1 设计模式概述

1.1 什么是设计模式

模式有4个基本要素：

1. 模式名：描述模式的问题、解决方案和效果。
2. 问题：描述应该在何时使用模式。
3. 解决方案：描述设计的组成成分、它们之间的相互关系及各自的职责和协作方式。
4. 效果：描述模式应用的效果及使用模式应权衡的问题。

设计模式是对用来在特定场景下解决一般设计问题的类和相互通信的对象的描述。

1.2 设计模式的描述

1. 模式名和分类：模式名简洁地描述了模式的本质。
2. 意图：设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？
3. 别名：模式的其他名称。
4. 动机：用以说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景。
5. 适用性：什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？怎样识别这种情况？
6. 结构：采用基于对象建模技术的表示法对模式中的类进行图形描述。
7. 参与者：指设计模式中的类和/或对象以及它们各自的职责。
8. 协作：模式的参与者怎样协作以实现它们的职责。
9. 效果：模式怎样支持它的目标？使用模式的效果和所需做的权衡是什么？系统结构的哪些方面可以独立改变？

10. 实现： 实现模式时需要知道的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。
11. 代码示例
12. 已知应用：实际系统中发现的模式的例子。
13. 相关模式：与这个模式紧密相关的模式有哪些？其间重要的不同之处是什么？这个模式应与哪些其他模式一起使用？

1.3 设计模式的分类

		目的		
		创建型（与对象的创建有关）	**结构型（处理类或对象的组合）**	**行为型（描述类或对象怎样交互和怎样分配职责）**
范围	**类**	Factory Method	Adapter（类）	Interpreter Template Method
	对象	Abstract Factory Builder Prototype Singleton	Adapter（对象） Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

1.4 设计模式之间的关系

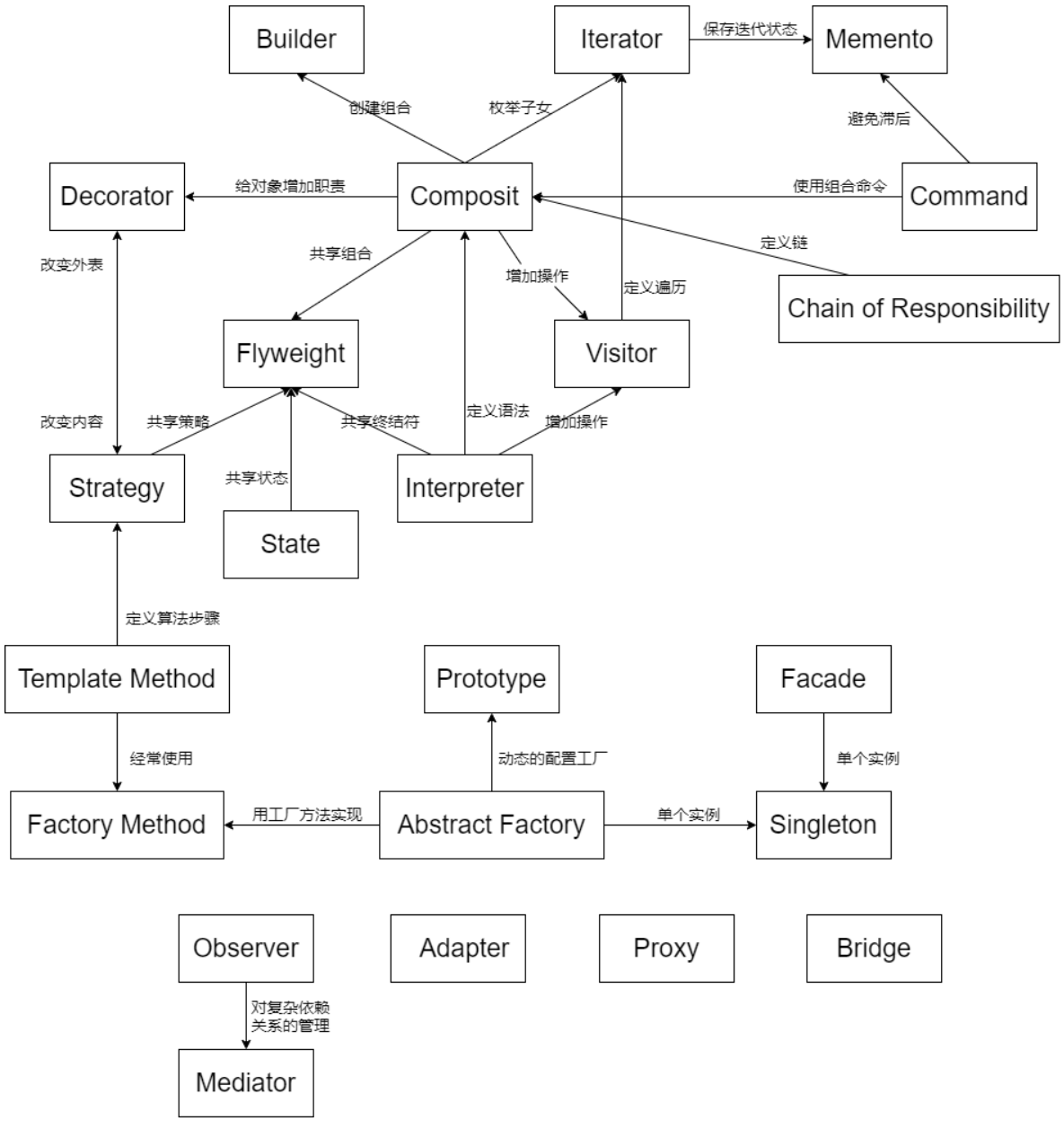

图1.1 设计模式之间的关系

1.5 设计模式怎样解决问题

1. 寻找合适的对象：设计模式帮你确定并不明显的抽象和描述这些抽象的对象。
2. 决定对象的粒度
3. 指定对象接口：设计模式通过确定接口的主要组成成分及经接口发送的数据类型来帮助你定义接口。
4. 描述对象的实现
5. 运用复用机制
6. 关联运行时和编译时的结构
7. 确保系统以特定方式变化，以满足新需求和已有需求的变化

1.6 怎样选择设计模式

- 考虑设计模式是怎样解决设计问题的
- 浏览模式的意图
- 研究模式怎样互相关联
- 研究目的相似的模式
- 检查重新设计的原因
- 考虑你的设计中哪些是可变的

下表列出了设计模式所支持的设计的可变方面：

目的	设计模式	可变的方面
创建	Abstract Factory	产品对象家族
	Builder	如何创建一个组合对象
	Factory Method	被实例化的子类
	Prototype	被实例化的类
	Singleton	一个类的唯一实例
结构	Adapter	对象的接口
	Bridge	对象的实现
	Composite	一个对象的结构和组成
	Decorator	对象的职责，不生成子类
	Facade	一个子系统的接口
	Flyweight	对象的存储开销
	Proxy	如何访问一个对象；该对象的位置
行为	Chain of Respomsobility	满足一个请求的对象
	Command	何时、怎样满足一个请求
	Interpreter	一个语言的文法及解释
	Iterator	如何遍历、访问一个聚合的各元素
	Mediator	对象间如何交互、和谁交互
	Memento	一个对象中哪些私有信息存放在该对象之外，以及在什么时候进行存储
	Observer	多个对象依赖于另外一个对象，而这些对象如何保持一致
	State	对象的状态
	Strategy	算法
	Template Method	算法中的某些步骤
	Visitor	某些可作用于一个（组）对象上的操作，但不修改这些对象的类

1.7 怎样使用设计模式

1. 大致浏览一遍模式，特别注意其适用性和效果，确定它适合你的问题。
2. 研究结构、参与者和协作，确保你理解这个模式的类和对象以及它们是怎样关联的。

3. 看代码示例。
4. 选择模式参与者的名字，使它们在应用上下文中有意义。
5. 定义类，声明它们的接口，建立它们的继承关系，定义代表数据和对象引用的实例变量。
6. 定义模式中专用于应用的操作名称。
7. 实现执行模式中责任和协作的操作。

2 面向对象设计原则

2.1 开闭原则

开闭原则（Open-Closed Principle，OCP）：类、模块、函数等设计元素应该对扩展开放，对修改关闭。

开闭原则是最基本、最重要的设计原则，其他设计原则可以看成支持开闭原则的手段。

遵循开闭原则，可以使得软件系统可复用、易于维护。

例如，图 2.1 展示了一个违反开闭原则的例子。新增图形时，需要在 `GraphicEditor` 类中增加一个方法，并且需要在 `drawShape` 方法中增加一个条件分支，无法对修改关闭。

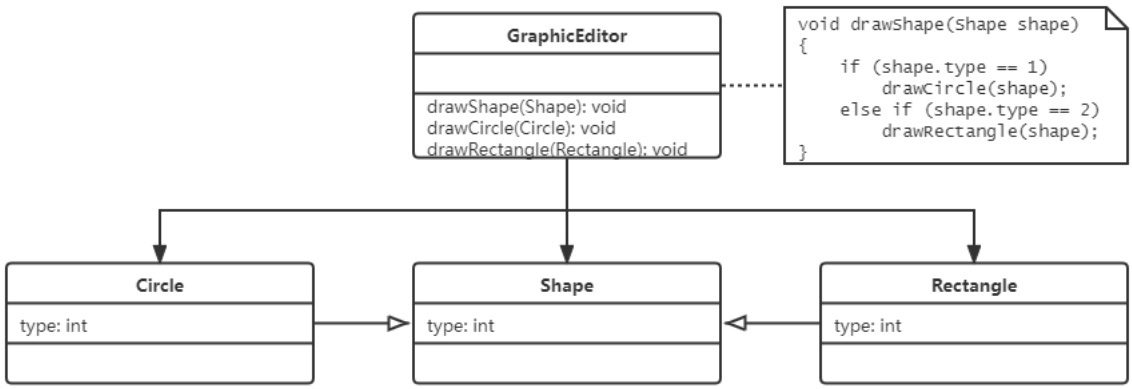

图2.1 违反开闭原则的例子

改进：将 `Shape` 类作为抽象类，并提供一个抽象的 `draw` 方法。新增图形类时，只需继承 `Shape` 类并实现 `draw` 方法即可，无需修改 `GraphicEditor` 类。

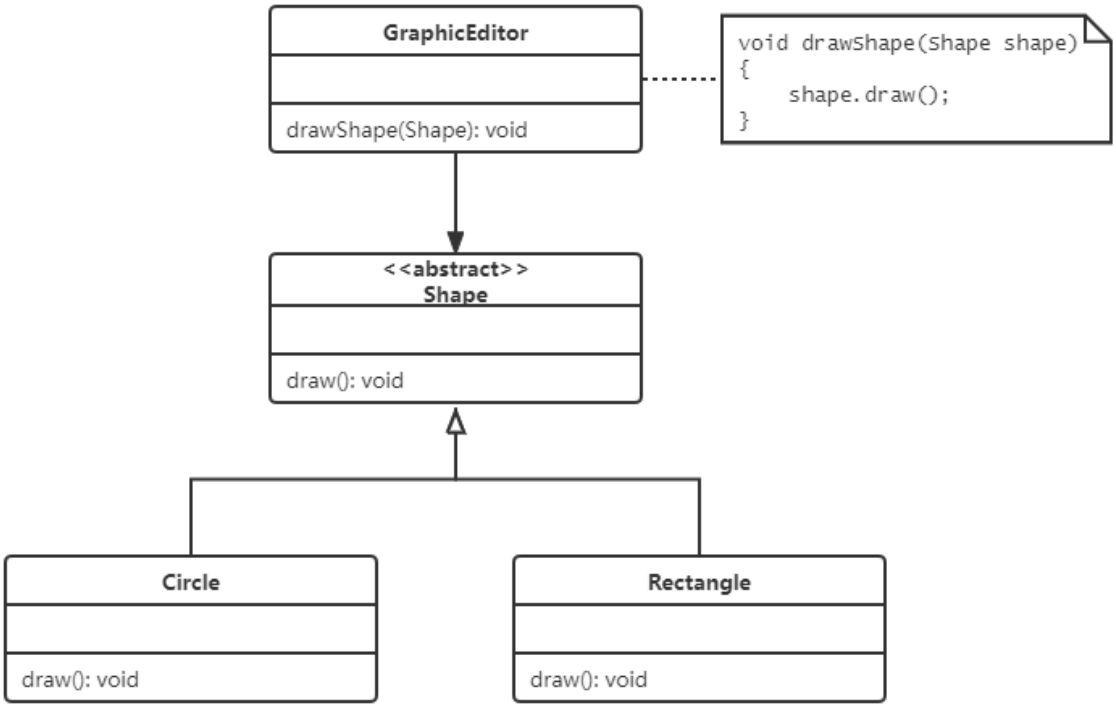

图2.2 对图2.1的改进

2.2 单一职责原则

单一职责原则（Single Responsibility Principle，SRP）：一个类应该只负责一项职责。

类的一个职责是指引起该类变化的一个原因。若一个类有一个以上的职责，就会有多种不同的原因引起该类变化，这种变化可能会影响到该类的不同种类的客户程序。例如：

```
class Modem
{
    public void dial(String pno); // 拨号
    public void hangup(); // 挂断
    public void send(char c); // 发送数据
    public char recv(); // 接收数据
}
```

`Modem`类包含了两个职责，一个是连接管理（`dial`和`hangup`），另一个是数据通信（`send`和`recv`）。这两个职责没有共同的部分，它们可能分别被不同部分的程序调用，并且可能因为不同的理由而改变，所以它违反了单一职责原则。

一种重构方案：提供两个接口，各包含一个职责，`Modem`类实现这两个接口。客户程序依赖于接口，所以从客户程序角度看是单一职责，但实现还是合在一起。

```
interface Connection
{
    public void dial(String pno); // 拨号
    public void hangup(); // 挂断
}

interface DataChannel
{
    public void send(char c); // 发送数据
    public char recv(); // 接收数据
}
```

```
12
13 class Modem implements Connection, DataChannel
14 {}
```

另一种重构方案：接口与实现都分开，职责单一。

```
1  interface Connection
2  {
3      public void dial(String pno); // 拨号
4      public void hangup(); // 挂断
5  }
6
7  class ModemConnection implements Connection // 连接管理类
8  {}
9
10 interface DataChannel
11 {
12     public void send(char c); // 发送数据
13     public char recv(); // 接收数据
14 }
15
16 class ModemDataChannel implements DataChannel // 数据通信类
17 {}
```

单一职责原则并不是极端地要求只为对象定义一个职责，而是重点强调：定义对象职责时，必须考虑职责与对象之间的关系，斟酌一个职责是否应该属于该类对象。

单一职责原则不应该教条式遵循，应该把该规则看成类或设计元素设计时的指导性原则，用它来检验设计元素设计时的合理性。

广义上，单一职责原则重点规范、针对的是接口及具有接口性质的元素，而不是单纯的类。对类来说，更多的是参考、指导、检验。

1. 若把类的非 private 访问权限下的成员视为类的接口，则这些成员应该遵循单一职责原则。
2. 若不把类的非 private 访问权限下的成员视为类的接口，而是类之外有明确定义的相关接口，则在设计实现该类时，不必教条遵循单一职责原则。

在设计一个方法时，也应该遵循单一职责原则，不要让一个方法承担过多的职责。

综上，单一职责原则可以用来规范接口、类、方法等的设计，但最直接有效的是接口，包括起到接口作用的元素。

2.3 接口隔离原则

接口隔离原则（Interface Segregation Principle，ISP）：客户程序不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口上。

接口隔离原则包含了 2 层意思：

1. 接口的设计原则：接口的设计应该遵循最小接口原则，不要把客户程序不使用的方法塞进同一个接口里。如果接口中的方法没有被用到，就说明该接口过胖，应该将其分割成几个功能专一的接口。
2. 接口的依赖（扩展）原则：如果一个接口 Ia 扩展了另一个接口 Ib，那么扩展后的接口 Ia 也应该遵循上述原则，不应该包含用户不使用的方法。如果 Ia 包含用户不使用的方法，说明接口 Ia 被 Ib 污染了，应该重新设计它们的关系。

若某客户程序被迫依赖其不使用的接口，当其他客户程序的需求变更导致接口发生改变时，该客户程序也不得不跟着改变。

示例：门具有上锁、开锁功能，也可以在门上安装报警器使其具有报警功能。用户可以选择普通的门，也可以选择具有报警功能的门。

方案 1：接口`Door`定义所有方法，`CommonDoor`必须实现`alarm`，违反接口隔离原则。

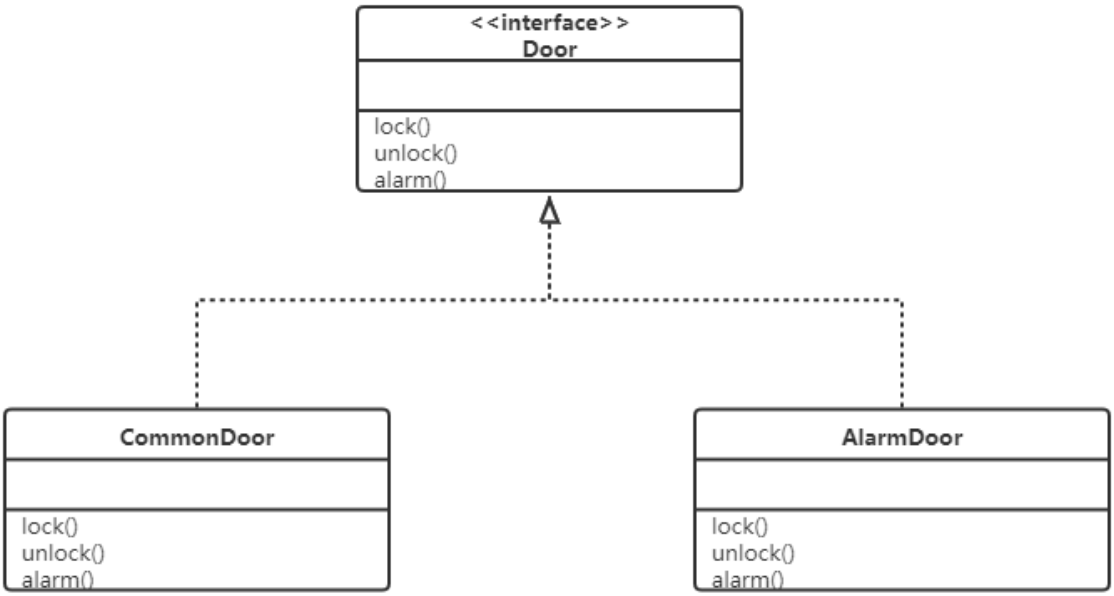

图2.3 接口隔离原则：方案1

方案 2：将`alarm`方法分离出去，单独作为一个接口，遵循接口隔离原则，但存在冗余实现现象。

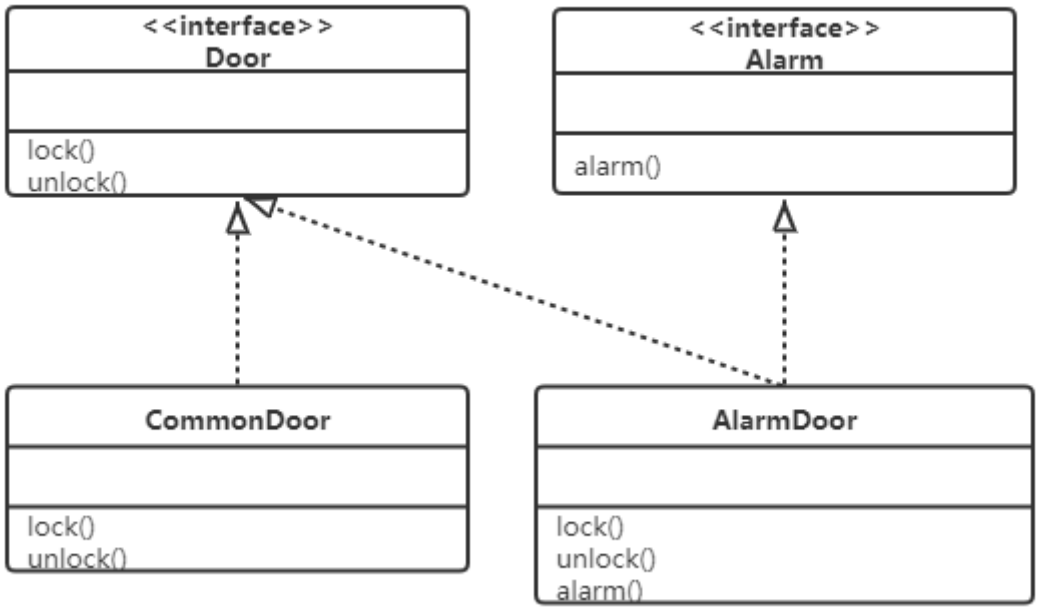

图2.4 接口隔离原则：方案2

方案 3：让`AlarmDoor`继承`CommonDoor`，解决了冗余实现问题，但`AlarmDoor`与`CommonDoor`关系密切。

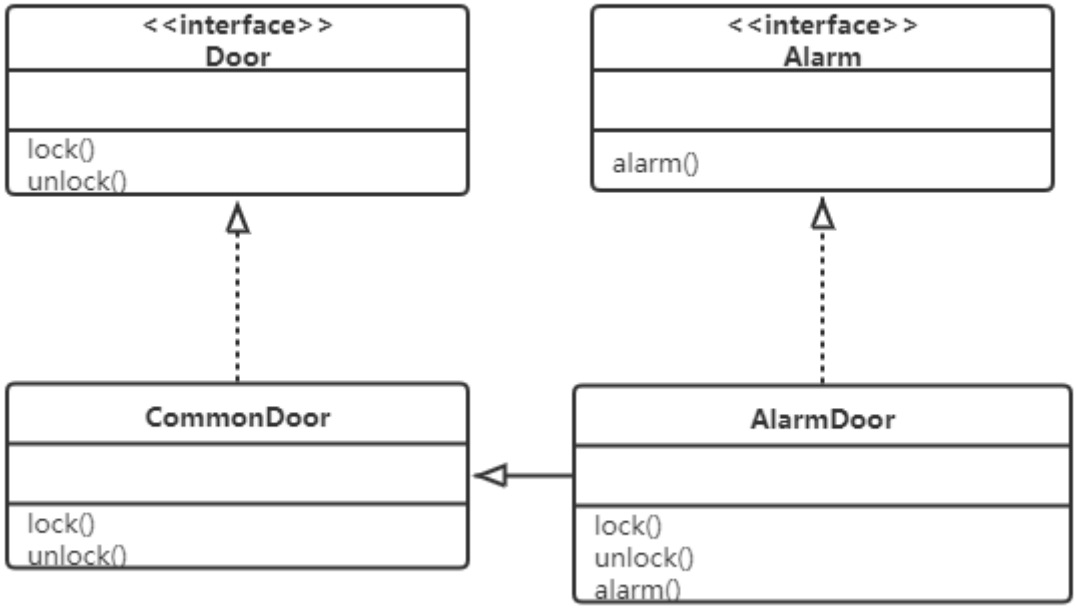

图2.5 接口隔离原则：方案3

方案 4：把 `AlarmDoor` 与 `CommonDoor` 之间的继承关系改为聚合或关联关系。

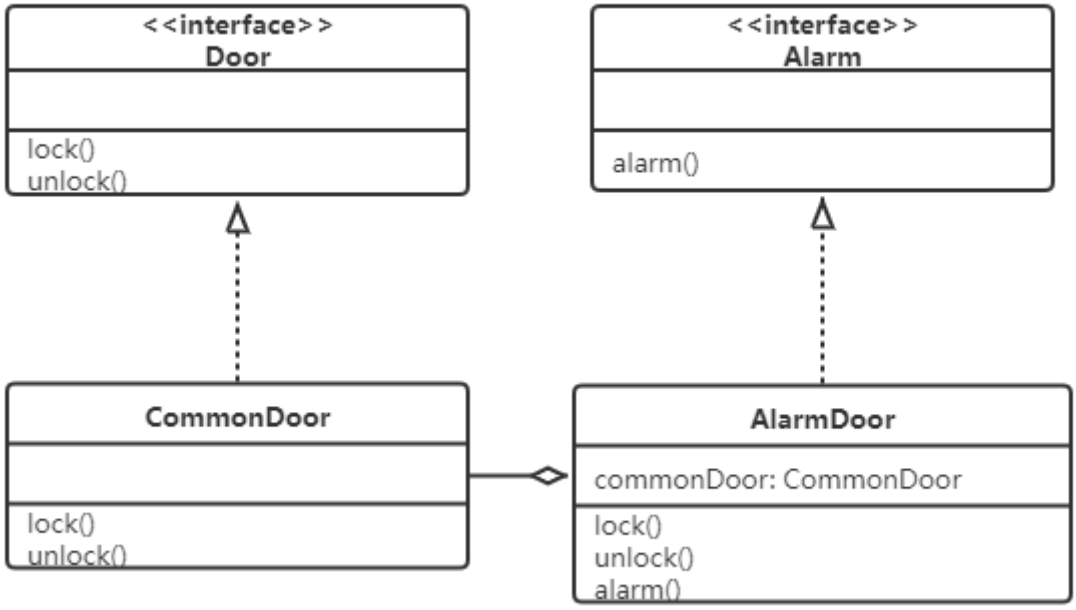

图2.6 接口隔离原则：方案4

2.4 依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）：

1. 高层模块不应该依赖于低层模块，二者都应该依赖于抽象。
2. 抽象不应该依赖于细节，细节应该依赖于抽象。

依赖倒置原则的中心思想是面向接口编程。相对于细节的多变性，抽象的东西要稳定得多，以抽象为基础搭建的架构比以细节为基础的架构要稳定得多。在 Java 中，抽象指的是接口或抽象类，细节就是具体的实现类。

使用接口或抽象类的目的是制定规范，而不涉及任何具体操作，把展现细节的任务交给实现类去完成。

如果高层模块过分依赖低层模块，一方面，一旦低层模块需要替换或修改，高层模块将受到影响；另一方面，高层模块难以复用。

解决方案：在高层模块与低层模块之间引入一个抽象接口层，如图 2.7 所示。

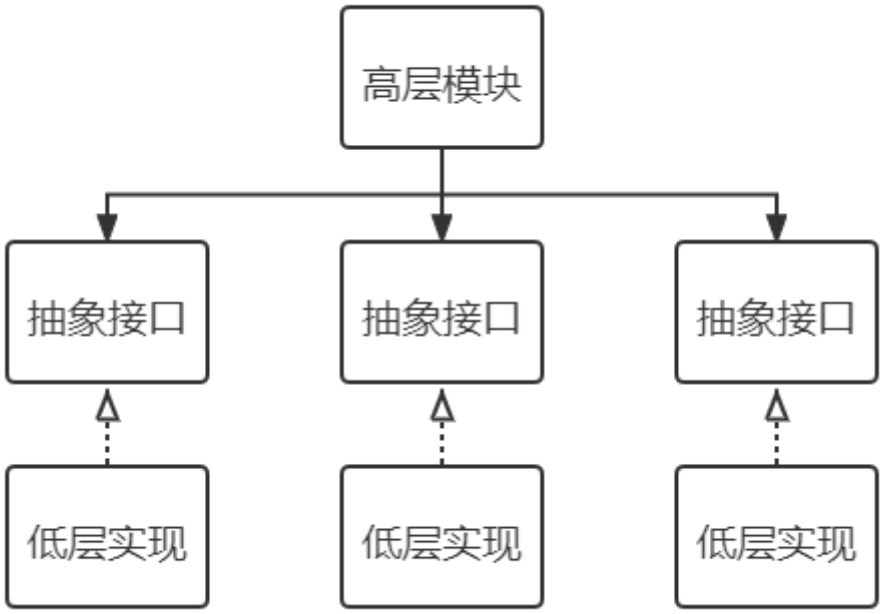

图2.7 依赖倒置原则

效果：高层模块不直接依赖低层模块，而是依赖抽象接口层。抽象接口也不依赖低层模块的实现细节，而是低层模块依赖（继承或实现）抽象接口。类与类之间都通过抽象接口层来建立关系。

例如：一个 Copy 模块，把来自 Keyboard 的输入复制到 Print。如果在 Copy 模块中直接使用 Keyboard 和 Print，当输入输出设备改变时，Copy 模块很难被重用。可以定义输入和输出设备的抽象接口，Copy 模块依赖于接口，如图 2.8 所示。

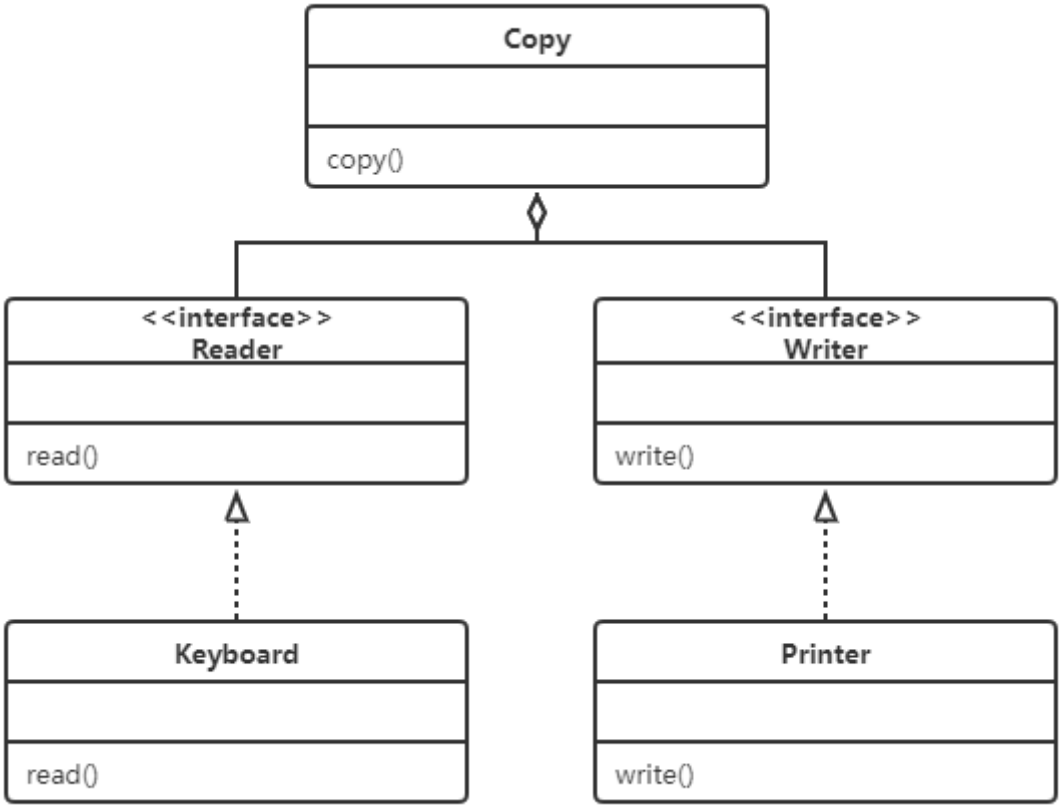

图2.8 符合依赖倒置原则的例子

Java 依赖注入的 3 种方式：

```
interface ITV
{
    public void play();
}

class TV implements ITV
{
    public void play()
    {
        System.out.println("打开");
    }
}

public class Test
{
    public static void main(String[] args)
    {
        ITV tv = new TV();

        // 1.通过接口注入依赖
        OpenAndClose1 obj1 = new OpenAndClose1();
        obj1.open(tv);

        // 2.通过构造方法注入依赖
        OpenAndClose2 obj2 = new OpenAndClose2(tv);
        obj2.open();

        // 3.通过setter方法注入依赖
        OpenAndClose3 obj3 = new OpenAndClose3();
```

```
        obj3.setTV(tv);
        obj3.open();
    }
}

// 1.通过接口注入依赖
interface IOpenAndClose1
{
    public void open(ITV tv);
}

class OpenAndClose1 implements IOpenAndClose1
{
    public void open(ITV tv)
    {
        tv.play();
    }
}

// 2.通过构造方法注入依赖
interface IOpenAndClose2
{
    public void open();
}

class OpenAndClose2 implements IOpenAndClose2
{
    private ITV tv;

    public OpenAndClodse(ITV tv)
    {
        this.tv = tv
    }

    public void open()
    {
        this.tv.play();
    }
}

// 3.通过setter方法注入依赖
interface IOpenAndClose3
{
    public void open();
}

class OpenAndClose3 implements IOpenAndClose3
{
    private ITV tv;

    public void setTV(ITV tv)
    {
        this.tv = tv;
    }

    public void open()
    {
        this.tv.play();
```

```
88      }
89  }
```

注意事项：

1. 低层模块尽量都要有抽象类或接口，或者两者都有，避免从具体类派生。
2. 变量不应该持有一个指向具体类的指针或引用，变量的声明类型尽量是抽象类或接口，在变量和实际对象之间引入一个缓冲层，利于程序扩展和优化。
3. 设计接口而非设计实现。
4. 利用抽象类、接口隔断依赖传递。如图 2.9 所示。

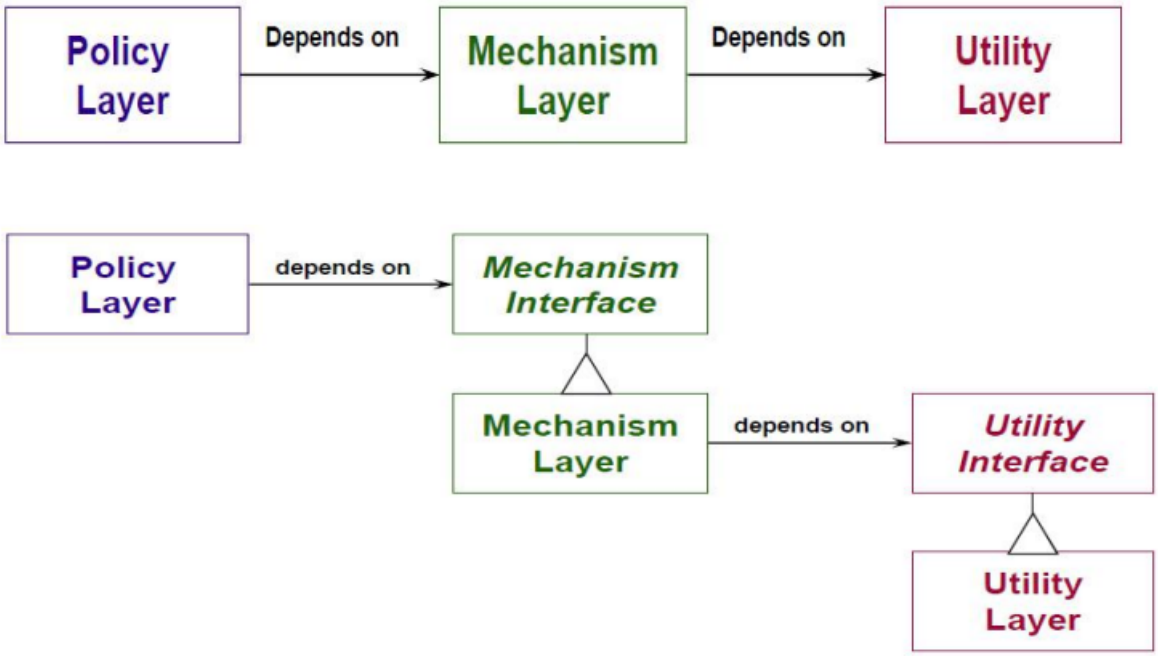

图2.9 利用接口隔断依赖传递

2.5 里氏替换原则

里氏替换原则（Liskov Substitution Principle，LSP）：所有引用基类的地方必须能透明地使用其派生类的对象。

满足里氏替换原则的条件：

1. 不应该在代码中出现对派生类类型进行判断的 if 语句。
2. 派生类应当可以替换基类并出现在基类能够出现的任何地方。换句话说，如果把代码中使用基类的地方用派生类代替，代码还能正常工作。

如果一个派生类的对象可能会在基类出现的地方出现运行错误，则该派生类不应该从该基类继承，应该重新设计它们的关系。

里氏替换原则保证了类的扩展不会给已有的系统引入新的错误。

如果基类和派生类之间的关系违反了里氏替换原则，有两种方法进行重构：

1. 创建一个新的抽象类，作为这两个类的基类，将这两个类的共同行为移动到抽象类中。
2. 将这两个类的继承关系改为关联、依赖、聚合或组合关系。

例如，鲸鱼属于哺乳动物，而不是鱼类，鱼类的很多特性是鲸鱼所不具备的，因此鲸鱼类不应该继承鱼类。鲸鱼和鱼的共性是，它们都可以游泳，可以将它们的共同行为抽象成接口，如图 2.10 所示。

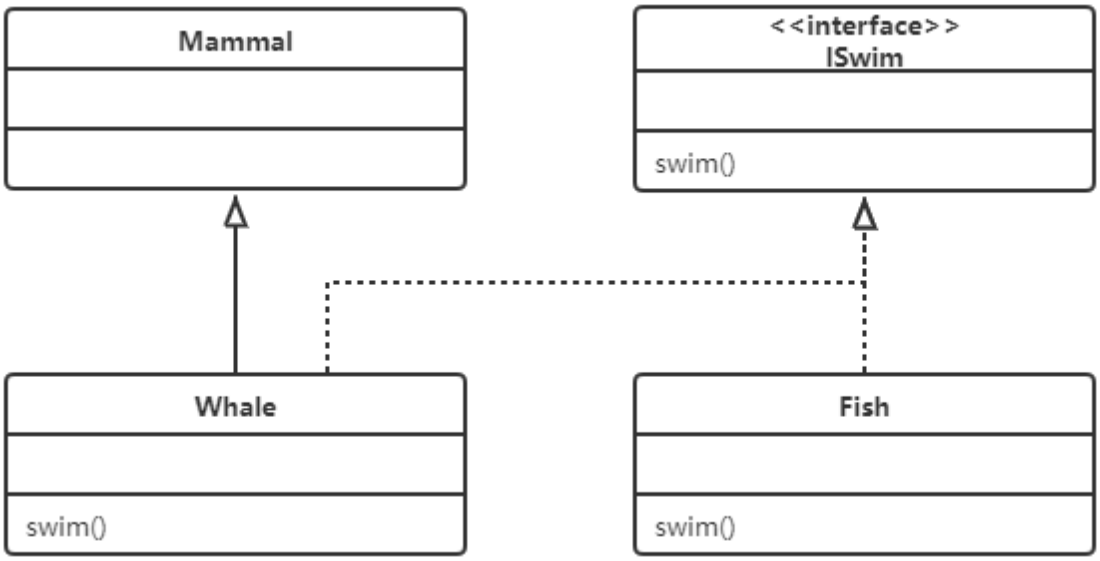

图2.10 鲸鱼类和鱼类

2.6 迪米特原则

迪米特原则（Law of Demeter，LoD）：一个对象应该对其他对象保持最少的了解。

一个软件实体应当尽可能少地与其他实体发生相互作用。每一个软件单元对其他的软件单元只需要最少的知识，而且局限于那些与本单元密切相关的软件单元。

迪米特原则又叫最少知道原则，即一个类对自己依赖的类知道得越少越好。也就是说，被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部，对外尽量减少暴露。

迪米特原则还有一个更简单的定义：只与直接的朋友通信。“直接的朋友”包括：

1. 当前对象本身。
2. 以参数形式传入到当前对象方法中的对象。
3. 当前对象的成员变量直接引用的对象。
4. 类的成员函数返回的对象，相当于当前对象明确获得或创建的对象。

出现在局部变量中的对象不是直接的朋友，因此陌生的类最好不要以局部变量的形式出现在类的内部。

迪米特原则的目的：降低类与类之间的耦合，迎合开闭原则。

例如，下面的代码段不符合迪米特原则：

```
class Teacher
{
    public void command(GroupLeader groupLeader)
    {
        List<Student> listStudents = new ArrayList<Student>();
        for (int i = 0; i < 20; i++)
        {
            listStudents.add(new Student());
        }
        groupLeader.countStudents(listStudents);
    }
}
```

`Student`类出现在局部变量中，不是直接的朋友，违反了迪米特原则。重构如下：

```
class GroupLeader
{
    private List<Student> listStudents;

    public GroupLeader(List<Students> listStudents)
    {
        this.listStudents = listStudents;
    }

    public void countStudents()
    {
        System.out.println("学生人数是: " + listStudents.size());
    }
}

class Teacher
{
    public void command(GroupLeader groupLeader)
    {
        groupLeader.countStudents();
    }
}
```

2.7 组合/聚合复用原则

组合/聚合复用原则（Composite/Aggregate Reuse Principle，CARP）：在可以淡化关系语义的情况下，尽量使用组合/聚合，不要使用类继承。

通过组合/聚合复用的优点：

1. 新对象存取子对象的唯一方法是通过子对象的接口。
2. 这种复用是黑箱复用，因为子对象的内部细节是新对象所看不见的。
3. 这种复用更好地支持封装性。
4. 这种复用实现上的相互依赖性比较小。
5. 每一个新的类可以将焦点集中在一个任务上。
6. 这种复用可以在运行时间内动态进行，新对象可以动态的引用与子对象类型相同的对象。
7. 作为复用手段可以应用到几乎任何环境中去。

通过组合/聚合复用的缺点：系统中会有较多的对象需要管理。

通过继承来进行复用的优点：

1. 新的实现较为容易，因为基类的大部分功能可以通过继承的关系自动进入派生类。
2. 修改和扩展继承而来的实现较为容易。

通过继承来进行复用的缺点：

1. 继承复用破坏封装性，因为继承将基类的实现细节暴露给派生类。由于基类的内部细节常常是对于派生类透明的，所以这种复用是透明的复用，又称“白箱”复用。
2. 如果基类发生改变，那么派生类的实现也不得不发生改变。
3. 从基类继承而来的实现是静态的，不可能在运行时间内发生改变，没有足够的灵活性。

3 创建型模式

创建型模式关注的是对象的创建，将创建对象（类的实例化）的过程进行了抽象和封装，分离了对象创建和对象使用。作为客户程序仅仅需要去使用对象，而不再关心创建对象过程中的逻辑。创建型模式帮助一个系统独立于如何创建、组合和表示它的那些对象。

类创建型模式使用继承改变被实例化的类，对象创建型模式将实例化委托给另一个对象。

创建型模式有两个重要的特点：

1. 客户不知道对象的具体类是什么（除非看源代码）。
2. 隐藏了对象实例是如何被创建和组织的。

3.1 单例模式（Singleton Pattern）

确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这样的类称为单例类，它提供全局访问的方法。

单例模式的结构如图 3.1 所示。

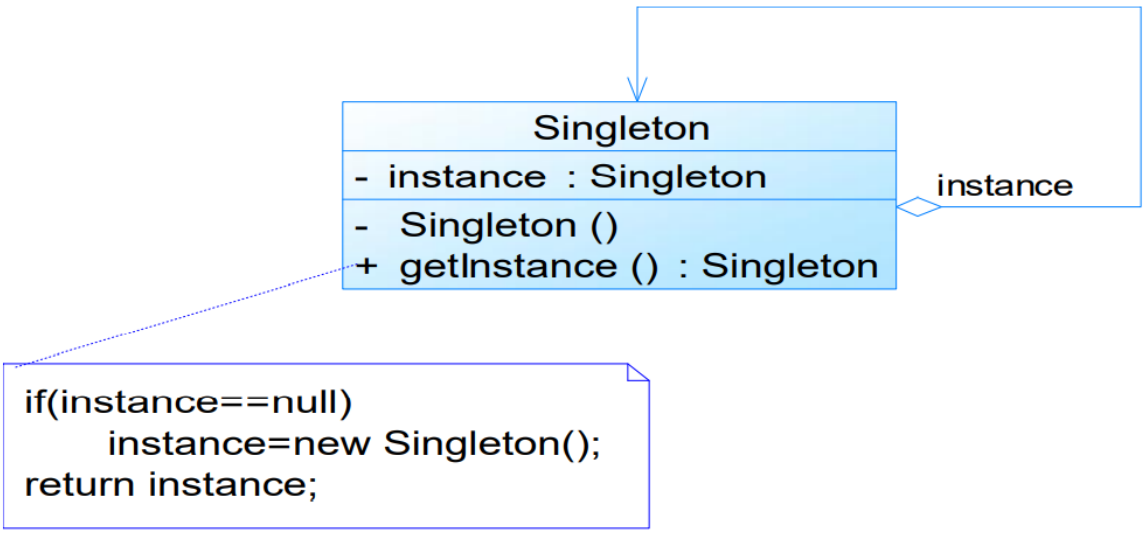

图3.1 单例模式

违背的设计原则：单一职责原则。

优点：

1. 提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。
2. 节约系统资源。单例模式保证了内存中只存在一个单例类的对象，节约系统资源。对于一些需要频繁创建和销毁的对象，单例模式可以提高系统的性能。
3. 允许可变数目的实例。可以基于单例模式进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例。
4. 缩小名字空间。单例模式是对全局变量的一种改进，它避免了那些存储唯一实例的全局变量污染名字空间。

缺点：

1. 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
2. 单例类的职责过重，在一定程度上违背了单一职责原则。因为单例类充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品本身的功能融合到一起。
3. 滥用单例将带来一些负面问题。如为了节省资源将数据库连接池对象设计为单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出。

在 Java 中，单例模式有 7 种实现方式。

3.1.1 饿汉式

要点：

1. 构造器私有化。
2. 在类的内部创建对象。
3. 对外暴露一个静态的公共方法，用于获得单例对象。

代码实现 1（静态常量）：

```
class Singleton
{
    // 在类的内部创建对象
    private final static Singleton instance = new Singleton();

    private Singleton() {} // 构造器私有化

    // 对外暴露一个静态的公共方法，用于获得单例对象
    public static Singleton getInstance()
    {
        return instance;
    }
}
```

代码实现 2（静态代码块）：

```
class Singleton
{
    // 在类的内部创建对象
    private static Singleton instance;

    static // 在静态代码块中实例化单例对象
    {
        instance = new Singleton();
    }

    private Singleton() {} // 构造器私有化

    // 对外暴露一个静态的公共方法，用于获得单例对象
    public static Singleton getInstance()
    {
        return instance;
    }
}
```

优点：写法简单，在类装载的时候就完成实例化，避免了线程同步问题。

缺点：在类装载的时候就完成实例化，没有达到懒加载的效果。如果从未使用过这个实例，就会造成内存的浪费。

3.1.2 懒汉式（线程不安全）

代码实现：

```
class Singleton
{
```

```
    private static Singleton instance;

    private Singleton() {}

    // 对外暴露一个静态的公共方法，第一次调用时才实例化单例对象
    public static Singleton getInstance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

优点：起到了懒加载的效果，但是只能在单线程下使用。

缺点：线程不安全，在多线程环境下可能会产生多个实例，所以在多线程环境下不能使用这种方式。

3.1.3 懒汉式（同步方法）

代码实现：

```
class Singleton
{
    private static Singleton instance;

    private Singleton() {}

    // 将对外接口声明为同步方法
    public static synchronized Singleton getInstance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

优点：解决了线程不安全问题。

缺点：效率低。单例对象的实例化只需执行一次，而且只有实例化的时候需要同步，后续获得单例对象时并不需要同步。整个方法进行同步会降低效率。

3.1.4 懒汉式（同步代码块）

代码实现：

```
class Singleton
{
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance()
    {
```

```
        if (instance == null)
        {
            synchronized (Singleton.class)
            {
                instance = new Singleton(); // 将实例化放在同步代码块中
            }
        }
        return instance;
    }
}
```

这种方式本意是想对懒汉式（同步方法）进行改进，但是这种同步并不能起到线程同步的作用。与懒汉式（线程不安全）一样，如果有多个线程通过了 if 条件判断，就会产生多个实例。在实际开发中不能使用这种方式。

3.1.5 双重检查

代码实现：

```
class Singleton
{
    // volatole 关键字，将修改立即反映到内存中
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (instance == null)
        {
            synchronized (Singleton.class)
            {
                if (instance == null) // 在同步代码块中再次判断
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

优点：线程安全，懒加载，效率高。

在实际开发中推荐使用这种实现方式。

3.1.6 静态内部类

代码实现：

```
class Singleton
{
    private Singleton() {}

    private static class SingletonInstance // 静态内部类
    {
```

```
        // 静态属性
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance()
    {
        return SingletonInstance.INSTANCE;
    }
}
```

当外部类被装载时，静态内部类不会被装载；当使用静态内部类时，才会装载静态内部类；类的静态属性只会在第一次加载类的时候初始化，JVM 保证了初始化时的线程安全。

优点：线程安全，懒加载，效率高。

在实际开发中推荐使用这种实现方式。

3.1.7 枚举

代码实现：

```
enum Singleton
{
    INSTANCE; // 单例对象

    // 其他方法
    public void method
    {
        // ...
    }
}
```

优点：不仅能避免线程同步问题，还能防止反序列化重新创建新的对象。

在实际开发中推荐使用这种实现方式。

3.1.8 单例模式应用实例

在 JDK 中，`java.lang.Runtime`类使用了单例模式，代码如下：

```
public class Runtime
{
    private static Runtime currentRuntime = new Runtime();

    public static Runtime getRuntime()
    {
        return currentRuntime;
    }

    private Runtime();
}
```

3.2 简单工厂模式（Simple Factory Pattern）

3.2.1 引例：披萨订购

问题描述：披萨有很多种类（如 GreekPizza、CheesePizza 等），披萨的制作有 prepare、bake、cut、box 等步骤。完成披萨店订购功能。

初步实现：

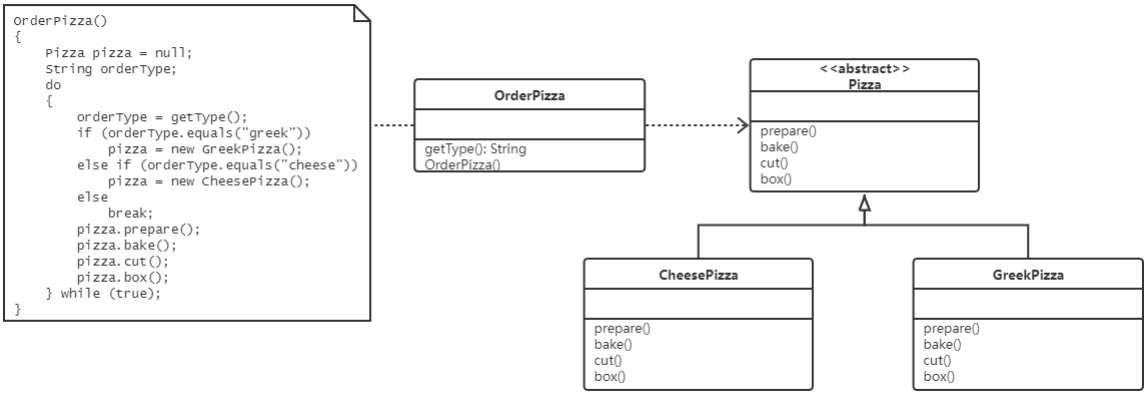

图3.2 披萨订购：初步实现

这种方法的优点是好理解、简单易操作。缺点是违反了开闭原则，当增加披萨种类时，需要在`OrderPizza`方法中增加 if 条件判断，并且所有创建`Pizza`对象的地方都需要修改。

改进：把创建`Pizza`对象的操作封装到一个类中，当增加披萨种类时，只需要修改该类即可，其他创建`Pizza`对象的代码就不需要修改了。

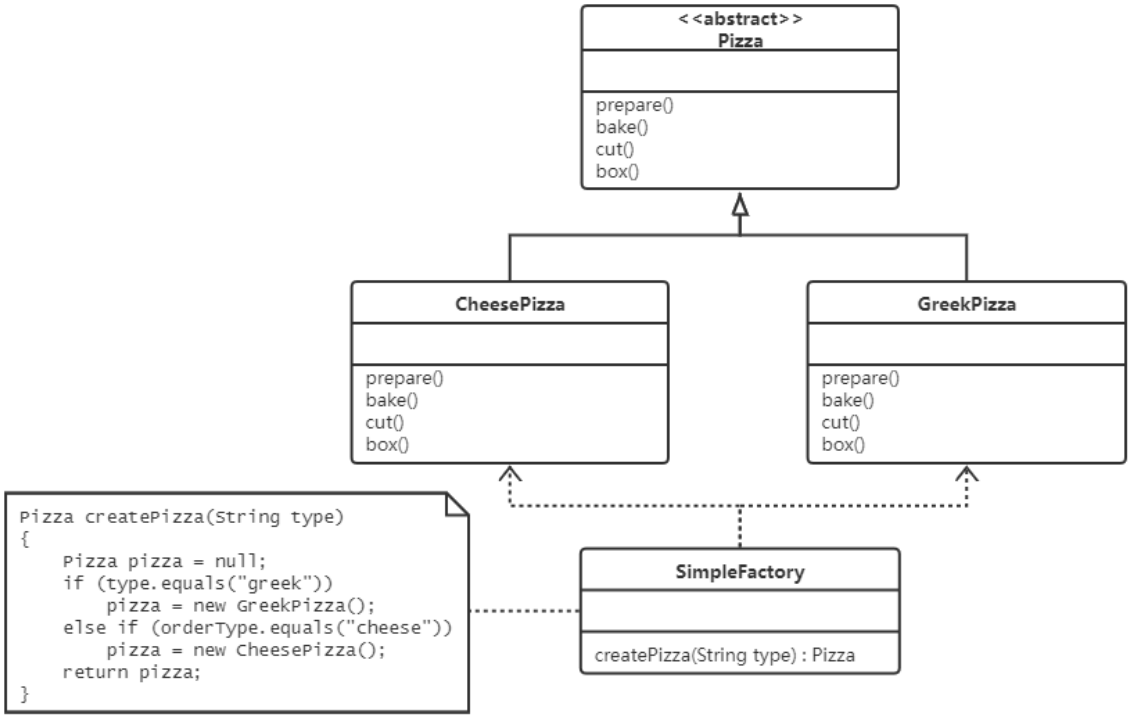

图3.3 披萨订购：简单工厂

3.2.2 简单工厂模式介绍

简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为。用户将需要的产品类型作为参数传入，就可以获取对象，而无须知道其创建细节。

简单工厂模式的结构如图 3.4 所示。

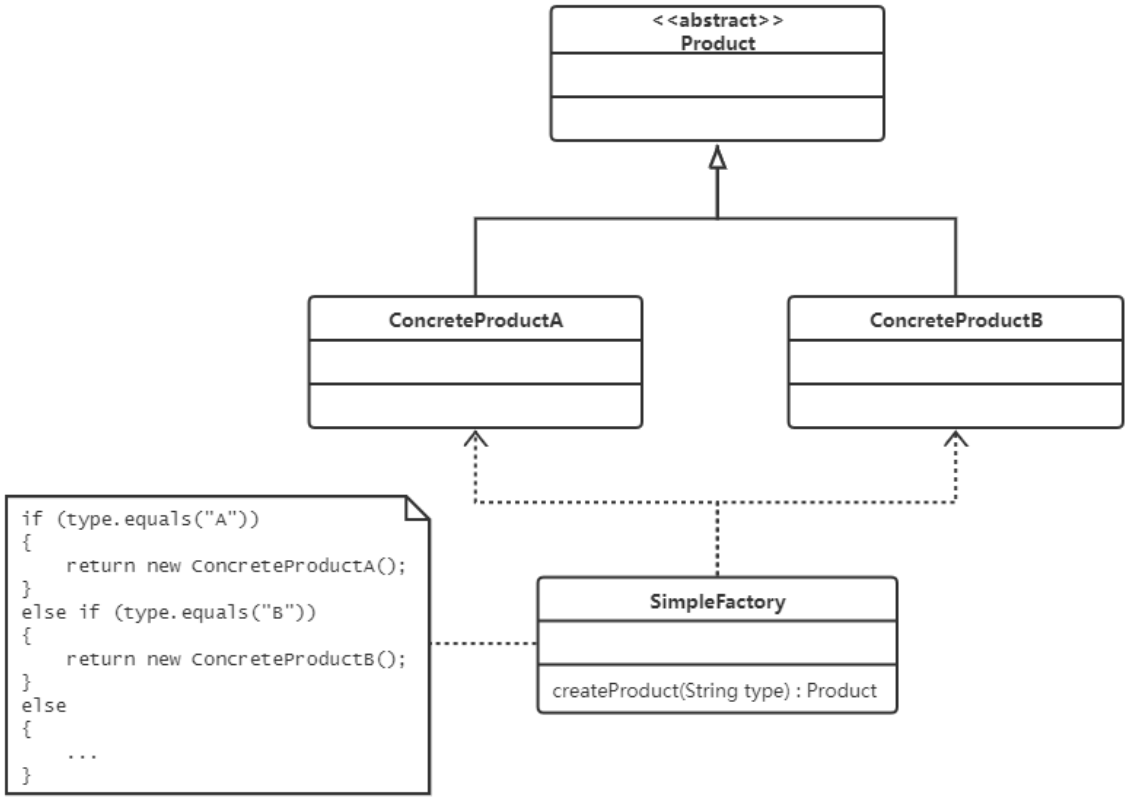

图3.4 简单工厂模式

可以把工厂类的工厂方法设计成静态方法，可以通过类名直接调用，使用方便。

简单工厂模式的缺点：

1. 工厂类的工厂方法职责相对过重，业务逻辑较为复杂，具体产品与工厂类之间的耦合度高，严重影响了系统的灵活性和扩展性。
2. 增加新的产品需要修改工厂类的判断逻辑，违背了开闭原则。

简单工厂模式的适用场合：需要创建的对象类特别少，工厂类的工厂方法实现时分支少。

简单工厂模式的应用实例：JDK 中的 `java.util.Calendar` 类使用了简单工厂模式，代码如下：

```
1  public abstract class Calendar implements Serializable, Cloneable,
   Comparable<Calendar>
2  {
3      // ...
4
5      public static Calendar getInstance()
6      {
7          return createCalendar(TimeZone.getDefault(),
   Locale.getDefault(Locale.Category.FORMAT));
8      }
9
10     private static Calendar createCalendar(TimeZone zone, Locale aLocale)
11     {
12         CalendarProvider provider =
13             LocaleProviderAdapter.getAdapter(CalendarProvider.class,
   aLocale)
14                                  .getCalendarProvider();
15         if (provider != null)
16         {
17             try
18             {
```

```
                return provider.getInstance(zone, aLocale);
            }
            catch (IllegalArgumentException iae)
            {
                // fall back to the default instantiation
            }
        }

        Calendar cal = null;

        if (aLocale.hasExtensions())
        {
            String caltype = aLocale.getUnicodeLocaleType("ca");
            if (caltype != null)
            {
                switch (caltype)
                {
                case "buddhist":
                cal = new BuddhistCalendar(zone, aLocale);
                    break;
                case "japanese":
                    cal = new JapaneseImperialCalendar(zone, aLocale);
                    break;
                case "gregory":
                    cal = new GregorianCalendar(zone, aLocale);
                    break;
                }
            }
        }
        if (cal == null)
        {
            // If no known calendar type is explicitly specified,
            // perform the traditional way to create a Calendar:
            // create a BuddhistCalendar for th_TH locale,
            // a JapaneseImperialCalendar for ja_JP_JP locale, or
            // a GregorianCalendar for any other locales.
            // NOTE: The language, country and variant strings are interned.
            if (aLocale.getLanguage() == "th" && aLocale.getCountry() ==
"TH")
            {
                cal = new BuddhistCalendar(zone, aLocale);
            }
            else if (aLocale.getVariant() == "JP" && aLocale.getLanguage()
== "ja"
                     && aLocale.getCountry() == "JP")
            {
                cal = new JapaneseImperialCalendar(zone, aLocale);
            } else
            {
                cal = new GregorianCalendar(zone, aLocale);
            }
        }
        return cal;
    }

    // ...
}
```

3.3 工厂方法模式（Factory Method Pattern）

3.3.1 引例：披萨订购

问题描述：客户在订购披萨时，可以选择披萨的种类，也可以选择披萨的产地。例如，北京的 CheesePizza、北京的 GreekPizza、伦敦的 CheesePizza、伦敦的 GreekPizza 等。

可以使用简单工厂模式，对披萨的种类和产地分别进行判断。这种方法逻辑复杂，if 分支较多，不易于扩展和维护。

改进：定义一个抽象的工厂类，并为每种披萨定义一个具体工厂子类，让具体工厂子类来创建实例。当有新的地区和披萨种类时，只需要增加具体工厂子类即可。

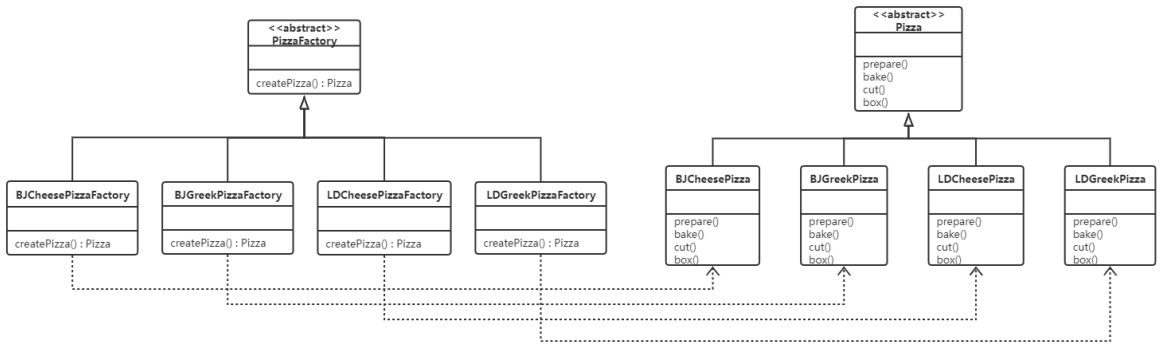

图3.5 披萨订购：工厂方法模式

3.3.2 工厂方法模式介绍

工厂方法模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类，将类的实例化延迟到子类。

别名：虚拟构造器模式（Virtual Constructor Pattern）、多态工厂模式（Polymorphic Factory Pattern）。

工厂方法模式相当于利用里氏替换原则对简单工厂模式的重构。

工厂方法模式的结构如图 3.6 所示。

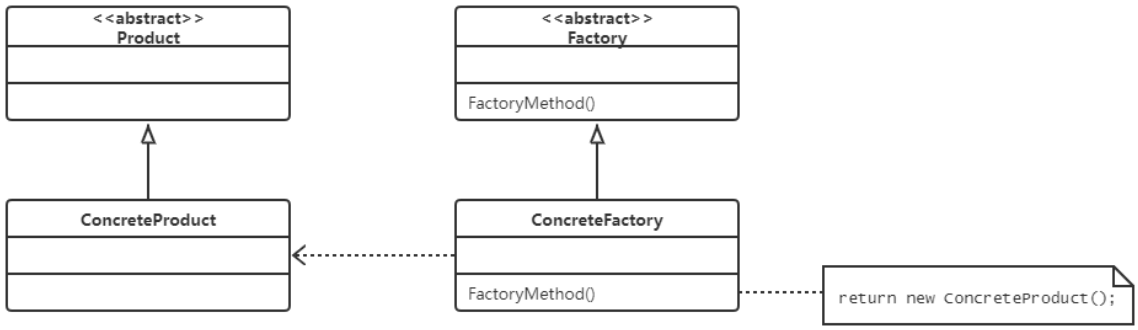

图3.6 工厂方法模式

在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。当系统需要添加新的产品对象时，仅仅需要添加一个具体产品对象和一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，符合开闭原则。

符合的设计原则：迪米特原则、依赖倒置原则、里氏替换原则。

工厂方法模式的适用场合：

1. 客户程序不需要知道具体产品类的类名，只需要知道所对应的工厂，具体的产品对象由具体工厂类创建。
2. 当一个类希望由它的子类来指定它所创建的对象的时候。
3. 把创建对象的任务委托给多个工厂子类中的一个，客户程序可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定。

如果工厂等级结构中只有一个具体工厂类，抽象工厂类就可以省略，工厂方法模式就发生了退化。当只有一个具体工厂，在具体工厂中可以创建所有的产品对象时，工厂方法模式就退化成简单工厂模式。

工厂方法模式的缺点：

1. 在添加新产品时，需要增加新的具体产品类和具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，给系统带来一些额外的开销。
2. 由于考虑到系统的可扩展性，需要引入抽象层，在客户程序中均使用抽象层进行定义，增加了系统的抽象性和理解难度。且在实现时可能需要用到 DOM、反射等技术，增加了系统的实现难度。

3.4 抽象工厂模式（Abstract Factory Pattern）

3.4.1 引例：披萨订购

对工厂方法模式实现的披萨订购功能进行进一步改进，做如下考虑：对于 CheesePizza，有 BJCheesePizza 和 LDCheesePizza 两种；对于 GreekPizza，也有 BJGreekPizza 和 LDGreekPizza 两种。这样可以将披萨产品先按照种类分类，每个种类有不同产地。工厂可以按照产地分为北京工厂和伦敦工厂，每个地区的工厂可以生产不同种类的披萨。

基于以上分析做出设计，如图 3.7 所示。

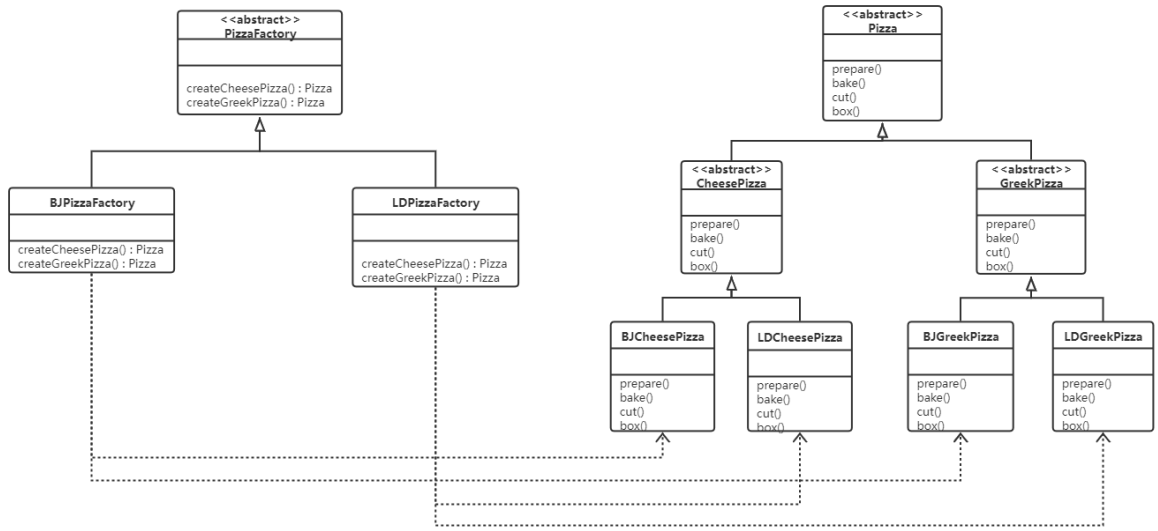

图3.7 披萨订购：抽象工厂模式

3.4.2 抽象工厂模式介绍

在工厂方法模式中，每个具体工厂对应一种具体产品。但有时需要一个工厂提供多种产品对象，而不是单一的产品对象。

引入两个概念：产品等级结构、产品族。

产品等级结构：产品的继承结构。如`BJPizza`是抽象类，`BJCheesePizza`和`BJGreekPizza`是`BJPizza`的具体子类，这三个类构成了一个产品等级结构。相应地，`LDPizza`、`LDCheesePizza`、`LDGreekPizza`构成另一个产品等级结构。

产品族：同一个工厂生产的、位于不同产品等级结构中的一组产品。如`BJPizzaFactory`生产的`BJCheesePizza`和`BJGreekPizza`，它们由同一个工厂生产，属于不同的产品等级结构，构成了一个产品族。

产品等级结构与产品族的关系如图 3.8 所示。

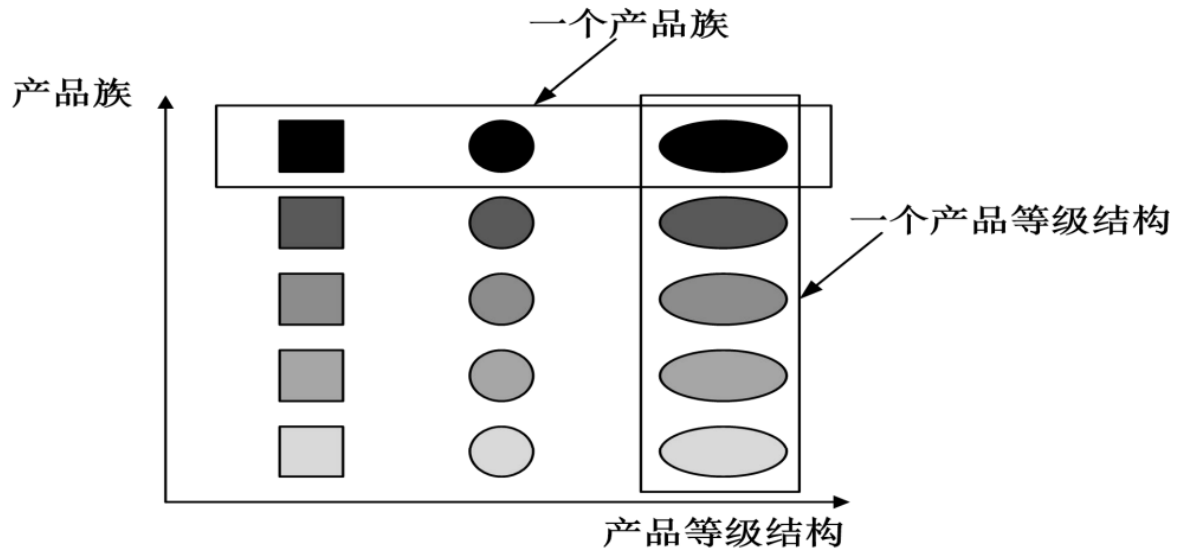

图3.8 产品等级结构与产品族

当具体产品位于多个不同产品等级结构，每个产品等级结构有多种类型时，需要使用抽象工厂模式。

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。

抽象工厂模式是所有工厂模式中最抽象、最具一般性的一种。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构。在抽象工厂模式中，一个工厂等级结构可以创建出多个不同产品等级结构的产品，因此抽象工厂模式比工厂方法模式更为简单、有效率。

抽象工厂模式的设计思路如图 3.9 所示。

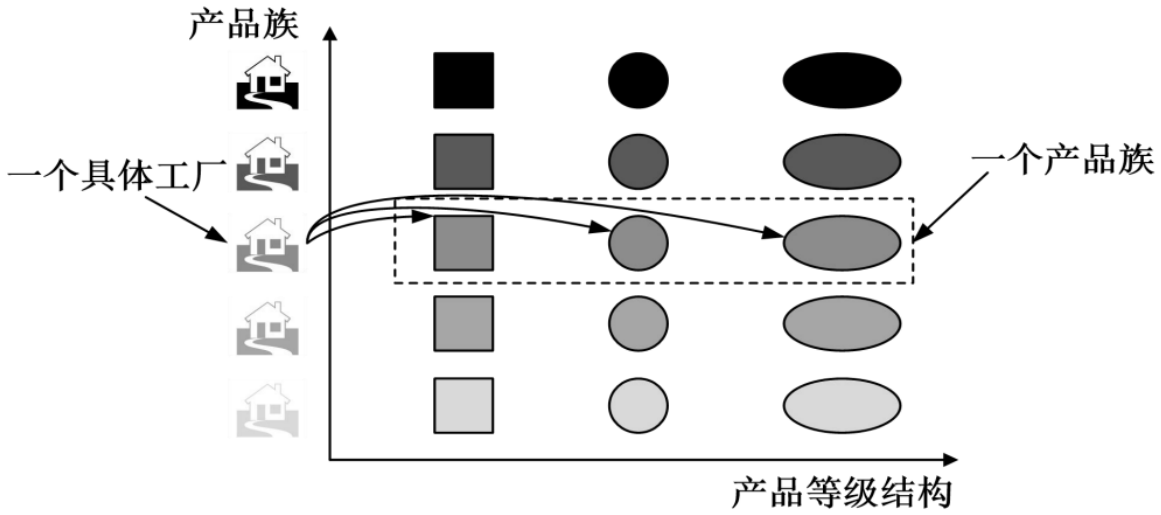

图3.9 抽象工厂模式设计思路

抽象工厂模式的结构如图 3.10 所示。

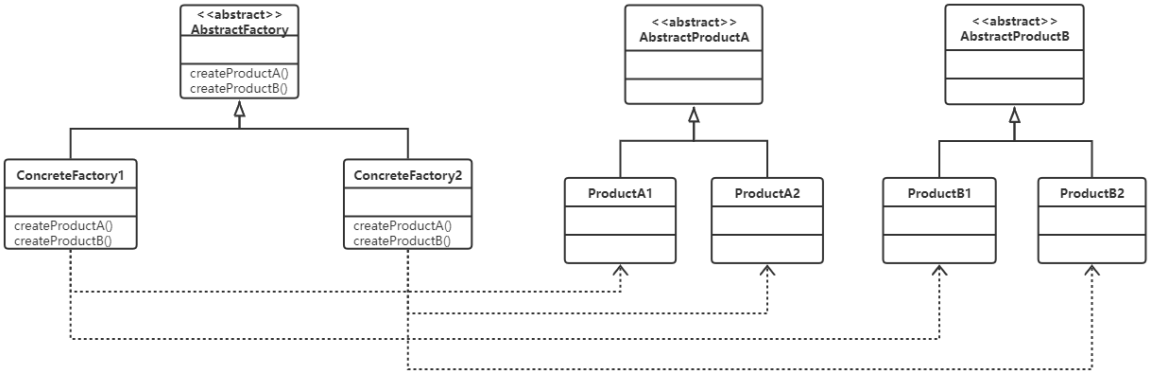

图3.10 抽象工厂模式

抽象工厂模式的优点：

1. 隔离了具体类的生成，使得客户程序并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。
2. 抽象工厂模式可以实现高内聚、低耦合的设计目的。
3. 当一个产品族中的多个对象被设计成一起工作时，抽象工厂模式可以保证客户端始终只使用同一个产品族中的对象。这对一些需要根据当前环境来决定其行为的系统来说，是一种非常实用的设计模式。
4. 增加新的具体工厂和产品族很方便，无须修改已有系统，符合开闭原则。

抽象工厂模式的缺点：开闭原则的倾斜性，增加新的工厂和产品族容易，增加新的产品等级结构困难。因为在抽象工厂中规定了所有的产品等级结构，导致在添加新的产品等级结构时，难以扩展抽象工厂来生产新种类的产品。要支持新的产品等级结构就要对该接口进行扩展，而这将涉及对抽象工厂及其所有子类的修改，显然会带来较大的不便。

抽象工厂模式的适用场合：

1. 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节。
2. 系统中有多于一个的产品族，而每次只使用其中某一产品族。
3. 属于同一个产品族的产品将在一起使用。
4. 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于具体实现。

3.5 原型模式（Prototype Pattern）

原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

原型是允许一个对象创建另外一个可定制的对象，无须知道任何创建的细节。

基本工作原理：将一个原型对象传给那个要发起创建的对象，这个要发起创建的对象通过请求原型对象拷贝自己来实现创建过程。

原型模式的结构如图 3.11 所示。

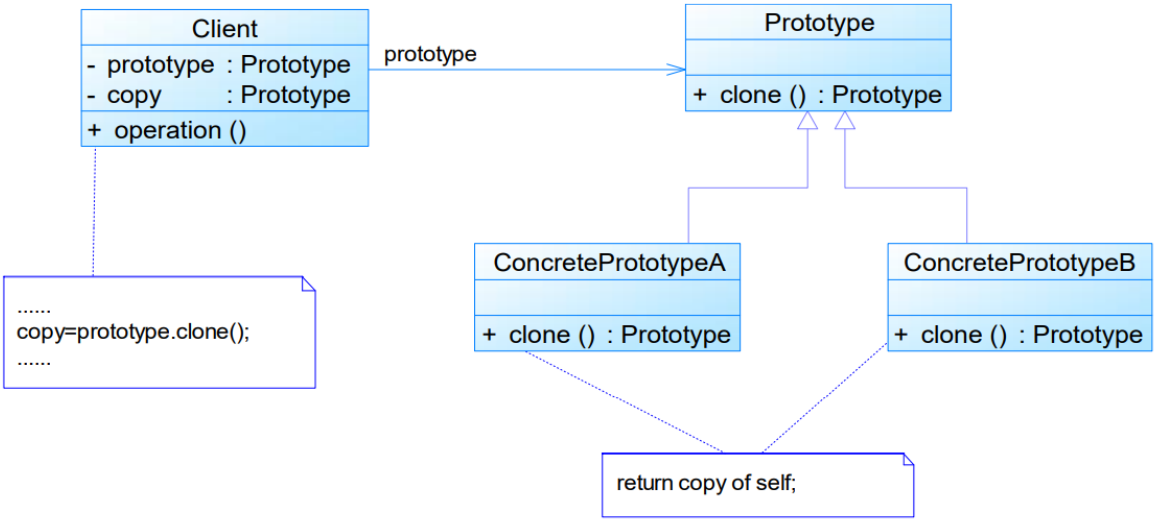

图3.11 原型模式

原型模式的优点：

1. 原型模式提供了简化的创建结构，产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品，减少了类的数目。
2. 可以使用深拷贝的方式保存对象的状态，以便需要的时候使用。
3. 允许客户在运行时增加和删除产品，灵活性强。
4. 对客户隐藏了具体的产品类，减少了客户知道的类的数量。

原型模式的缺点：

1. 需要为每个子类实现一个克隆方法，这对全新的类来说不难，但对已有的类进行改造时会很困难，必须修改其源代码，违背了开闭原则。
2. 实现深拷贝时代码比较复杂，而且当对象之间存在多重嵌套引用时，为了实现深拷贝，每一层对象对应的类都必须支持深拷贝，实现起来会比较麻烦。

原型模式的适用场合：

1. 创建新对象成本较大，而新对象中的信息和已有对象相比变化不大，就可以通过原型模式对已有对象进行复制来获得新对象。
2. 如果系统要保存对象的状态，而对象的状态变化很小，或者对象本身占内存不大的时候，可以使用原型模式配合备忘录模式来应用。
3. 需要避免创建一个与产品类层次平行的工厂类层次时，可以使用原型模式。
4. 当一个类的实例只能有几个不同状态组合中的一种时，可以建立相应数目的原型并克隆它们。
5. 当一个系统应该独立于它的产品创建、构成和表示时。
6. 当要实例化的类是在运行时指定时。

原型模式的应用实例：Java 中 `java.lang.Object` 类是所有类的基类，`Object` 类提供了一个 `clone` 方法，可以将一个 Java 对象复制一份。要实现 `clone` 方法的类必须实现 `Cloneable` 接口，如果一个类没有实现 `Cloneable` 接口但是调用了 `clone` 方法，将抛出一个 `CloneNotSupportedException` 异常。

在 Java 中，深拷贝有两种实现方式：重写 `clone` 方法、对象序列化。例如：

```
public class DeepCloneableTarget implements Serializable, Cloneable
{
    private static final long serialVersionUID = 1L;
    private String cloneName;
    private String cloneClass;

    public DeepCloneableTarget(String cloneName, String cloneClass)
    {
```

```
        this.cloneName = cloneName;
        this.cloneClass = cloneClass;
    }

    // 此类的属性都是基本类型和 String 类型，因此使用默认的 clone 方法即可
    protected Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

public class DeepProtoType implements Serializable, Cloneable
{
    public String name;
    public DeepCloneableTarget deepCloneableTarget;

    public DeepProtoType()
    {
        super();
    }

    // 方式 1: 重写 clone 方法
    protected Object clone() throws CloneNotSupportedException
    {
        Object deep = null;
        // 完成对基本数据类型和 String 类型的拷贝
        deep = super.clone();
        // 对引用类型的属性单独处理
        DeepProtoType deepProtoType = (DeepProtoType)deep;
        deepProtoType.deepCloneableTarget =
(DeepCloneableTarget)deepCloneableTarget.clone();

        return deepProtoType;
    }

    // 方式 2: 对象序列化（推荐使用）
    public Object deepClone()
    {
        // 创建流对象
        ByteArrayOutputStream bos = null;
        ObjectOutputStream oos = null;
        ByteArrayInputStream bis = null;
        ObjectInputStream ois = null;

        try
        {
            // 序列化
            bos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(bos);
            oos.writeObject(this); // 将当前对象以对象流的方式输出

            // 反序列化
            bis = new ByteArrayInputStream(bos.toByteArray());
            ois = new ObjectInputStream(bis);
            DeepProtoType copy = ois.readObject();
            return copy;
        }
        catch (Exception e)
```

```
66          {
67              e.printStackTrace();
68              return null;
69          }
70          finally
71          {
72              try
73              {
74                  bos.close();
75                  oos.close();
76                  bis.close();
77                  ois.close();
78              }
79              catch (Exception e2)
80              {
81                  System.out.println(e2.getMessage());
82              }
83          }
84      }
85  }
```

3.6 建造者模式（Builder Pattern）

建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

建造者模式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，而不需要知道其内部的具体构建细节。

建造者模式的结构如图 3.12 所示。

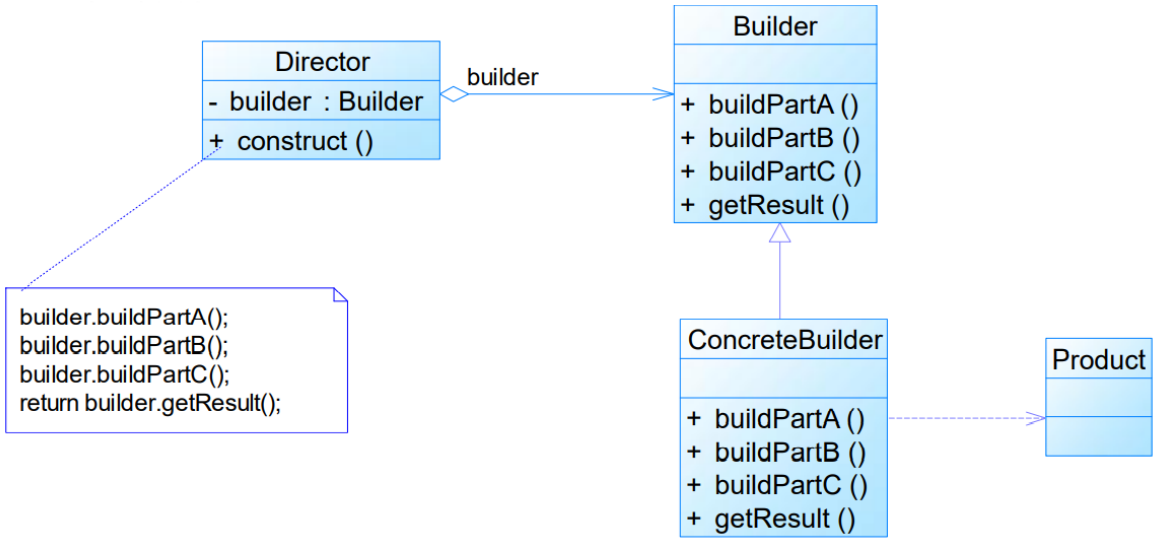

图3.12 建造者模式

- Builder：抽象建造者，为创建一个产品对象的各个部件指定抽象接口。
- ConcreteBuilder：具体建造者，实现了抽象建造者接口，实现各个部件的构造和装配方法，定义并明确它所创建的复杂对象。也可以提供一个方法返回创建好的复杂产品对象。
- Director：指挥者，负责安排复杂对象的建造次序。Director 与 Builder 存在关联关系，可以在 construct 方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造。
- Product：产品，被构建的复杂对象，包含多个组成部件。

Director 的作用主要体现在两方面：

1. 隔离了客户程序与生产过程。指挥者针对抽象建造者编程，客户程序只需要知道具体建造者的类型，即可通过指挥者类调用建造者的相关方法，返回一个完整的产品对象。

2. 负责控制产品的生产过程。

示例代码：

```
public class Product
{
    private String partA;
    private String partB;
    private String partC;

    // setter 方法省略
}

public abstract class Builder
{
    protected Product product = new Product();

    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();

    public Product getResult()
    {
        return product;
    }
}

public class ConcreteBuilder extends Builder
{
    public void buildPartA()
    {
        product.setPartA("A");
    }

    public void buildPartB()
    {
        product.setPartB("B");
    }

    public void buildPartC()
    {
        product.setPartC("C");
    }
}

public class Director
{
    private Builder builder;

    public Director(Builder builder)
    {
        this.builder = builder;
    }

    public void setBuilder(Builder builder)
    {
        this.builder = builder;
```

```
    }

    public Product construct()
    {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.getResult();
    }
}

public class Client
{
    public static void main(String[] args)
    {
        Builder builder = new ConcreteBuilder();
        Director director = new Director(builder);
        Product product = director.construct();
    }
}
```

建造者模式的优点：

1. 客户程序不必知道产品内部组成的细节，把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。
2. 每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。
3. 可以更加精细地控制产品的创建过程。
4. 增加新的具体建造者无须修改原有的代码，指挥者类针对抽象建造者类编程，系统扩展方便，符合开闭原则。

建造者模式的缺点：

1. 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。
2. 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

建造者模式的适用场合：

1. 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。
2. 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。
3. 对象的创建过程独立于创建该对象的类。在建造者模式中引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类中。
4. 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品对象。

建造者模式的简化：如果系统中只有一个具体建造者，可以省略掉抽象建造者。进一步还可以省略指挥者，将指挥者的 construct 方法写在具体建造者中。

建造者模式的应用实例：JDK 中的`StringBuilder`类使用了建造者模式。`String`类的对象是不可变的，它关注的是对字符串本身的操作，相当于产品；`StringBuilder`类关注的是字符串的构造，相当于建造者；指挥者是`StringBuilder`类的客户程序，因为所有字符串组成的集合是一个无限集，不可能有一个统一的指挥者创建所有的字符串，因此将指挥者的功能交给客户程序实现。相关代码如下：

```
// 抽象建造者
public interface Appendable
{
    Appendable append(CharSequence csq) throws IOException;
```

```
    Appendable append(CharSequence csq, int start, int end) throws
IOException;
    Appendable append(char c) throws IOException;
}

// 具体建造者
abstract class AbstractStringBuilder implements Appendable, CharSequence
{
    // ...

    public AbstractStringBuilder append(String str)
    {
        if (str == null)
            return appendNull();
        int len = str.length();
        ensureCapacityInternal(count + len);
        str.getChars(0, len, value, count);
        count += len;
        return this;
    }

    // ...
}

// 具体建造者
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, CharSequence
{
    // ...

    @Override
    public StringBuilder append(String str)
    {
        super.append(str);
        return this;
    }

    // ...
}
```

4 结构型模式

结构型模式涉及如何组合类和对象以获得更大的结构。结构型类模式采用继承机制来组合接口或实现。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。

4.1 适配器模式（Adapter Pattern）

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。

别名：包装器模式（Wrapper Pattern）。

适配器分为三类：类适配器、对象适配器、接口适配器。

工作原理：

1. 定义一个包装类，包装不兼容接口的类或对象，这个包装类就是适配器（Adapter），它所包装的类或对象就是适配者（Adaptee）。
2. 适配器提供客户需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。客户调用适配器的方法，在适配器内部调用适配者的方法，这个过程对客户是透明的，客户并不直接访问适配者。

适配器模式的优点：

1. 把目标类和适配者解耦，通过引入一个适配器类来复用现有的适配者，而无须修改原有代码。
2. 增加了类的透明性和复用性，把具体的实现封装在适配者类中，对于客户来说是透明的，而且提高了适配者的复用性。
3. 支持灵活性和扩展性，通过配置文件可以更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，符合开闭原则。

适配器模式的适用场合：

1. 客户程序需要使用现有的类，而这些类的接口不符合客户程序的需要。
2. 想要创建一个可以重复使用的类（适配器类），该类可以与其他不相关的类或不可预见的类（可能在将来引入的类）协同工作。

4.1.1 类适配器

类适配器的结构如图 4.1 所示。

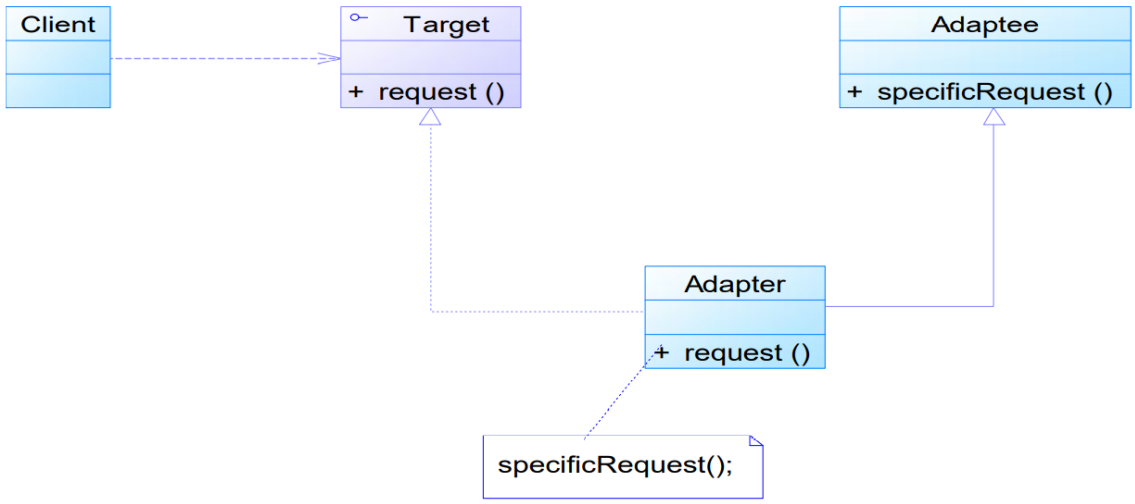

图4.1 类适配器

- Target：客户程序期望的接口。
- Adapter：适配器。
- Adaptee：适配者，已经存在的类。

代码示例：

```
class Adaptee
{
    publiv void specificRequest()
    {
        // 适配者方法
    }
}

interface Target
{
    public void request();
```

```
}

public class Adapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest(); // 调用适配者方法
        // 做其他处理
    }
}
```

类适配器的优点：由于适配器类是适配者类的子类，因此可以在适配器类中重写一些适配者类的方法，使得适配器的灵活性更强。

类适配器的缺点：

1. 对于 Java、C# 等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为接口，不能为具体类，其使用有一定的局限性。
2. 不能将一个适配者类和它的子类都适配到目标接口。

4.1.2 对象适配器

对象适配器的结构如图 4.2 所示。

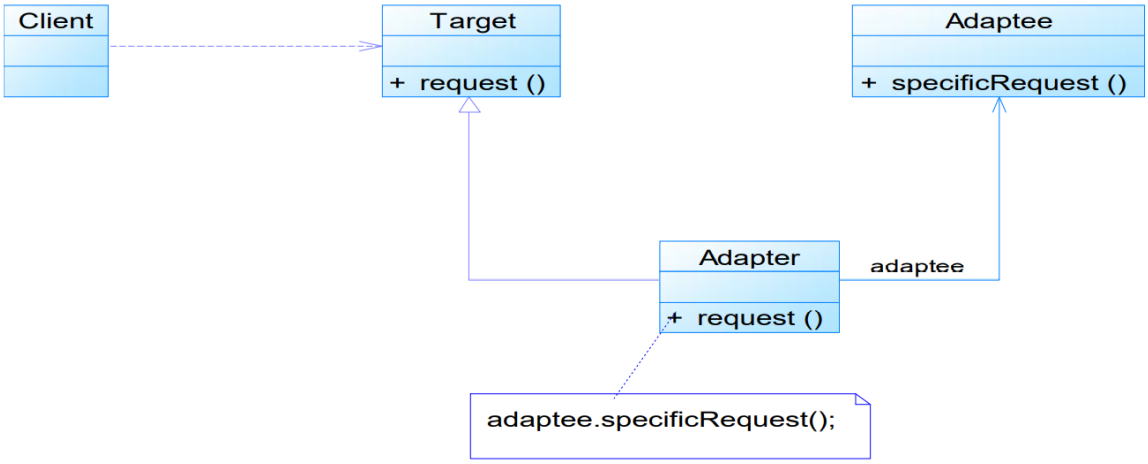

图4.2 对象适配器

代码示例：

```
class Adaptee
{
    publiv void specificRequest()
    {
        // 适配者方法
    }
}

abstract class Target
{
    public void request();
}

public class Adapter extends Target
{
    private Adaptee adaptee; // 关联适配者对象
```

```
17
18      public Adapter(Adaptee adaptee) // 通过构造器传入适配者对象
19      {
20          this.adaptee = adaptee;
21      }
22
23      public void request()
24      {
25          adaptee.specificRequest(); // 调用适配者方法
26          // 做其他处理
27      }
28  }
```

对象适配器的优点：一个可以把多个不同的适配者适配到同一个目标，同一个适配器可以把适配者类和它的子类都适配到目标接口。

对象适配器的缺点：与类适配器相比，想要重写适配者类的方法不容易。如果要重写适配者类的方法，就要先写适配者类的子类，在子类中重写适配者类的方法，然后再对子类进行适配，实现过程较为复杂。

对象适配器的应用实例：JDK 类库中定义了一系列适配器类，如 `com.sun.imageio.plugins.common.InputStreamAdapter` 类用于包装 `ImageInputStream` 接口及其子类，代码如下：

```
1   public abstract class InputStream implements Closeable
2   {
3       // ...
4
5       public abstract int read() throws IOException;
6
7       public int read(byte b[], int off, int len) throws IOException
8       {
9           // ...
10      }
11  }
12
13  public class InputStreamAdapter extends InputStream
14  {
15
16      ImageInputStream stream;
17
18      public InputStreamAdapter(ImageInputStream stream)
19      {
20          super();
21
22          this.stream = stream;
23      }
24
25      public int read() throws IOException
26      {
27          return stream.read();
28      }
29
30      public int read(byte b[], int off, int len) throws IOException
31      {
32          return stream.read(b, off, len);
33      }
```

```
}
```

4.1.3 接口适配器

接口适配器模式又叫做默认适配器模式（Default Adapter Pattern）或缺省适配器模式。

当不需要全部实现接口提供的方法时，可以先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），该抽象类的子类可以有选择地覆盖抽象类的某些方法来实现需求。

接口适配器的应用实例：JDK 类库的事件处理包`java.awt.event`中广泛使用了接口适配器，包括`WindowAdapter`、`KeyAdapter`、`MouseAdapter`等。代码如下：

```
public interface WindowListener extends EventListener
{
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}

public interface WindowStateListener extends EventListener
{
    public void windowStateChanged(WindowEvent e);
}

public interface WindowFocusListener extends EventListener
{
    public void windowGainedFocus(WindowEvent e);
    public void windowLostFocus(WindowEvent e);
}

public abstract class WindowAdapter
    implements WindowListener, WindowStateListener, WindowFocusListener
{
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowStateChanged(WindowEvent e) {}
    public void windowGainedFocus(WindowEvent e) {}
    public void windowLostFocus(WindowEvent e) {}
}
```

4.2 桥接模式（Bridge Pattern）

4.2.1 桥接模式介绍

桥接模式：将抽象部分与它的实现部分分离，使它们可以独立地变化。

桥接模式基于类的最小设计原则，通过使用封装、聚合及继承等行为让不同的类承担不同的职责。它的主要特点是把抽象与实现分离，从而可以保持各部分的独立性以及应对它们的功能扩展。

桥接模式的结构如图 4.3 所示。

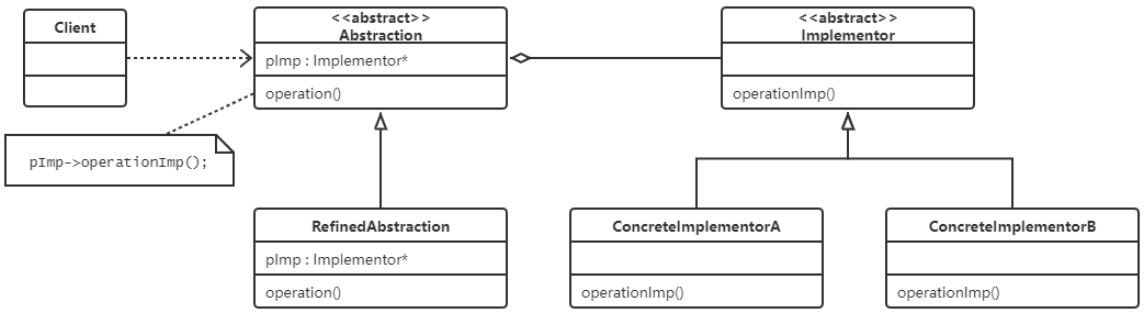

图4.3 桥接模式

- Abstraction：定义抽象类的接口，维护一个指向 Implementor 类型对象的指针，充当桥接类。
- RefinedAbstraction：扩充由 Abstraction 定义的接口。
- Implementor：定义实现类的接口，该接口不一定要与 Abstraction 的接口完全一致。
- ConcreteImplementor：实现 Implementor 接口并定义它的具体实现。

Abstraction 和 RefinedAbstraction 构成了抽象部分的继承层次，Implementor 和 ConcreteImplementor 构成了实现部分的继承层次。这两部分通过 Abstraction 中的指向 Implementor 类型对象的指针联系在一起，因此 Abstraction 充当了桥接的作用。

桥接模式的注意事项：

1. 实现了抽象和实现的分离，从而极大地提高了系统的灵活性，有助于系统进行分层设计，从而产生更好的结构化系统。
2. 对于系统的高层部分，只需要知道抽象部分和实现部分的接口就可以了，其他的部分由具体业务来完成。
3. 桥接模式替代多层继承方案，可以减少子类的个数，降低系统的管理和维护成本。
4. 桥接模式的引入增加了系统的理解和设计难度。
5. 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围有一定的局限性。

桥接模式的适用场合：对于那些不希望使用继承或因为多层次继承导致类的个数急剧增加的系统，桥接模式尤为适用。

桥接模式的应用实例：JDBC 中，`Driver`接口和`Connection`接口之间通过`DriverManager`类连接，使用了桥接模式，如图 4.4 所示。这里的桥接是通过`DriverManager`类实现的，而不是通过聚合关系。

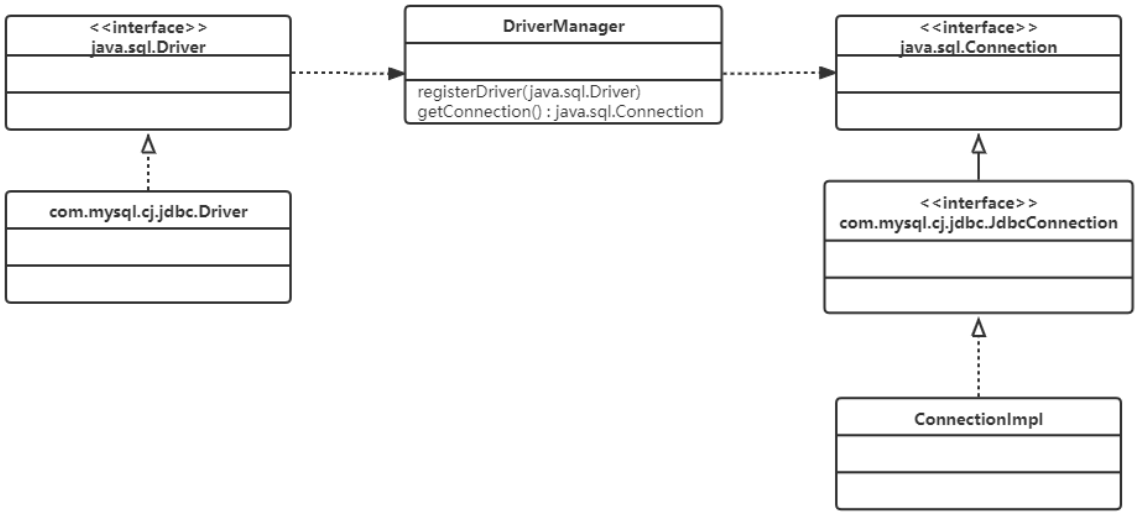

图4.4 JDBC中的桥接模式

4.2.2 示例：手机问题

问题描述：手机分为不同样式，如折叠式、直立式、滑盖式等；同时手机也有不同品牌，如小米、Vivo、华为等。不同样式和品牌的手机，对它们的操作相同（如开机、关机、打电话等），但操作的实现不同。实现不同样式、不同品牌手机的操作。

初步设计：手机有不同样式，每个样式的手机又有不同品牌，据此可以给出如图 4.5 所示的继承结构。

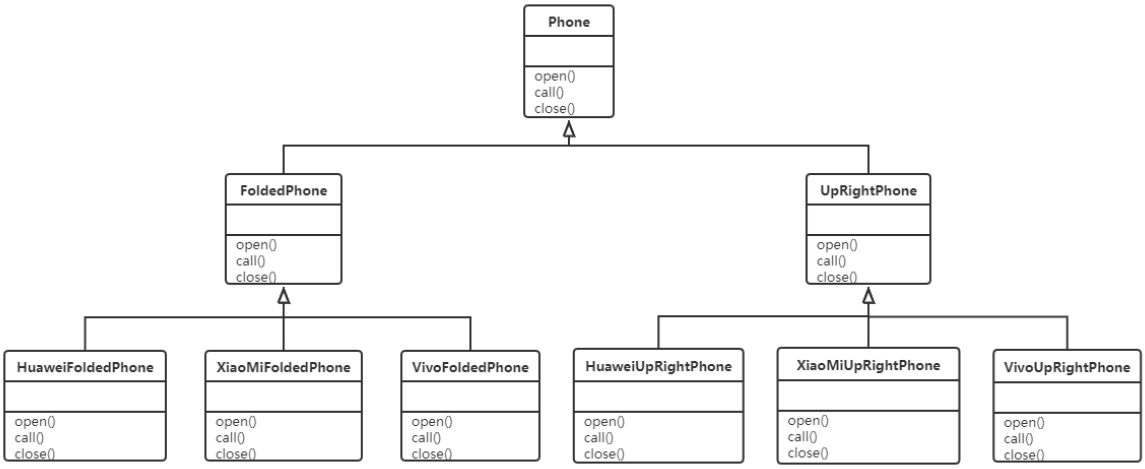

图4.5 手机问题：初步设计

这种设计的问题在于，当增加手机样式时，就需要增加各个品牌手机的类；当增加手机品牌时，也要在各个手机样式类之下增加品牌类。当系统扩展时，需要增加很多类，发生类爆炸，增加了代码维护成本。

用桥接模式进行改进：将手机样式看做抽象部分，将手机品牌看做实现部分，使用桥接模式将二者分离。

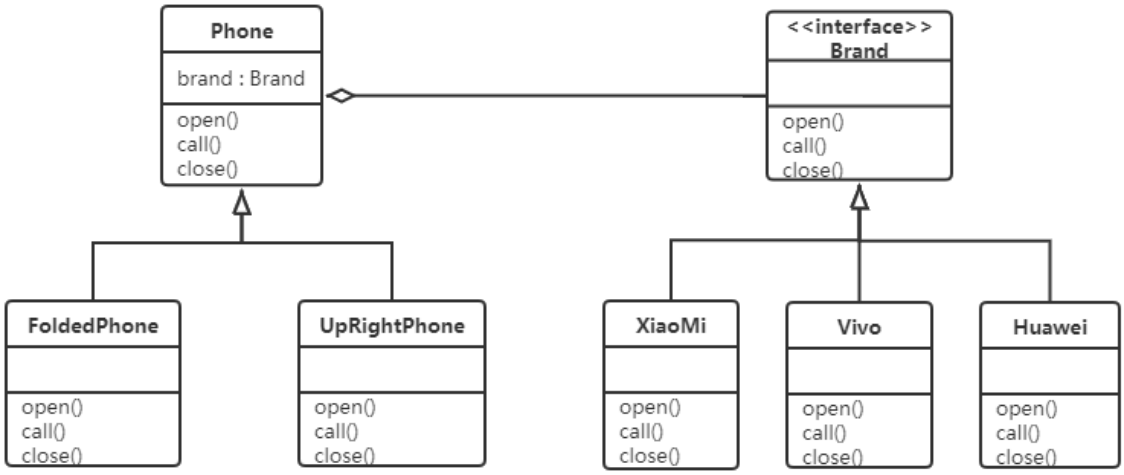

图4.6 手机问题：桥接模式

调用 `FoldedPhone` 的方法时，通过其内部指向 `Brand` 接口的指针，调用了品牌类的相应方法，这体现了 `Phone` 类的桥接作用。当增加手机样式时，只需要增加 `Phone` 类的子类；当增加手机品牌时，只需要增加一个实现 `Brand` 接口的类。通过桥接模式解决了类爆炸问题，提高了代码的可维护性。

示例代码如下（为简便，只写了一个样式和一个品牌，其他样式和品牌类似）：

```
abstract class Phone
{
    private Brand brand;

    public Phone(Brand brand)
    {
        this.brand = brand;
    }

    protected void open()
    {
        brand.open();
    }
}

class FoldedPhone extends Phone
{
    public FoldedPhone(Brand brand)
    {
        super(brand);
    }

    public void open()
    {
        super.open();
        System.out.println("折叠样式手机");
    }
}

interface Brand
{
    void open();
}

```

```java
class XiaoMi implements Brand
{
    public void open()
    {
        System.out.println("小米手机开机");
    }
}

public class Client
{
    public static void main(String[] args)
    {
        Phone foldedPhone = new FoldedPhone(new XiaoMi());
        foldedPhone.open();
    }
}
```

4.3 装饰模式（Decorator Pattern）

4.3.1 装饰模式介绍

装饰模式：动态地将新功能附加到对象上。

在对象的功能扩展方面，装饰模式比继承更灵活，体现了开闭原则。

装饰模式的结构如图 4.7 所示。

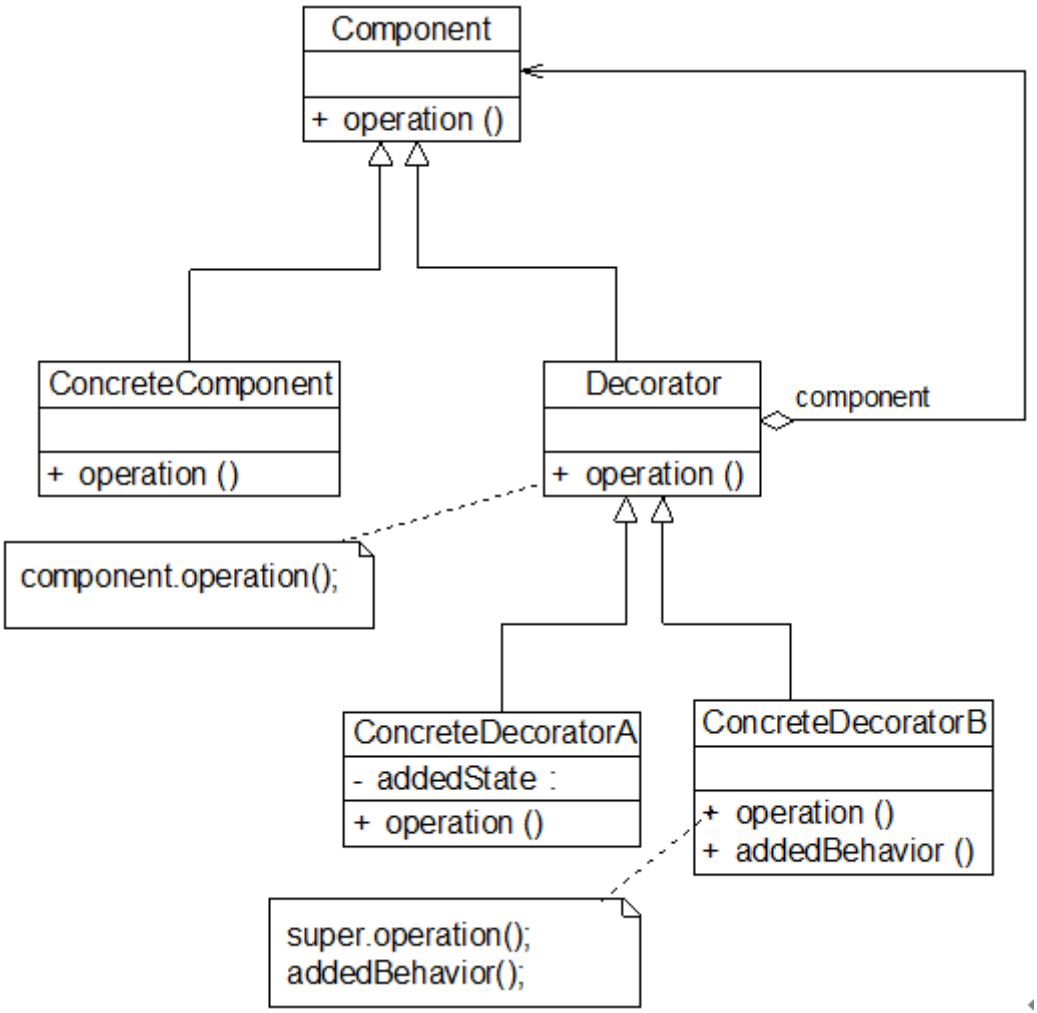

图4.7 装饰模式

- Component：抽象构件类。它是具体构件和抽象装饰类的共同父类，声明了在具体构件中实现的业务方法，它的引入可以使客户端以一致的方式处理未被装饰的对象以及装饰之后的对象，实现客户端的透明操作。
- ConcreteComponent：具体构件类。它是抽象构件类的子类，用于定义具体的构件对象，实现了在抽象构件中声明的方法，装饰器可以给它增加额外的职责。
- Decorator：抽象装饰类。它也是抽象构件类的子类，用于给具体构件增加职责，但是具体职责在其子类中实现。它维护一个指向抽象构件对象的引用，通过该引用可以调用装饰之前构件对象的方法，并通过其子类扩展该方法，以达到装饰的目的。
- ConcreteDecorator：具体装饰类。它是抽象装饰类的子类，负责向构件添加新的职责。每一个具体装饰类都定义了一些新的行为，它可以调用在抽象装饰类中定义的方法，并可以增加新的方法用以扩充对象的行为。

注意事项：

1. 接口的一致性。装饰类的接口必须与它所装饰的构件类的接口保持一致，因此，所有的具体构件类必须有一个公共的父类。
2. 当只有一个具体装饰类时，可以省略抽象装饰类。
3. 保持抽象构件类的简单性。为了保证接口的一致性，构件类和装饰类必须有一个公共父类，保持这个类的简单性是很重要的。抽象构件类应集中于定义接口而不是存储数据，对数据表示的定义应延迟到子类中，否则抽象构件类会变得过于复杂和庞大而难以大量使用。

装饰模式的优点：

1. 比继承更灵活。与继承相比，装饰模式提供了更加灵活地向对象添加职责的方式。
2. 避免在层次结构高层的类有太多的特征。
3. 具体构件类和具体装饰类可以独立变化，用户可以根据需要增加新的具体构建类和具体装饰类，且原有类库代码无须改变，符合开闭原则。
4. 可以对一个对象进行多次装饰。

装饰模式的缺点：

1. 使用装饰模式时会产生很多小对象，如果过度使用，会让程序变得很复杂，影响程序的性能。
2. 比继承更加容易出错，遇到问题也难以发现。对于多次装饰的对象，调试时需要逐级排查，较为繁琐。

装饰模式的适用场合：

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. 当不能采用继承的方法进行扩充时。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式增长。另一种情况是，类定义被隐藏，或类定义不能用于生成子类。

4.3.2 示例：咖啡订单问题

问题描述：单品咖啡的种类有Espresso、ShortBlack、LongBlack、Decaf等，调料的种类有Milk、Soy、Chocolate等。客户在购买咖啡时，要先选择一个单品咖啡，再选择调料。可以不加调料，可以只加一种调料，也可以加多种调料，每种调料也可以加多份。

用装饰模式进行设计，如图 4.8 所示。

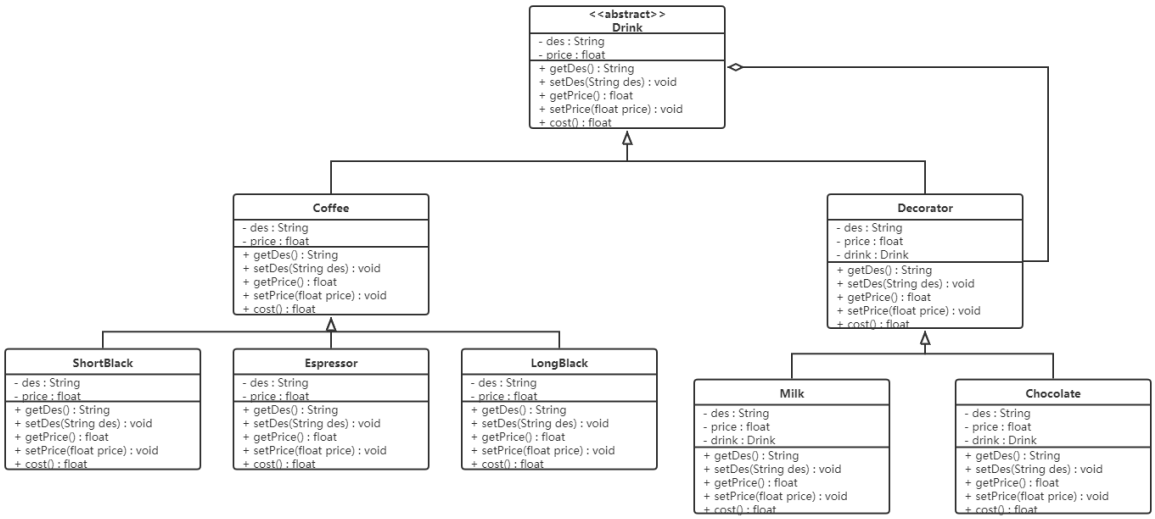

图4.8 咖啡订单问题：装饰模式

示例代码如下（为了简便，这里只写了一个具体咖啡类，其他咖啡同理）：

```
abstract class Drink
{
    private String des; // 描述
    private float price = 0.0f;

    public String getDes() { return des; }
    public void setDes(String des) { this.des = des; }
    public float getPrice() { return price; }
    public void setPrice(float price) { this.price = price; }

    public abstract float cost(); // 计算费用
}

class Coffee extends Drink
{
    public float cost()
    {
        return super.getPrice();
    }
}

class Espressor extends Coffee
{
    public Espressor()
    {
        setDes("意大利咖啡");
        setPrice(6.0f);
    }
}

class Decorator extends Drink
{
    private Drink drink;

    public Decorator(Drink drink)
    {
        this.drink = drink;
    }
```

```
39
40      public float cost()
41      {
42          return super.getPrice() + drink.cost();
43      }
44
45      public String getDes()
46      {
47          return super.getDes() + " " + drink.getDes();
48      }
49  }
50
51  class Chocolate extends Decorator
52  {
53      public Chocolate(Drink drink)
54      {
55          super(drink);
56          setDes("巧克力");
57          setPrice(3.0f);
58      }
59  }
60
61  class Milk extends Decorator
62  {
63      public Milk(Drink drink)
64      {
65          super(drink);
66          setDes("牛奶");
67          setPrice(2.0f);
68      }
69  }
70
71  public class Client
72  {
73      public static void main(String[] args)
74      {
75          // 点一份 Espressor
76          Drink order = new Espressor();
77          System.out.println("des = " + order.getDes());
78          System.out.println("cost = " + order.cost());
79          // 加入一份牛奶
80          order = new Milk(order);
81          System.out.println("des = " + order.getDes());
82          System.out.println("cost = " + order.cost());
83          // 加入一份巧克力
84          order = new Chocolate(order);
85          System.out.println("des = " + order.getDes());
86          System.out.println("cost = " + order.cost());
87      }
88  }
```

4.3.3 装饰模式应用实例

JDK 的 IO 流中，`FilterInputStream`是一个装饰类。图 4.9 展示了`InputStream`的类继承层次。

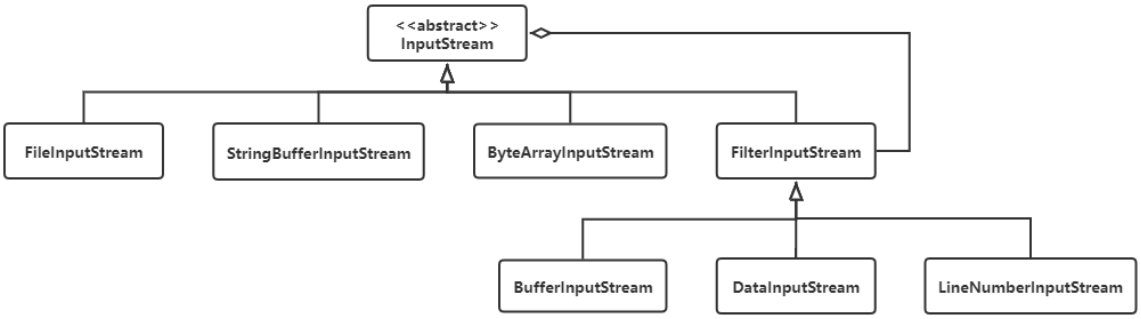

图4.9 InputStream类继承层次

其中`InputStream`抽象类相当于 Component，`FileInputStream`、`StringBufferInputStream`和`ByteArrayInputStream`相当于 ConcreteComponent，`FilterInputStream`相当于 Decorator，`BufferInputStream`、`DataInputStream`、`LineNumberInputStream`相当于 ConcreteDecorator。

代码如下：

```
public class FilterInputStream extends InputStream
{
    protected volatile InputStream in;

    protected FilterInputStream(InputStream in)
    {
        this.in = in;
    }

    public int read() throws IOException
    {
        return in.read();
    }

    // ...
}
```

4.4 组合模式（Composite Pattern）

组合模式：将对象组合成树形结构以表示 “部分-整体” 的层次结构。组合模式使得用户对单个对象和组合对象的访问具有一致性。

组合模式的结构如图 4.10 所示。

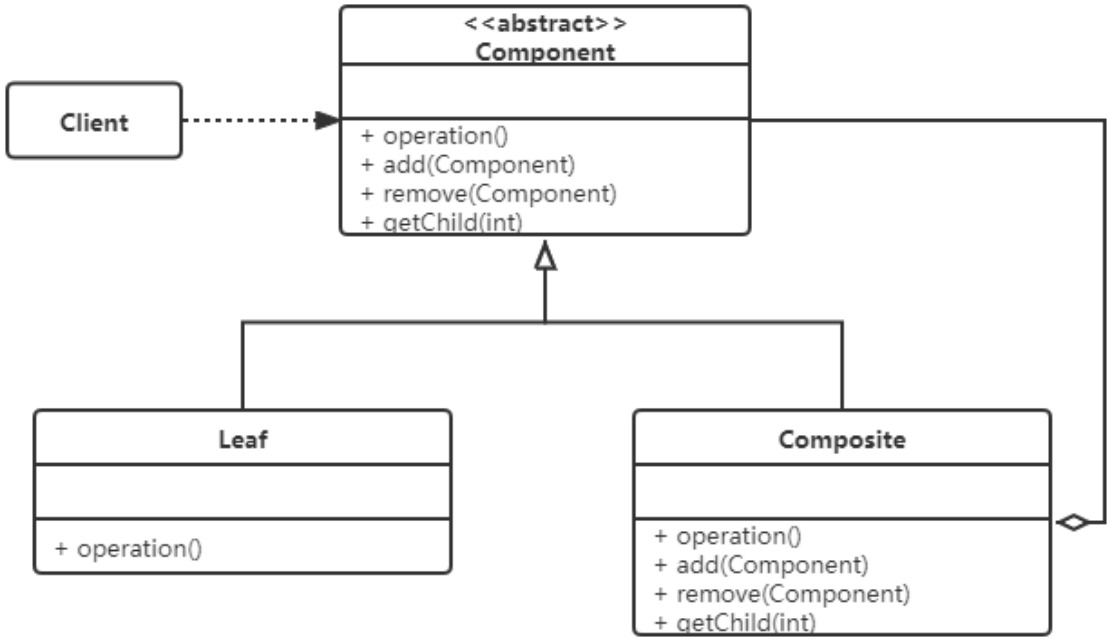

图4.10 组合模式

- Component：为组合中的对象声明接口，在适当情况下实现所有类共有接口的缺省行为，用于访问和管理子部件。
- Leaf：叶节点，没有子节点。
- Composite：非叶节点，用于存储子部件，实现与子部件有关的操作。

如果叶节点和非叶节点有很大的差异，就不适合使用组合模式。

组合模式的优点：

1. 简化客户端的操作。客户端可以一致地使用组合对象和单个对象，不需要知道处理的是组合对象还是单个对象。
2. 具有较强的扩展性，容易增加新的组件。
3. 方便创建出复杂的层次结构。

组合模式的适用场合：

1. 在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们。
2. 在一个使用面向对象语言开发的系统中需要处理一个树形结构。
3. 在一个系统中能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型。

4.5 外观模式（Facade Pattern）

外观模式：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这个子系统更加容易使用。

外观模式通过定义一个一致的接口，用以屏蔽子系统内部的细节，使得客户只需调用这个接口，而无需关心子系统的内部细节。

外观模式的适用场合：

1. 需要为一个复杂子系统提供一个简单接口时。
2. 客户程序与抽象类的实现部分之间存在很大的依赖性。引入外观类将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。
3. 当需要构建一个层次结构的子系统时，使用外观模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，可以让它们仅通过接口进行通信，从而简化了它们之间的依赖关系。

外观模式的优点：

1. 对客户屏蔽子系统组件，减少了客户处理的对象的数目，并使得子系统使用起来更加方便。
2. 实现了子系统与客户之间的松散耦合关系，使得子系统的组件变化不会影响到它的客户，提高子系统的独立性和可移植性。
3. 外观模式有助于建立层次结构系统，也有助于对对象之间的依赖关系分层。外观模式可以消除复杂的循环依赖问题。

4.6 享元模式（Flyweight Pattern）

享元模式：运用共享技术有效地支持大量细粒度的对象。

享元模式常用于系统底层开发，解决系统的性能问题。

享元模式能够解决重复对象的内存浪费问题。当系统中有大量相似对象时，不需要总是创建新对象，而是直接从缓冲池里拿，这样可以减少内存占用，提高效率。

池技术是享元模式的经典应用场景。Java `String`常量池、数据库连接池、缓冲池等都是享元模式的应用，享元模式是池技术的重要实现方式。

享元模式的结构如图 4.11 所示。

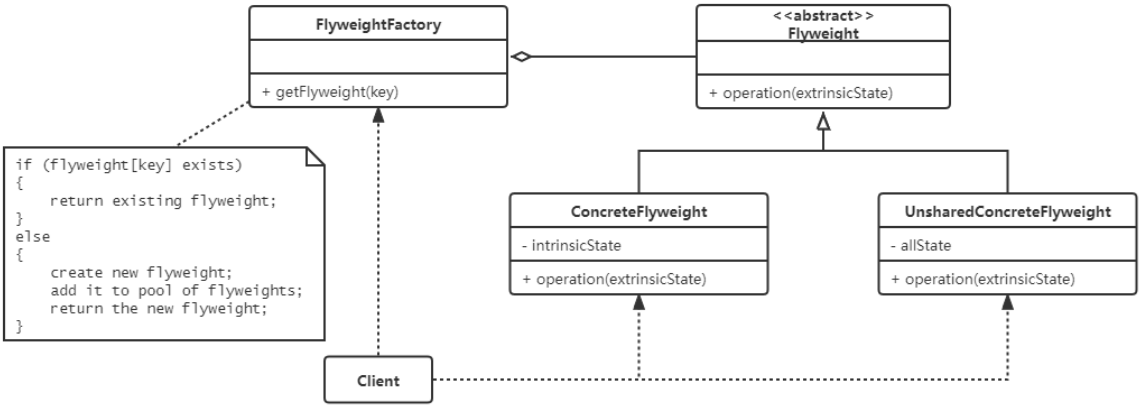

图4.11 享元模式

- Flyweight：抽象的享元角色，描述一个接口，同时定义出对象的外部状态和内部状态。
- ConcreteFlyweight：具体的享元角色，实现抽象角色定义的相关接口，并为内部状态增加存储空间。
- UnsharedConcreteFlyweight：不可共享的角色。
- FlyweightFactory：享元工厂类，创建并管理享元对象，同时提供获取享元对象的方法。

内部状态指对象共享出来的信息，存储在享元对象内部且不会随环境的改变而改变。外部状态指对象得以依赖的一个标记，是随环境改变而改变的、不可共享的状态。例如，围棋中有大量的棋子对象，棋子的颜色是不会随环境改变的，因此棋子的颜色是内部状态，存储在对象内部，可以共享；棋子的位置是随环境改变而改变的，每个棋子的位置都不同，因此棋子的位置是外部状态，不会存储在对象内部，不可共享。

享元对象所需的状态必定是内部的或外部的，内部状态存储于 ConcreteFlyweight 中，外部状态由客户程序存储或计算。当客户调用享元对象的方法时，要将外部状态传递给它。

用户不应直接对 ConcreteFlyweight 进行实例化，而只能从 FlyweightFactory 获得 ConcreteFlyweight 对象，这样可以保证对它们适当地进行共享。

当以下条件都成立时可以使用享元模式：

1. 一个应用程序使用了大量的对象。
2. 完全由于使用大量的对象造成很大的存储开销。
3. 对象的大多数状态都可变为外部状态。

4. 如果删除对象的外部状态，可以用相对较少的共享对象取代很多组对象。
5. 应用程序不依赖于对象标识。

享元模式的应用实例：JDK 中的`Integer`类使用了享元模式，代码如下：

```
public final class Integer extends Number implements Comparable<Integer>
{
    //...

    // 缓冲池内部类，创建了 -128 到 127 的 Integer 对象
    private static class IntegerCache
    {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static
        {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =

 sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null)
            {
                try
                {
                    int i = parseInt(integerCacheHighPropValue);
                    i = Math.max(i, 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
                }
                catch( NumberFormatException nfe)
                {
                    // If the property cannot be parsed into an int, ignore
it.
                }
            }
            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }

    public static Integer valueOf(int i)
    {
        // 如果数值在 [-128, 127] 的范围内，则返回缓冲池中的对象
        // 否则，创建新对象并返回
        if (i >= IntegerCache.low && i <= IntegerCache.high)
```

```
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }

    //...
}
```

下面的代码体现了这一特性：

```
Integer x = Integer.valueOf(127); // 从缓冲池中获得对象
Integer y = new Integer(127); // 创建新对象
Integer z = Integer.valueOf(127); // 从缓冲池中获得对象
Integer w = new Integer(127); // 创建新对象

System.out.println(x == y); // false
System.out.println(x == z); // true
System.out.println(w == x); // false
System.out.println(w == y); // false

Integer x1 = Integer.valueOf(200); // 创建新对象
Integer x2 = Integer.valueOf(200); // 创建新对象
System.out.println(x1 == x2); // false
```

由于这一特性，当数值范围在 -128 到 127 之间时，应该使用`valueOf`方法得到`Integer`对象，这样效率更高、消耗内存更少。

4.7 代理模式（Proxy Pattern）

4.7.1 代理模式介绍

代理模式：为其他对象提供一种代理以控制对这个对象的访问。

代理模式的结构如图 4.12 所示。

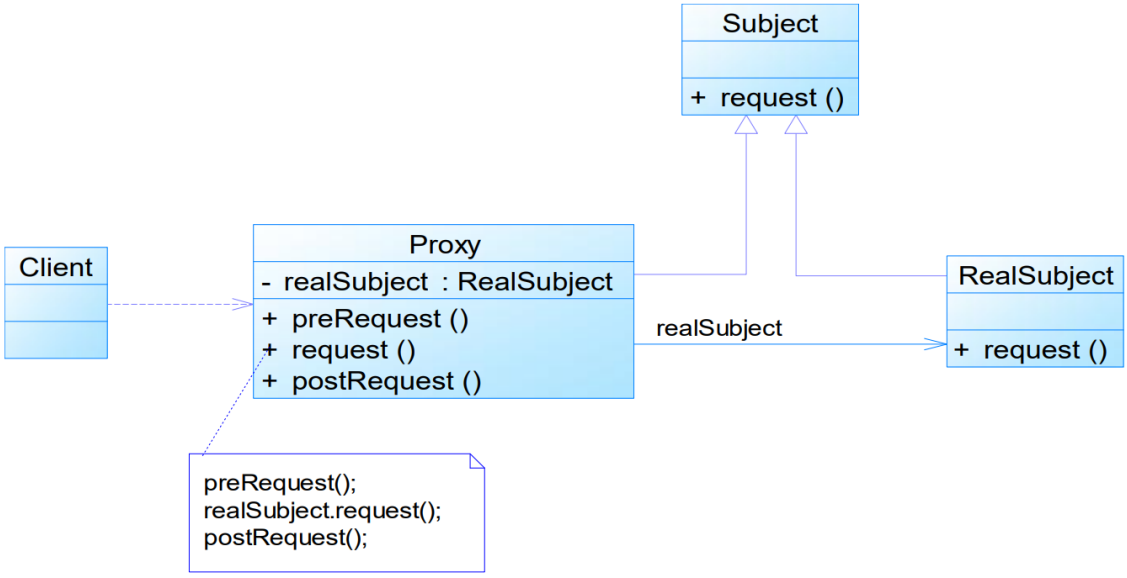

图4.12 代理模式

代理模式的主要种类：

1. 远程代理（Remote Proxy）：为一个位于不同地址空间的对象提供一个本地的代理对象。
2. 虚拟代理（Virtual Proxy）：如果需要创建一个资源消耗较大的对象，先创建一个消耗相对较小的对象来表示，真实对象只在需要时才会被真正创建。

3. 保护代理（Protect Proxy）：控制对一个对象的访问，可以给不同的用户提供不同级别的使用权限。
4. 缓冲代理（Cache Proxy）：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
5. 智能引用代理（Smart Reference Proxy）：当一个对象被引用时，提供一些额外的操作，例如将对象被调用的次数记录下来等。
6. 防火墙代理（Firewall Proxy）：控制网络资源的访问，保护主题免于 “坏客户” 的侵害。
7. 同步代理（Synchronization Proxy）：在多线程的情况下为主题提供安全的访问，完成线程间的同步工作。
8. 写入时复制代理（Copy-On-Write Proxy）：用来控制对象的复制。其方法是延迟对象的复制，只有当对象被修改时才进行复制。这是虚拟代理的变体。

代理模式的形式有：静态代理、动态代理、Cglib 代理。

4.7.2 静态代理

静态代理在使用时，需要定义接口或父类，被代理对象（即目标对象）与代理对象一起实现相同的接口或者是继承相同的父类，通过调用代理对象的方法来调用目标对象的方法。

例如：

```
// 老师接口
interface ITeacher
{
    void teach();
}

// 目标对象类
class Teacher implements ITeacher
{
    public void teach()
    {
        System.out.println("老师上课");
    }
}

// 代理对象类
class TeacherProxy implements ITeacher
{
    private ITeacher target; // 目标对象

    public TeacherProxy(Iteacher target)
    {
        this.target = target;
    }

    public void teach()
    {
        System.out.println("代理开始"); // 扩展功能
        target.teach();
        System.out.println("代理结束"); // 扩展功能
    }
}

public class Client
{
    public static void main(String[] args)
```

```
    {
        // 创建目标对象
        ITeacher teacher = new Teacher();
        // 创建代理对象，同时将目标对象传入
        ITeacher teacherProxy = new TeacherProxy(teacher);
        // 通过代理对象调用目标对象的方法
        teacherProxy.teach();
    }
}
```

静态代理的优点：在不修改目标对象功能的前提下，能通过代理对象对目标的功能进行扩展。

静态代理的缺点：因为代理对象和目标对象实现同样的接口，所以会有很多代理类。一旦接口增加方法，目标对象与代理对象都要维护。

4.7.3 动态代理

在动态代理中，目标对象仍然要实现接口，但代理对象不需要实现接口。代理对象的生成是通过 JDK 的 API，动态地在内存中构建代理对象。

JDK 中生成代理对象的 API：代理类在 `java.lang.reflect.Proxy` 包中，JDK 实现代理只需要调用静态 `newProxyInstance` 方法。`newProxyInstance` 方法的签名如下：

```
/* java.lang.reflect.Proxy */
public static Object newProxyInstance(ClassLoader loader,
                                       Class<?>[] interfaces,
                                       InvocationHandler h)
// ClassLoader loader：当前目标对象使用的类加载器
// Class<?>[] interfaces：目标对象实现的接口类型
// InvocationHandler h：事件处理，执行目标对象的方法时会触发事件处理器方法
```

例如：

```
// 老师接口
interface ITeacher
{
    void teach();
}

// 目标对象类
class Teacher implements ITeacher
{
    public void teach()
    {
        System.out.println("老师上课");
    }
}

class ProxyFactory
{
    private Object target; // 目标对象

    // 通过构造器对目标对象进行初始化
    public ProxyFactory(Object target)
    {
        this.target = target;
```

```
    }

    // 返回代理对象
    public Object getProxyInstance()
    {
        return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method,
Object[] args)
                    {
                        System.out.println("动态代理开始");
                        // 通过反射机制调用目标对象的方法
                        Object returnVal = method.invoke(target, args);
                        System.out.println("动态代理结束");
                        return returnVal;
                    }
                });
    }
}

public class Client
{
    public static void main(String[] args)
    {
        // 创建目标对象
        ITeacher teacher = new Teacher();
        // 创建代理对象
        ITeacher proxyInstance = (ITeacher) new
ProxyFactory(teacher).getProxyInstance();
        System.out.println(proxyInstance.getClass()); // class
com.sun.proxy.$Proxy0
        // 通过代理对象调用目标对象的方法
        proxyInstance.teach();
    }
}
```

4.7.4 Cglib代理

静态代理和动态代理都要求目标对象实现一个接口，而 Cglib 代理没有这个要求。Cglib 代理也叫做子类代理，它是在内存中构建一个子类对象从而实现对目标对象的功能扩展。

Cglib 是一个强大的、高性能的代码生成包，它可以在运行时扩展 Java 类与实现 Java 接口。Cglib 包的底层使用字节码处理框架 ASM 来转换字节码并生成新的类。

注意事项：

1. 需要引入相关的 jar 包：asm.jar、asm-commons.jar、asm-tree.jar、cglib-2.2.jar。
2. 目标对象类不能为 `final`，因为需要构建目标对象的子类对象，否则抛出 `java.lang.IllegalArgumentException`。
3. 如果目标对象的方法为 `final` 或 `static`，就不会拦截，即不会执行该方法。

例如：

```
// 目标对象类
class Teacher implements ITeacher
```

```
{
    public void teach()
    {
        System.out.println("老师上课");
    }
}

class ProxyFactory implements MethodInterceptor // MethodInterceptor接口在cglib包中
{
    private Object target; // 目标对象

    // 通过构造器对目标对象进行初始化
    public ProxyFactory(Object target)
    {
        this.target = target;
    }

    // 返回代理对象
    public Object getProxyInstance()
    {
        // 创建一个工具类对象
        Enhancer enhancer = new Enhancer(); // Enhancer类在cglib包中
        // 设置父类
        enhancer.setSuperclass(target.getClass());
        // 设置回调函数
        enhancer.setCallback(this);
        // 创建子类对象，即代理对象
        return enhancer.create();
    }

    // 实现intercept方法，在intercept方法中会调用目标对象的方法
    @Override
    public void intercept(Object arg0, Method method, Object[] args, MethodProxy arg3)
        throws Throwable
    {
        System.out.println("Cglib代理开始");
        Object returnVal = method.invoke(target, args);
        System.out.println("Cglib代理结束");
        return returnVal;
    }
}

public class Client
{
    public static void main(String[] args)
    {
        // 创建目标对象
        Teacher teacher = new Teacher();
        // 创建代理对象
        Teacher proxyInstance = (Teacher) new ProxyFactory(teacher).getProxyInstance();
        // 通过代理对象调用目标对象的方法，触发intercept方法
        proxyInstance.teach();
    }
}
```

5 行为型模式

行为型模式涉及算法和对象间职责的分配。行为型模式不仅描述对象或类的模式，还描述它们之间的通信模式。这些模式刻画了在运行时难以跟踪的复杂的控制流，它们将你的注意力从控制流转移到对象间的联系上来。

类行为型模式使用继承机制在类间分派行为。模板方法模式和解释器模式属于类行为型模式。

对象行为型模式使用对象组合而不是继承。

5.1 模板方法模式（Template Method Pattern）

5.1.1 模板方法模式介绍

模板方法模式：定义一个操作中的算法的骨架，而将一些步骤的实现延迟到子类中。模板方法模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

模板方法模式的结构如图 5.1 所示。

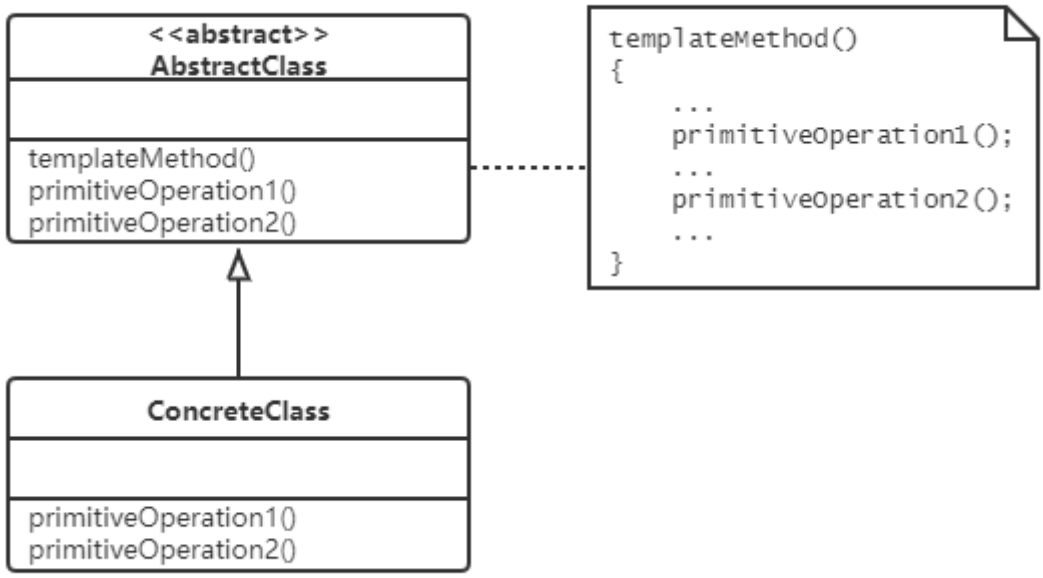

图5.1 模板方法模式

- AbstractClass：定义抽象的原语操作（primitive operation）。实现一个模板方法（templateMethod），定义一个算法的骨架。该模板方法不仅调用原语操作，也调用其他可能的操作。
- ConcreteClass：实现原语操作，以完成算法中的特定步骤。

模板方法中可以调用的方法：

1. 具体的方法（ConcreteClass 或对客户类的操作）。
2. AbstractClass 中的具体方法。
3. 抽象方法。
4. 工厂方法。
5. 钩子方法。

模板方法模式的优点：

1. 实现了最大化代码复用，父类的模板方法和已实现的某些步骤会被子类继承并直接使用。
2. 既统一了算法，也提供了很大的灵活性。父类的模板方法保证算法的结构保持不变，同时由子类提供部分步骤的实现。

模板方法的缺点：每一个不同的实现都需要定义一个子类，导致类的个数增加，使得系统更加庞大。

模板方法的适用场合：

1. 要完成某个过程，该过程要执行一系列步骤，这一系列的步骤基本保持不变，只是个别步骤的实现可能不同。将不变的部分由父类实现，将可变的行为留给子类来实现。
2. 各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。
3. 控制子类扩展。模板方法只在特定点调用钩子方法，这样就只允许在这些点进行扩展。

5.1.2 示例：豆浆制作问题

制作豆浆的流程为：选材→添加配料→浸泡→放到豆浆机中打碎。通过添加不同的配料，可以制作出不同口味的豆浆（如红豆豆浆、花生豆浆等）。选材、浸泡、放到豆浆机打碎这几个步骤对于制作每种口味的豆浆都是一样的。

示例代码如下：

```
// 抽象类，表示豆浆
abstract class SoyaMilk
{
    // 模板方法：制作豆浆
    public final void make()
    {
        select();
        add();
        soak();
        beat();
    }

    // 选材
    protected void select()
    {
        System.out.println("选择上好的新鲜黄豆");
    }

    protected abstract void add(); // 添加配料

    // 浸泡
    protected void soak()
    {
        System.out.println("黄豆和配料一起浸泡");
    }

    // 打碎
    protected void beat()
    {
        System.out.println("黄豆和配料放入豆浆机打碎");
    }
}

// 红豆豆浆
class RedBeanSoyaMilk extends SoyaMilk
{
    protected void add()
    {
        System.out.println("加入红豆");
    }
```

```
}

// 花生豆浆
class PeanutSoyaMilk extends SoyaMilk
{
    protected void add()
    {
        System.out.println("加入花生");
    }
}

public class Client
{
    public static void main(String[] args)
    {
        System.out.println("制作红豆豆浆");
        SoyaMilk redBeanSoyaMilk = new RedBeanSoyaMilk();
        redBeanSoyaMilk.make();

        System.out.println("制作花生豆浆");
        SoyaMilk peanutSoyaMilk = new PeanutSoyaMilk();
        peanutSoyaMilk.make();
    }
}
```

5.1.3 钩子方法

在抽象类中可以定义一个方法，它提供了缺省的行为，子类可以视情况选择是否扩展，这样的方法称为钩子方法。钩子方法在缺省情况下通常是空操作。

例如，在上面的豆浆制作问题中，可以将 `add` 方法作为钩子方法，在 `SoyaMilk` 类中提供空操作，表示默认情况下不添加配料，子类可以根据是否添加配料选择是否重写该方法。

注意：在抽象类中要指明哪些方法是钩子方法（可以重写），哪些方法是抽象方法（必须重写）。

5.2 命令模式（Command Pattern）

5.2.1 命令模式介绍

命令模式：将一个请求封装为一个对象，以便使用不同的请求对客户进行参数化。

命令模式的本质是对请求进行封装，请求本身成为一个对象，这个对象和其他对象一样可以被存储和传递。

命令模式的结构如图 5.2 所示。

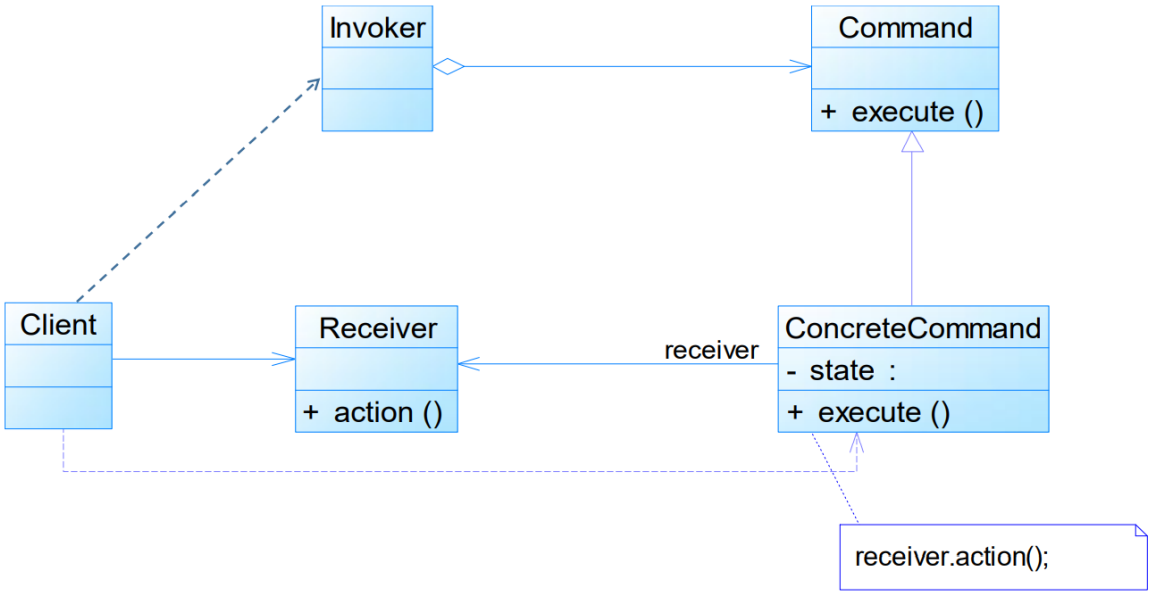

图5.2 命令模式

- Command：抽象命令类，声明执行操作的接口。
- ConcreteCommand：具体命令类，实现了在抽象命令类中声明的方法。将一个接收者对象与一个动作绑定，调用接收者相应的操作来实现 execute。
- Invoker：调用者，请求发送者，要求命令对象执行请求。
- Receiver：接收者，知道如何实施与执行一个请求相关的操作，具体实现对请求的业务处理。任何类都可能成为接收者。

关键代码示意：

```
abstract class Command
{
    public abstract void execute();
}

class ConcreteCommand extends Command
{
    private Receiver receiver; // 维护一个对接收者对象的引用

    public ConcreteCommand(Receiver receiver)
    {
        this.receiver = receiver;
    }

    public void execute()
    {
        receiver.action(); // 调用接收者对象的方法
    }
}

class Invoker
{
    private Command command; // 维护一个对命令对象的引用

    public Invoker(Command command)
    {
        this.command = command;
    }

```

```
    public void setCommand(Command command)
    {
        this.command = command;
    }

    public void call()
    {
        command.execute(); // 调用命令对象的方法
    }
}

class Receiver
{
    public void action()
    {
        // 具体操作
    }
}
```

可以提供空命令，即 execute 方法为空实现的命令类，这样可以省去判空操作。

命令模式的优点：

1. 将请求发送者和请求接收者解耦，让对象之间的调用关系更加灵活。请求发送者不必知道关于被请求的操作或请求接收者的任何信息。
2. 增加新的 Command 很容易，无须改变已有的类。

命令模式的缺点：可能导致过多的具体命令类，增加了系统的复杂度。

5.2.2 命令模式扩展：命令队列

有时一个请求发送者不止发送一个请求，不止一个请求接收者产生响应，此时可以把多个请求排队，当发送请求时，请求接收者将逐个执行业务方法，完成对请求的处理。

命令队列的实现方法有很多，其中最常用、灵活性最好的一种方法是增加一个 CommandQueue 类，由该类来负责存储多个命令对象，而不同的命令对象可以对于不同的请求接收者。示例代码如下：

```
class CommandQueue
{
    private ArrayList<Command> commands = new ArrayList<Command>();

    public void addCommand(Command command)
    {
        commands.add(command);
    }

    public void removeCommand(Command command)
    {
        commands.remove(command);
    }

    public void execute()
    {
        for (Command command : commands)
        {
            command.execute();
        }
    }
```

```
}

class Invoker // 针对 CommandQueue 编程
{
    private CommandQueue commandQueue;

    public Invoker(CommandQueue commandQueue)
    {
        this.commandQueue = commandQueue;
    }

    public void setCommandQueue(CommandQueue commandQueue)
    {
        this.commandQueue = commandQueue;
    }

    public void call
    {
        commandQueue.execute();
    }
}
```

命令队列可以用于设计批处理应用程序，对一组命令对象进行批量处理，当一个发送者发送请求后，将有一系列接收者对请求做出响应。如果请求接收者的接收次序没有严格要求，还可以使用多线程来并发执行，从而提高程序的执行效率。

5.2.3 命令模式扩展：请求日志

请求日志就是将请求的历史记录保存下来，通常以日志文件的形式永久存储。请求日志的常用功能如下：

1. 日志文件可以为系统提供恢复机制，一旦系统发生故障，可以让系统恢复到某个特定的状态。
2. 请求日志可以用于实现批处理。
3. 可以将命令队列中的所有命令对象存储在一个日志文件中，每执行一个命令就从日志文件中删除一个命令对象，防止因为断电或系统重启等原因造成请求丢失，而且可以避免重新发送全部请求时造成某些命令的重复执行。

例如：通过一个可视化界面对配置文件进行增删改等操作，把对配置文件的操作请求记录在日志文件中，设计方案如图 5.3 所示。

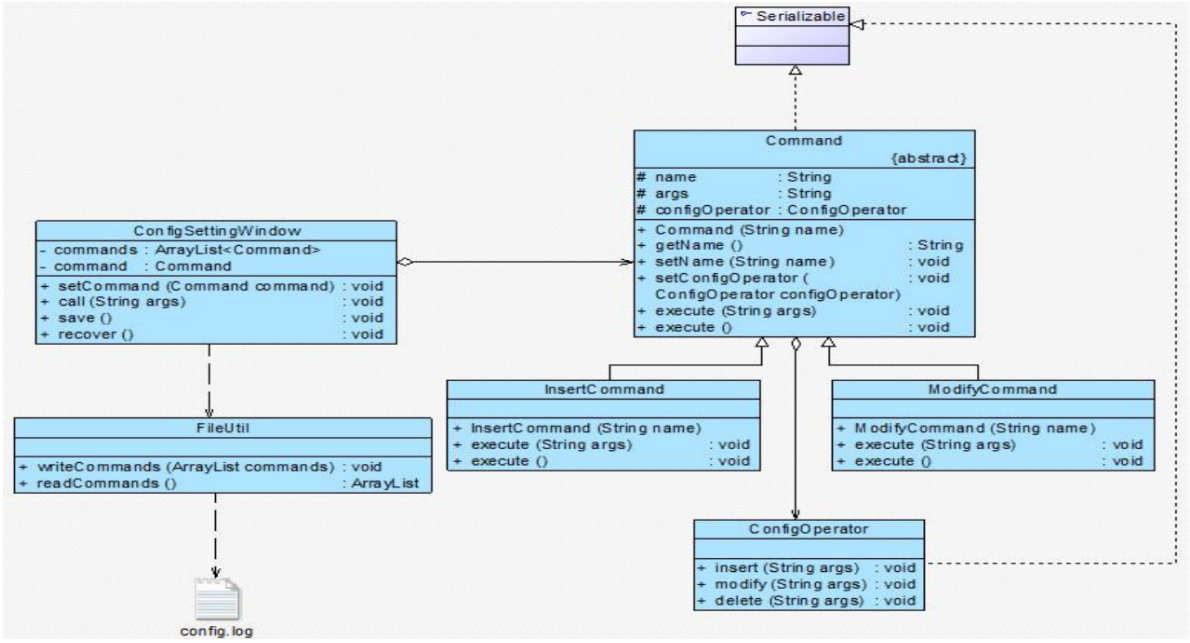

图5.3 请求日志

5.2.4 命令模式扩展：撤销操作

如果需要撤销操作，可以通过在命令类中增加一个逆向操作来实现。也可以使用备忘录模式，通过保存对象的历史状态来实现撤销。

5.2.5 命令模式扩展：宏命令

宏命令又称为组合命令，它是命令模式和组合模式联合的产物。

宏命令是一个具体命令类，它拥有一个集合属性，在该集合中包含对其他命令对象的引用。通常宏命令不直接与请求接收者交互，而是通过它的成员来调用接收者的方法。当调用宏命令的`execute`方法时，将递归调用它所包含的每个成员命令的`execute`方法。宏命令的成员可以是简单命令，也可以是宏命令。执行一个宏命令将触发多个具体命令的执行，从而实现对命令的批处理。

宏命令的结构如图 5.4 所示。

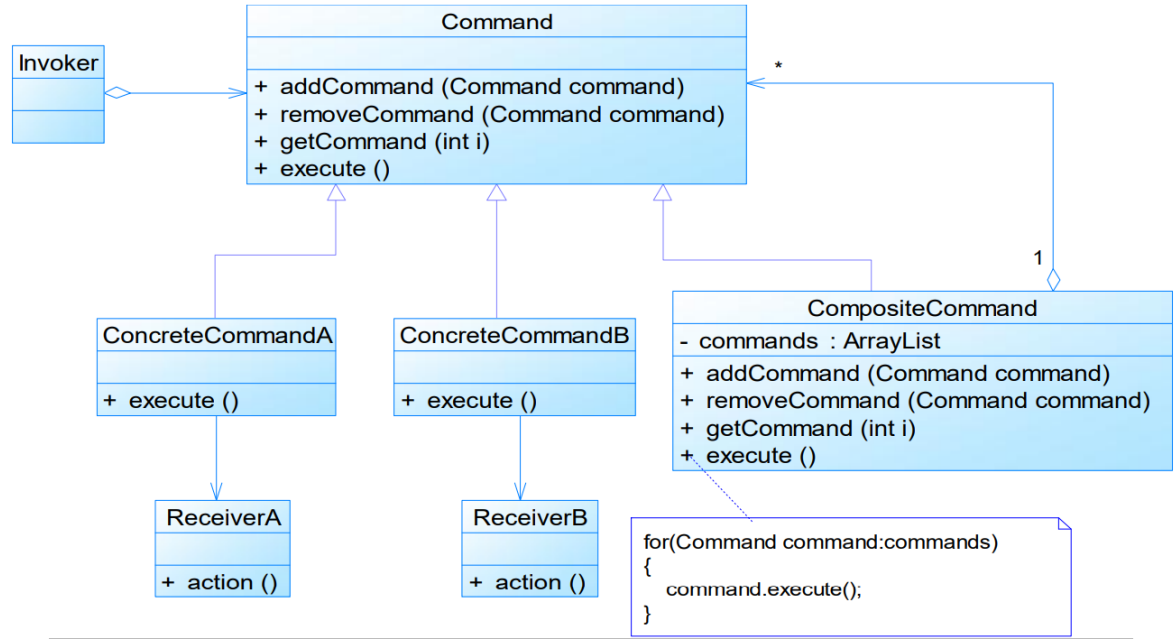

图5.4 宏命令

5.3 访问者模式（Visitor Pattern）

访问者模式：封装一些作用于某种对象结构的各元素的操作，它可以在不改变对象结构的前提下定义作用于这些元素的新的操作。

访问者模式的基本工作原理：在被访问的类里面加一个接待访问者的接口。

访问者模式的结构如图 5.5 所示。

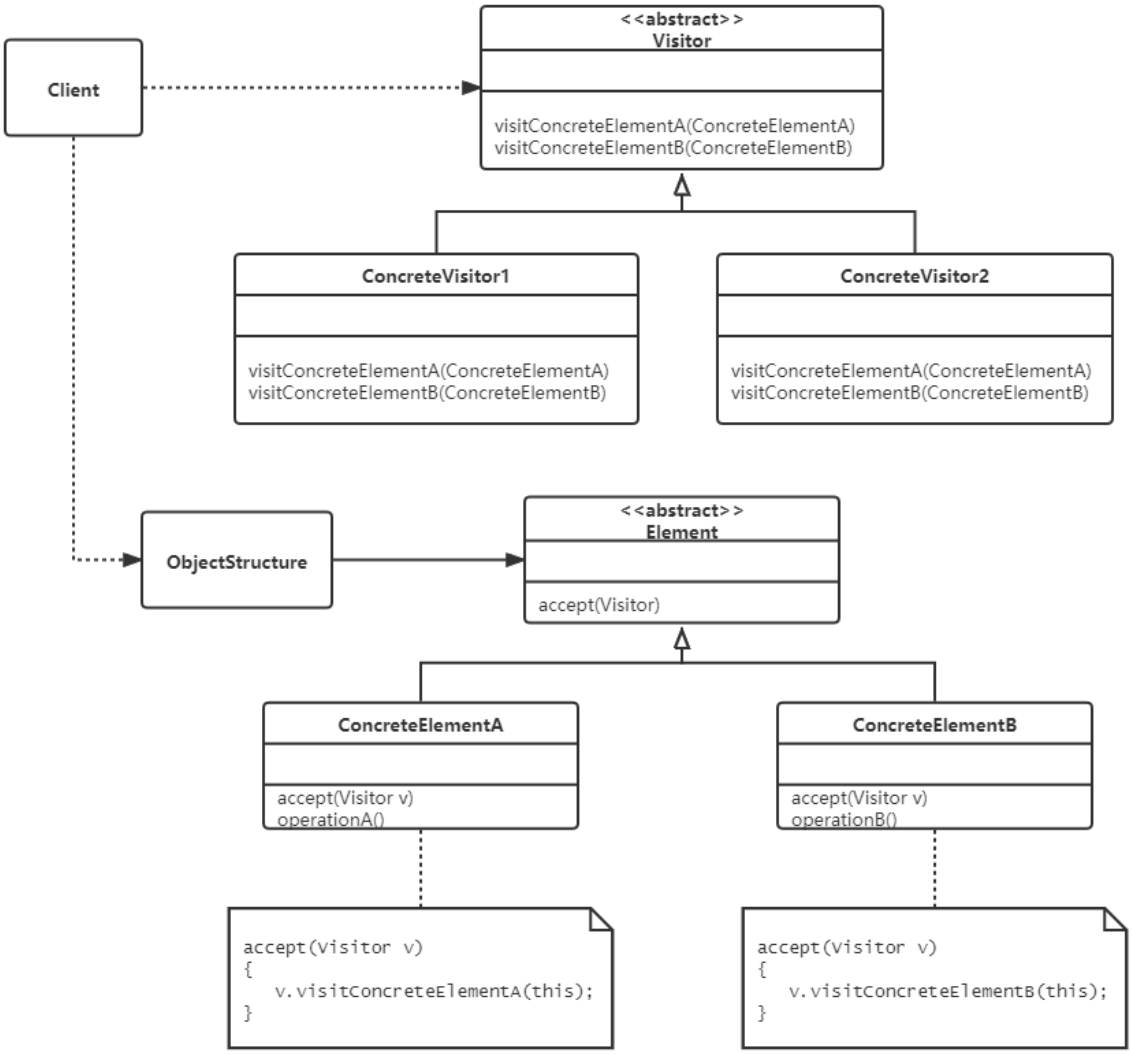

图5.5 访问者模式

- Visitor：抽象访问者，为对象结构中的每一个类声明一个访问操作。
- ConcreteVisitor：具体访问者，实现每个由 Visitor 声明的操作。
- ObjectStructure：对象结构，能枚举它的元素，可以提供一个高层的接口以允许访问者访问它的元素。
- Element：元素，定义一个 accept 操作，接收一个访问者作为参数。
- ConcreteElement：具体元素，实现 accept 操作。

访问者模式的优点：

1. 将数据结构与数据操作分离，符合单一职责原则，让程序具有优秀的扩展性和灵活性。
2. 易于增加新的操作，仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。
3. 访问者集中相关的操作而分离无关的操作。相关的行为集中在一个访问者中，无关的行为分散在不同的访问者中，这既简化了这些元素的类，也简化了在访问者中定义的算法。

访问者模式的缺点：

1. 具体元素对访问者公布细节，违反了迪米特原则，破坏封装，造成具体元素变更比较困难。
2. 违背了依赖倒转原则，访问者依赖的是具体元素，而不是抽象元素。

3. 增加新的具体元素比较困难。每添加一个新的具体元素都要在 Visitor 中添加一个新的抽象方法，并在每一个 ConcreteVisitor 中实现相应的操作。

访问者模式的适用场合：

1. 需要对一个对象结构中的对象进行很多不同并且不相关的操作，同时避免让这些操作 “污染” 这些对象的类。
2. 一个对象结构包含很多对象，它们有不同的接口，需要对这些对象实施一些依赖于其具体类的操作。
3. 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。

5.4 迭代器模式（Iterator Pattern）

迭代器模式：提供一种方法顺序访问一个聚合对象中的各个元素，而不需要暴露该聚合对象的内部结构。

迭代器模式的结构如图 5.6 所示。

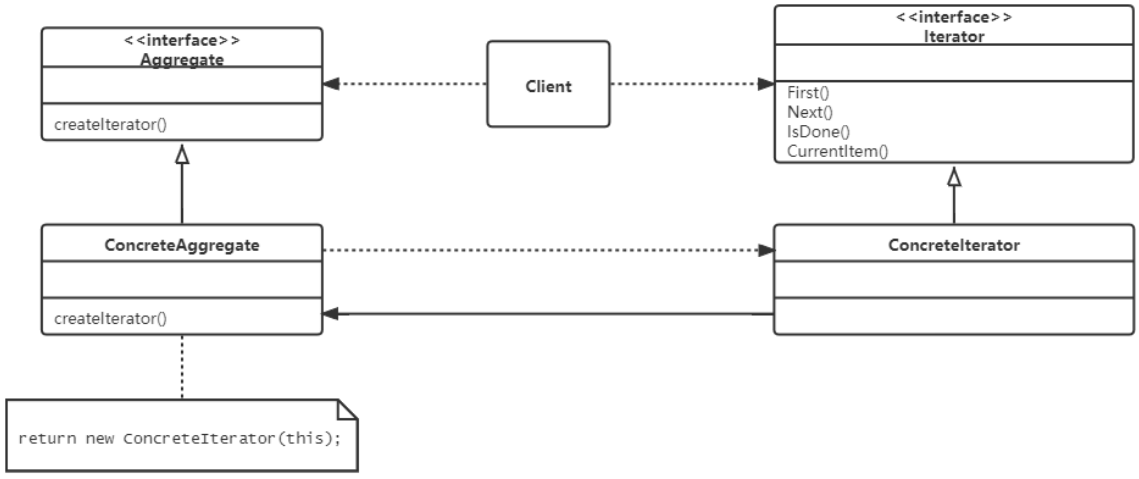

图5.6 迭代器模式

- Iterator：迭代器，定义访问和遍历聚合元素的接口。
- ConcreteIterator：具体迭代器，实现迭代器接口，完成对聚合对象的遍历，记录遍历的当前位置。
- Aggregate：聚合接口，定义创建相应迭代器的接口。
- ConcreteAggregate：具体聚合，实现创建相应迭代器的接口。

迭代器模式的优点：

1. 提供一个统一的方法遍历对象，客户不用考虑聚合的类型。
2. 隐藏了聚合的内部结构。
3. 将聚合与迭代器分开，符合单一职责原则。
4. 支持以不同的方式遍历一个聚合。
5. 在同一个聚合上可以有多个遍历。

迭代器模式的缺点：每个聚合对象都需要一个迭代器，使系统中类的数量增加，不易于管理。

迭代器模式的适用场合：

1. 访问一个聚合对象的内容而无须暴露它的内部表示。
2. 支持对聚合对象的多种遍历方式。
3. 为遍历不同的聚合结构提供一个统一的接口。

迭代器模式的应用实例：JDK 中的`ArrayList`类使用了迭代器模式，如图 5.7 所示。

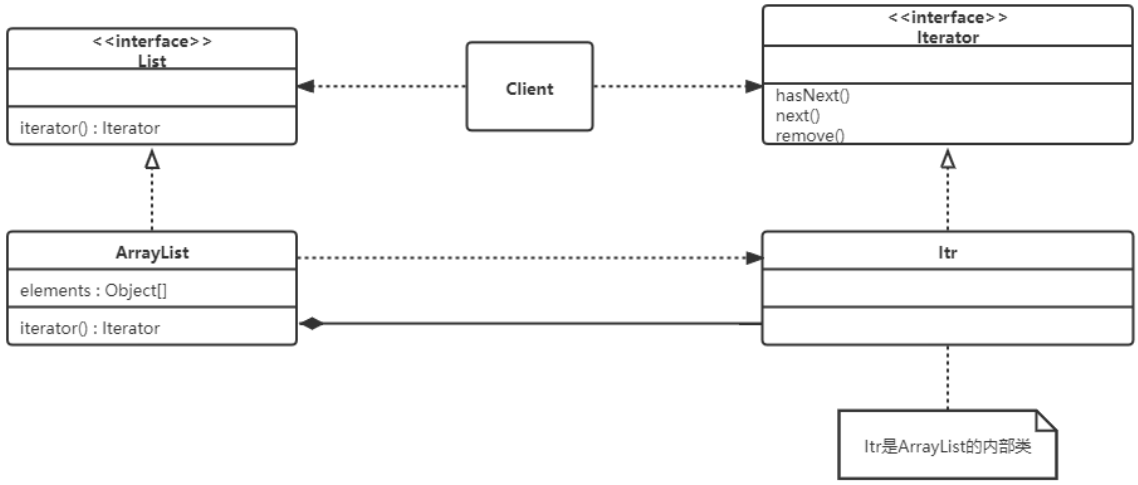

图5.7 迭代器模式应用实例

5.5 观察者模式（Observer Pattern）

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

观察者模式的结构如图 5.8 所示。

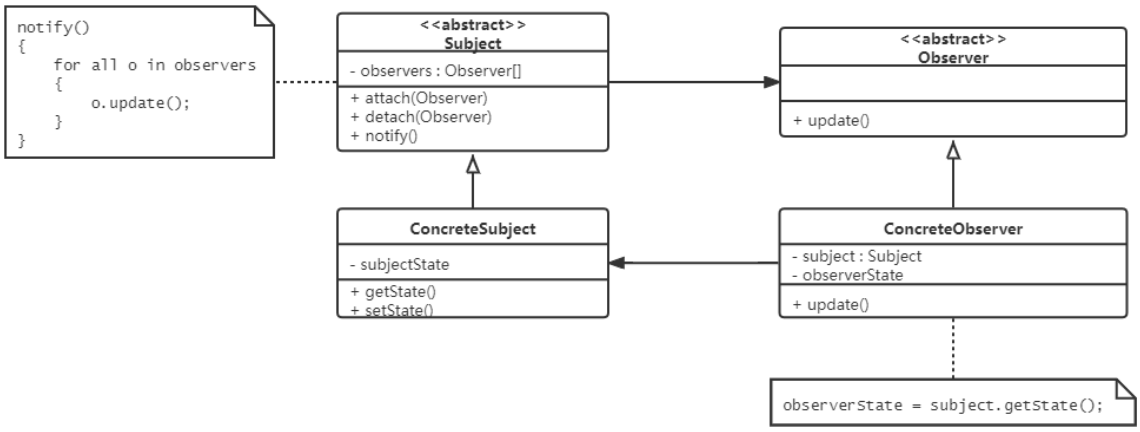

图5.8 观察者模式

- Subject：观察者观察的目标，提供注册和删除观察者对象的接口。
- ConcreteSubject：将有关状态存入各 ConcreteSubject 对象，当状态改变时，向各个观察者发出通知。
- Observer：抽象观察者，定义一个更新接口。
- ConcreteObserver：具体观察者，维护一个指向 ConcreteSubject 对象的引用，存储有关状态，实现更新接口，使自身状态与目标的状态保持一致。

观察者模式的优点：

1. 可以独立地改变目标和观察者，也可以在不改动目标和其他观察者的前提下增加观察者，符合开闭原则。
2. 目标和观察者之间抽象耦合。一个目标只知道它有观察者，但不知道观察者属于哪个具体的类，这样目标和观察者之间的耦合是抽象的和最小的。
3. 支持广播请求。

观察者模式的适用场合：

1. 一个抽象模型有两个方面，其中一个方面依赖于另一方面。将这二者封装在独立的对象中，以使它们可以各自独立地改变和复用。

2. 对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象需要改变。
3. 一个对象必须通知其他对象，而它又不能假定其他对象是谁，即对象之间是松散耦合的。

观察者模式的应用实例：JDK 中的`java.util.Observable`类使用了观察者模式，代码如下：

```
public class Observable // 具体目标
{
    private boolean changed = false;
    private Vector<Observer> obs;

    public synchronized void addObserver(Observer o)
    {
        if (o == null)
        {
            throw new NullPointerException();
        }
        if (!obs.contains(o))
        {
            obs.addElement(o);
        }
    }

    public synchronized void deleteObserver(Observer o)
    {
        obs.removeElement(o);
    }

    public void notifyObservers(Object arg)
    {
        Object[] arrLocal;

        synchronized (this)
        {
            if (!changes)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }

        for (int i = arrLocal.length - 1; i >= 0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }

    public void notifyObservers()
    {
        notifyObservers(null);
    }
}

public interface Observer // 观察者接口
{
    void update(Observable o, Object arg);
}
```

5.6 中介者模式（Mediator Pattern）

中介者模式：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

中介者模式的结构如图 5.9 所示。

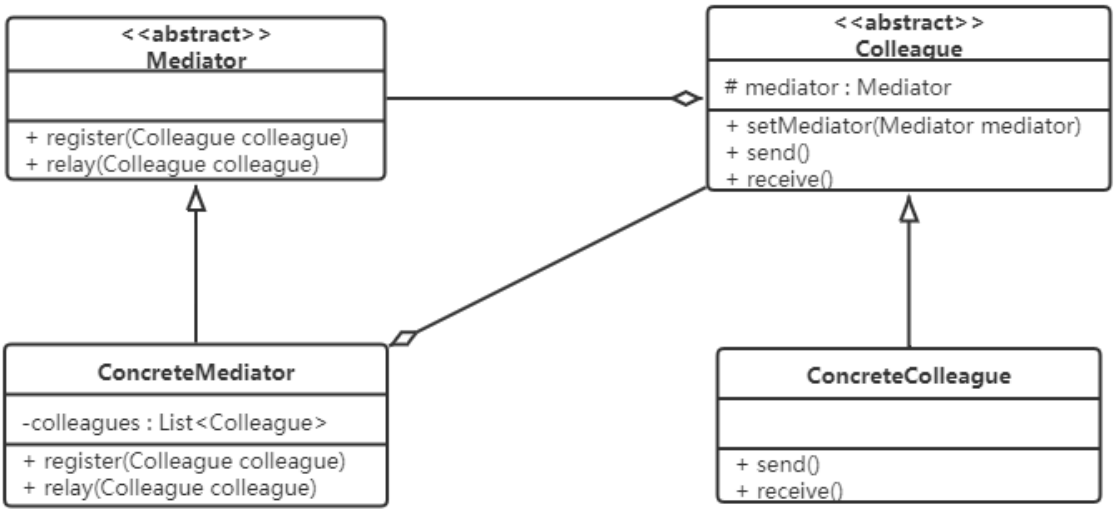

图5.9 中介者模式

- Mediator：抽象中介者，定义了同事对象注册和转发同事对象信息的接口。
- ConcreteMediator：具体中介者，了解并维护它的同事，通过协调各同事对象实现协作行为。
- Colleague：抽象同事类，保存中介者对象，提供同事对象交互的接口。
- ConcreteColleague：具体同事类，每个同事类都知道它的中介者对象。每个同事对象在需要与其他同事通信时，与它的中介者通信，由中介者对象负责后续的交互。每个同事只知道自己的行为，而不了解其他同事类的行为。

中介者模式的优点：

1. 减少了子类的数量。中介者将原本分布于多个对象间的行为集中在一起，改变这些行为只需要生成中介者类的子类即可。
2. 中介者模式将各同事类解耦，可以独立地改变和复用各同事类和中介者类。减少类间依赖，降低了耦合，符合迪米特原则。
3. 简化了对象协议。用中介者类和同事类间的一对多关系来代替多对多关系，一对多关系更易于理解、维护和扩展。
4. 对对象如何协作进行了抽象。将中介作为一个独立的概念并将其封装在一个对象中，将注意力从对象各自的行为转移到它们之间的交互上来，这有助于弄清楚一个系统中的对象是如何交互的。

中介者模式的缺点：

1. 中介者承担了较多的责任，一旦中介者出现了问题，整个系统都会受到影响。
2. 将交互的复杂性变为中介者的复杂性，如果设计不当，可能使中介者本身变得过于复杂而难以维护。

中介者模式的适用场合：

1. 一组对象以定义良好但复杂的方式进行通信，产生的相互依赖关系结构混乱且难以理解。
2. 一个对象引用其他很多对象并且直接与这些对象通信，导致难以复用该对象。
3. 想定制一个分布在多个类中的行为，而又不想生成太多的子类。

5.7 备忘录模式（Memento Pattern）

备忘录模式：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。这样以后就可以将该对象恢复到原先保存的状态。

备忘录模式的结构如图 5.10 所示。

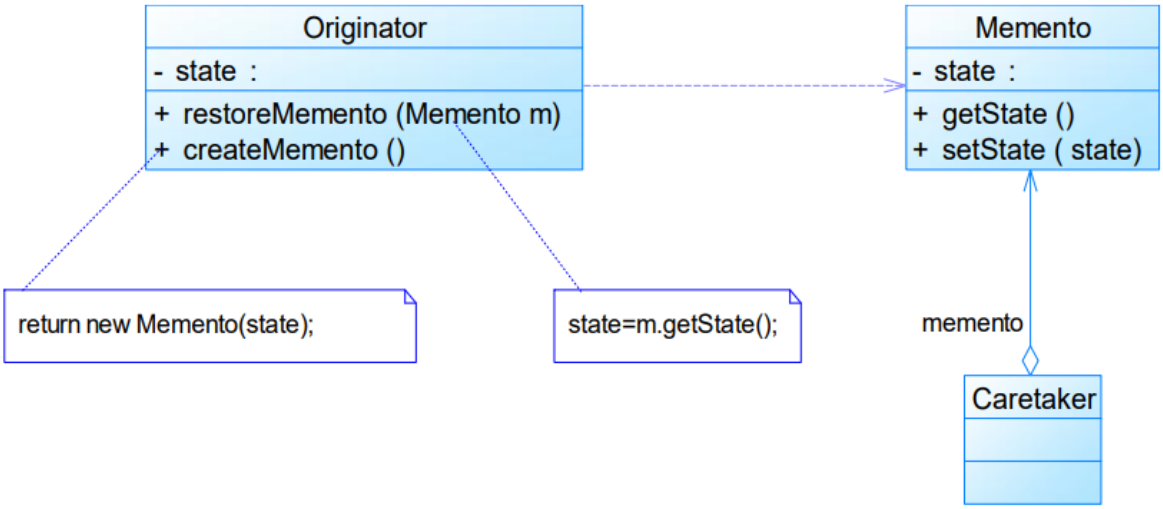

图5.10 备忘录模式

- Originator：原发器，创建备忘录用以记录当前时刻它的内部状态。
- Memento：备忘录，存储原发器对象的内部状态。除了原发器和管理者之外，备忘录对象不能直接供其他类使用。
- Caretaker：管理者，负责保存备忘录，但是不能对备忘录的内容进行操作或检查。

示例代码：

```
class Originator
{
    private String state;

    public String getState()
    {
        return state;
    }

    public setState(String state)
    {
        this.state = state;
    }

    // 将当前状态保存到备忘录中
    public Memento createMemento()
    {
        return new Memento(state);
    }

    public void getStateFromMemento(Memento memento)
    {
        state = memento.getState();
    }
}

class Memento
{
    private String state;

    public Memento(String state)
```

```
    {
        this.state = state;
    }

    public String getState()
    {
        return state;
    }
}

class Caretaker
{
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento memento)
    {
        mementoList.add(memento)
    }

    public Memento get(int index)
    {
        return mementoList.get(index);
    }
}

public class Client
{
    public static void main(String[] args)
    {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        originator.setState("状态1");
        caretaker.add(originator.createMemento()); // 保存当前状态

        originator.setState("状态2");
        caretaker.add(originator.createMemento()); // 保存当前状态

        originator.setState("状态3");
        caretaker.add(originator.createMemento()); // 保存当前状态
        System.out.println(originator.getState());

        originator.getStateFromMemento(caretaker.get(0)); // 恢复到状态1
        System.out.println(originator.getState());
    }
}
```

备忘录模式的优点：

1. 保持封装边界。使用备忘录模式可以避免暴露一些只应由原发器管理却又必须存储在原发器之外的信息。备忘录模式把可能很复杂的原发器内部信息对其他对象屏蔽起来，保持了封装边界，使得客户不需要关心状态的保存细节。
2. 简化了原发器。让客户管理请求的状态将会简化原发器，并且使得客户工作结束时无须通知原发器。

备忘录模式的缺点：

1. 如果原发器在生成备忘录时必须拷贝并存储大量的信息，或者客户非常频繁地创建备忘录和恢复原发器状态，可能会导致非常大的开销。为了节约内存，可以将备忘录模式和原型模式结合使用。
2. 维护备忘录的潜在代价。管理者负责删除它所维护的备忘录，然而管理者不知道备忘录中有多少状态，因此当存储备忘录时，一个本来很小的管理者可能会产生大量的存储开销。

备忘录模式的适用场合：

1. 必须保存一个对象在某个时刻的状态，以后需要时可以恢复到先前的状态。
2. 如果一个接口让其他对象直接得到这些状态，将会暴露对象的实现细节并破坏封装性。

备忘录模式可以与命令模式组合使用。在命令模式中，实现命令的撤销和重做时可以使用备忘录模式。在命令操作的时候将操作前后的状态记录在备忘录对象中，在命令撤销和重做的时候使用相应的备忘录对象来恢复状态。

备忘录模式可以和原型模式组合使用。在原发器对象创建备忘录对象的时候，如果原发器对象中全部或大部分的状态都需要保存，一个简洁的方式就是直接克隆一个原发器对象，在备忘录对象中保存一个原发器对象的拷贝。

5.8 状态模式（State Pattern）

状态模式：允许一个对象在其内部状态改变时改变它的行为。

状态模式的结构如图 5.11 所示。

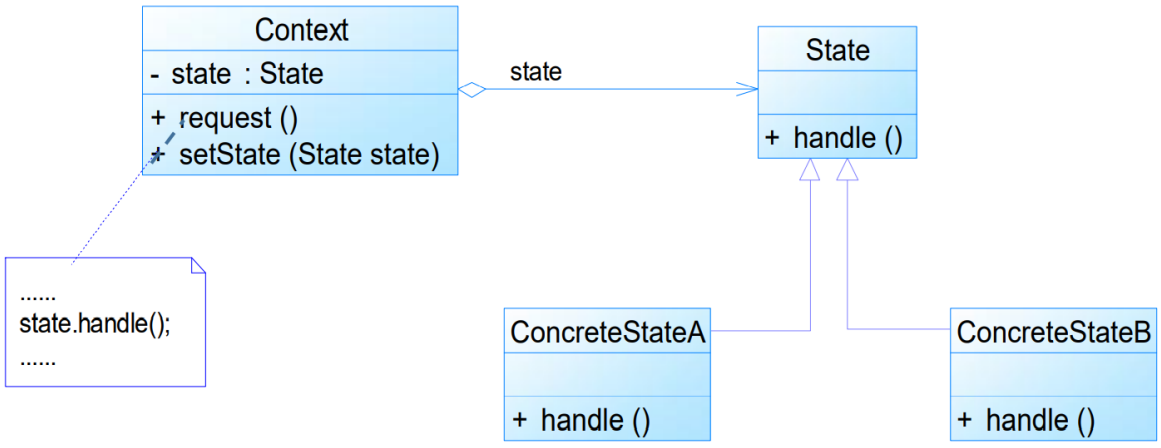

图5.11 状态模式

- Context：环境，维护一个 ConcreteState 的实例，这个实例定义当前状态。
- State：抽象状态，定义一个接口以封装与 Context 的一个特定状态相关的行为。
- ConcreteState：具体状态，每个子类实现一个与 Context 的一个状态相关的行为。所有具体状态类的操作是相同的，只是实现不同。

状态模式用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题。状态模式把一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化。对于客户端而言，无须关心对象状态的转换以及对象所处的当前状态，无论对于何种状态的对象，客户端都可以一致处理。

状态模式不指定哪个参与者定义状态转换规则。如果该规则是固定的，可以由 Context 类自己控制、集中管理。如果让具体状态类指定它们的后继状态以及何时进行转换，通常更灵活，这需要 Context 增加一个接口，让 State 对象显式地设置 Context 的当前状态。将状态转换逻辑分散，可以很容易地定义新的 State 子类来修改和扩展该逻辑，但缺点在于，一个 State 子类至少拥有一个其他子类的信息，这就在各子类之间产生了依赖。

状态模式的优点：

1. 状态转换代码相对集中。可以把状态转换代码封装在环境类或具体状态类中，而不是分散在业务方法中。

2. 把所有与某个状态有关的行为放到一个类层次中，只需要注入一个不同的状态对象即可使环境对象拥有不同的行为方式。
3. 允许状态转换逻辑与状态对象合成一体，这样可以避免使用庞大的条件语句来将业务方法和状态转换代码交织在一起。
4. 可以让多个同类的环境对象共享一个状态对象，从而减少系统中对象的个数。
5. 使得状态转换显式化。为不同的状态引入独立的对象使得转换变得更加明确。而且 State 对象可以保证 Context 对象不会发送内部状态不一致的情况，因为从 Context 的角度看，状态转换是原子的，只需重新绑定一个变量。

状态模式的缺点：

1. 状态模式的使用会增加系统中类和对象的数目，可能导致系统运行开销增大。
2. 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，增加系统设计的难度。
3. 状态模式对开闭原则的支持不太好，增加新的状态类需要修改那些负责状态转换的源代码。
4. 具体状态对象的行为的种类是受限的，受限于抽象状态类的接口。有时抽象状态类定义的操作对某些具体状态类是无用的，这会导致对里氏替换原则的支持减弱。

状态模式的适用场合：

1. 一个对象的行为取决于它的状态，状态的改变将导致行为的变化。
2. 在代码中包含大量与对象状态有关的条件语句，这些条件语句的出现会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，并且导致客户类与类库之间的耦合增强。
3. 在描述具有复杂行为的对象时，往往需要给出状态模型，状态模式可应用于状态模型的实现。

5.9 策略模式（Strategy Pattern）

策略模式：定义一系列算法，把它们分别封装起来，并且使它们可以相互替换。策略模式使得算法可以独立于使用它的客户而变化。

策略模式的结构如图 5.12 所示。

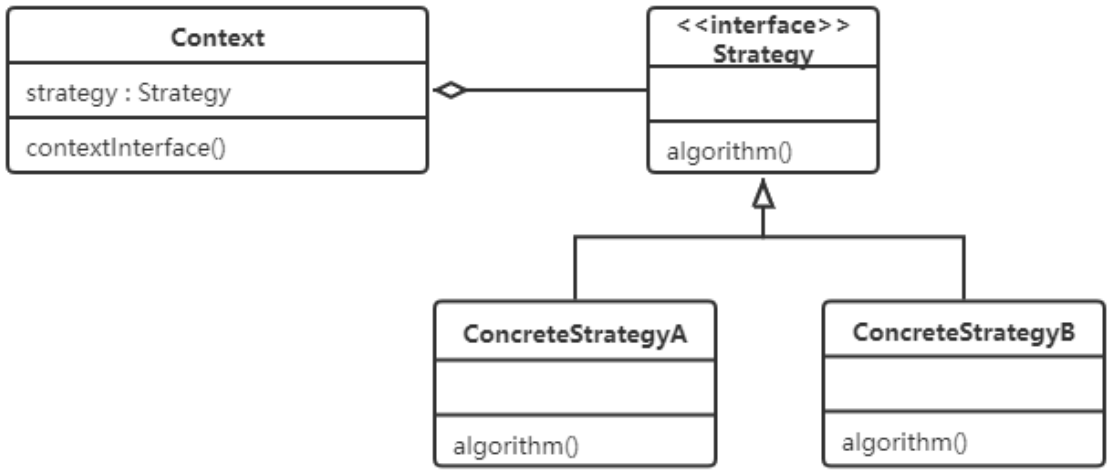

图5.12 策略模式

- Context：上下文，维护一个 Strategy 对象的引用，可以定义一个接口来让 Strategy 对象访问它的数据。
- Strategy：抽象策略，定义所有支持的算法的公共接口，Context 使用这个接口来调用某个算法。
- ConcreteStrategy：具体策略，实现某个算法。

策略模式体现的设计原则：

1. 把变化的代码从不变的代码中分离出来。
2. 依赖倒置原则：面向接口编程，而不是具体类。

3. 组合/聚合复用原则：多用组合/聚合，少用继承。
4. 开闭原则：要想增加行为，只需增加一个 ConcreteStrategy 类即可。

策略模式的优点：

1. Strategy 类层次为 Context 定义了一系列可供复用的算法或行为。
2. 提供了替代继承的方法。将算法封装在独立的 Strategy 类层次中，使得算法可以独立于 Context 而改变，算法易于切换、易于理解、易于扩展。
3. 消除了一些条件语句。当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在 Strategy 类中消除了这些条件语句。
4. 策略模式可以提供相同行为的不同实现，客户可以根据需要做出选择。

策略模式的缺点：

1. 客户必须了解不同的 Strategy，只有这样才能选出一个合适的策略。此时可能不得不向客户暴露具体实现。
2. 所有的 ConcreteStrategy 共享 Strategy 定义的接口，很可能某些 ConcreteStrategy 不会用到所有通过这个接口传递进来的信息，这意味着有时 Context 会创建和初始化一些永远不会用到的参数。如果存在这样的问题，需要在 Strategy 和 Context 之间进行更紧密的耦合。
3. 每添加一个策略就要增加一个类，当策略过多时会导致类数目庞大。
4. 策略模式增加了应用中对象的数目。可以将 Strategy 设计成无状态的对象，让各 Context 共享 Strategy对象，以减少开销（享元模式）。

策略模式的应用实例：JDK 中的`Arrays.sort`方法使用了策略模式。代码如下：

```
@FunctionalInterface
public interface Comparator<T> // 策略接口
{
    int compare(T o1, T o2);
}

public class Arrays
{
    public static <T> void sort(T[] a, Comparator<? super T> c)
    {
        if (c == null)
        {
            sort(a); // 使用默认方式排序
        }
        else
        {
            if (LegacyMergeSort.userRequested)
            {
                legacyMergeSort(a, c); // 使用给定的策略进行排序
            }
            else
                TimSort.sort(a, 0, a.length, c, null, 0, 0);
        }
    }
}

public class Test
{
    public static void main(String[] args)
    {
        // 指定策略
        Comparator<Integer> comparator = new Comparator<Integer>() {
```

```
33              public int compare(Integer o1, Integer o2)
34              {
35                  if (o1 > o2)
36                      return 1;
37                  else
38                      return -1;
39              }
40          };
41
42          Integer[] array = {9, 1, 2, 8, 4, 3};
43          Arrays.sort(array, comparator); // 使用策略进行排序
44          System.out.println(Arrays.toString(array));
45      }
46  }
```

5.10 职责链模式（Chain of Responsibility Pattern）

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

职责链模式的结构如图 5.13 所示。

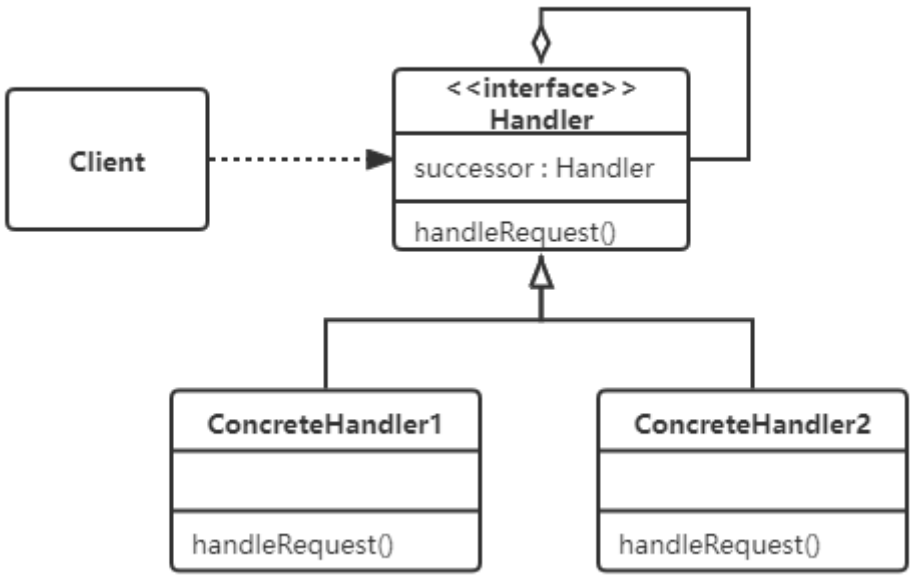

图5.13 职责链模式

- Handler：抽象处理者，定义一个处理请求的接口，并维护另一个处理者作为自己的后继。
- ConcreteHandler：具体处理者，处理它所负责的请求。如果可以处理该请求，就直接处理，否则将该请求转发给后继者。

职责链模式的优点：

1. 降低耦合度。将请求者和处理者分开，使得请求者无须知道是哪一个处理者处理请求。接收者和发送者都没有对方的明确信息。
2. 简化对象的相互连接，链中的对象不需要知道链的结构。
3. 增强了给对象指派职责的灵活性。可以在运行时对职责链进行动态的增加或修改，从而增加或改变处理一个请求的那些职责。

职责链模式的缺点：

1. 不能保证请求一定被处理。
2. 当链比较长的时候，性能会受到影响。因此需要控制链中的最大节点数量。

3. 调试不方便。

职责链模式的适用场合：

1. 有多个对象可以处理一个请求，在运行时自动确定由哪个对象处理该请求。
2. 在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
3. 可处理一个请求的对象集合应被动态指定。

5.11 解释器模式（Interpreter Pattern）

6 其他话题

6.1 双向一对一关联的实现

以夫妻关系为例，有 `Male` 类和 `Female` 类，如何实现双向一对一的夫妻关系。

方案一：相互引用，如图 6.1 所示。

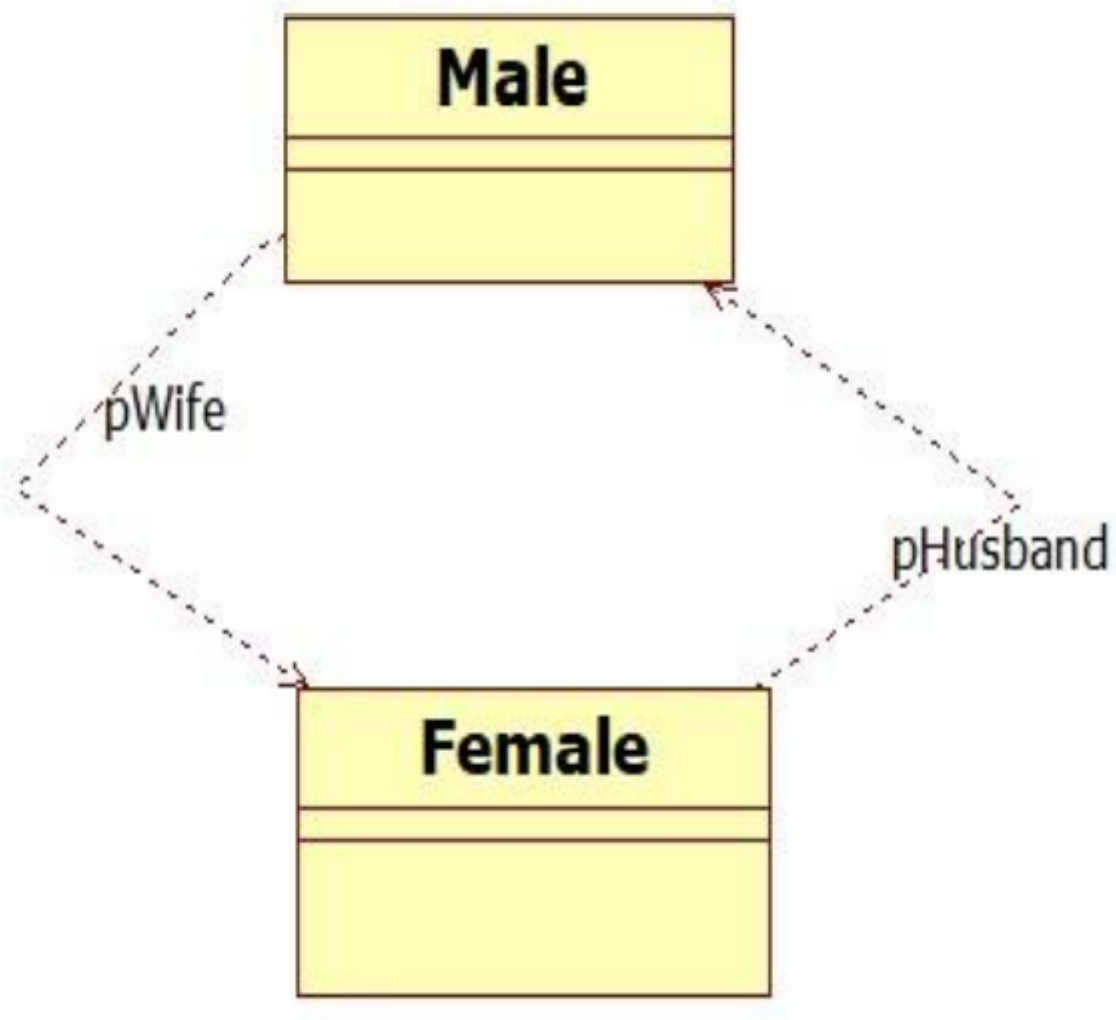

图6.1 夫妻关系：方案一

优点：直观

缺点：关系维护复杂，由两个类共同维护，涉及关系的一致性问题。

方案二：一方指向另一方，如图 6.2 所示。

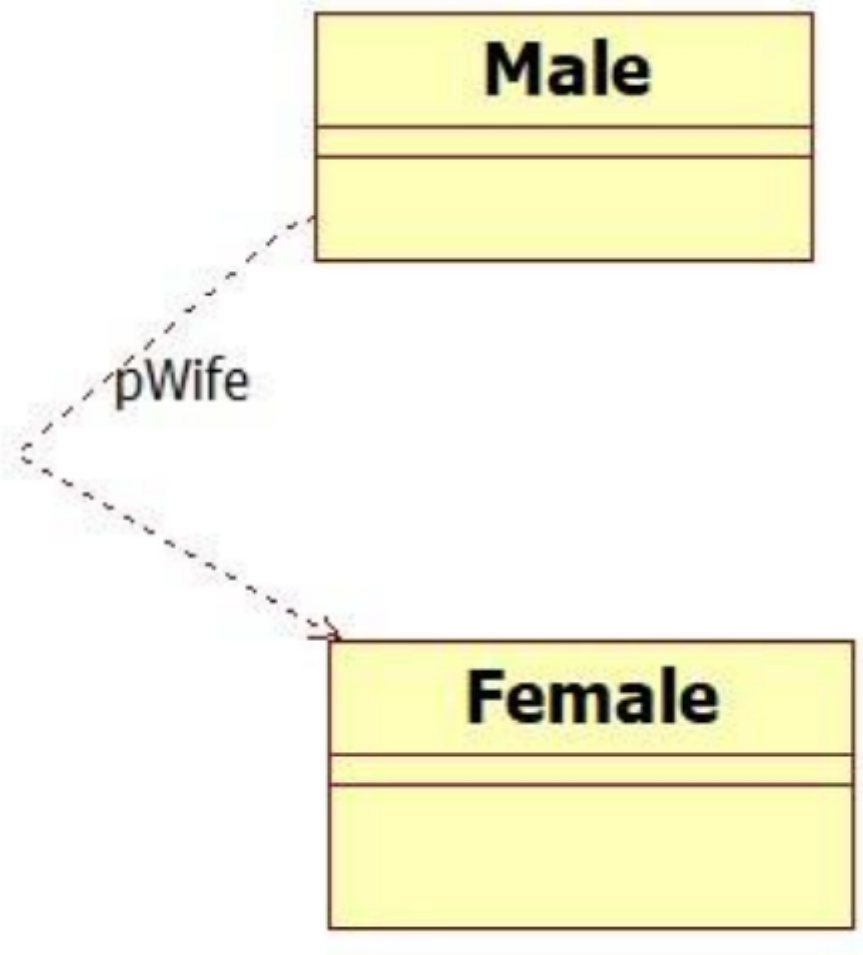

图6.2 夫妻关系：方案二

优点：关系的维护方便，关系的一致性容易得到保证。

缺点：另一方查找带有引用的一方时复杂一些。

方案二适用于双方关系不平衡的场合。

方案三：关系由第三方维护，如图 6.3 所示。

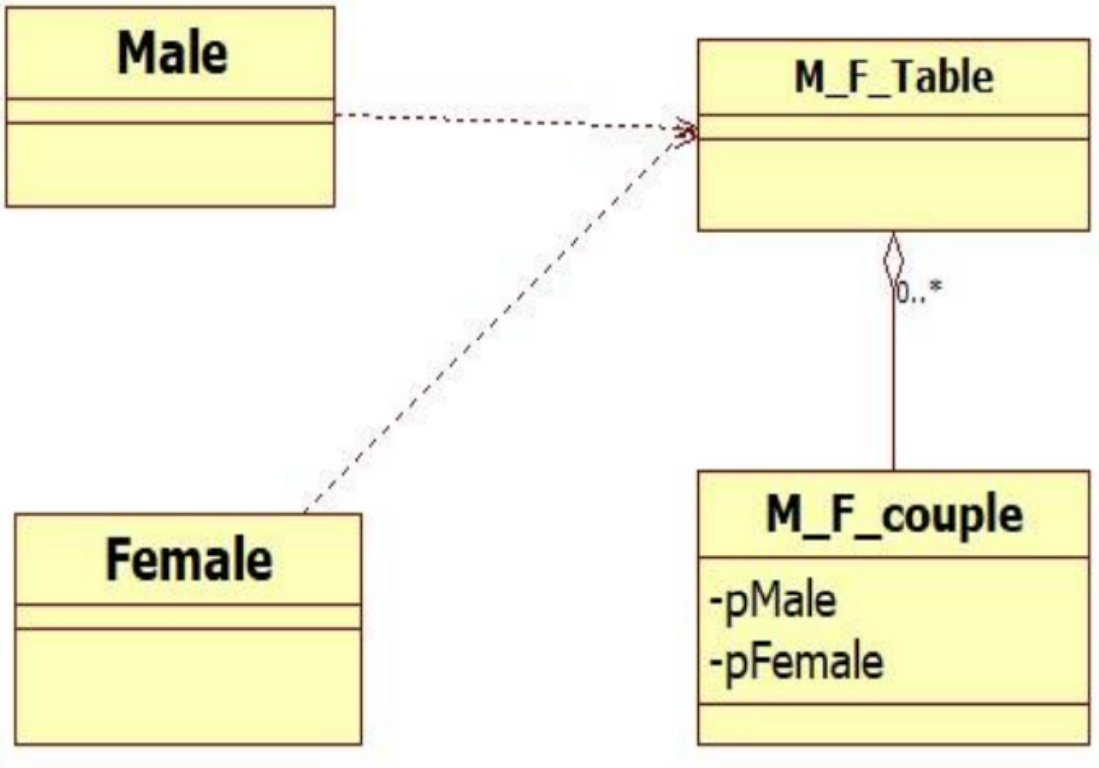

图6.3 夫妻关系：方案三

优点：关系的维护方便，关系的一致性容易得到保证。

方案三是更一般的方法。

6.2 COM：组件对象模型

6.2.1 COM简介

组件：一些可执行的二进制程序，它可以给其他的应用程序、操作系统或其他组件提供功能。

面向组件编程：将功能和数据封装成二进制代码，采用搭积木的方式实现软件的一种编程方式。

组件的优点：

1. 可以方便地提供软件定制机制。
2. 可以很灵活地提供功能。
3. 可以很方便地实现程序的分布式开发。

COM：Component Object Model，组件对象模型。COM 是一种编程规范，定义了组件的操作、接口的访问等。

COM 规范的主要内容：

1. 组件由接口部分和实现部分构成。
2. 接口部分有若干接口，每个接口相当于组件对外的某一方面的承诺。一个接口包括一组函数，这些函数共同代表组件具有的某一方面的能力。
3. 实现部分需要实现接口部分的每一个接口里面的每一个函数，实现方式不做约束。实现部分是黑盒，被封装，不被任何客户程序所见，不被依赖。

COM 组件：在 Windows 平台下，封装在动态链接库（DLL）或可执行文件（EXE）中的一段代码，这些代码按照 COM 的规范实现。

COM 组件的特点：

1. 动态链接。

2. 与编程语言无关。
3. 以二进制形式发布。
4. 向前兼容。

6.2.2 COM接口

COM 接口：COM 接口是组件的核心，为用户提供了访问组件的方式，通过 COM 接口提供的函数使用组件的功能。

COM 接口的实现：一个包含了一组函数指针的数据结构，这些函数是由组件实现的。

`IUnknown` 接口是微软定义的标准接口，任何一个 COM 组件都必须提供 `IUnknown` 接口。`IUnknown` 接口定义了 3 个函数：`QueryInterface`、`AddRef`、`Release`。

`QueryInterface` 函数：查询组件是否具有某个接口。

`QueryInterface` 函数的作用：

1. 支持动态链接。调用函数时需要检查调用关系的合理性。静态链接时可以由连接程序检查调用关系的合理性。而动态链接是运行时连接，不存在连接程序，需要其他机制来检查调用关系的合理性。为了满足调用关系的合理性，客户程序需要先查询接口是否存在，只有接口存在才可以使用。
2. 客户程序在使用组件时，往往只需要用到组件的接口集的一个子集，并且可能并不需要同时用到多个接口，而是依次使用不同的接口。所以组件运行时，不一定需要整体运行，而是客户程序用到哪个接口，就产生该接口对应的实现体，用完该接口后，对应的实现体可以释放所占资源。这样，客户程序使用组件的过程相当于组件支持不同接口的实现体的状态变迁过程。可以把组件的接口集看成是组件运行时的状态集，每个接口对应一个合法的状态，每种状态下都要支持 `QueryInterface` 函数的调用，而 `QueryInterface` 函数的调用导致组件的状态变迁。因此，组件的其他所有接口中都必须有 `QueryInterface` 函数。

组件中有一个引用计数器，`AddRef` 和 `Release` 函数共同维护这个引用计数器，`AddRef` 函数使得计数器加 1，`Release` 函数使得计数器减 1。引用计数器用于记录当前正在使用组件的客户程序的数量。当客户程序使用组件时，调用 `AddRef` 函数，使得组件的计数器加 1；当客户程序使用完毕后，调用 `Release` 函数，使得计数器减 1。当计数器的值变为 0 时，组件已经没有客户程序，此时可以将组件释放。

`IUnknown` 接口之外的其他接口约束：

1. 每个接口都必须包括 `IUnknown` 接口中的 3 个函数，可以看成 `IUnknown` 接口的子接口。
2. 虽然 COM 规范不约束实现，但因为每个接口中都有 `QueryInterface` 函数，所以从组件的外在表现方面进行了与该函数相关的约束：

   （1）一旦某个接口能够被客户程序查询到，那么它应该始终都被查到。

   （2）一旦某个接口能够被客户程序查询到，那么通过该组件的任何一个接口中的 `QueryInterface` 函数都能查到它。

接口的不变性约束：COM 组件的任何一个接口一旦发布出去，就不能对该接口做任何变化。变化包括：接口内每一个函数的原型、函数的顺序等。

其他约束：任何一个组件有唯一的一个标识，组件的任何一个接口也有唯一的一个标识。

代码示例：

```
// InterfaceDef.h
#ifndef _interfaceDef_h_
#define _interfaceDef_h_

#define IUnknownID  0
```

```
#define IXID  1
#define IYID  2
#define IClassFactoryID 10
#define CompAID 100

class IMyUnknown // 纯的抽象类间接表示接口
{
public:
    virtual bool QueryInterface(int interfaceID,void** pValue) = 0;
    virtual void AddRef( ) = 0;
    virtual void Release( ) = 0;
};

class IX :  public IMyUnknown
{
public:
    virtual void f1( ) = 0;
    virtual void f2( ) = 0;
};

class IY : public IMyUnknown
{
public:
    virtual void g1( ) = 0;
    virtual void g2( ) = 0;
};

/*
1. 随后工厂接口的定义可以忽略从IMyUnknown扩展.
2. 这里从IMyUnknown扩展，并不意味着它是组件的接口。
3. 从IMyUnknown扩展，是考虑复杂情形：组件有多个工厂接口。
    工厂实例可以类似组件实例一样改变状态，一个工厂可以从支持
    某一个工厂接口的状态变成支持另一工厂接口的状态。
*/
class IMyClassFactory : public IMyUnknown
{
public:
    virtual bool CreateInstance(int interfaceID, void** pValue) = 0;
/* 若不考虑支持多个工厂接口，下面这个就可以。
    virtual IMyUnknown* CreateInstance( ) = 0;
*/
};

#endif
```

```
// ComponentA.cpp
#include <windows.h>
#include <iostream>
#include "InterfaceDef.h"

using namespace std;

class C : public IX, public IY  // 顶层结构上用一个类实现所有接口
{
public:
    C( ) { theCounter = 0; otherData = 0; }
    virtual ~C( ) { }
```

```
    virtual bool QueryInterface(int interfaceID,void** pValue);
    // 组件接口的标识作为参数

    virtual void AddRef( ) { theCounter++ ; }
    virtual void Release( )
    {
        if (theCounter > 0)
            theCounter--;
        if (theCounter == 0)
            delete this;
    }
    virtual void f1( )
    {
        cout << "接口Ix中f1 被触发" << endl;
    }
    virtual void f2( )
    {
        cout << "接口Ix中f2 被触发" << endl;
    }
    virtual void g1( )
    {
        cout << "接口Iy中g1 被触发" << endl;
    }
    virtual void g2( )
    {
        cout << "接口Iy中g2 被触发" << endl;
    }

private:
    long theCounter;
    int otherData;
};

bool C::QueryInterface(int interfaceID, void** pValue)
{
    switch (interfaceID)
    {
        case IUnknownID :
        {
            *pValue = static_cast<IX*> (this);
            break;
        }
        case IXID :
        {
            *pValue = static_cast<IX*> (this);
            break;
        }
        case IYID :
        {
            *pValue = static_cast<IY*> (this);
            break;
        }
        default:
            return false;
    }
    this->AddRef( ); // 本例子可以忽略掉引用计数。
    return true;
}
```

```
71
72  /* 因为本例中，IMyClassFactory从IMyUnknown扩展定义，
73      所以，下面工厂类的实现略复杂了一些，支持工厂状态变化。
74      若简化，可简单理解为：只有CreateInstance函数即可。
75  */
76  class Factory : public IMyClassFactory // 类厂接口，工厂接口
77  {
78  public:
79      Factory( ) { theCounter = 0; }
80      virtual ~Factory( ) {  }
81      virtual void AddRef( ) { theCounter++; }
82      virtual void Release( )
83      {
84          if (theCounter > 0)
85              theCounter--;
86          if (theCounter == 0)
87              delete this;
88      }
89      virtual bool QueryInterface(int interfaceID,void** pValue)
90      {
91          if ((interfaceID == IUnknownID ) ||
92              (interfaceID == IClassFactoryID ))
93          {
94              *pValue = static_cast<IMyClassFactory* > (this);
95              return true;
96          }
97          else
98              return  false;
99      }
100
101     virtual bool CreateInstance(int interfaceID, void** pValue)
102     {
103         C* pC = new C;
104         return (pC->QueryInterface(interfaceID,pValue));
105     }
106 private:
107     long theCounter;
108 };
109
110
111 BOOL APIENTRY DllMain( HANDLE hModule,
112                        DWORD  ul_reason_for_call,
113                        LPVOID lpReserved
114                      )
115 {
116     if (ul_reason_for_call == DLL_PROCESS_ATTACH)
117     {
118         char name[100];
119         ::GetModuleFileName(HINSTANCE(hModule), name, 100);
120         cout << "组件所在位置是：" << name << endl;
121     }
122     else if (ul_reason_for_call == DLL_PROCESS_DETACH)
123     {
124
125     }
126     return TRUE;
127 }
128
```

```
/* 1.  最简单形式，直接返回一个组件实例，
         支持IMyUnknown的实体就是组件实例。
extern "C" __declspec(dllexport) IMyUnknown* getCompInstance( )
{
    C* pC = new C;
    return (IMyUnknown*)(static_cast<IX*>(pC));
}

     2. 允许客户程序指定组件实例的初始状态，
         而不是缺省地返回一个支持IMyUnknown的实体。
extern "C" __declspec(dllexport) bool getCompInstance(int InterfaceID,
void** theRet )
{
    C* pC = new C;
    return pC->QueryInterface(InterfaceID,theRet);
}
*/

// 随后的实现考虑得比较复杂：
// 假设该动态连接库支持多个COM组件，（第一个参数）
// 并假设一个COM组件可以支持多个工厂接口。（第二个参数）
// 该导出函数的含义是：
// 试图通过参数返回一个客户程序指定的某个组件的某种工厂的实例.
// 即：按照客户程序指定的方式返回某个组件的某种工厂实例。
extern "C" __declspec(dllexport) bool GetClassObject(int ComponetID, int
ClassFactoryIID, void** pValue)
{
    if (ComponetID != CompAID)
        return false;
    if (ClassFactoryIID != IClassFactoryID)
        return false;
    IMyClassFactory* pTheFactory = new Factory;
    return pTheFactory->QueryInterface(ClassFactoryIID,pValue);
}
```