际类型。比如 `Map map = new HashMap()`，`map`变量的静态类型是 `Map`，实际类型是 `HashMap`。根据对象的类型而对方法进行选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即静态分派和动态分派。

**静态分派(Static Dispatch)** 发生在编译时期，分派根据静态类型信息发生。静态分派对于我们来说并不陌生，方法重载就是静态分派。

**动态分派(Dynamic Dispatch)** 发生在运行时期，动态分派动态地替换掉某个方法。Java通过方法的重写支持动态分派。

## 2, 动态分派:

通过方法的重写支持动态分派。

```
1 public class Animal {
2     public void execute() {
3         System.out.println("Animal");
4     }
5 }
6
7 public class Dog extends Animal {
8     @Override
9     public void execute() {
10        System.out.println("dog");
11    }
12 }
13
14 public class Cat extends Animal {
15     @Override
16     public void execute() {
17        System.out.println("cat");
18    }
19 }
20
21 public class Client {
22     public static void main(String[] args) {
23         Animal a = new Dog();
24         a.execute();
25
26         Animal a1 = new Cat();
27         a1.execute();
28     }
29 }
```

上面代码的结果大家应该直接可以说出来，这不就是多态吗！运行执行的是子类中的方法。

Java编译器在编译时期并不总是知道哪些代码会被执行，因为编译器仅仅知道对象的静态类型，而不知道对象的真实类型；而方法的调用则是根据对象的真实类型，而不是静态类型。

## 3, 静态分派:

通过方法重载支持静态分派。

```
1 public class Animal {
2 }
3
4 public class Dog extends Animal {
```



```

5   }
6
7   public class Cat extends Animal {
8   }
9
10  public class Execute {
11      public void execute(Animal a) {
12          System.out.println("Animal");
13      }
14
15      public void execute(Dog d) {
16          System.out.println("dog");
17      }
18
19      public void execute(Cat c) {
20          System.out.println("cat");
21      }
22  }
23
24  public class Client {
25      public static void main(String[] args) {
26          Animal a = new Animal();
27          Animal a1 = new Dog();
28          Animal a2 = new Cat();
29
30          Execute exe = new Execute();
31          exe.execute(a);
32          exe.execute(a1);
33          exe.execute(a2);
34      }
35  }

```

运行结果:

```

D:\develop\java\jdk1.8\bin\java.exe ...
animal
animal
animal

```

这个结果可能出乎一些人的意料了，为什么呢？

**重载方法的分派是根据静态类型进行的，这个分派过程在编译时期就完成了。**

#### 4, 双分派:

所谓双分派技术就是在选择一个方法的时候，不仅仅要根据消息接收者 (receiver) 的运行时区别，还要根据参数的运行时区别。

```

1   public class Animal {
2       public void accept(Execute exe) {
3           exe.execute(this);
4       }
5   }
6
7   public class Dog extends Animal {
8       public void accept(Execute exe) {
9           exe.execute(this);
10      }

```

```

11 }
12
13 public class Cat extends Animal {
14     public void accept(Execute exe) {
15         exe.execute(this);
16     }
17 }
18
19 public class Execute {
20     public void execute(Animal a) {
21         System.out.println("animal");
22     }
23
24     public void execute(Dog d) {
25         System.out.println("dog");
26     }
27
28     public void execute(Cat c) {
29         System.out.println("cat");
30     }
31 }
32
33 public class Client {
34     public static void main(String[] args) {
35         Animal a = new Animal();
36         Animal d = new Dog();
37         Animal c = new Cat();
38
39         Execute exe = new Execute();
40         a.accept(exe);
41         d.accept(exe);
42         c.accept(exe);
43     }
44 }

```

在上面代码中，客户端将Execute对象做为参数传递给Animal类型的变量调用的方法，这里完成第一次分派，这里是方法重写，所以是动态分派，也就是执行实际类型中的方法，同时也将自己this作为参数传递进去，这里就完成了第二次分派，这里的Execute类中有多个重载的方法，而传递进行的是this，就是具体的实际类型的对象。

说到这里，我们已经明白双分派是怎么回事了，但是它有什么效果呢？就是可以实现方法的动态绑定，我们可以对上面的程序进行修改。

运行结果如下：

```

D:\develop\java\jdk1.8\bin\java.exe ...
animal
dog
cat

```

双分派实现动态绑定的本质，就是在重载方法委派的前面加上了继承体系中覆盖的环节，由于覆盖是动态的，所以重载就是动态的了。

## 6.10 备忘录模式

## 6.10.1 概述

备忘录模式提供了一种状态恢复的实现机制，使得用户可以方便地回到一个特定的历史步骤，当新的状态无效或者存在问题时，可以使用暂时存储起来的备忘录将状态复原，很多软件都提供了撤销

(Undo) 操作，如 Word、记事本、Photoshop、IDEA等软件在编辑时按 Ctrl+Z 组合键时能撤销当前操作，使文档恢复到之前的状态；还有在 浏览器 中的后退键、数据库事务管理中的回滚操作、玩游戏时的中间结果存档功能、数据库与操作系统的备份操作、棋类游戏悔棋功能等都属于这类。

**定义：**

又叫快照模式，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，以便以后当需要时能将该对象恢复到原先保存的状态。

## 6.10.2 结构

备忘录模式的主要角色如下：

- 发起人 (Originator) 角色：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。
- 备忘录 (Memento) 角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。
- 管理者 (Caretaker) 角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

备忘录有两个等效的接口：

- **窄接口**：管理者 (Caretaker) 对象 (和其他发起人对象之外的任何对象) 看到的是备忘录的窄接口 (narrow Interface)，这个窄接口只允许他把备忘录对象传给其他的对象。
- **宽接口**：与管理者看到的窄接口相反，发起人对象可以看到一个宽接口 (wide Interface)，这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态。

## 6.10.3 案例实现

**【例】** 游戏挑战BOSS

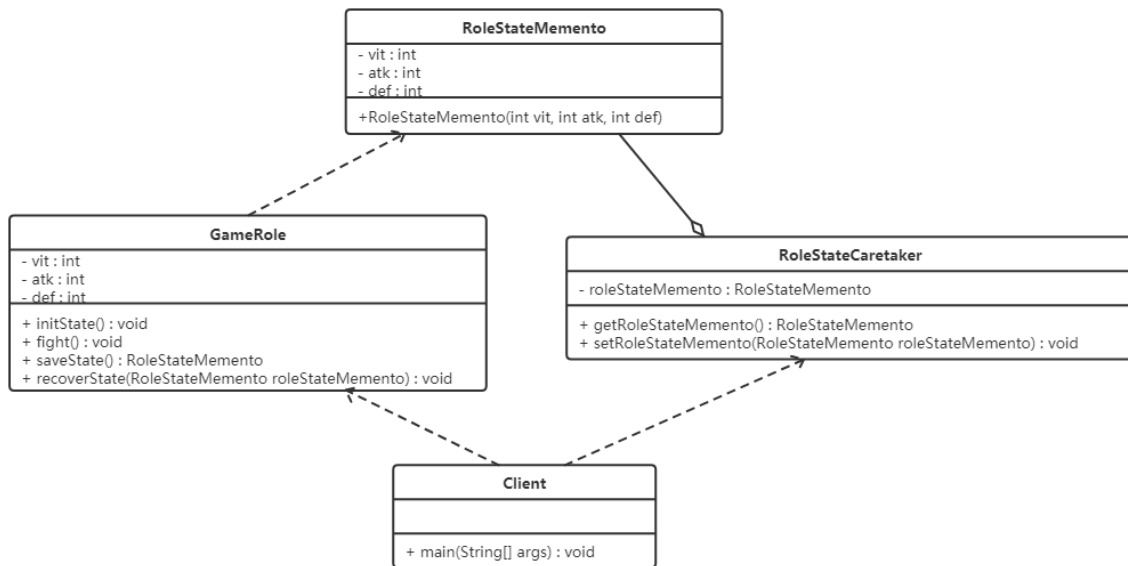
游戏中的某个场景，一游戏角色有生命力、攻击力、防御力等数据，在打Boss前和后一定会不一样的，我们允许玩家如果感觉与Boss决斗的效果不理想可以让游戏恢复到决斗之前的状态。

要实现上述案例，有两种方式：

- “白箱”备忘录模式
- “黑箱”备忘录模式

### 6.10.3.1 “白箱”备忘录模式

备忘录角色对任何对象都提供一个接口，即宽接口，备忘录角色的内部所存储的状态就对所有对象公开。类图如下：



代码如下:

```

1 //游戏角色类
2 public class GameRole {
3     private int vit; //生命力
4     private int atk; //攻击力
5     private int def; //防御力
6
7     //初始化状态
8     public void initState() {
9         this.vit = 100;
10        this.atk = 100;
11        this.def = 100;
12    }
13
14    //战斗
15    public void fight() {
16        this.vit = 0;
17        this.atk = 0;
18        this.def = 0;
19    }
20
21    //保存角色状态
22    public RoleStateMemento saveState() {
23        return new RoleStateMemento(vit, atk, def);
24    }
25
26    //回复角色状态
27    public void recoverState(RoleStateMemento roleStateMemento) {
28        this.vit = roleStateMemento.getVit();
29        this.atk = roleStateMemento.getAtk();
30        this.def = roleStateMemento.getDef();
31    }
32
33    public void stateDisplay() {
34        System.out.println("角色生命力: " + vit);
35        System.out.println("角色攻击力: " + atk);
36        System.out.println("角色防御力: " + def);
37    }
38
  
```

```
39     public int getVit() {
40         return vit;
41     }
42
43     public void setVit(int vit) {
44         this.vit = vit;
45     }
46
47     public int getAtk() {
48         return atk;
49     }
50
51     public void setAtk(int atk) {
52         this.atk = atk;
53     }
54
55     public int getDef() {
56         return def;
57     }
58
59     public void setDef(int def) {
60         this.def = def;
61     }
62 }
63
64 //游戏状态存储类(备忘录类)
65 public class RoleStateMemento {
66     private int vit;
67     private int atk;
68     private int def;
69
70     public RoleStateMemento(int vit, int atk, int def) {
71         this.vit = vit;
72         this.atk = atk;
73         this.def = def;
74     }
75
76     public int getVit() {
77         return vit;
78     }
79
80     public void setVit(int vit) {
81         this.vit = vit;
82     }
83
84     public int getAtk() {
85         return atk;
86     }
87
88     public void setAtk(int atk) {
89         this.atk = atk;
90     }
91
92     public int getDef() {
93         return def;
94     }
95
96     public void setDef(int def) {
```

```

97     this.def = def;
98     }
99 }
100
101 //角色状态管理者类
102 public class RoleStateCaretaker {
103     private RoleStateMemento roleStateMemento;
104
105     public RoleStateMemento getRoleStateMemento() {
106         return roleStateMemento;
107     }
108
109     public void setRoleStateMemento(RoleStateMemento roleStateMemento) {
110         this.roleStateMemento = roleStateMemento;
111     }
112 }
113
114 //测试类
115 public class Client {
116     public static void main(String[] args) {
117         System.out.println("-----大战Boss前-----");
118         //大战Boss前
119         GameRole gameRole = new GameRole();
120         gameRole.initState();
121         gameRole.stateDisplay();
122
123         //保存进度
124         RoleStateCaretaker roleStateCaretaker = new RoleStateCaretaker();
125         roleStateCaretaker.setRoleStateMemento(gameRole.saveState());
126
127         System.out.println("-----大战Boss后-----");
128         //大战Boss时，损耗严重
129         gameRole.fight();
130         gameRole.stateDisplay();
131         System.out.println("-----恢复之前状态-----");
132         //恢复之前状态
133         gameRole.recoverState(roleStateCaretaker.getRoleStateMemento());
134         gameRole.stateDisplay();
135
136     }
137 }

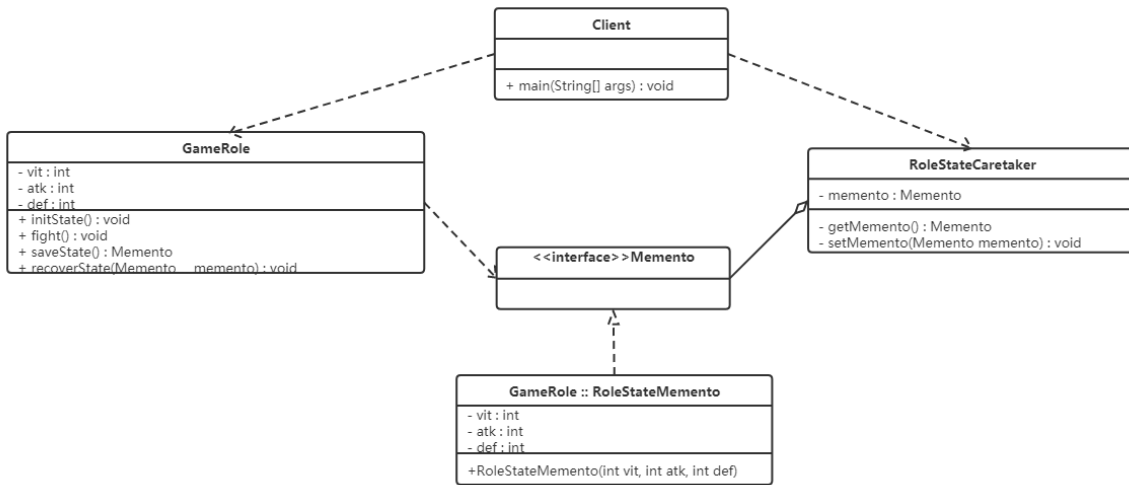
```

分析：白箱备忘录模式是破坏封装性的。但是通过程序员自律，同样可以在一定程度上实现模式的大部分用意。

### 6.10.3.2 “黑箱”备忘录模式

备忘录角色对发起人对象提供一个宽接口，而为其他对象提供一个窄接口。在Java语言中，实现双重接口的办法就是将**备忘录类**设计成**发起人类**的内部成员类。

将 `RoleStateMemento` 设为 `GameRole` 的内部类，从而将 `RoleStateMemento` 对象封装在 `GameRole` 里面；在外面提供一个标识接口 `Memento` 给 `RoleStateCaretaker` 及其他对象使用。这样 `GameRole` 类看到的是 `RoleStateMemento` 所有的接口，而 `RoleStateCaretaker` 及其他对象看到的仅仅是标识接口 `Memento` 所暴露出来的接口，从而维护了封装型。类图如下：



代码如下：

窄接口 Memento，这是一个标识接口，因此没有定义出任何的方法

```

1 public interface Memento {
2 }

```

定义发起人类 GameRole，并在内部定义备忘录内部类 RoleStateMemento（该内部类设置为私有的）

```

1 //游戏角色类
2 public class GameRole {
3     private int vit; //生命力
4     private int atk; //攻击力
5     private int def; //防御力
6
7     //初始化状态
8     public void initState() {
9         this.vit = 100;
10        this.atk = 100;
11        this.def = 100;
12    }
13
14    //战斗
15    public void fight() {
16        this.vit = 0;
17        this.atk = 0;
18        this.def = 0;
19    }
20
21    //保存角色状态
22    public Memento saveState() {
23        return new RoleStateMemento(vit, atk, def);
24    }
25
26    //回复角色状态
27    public void recoverState(Memento memento) {
28        RoleStateMemento roleStateMemento = (RoleStateMemento) memento;
29        this.vit = roleStateMemento.getVit();
30        this.atk = roleStateMemento.getAtk();
31        this.def = roleStateMemento.getDef();
32    }

```

```
33
34     public void stateDisplay() {
35         System.out.println("角色生命力: " + vit);
36         System.out.println("角色攻击力: " + atk);
37         System.out.println("角色防御力: " + def);
38
39     }
40
41     public int getVit() {
42         return vit;
43     }
44
45     public void setVit(int vit) {
46         this.vit = vit;
47     }
48
49     public int getAtk() {
50         return atk;
51     }
52
53     public void setAtk(int atk) {
54         this.atk = atk;
55     }
56
57     public int getDef() {
58         return def;
59     }
60
61     public void setDef(int def) {
62         this.def = def;
63     }
64
65     private class RoleStateMemento implements Memento {
66         private int vit;
67         private int atk;
68         private int def;
69
70         public RoleStateMemento(int vit, int atk, int def) {
71             this.vit = vit;
72             this.atk = atk;
73             this.def = def;
74         }
75
76         public int getVit() {
77             return vit;
78         }
79
80         public void setVit(int vit) {
81             this.vit = vit;
82         }
83
84         public int getAtk() {
85             return atk;
86         }
87
88         public void setAtk(int atk) {
89             this.atk = atk;
90         }

```



```

91     public int getDef() {
92         return def;
93     }
94
95
96     public void setDef(int def) {
97         this.def = def;
98     }
99 }
100 }

```

负责人角色类 `RoleStateCaretaker` 能够得到的备忘录对象是以 `Memento` 为接口的，由于这个接口仅仅是一个标识接口，因此负责人角色不可能改变这个备忘录对象的内容

```

1 //角色状态管理者类
2 public class RoleStateCaretaker {
3     private Memento memento;
4
5     public Memento getMemento() {
6         return memento;
7     }
8
9     public void setMemento(Memento memento) {
10        this.memento = memento;
11    }
12 }

```

客户端测试类

```

1 public class Client {
2     public static void main(String[] args) {
3         System.out.println("-----大战Boss前-----");
4         //大战Boss前
5         GameRole gameRole = new GameRole();
6         gameRole.initState();
7         gameRole.stateDisplay();
8
9         //保存进度
10        RoleStateCaretaker roleStateCaretaker = new RoleStateCaretaker();
11        roleStateCaretaker.setMemento(gameRole.saveState());
12
13        System.out.println("-----大战Boss后-----");
14        //大战Boss时，损耗严重
15        gameRole.fight();
16        gameRole.stateDisplay();
17        System.out.println("-----恢复之前状态-----");
18        //恢复之前状态
19        gameRole.recoverState(roleStateCaretaker.getMemento());
20        gameRole.stateDisplay();
21    }
22 }
23

```

## 6.10.4 优缺点

## 1, 优点:

- 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。
- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
- 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

## 2, 缺点:

- 资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

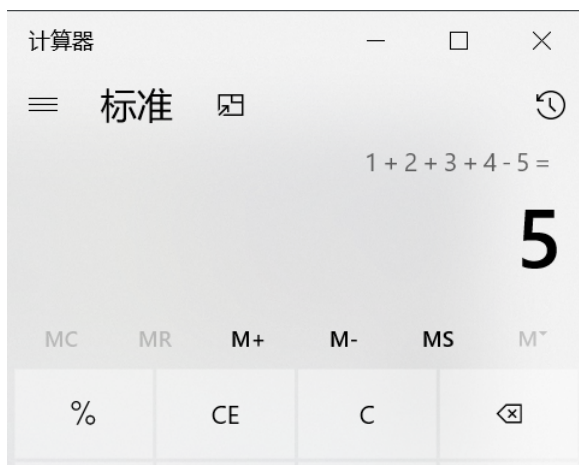
## 6.10.5 使用场景

- 需要保存与恢复数据的场景，如玩游戏时的中间结果的存档功能。
- 需要提供一个可回滚操作的场景，如 Word、记事本、Photoshop, idea等软件在编辑时按 Ctrl+Z 组合键，还有数据库中事务操作。

## 6, 行为型模式

### 6.11 解释器模式

#### 6.11.1 概述



如上图，设计一个软件用来进行加减计算。我们第一想法就是使用工具类，提供对应的加法和减法的工具方法。

```
1 //用于两个整数相加
2 public static int add(int a,int b){
3     return a + b;
4 }
5
6 //用于两个整数相加
7 public static int add(int a,int b,int c){
8     return a + b + c;
9 }
10
11 //用于n个整数相加
12 public static int add(Integer ... arr) {
13     int sum = 0;
14     for (Integer i : arr) {
15         sum += i;
16     }
17     return sum;
18 }
```

上面的形式比较单一、有限，如果形式变化非常多，这就不符合要求，因为加法和减法运算，两个运算符与数值可以有无限种组合方式。比如  $1+2+3+4+5$ 、 $1+2+3-4$  等等。

显然，现在需要一种翻译识别机器，能够解析由数字以及  $+$   $-$  符号构成的合法的运算序列。如果把运算符和数字都看作节点的话，能够逐个节点的进行读取解析运算，这就是解释器模式的思维。

#### 定义：

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

在解释器模式中，我们需要将待解决的问题，提取出规则，抽象为一种“语言”。比如加减法运算，规则为：由数值和+-符号组成的合法序列，“1+3-2”就是这种语言的句子。

解释器就是要解析出来语句的含义。但是如何描述规则呢？

### 文法 (语法) 规则:

文法是用来描述语言的语法结构的形式规则。

```
1 expression ::= value | plus | minus
2 plus ::= expression '+' expression
3 minus ::= expression '-' expression
4 value ::= integer
```

注意：这里的符号“::=”表示“定义为”的意思，竖线 | 表示或，左右的其中一个，引号内为字符本身，引号外为语法。

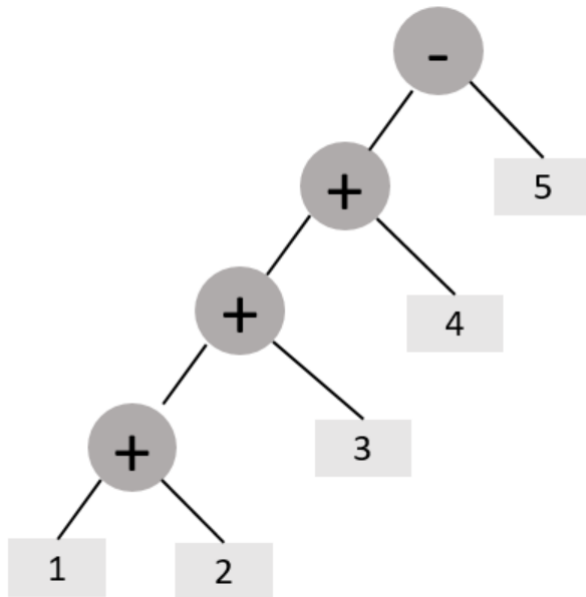
上面规则描述为：

表达式可以是一个值，也可以是plus或者minus运算，而plus和minus又是由表达式结合运算符构成，值的类型为整型数。

### 抽象语法树:

在计算机科学中，抽象语法树 (Abstract Syntax Tree, AST)，或简称语法树 (Syntax tree)，是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。

用树形来表示符合文法规则的句子。



## 6.11.2 结构

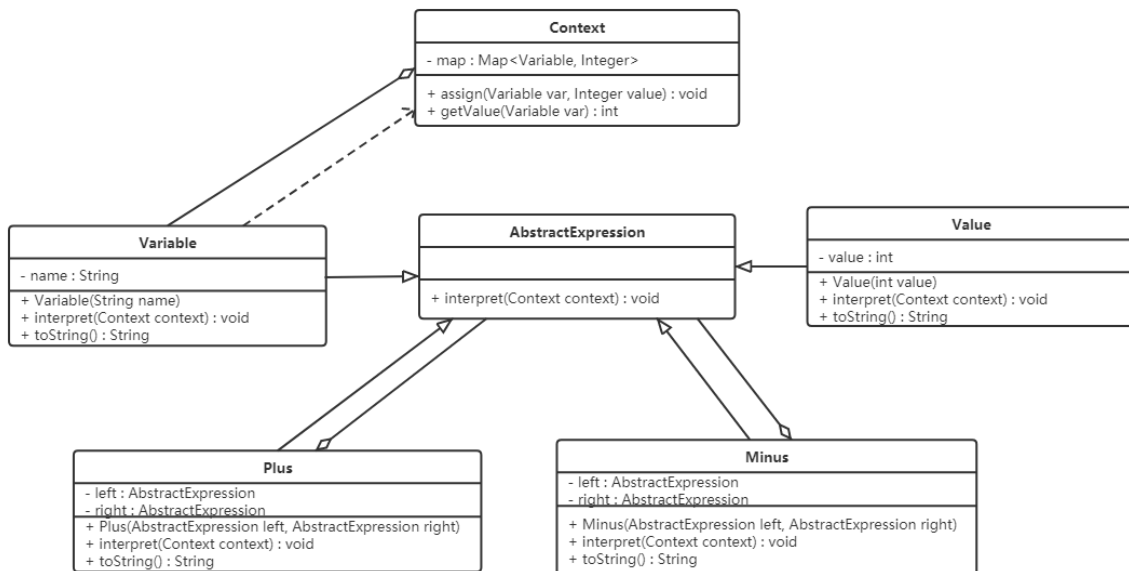
解释器模式包含以下主要角色。

- 抽象表达式 (Abstract Expression) 角色：定义解释器的接口，约定解释器的解释操作，主要包含解释方法 `interpret()`。
- 终结符表达式 (Terminal Expression) 角色：是抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应。
- 非终结符表达式 (Nonterminal Expression) 角色：也是抽象表达式的子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式。

- 环境 (Context) 角色: 通常包含各个解释器需要的数据或是公共的功能, 一般用来传递被所有解释器共享的数据, 后面的解释器可以从这里获取这些值。
- 客户端 (Client): 主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树, 然后调用解释器的解释方法, 当然也可以通过环境角色间接访问解释器的解释方法。

### 6.11.3 案例实现

【例】设计实现加减法的软件



代码如下:

```

1 //抽象角色AbstractExpression
2 public abstract class AbstractExpression {
3     public abstract int interpret(Context context);
4 }
5
6 //终结符表达式角色
7 public class Value extends AbstractExpression {
8     private int value;
9
10    public Value(int value) {
11        this.value = value;
12    }
13
14    @Override
15    public int interpret(Context context) {
16        return value;
17    }
18
19    @Override
20    public String toString() {
21        return new Integer(value).toString();
22    }
23 }
24
25 //非终结符表达式角色 加法表达式
26 public class Plus extends AbstractExpression {
27     private AbstractExpression left;
  
```

```

28     private AbstractExpression right;
29
30     public Plus(AbstractExpression left, AbstractExpression right) {
31         this.left = left;
32         this.right = right;
33     }
34
35     @Override
36     public int interpret(Context context) {
37         return left.interpret(context) + right.interpret(context);
38     }
39
40     @Override
41     public String toString() {
42         return "(" + left.toString() + " + " + right.toString() + ")";
43     }
44 }
45
46 ///非终结符表达式角色 减法表达式
47 public class Minus extends AbstractExpression {
48     private AbstractExpression left;
49     private AbstractExpression right;
50
51     public Minus(AbstractExpression left, AbstractExpression right) {
52         this.left = left;
53         this.right = right;
54     }
55
56     @Override
57     public int interpret(Context context) {
58         return left.interpret(context) - right.interpret(context);
59     }
60
61     @Override
62     public String toString() {
63         return "(" + left.toString() + " - " + right.toString() + ")";
64     }
65 }
66
67 //终结符表达式角色 变量表达式
68 public class Variable extends AbstractExpression {
69     private String name;
70
71     public Variable(String name) {
72         this.name = name;
73     }
74
75     @Override
76     public int interpret(Context ctx) {
77         return ctx.getValue(this);
78     }
79
80     @Override
81     public String toString() {
82         return name;
83     }
84 }
85

```

```

86 //环境类
87 public class Context {
88     private Map<Variable, Integer> map = new HashMap<Variable, Integer>();
89
90     public void assign(Variable var, Integer value) {
91         map.put(var, value);
92     }
93
94     public int getValue(Variable var) {
95         Integer value = map.get(var);
96         return value;
97     }
98 }
99
100 //测试类
101 public class Client {
102     public static void main(String[] args) {
103         Context context = new Context();
104
105         Variable a = new Variable("a");
106         Variable b = new Variable("b");
107         Variable c = new Variable("c");
108         Variable d = new Variable("d");
109         Variable e = new Variable("e");
110         //Value v = new Value(1);
111
112         context.assign(a, 1);
113         context.assign(b, 2);
114         context.assign(c, 3);
115         context.assign(d, 4);
116         context.assign(e, 5);
117
118         AbstractExpression expression = new Minus(new Plus(new Plus(new
Plus(a, b), c), d), e);
119
120         System.out.println(expression + "= " +
expression.interpret(context));
121     }
122 }

```

## 6.11.4 优缺点

### 1, 优点:

- 易于改变和扩展文法。

由于在解释器模式中使用类来表示语言的语法规则，因此可以通过继承等机制来改变或扩展文法。每一条语法规则都可以表示为一个类，因此可以方便地实现一个简单的语言。

- 实现文法较为容易。

在抽象语法树中每一个表达式节点类的实现方式都是相似的，这些类的代码编写都不会特别复杂。

- 增加新的解释表达式较为方便。

如果用户需要增加新的解释表达式只需要对应增加一个新的终结符表达式或非终结符表达式类，原有表达式类代码无须修改，符合“开闭原则”。

## 2, 缺点:

- 对于复杂文法难以维护。

在解释器模式中，每一条规则至少需要定义一个类，因此如果一个语言包含太多语法规则，类的个数将会急剧增加，导致系统难以管理和维护。

- 执行效率较低。

由于在解释器模式中使用了大量的循环和递归调用，因此在解释较为复杂的句子时其速度很慢，而且代码的调试过程也比较麻烦。

## 6.11.5 使用场景

- 当语言的文法较为简单，且执行效率不是关键问题时。
- 当问题重复出现，且可以用一种简单的语言来进行表达时。
- 当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候。

# 7, 自定义Spring框架

## 7.1 spring使用回顾

自定义spring框架前，先回顾一下spring框架的使用，从而分析spring的核心，并对核心功能进行模拟。

- 数据访问层。定义UserDao接口及其子实现类

```
1 public interface UserDao {
2     public void add();
3 }
4
5 public class UserDaoImpl implements UserDao {
6
7     public void add() {
8         System.out.println("userDaoImpl ....");
9     }
10 }
```

- 业务逻辑层。定义UserService接口及其子实现类

```
1 public interface UserService {
2     public void add();
3 }
4
5 public class UserServiceImpl implements UserService {
6
7     private UserDao userDao;
8
9     public void setUserDao(UserDao userDao) {
10         this.userDao = userDao;
11     }
12
13     public void add() {
```



```

14     System.out.println("userServiceImpl ...");
15     userDao.add();
16 }
17 }

```

- 定义UserController类，使用main方法模拟controller层

```

1 public class UserController {
2     public static void main(String[] args) {
3         //创建spring容器对象
4         ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
5         //从IOC容器中获取UserService对象
6         UserService userService =
applicationContext.getBean("userService", UserService.class);
7         //调用UserService对象的add方法
8         userService.add();
9     }
10 }

```

- 编写配置文件。在类路径下编写一个名为ApplicationContext.xml的配置文件

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://www.springframework.org/schema/beans"
4     xmlns:context="http://www.springframework.org/schema/context"
5     xsi:schemaLocation="http://www.springframework.org/schema/beans
6     http://www.springframework.org/schema/beans/spring-beans.xsd
7     http://www.springframework.org/schema/context
8     http://www.springframework.org/schema/context/spring-
context.xsd">
9
10     <bean id="userService"
class="com.itheima.service.impl.UserServiceImpl">
11         <property name="userDao" ref="userDao"></property>
12     </bean>
13
14     <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
15
16 </beans>

```

代码运行结果如下：

```

UserController x
D:\develop\java\jdk1.8\bin\java.exe ...
userServiceImpl ...
userDaoImpl ....

```

通过上面代码及结果可以看出：

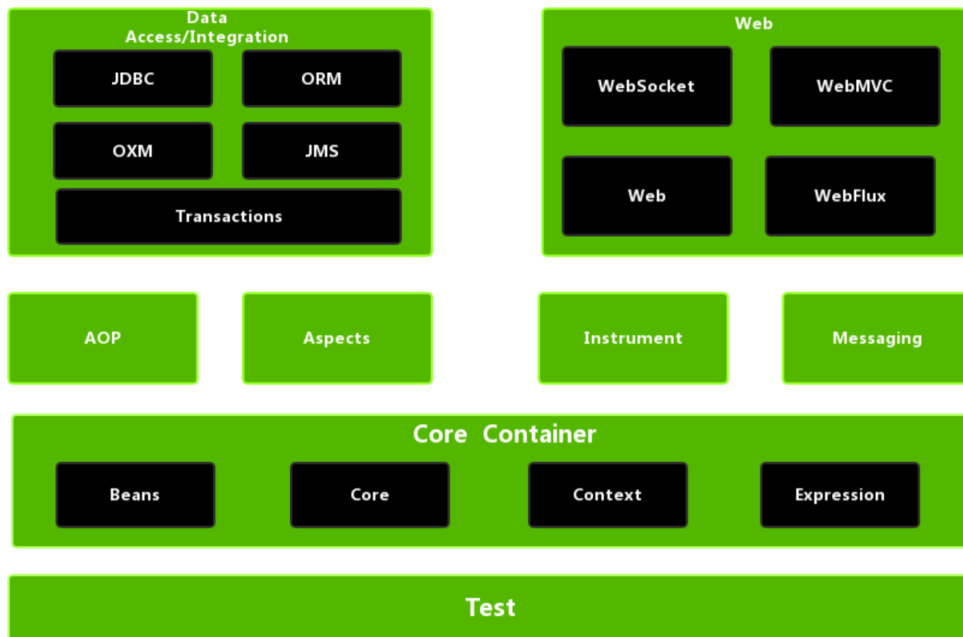
- userService对象是从applicationContext容器对象获取到的，也就是userService对象交由spring进行管理。
- 上面结果可以看到调用了UserDao对象中的add方法，也就是说UserDao子实现类对象也交由spring管理了。
- UserService中的userDao变量我们并没有进行赋值，但是可以正常使用，说明spring已经将UserDao对象赋值给了userDao变量。

上面三点体现了Spring框架的IOC (Inversion of Control) 和DI (Dependency Injection, DI)

## 7.2 spring核心功能结构

Spring大约有20个模块，由1300多个不同的文件构成。这些模块可以分为：

核心容器、AOP和设备支持、数据访问与集成、Web组件、通信报文和集成测试等，下面是 Spring 框架的总体架构图：



核心容器由 beans、core、context 和 expression (Spring Expression Language, SpEL) 4个模块组成。

- spring-beans和spring-core模块是Spring框架的核心模块，包含了控制反转 (Inversion of Control, IOC) 和依赖注入 (Dependency Injection, DI)。BeanFactory使用控制反转对应用程序的配置和依赖性规范与实际的应用程序代码进行了分离。BeanFactory属于延时加载，也就是说在实例化容器对象后并不会自动实例化Bean，只有当Bean被使用时，BeanFactory才会对该 Bean 进行实例化与依赖关系的装配。
- spring-context模块构架于核心模块之上，扩展了BeanFactory，为它添加了Bean生命周期控制、框架事件体系及资源加载透明化等功能。此外，该模块还提供了许多企业级支持，如邮件访问、远程访问、任务调度等，ApplicationContext 是该模块的核心接口，它的超类是 BeanFactory。与BeanFactory不同，ApplicationContext实例化后会自动对所有的单实例Bean进行实例化与依赖关系的装配，使之处于待用状态。
- spring-context-support模块是对Spring IoC容器及IoC子容器的扩展支持。
- spring-context-indexer模块是Spring的类管理组件和Classpath扫描组件。
- spring-expression 模块是统一表达式语言 (EL) 的扩展模块，可以查询、管理运行中的对象，同时也可以方便地调用对象方法，以及操作数组、集合等。它的语法类似于传统EL，但提供了额外的功能，最出色的要数函数调用和简单字符串的模板函数。EL的特性是基于Spring产品的需求而设计的，可以非常方便地同Spring IoC进行交互。

### 7.1.1 bean概述

Spring 就是面向 **Bean** 的编程 (BOP, Bean Oriented Programming), Bean 在 Spring 中处于核心地位。Bean对于Spring的意义就像Object对于OOP的意义一样, Spring中没有Bean也就没有Spring存在的意义。Spring IoC容器通过配置文件或者注解的方式来管理bean对象之间的依赖关系。

spring中bean用于对一个类进行封装。如下面的配置:

```
1 <bean id="userService" class="com.itheima.service.impl.UserServiceImpl">
2     <property name="userDao" ref="userDao"></property>
3 </bean>
4 <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
```

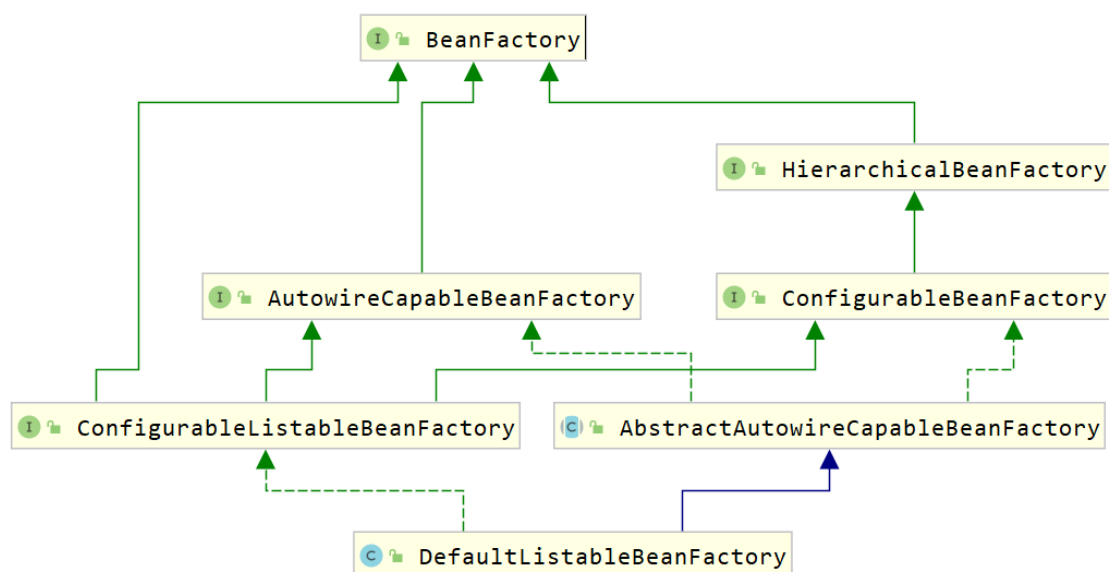
为什么Bean如此重要呢?

- spring 将bean对象交由一个叫IoC容器进行管理。
- bean对象之间的依赖关系在配置文件中体现, 并由spring完成。

## 7.3 Spring IOC相关接口分析

### 7.3.1 BeanFactory解析

Spring中Bean的创建是典型的工厂模式, 这一系列的Bean工厂, 即IoC容器, 为开发者管理对象之间的依赖关系提供了很多便利和基础服务, 在Spring中有许多IoC容器的实现供用户选择, 其相互关系如下图所示。



其中, BeanFactory作为最顶层的一个接口, 定义了IoC容器的基本功能规范, BeanFactory有三个重要的子接口: ListableBeanFactory、HierarchicalBeanFactory和 AutowireCapableBeanFactory。但是从类图中我们可以发现最终的默认实现类是 DefaultListableBeanFactory, 它实现了所有的接口。

那么为何要定义这么多层次的接口呢?

每个接口都有它的使用场合, 主要是为了区分在Spring内部操作过程中对象的传递和转化, 对对象的数据访问所做的限制。例如,

- ListableBeanFactory接口表示这些Bean可列表化。
- HierarchicalBeanFactory表示这些Bean 是有继承关系的, 也就是每个 Bean 可能有父 Bean

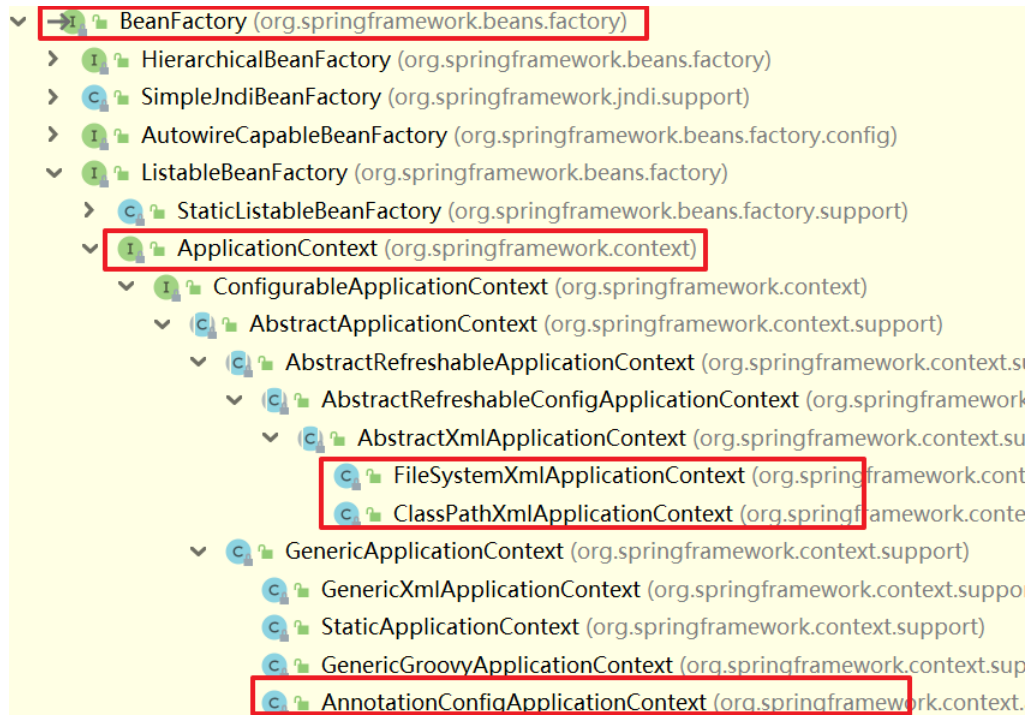
- AutowireCapableBeanFactory 接口定义Bean的自动装配规则。

这三个接口共同定义了Bean的集合、Bean之间的关系及Bean行为。最基本的IoC容器接口是BeanFactory，来看一下它的源码：

```
1 public interface BeanFactory {
2
3     String FACTORY_BEAN_PREFIX = "&";
4
5     //根据bean的名称获取IOC容器中的的bean对象
6     Object getBean(String name) throws BeansException;
7     //根据bean的名称获取IOC容器中的的bean对象，并指定获取到的bean对象的类型，这样我们使
8     //用时就不需要进行类型强转了
9     <T> T getBean(String name, Class<T> requiredType) throws
10    BeansException;
11    Object getBean(String name, Object... args) throws BeansException;
12    <T> T getBean(Class<T> requiredType) throws BeansException;
13    <T> T getBean(Class<T> requiredType, Object... args) throws
14    BeansException;
15
16    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);
17    <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);
18
19    //判断容器中是否包含指定名称的bean对象
20    boolean containsBean(String name);
21    //根据bean的名称判断是否是单例
22    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
23    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
24    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws
25    NoSuchBeanDefinitionException;
26    boolean isTypeMatch(String name, Class<?> typeToMatch) throws
27    NoSuchBeanDefinitionException;
28    @Nullable
29    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
30    String[] getAliases(String name);
31 }
```

在BeanFactory里只对IoC容器的基本行为做了定义，根本不关心你的Bean是如何定义及怎样加载的。正如我们只关心能从工厂里得到什么产品，不关心工厂是怎么生产这些产品的。

BeanFactory有一个很重要的子接口，就是ApplicationContext接口，该接口主要来规范容器中的bean对象是非延时加载，即在创建容器对象的时候就对对象bean进行初始化，并存储到一个容器中。



要知道工厂是如何产生对象的，我们需要看具体的IoC容器实现，Spring提供了许多IoC容器实现，比如：

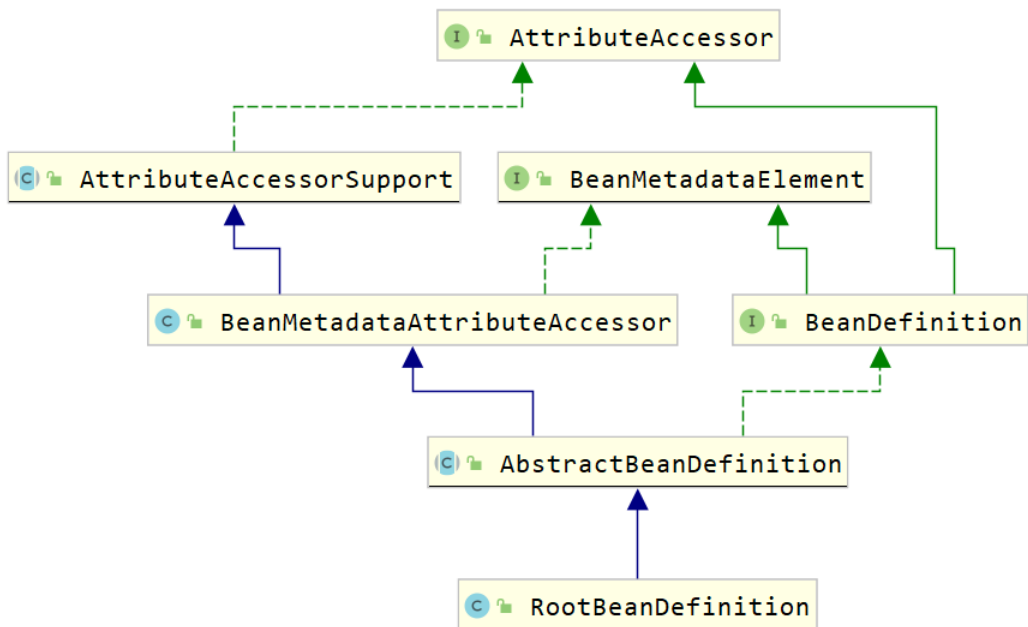
- ClasspathXmlApplicationContext：根据类路径加载xml配置文件，并创建IoC容器对象。
- FileSystemXmlApplicationContext：根据系统路径加载xml配置文件，并创建IoC容器对象。
- AnnotationConfigApplicationContext：加载注解类配置，并创建IoC容器。

### 7.3.2 BeanDefinition解析

Spring IoC容器管理我们定义的各种Bean对象及其相互关系，而Bean对象在Spring实现中是以BeanDefinition来描述的，如下面配置文件

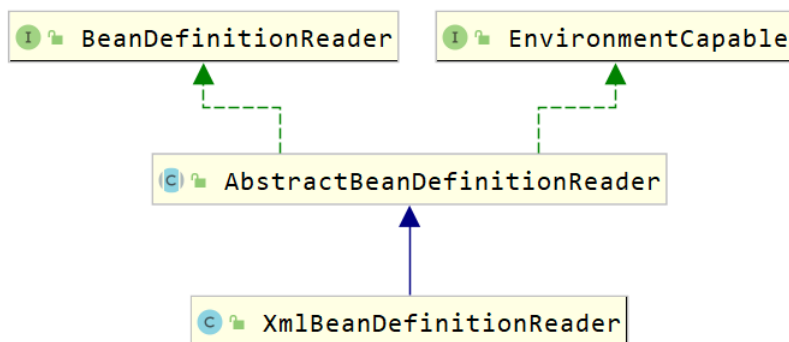
```
1 <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
2
3 bean标签还有很多属性：
4     scope、init-method、destory-method等。
```

其继承体系如下图所示。



### 7.3.3 BeanDefinitionReader解析

Bean的解析过程非常复杂，功能被分得很细，因为这里需要被扩展的地方很多，必须保证足够的灵活性，以应对可能的变化。Bean的解析主要就是对Spring配置文件的解析。这个解析过程主要通过BeanDefinitionReader来完成，看看Spring中BeanDefinitionReader的类结构图，如下图所示。



看看BeanDefinitionReader接口定义的功能来理解它具体的作用：

```

1 public interface BeanDefinitionReader {
2
3     //获取BeanDefinitionRegistry注册器对象
4     BeanDefinitionRegistry getRegistry();
5
6     @Nullable
7     ResourceLoader getResourceLoader();
8
9     @Nullable
10    ClassLoader getBeanClassLoader();
11
12    BeanNameGenerator getBeanNameGenerator();
13
14    /*
15     下面的loadBeanDefinitions都是加载bean定义，从指定的资源中
16    */
  
```

```

17     int loadBeanDefinitions(Resource resource) throws
BeanDefinitionStoreException;
18     int loadBeanDefinitions(Resource... resources) throws
BeanDefinitionStoreException;
19     int loadBeanDefinitions(String location) throws
BeanDefinitionStoreException;
20     int loadBeanDefinitions(String... locations) throws
BeanDefinitionStoreException;
21 }

```

### 7.3.4 BeanDefinitionRegistry解析

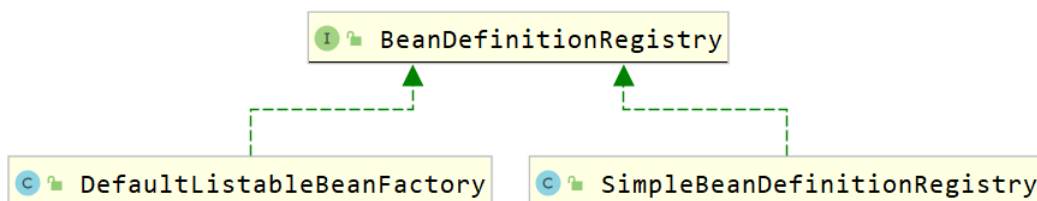
BeanDefinitionReader用来解析bean定义，并封装BeanDefinition对象，而我们定义的配置文件中定义了很多bean标签，所以就有一个问题，解析的BeanDefinition对象存储到哪儿？答案就是BeanDefinition的注册中心，而该注册中心顶层接口就是BeanDefinitionRegistry。

```

1 public interface BeanDefinitionRegistry extends AliasRegistry {
2
3     //往注册表中注册bean
4     void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
5         throws BeanDefinitionStoreException;
6
7     //从注册表中删除指定名称的bean
8     void removeBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException;
9
10    //获取注册表中指定名称的bean
11    BeanDefinition getBeanDefinition(String beanName) throws
NoSuchBeanDefinitionException;
12
13    //判断注册表中是否已经注册了指定名称的bean
14    boolean containsBeanDefinition(String beanName);
15
16    //获取注册表中所有的bean的名称
17    String[] getBeanDefinitionNames();
18
19    int getBeanDefinitionCount();
20    boolean isBeanNameInUse(String beanName);
21 }

```

继承结构图如下：



从上面类图可以看到BeanDefinitionRegistry接口的子实现类主要有以下几个：

- DefaultListableBeanFactory

在该类中定义了如下代码，就是用来注册bean

```
1 private final Map<String, BeanDefinition> beanDefinitionMap = new
  ConcurrentHashMap<>(256);
```

- SimpleBeanDefinitionRegistry

在该类中定义了如下代码，就是用来注册bean

```
1 private final Map<String, BeanDefinition> beanDefinitionMap = new
  ConcurrentHashMap<>(64);
```

### 7.3.5 创建容器

ClassPathXmlApplicationContext对Bean配置资源的载入是从refresh () 方法开始的。refresh () 方法是一个模板方法，规定了 IoC 容器的启动流程，有些逻辑要交给其子类实现。它对 Bean 配置资源进行载入，ClassPathXmlApplicationContext通过调用其父类 AbstractApplicationContext的refresh () 方法启动整个IoC容器对Bean定义的载入过程。

## 7.4 自定义SpringIOC

现要对下面的配置文件进行解析，并自定义Spring框架的IOC对涉及到的对象进行管理。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans>
3     <bean id="userService" class="com.itheima.service.impl.UserServiceImpl">
4         <property name="userDao" ref="userDao"></property>
5     </bean>
6     <bean id="userDao" class="com.itheima.dao.impl.UserDaoImpl"></bean>
7 </beans>
```

### 7.4.1 定义bean相关的pojo类

#### 7.4.1.1 PropertyValue类

用于封装bean的属性，体现到上面的配置文件就是封装bean标签的子标签property标签数据。

```
1 public class PropertyValue {
2
3     private String name;
4     private String ref;
5     private String value;
6
7     public PropertyValue() {
8     }
9
10    public PropertyValue(String name, String ref,String value) {
11        this.name = name;
12        this.ref = ref;
13        this.value = value;
14    }
```



```

15
16     public String getName() {
17         return name;
18     }
19
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public String getRef() {
25         return ref;
26     }
27
28     public void setRef(String ref) {
29         this.ref = ref;
30     }
31
32     public String getValue() {
33         return value;
34     }
35
36     public void setValue(String value) {
37         this.value = value;
38     }
39 }

```

#### 7.4.1.2 MutablePropertyValues类

一个bean标签可以有多个property子标签，所以再定义一个MutablePropertyValues类，用来存储并管理多个PropertyValue对象。

```

1     public class MutablePropertyValues implements Iterable<PropertyValue> {
2
3         private final List<PropertyValue> propertyValueList;
4
5         public MutablePropertyValues() {
6             this.propertyValueList = new ArrayList<PropertyValue>();
7         }
8
9         public MutablePropertyValues(List<PropertyValue> propertyValueList) {
10            this.propertyValueList = (propertyValueList != null ?
propertyValueList : new ArrayList<PropertyValue>());
11        }
12
13        public PropertyValue[] getPropertyValues() {
14            return this.propertyValueList.toArray(new PropertyValue[0]);
15        }
16
17        public PropertyValue getPropertyValue(String propertyName) {
18            for (PropertyValue pv : this.propertyValueList) {
19                if (pv.getName().equals(propertyName)) {
20                    return pv;
21                }
22            }
23            return null;

```

```

24     }
25
26     @Override
27     public Iterator<PropertyValue> iterator() {
28         return propertyValueList.iterator();
29     }
30
31     public boolean isEmpty() {
32         return this.propertyValueList.isEmpty();
33     }
34
35     public MutablePropertyValues addPropertyValue(PropertyValue pv) {
36         for (int i = 0; i < this.propertyValueList.size(); i++) {
37             PropertyValue currentPv = this.propertyValueList.get(i);
38             if (currentPv.getName().equals(pv.getName())) {
39                 this.propertyValueList.set(i, new
PropertyValue(pv.getName(),pv.getRef(), pv.getValue()));
40                 return this;
41             }
42         }
43         this.propertyValueList.add(pv);
44         return this;
45     }
46
47     public boolean contains(String propertyName) {
48         return getPropertyValue(propertyName) != null;
49     }
50 }

```

### 7.4.1.3 BeanDefinition类

BeanDefinition类用来封装bean信息的，主要包含id（即bean对象的名称）、class（需要交由spring管理的类的全类名）及子标签property数据。

```

1 public class BeanDefinition {
2     private String id;
3     private String className;
4
5     private MutablePropertyValues propertyValues;
6
7     public BeanDefinition() {
8         propertyValues = new MutablePropertyValues();
9     }
10
11     public String getId() {
12         return id;
13     }
14
15     public void setId(String id) {
16         this.id = id;
17     }
18
19     public String getClassName() {
20         return className;
21     }

```

```

22
23     public void setClassName(String className) {
24         this.className = className;
25     }
26
27     public void setPropertyValues(MutablePropertyValues propertyValues) {
28         this.propertyValues = propertyValues;
29     }
30
31     public MutablePropertyValues getPropertyValues() {
32         return propertyValues;
33     }
34 }

```

## 7.4.2 定义注册表相关类

### 7.4.2.1 BeanDefinitionRegistry接口

BeanDefinitionRegistry接口定义了注册表的相关操作，定义如下功能：

- 注册BeanDefinition对象到注册表中
- 从注册表中删除指定名称的BeanDefinition对象
- 根据名称从注册表中获取BeanDefinition对象
- 判断注册表中是否包含指定名称的BeanDefinition对象
- 获取注册表中BeanDefinition对象的个数
- 获取注册表中所有的BeanDefinition的名称

```

1  public interface BeanDefinitionRegistry {
2
3      //注册BeanDefinition对象到注册表中
4      void registerBeanDefinition(String beanName, BeanDefinition
      beanDefinition);
5
6      //从注册表中删除指定名称的BeanDefinition对象
7      void removeBeanDefinition(String beanName) throws Exception;
8
9      //根据名称从注册表中获取BeanDefinition对象
10     BeanDefinition getBeanDefinition(String beanName) throws Exception;
11
12     boolean containsBeanDefinition(String beanName);
13
14     int getBeanDefinitionCount();
15
16     String[] getBeanDefinitionNames();
17 }

```

### 7.4.2.2 SimpleBeanDefinitionRegistry类

该类实现了BeanDefinitionRegistry接口，定义了Map集合作为注册表容器。

```

1  public class SimpleBeanDefinitionRegistry implements BeanDefinitionRegistry
    {

```

```

2
3     private Map<String, BeanDefinition> beanDefinitionMap = new
HashMap<String, BeanDefinition>();
4
5     @Override
6     public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition) {
7         beanDefinitionMap.put(beanName, beanDefinition);
8     }
9
10    @Override
11    public void removeBeanDefinition(String beanName) throws Exception {
12        beanDefinitionMap.remove(beanName);
13    }
14
15    @Override
16    public BeanDefinition getBeanDefinition(String beanName) throws
Exception {
17        return beanDefinitionMap.get(beanName);
18    }
19
20    @Override
21    public boolean containsBeanDefinition(String beanName) {
22        return beanDefinitionMap.containsKey(beanName);
23    }
24
25    @Override
26    public int getBeanDefinitionCount() {
27        return beanDefinitionMap.size();
28    }
29
30    @Override
31    public String[] getBeanDefinitionNames() {
32        return beanDefinitionMap.keySet().toArray(new String[1]);
33    }
34 }

```

## 7.4.3 定义解析器相关类

### 7.4.3.1 BeanDefinitionReader接口

BeanDefinitionReader是用来解析配置文件并在注册表中注册bean的信息。定义了两个规范：

- 获取注册表的功能，让外界可以通过该对象获取注册表对象。
- 加载配置文件，并注册bean数据。

```

1 public interface BeanDefinitionReader {
2
3     //获取注册表对象
4     BeanDefinitionRegistry getRegistry();
5     //加载配置文件并在注册表中进行注册
6     void loadBeanDefinitions(String configLocation) throws Exception;
7 }

```

### 7.4.3.2 XmlBeanDefinitionReader类

XmlBeanDefinitionReader类是专门用来解析xml配置文件的。该类实现BeanDefinitionReader接口并实现接口中的两个功能。

```
1 public class XmlBeanDefinitionReader implements BeanDefinitionReader {
2
3     private BeanDefinitionRegistry registry;
4
5     public XmlBeanDefinitionReader() {
6         this.registry = new SimpleBeanDefinitionRegistry();
7     }
8
9     @Override
10    public BeanDefinitionRegistry getRegistry() {
11        return registry;
12    }
13
14    @Override
15    public void loadBeanDefinitions(String configLocation) throws Exception
16    {
17        InputStream is =
18        this.getClass().getClassLoader().getResourceAsStream(configLocation);
19        SAXReader reader = new SAXReader();
20        Document document = reader.read(is);
21        Element rootElement = document.getRootElement();
22        //解析bean标签
23        parseBean(rootElement);
24
25        private void parseBean(Element rootElement) {
26
27            List<Element> elements = rootElement.elements();
28            for (Element element : elements) {
29                String id = element.attributeValue("id");
30                String className = element.attributeValue("class");
31                BeanDefinition beanDefinition = new BeanDefinition();
32                beanDefinition.setId(id);
33                beanDefinition.setClassName(className);
34                List<Element> list = element.elements("property");
35                MutablePropertyValues mutablePropertyValues = new
36                MutablePropertyValues();
37                for (Element element1 : list) {
38                    String name = element1.attributeValue("name");
39                    String ref = element1.attributeValue("ref");
40                    String value = element1.attributeValue("value");
41                    PropertyValue propertyValue = new
42                    PropertyValue(name, ref, value);
43                    mutablePropertyValues.addPropertyValue(propertyValue);
44                }
45                beanDefinition.setPropertyValues(mutablePropertyValues);
46
47                registry.registerBeanDefinition(id, beanDefinition);
48            }
49        }
50    }
51 }
```

## 7.4.4 IOC容器相关类

### 7.4.4.1 BeanFactory接口

在该接口中定义IOC容器的统一规范即获取bean对象。

```
1 public interface BeanFactory {
2     //根据bean对象的名称获取bean对象
3     Object getBean(String name) throws Exception;
4     //根据bean对象的名称获取bean对象，并进行类型转换
5     <T> T getBean(String name, Class<? extends T> clazz) throws Exception;
6 }
```

### 7.4.4.2 ApplicationContext接口

该接口的所以的子实现类对bean对象的创建都是非延时的，所以在该接口中定义 refresh() 方法，该方法主要完成以下两个功能：

- 加载配置文件。
- 根据注册表中的BeanDefinition对象封装的数据进行bean对象的创建。

```
1 public interface ApplicationContext extends BeanFactory {
2     //进行配置文件加载并进行对象创建
3     void refresh() throws IllegalStateException, Exception;
4 }
```

### 7.4.4.3 AbstractApplicationContext类

- 作为ApplicationContext接口的子类，所以该类也是非延时加载，所以需要在该类中定义一个Map集合，作为bean对象存储的容器。
- 声明BeanDefinitionReader类型的变量，用来进行xml配置文件的解析，符合单一职责原则。

BeanDefinitionReader类型的对象创建交由子类实现，因为只有子类明确到底创建BeanDefinitionReader哪个子实现类对象。

```
1 public abstract class AbstractApplicationContext implements
2     ApplicationContext {
3     protected BeanDefinitionReader beanDefinitionReader;
4     //用来存储bean对象的容器 key存储的是bean的id值，value存储的是bean对象
5     protected Map<String, Object> singletonObjects = new HashMap<String,
6     Object>();
7     //存储配置文件的路径
8     protected String configLocation;
9
10    public void refresh() throws IllegalStateException, Exception {
11
12        //加载BeanDefinition
13        beanDefinitionReader.loadBeanDefinitions(configLocation);
```

```

14
15     //初始化bean
16     finishBeanInitialization();
17 }
18
19 //bean的初始化
20 private void finishBeanInitialization() throws Exception {
21     BeanDefinitionRegistry registry =
beanDefinitionReader.getRegistry();
22     String[] beanNames = registry.getBeanDefinitionNames();
23
24     for (String beanName : beanNames) {
25         BeanDefinition beanDefinition =
registry.getBeanDefinition(beanName);
26         getBean(beanName);
27     }
28 }
29 }

```

注意：该类`finishBeanInitialization()`方法中调用`getBean()`方法使用到了模板方法模式。

#### 7.4.4.4 ClassPathXmlApplicationContext类

该类主要是加载类路径下的配置文件，并进行bean对象的创建，主要完成以下功能：

- 在构造方法中，创建`BeanDefinitionReader`对象。
- 在构造方法中，调用`refresh()`方法，用于进行配置文件加载、创建bean对象并存储到容器中。
- 重写父接口中的`getBean()`方法，并实现依赖注入操作。

```

1 public class ClassPathXmlApplicationContext extends
AbstractApplicationContext{
2
3     public ClassPathXmlApplicationContext(String configLocation) {
4         this.configLocation = configLocation;
5         //构建XmlBeanDefinitionReader对象
6         beanDefinitionReader = new XmlBeanDefinitionReader();
7         try {
8             this.refresh();
9         } catch (Exception e) {
10            }
11    }
12
13    //根据bean的id属性值获取bean对象
14    @Override
15    public Object getBean(String name) throws Exception {
16
17        //return singletonObjects.get(name);
18        Object obj = singletonObjects.get(name);
19        if(obj != null) {
20            return obj;
21        }
22
23        BeanDefinitionRegistry registry =
beanDefinitionReader.getRegistry();

```

```

24     BeanDefinition beanDefinition = registry.getBeanDefinition(name);
25     if(beanDefinition == null) {
26         return null;
27     }
28     String className = beanDefinition.getClassName();
29     Class<?> clazz = Class.forName(className);
30     Object beanObj = clazz.newInstance();
31     MutablePropertyValues propertyValues =
beanDefinition.getPropertyValues();
32     for (PropertyValue propertyValue : propertyValues) {
33         String propertyName = propertyValue.getName();
34         String value = propertyValue.getValue();
35         String ref = propertyValue.getRef();
36         if(ref != null && !"".equals(ref)) {
37
38             Object bean = getBean(ref);
39             String methodName =
StringUtils.getSetterMethodNameByFieldName(propertyName);
40             Method[] methods = clazz.getMethods();
41             for (Method method : methods) {
42                 if(method.getName().equals(methodName)) {
43                     method.invoke(beanObj, bean);
44                 }
45             }
46         }
47
48         if(value != null && !"".equals(value)) {
49             String methodName =
StringUtils.getSetterMethodNameByFieldName(propertyName);
50             Method method = clazz.getMethod(methodName, String.class);
51             method.invoke(beanObj, value);
52         }
53     }
54     singletonObjects.put(name, beanObj);
55     return beanObj;
56 }
57
58 @Override
59 public <T> T getBean(String name, Class<? extends T> clazz) throws
Exception {
60
61     Object bean = getBean(name);
62     if(bean != null) {
63         return clazz.cast(bean);
64     }
65     return null;
66 }
67 }

```

## 7.4.5 自定义Spring IOC总结

### 7.4.5.1 使用到的设计模式

- 工厂模式。这个使用工厂模式 + 配置文件的方式。



- 单例模式。Spring IOC管理的bean对象都是单例的，此处的单例不是通过构造器进行单例的控制的，而是spring框架对每一个bean只创建了一个对象。
- 模板方法模式。AbstractApplicationContext类中的finishBeanInitialization()方法调用了子类的getBean()方法，因为getBean()的实现和环境息息相关。
- 迭代器模式。对于MutablePropertyValues类定义使用到了迭代器模式，因为此类存储并管理PropertyValue对象，也属于一个容器，所以给该容器提供一个遍历方式。

spring框架其实使用到了很多设计模式，如AOP使用到了代理模式，选择JDK代理或者CGLIB代理使用到了策略模式，还有适配器模式，装饰者模式，观察者模式等。

#### 7.4.5.2 符合大部分设计原则

#### 7.4.5.3 整个设计和Spring的设计还是有一定的出入

spring框架底层是很复杂的，进行了很深入的封装，并对外提供了很好的扩展性。而我们自定义SpringIOC有以下几个目的：

- 了解Spring底层对对象的大体管理机制。
- 了解设计模式在具体的开发中的使用。
- 以后学习spring源码，通过该案例的实现，可以降低spring学习的入门成本。