Java设计模式导读

Author : ramostear

Email : ramostear@163.com

网址:https://www.ramostear.com

开端-Java设计模式导读

> 和往常一样，本篇文章依旧采用**“3W”**顺序（即What,Why和Where）来回答软件工程中的设计模式是什么，为什么需要设计模式以及在什么地方使用设计模式这三个问题。

本篇文章时Java设计模式系列技术文章的开篇，作为导读文章，将快速地对设计模式的基本概念、模式分类和适用范围进行解读。在后续的章节中，将对每一种类别的设计模式进行详细的讲解，讲解的内容包括每种设计模式的基本原理、适用范围和实战案例剖析三个部分。

1 模式的基本概念

模式是指解决某个特定领域问题，实现既定目标的方法或思想。具体来说，模式是那些身处于某个行业的从业人员根据实际的工作经验总结出的，具有通用性的且被行业公认的解决问题的方法或流程。模式并非只在软件工程中被应用，其在日常的生产活动中被广泛地使用，如制造业，餐饮业，建筑设计、医疗卫生、教育培训以及软件工程等都有模式的身影。

2 什么是设计模式？

首先，设计模式是一种模式。在软件工程中，设计模式是一种通用的、可重复使用的用于解决既定范围内普遍发生的重复性问题的软件设计方法。使用成熟可靠的设计模式，可以提高代码复用性，节省开发时间，从而实现功能更强大、高度可维护的代码。这有助于降低软件产品的总体拥有成本，即TCO(Total Cost of Ownership)。另一方面，由于采用了统一的标准设计方法(思想或理论知识)，可以显著提升开发团队的生产效率和协作能力。

3 Java设计模式的分类

在Java编程语言中，常用的设计模式可分为三种类型：

- 建造类设计模式：主要用于定义和约束如何创建一个新的对象
- 结构类设计模式：主要用于定义如何使用多个对象组合出一个或多个复合对象
- 行为类设计模式：主要用于定义和描述对象之间的交互规则和限定对象的职责边界线

图3-1 设计模式分类

3.1 建造类设计模式

建造类共包括五（5）种基本设计模式：单例模式，工厂模式，抽象工厂模式，建造器模式和原型模式，如图3-2所示：

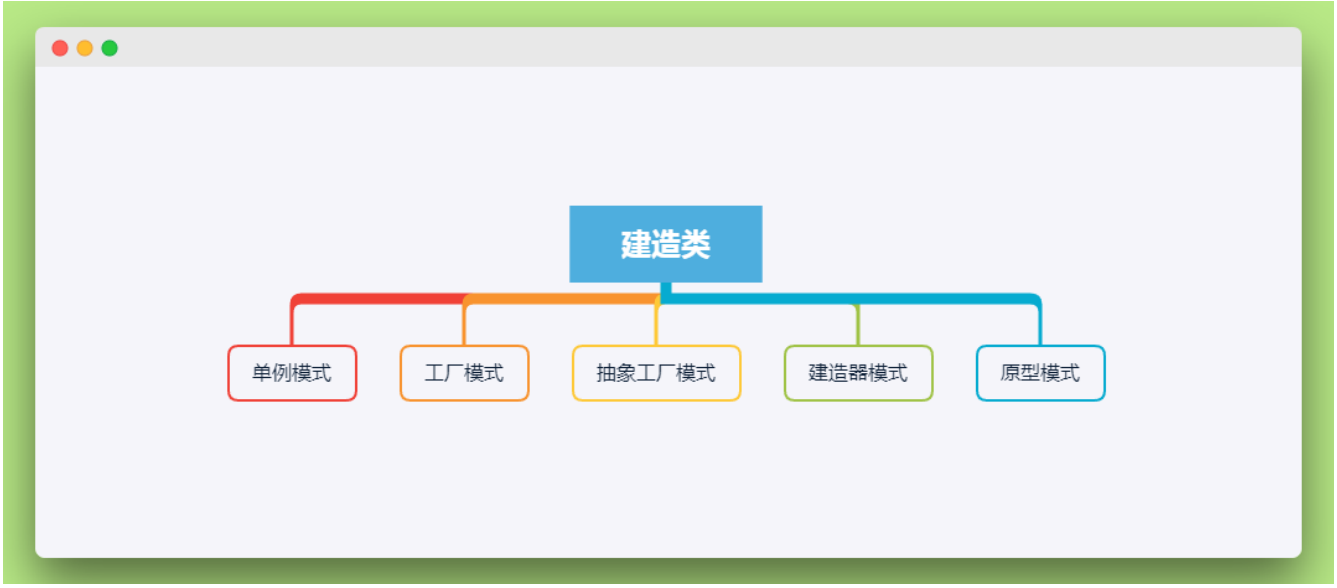

图3-2 建造类设计模式

3.2 结构类设计模式

结构类共包括八（8）种基本设计模式：适配器模式，组合模式，代理模式，享元模式，过滤器模式，桥接模式，修饰模式和外观模式，如图3-3所示：

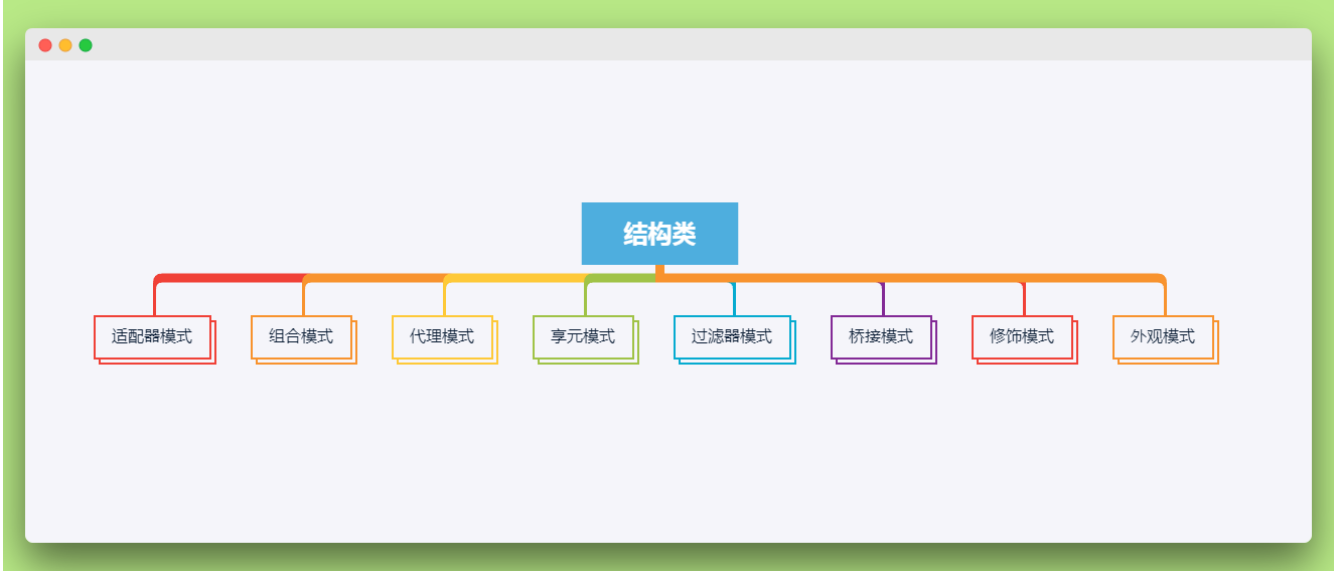

图3-3 结构类设计模式

3.3 行为类设计模式

行为类共包括十一（11）种基本设计模式：模板方法模式，解释器模式，责任链模式，观察者模式，战略模式，命令模式，状态模式，访客模式，转义模式，迭代器模式和备忘录模式，如图3-4所示：

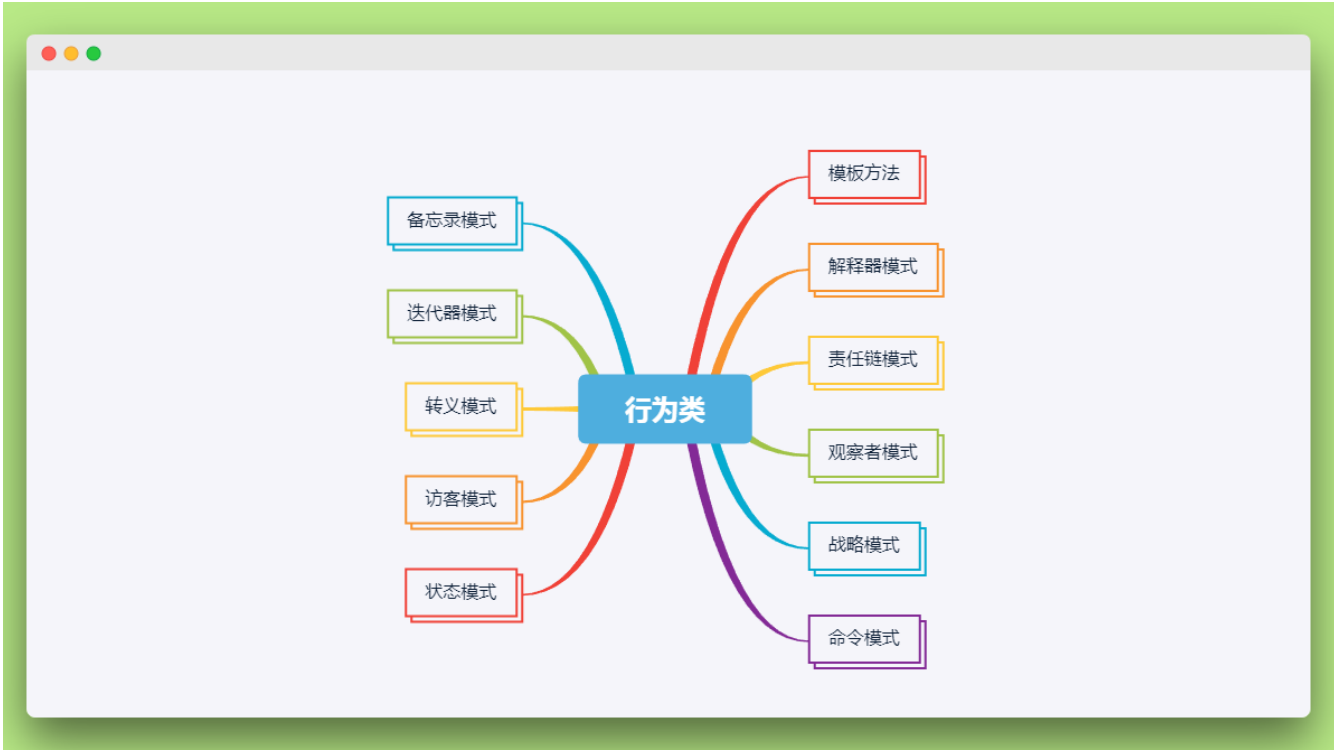

图3-4 行为类设计模式

设计模式不仅仅只有上述描述的这三大类，除此之外还有许多的设计模式。现已知的设计模式还有100多种，如DAO模式，依赖注入模式和MVC模式等。

4 快速理解设计模式

在接下来的内容中，将快速对Java中常见的24中设计模式的基本概念进行梳理，以求对各种设计模式的原理和适用范围有一个大致的认识。

4.1 建造类

建造类设计模式提供了对创建对象的基本定义和约束条件，以寻求最佳的实例化Java对象解决方案。

4.1.1 单例模式-Singleton

单例模式限制类的实例化过程，以确保在Java虚拟机（JVM）中有且只有一个类的实例化对象。单例模式是Java中最常用，也是最简单的设计模式之一。单例模式通常需具备如下的几个特征：

- 单例模式限制类的实例化，且Java虚拟机中只能存在一个该类的示例化对象
- 单例模式必须提供一个全局可用的访问入口来获取该类的实例化对象
- 单例模式常被用于日志记录，驱动程序对象设计，缓存以及线程池
- 单例模式也会被用于其他的设计模式当中，如抽象工厂模式，建造者模式，原型模式等

单例模式的Java类的内部结构如图4-1所示：

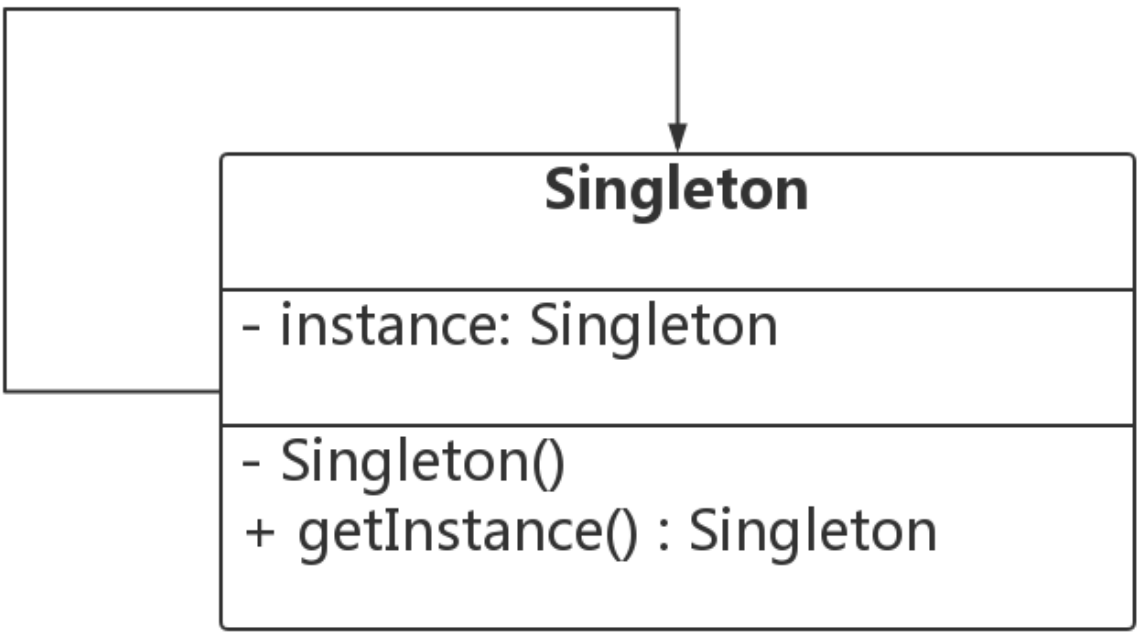

图4-1 单例模式类图

下面是单例模式的一份示例代码清单：

```java
/**
 * Implementing the Singleton Pattern in Java
*/
public class Singleton{

    // the one and only instance
    private static Singleton instance = null;
    ... // other fields

    private Singleton(){
        ... // initialization code
    }

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

}
```

4.1.2 工厂模式-Factory

在Java程序设计过程中，当一个超类(super class)具有多个子类(sub class)，且需要频繁的创建子类对象时，我们可以采用工厂模式。工厂模式的作用是将子类的实例化工作统一交由工厂类来完成，通过对输入参数的判断，工厂类自动实例化具体的子类。实现工厂模式需要满足三个条件：

- 超类（super class）：超类是一个抽象类
- 子类（sub class）: 子类需继承超类
- 工厂类（factory class）:工厂类根据输入参数实例化子类

图4-2为Java工厂模式的类图：

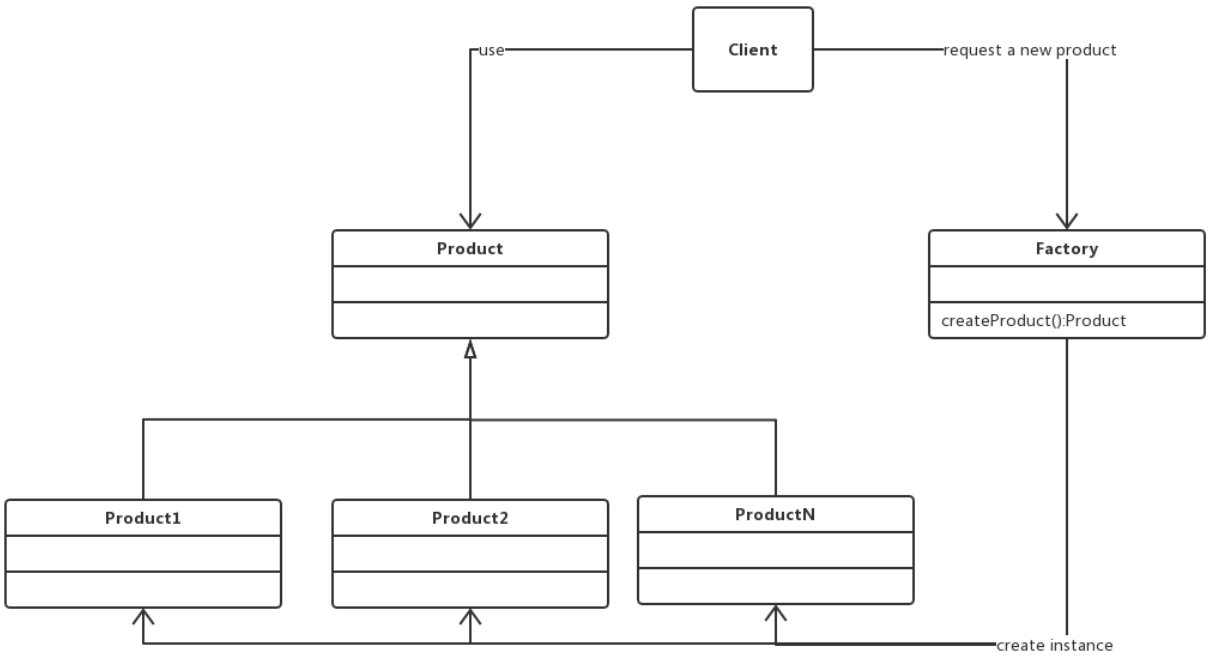

图4-2 工厂模式UML类图

下面是工厂模式的一份示例代码清单：

```java
/**
 * Implementing the Factory Pattern in Java
*/
// super class : Computer
public abstract class Computer{


  public abstract String getMemory();

  public abstract String getDisk();

  public abstract String getCpu();

}

// sub class : pc and server
public class PC extends Computer{

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram,String hdd,String cpu){
        this.ram = ram;
        this.hdd = hdd;
        this.cpu = cpu;
    }

    @Override
    public String getMemory(){
        return this.ram;
    }
    @Override
    public String getDisk(){
        return this.hdd;
    }
    @Override
    public String getCpu(){
        return this.cpu;
    }
}

public class Server extends Computer{

    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram,String hdd,String cpu){
        this.ram = ram;
        this.hdd = hdd;
        this.cpu = cpu;
    }

    @Override
    public String getMemory(){
        return this.ram;
    }
    @Override
    public String getDisk(){
        return this.hdd;
    }
    @Override
```

```
    public String getCpu(){
        return this.cpu;
    }
}

// Factory class
public class ComputerFactory {
    public static Computer create(String type,String ram,String hdd,String cpu){
        if("PC".equalsIgnoreCase(type)){
            return new PC(ram,hdd,cpu);
        }else if("Server".equalsIgnoreCase(type)){
            return new Server(ram,hdd,cpu);
        }else{
            return null;
        }
    }
}

// Test Factory class
public class Test{
    public static void main(String[] args){
        Computer pc = ComputerFactory.create("pc","8GB","512GB","2.3GHz");
        Computer server = ComputerFactory.create("server","128GB","1000GB","3.2GHz");
        ... //other operation codes
    }
}
```

4.1.3 抽象工厂模式-Abstract Factory

抽象工厂模式与工厂模式很类似，抽象工厂模式可以简单的理解为“工厂的工厂”。在工厂模式中，根据提供的输入参数返回产品类的实例化对象，这个过程需要通过if-else或者switch这样的逻辑判断语句来完成具体子类的判定。而在抽象工厂模式中，每种产品都有具体的工厂类与之对应，从而避免在编码过程中使用大量的逻辑判断代码。抽象工厂模式会根据输入的工厂类型以返回具体的工厂子类。抽象工厂类只负责实例化工厂子类，不参与商品子类的实例化工作。图4-3是抽象工厂模式的UML类图：

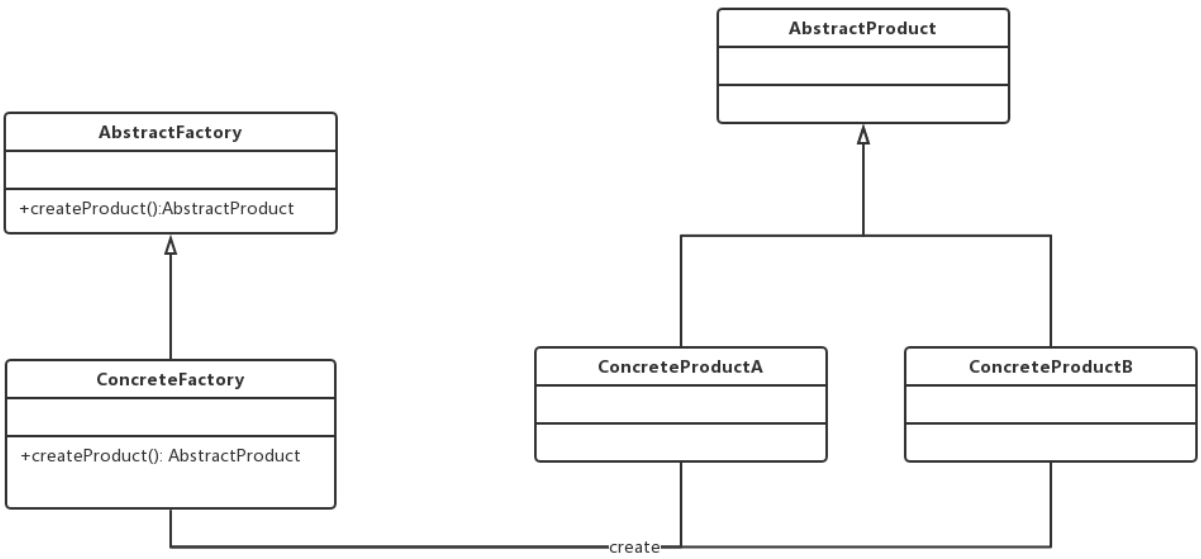

图4-3 抽象工厂模式

4.1.4 建造器模式-Builder

建造者模式通常被用于需要多个步骤创建对象的场景中。建造者模式的主要意图是将类的构建逻辑转移到类的实例化之外，当一个类有许多的属性，当在实例化该类的对象时，并不一定拥有该实例化对象的全部属性信息，便可使用建造者模式通过逐步获取实例化对象的属性信息，来完成该类的实例化过程。而工厂模式和抽象工厂模式需要在实例化时获取该类实例化对象的全部属性信息。图4-4展示了建造器模式的基本逻辑关系：

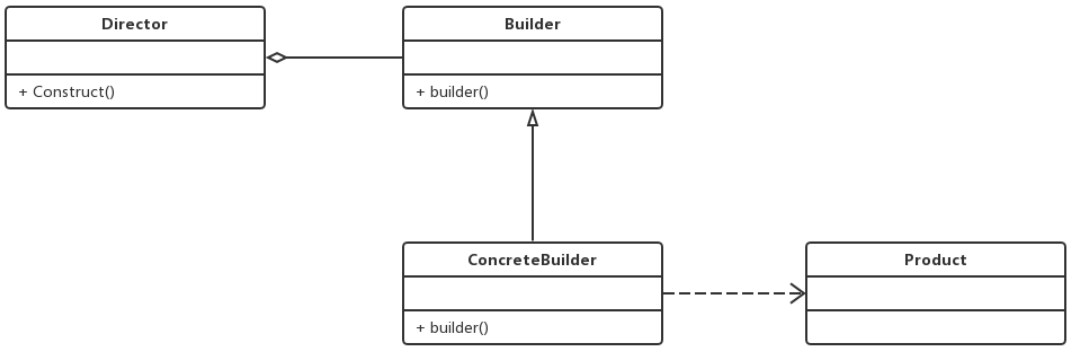

图 4-4 建造器模式UML类图

4.1.5 原型模式-Prototype

原型模式的主要作用是可以利用现有的类通过复制（克隆）的方式创建一个新的对象。当示例化一个类的对象需要耗费大量的时间和系统资源时，可是采用原型模式，将原始已存在的对象通过复制（克隆）机制创建新的对象，然后根据需要，对新对象进行修改。原型模式要求被复制的对象自身具备拷贝功能，此功能不能由外界完成。图4-5展示了原型模式的基本逻辑：

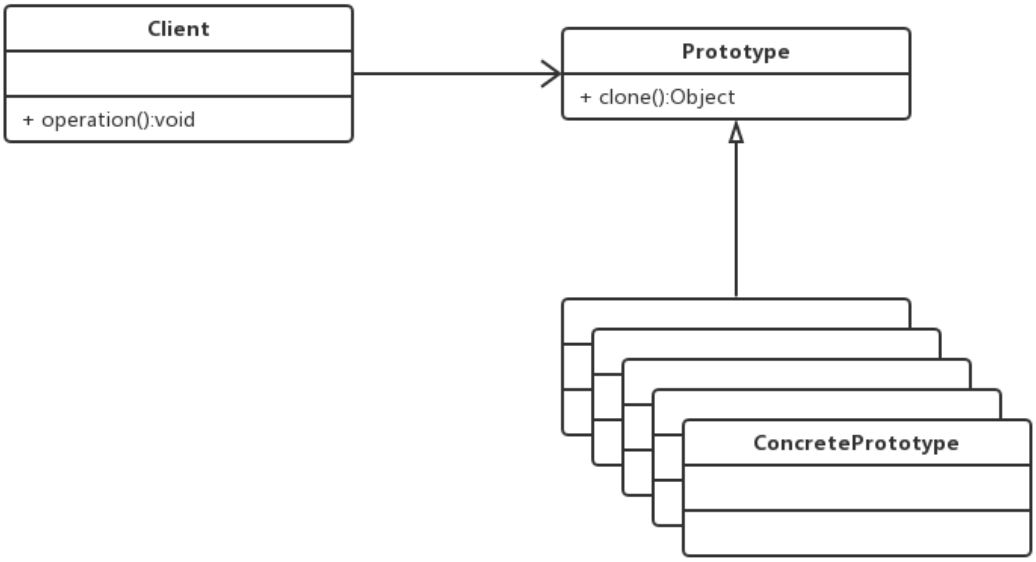

图4-5 原型模式UML类图

4.2 结构类

结构类设计模式主要解决如何通过多个小对象组合出一个大对象的问题，如使用继承和接口实现将多个类组合在一起。

4.2.1 适配器模式-Adapter

适配器模式的主要作用是使现有的多个可用接口能够在一起为客服端提供新的接口服务。在适配器模式中，负责连接不同接口的对象成为适配器。在现实生活中，我们也能够找到很多实际的案例来理解适配器的工作原理，例如常用的手机充电头，在手机和电源插座之间，手机充电头就扮演一个适配器的角色，它能够同时适配220V，200V，120V等不同的电压，最终将电转换成手机可用的5V电压为手机进行充电。图4-6展示了适配器的基本原理：

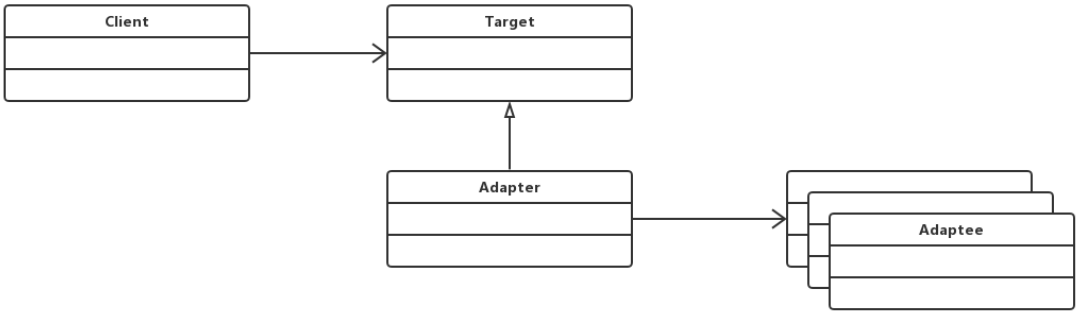

图 4-6 适配器模式UML类图

4.2.2 组合模式-Composite

组合模式的主要作用是让整体与局部之前具有相同的行为。例如我们需要绘制一个图形（正方形，三角形，圆形或其他多边形），首先需要准备一张空白的纸，然后是选择一种绘制图案的颜色，再次是确定绘制图案的大小，最后是绘制图案。不管是绘制正方形还是三角形，都需要按照这个步骤进行。在软件设计过程中，组合模式的最大意义在于保证了客户端在调用单个对象与组合对象时，在其操作流程上是保持一致的。图4-7展示了组合模式的基本原理：

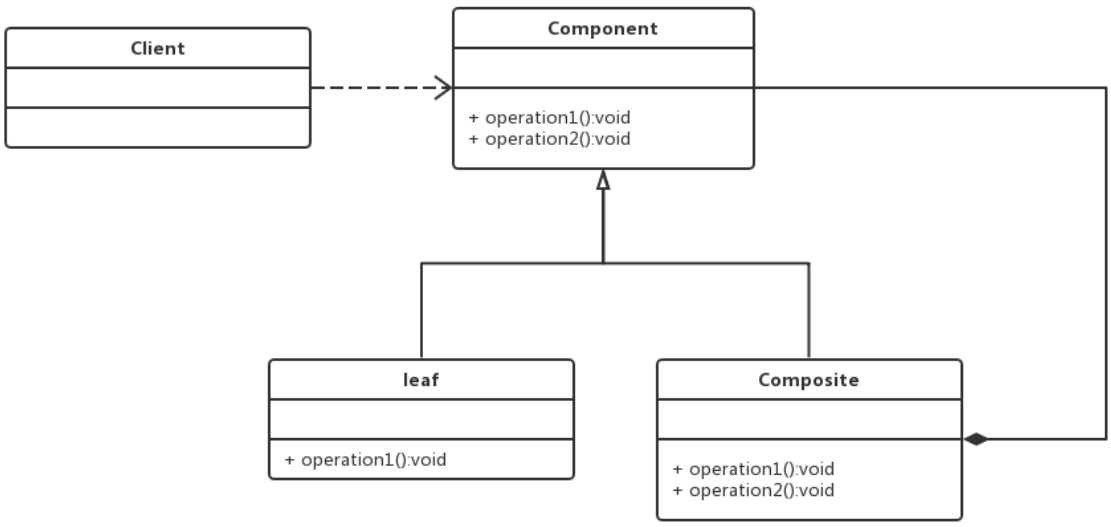

图 4-7 组合模式UML类图

4.2.3 代理模式-Proxy

代理模式的主要作用是通过提供一个代理对象或者一个占位符来控制对实际对象的访问行为。代理模式通常用于需要频繁操作一些复杂对象的地方，通过使用代理模式，可以借由代理类来操作目标对象，简化操作流程。图4-8展示了代理模式的基本原理：

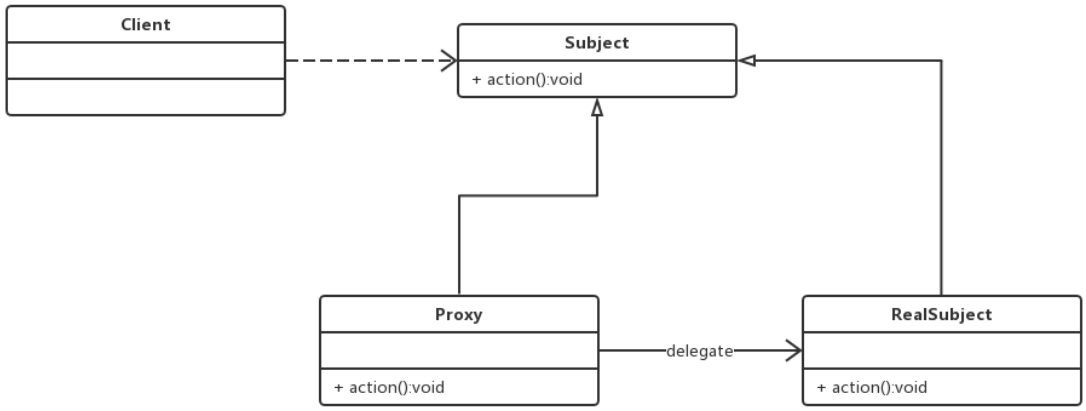

图 4-8 代理模式UML类图

4.2.4 享元模式-Flywight

享元模式的主要作用是通过共享来有效地支持大量细粒度的对象。例如当需要创建一个类的很多对象时，可以使用享元模式，通过共享对象信息来减轻内存负载。如果在软件设计过程中采用享元模式，需要考虑以下三个问题：

- 应用程序需要创建的对象数量是否很大？
- 对象的创建对内存消耗和时间消耗是否有严格的要求？
- 对象的属性是否可以分为内在属性和外在属性？对象的外在属性是否支持有客户端定义？

图4-9展示了享元模式的基本原理：

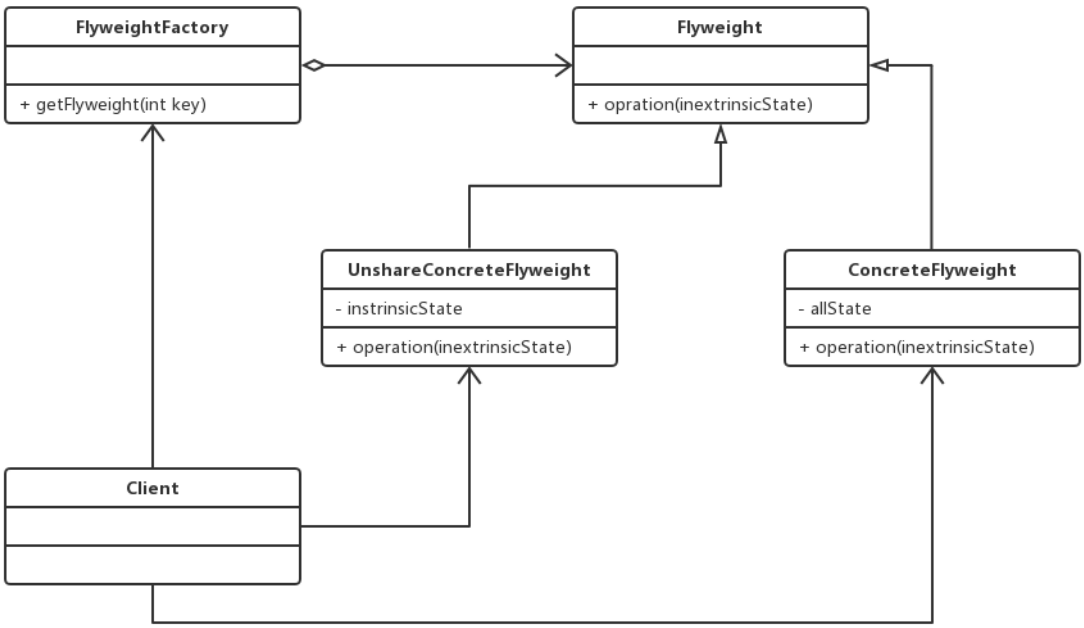

图 4-9 享元模式UML类图

4.2.5 外观模式-Facade

外观模式的主要作用是为子系统中的一组接口提供一个统一的接口，以便客户端更容易去使用子系统中的接口。简单的理解是外观模式为众多复杂接口定义了一个更高级别的接口。外观模式的目的是让接口更容易被使用，图4-10展示了外观模式的基本原理：

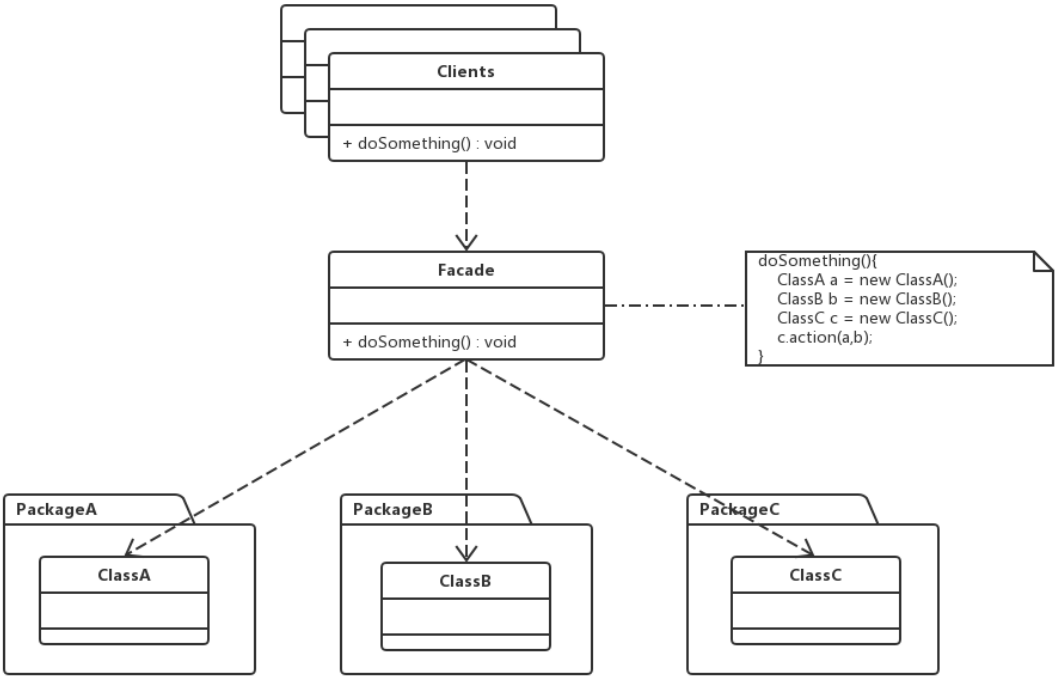

图 4-10 外观模式UML类图

4.2.6 桥接模式-Bridge

桥接模式的主要用途是将抽象类与抽象类的具体实现相分离，以实现结构上的解耦，使抽象和实现可以独立的进行变化。桥接模式的实现优先遵循组合而不是继承，当使用桥接模式时，在一定程度上可以在客户端中因此接口的内部实现。图4-11展示了桥接模式的基本原理：

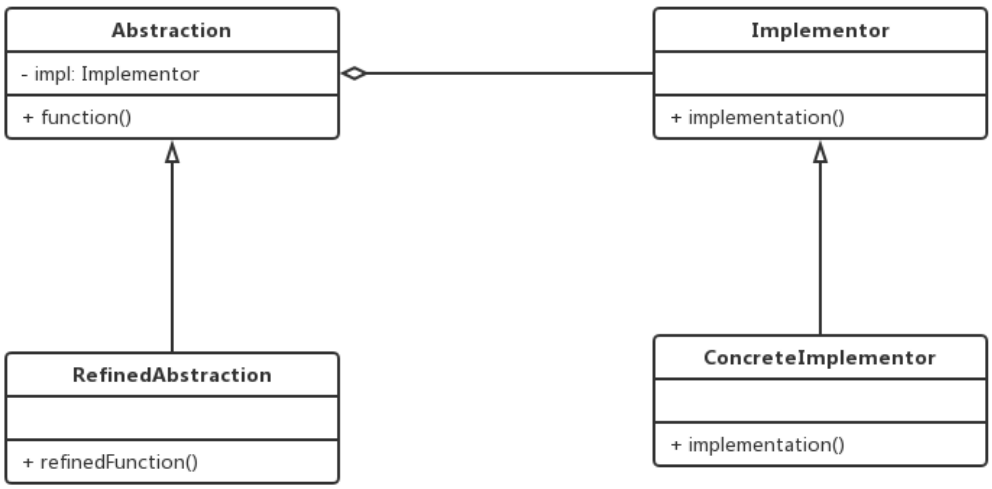

图 4-11 桥接模式UML类图

4.2.7 修饰模式-Decorator

修饰模式的主要作用是在运行时动态的组合类的行为。通常，你会添加一些新的类或者新的方法来扩展已有的代码库，然而，在某些情况下你需要在程序运行时为某个对象组合新的行为，此时你可以采用修饰模式。图4-12展示了修饰模式的基本原理：

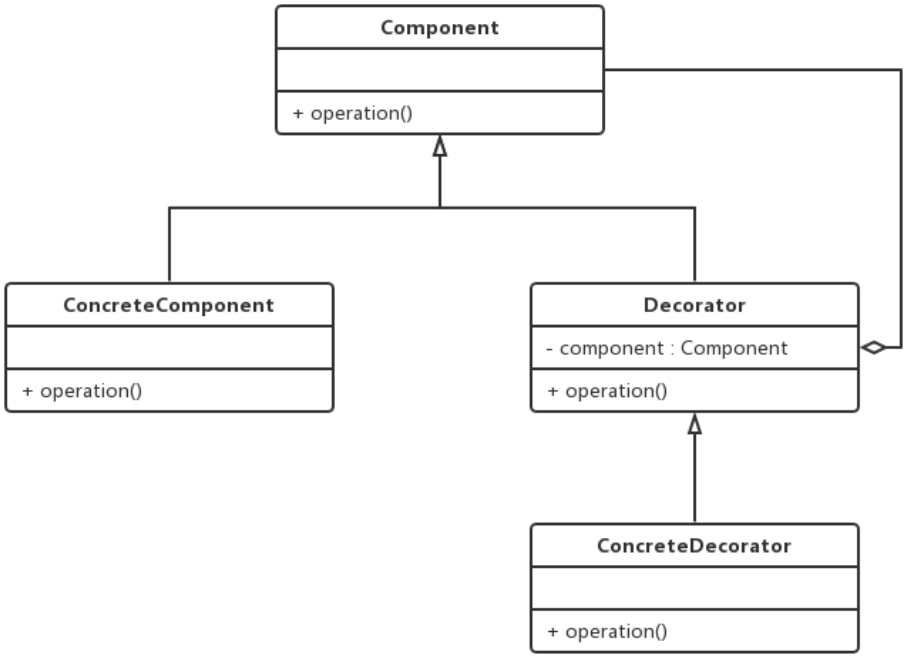

图 4-12 修饰模式UML类图

4.2.8 过滤器模式-Filter

过滤器模式是使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式将对象组合起来。图 4-13展示了过滤器模式的基本原理：

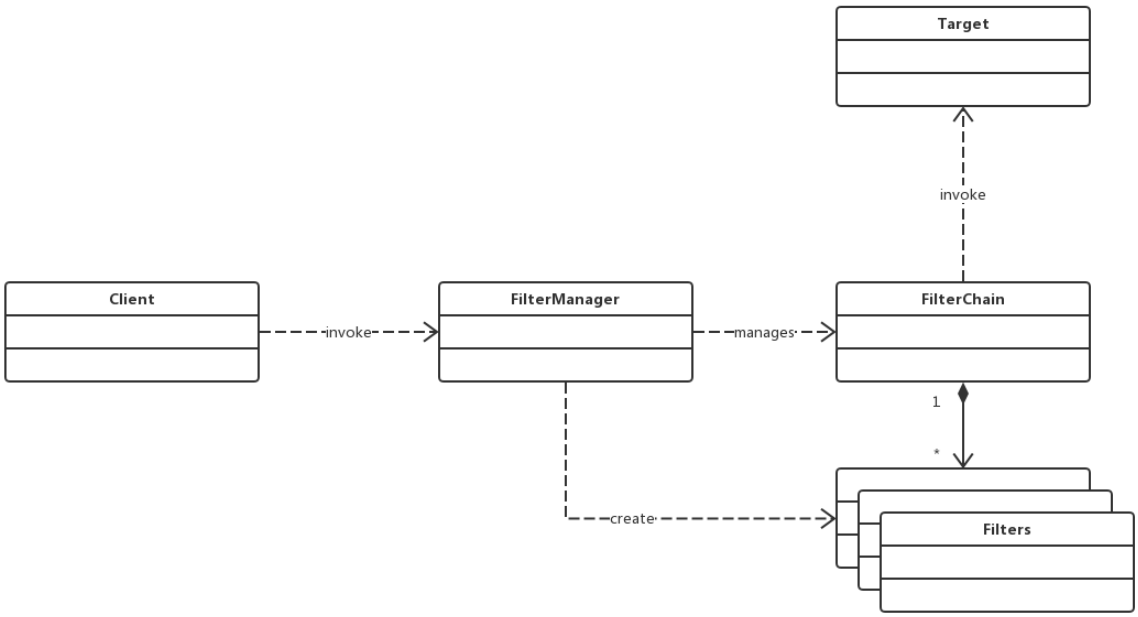

图 4-13 过滤器模式

4.3 行为类

行为类设计模式主要用于定义和描述对象之间的交互规则和职责边界，为对象之间更好的交互提供解决方案。

4.3.1 模板方法模式-Template Method

模板方法模式的主要作用是在一个方法里实现一个算法，可以将算法中的的一些步骤抽象为方法，并将这些方法的实现推迟到子类中去实现。例如建造一栋房子，我们需要设计图纸，打地基，构筑墙体，安装门窗和内部装修。我们可以设计不同的房屋样式（别墅，高楼，板房等）,不同的门窗和不同的装修材料和风格，但是其顺序不能颠倒。在这种情况下，我们可以定义一个模板方法，规定方法的执行顺序，而将方法的实现推迟到子类中完成。图4-14展示了模板方法模式的基本原理：

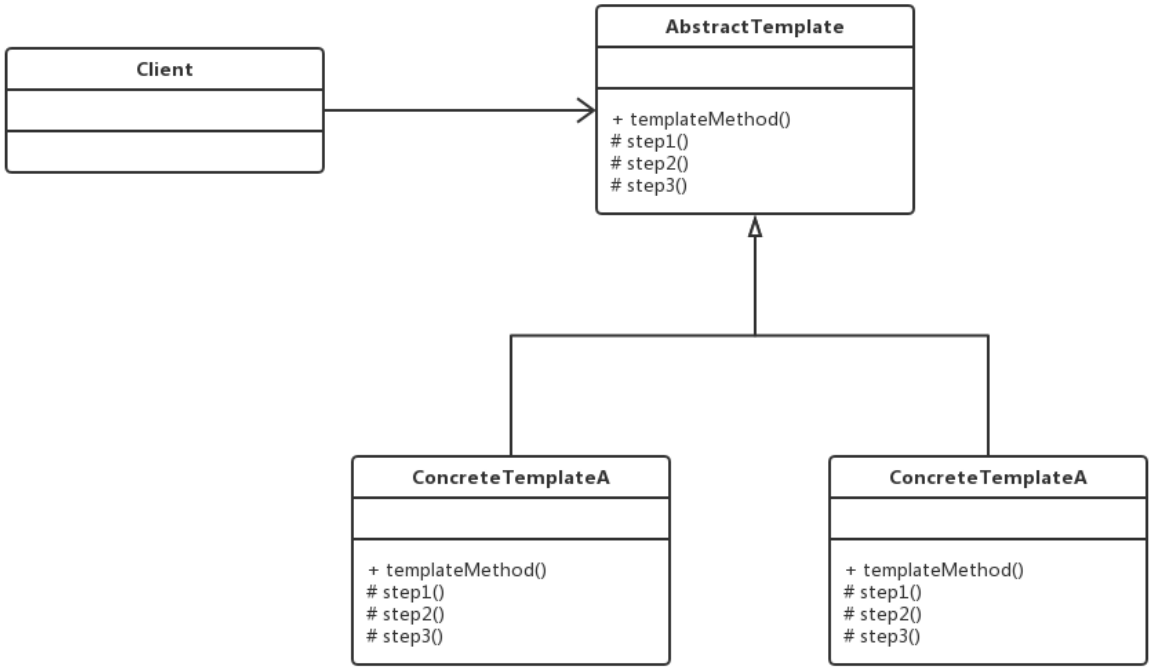

图 4-14 模板方法模式UML类图

4.3.2 解释器模式-Mediator

解释器（中介）模式的主要设计意图是定义一个中间对象，封装一组对象的交互，从而降低对象的耦合度，避免了对象间的显示引用，并可以独立地改变对象的行为。解释器（中介）模式可以在系统中的不同对象之间提供集中式的交互介质，降低系统中各组件的耦合度。图 4-15展示了解释器（中介）模式的基本原理：

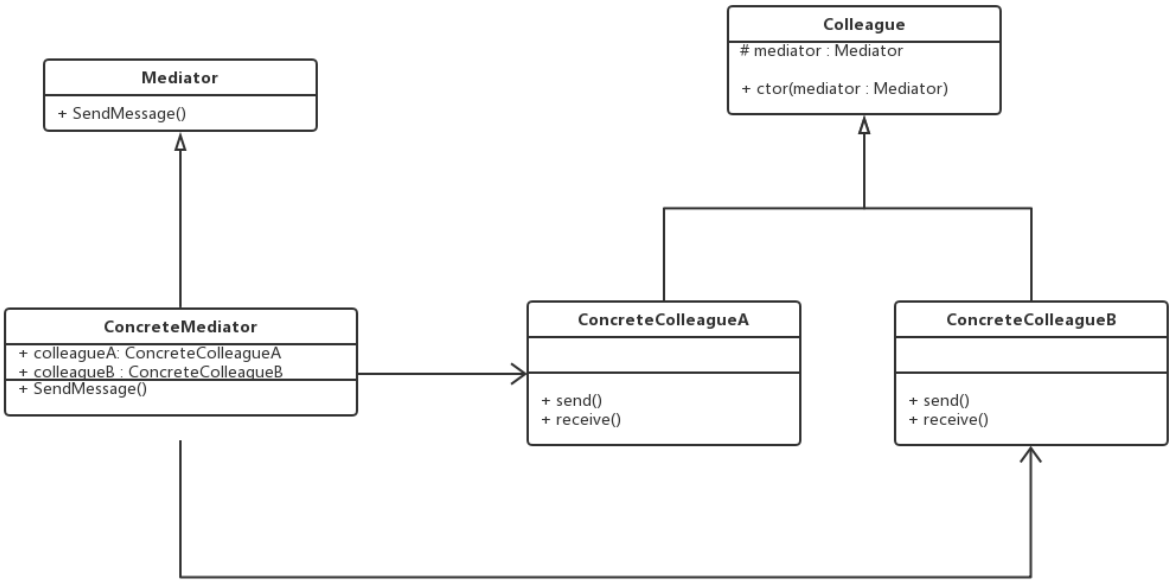

图 4-15 解释器（中介）模式UML类图

4.3.3 责任链模式-Chain of Responsibility

责任链模式主要作用是让多个对象具有对同一任务（请求）的处理机会，以解除请求发送者与接收者之间的耦合度。try-catch就是一个典型的责任链模式的应用案例。在try-catch语句中，可以同时存在多个catch语句块，每个catch语句块都是处理该特定异常的处理器。当try语句块中发生异常是，异常将被发送到第一个catch语句块进行处理，如果第一个语句块无法处理它，它将会被请求转发到链中的下一个catch语句块。如果最后一个catch语句块仍然不能处理该异常，则该异常将会被向上抛出。图4-16展示了责任链模式的基本原理：

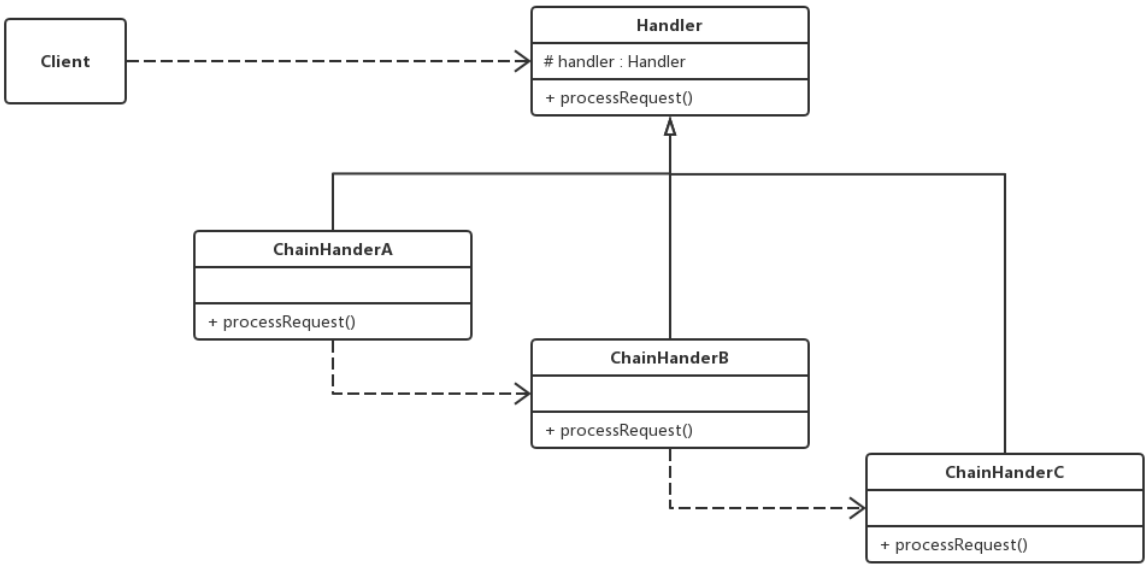

图 4-16 责任链模式UML类图

4.3.4 观察者模式-Observer

观察者模式的目的是在多个对象之间定义一对多的依赖关系，当一个对象的状态发生改变时，观察者会通知依赖它的对象，并根据新状态做出相应的反应。简单来说，如果你需要在对象状态发生改变时及时收到通知，你可以定义一个监听器，对该对象的状态进行监听，此时的监听器即为观察者（Observer），被监听对象称为主题（Subject）。Java消息服务（JMS）即采用了观察者设计模式（同时还使用了中介模式），允许应用程序订阅数据并将数据发布到其他应用程序中。图4-17展示了观察者模式的基本原理：

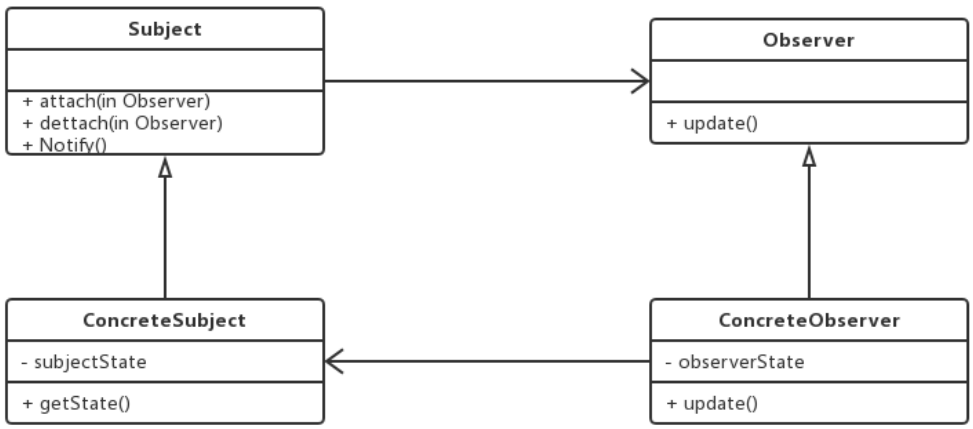

图 4-17 观察者模式UML类图

4.3.5 策略模式-Strategy

策略模式的主要目的是将可互换的方法封装在各自独立的类中，并且让每个方法都实现一个公共的操作。策略模式定义了策略的输入与输出，实现则由各个独立的类完成。策略模式可以让一组策略共存，代码互不干扰，它不仅将选择策略的逻辑从策略本身中分离出来，还帮助我们组织和简化了代码。一个典型的例子是Collections.sort()方法，采用Comparator作为方法参数，根据Comparator接口实现类的不同，对象将以不同的方式进行排序。图 4-18 展示了策略模式的基本原理：

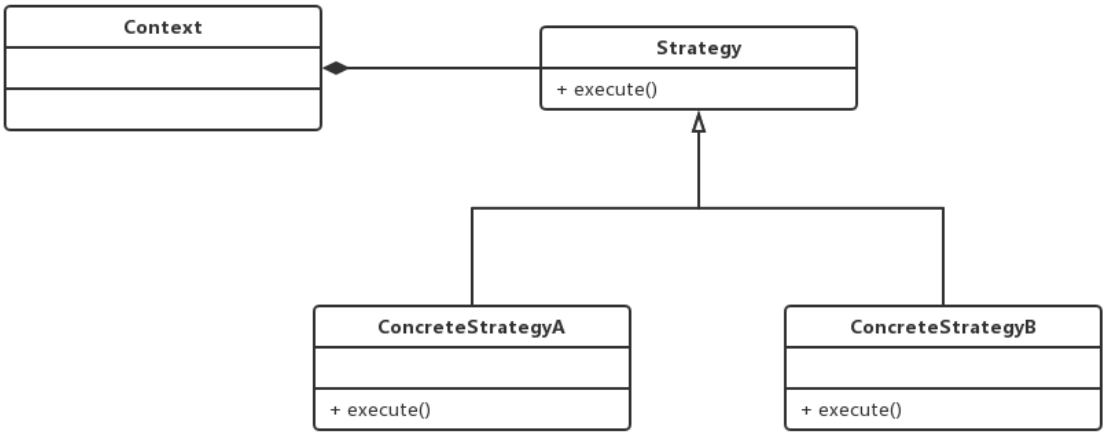

图 4-18 策略模式UML类图

4.3.6 命令模式-Command

命令模式的设计意图是将请求封装在对象的内部。直接调用是执行方法的通常做法，然而，在有些时候我们无法控制方法被执行的时机和上下文信息。在这种情况下，可以将方法封装到对象的内部，通过在对象内部存储调用方所需要的信息，就可以让客户端或者服务自由决定何时调用方法。图 4-19 展示了命令模式的基本原理：

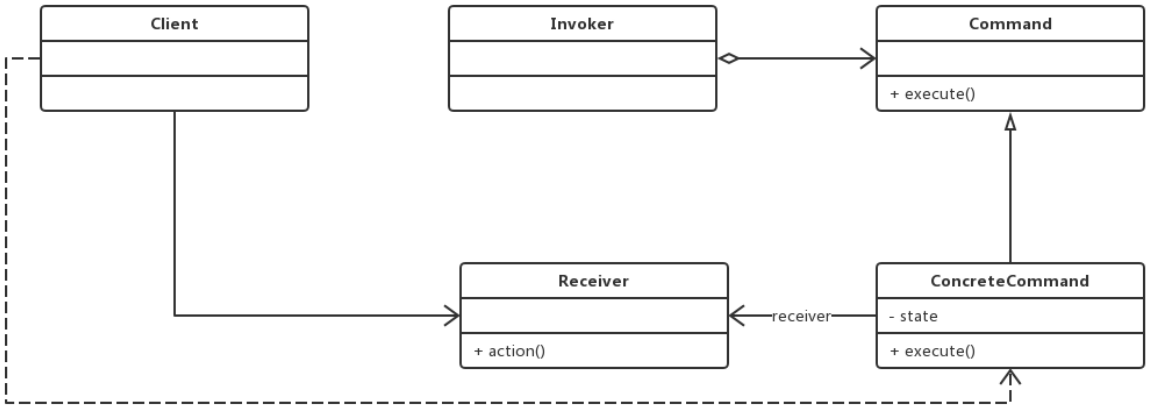

图 4-19 命令模式UML类图

4.37 状态模式-State

状态模式的设计意图是更具对象的状态改变其行为。如果我们必须根据对象的状态改变对象的行为，可以在对象中定义一个状态变量，并使用逻辑判断语句块（如if-else）根据状态执行不同的操作。图4-20展示了状态模式的基本原理：

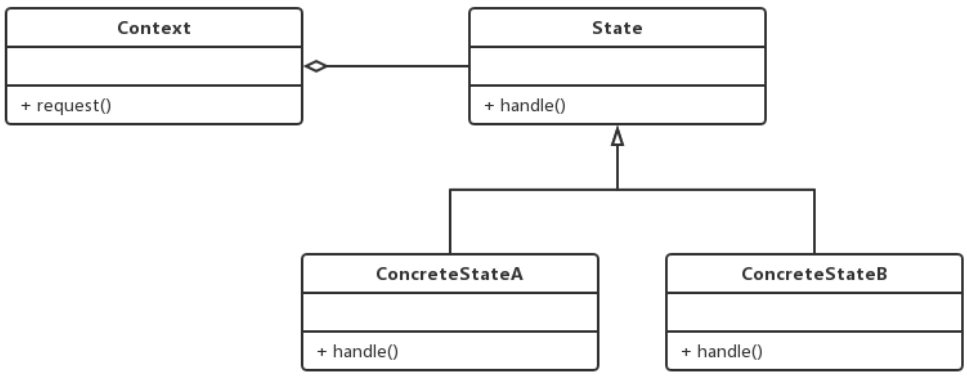

图 4-20 状态模式UML类图

4.3.8 访客模式-Visitor

访客模式的设计意图是在不改变现有类层次结构的前提下，对该层次结构进行扩展。例如在购物网站中，我们将不同的商品添加进购物车，然后支付按钮时，它会计算出需要支付的总金额数。我们可以在购物车类中完成金额的计算，也可以使用访客模式，将购物应付金额逻辑转移到新的类中。图 4-21展示了访客模式的基本原理：

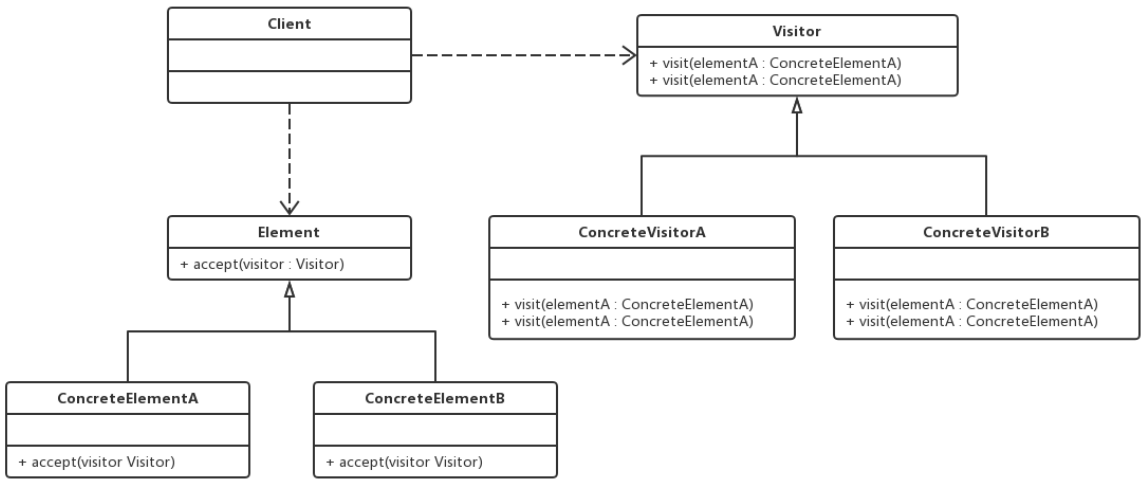

图 4-21 访客模式UML类图

4.3.9 转义（翻译）模式-Interpreter

转义（翻译）模式的设计意图是让你根据事先定义好的一系列组合规则，组合可执行的对象。实现转义（翻译）模式的一个基本步骤如下：

- 创建执行解释工作的上下文引擎
- 根据不同的表达式实现类，实现上下文中的解释工作
- 创建一个客户端，客户端从用户那里获取输入，并决定使用哪一种表达式来输出转义后的内容

图4-22展示了转义（翻译）模式的基本原理：

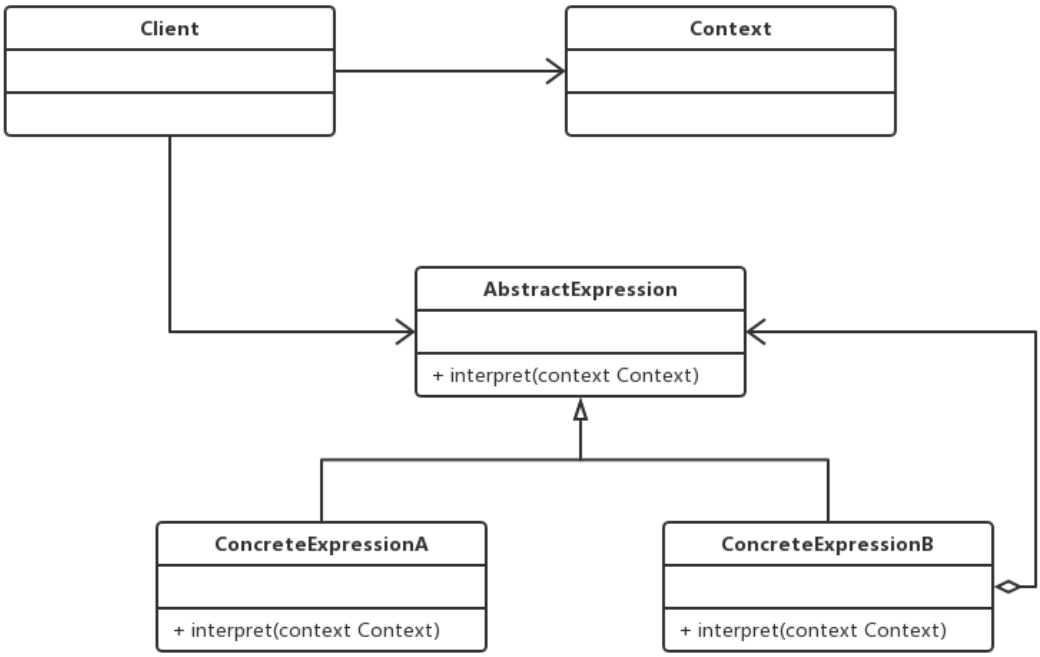

图 4-22 转义（翻译）模式UML类图

4.3.10 迭代器模式-Iterator

迭代器模式为迭代一组对象提供了一个标准的方法。迭代器模式被广泛的应用于Java Collection框架中，Iterator接口提供了遍历集合元素的方法。迭代器模式不仅仅是遍历集合，我们还可以根据不同的要求提供不同类型的迭代器。迭代器模式通过集合隐藏内部的遍历细节，客户端只需要使用对应的迭代方法即可完成元素的遍历操作。图4-23 展示了迭代器的基本原理：

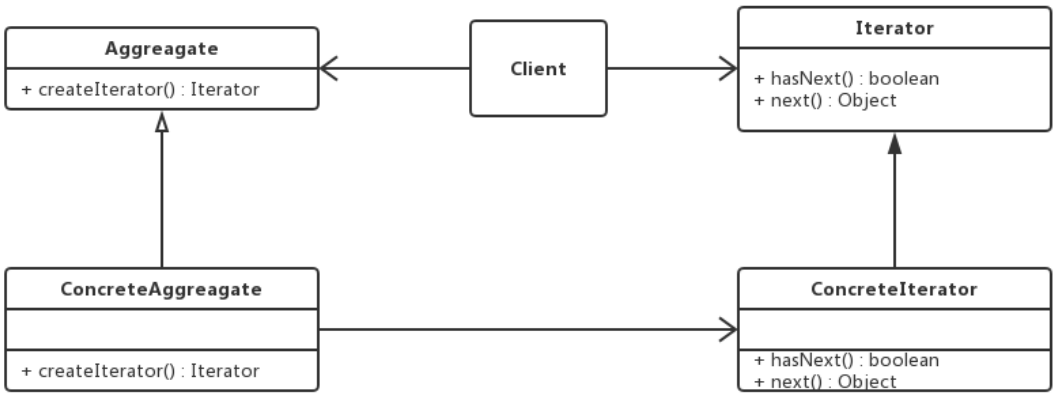

图 4-23 迭代器模式UML类图

4.3.11 备忘录模式-Memento

备忘录模式的设计意图是为对象的状态提供存储和恢复功能。备忘录模式由两个对象来实现-Originator和Caretaker。Originator需要具有保存和恢复对象状态的能力，它使用内部类来保存对象的状态。内部内则称为备忘录，因为它是对象私有的，因此外部类不能直接访问它。图4-24展示了备忘录模式的基本原理：

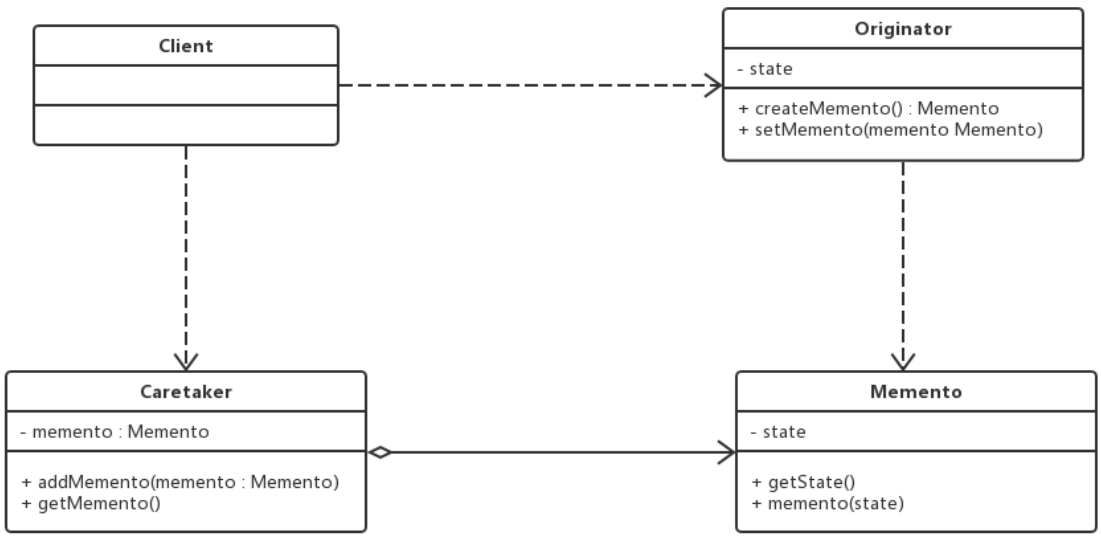

图 4-24 备忘录模式UML类图

小节

在本篇文章中，说明了模式是指解决某个特定领域问题，实现既定目标的方法或思想；设计模式是一种通用的、可重复使用的用于解决既定范围内普遍发生的重复性问题的软件设计方法。同时，对Java中常见的设计模式进行了分类，设计模式分为建造、结构和行为三种类型，并对每种类型的设计模式的基本概念和原理进行了介绍，在后续的章节中，将详细的介绍每种设计模式的原理、使用方式和适用范围，并给出相应的实战源码。