【**证券交易系统**】

# ——股票信息检索子系统

设计模式研究报告

组长：王俊 [3170100186@zju.edu.cn](mailto:3170103456@zju.edu.cn)

组员：李承泽 欧阳皓东 王荣浩 张律勤日期：2021/06/01

版本：Version 1.0

目录

[1 设计模式简介](#_Toc437408679_WPSOffice_Level1) [6](#_Toc437408679_WPSOffice_Level1)

[2 系统体系分析](#_Toc1805890597_WPSOffice_Level1) [7](#_Toc1805890597_WPSOffice_Level1)

[3 GoF 设计模式应用–创建型模式](#_Toc1216880728_WPSOffice_Level1) [8](#_Toc1216880728_WPSOffice_Level1)

[3.1 工厂模式](#_Toc1805890597_WPSOffice_Level2) [8](#_Toc1805890597_WPSOffice_Level2)

[3.1.1 介绍](#_Toc1805890597_WPSOffice_Level3) [8](#_Toc1805890597_WPSOffice_Level3)

[3.1.3 优缺点分析](#_Toc1216880728_WPSOffice_Level3) [9](#_Toc1216880728_WPSOffice_Level3)

[3.2 抽象工厂模式](#_Toc1216880728_WPSOffice_Level2) [9](#_Toc1216880728_WPSOffice_Level2)

[3.2.1 介绍](#_Toc1627625115_WPSOffice_Level3) [9](#_Toc1627625115_WPSOffice_Level3)

[3.2.2 举例](#_Toc848612319_WPSOffice_Level3) [10](#_Toc848612319_WPSOffice_Level3)

[3.2.3 优缺点分析](#_Toc1188345706_WPSOffice_Level3) [10](#_Toc1188345706_WPSOffice_Level3)

[3.3 单例模式](#_Toc1627625115_WPSOffice_Level2) [10](#_Toc1627625115_WPSOffice_Level2)

[3.3.1 介绍](#_Toc928363642_WPSOffice_Level3) [11](#_Toc928363642_WPSOffice_Level3)

[3.3.2 举例](#_Toc1539035639_WPSOffice_Level3) [11](#_Toc1539035639_WPSOffice_Level3)

[3.3.3 优缺点分析](#_Toc131456558_WPSOffice_Level3) [11](#_Toc131456558_WPSOffice_Level3)

[3.4 建造者模式](#_Toc848612319_WPSOffice_Level2) [11](#_Toc848612319_WPSOffice_Level2)

[3.4.1 介绍](#_Toc1777181190_WPSOffice_Level3) [11](#_Toc1777181190_WPSOffice_Level3)

[3.4.2 举例](#_Toc1881697854_WPSOffice_Level3) [12](#_Toc1881697854_WPSOffice_Level3)

[3.4.3 优缺点分析](#_Toc1851646456_WPSOffice_Level3) [12](#_Toc1851646456_WPSOffice_Level3)

[3.5 原型模式](#_Toc1188345706_WPSOffice_Level2) [13](#_Toc1188345706_WPSOffice_Level2)

[3.5.1 介绍](#_Toc1436457315_WPSOffice_Level3) [13](#_Toc1436457315_WPSOffice_Level3)

[3.5.2 举例](#_Toc526933631_WPSOffice_Level3) [13](#_Toc526933631_WPSOffice_Level3)

[3.5.3 优缺点分析](#_Toc2098459636_WPSOffice_Level3) [14](#_Toc2098459636_WPSOffice_Level3)

[4 GoF 设计模式应用-结构性模型](#_Toc1627625115_WPSOffice_Level1) [15](#_Toc1627625115_WPSOffice_Level1)

[4.1 适配器（Adaptor）](#_Toc928363642_WPSOffice_Level2) [15](#_Toc928363642_WPSOffice_Level2)

[4.1.1 介绍](#_Toc801123385_WPSOffice_Level3) [15](#_Toc801123385_WPSOffice_Level3)

[4.1.2 举例](#_Toc1905748652_WPSOffice_Level3) [16](#_Toc1905748652_WPSOffice_Level3)

[4.1.3 优缺点分析](#_Toc198999159_WPSOffice_Level3) [16](#_Toc198999159_WPSOffice_Level3)

[4.2 桥接（Bridge）](#_Toc1539035639_WPSOffice_Level2) [16](#_Toc1539035639_WPSOffice_Level2)

[4.2.1 介绍](#_Toc946826934_WPSOffice_Level3) [16](#_Toc946826934_WPSOffice_Level3)

[4.2.2 优缺点分析](#_Toc466455468_WPSOffice_Level3) [17](#_Toc466455468_WPSOffice_Level3)

[4.3 组合（Composite）](#_Toc131456558_WPSOffice_Level2) [17](#_Toc131456558_WPSOffice_Level2)

[4.3.1. 介绍](#_Toc1401739126_WPSOffice_Level3) [17](#_Toc1401739126_WPSOffice_Level3)

[4.3.2 举例](#_Toc1133883092_WPSOffice_Level3) [18](#_Toc1133883092_WPSOffice_Level3)

[4.3.3 优缺点分析](#_Toc403243766_WPSOffice_Level3) [19](#_Toc403243766_WPSOffice_Level3)

[4.4 外观（Facade）](#_Toc1777181190_WPSOffice_Level2) [19](#_Toc1777181190_WPSOffice_Level2)

[4.4 外观（Facade）](#_Toc1881697854_WPSOffice_Level2) [19](#_Toc1881697854_WPSOffice_Level2)

[4.4.1 介绍](#_Toc2007068877_WPSOffice_Level3) [19](#_Toc2007068877_WPSOffice_Level3)

[4.4.2 举例：](#_Toc133488663_WPSOffice_Level3) [19](#_Toc133488663_WPSOffice_Level3)

[4.4.3 优缺点分析](#_Toc1571031573_WPSOffice_Level3) [20](#_Toc1571031573_WPSOffice_Level3)

[4.5 Flyweight（享元）](#_Toc1851646456_WPSOffice_Level2) [21](#_Toc1851646456_WPSOffice_Level2)

[4.5.1 介绍](#_Toc1016207546_WPSOffice_Level3) [21](#_Toc1016207546_WPSOffice_Level3)

[4.5.2 举例](#_Toc462781031_WPSOffice_Level3) [21](#_Toc462781031_WPSOffice_Level3)

[4.5.3 应用场景和扩展](#_Toc1922502230_WPSOffice_Level3) [22](#_Toc1922502230_WPSOffice_Level3)

[4.6 Proxy（代理）](#_Toc1436457315_WPSOffice_Level2) [23](#_Toc1436457315_WPSOffice_Level2)

[4.6.1 介绍](#_Toc456026848_WPSOffice_Level3) [23](#_Toc456026848_WPSOffice_Level3)

[4.6.2 举例：](#_Toc74098193_WPSOffice_Level3) [24](#_Toc74098193_WPSOffice_Level3)

[4.6.3 优缺点分析](#_Toc1975298138_WPSOffice_Level3) [25](#_Toc1975298138_WPSOffice_Level3)

[4.7 Decorator（装饰）](#_Toc526933631_WPSOffice_Level2) [25](#_Toc526933631_WPSOffice_Level2)

[4.7.1 介绍](#_Toc1249499907_WPSOffice_Level3) [25](#_Toc1249499907_WPSOffice_Level3)

[4.7.2 适用性：](#_Toc102352936_WPSOffice_Level3) [25](#_Toc102352936_WPSOffice_Level3)

[4.7.3 举例：](#_Toc111394105_WPSOffice_Level3) [26](#_Toc111394105_WPSOffice_Level3)

[4.7.4 优缺点：](#_Toc1742466198_WPSOffice_Level3) [26](#_Toc1742466198_WPSOffice_Level3)

[4.7.5 相关模式：](#_Toc394895647_WPSOffice_Level3) [27](#_Toc394895647_WPSOffice_Level3)

[5 GoF 设计模式应用-行为模式](#_Toc848612319_WPSOffice_Level1) [27](#_Toc848612319_WPSOffice_Level1)

[5.1 Interpreter（解释器）](#_Toc2098459636_WPSOffice_Level2) [27](#_Toc2098459636_WPSOffice_Level2)

[5.1.1 介绍](#_Toc1286669899_WPSOffice_Level3) [27](#_Toc1286669899_WPSOffice_Level3)

[5.1.2 相关模式](#_Toc2048150850_WPSOffice_Level3) [28](#_Toc2048150850_WPSOffice_Level3)

[5.1.3 效果](#_Toc1255958187_WPSOffice_Level3) [28](#_Toc1255958187_WPSOffice_Level3)

[5.2 模板方法（Template Method）](#_Toc687167571_WPSOffice_Level2) [29](#_Toc687167571_WPSOffice_Level2)

[5.2.1 介绍：](#_Toc1272482546_WPSOffice_Level3) [29](#_Toc1272482546_WPSOffice_Level3)

[5.2.2 模式的结构：](#_Toc1971993796_WPSOffice_Level3) [29](#_Toc1971993796_WPSOffice_Level3)

[5.2.2. 举例](#_Toc1184605221_WPSOffice_Level3) [29](#_Toc1184605221_WPSOffice_Level3)

[5.2.3. 优缺点分析](#_Toc339058010_WPSOffice_Level3) [30](#_Toc339058010_WPSOffice_Level3)

[5.3 责任链（Chain of Responsibility）](#_Toc58312231_WPSOffice_Level2) [30](#_Toc58312231_WPSOffice_Level2)

[5.3.1 介绍](#_Toc1273858579_WPSOffice_Level3) [30](#_Toc1273858579_WPSOffice_Level3)

[5.3.2 适用性](#_Toc1476660310_WPSOffice_Level3) [31](#_Toc1476660310_WPSOffice_Level3)

[5.3.3 参与角色和分析：首先，责任链模式中，存在着这么几个角色：](#_Toc1908805438_WPSOffice_Level3) [32](#_Toc1908805438_WPSOffice_Level3)

[5.4 Command（命令）](#_Toc801123385_WPSOffice_Level2) [32](#_Toc801123385_WPSOffice_Level2)

[5.4.1 介绍](#_Toc34793933_WPSOffice_Level3) [32](#_Toc34793933_WPSOffice_Level3)

[5.4.2 适用情况](#_Toc666079947_WPSOffice_Level3) [33](#_Toc666079947_WPSOffice_Level3)

[5.4.3 举例：](#_Toc2120901065_WPSOffice_Level3) [34](#_Toc2120901065_WPSOffice_Level3)

[5.4.4 优缺点分析](#_Toc2050626549_WPSOffice_Level3) [34](#_Toc2050626549_WPSOffice_Level3)

[5.5 Iterator（迭代器）](#_Toc1905748652_WPSOffice_Level2) [34](#_Toc1905748652_WPSOffice_Level2)

[5.5.1 介绍](#_Toc2062841987_WPSOffice_Level3) [34](#_Toc2062841987_WPSOffice_Level3)

[5.5.2 适用情况](#_Toc1209278341_WPSOffice_Level3) [35](#_Toc1209278341_WPSOffice_Level3)

[5.5.3 举例](#_Toc555841979_WPSOffice_Level3) [36](#_Toc555841979_WPSOffice_Level3)

[5.5.4 优缺点分析](#_Toc482276603_WPSOffice_Level3) [36](#_Toc482276603_WPSOffice_Level3)

[5.6 Mediator（中介者）](#_Toc198999159_WPSOffice_Level2) [36](#_Toc198999159_WPSOffice_Level2)

[5.6.1 介绍](#_Toc1019582843_WPSOffice_Level3) [36](#_Toc1019582843_WPSOffice_Level3)

[5.6.2 适用情况](#_Toc1356822888_WPSOffice_Level3) [37](#_Toc1356822888_WPSOffice_Level3)

[5.6.3 举例](#_Toc2140914770_WPSOffice_Level3) [38](#_Toc2140914770_WPSOffice_Level3)

[5.6.4 优缺点分析](#_Toc1266033905_WPSOffice_Level3) [38](#_Toc1266033905_WPSOffice_Level3)

[5.7 Memento（备忘录）](#_Toc946826934_WPSOffice_Level2) [38](#_Toc946826934_WPSOffice_Level2)

[5.7.1 介绍](#_Toc963866859_WPSOffice_Level3) [38](#_Toc963866859_WPSOffice_Level3)

[5.7.2 适用性分析](#_Toc1241149892_WPSOffice_Level3) [39](#_Toc1241149892_WPSOffice_Level3)

[5.7.3 举例](#_Toc1497571533_WPSOffice_Level3) [40](#_Toc1497571533_WPSOffice_Level3)

[5.7.4 优缺点分析](#_Toc1176412291_WPSOffice_Level3) [40](#_Toc1176412291_WPSOffice_Level3)

[5.8 Observer（观察者）](#_Toc466455468_WPSOffice_Level2) [40](#_Toc466455468_WPSOffice_Level2)

[5.8.1 介绍](#_Toc79436908_WPSOffice_Level3) [40](#_Toc79436908_WPSOffice_Level3)

[5.8.2 适用情况](#_Toc1508767969_WPSOffice_Level3) [41](#_Toc1508767969_WPSOffice_Level3)

[5.8.3 举例](#_Toc376351207_WPSOffice_Level3) [42](#_Toc376351207_WPSOffice_Level3)

[5.8.4 优缺点分析](#_Toc995395634_WPSOffice_Level3) [42](#_Toc995395634_WPSOffice_Level3)

[5.9 State（状态）](#_Toc1401739126_WPSOffice_Level2) [42](#_Toc1401739126_WPSOffice_Level2)

[5.9.1 介绍](#_Toc716810508_WPSOffice_Level3) [42](#_Toc716810508_WPSOffice_Level3)

[5.9.2 起源](#_Toc50948286_WPSOffice_Level3) [42](#_Toc50948286_WPSOffice_Level3)

[5.9.3 适用范围](#_Toc1589351296_WPSOffice_Level3) [43](#_Toc1589351296_WPSOffice_Level3)

[5.9.4 设计模式结构](#_Toc1825630486_WPSOffice_Level3) [44](#_Toc1825630486_WPSOffice_Level3)

[5.9.5 效果：](#_Toc125229866_WPSOffice_Level3) [44](#_Toc125229866_WPSOffice_Level3)

[5.9.6 实现中产生的问题](#_Toc204383802_WPSOffice_Level3) [45](#_Toc204383802_WPSOffice_Level3)

[5.10 Strategy（策略）](#_Toc1133883092_WPSOffice_Level2) [47](#_Toc1133883092_WPSOffice_Level2)

[5.10.1 简介](#_Toc1252208661_WPSOffice_Level3) [47](#_Toc1252208661_WPSOffice_Level3)

[5.10.2 起源](#_Toc531224827_WPSOffice_Level3) [47](#_Toc531224827_WPSOffice_Level3)

[5.10.3 适用范围](#_Toc1206146810_WPSOffice_Level3) [48](#_Toc1206146810_WPSOffice_Level3)

[5.10.4 设计模式结构](#_Toc1611291637_WPSOffice_Level3) [48](#_Toc1611291637_WPSOffice_Level3)

[5.10.5 效果](#_Toc1209754389_WPSOffice_Level3) [49](#_Toc1209754389_WPSOffice_Level3)

[5.10.6 实现中产生的问题](#_Toc2114329774_WPSOffice_Level3) [50](#_Toc2114329774_WPSOffice_Level3)

[5.11 Visitor（访问者）](#_Toc403243766_WPSOffice_Level2) [51](#_Toc403243766_WPSOffice_Level2)

[5.11.1 简介](#_Toc1128604709_WPSOffice_Level3) [51](#_Toc1128604709_WPSOffice_Level3)

[5.11.2 起源](#_Toc1883773859_WPSOffice_Level3) [51](#_Toc1883773859_WPSOffice_Level3)

[5.11.3 适用范围](#_Toc235840492_WPSOffice_Level3) [52](#_Toc235840492_WPSOffice_Level3)

[5.11.4 设计模式结构](#_Toc1663820329_WPSOffice_Level3) [53](#_Toc1663820329_WPSOffice_Level3)

[参考文献](#_Toc1188345706_WPSOffice_Level1) [53](#_Toc1188345706_WPSOffice_Level1)

# 设计模式简介

为什么要学设计模式？

很赞同《大话设计模式》一书作者所言：重要的不是你将来会不会用到这些模式，而是通过这些模式让你找到“封装变化”、“对象间松散耦合”、“针对接口编程”的感觉，从而设计出易维护、易扩展、易复用、灵活性好的程序。

那么什么是设计模式呢？

我们认为，设计模式是一套代码设计「经验的总结」。项目中「合理的」运用设计模式可以「巧妙的解决很多问题」。

* 经验的总结：则是人们在不停开发编写研究代码中的心路历程，最终总结出来得出来的「套路」。
* 合理的：要对设计模式的使用场景有一定的认识后才使用，「不要滥用」。如：输出一句“hello world”，非要强行给加上各种模式。
* 巧妙的解决了很多问题：被广泛应用的原因。

为什么要提倡“Design Pattern呢？根本原因是为了代码复用，增加可维护性。

大家也知道目前的设计模式很多，所以有的研究者关注设计模式之间的关系和联系，组织大约二十个设计模式及其相互关系，将其分成不同的类别并修改设计模式。 方便于后来的人安排不同层的设计模式。 结果简化了解目录的整体结构，从而更容易对其他设计进行分类模式，并将这些设计模式应用于软件发展。

有的研究者，研究面向对象的设计模式：设计模式作为表达面向对象设计体验的新机制。 设计模式识别、命名和抽象面向对象设计中的常见主题。 他们通过识别对象、它们的协作和职责分配来捕捉设计背后的意图。

有的研究者关注设计模式的有效性。针对设计模式编码大量信息的特征，有研究者提出使用特征图算法对源代码的设计实例进行检测，可以有效揭示隐藏在源代码中的有用信息。有研究者基于图论提出了一种两阶段检测设计模式的方法，第一阶段将模式和语义转换为语义图，第二阶段应用语义匹配算法获得候选实例，最终得到模式的行为特征。

有的研究者关注设计模式的自动识别。有研究者在 UML 和文本分类方法的基础上改进了自动选择设计模式的技术，通过提出新的框架和基于无监督学习的评估模型改进自动化技术 。有研究者开发了完全基于机器学习技术的识别方法，针对现有的识别工具构建数据集，通过自动检测选择设计模式，提高程序的可维护性和可靠性。有研究者关注模式实例的查找。针对从源代码中发现设计模式实例的现有方法的低效，有研究者提出了一种基于有序序列快速检索候选模式实例的方法 [5]，大大降低搜索空间，提升响应速度。有研究者将基于可视语言解析和模型检查的静态分析与基于源代码工具的动态分析相结合，提出了一种用于在源代码中检测模式实例的工具，可以获得更好的正确性与完整性。有研究者关注设计模式的教学。针对学生，为软件工程设计模式构建了一个交互式学习环境， 用于帮助学生改进解决方案并加深对设计模式的理解。

# 系统体系分析

本项目围绕着证券交易系统展开，是一个股票监管与管理平台，用户可以在该平台上获得感兴趣个股的相关信息以及模拟其曲线为了能更方便用户能直观地获取想要的形式。

本项目也是围绕着数据的增删改查的数据中心模型，项目的主要操作是有关数据的操作。项目中某个股中的新闻列表，市场模块里的信息、股票池里面的信息，以及策略的使用。由于我们使用了前后端分离的架构进行项目编写，这在不知不觉中其实已经用到许多设计模式。

而我们的产品目标是能够便利所有使用的客户，爬取各大主流信息平台，清洗并且可视化数据，让使用者能够更直观的观察股市概况。同时透过辅助判断模块，根据用户的要求（不同概念股、风险等）、预算，推荐合理的投资组合，并辅助判断进、退场时机，不仅是被动的呈现信息，更是结合历史数据提供建议，让新进投资者更有方向，资深理财者也有参考的依据。所以更加需要设计模式，特别是面向对象的一种设计模式。

# GoF 设计模式应用–创建型模式

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，对象的创建由相关的工厂来完成。就像我们去商场购买商品时，不需要知道商品是怎么生产出来一样，因为它们由专门的厂商生产。

## 工厂模式

工厂模式（Factory Pattern）是最常用的设计模式之一。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。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

### 介绍

--意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

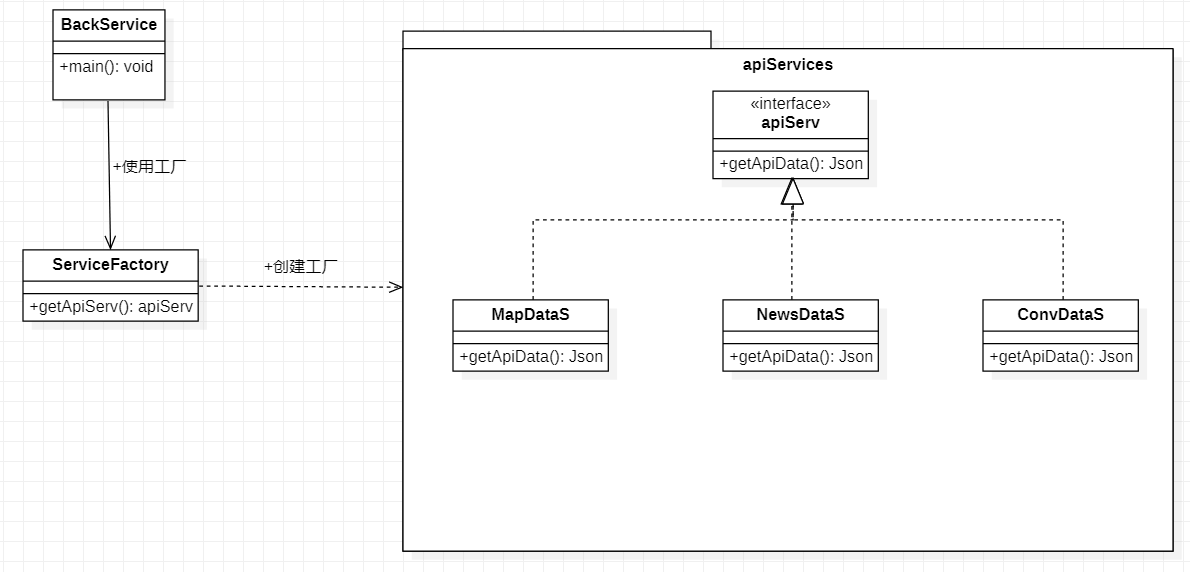
--主要解决：主要解决接口选择的问题。

--如何使用：我们明确地计划不同条件下创建不同实例时。

--如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

--关键代码：创建过程在其子类执行。

### 举例



在我们的工程中，我们使用了Django的一种架构 ，服务器端是通过dwebsoccket进行搭建的，支持usgwi多线程的链接，比较稳定。但是在线程调用数据的时候互相可能会出现问题，调用服务的时候也可能会出现很多的问题，所以可以将线程定义model化，将数据接口函数化。也是应用了这么一种思想

### 优缺点分析

工厂方法模式的主要优点有：

--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；

其缺点是：

--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。

## 抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。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。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象。

### 介绍

--意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

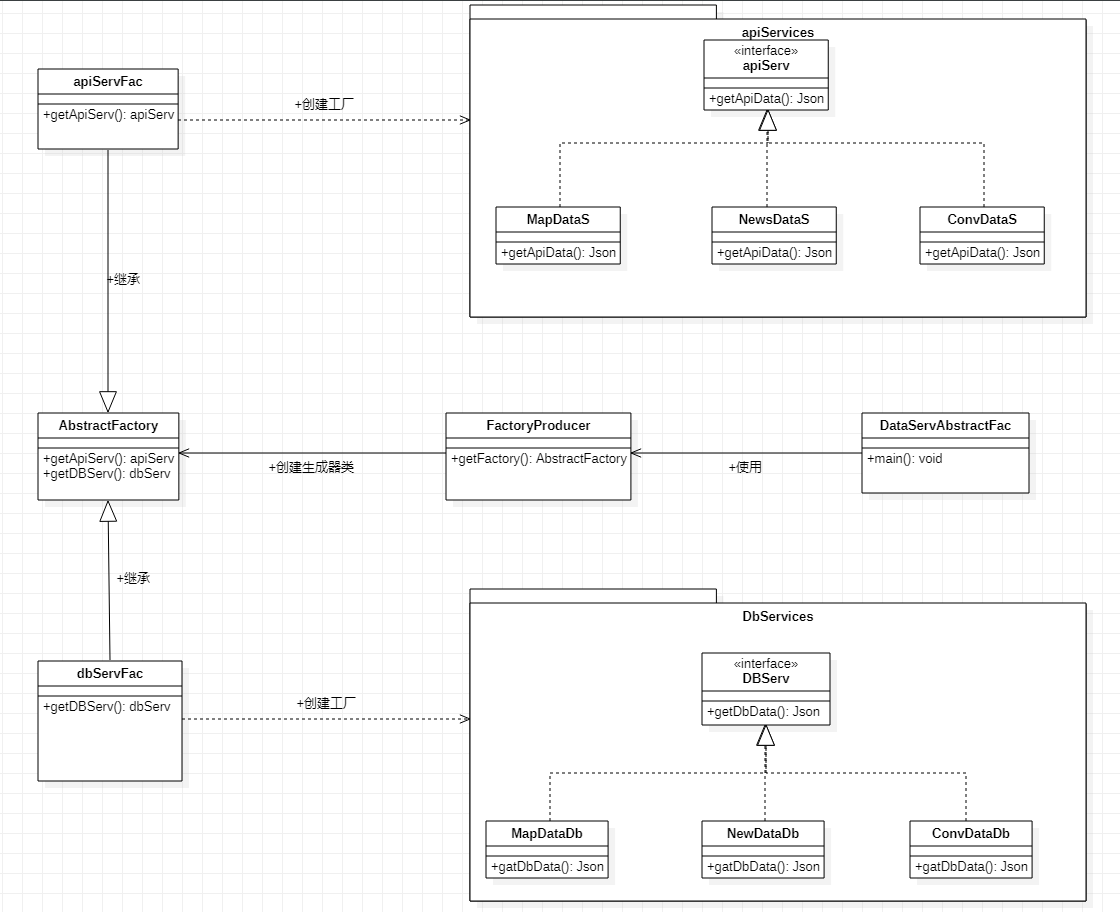
--主要解决：主要解决接口选择的问题。

--如何使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。

--如何解决：在一个产品族里面，定义多个产品。

--关键代码：在一个工厂里聚合多个同类产品。

### 举例



适用性：

1. 一个系统要独立于它的产品的创建、组合和表示时。

2. 一个系统要由多个产品系列中的一个来配置时。

3. 当你要强调一系列相关的产品对象的设计以便进行联合使用时。

4. 当你提供一个产品类库，而只想显示它们的接口而不是实现时。

### 优缺点分析

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：产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码。

## 单例模式

单例模式（Singleton Pattern）是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。

### 介绍

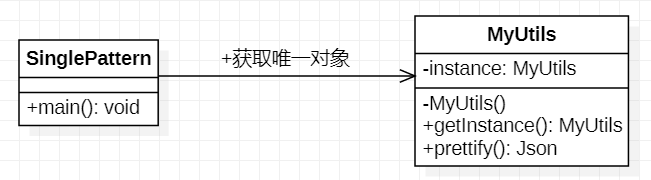
• 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

• 主要解决：一个全局使用的类频繁地创建与销毁。

• 如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

• 关键代码：构造函数是私有的。

### 举例



该类适用于单例模式的原因是，其目的在于格式化 json 数据包，任何 *controller* 在返回数据前均需要使用该类，如果不是单例模式，该类会被频繁的创建与销毁，这种效率浪费是没有必要的。该类并不需要对不同的 *controller* 进行特异化处理，因此可以作为单例被广泛使用。

### 优缺点分析

适用性：

1. 当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时。
2. 当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

缺点：

没有接口，不能继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化。

## 建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。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。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

### 介绍

--意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

--主要解决：主要解决在软件系统中，有时候面临着” 一个复杂对象” 的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

--何时使用：一些基本部件不会变，而其组合经常变化的时候。

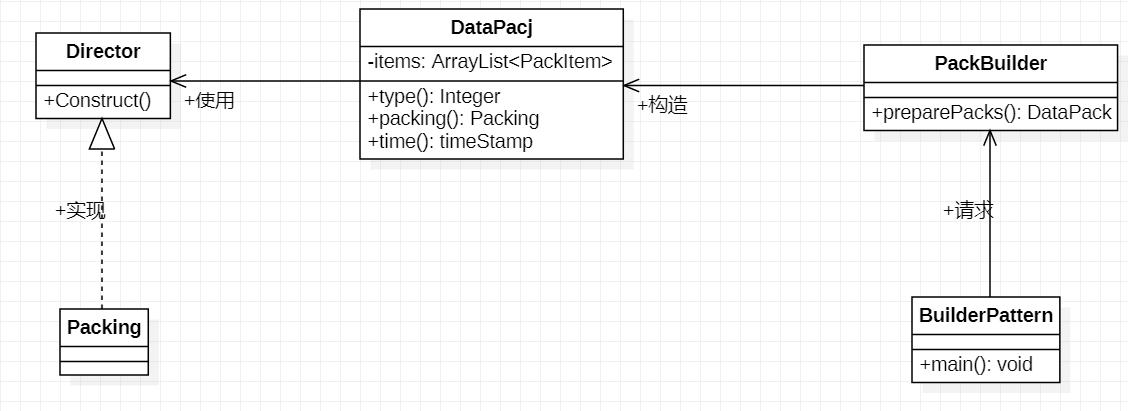
--如何解决：将变与不变分离开。

--关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

### 举例

当从服务器端从数据库向客户端返回数据时，需要根据数据库内容构建json 数据包，为了使得字段的映射关系清晰，构建json 数据包简洁，选择使用类进行数据处理。

但是客户端不同页面请求的数据内容显然存在不同字段，因此需要根据请求的类型特异性地构建相关类，由于json 数据包的字段较多，类的结构复杂，不同类的组成部分不同，但是组合的算法逻辑相同，因此使用建造者模式是一个比较好的选择。



举例，图中需要构建地图数据和新闻数据两种不同的数据包，在 *controller* 进行处理时，对象可以视作相同进行泛化处理，而在实际的 *service* 层，，算法动作包含获取数据和封装 json 数据包两个基础性的动作。

地图数据分为国家的地图疫情数据和附近的医院地理数据，但是 json 的数据字段时类似的，可以泛化为地图数据包，二者在获取数据上有一些不同，但是在封装为 json 时的逻辑是一致的。

新闻数据分为世界的新闻和用户收藏的新闻，json 的数据字段和地图数据不同，但是两种新闻的数据字段相同，可以泛化为新闻数据，两种特化子类在获取数据上有不同，但是在封装为 json 时的逻辑是一致的。

可以看到，通过将复杂类的不变部分和变化部分进行抽取，对公有方法提供接口封装，缩减了类的体积，各个功能子类的分界线变得清晰明确，易于管理。

### 优缺点分析

优点：建造者独立，易扩展；便于控制细节风险。

适用性：

1. 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

2. 当构造过程必须允许被构造的对象有不同的表示时。

缺点：需要生成的对象具有复杂的内部结构；需要生成的对象内部属性本身相互依赖

## 原型模式

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。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。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

### 介绍

• 意图：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

• 主要解决：在运行期建立和删除原型。

• 何时使用：

1、当一个系统应该独立于它的产品创建，构成和表示时。

2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。

3、为了避免创建一个与产品类层次平行的工厂类层次时。

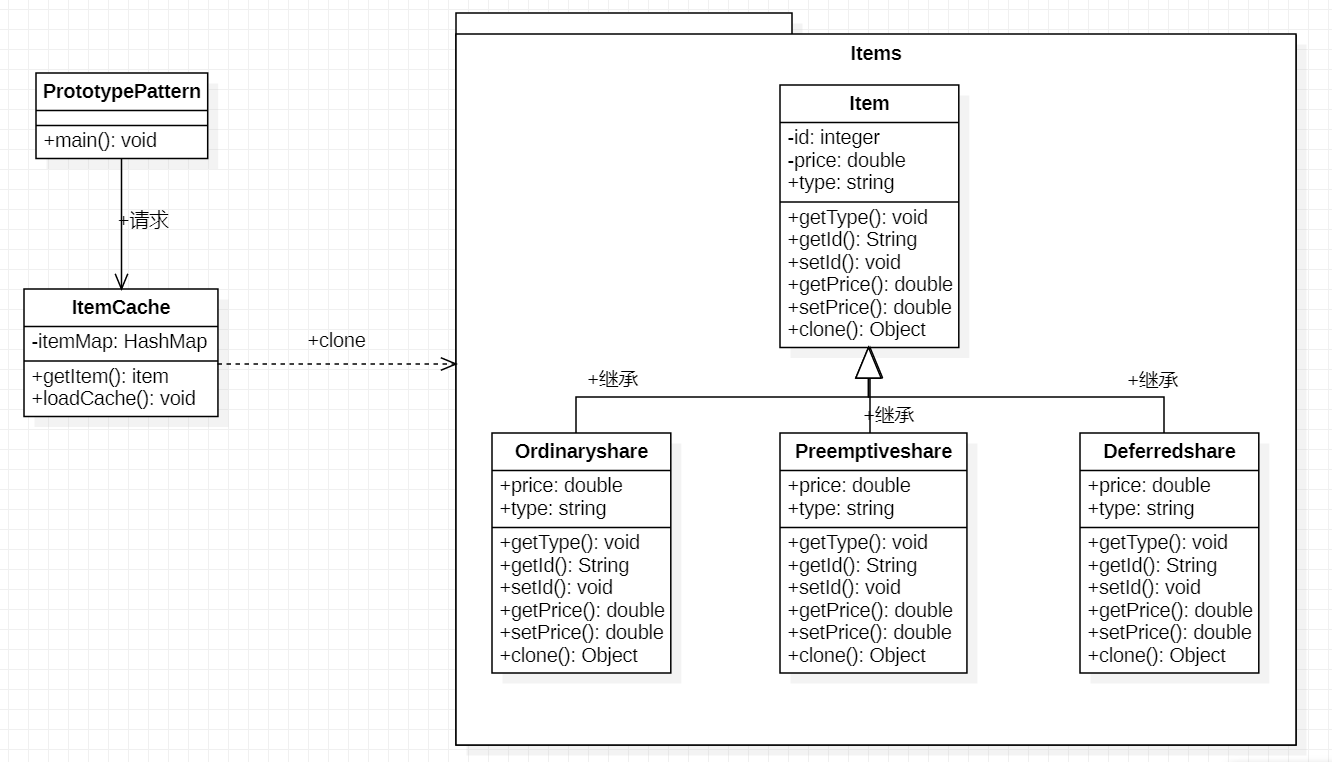
4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆

它们可能比每次用合适的状态手工实例化该类更方便一些。

• 如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。

• 关键代码：实现克隆操作，在JAVA 继承Cloneable，重写clone()。原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些“易变类”拥有稳定的接口。

### 举例



有三种不同的股票类型，显然每个股票类型实例只能是一种股票类型，但是每个实例都需要维护id, type和price字段，因此必须考虑使用继承基类的方式进行处理，使用原型模式可以规避掉每次冗余的实例初始化过程，而只需要进行clone即可。客户端的订单是动态变化的，不能保证每次的股票类型种类是相同的但是维护所有的可能性或者每次均创建一个新的实例，相对麻烦。使用clone，处理成本会降低许多。

### 优缺点分析

优点：

1. 性能提高
2. 逃避构造函数的约束

缺点：

1. 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。
2. 必须实现 Cloneable 接口。

# 4 GoF 设计模式应用-结构性模型

结构型模式涉及到如何组合类和对象以获得更大的结构。结构型模式采用继承机制来实现组合接口或实现。一个简单的例子是采用多重继承方法将两个以上的类组合成一个类，结果这个类包含了所有父类的性质。这一模式尤其有助于多个独立开发的的类库协同工作。

结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对象组合方式具有更大的灵活性， 而这种机制用静态类组合是不可能实现的。

## 4.1 适配器（Adaptor）

### 4.1.1 介绍

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。适配器模式将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

以下情况可使用Adapter 模式：

想使用一个已经存在的类，而它的接口不符合需求。

像创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即接口可能不兼容的类）协同合作。

（仅适用于对象Adapter）想使用一些已存在的子类，但不可能对每一个都进行子类化一匹配它们的接口。对象适配器可以适配它的父类接口。

下图是适配器的结构图：



对象适配器依赖于对象组合。

在上图中，我们可以看到一共有 4 类参与者：

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

### 4.1.2 举例

当代码有多人协作时，我们最初定义的类之间的接口调用可能不再得到实现，从而导致接口相互不兼容，因此我们可以利用适配器模式来实现接口的兼容。

例如，我们可能已经开发了一个类 Adaptee，但是我们所开发的另一个却是以 Target 类的形式想要调用 Adaptee 这个类的，那么我们需要开发一个新的类 Adapter 去适配这一接口的不兼容情况。

### 4.1.3 优缺点分析

类适配器和对象适配器有不同的权衡。

优点：

1) 可以让任何两个没有关联的类一起运行。

2) 提高了类的复用。

3) 增加了类的透明度。

4) 灵活性好。

缺点：

过多地使用适配器，会让系统非常零乱，不易整体进行把握。

## 4.2 桥接（Bridge）

### 4.2.1 介绍

桥接模式旨在将抽象部分与它的实现部分分离，使它们都可以独立地变化。当一个抽象可能有多个实现时，通常用继承来协调它们。抽象类定义对该抽象的接口，而具体的子类则用不同方式加以实现。但是此方法有时不够灵活。继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改、扩充和重用。

以下一些情况适用Bridge 模式:

• 不希望在抽象和它的实现部分之间有一个固定的绑定关系。

• 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。

• 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

• 在C++ 中想对客户完全隐藏抽象的实现部分。

• 在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

下图是桥接的结构图：



### 4.2.2 优缺点分析

优点：

1) 抽象和实现的分离。

2) 优秀的扩展能力。

3) 实现细节对客户透明。

缺点：

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

## 4.3 组合（Composite）

### 4.3.1. 介绍

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

以下一些情况适用 Composite 模式:

* + - * 想表示对象的部分-整体层次结构。
      * 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。组合模式的结构图如下：



在上图中，我们可以看到一共有4 类参与者：

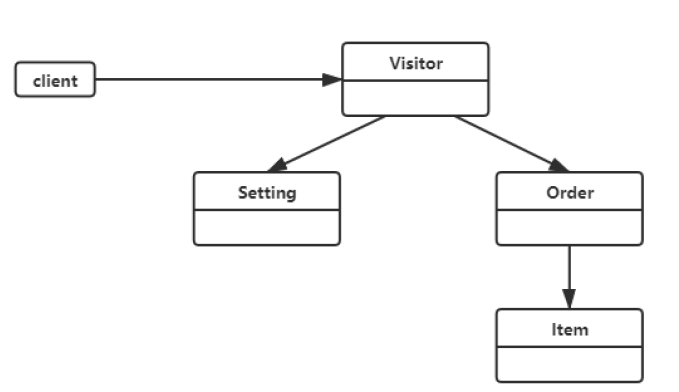
• Component ——为组合中的对象声明接口; 在适当的情况下，实现所有类共有接口的缺省行为; 声明一个接口用于访问和管理Component 的子组件。

• Leaf ——在组合中表示叶节点对象，叶节点没有子节点; 在组合中定义图元对象的行为。

• Composite ——定义有子部件的那些部件的行为; 存储子部件; 在Component 接口中实现与子部件有关的操作。

• Client ——通过Component 接口操纵组合部件的对象。

### 4.3.2 举例



在上图举例的系统中，Setting 类和Order 类是Visitor 类的一部分，而Item 类又是Order 类的一部分。其中Visitor 类是指使用小程序且可以发起或参与拼团的用户；Setting 类用于保存用户的个人信息；Order 类用于记录用户拼团生成的订单；Item 类表示一件商品在数据库中的信息。用户使用Component 类接口与组合结构中的对象进行交互。如果接收者是一个叶节点, 则直接处理请求。如果接收者是Composite, 它通常将请求发送给它的子部件，在转发请求之前与/或之后可能执行一些辅助操作。

### 4.3.3 优缺点分析

优点:

1. 定义了包含基本对象和组合对象的类层次结构，客户代码中，任何用到基本对象的地方都可以使用组合对象。
2. 简化客户代码，客户可以一致地使用组合结构和单个对象。
3. 使得更容易增加新类型的组件。

缺点：

* 1. 使用Composite 时，不能依赖类型系统施加这些约束，而必须在运行时刻进行检查。
  2. 使你的设计变得更加一般化，很难限制组合中的组件，无法实现在一个组合中只有某些特定的组件

## 4.4 外观（Facade）

## 4.4 外观（Facade）

### 4.4.1 介绍

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

将一个系统划分成为若干个子系统有利于降低系统的复杂性。一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小。达到该目标的途径之一是就是引入一个外观（facade）对象，它为子系统中较一般的设施提供了一个单一而简单的界面。

以下一些情况适用 Facade 模式:

-- 要为一个复杂子系统提供一个简单接口时。

-- 客户程序与抽象类的实现部分之间存在着很大的依赖。

-- 构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。

### 4.4.2 举例：

用“外观模式”设计一个婺源特产的选购界面。

其结构图如下图所示：



使用外观模式分析如下：

1) 本实例的外观角色 WySpecialty 是 JPanel 的子类，它拥有 8 个子系统角色

Specialty1~Specialty8，它们是图标类（ImageIcon）的子类对象，用来保存该婺源特产

的图标。

2) 外观类（WySpecialty）用 JTree 组件来管理婺源特产的名称，并定义一个事件处理方

法 valueClianged(TreeSelectionEvent e)，当用户从树中选择特产时，该特产的图标对象

保存在标签（JLabd）对象中。

3) 客户窗体对象用分割面板来实现，左边放外观角色的目录树，右边放显示所选特产图像

的标签。

**4.4.3 优缺点分析**

优点：

1) 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。

2) 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。

3) 降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因

为编译一个子系统不会影响其他的子系统，也不会影响外观对象。

缺点：

1) 不能很好地限制客户使用子系统类，很容易带来未知风险。

2) 增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

## 4.5 Flyweight（享元）

### 4.5.1 介绍

* 意图：

运用共享技术有效地支持大量细粒度的对象。

* 适用性：

一个应用程序使用了大量的对象。

完全由于使用大量的对象，造成很大的存储开销。

对象的大多数状态都可变为外部状态。

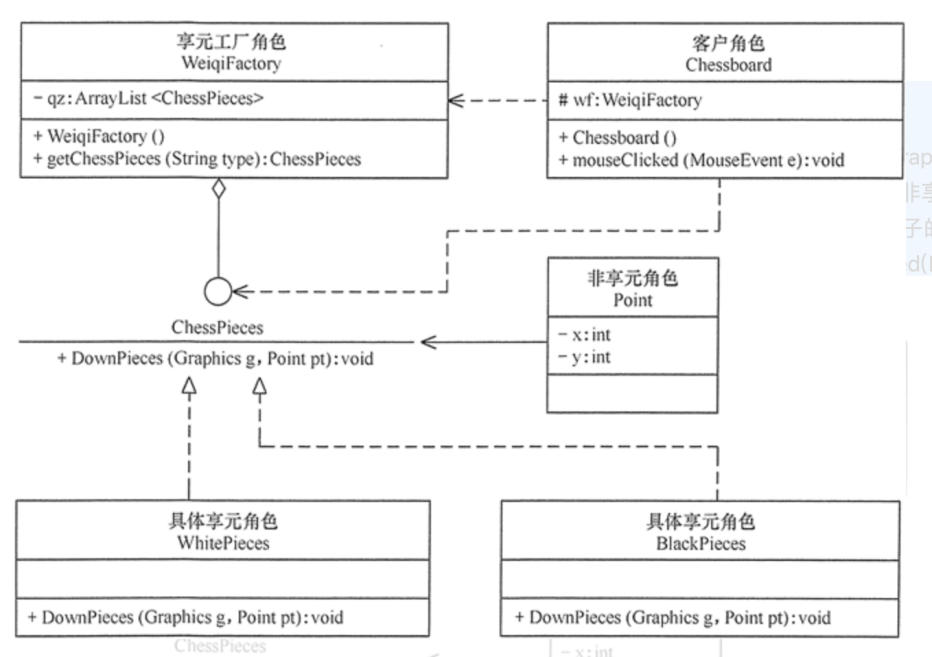
如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

应用程序不依赖于对象标识。由于Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

### 4.5.2 举例

五子棋同围棋一样，包含多个“黑”或“白”颜色的棋子，所以用享元模式比较好。

本实例中的棋子（ChessPieces）类是抽象享元角色，它包含了一个落子的 DownPieces(Graphics g,Point pt) 方法；白子（WhitePieces）和黑子（BlackPieces）类是具体享元角色，它实现了落子方法；Point 是非享元角色，它指定了落子的位置；WeiqiFactory 是享元工厂角色，它通过 ArrayList 来管理棋子，并且提供了获取白子或者黑子的 getChessPieces(String type) 方法；客户类（Chessboard）利用 Graphics 组件在框架窗体中绘制一个棋盘，并实现 mouseClicked(MouseEvent e) 事件处理方法，该方法根据用户的选择从享元工厂中获取白子或者黑子并落在棋盘上。图 2 所示是其结构图。



### 4.5.3 应用场景和扩展

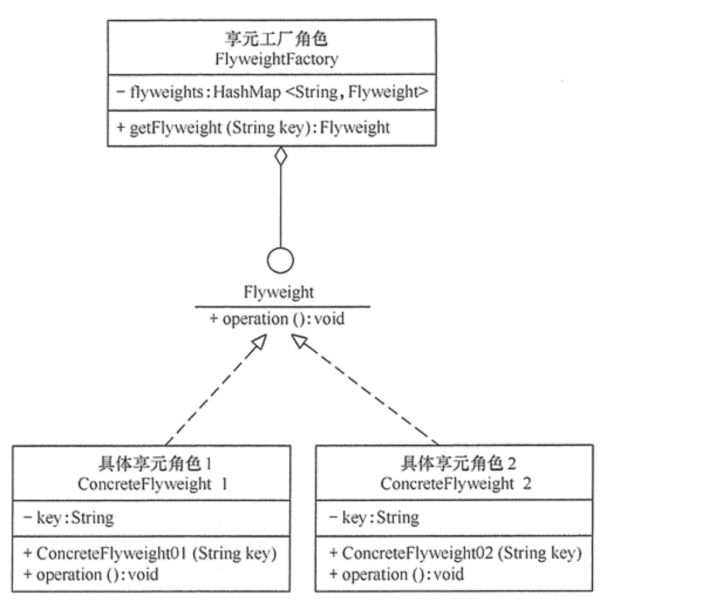
以下下几种情形适合采用享元模式：

1. 系统中存在大量相同或相似的对象，这些对象耗费大量的内存资源。

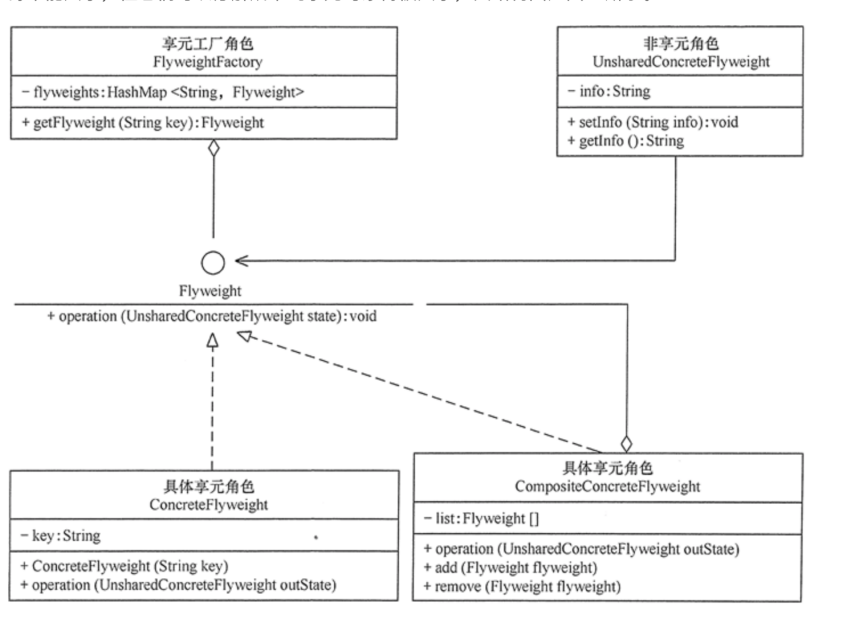
2. 大部分的对象可以按照内部状态进行分组，且可将不同部分外部化，这样每一个组只需保存一个内部状态。

3. 由于享元模式需要额外维护一个保存享元的[数据结构](http://c.biancheng.net/data_structure/)，所以应当在有足够多的享元实例时才值得使用享元模式。

1. 单纯享元模式，这种享元模式中的所有的具体享元类都是可以共享的，不存在非共享的具体享元类，其结构图如下：



1. 复合享元模式，这种享元模式中的有些享元对象是由一些单纯享元对象组合而成的，它们就是复合享元对象。虽然复合享元对象本身不能共享，但它们可以分解成单纯享元对象再被共享：



## 4.6 Proxy（代理）

### 4.6.1 介绍

* 定义：

代理模式（Proxy）为另一个对象提供一个替身或占位符以控制对这个对象的访问，简而言之就是用一个对象来代表另一个对象。

从字面上看也很明显，就是代表某个对象去做某件事，比如你办房产证懒得跑路，怎么办呢，可以找个结构代理你本人跑一下咯，你只需要提供你的主要信息（身份证，购房合同，房款发票等），机构就能全权代表你去做这件事了，同样的对象也只需要提供它的接口和重要信息，代理类也能完成它的工作，并且还可以加入很多额外的动作，比如在访问对象前做什么事，访问后做什么事等。

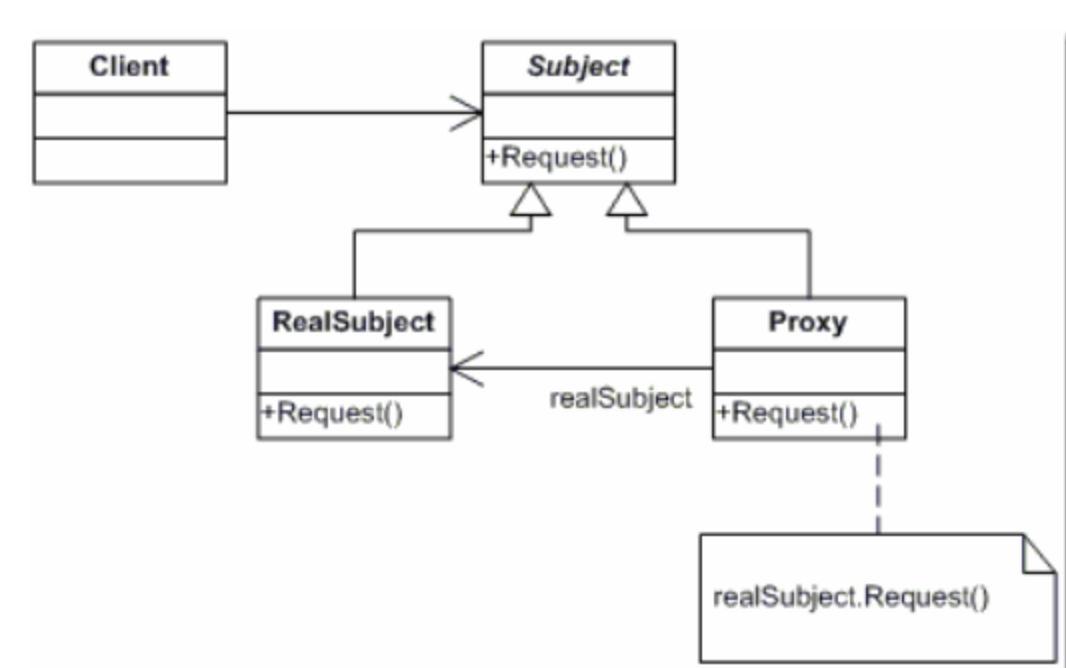
* 意图：

提供其他对象一个代理或占位符，来控制该对象的访问权限。

* 动机：

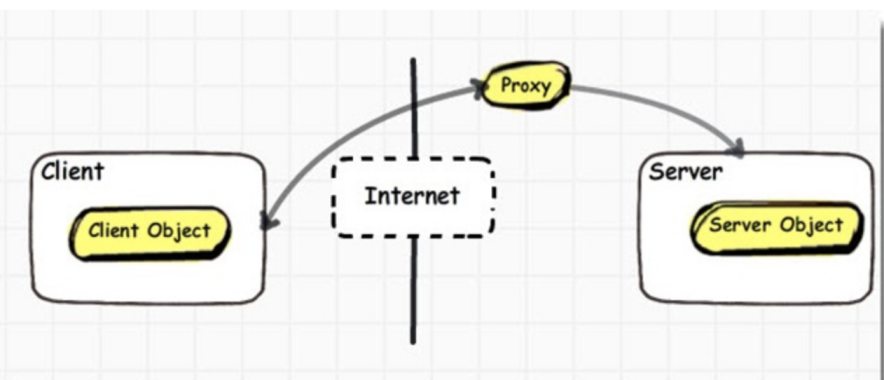
      为什么我们要控制对象的访问权限呢？其中一个原因是通过控制来延迟对象的创建和实例化，直到真正需要使用该对象才进行创建和实例化。由于一些对象创建和实例化需要占用大量系统资源，但我们并不能确定用户一定会调用该对象，所以通过延迟对象实例化来减缓系统资源的消耗。例如文档编辑器如word，我们可以在里面插入链接、图片等，但是并不是我们每次打开word时都有创建和实例化这些对象，特别是实例化图片对象很消耗资源，而且我们有必要实例化所有图片吗？当我们在查看word时只是看到其中的一部分，所以没有必要实例化所以资源，当我们看下一页时再实例化也不迟。

* 结构图：

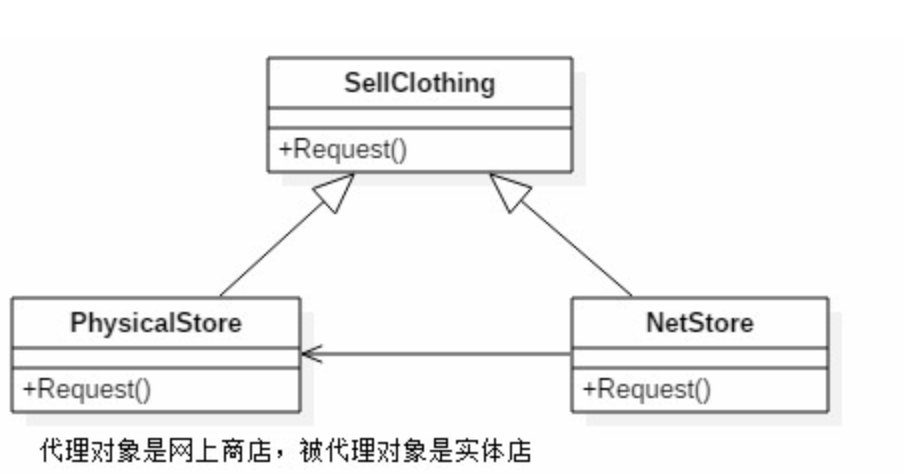


### 4.6.2 举例：

### 案例1: 网络中的代理模式



### 案例2: 针对卖衣服这种商业买卖，可以在实体店销售，也可以授权于网上商店，也就是说网上商店可以代理实体店的销售，网上商店可以增加自身的操作，如双十一打折。



### 4.6.3 优缺点分析

优点：

1. 职责清晰。
2. 高扩展性。

3、智能化。

缺点：

1. 由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

## 4.7 Decorator（装饰）

### 4.7.1 介绍

**概述：**

### 若你从事过面向对象开发，实现给一个类或对象增加行为，使用继承机制，这是所有面向对象语言的一个基本特性。如果已经存在的一个类缺少某些方法，或者须要给方法添加更多的功能（魅力），你也许会仅仅继承这个类来产生一个新类—这建立在额外的代码上。

### 通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。如果  你希望改变一个已经初始化的对象的行为，你怎么办？或者，你希望继承许多类的行为，改怎么办?前一个，只能在于运行时完成，后者显然时可能的，但是可能会导致产生大量的不同的类—可怕的事情。

**问题：你如何组织你的代码使其可以容易的添加基本的或者一些很少用到的 特性，而不是直接不额外的代码写在你的类的内部？**

**解决方案：**

[装饰器模式](http://blog.csdn.net/hguisu/article/details/7531960)： 动态地给一个对象添加一些额外的职责或者行为。就增加功能来说， Decorator模式相比生成子类更为灵活。

[装饰器模式](http://blog.csdn.net/hguisu/article/details/7531960)提供了改变子类的灵活方案。装饰器模式在不必改变原类文件和使用继承的情况下，动态的扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

       当用于一组子类时，装饰器模式更加有用。如果你拥有一族子类（从一个父类派生而来），你需要在与子类独立使用情况下添加额外的特性，你可以使用装饰器模式，以避免代码重复和具体子类数量的增加。

### 4.7.2 适用性：

以下情况使用Decorator模式

1)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。

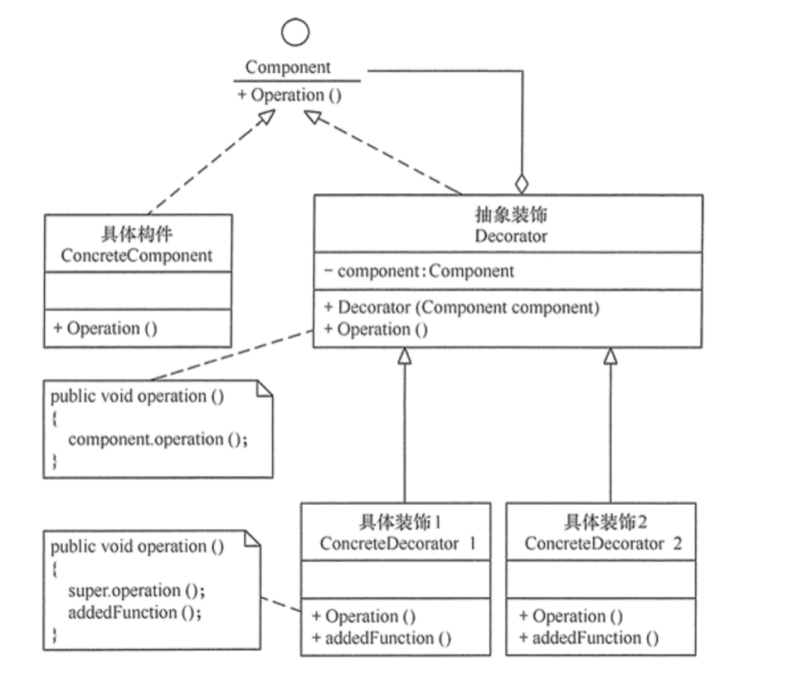
2)处理那些可以撤消的职责。

3)当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，

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。

另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

**类图：**



### 4.7.3 举例：

 linux串行子系统，其层次结构如下，

（1） 关注UART或者其他硬件的驱动

（2） tty驱动层，将上层驱动程序和硬件隔离

（3） 线路规程，以特殊方式格式化从客户或者硬件上得到的数据，有助于复用底层代码

这个子系统中，线路规程和装饰模式类似。

### 4.7.4 优缺点：

**优点：**

比静态继承更灵活。

避免在层次结构高层的类有太多的特征。

**缺点：**

Decorator 是一个透明的包装，其与 Component 还是有些差别的。

采用 Decorator 模式进行系统设计往往会产生许多看上去类似的小对象。导致很难学习系统，排错也很困难。

### 4.7.5 相关模式：

* Decorator 模式不同于 Adapter 模式，因为 Decorator 仅改变对象的职责而不改变它的接口，而 Adapter 将给对象一个全新的接口。
* 可以将 Decorator 视为一个退化的、仅有一个组件的 Composite。然而，Decorator 仅给对象添加额外的职责，它的目的不在于对象聚集。
* 用一个 Decorator 可以改变对象的外表，而 Strategy 模式使得你可以改变对象的内核。这是改变对象的两种途径。
* 当 Component 类原本就很庞大时，使用 Decorator 模式的代价太高，Strategy 模式相对更好一些。

**5 GoF 设计模式应用-行为模式**

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。

结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对象组合方式具有更大的灵活性，而这种机制用静态类组合是不可能实现的。

**5.1 Interpreter（解释器）**

**5.1.1 介绍**

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

2. 别名: 无

3. 动机:

如果一个特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构造一个解释器，该解释器通过解释这些句子来解决该问题。

例如，搜素匹配一个模式的字符串是一个常见问题。正则表达式是描述字符串模式的一个标准语言。与其为每一个的模式都构造一个特定的算法，不如使用一种通用的搜索算法来执行一个正则表达式，该正则表达式定义了待匹配字符串的集合。

解释器模式描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。在下面的例子中，本设计模式描述了如何为正则表达式定义一个文法，如何表达一个特定的正则表达式，以及如何解释这个正则表达式。

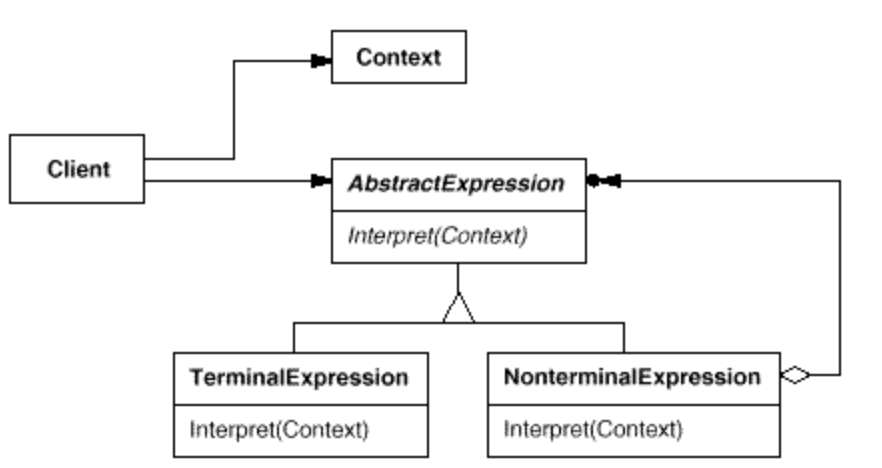
4. 适用性:

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

文法很简单 对于复杂的文法，文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式，这样可以节省空间而且还可能节省时间。

效率不是一个关键问题 最高效的解释器通常不是通过直接解释语法分析树实现的，而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机，但即使在这种情况下，转换器仍可用解释器模式实现，该模式仍是可用的。

5. 结构图:



**5.1.2 相关模式**

**--抽象语法树是一个 Composite 模式的实例。**

**--可以使用 Flyweight 模式在抽象语法树中共享终结符。**

**--可以使用 Iterator 模式遍历解释器结构。**

**--可以使用 Visitor 模式在一个类中维护抽象语法树中的各个节点的行为。**

**5.1.3 效果**

解释器模式有下列的效果：

1） 易于改变和扩展文法 因为该模式使用类来表达文法规则。你可以使用1继承来改变或扩展该文法。已有的表达式可被增量式的改变，而新的表达式可以定义为旧表达式的变体。

2） 易于实现文法 定义抽象语法树种的各个节点的类的实现大体类似。这些类易于直接编写，通常他们也可用一个编译器或语法分析程序器自动生成。

3） 增加了新的解释器表达式的方式 解释器模式使得实现新表达式"计算"变得容易。例如，你可以在表达式类上定义一个新的操作以支持优美打印或者表达式的类型检查。如果你经常创新的解释表达式的方式，那么可以考虑使用Visitor模式来避免修改这些代表文法的类。

**5.2 模板方法（Template Method）**

**5.2.1 介绍：**

**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。它是一种类行为型模式。**

**5.2.2 模式的结构：**

模板方法模式包含以下主要角色。

1）抽象类/抽象模板（Abstract Class）

抽象模板类，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。这些方法的定义如下。  
① 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。  
② 基本方法：是整个算法中的一个步骤，包含以下几种类型。

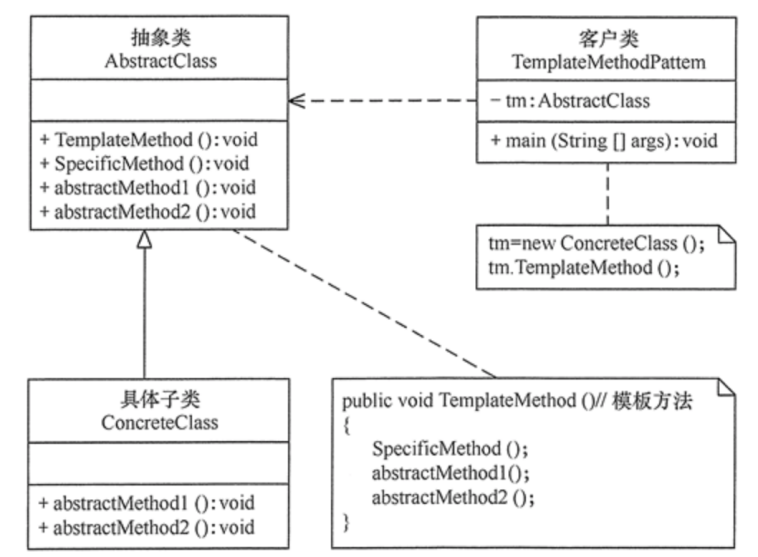
抽象方法：在抽象类中声明，由具体子类实现。

具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它。

钩子方法：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。

2）具体子类/具体实现（Concrete Class）

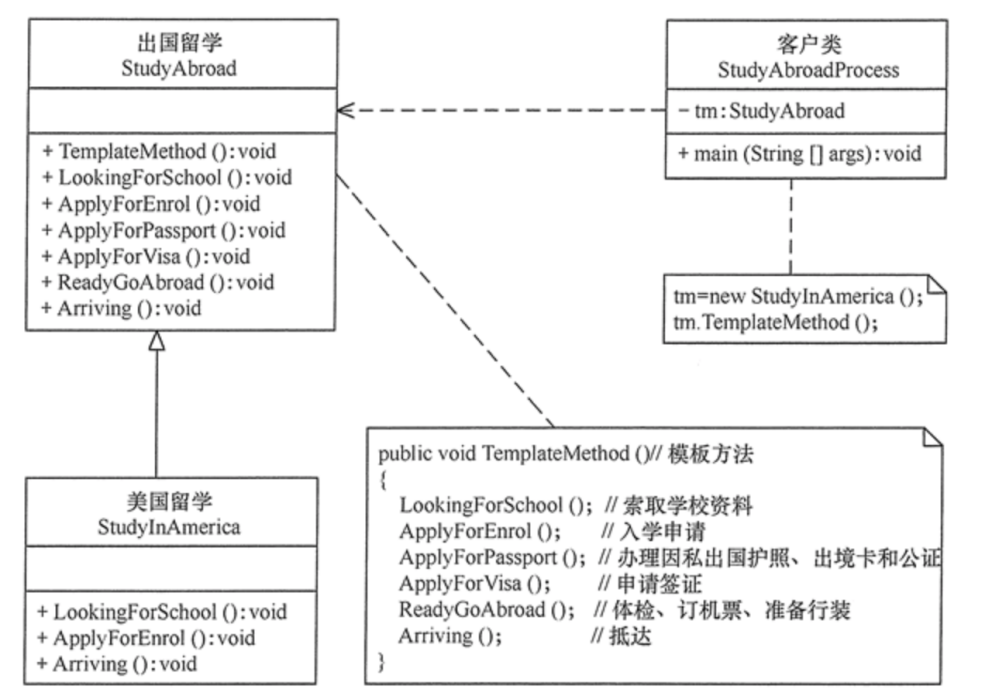
具体实现类，实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。  
**模板方法模式的结构图下图所示：**



**5.2.2. 举例**

用模板方法模式实现出国留学手续设计程序：

分析：出国留学手续一般经过以下流程：索取学校资料，提出入学申请，办理因私出国护照、出境卡和公证，申请签证，体检、订机票、准备行装，抵达目标学校等，其中有些业务对各个学校是一样的，但有些业务因学校不同而不同，所以比较适合用模板方法模式来实现。  
  
在本实例中，我们先定义一个出国留学的抽象类 StudyAbroad，里面包含了一个模板方法 TemplateMethod()，该方法中包含了办理出国留学手续流程中的各个基本方法，其中有些方法的处理由于各国都一样，所以在抽象类中就可以实现，但有些方法的处理各国是不同的，必须在其具体子类（如美国留学类 StudyInAmerica）中实现。如果再增加一个国家，只要增加一个子类就可以了：



**5.2.3. 优缺点分析**

该模式的主要优点如下。

--它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。

--它在父类中提取了公共的部分代码，便于代码复用。

--部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

该模式的主要缺点如下。

--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，间接地增加了系统实现的复杂度。

--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。

--由于继承关系自身的缺点，如果父类添加新的抽象方法，则所有子类都要改一遍。

**5.3 责任链（Chain of Responsibility）**

**5.3.1 介绍**

**意图：**

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

**基本思想：**

责任链模式描述的就是如何推卸责任，说的简洁点，就是踢皮球哈哈。举个例子，有时候，出了某件事，我们去解决，找到A，结果A踢皮球，说这不关我的事，去找B解决，然后我们就去找B，结果B也说，这跟我没关系，快去找C，就这样，我们就被踢来踢去，这就是责任链模式的思想，在找到正确的人解决之前，我们被不断的踢给一个有一个人，就是推卸责任。

上面的例子，可能有点贬义，但在实际编程中，有时候确实存在需要推卸责任的情况，比如，当我们接受到一个请求时，当前的程序暂时无法处理这个请求，于是就需要把请求给别人去处理。如果是web开发人员，对此应该很熟悉，当服务器收到一个客户端的请求时，首先会解析请求，action层不会处理请求，而是将请求的参数等信息进行简单的解析处理，然后根据请求的内容信息等将请求具体转发给service去处理。

当一个人被要求做一件事的时候，如果他自己可以做，那他就自己做了，如果他自己做不了，那就转发给另一个人做，另一个人也是一样，如果他自己可以做，就做，不可以做，就给别人做。这就是责任链模式的基本思想

**5.3.2 适用性**

--弱化了发出请求的人和处理请求的人之间的关系  
发出请求的人只需要向第一个具体的处理者发送请求，然后就可以不用管了，处理者会在责任链上自己寻找处理的方法。这样就解耦了处理者和请求者之间的关系。如果我们不采取责任链模式，那么请求者就必须要很清楚哪个处理者能处理它的请求，就必须对所有的处理者都有所了解，类似于上帝视角，然而在实际中，要求请求这了解这么多是不实际的

--可以动态的改变责任链  
责任链还有的好处就是可以动态的改变责任，删除或者添加或者改变顺序。

--让各个处理者专注于实现自己的职责  
责任链模式同时还做到了处理者之间的解耦，处理者自己专注于自己的处理逻辑就好，不管其他处理者干什么。

--推卸责任也可能导致处理延迟  
我们可以责任链模式需要在责任链上传播责任，直至找到合适的处理对象。这样提高了程序的灵活性，但同时也出现了处理的延迟，因为有一个寻找的过程。所以需要低延迟的情况下，就不应该使用责任链模式

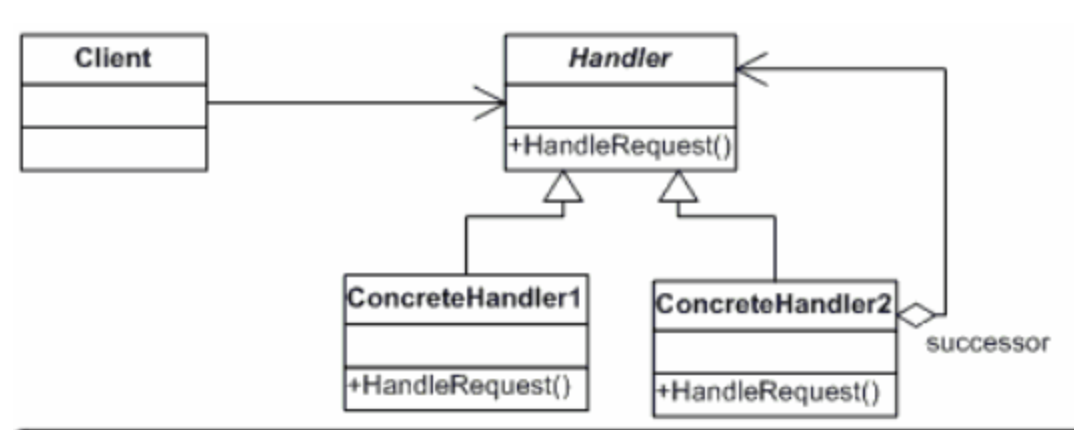
**5.3.3 参与角色和分析：首先，责任链模式中，存在着这么几个角色：**

1. Handler处理者  
handler金额use定义了处理请求的接口，handler知道，下一个处理者是谁，如果自己无法处理请求，就转给下一个处理者。  
在实例中对应的是，support类和support方法

2. concreteHandler(具体的处理者)  
具体的处理者是处理请求的具体角色。  
在此实例中，由NoSupport角色和其他几个类扮演

3. Client  
请求者角色，就是向第一个具体的handler发送请求的角色，并连接好责任链，实例中对应的是main类的main方法。

类图：



**5.4 Command（命令）**

**5.4.1 介绍**

将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化;对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作 (Action) 模式或事务 (Transaction) 模式。

命令模式可以将请求发送者和接收者完全解耦，发送者与接收者之间没有直接引用关系，发送请求的对象只需要知道如何发送请求，而不必知道如何完成请求。

**5.4.2 适用情况**

以下一些情况适用 Command 模式:

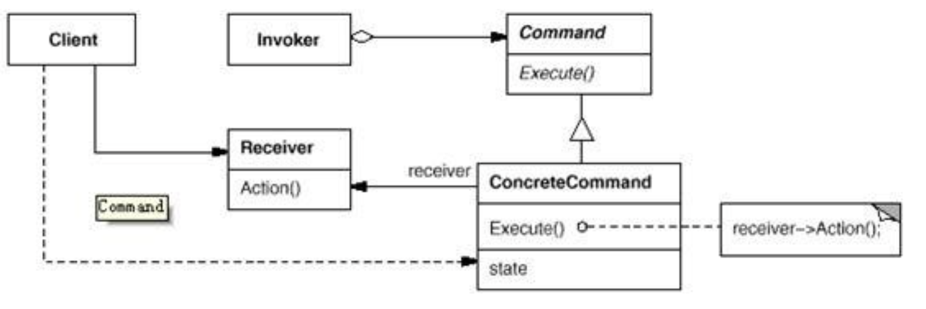
1. 当系统需要将请求调用者与请求接收者解耦时，命令模式使得调用者和接收者不直接交互。

2. 当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能。

3. 当系统需要执行一组操作时，命令模式可以定义宏命令来实现该功能。

4. 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。

命令模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

Command(抽象命令类):抽象命令类一般是一个抽象类或接口，在其中声明了用于执行请求的 execute() 等方法，通过这些方法可以调用请求接收者的相关操作。

ConcreteCommand(具体命令类):具体命令类是抽象命令类的子类，实现了在抽象命令类中 声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中。在实现 execute() 方 法时，将调用接收者对象的相关操作 (Action)。

Invoker(调用者):调用者即请求发送者，它通过命令对象来执行请求。一个调用者并不需要 在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以将一 个具体命令对象注入其中，再调用具体命令对象的 execute() 方法，从而实现间接调用请求接 收者的相关操作。

Receiver(接收者):接收者执行与请求相关的操作，它具体实现对请求的业务处理。

**5.4.3 举例：**

在本系统中，我们的股市数据库需要收集大量的交易信息，系统收集统整交易信息后，将处理好的数据发给后台，协助后台更新数据库。如果单纯以每交易的形式发送给后台，后台可能接受到大量重复的信息，导致后台资源大量被占用。所以系统进行初步整合减少耦合。

**5.4.4 优缺点分析**

**优点:**

降低系统的耦合度。由于请求者与接收者之间不存在直接引用，因此请求者与接收者之间实现完全解耦，相同的请求者可以对应不同的接收者，同样，相同的接收者也可以供不同的请求 者使用，两者之间具有良好的独立性。

新的命令可以很容易地加入到系统中。由于增加新的具体命令类不会影响到其他类，因此增 加新的具体命令类很容易，无须修改原有系统源代码，甚至客户类代码，满足“开闭原则”的 要求。可以比较容易地设计一个命令队列或宏命令(组合命令)。

**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。因为针对每一个对请求接收者的调用操作都需要设计一个具体命令类，因此在某些系统中可能需要提供大量的具体命令类，这将影响命令模式的使用。

**5.5 Iterator（迭代器）**

**5.5.1 介绍**

既然将遍历方法封装在聚合类中不可取，那么聚合类中不提供遍历方法，将遍历方法由用户自己实现是否可行呢? 答案是同样不可取，因为这种方式会存在两个缺点：暴露了聚合类的内部表示，使其数据不安全，增加了客户的负担。

迭代器模式能较好地克服以上缺点，它在客户访问类与聚合类之间插入一个迭代器，这分离 了聚合对象与其遍历行为，对客户也隐藏了其内部细节，且满足“单一职责原则”和“开闭原则”， 如 Java 中的 Collection、List、Set、Map 等都包含了迭代器。

**5.5.2 适用情况**

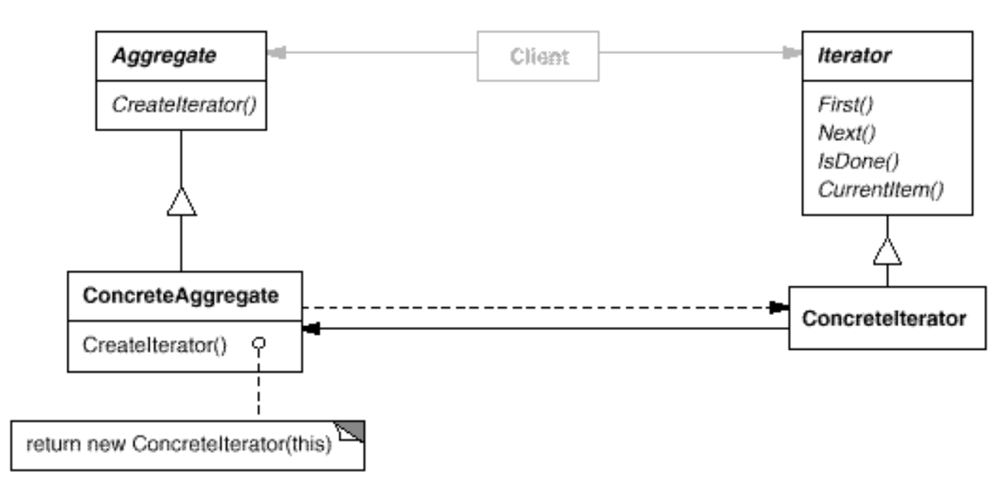
以下一些情况适用 Iterator 模式:

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。

迭代器模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

Aggregate(抽象聚合):定义存储、添加、删除聚合对象以及创建迭代器对象的接口。

ConcreteAggregate(具体聚合):实现抽象聚合类，返回一个具体迭代器的实例。

Iterator(抽象迭代器):定义访问和遍历聚合元素的接口，通常包含 hasNext()、first()、next() 等方法。

Concretelterator(具体迭代器):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。

**5.5.3 举例**

在本系统中，没有用到迭代器这一设计模式，这里用列表来进行举例说明。当我们在设计列表的时候，应该提供一种方法来让别人可以访问它的元素，而又不需暴露它的内部结构。同时，针对不同的需要，可能要以不同的方式遍历这个列表。为了解决列表接口出现多种遍历操作的问题，我们采用迭代器模式实现设计。

迭代器模式将对列表的访问和遍历从列表对象中分离出来并放入一个迭代器(iterator)对象中。迭代器类定义了一个访问该列表元素的接口。迭代器对象负责跟踪当前的元素; 即, 它知道哪些元素已经遍历过了。

**5.5.4 优缺点分析**

优点:

访问一个聚合对象的内容而无须暴露它的内部表示。

遍历任务交由迭代器完成，这简化了聚合类。

它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。

增加新的聚合类和迭代器类都很方便，无须修改原有代码。

封装性良好，为遍历不同的聚合结构提供一个统一的接口。

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。

**5.6 Mediator（中介者）**

**5.6.1 介绍**

在现实生活中，常常会出现好多对象之间存在复杂的交互关系，这种交互关系常常是“网状结构”， 它要求每个对象都必须知道它需要交互的对象。

中介者模式就是定义一个中介对象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。中介者模式又叫调停模式，它是迪米特法则的典型应用。

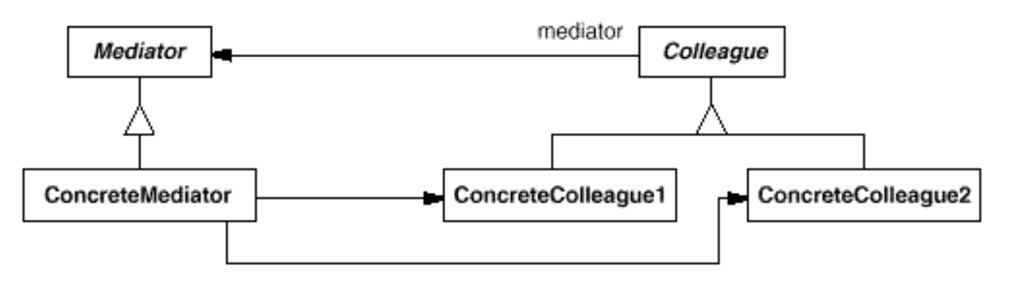
**5.6.2 适用情况**

以下一些情况适用 Mediator 模式:

当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用时。

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。

中介者模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

Mediator(抽象中介者):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。

ConcreteMediator(具体中介者):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。

Colleague(抽象同事类):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。

ConcreteColleague(具体同事类):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。

**5.6.3 举例**

在本系统的购买股票本身就是中介者模式的一种实例。作为股票市场和客户的中间环节。

**5.6.4 优缺点分析**

优点:

降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。

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调动它们的次序，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。每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，不该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。

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。

**5.7 Memento（备忘录）**

**5.7.1 介绍**

备忘录(Memento)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。该模式又叫快照模式。

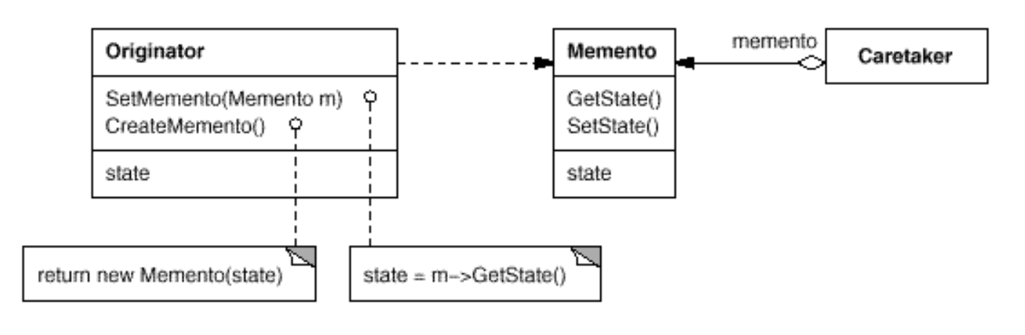
**5.7.2 适用性分析**

以下一些情况适用 Memento 模式:

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，Eclipse 等软件在编辑时按 Ctrl+Z 组合键，还有数据库中事务操作。

备忘录模式的结构图如下:



在上图中，我们可以看到一共有 3 类参与者:

Originator(发起人):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。

Memento(备忘录):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。

Caretaker(管理者):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。

**5.7.3 举例**

在本系统中，并没有使用到备忘录模式，在这里举图形编辑器的例子进行说明。这个编辑器支持 图形对象间的连线。用户可用一条直线连接两个矩形, 而当用户移动任意一个矩形时, 这两个矩形仍 能保持连接。在移动过程中，编辑器自动伸展这条直线以保持该连接。

为了实现取消或者撤回操作，我们采用备忘录设计模式。一个备忘录(memento)是一个对象, 它存储另一个对象在某个瞬间的内部状态，而后者称为备忘录的原发器 (originator)。当需要设置原发器的检查点时, 取消操作机制会向原发器请求一个备忘录。原发器用描述当前状态的信息初始化 该备忘录。只有原发器可以向备忘录中存取信息，备忘录对其他的对象“不可见”。

**5.7.4 优缺点分析**

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。

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。

**5.8 Observer（观察者）**

**5.8.1 介绍**

观察者模式是用于建立一种对象与对象之间的依赖关系，一个对象发生改变时将自动通知其他对象，其他对象将相应作出反应。在观察者模式中，发生改变的对象称为观察目标，而被通知的对象称为观察者，一个观察目标可以对应多个观察者，而且这些观察者之间可以没有任何相互联系，可以根据需要增加和删除观察者，使得系统更易于扩展。

**5.8.2 适用情况**

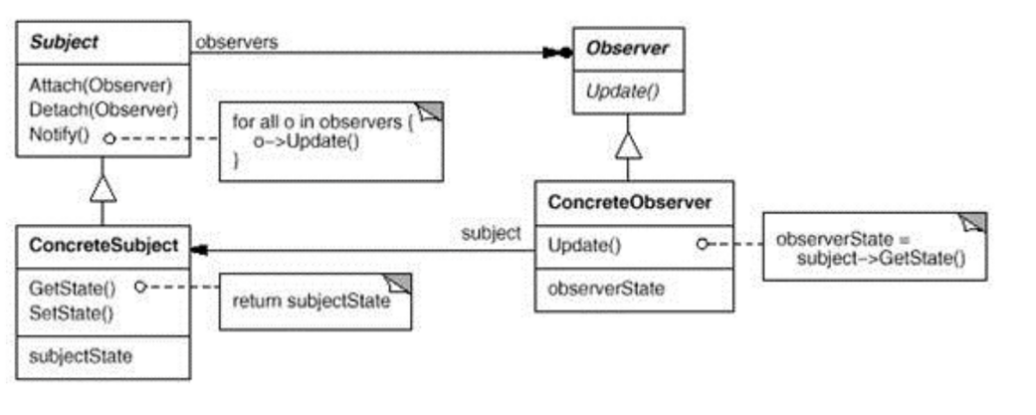
以下一些情况适用 Observer 模式:

要为一个复杂子系统提供一个简单接口时。

客户程序与抽象类的实现部分之间存在着很大的依赖。

构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。

观察者模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

Subject(被观察者或目标，抽象主题):被观察的对象。当需要被观察的状态发生变化时，需 要通知队列中所有观察者对象。Subject 需要维持(添加，删除，通知)一个观察者对象的队 列列表。

ConcreteSubject(具体被观察者或目标，具体主题):被观察者的具体实现。包含一些基本的 属性状态及其他操作。

Observer(观察者):接口或抽象类。当 Subject 的状态发生变化时 Observer 对象将通过一个 callback 函数得到通知。

ConcreteObserver(具体观察者):观察者的具体实现。得到通知后将完成一些具体的业务逻 辑处理。

**5.8.3 举例**

观察者机制常被使用在前端中，比如说页面上有许多地方显示用户信息，而当我们更改用户信息后，会发送信号给各个观察者更改显示信息，而不需要要在这些现实模块中不断地获取用户信息，降低了两者间的耦合关系。

**5.8.4 优缺点分析**

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。 • 目标与观察者之间建立了一套触发机制。

缺点:

目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。 • 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

**5.9 State（状态）**

**5.9.1 介绍**

状态模式允许一个对象当其状态改变时更改它的行为。对象将会对它所在的类进行更改。状态模式也被称为状态对象（Objects for States）。

**5.9.2 起源**

我们用一个例子来介绍状态模式。考虑一个代表网络连接的TCPConnection类。一个TCPConnection对象共有以下几种可能的状态：建立，倾听，关闭。当一个TCPConnection对象获取到来自其他对象的请求时，该对象基于自己所处的状态做出不同的反应。例如，Open请求的结果取决于该连接处于关闭状态还是建立状态。下图中的状态模式描述了TCPConnection是如何在每个状态展现出不同的行为的。

这个模式的核心思想是引入了一个名为TCPState的抽象类以代表网络连接的状态。TCPState类声明了一个面向不同状态的所有类的公共接口。TCPState的子类中定义了每个状态的行为函数。例如，TCPEstablished和TCPClosed实现了面向建立和关闭状态的TCPConnection行为。

TCPConnection维护了一个状态对象（一个TCPState类的子类实例）以表示TCPConnection的当前状态。TCPConnection将所有状态请求在这个状态对象中进行实现。TCPConnection使用TCPState子类的实例执行基于不同状态的网络连接操作。

一旦网络连接状态发生变化，TCPConnection对象就会改变其使用的状态对象。例如当网络连接从建立状态转换为关闭状态时，TCPConnection会使用TCPClosed实例取代TCPEstablished实例。

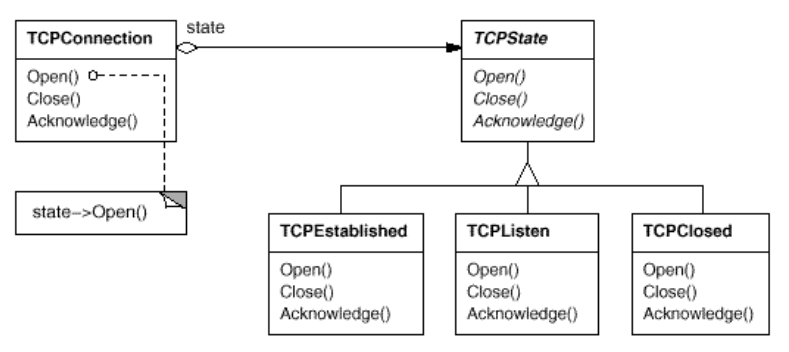


图1.1 TCPConnection实例[1]

**5.9.3 适用范围**

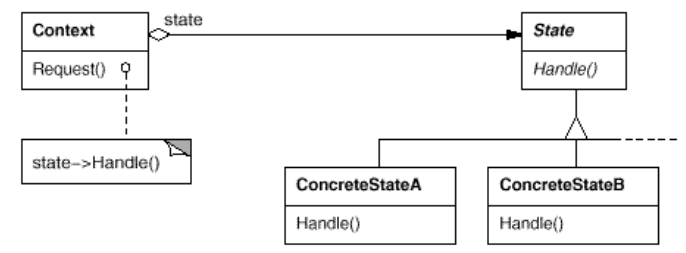
当出现以下几种情况时，可以考虑使用状态模式。

- 一个对象的行为取决于它所处的状态，并且该对象运行时会随着状态改变更改它的行为。

- 对象操作有基于对象状态的大量条件语句。状态通常使用一个或多个枚举常量表示。一般情况下，多种操作会包含相同的条件结构。状态模式可以将每一个分支条件结构封装入一个单独的类。这种实现方式使得对象的状态可以像一个对象一样脱离父对象进行独立变化。

**5.9.4 设计模式结构**

该设计模式的一般结构如下图所示。



其中，Context对应于上文实例中的TCPConnection类，定义程序接口，维护一个描述当前状态的ConcreteState子类实例。

State对应于上文实例中的TCPState类，定义了一个封装Context对象特定状态下行为函数的接口。

State类的子类对应于上文实例中的TCPEstablished，TCPListen，TCPClosed子类。每一个子类实现一组对应于Context特定状态的行为函数。

**5.9.5 效果：**

使用状态模式会产生以下的效果：

①状态模式缩小了特定状态行为的范围，对不同状态的行为进行了划分。

状态模式将与某个状态相关联的所有行为定义到一个对象中。因为所有特定状态行为函数的代码存在于一个状态子类中，新的行为函数和状态转换函数可以通过定义新的子类轻易地实现。

另外一种实现方式是使用一个变量值定义内部状态，并使用Context操作显示地检查该变量。但是如果使用这种实现方法，我们就会在Context的实现代码中散布大量看起来相似的条件或case语句。加入一个新的状态需要改变许多程序代码，这使得程序维护非常困难。

状态模式避免了这种问题，但是有可能引入其他问题，因为这种设计模式将不同状态对应的行为函数散步到多个状态子类。这种实现方法增加了类的数量，不如一个类简洁。但是如果有多种状态，这种分布实际上是有益的，因为如果不这样实现就会需要大量条件语句进行实现。

大量的条件语句就和代码量较大的函数一样不利于程序维护。这些实现会使得程序不容易读懂，因此使得它们难以维护和扩展。状态模式为包含多种状态的代码提供了一种更好的实现结构。定义状态转移的逻辑不依赖于switch语句，而是存在于状态子类中。将状态转移函数和行为函数封装到一个类中会使得对于状态的操作提升到对象状态层面。这种实现方式使得代码结构更加有层次，更加清晰。

②状态模式使得状态转移更清晰。

当一个对象仅仅基于一个内部变量值定义它的当前状态，该对象的状态转移没有显示的表达，而是仅仅表示为某些变量的赋值。为不同的状态引入不同的对象使得状态转移更加外显化。此外，状态对象可以保护Context不受不一致的内部状态的困扰，这是因为状态转移在Context的角度来说具有原子性：状态转移通过重新绑定一个而非多个变量（Context的状态对象变量）得以发生。

③状态对象可以被共享。

如果状态对象没有实例变量，也就是说状态对象表示的状态完全被定义到它们的类型中，这种情况下，程序可以共享状态对象。

**5.9.6 实现中产生的问题**

①谁定义状态转移函数？

状态模式没有指明哪个类应该定义状态转移函数。如果状态转移函数的位置是固定的，那么它们的全部内容都可以在Context中实现。但是一般情况下，状态转移函数都是由状态类的子类定义，并指定其下一个状态的。这种实现方式需要在Context中增加一个接口，使得状态对象可以显示地设定Context的当前状态。

将状态转移逻辑分散定义使得程序的维护和扩展可以通过定义新的状态类子类实现。状态转移逻辑分散定义的一个缺点是一个状态类子类需要获取至少一个其他类的信息，这在实现中引入了不同子类间的依赖关系。

②基于表的另一种实现方法。

在C++ Programming Style一书中，Cargill描述了另一种使得状态转移代码结构化的实现方式：他使用了表创建一个输入和状态转移的映射。对于每一个状态，总有一个表包含了所有可能的输入到对应的下一个状态的映射。实际上，这种实现方式将条件代码转换为了查表过程。

使用查表方法的主要优点是它们的规范性：你可以通过修改数据而不是修改代码来改变状态转移规则。这种实现方式的主要缺点如下：

- 查表通常不如调用（虚）函数高效。

- 将状态转移逻辑放到一个统一的表格格式中使得状态转移不直观，因此不易理解。

- 一般来说，为状态转移增加行为函数比较困难。查表驱动的方法捕获当前的状态和状态转移行为，但是这种方法需要为随机的状态转移情况增加函数定义。

查表驱动的状态机和状态模式的主要区别是，状态模式对特定状态的行为函数建模，而查表驱动的方法专注于定义状态转移过程。

③创造和销毁状态对象。

一个常见的，值得斟酌的实现上的权衡是（1）只在需要的时候建立状态对象，并且在使用完成后立即销毁，还是（2）提前创建状态对象，并且整个程序执行过程中都不销毁该对象。

当执行时不知道要进入什么状态，并且程序不会频繁地改变状态时，第一种选择是更好的。这种方式避免了创建不会被重新使用的对象，这在每个状态对象都会存储大量信息的条件下特别重要。第二种方式在程序频繁地改变状态时是更好的，因为你希望避免销毁一个马上要用到的状态。变量初始化的代价在程序起始阶段就完成了，代码没有变量销毁的代价。这种方式在某种情况下也不是很方便，因为程序需要维持所有可能进入状态的引用。

④使用动态继承。

改变一个特定请求的行为代码可以通过在程序执行时改变对象的类来实现，但是这种实现方式在大多数面向对象编程语言中是不可行的。对象可以通过将行为函数赋给其他对象实现动态继承。在程序执行时改变行为函数赋值的对象可以高效地改变类的继承结构。这种机制使得对象可以通过改变它们的类而改变行为。

**5.10 Strategy（策略）**

**5.10.1 简介**

策略模式定义了一组算法，对其中的每一个算法进行封装，并且使得它们拥有互换性。策略模式使得算法可以独立于用户端进行改变。

**5.10.2 起源**

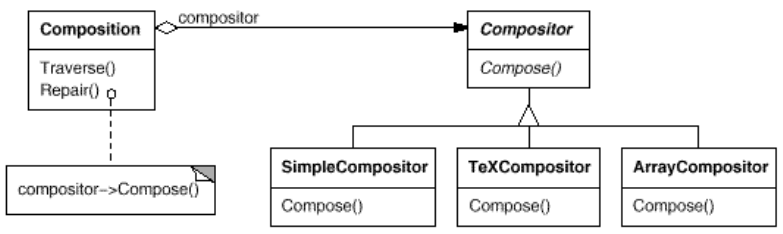
许多算法将文字流分解为多行文字。将所有这种算法包含在需要这种功能的类中并不是一种好的实现方式，这是因为：

- 这会使得需要分行的用户端段代码体量增大，难以维护，特别是当他们需要支持多种分行算法的时候。

- 不同算法适用于不同的场合。如果我们不会使用到所有的算法，我们不希望维护这些算法。

- 当分行算法是用户端的一部分时，增加新的算法或者更改现有算法十分困难。

我们可以通过定义封装不同分行算法的类来解决这些问题。一个被通过这种方式封装的算法被称为一个策略（strategy）。



假设一个Composition类被用于维护和更新一个文字阅读器的文字分行算法。分行策略不是在Composition类中实现的，相反，它们实在一个抽象类Compositor的子类中实现的。Compositor的子类包括：

- SimpleCompositor实现一个每次判断一个分行的简单策略。

- TeXCompositor实现寻找分行位置的TeX算法。这种策略对全局分行进行优化，也就是一次处理一段文字。

- ArrayCompositor选择分行位置使得每行有固定数量的字符。这种策略在将一组符号划分为行时比较有用。

Composition维护一个指向Compositor对象的引用。当Composition改变文字格式时，它将这个行为函数下放到一个Compositor对象中进行实现。Composition的用户端指定哪一个Compositor应该被使用。

**5.10.3 适用范围**

当出现以下情况时，可以使用策略模式。

- 多个互相联系的类只在它们的方法上有区别。策略模式提供了一种使用多种方法中的一种配置类的实现方式。

- 需要使用一种算法的多个变体。例如，你可能希望定义算法反应不同的时空代价权衡选择。当这些算法被组织到类的结构中时，可以使用策略模式进行实现。

- 算法数据对用户端不可见。使用策略模式来避免向用户端暴露复杂的算法内数据结构。

- 一个类定义了多种行为函数，这些行为在行为函数中被定义为多条件选择语句。除了多个条件语句的实现方式之外，将某一个条件分支移入一个策略类是一种不错的选择。

**5.10.4 设计模式结构**

策略模式的一般结构如下图所示。

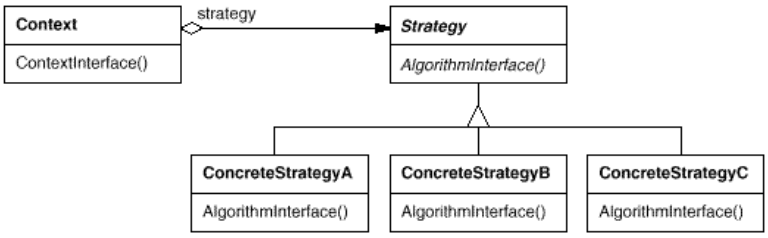


图2.2 策略模式结构[1]

其中，Strategy类对应上文实例中的Compositor，它声明了指向所有算法的公共接口。程序使用这些接口调用被某个子类定义的算法。

ConcreateStrategy子类对应上文实例中的SimpleCompositor，TeXCompositor，ArrayCompositor，它实现了使用Strategy接口的算法。

Context类对应于上文实例中的Composition类。它通过ConcreateStrategy对象进行配置，维护一个指向Strategy对象的引用，并且有可能定义一个允许Strategy访问它的数据的接口。

**5.10.5 效果**

①一组相关的算法。

Strategy类的层次结构定义了一组算法或者行为函数，提供给程序进行复用。继承关系有助于厘清算法的公共功能。

②除子类外的另一种实现方法。

继承提供了另一种支持多种算法或方法的实现方式。你可以在Context类中直接定义子类实现不同的方法。但这种实现方法将类方法与Context类进行了绑定。它将算法的实现代码与Context类的实现代码混淆在一起，使得Context类难以理解，维护，和扩展。程序员不能动态改变算法。这种方法将唯一区别是使用的算法种类的相关的类绑定在一起。将算法封装进不同的Strategy类使得程序员可以脱离上下文程序改变算法先实现代码，使得算法更容易改变，理解，扩展。

③策略模式消除了条件语句。

策略模式为条件语句提供了另一种实现方法。当不同方法函数落到同一个类中时，使用条件语句是不可避免的。将不同方法函数封装进不同的Strategy类中消除了条件语句。包含过多条件语句的代码往往需要使用策略模式进行优化。

④实现方法的选择。

策略模式可以为同一个功能提供不同的实现。用户端可以从不同策略中选择不同的算法用于不同的时空代价权衡。

⑤用户端必须了解不同的策略。

这种模式有一个潜在的弊端，即用户必须了解不同算法之间的区别来选择合适的算法。用户有可能面临实现上的问题，因此，策略模式当且仅当算法之间的区别和用户所处的情景相关时才有使用的价值。

⑥Strategy和Context类的过度结合。

Strategy类中定义的程序接口被所有的ConcreateStrategy子类共享，不管子类中的算法实现是简单的还是复杂的。因此有些ConcreateStrategy不会使用到接口传入的所有信息。有些特别简单的ConcreateStrategy可能一个变量也用不到。这意味着程序在有些时候会创建并初始化一些从不会被使用的变量。如果这个一个比较明显的问题，那么程序应该考虑缩紧Strategy和Context类之间的耦合程度。

⑦过量的对象。

策略模式会增加应用中的对象数量。有时程序员可以通过将不同算法实现为程序可共享的变量来缓解这一问题。

**5.10.6 实现中产生的问题**

策略模式在实现中可能遇到的问题如下。

①定义Strategy和Context类接口。

Strategy和Context类接口必须给Concreatestrategy用户程序所需的任何数据的访问权限，反过来也是如此。

一种方式是通过Context将数据通过参数的形式传递给Strategy方法，换言之，将数据传递给算法。这种方式使得Strategy和Context解耦合。另一方面，Context有可能会向Strategy传递不需要的数据。

另一种实现方式是将Context实例本身作为参数进行传递，Strategy实例显示地向Context实例获取参数。或者Strategy实例可以维护一个指向Context实例的引用，消除参数传递的需求。通过这两种方式的任意一种，Strategy实例都可以获取它所需要的数据。但是Context必须定义更复杂的接口，这使得Strategy和Context的耦合程度上升。

具体算法和其数据需求会决定哪一种实现方式是最优的。

②将Strategy实例定义为模板参数。

在C++中，模板可以被用于选择某个类中的策略。应用这种方法的条件包括（1）Strategy可以在编译时被选择，并且（2）它在程序运行时不会发生改变。这种情况下，将要被配置的类（即Context）会被定义为模板类，Strategy类是它的一个参数。

使用模板的情况下，我们就不需要定义一个包含指向Strategy接口的抽象类。使用Strategy作为模板参数也可以静态地绑定Strategy和它对应的Context，使得程序更加高效。

③使得Strategy对象变为可选对象。

在不包含Strategy对象仍然有意义的情况下，Context类可以被简化。Context实例在获取Strategy对象前检查它是否已经包含了Strategy对象。如果已经包含了，那么Context实例正常使用该对象。如果没有，那么Context对象执行默认方法。这种实现方式的优势是用户不必处理Strategy对象除非他们不喜欢使用默认方法。

**5.11 Visitor（访问者）**

**5.11.1 简介**

访问者模式使得程序员可以在不改变对象所属类的前提下定义一个新的方法。

**5.11.2 起源**

考虑一个将程序表示为抽象语法树的编译器。编译器程序需要对抽象语法树实行操作，实现静态语法分析。编译器程序还需要生成代码。所以它需要定义类型检查，代码优化，代码分析等的类方法。此外，我们可以使用抽象语法树进行可视化等。

这些操作中的绝大多数需要区别对待赋值语句节点和变量表达式节点。因此，程序中会有一个赋值语句类，一个变量访问类，一个算术表达式类，等等。一组节点类依赖于将要被编译的语言，但是对于已经给定的语言不会有较大的更改。

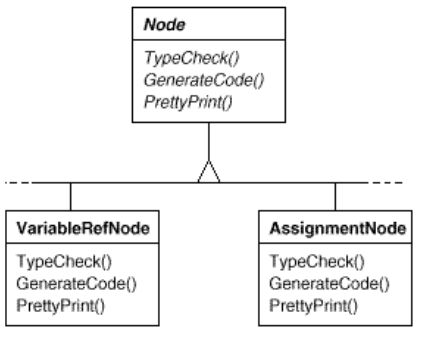


图3.1 访问者模式实例[1]

上图展示了节点类结构的一部分。这种实现的问题是将类方法分散到多种节点类中使得编译程序难以理解，难以维护，难以更改。将类型检查的代码与可视化代码或静态语法分析代码混合在一起会令程序难以理解。此外，加入一个新的类方法通常需要对所有类进行重新编译。如果可以的话，每个新运算可以被分别加入，并且节点类与运算独立是一种更好的实现方式。

我们可以通过将每个类中的相关运算打包进一个单独的访问者对象中实现上述两种需求，并将访问者对象传递给抽象语法树中正在遍历的节点。当一个节点接受了一个访问者对象，程序向访问者对象发出请求，并将该节点作为一个参数进行传递。这个访问者对象会执行相应的运算。

访问者模式定义两种类结构：一种是操作数，一种是定义运算过程的访问者。当定义一种新的运算时，我们向访问者类结构中加入一个新的子类。只要编译器接受的语法不变，我们就可以通过定义新的访问者子类加入新的功能。

**5.11.3 适用范围**

访问者模式适用于以下情形。

- 一个对象结构包含多种含有不同接口的类对象，并且程序希望对这些对象进行运算。

- 对象结构中的对象间需要进行多种不同的且不相关的运算，并且程序不希望因为这些运算破坏不同对象所属类的定义。访问者模式通过将运算定义在一个类中使得程序将相关的运算联系在一起。当对象结构被多种应用共享时，使用访问者模式将运算分配给需要它们的应用。

- 定义对象结构的类基本不变，但是程序希望经常在该结构上定义新的运算。改变对象结构类要求重新定义指向所有访问者实例的接口，这个操作代价很大。如果对象结构类经常改变，那么在那些类中定义运算或许是一种更好的方法。

**5.11.4 设计模式结构**

一般的访问者模式结构如下图所示。

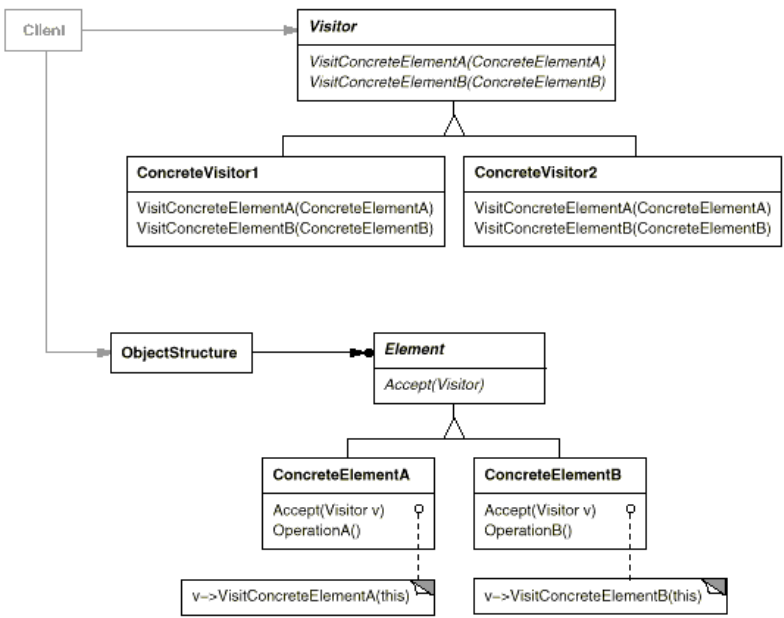


图3.2 访问者模式结构[1]

**参考文献**

1. Relationships between Design Patterns Walter Zimmer Forschungszentrum Informatik, Bereich Programmstrukturen Haid-und-Neu-Strasse 10-14, D-76131 Karlsruhe, Germany email: zimmer@fzi.de
2. The Timeless Way of Building, New York:Oxford University Press, 1979.
3. B. Anderson and P. Coad, "Patterns workshop", OOPSLA '93 Addendum to the Proceedings, 1994-January.
4. F. Buschmann, R. Meunier, H. Rohn, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley and Sons, 1996.
5. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software[M].Pearson Education. 2000
6. B. B. Mayvan and A. Rasoolzadegan, “Design pattern detection based on the graph theory,”

*Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

1. S. Hussain, J. Keung, M. K. Sohail, A. A. Khan, and M. Ilahi, “Automated framework for classification and selection of software design patterns,” *Applied Soft Computing*, vol. 75, pp. 1– 20, 2019.
2. S. Chaturvedi, A. Chaturvedi, A. Tiwari, and S. Agarwal, “Design pattern detection using machine learning techniques,” in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pp. 1–6, IEEE, 2018.
3. R. J. Erich Gamma, Richard Helm and J. Vlissides in *Design Patterns Elements of Reusable Object Oriented Software*, p. 94, Addison-Wesley, 1995.
4. Modularizing Design Patterns with Aspects: A Quantitative Study Alessandro Garcia Cludio Sant’Anna Eduardo Figueiredo Uir Kulesza Carlos Lucena Arndt von Staa