OOP与OOAD介绍

面向对象编程的英文缩写是 OOP，全称是 Object Oriented Programming。对应地，面向对象编程语言的英文缩写是 OOPL，全称是 Object Oriented Programming Language。

面向对象编程中有两个非常重要、非常基础的概念，那就是类（class）和对象（object）。

理解面向对象编程的四大特性：封装、继承、多态、抽象。

面向对象分析英文缩写是 OOA，全称是 Object Oriented Analysis；面向对象设计的英文缩写是 OOD，全称是 Object Oriented Design。OOA、OOD、OOP 三个连在一起就是面向对象分析、设计、编程（实现），正好是面向对象软件开发要经历的三个阶段。

讲到面向对象分析、设计、编程，我们就不得不提到另外一个概念，那就是 UML（Unified Model Language），统一建模语言。很多讲解面向对象或设计模式的书籍，常用它来画图表达面向对象或设计模式的设计思路。

UML类图及类图的关系

统一建模语言简介

统一建模语言（Unified Modeling Language，UML）是用来设计软件蓝图的可视化建模语言，1997 年被国际对象管理组织（OMG）采纳为面向对象的建模语言的国际标准。它的特点是简单、统一、图形化、能表达软件设计中的动态与静态信息。

统一建模语言能为软件开发的所有阶段提供模型化和可视化支持。而且融入了软件工程领域的新思想、新方法和新技术，使软件设计人员沟通更简明，进一步缩短了设计时间，减少开发成本。它的应用领域很宽，不仅适合于一般系统的开发，而且适合于并行与分布式系统的建模。

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

本课程主要介绍软件设计模式中经常用到的类图，以及类之间的关系。另外，在实验部分将简单介绍 UML 建模工具的使用方法，当前业界使用最广泛的是 Rational Rose。使用 Umlet 的人也很多，它是一个轻量级的开源 UML 建模工具，简单实用，常用于小型软件系统的开发与设计。

类、接口和类图

1. 类

类（Class）是指具有相同属性、方法和关系的对象的抽象，它封装了数据和行为，是面向对象程序设计（OOP）的基础，具有封装性、继承性和多态性等三大特性。在 UML 中，类使用包含类名、属性和操作且带有分隔线的矩形来表示。

(1) 类名（Name）是一个字符串，例如，Student。

(2) 属性（Attribute）是指类的特性，即类的成员变量。UML 按以下格式表示：

```
[可见性]属性名:类型[=默认值]
```

例如：-name:String

注意："可见性"表示该属性对类外的元素是否可见，包括公有（Public）、私有（Private）、受保护（Protected）和朋友（Friendly）4 种，在类图中分别用符号+、-、#、~表示。

(3) 操作（Operations）是类的任意一个实例对象都可以使用的行为，是类的成员方法。UML 按以下格式表示：

```
[可见性]名称(参数列表)[:返回类型]
```

例如：+display():void。

图 1 所示是学生类的 UML 表示。

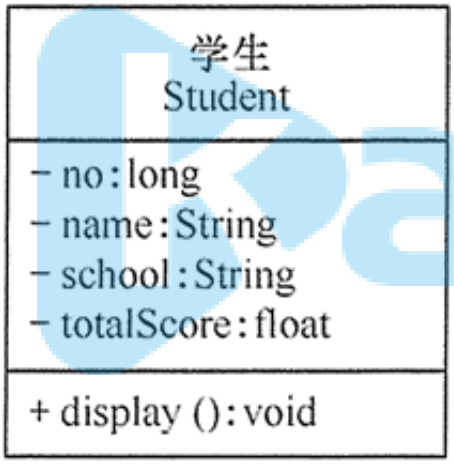


图1 Student 类

2. 接口

接口（Interface）是一种特殊的类，它具有类的结构但不可被实例化，只可以被子类实现。它包含抽象操作，但不包含属性。它描述了类或组件对外可见的动作。在 UML 中，接口使用一个带有名称的小圆圈来进行表示。

图 2 所示是图形类接口的 UMDL 表示。

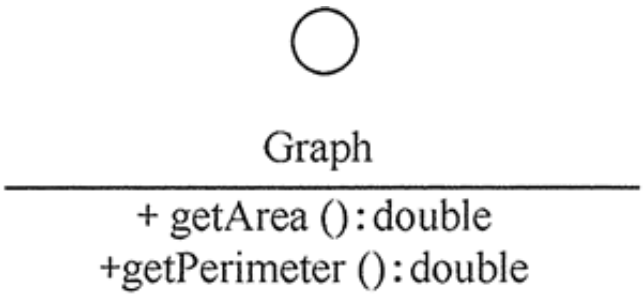


图2 Graph 接口

3. 类图

类图（ClassDiagram）是用来显示系统中的类、接口、协作以及它们之间的静态结构和关系的一种静态模型。它主要用于描述软件系统的结构化设计，帮助人们简化对软件系统的理解，它是系统分析与设计阶段的重要产物，也是系统编码与测试的重要模型依据。

类图中的类可以通过某种编程 语言直接实现。类图在软件系统开发的整个生命周期都是有效的，它是面向对象系统的建模中最常见的图。

图 3 所示是“计算长方形和圆形的周长与面积”的类图，图形接口有计算面积和周长的抽象方法，长方形和圆形实现这两个方法供访问类调用。

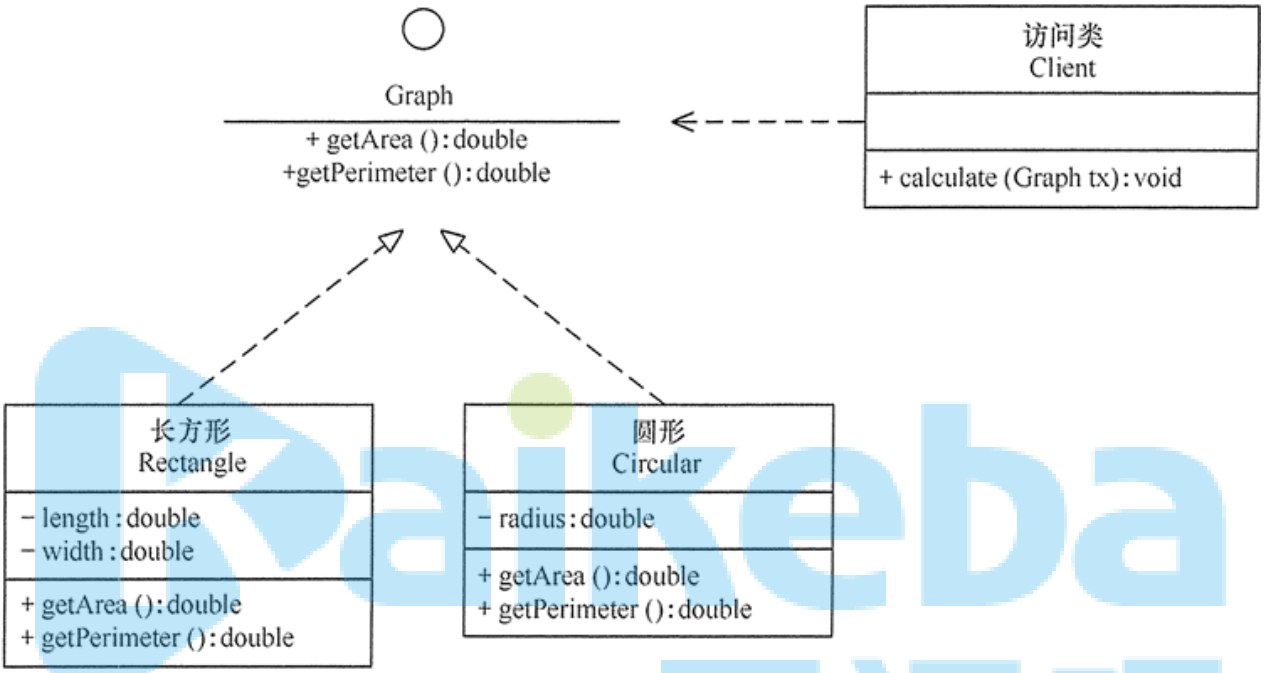


图3 “计算长方形和圆形的周长与面积”的类图

类之间的关系

在软件系统中，类不是孤立存在的，类与类之间存在各种关系。

根据类与类之间的耦合度从弱到强排列，UML 中的类图有以下几种关系：依赖关系、关联关系、聚合关系、组合关系、泛化关系和实现关系。其中泛化和实现的耦合度相等，它们是最强的。

1. 依赖关系

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

在 UML 类图中，依赖关系使用**带箭头的虚线**来表示，箭头从使用类指向被依赖的类。图 4 所示是人与手机的关系图，人通过手机的语音传送方法打电话。

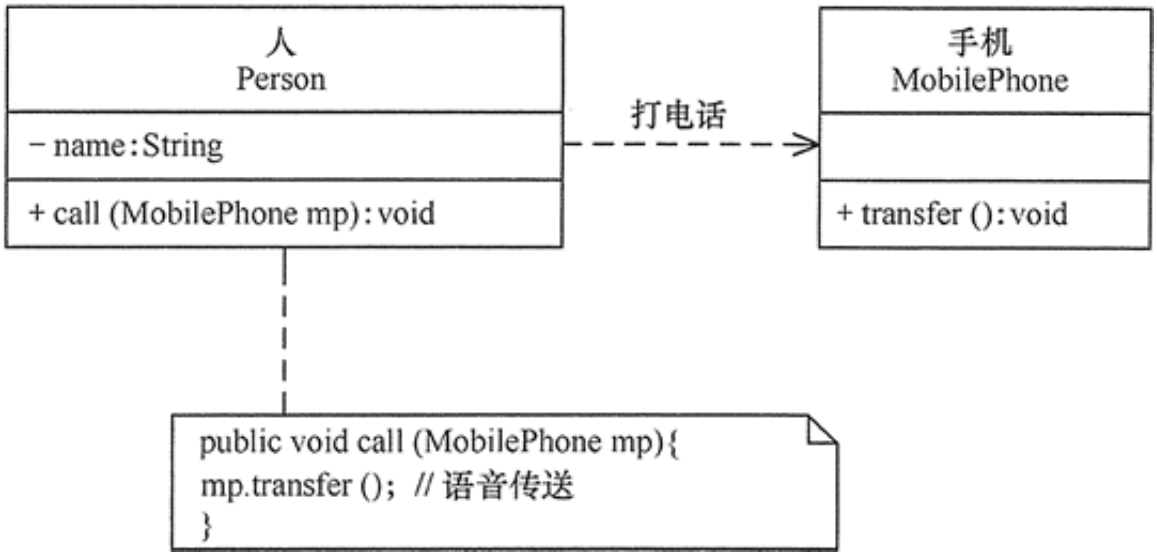


图4 依赖关系的实例

2. 关联关系

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

关联可以是双向的，也可以是单向的。

在 UML 类图中，双向的关联可以用**带两个箭头或者没有箭头的实线**来表示，单向的关联用**带一个箭头的实线**来表示，箭头从使用类指向被关联的类。也可以在关联线的两端标注角色名，代表两种不同的角色。

在代码中通常将一个类的对象作为另一个类的成员变量来实现关联关系。

图 5 所示是老师和学生的关系图，每个老师可以教多个学生，每个学生也可向多个老师学，他们是双向关联。

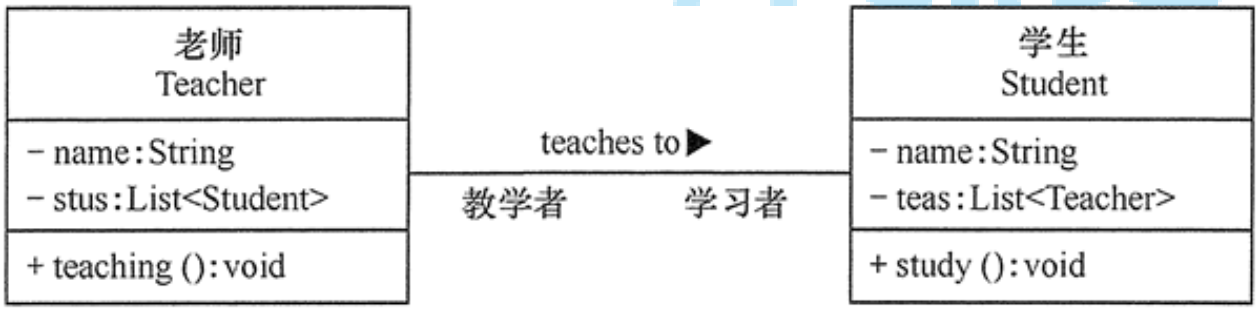


图5 关联关系的实例

3. 聚合关系

聚合（Aggregation）关系是关联关系的一种，是强关联关系，是整体和部分之间的关系，是 has-a 的关系。

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

在 UML 类图中，聚合关系可以用**带空心菱形的实线**来表示，菱形指向整体。

图 6 所示是大学和教师的关系图。

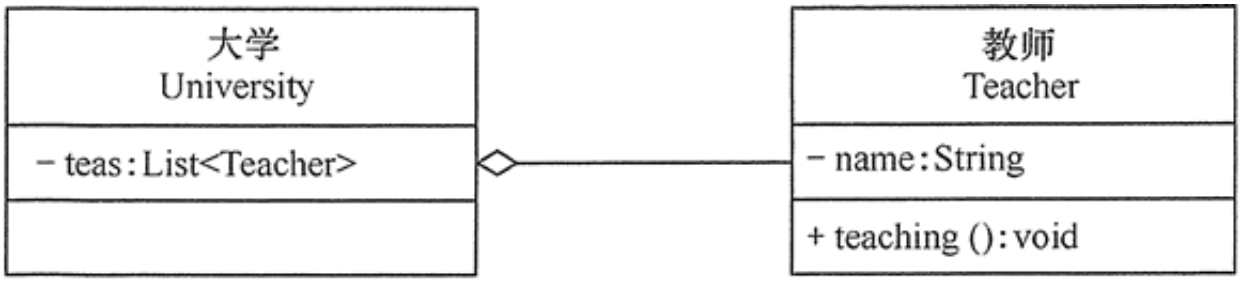


图6 聚合关系的实例

4.组合关系

组合（Composition）关系也是关联关系的一种，也表示类之间的整体与部分的关系，但它是一种更强烈的聚合关系，是 contains-a 关系。

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

在 UML 类图中，组合关系用**带实心菱形的实线**来表示，菱形指向整体。

图 7 所示是头和嘴的关系图。

图7 组合关系的实例

5.泛化关系

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

在 UML 类图中，泛化关系用**带空心三角箭头的实线**来表示，箭头从子类指向父类。在代码实现时，使用面向对象的继承机制来实现泛化关系。例如，Student 类和 Teacher 类都是 Person 类的子类，其类图如图 8 所示。

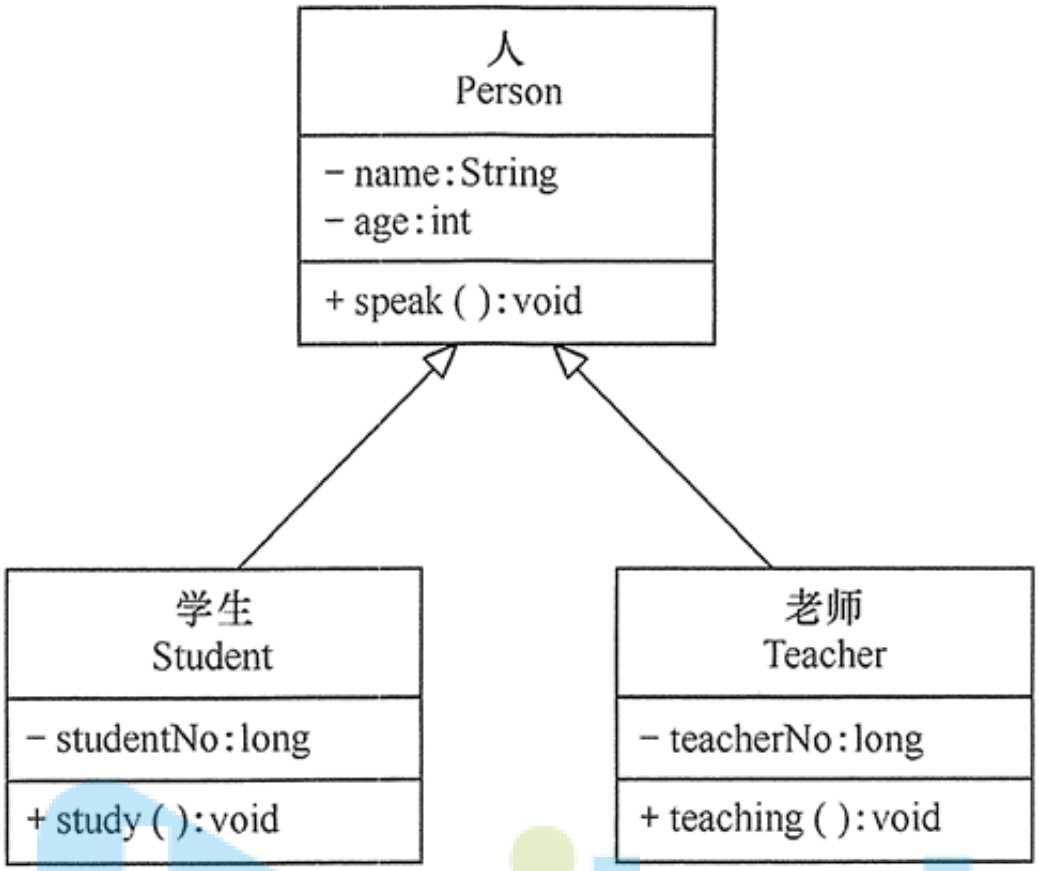


图8 泛化关系的实例

6.实现关系

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

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

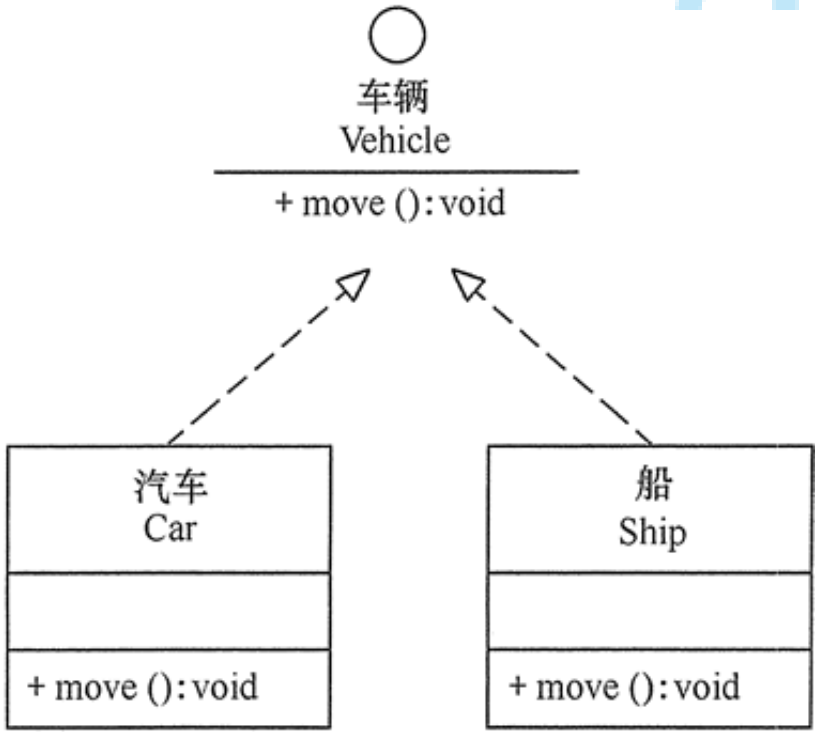


图9 实现关系的实例

面向对象设计原则概述

对于面向对象软件系统的设计而言，在支持可维护性的同时，提高系统的可复用性是一个至关重要的问题，**如何同时提高一个软件系统的可维护性和可复用性是面向对象设计需要解决的核心问题之一**。

在面向对象设计中，可维护性的复用是以设计原则为基础的。每一个原则都蕴含一些面向对象设计的思想，可以从不同的角度提升一个软件结构的设计水平。

面向对象设计原则为支持可维护性复用而诞生，这些原则蕴含在很多设计模式中，它们是从许多设计方案中总结出的指导性原则。面向对象设计原则也是我们用于评价一个设计模式的使用效果的重要指标之一，在设计模式的学习中，大家经常会看到诸如“XXX模式符合XXX原则”、“XXX模式违反了XXX原则”这样的语句。

最常见的7种面向对象设计原则如下表所示：

表1 7种常用的面向对象设计原则

设计原则名称	定 义	使用频率
单一职责原则(Single Responsibility Principle, SRP)	一个类只负责一个功能领域中的相应职责	★★★★☆
开闭原则(Open-Closed Principle, OCP)	软件实体应对扩展开放，而对修改关闭	★★★★★
里氏代换原则(Liskov Substitution Principle, LSP)	所有引用基类对象的地方能够透明地使用其子类的对象	★★★★★
依赖倒转原则(Dependence Inversion Principle, DIP)	抽象不应该依赖于细节，细节应该依赖于抽象	★★★★★
接口隔离原则(Interface Segregation Principle, ISP)	使用多个专门的接口，而不使用单一的总接口	★★☆☆☆
合成复用原则(Composite Reuse Principle, CRP)	尽量使用对象组合，而不是继承来达到复用的目的	★★★★☆
迪米特法则(Law of Demeter, LoD)	一个软件实体应当尽可能少地与其他实体发生相互作用	★★★☆☆

这 7 种设计原则是软件**设计模式**必须尽量遵循的原则，各种原则要求的侧重点不同。其中，【开闭原则】是总纲，它告诉我们要【对扩展开放，对修改关闭】；【里氏替换原则】告诉我们【不要破坏继承体系】；【依赖倒置原则】告诉我们要【面向接口编程】；【单一职责原则】告诉我们实现【类】要【职责单一】；【接口隔离原则】告诉我们在设计【接口】的时候要【精简单一】；【迪米特法则】告诉我们要【降低耦合度】；【合成复用原则】告诉我们要【优先使用组合或者聚合关系复用，少用继承

关系复用】。

面向对象七大设计原则

开闭原则

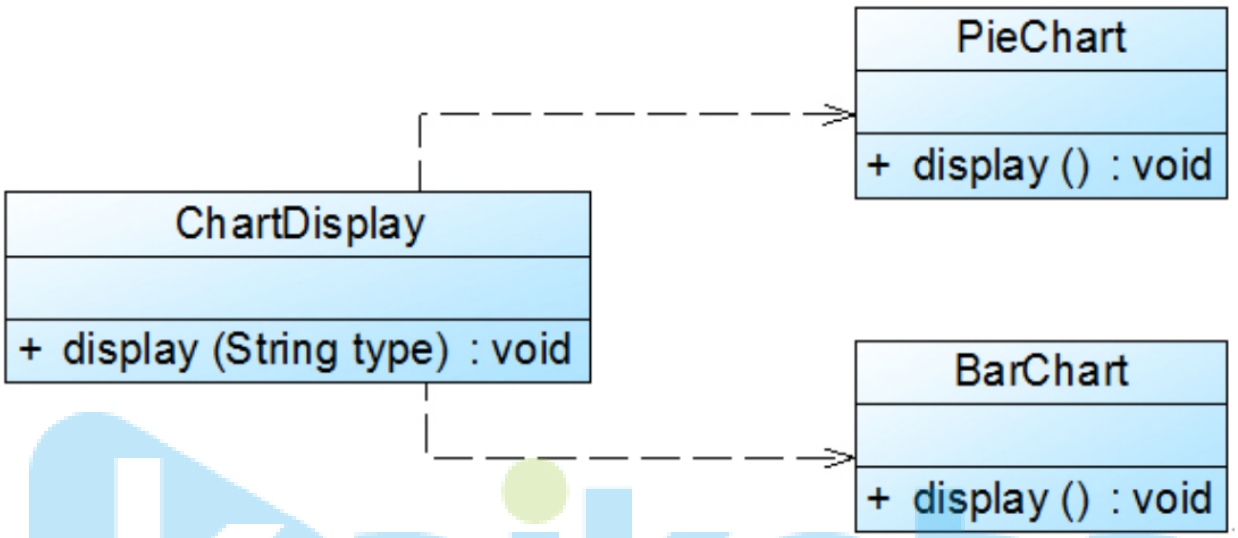


图1 初始设计方案结构图

在ChartDisplay类的display()方法中存在如下代码片段：

```
if (type.equals("pie")) {
  PieChart chart = new PieChart();
  chart.display();
}else if (type.equals("bar")) {
  BarChart chart = new BarChart();
  chart.display();
}
```

> 在该代码中，如果需要增加一个新的图表类，如折线图LineChart，则需要修改ChartDisplay类的display()方法的源代码，增加新的判断逻辑，违反了开闭原则。

现对该系统进行重构，使之符合开闭原则。

在本实例中，由于在ChartDisplay类的display()方法中针对每一个图表类编程，因此增加新的图表类不得不修改源代码。可以通过抽象化的方式对系统进行重构，使之增加新的图表类时无须修改源代码，满足开闭原则。具体做法如下：

- (1) 增加一个抽象图表类AbstractChart，将各种具体图表类作为其子类；
- (2) ChartDisplay类针对抽象图表类进行编程，由客户端来决定使用哪种具体图表。

重构后结构如图2所示：

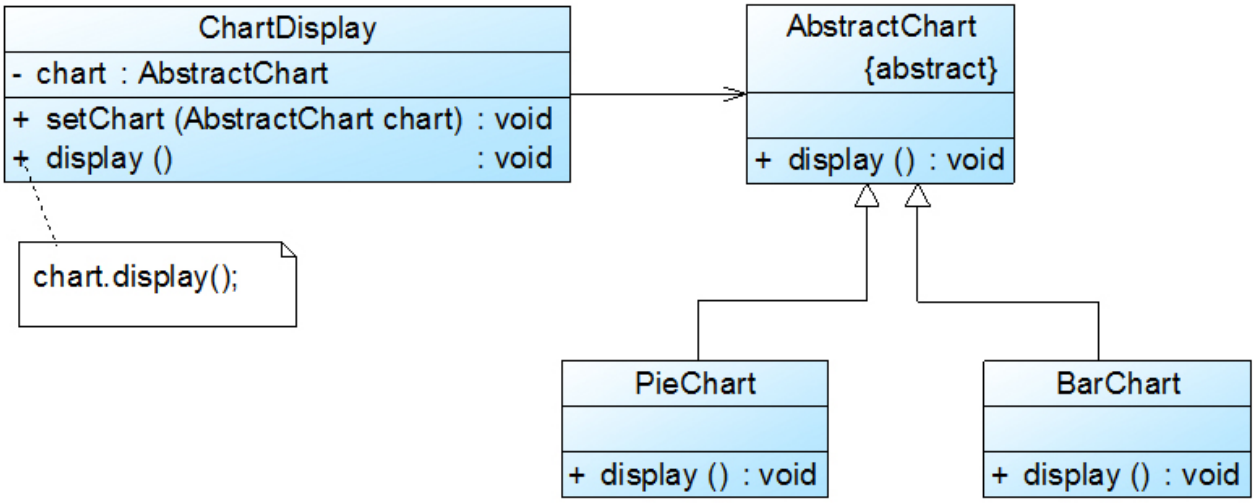


图2 重构后的结构图

> 在图2中，我们引入了抽象图表类AbstractChart，且ChartDisplay针对抽象图表类进行编程，并通过setChart()方法由客户端来设置实例化的具体图表对象，在ChartDisplay的display()方法中调用chart对象的display()方法显示图表。如果需要增加一种新的图表，如折线图LineChart，只需要将LineChart也作为AbstractChart的子类，在客户端向ChartDisplay中注入一个LineChart对象即可，无须修改现有类库的源代码。

注意：

> 因为xml和properties等格式的配置文件是纯文本文件，可以直接通过VI编辑器或记事本进行编辑，且无须编译，因此在软件开发中，一般不把对配置文件的修改认为是对系统源代码的修改。如果一个系统在扩展时只涉及到修改配置文件，而原有的Java代码或C#代码没有做任何修改，该系统即可认为是一个符合开闭原则的系统。

单一职责原则

单一职责原则是最简单的面向对象设计原则，它用于控制类的粒度大小。单一职责原则定义如下：

> 单一职责原则(Single Responsibility Principle, SRP)：一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。

单一职责原则告诉我们：一个类不能太“累”！

在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中，即将不同的变化原因封装在不同的类中，如果多个职责总是同时发生改变则可将它们封装在同一类中。

> 注意：

> 单一职责原则是实现高内聚、低耦合的指导方针，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验。

下面通过一个简单实例来进一步分析单一职责原则：

> Sunny软件公司开发人员针对某CRM（Customer Relationship Management，客户关系管理）系统中客户信息图形统计模块提出了如图1所示初始设计方案：

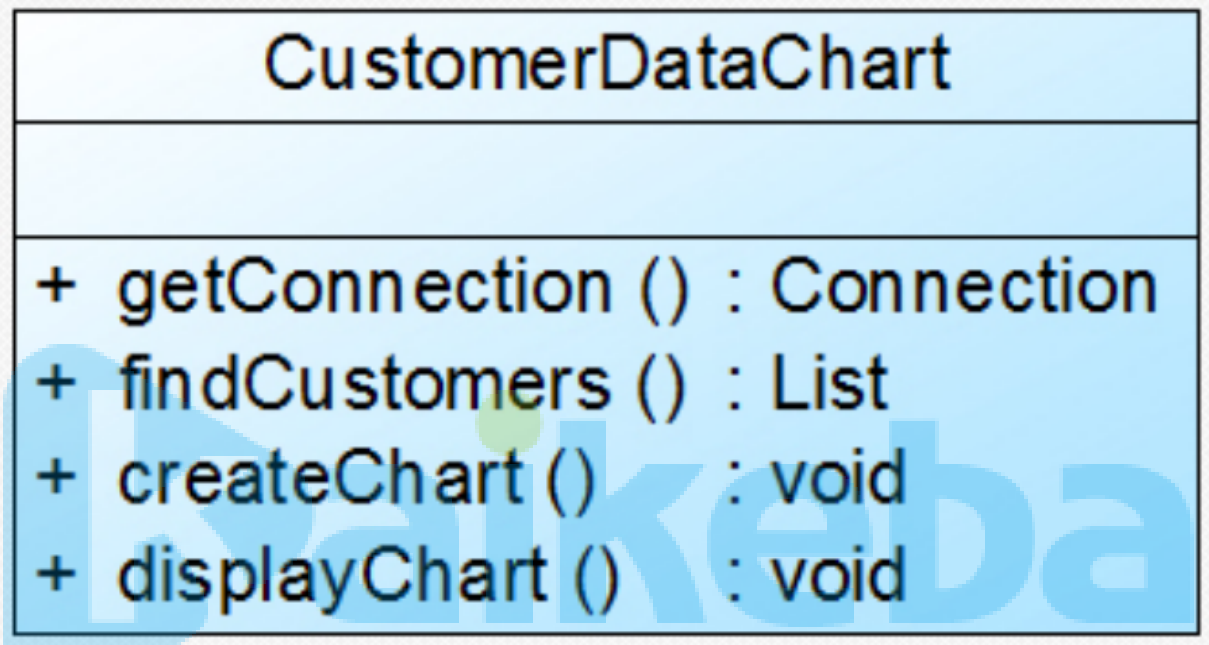


图1 初始设计方案结构图

在图1中，CustomerDataChart类中的方法说明如下：

- getConnection()方法用于连接数据库
- findCustomers()用于查询所有的客户信息
- createChart()用于创建图表
- displayChart()用于显示图表。

现使用单一职责原则对其进行重构。

> 在图1中，CustomerDataChart类承担了太多的职责，既包含与数据库相关的方法，又包含与图表生成和显示相关的方法。
>
> **如果在其他类中也需要连接数据库或者使用findCustomers()方法查询客户信息，则难以实现代码的重用。无论是修改数据库连接方式还是修改图表显示方式都需要修改该类，它不止一个引起它变化的原因，违背了单一职责原则。**因此需要对该类进行拆分，使其满足单一职责原则，类CustomerDataChart可拆分为如下三个类：

- (1) DBUtil：负责连接数据库，包含数据库连接方法getConnection()；
- (2) CustomerDAO：负责操作数据库中的Customer表，包含对Customer表的增删改查等方法，如findCustomers()；

- (3) CustomerDataChart：负责图表的生成和显示，包含方法createChart()和displayChart()。

使用单一职责原则重构后的结构如图2所示：

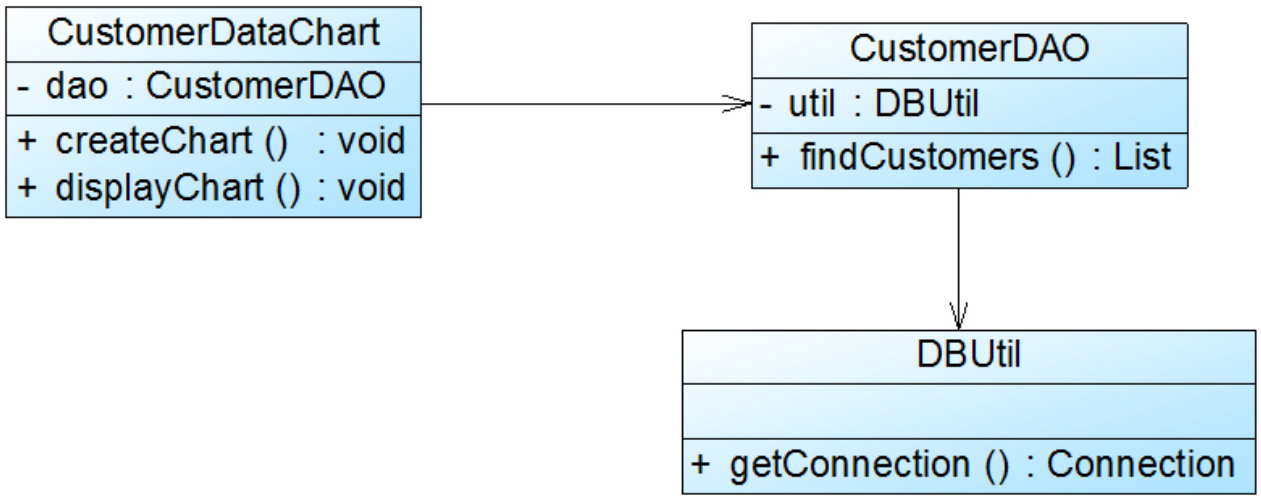


图2 重构后的结构图

里式替换原则

里氏代换原则由2008年图灵奖得主、美国第一位计算机科学女博士Barbara Liskov教授和卡内基·梅隆大学Jeannette Wing教授于1994年提出。其严格表述如下：

> 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1代换o2时，程序P的行为没有变化，那么类型S是类型T的子类型。这个定义比较拗口且难以理解，因此我们一般使用它的另一个通俗版定义：

里氏代换原则(Liskov Substitution Principle, LSP)：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

里氏代换原则告诉我们，**在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。**例如：我喜欢动物，那我一定喜欢狗，因为狗是动物的子类；但是我喜欢狗，不能据此断定我喜欢动物，因为我并不喜欢老鼠，虽然它也是动物。

> 例如有两个类，一个类为BaseClass，另一个是SubClass类，并且SubClass类是BaseClass类的子类，那么一个方法如果可以接受一个BaseClass类型的基类对象base的话，如：method1(base)，那么它必然可以接受一个BaseClass类型的子类对象sub，method1(sub)能够正常运行。反过来的代换不成立，如一个方法method2接受BaseClass类型的子类对象sub为参数：method2(sub)，那么一般而言不可以有method2(base)，除非是重载方法。

里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

在使用里氏代换原则时需要注意如下几个问题：

(1)、子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。根据里氏代换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

(2)、我们在运用里氏代换原则时，尽量把父类设计为抽象类或者接口，让子类继承父类或实现父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。里氏代换原则是开闭原则的具体实现手段之一。

(3)、Java语言中，在编译阶段，Java编译器会检查一个程序是否符合里氏代换原则，这是一个与实现无关的、纯语法意义上的检查，但Java编译器的检查是有局限的。

示例：

> 在Sunny软件公司开发的CRM系统中，客户(Customer)可以分为VIP客户(VIPCustomer)和普通客户(CommonCustomer)两类，系统需要提供一个发送Email的功能，原始设计方案如图1所示：

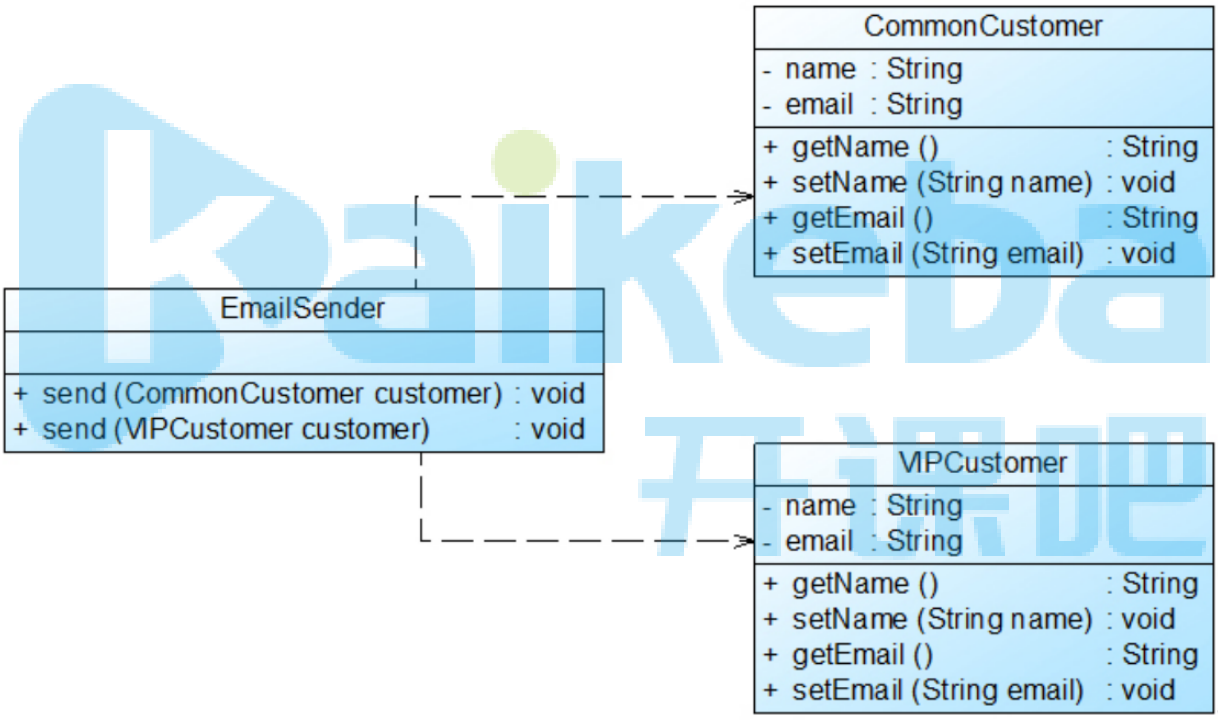


图1原始结构图

在对系统进行进一步分析后发现，无论是普通客户还是VIP客户，发送邮件的过程都是相同的，也就是说两个send()方法中的代码重复，而且在本系统中还将增加新类型的客户。

为了让系统具有更好的扩展性，同时减少代码重复，使用里氏代换原则对其进行重构。

> 在本实例中，可以考虑增加一个新的抽象客户类Customer，而将CommonCustomer和VIPCustomer类作为其子类，邮件发送类EmailSender类针对抽象客户类Customer编程，根据里氏代换原则，能够接受基类对象的地方必然能够接受子类对象，因此将EmailSender中的send()方法的参数类型改为Customer，如果需要增加新类型的客户，只需将其作为Customer类的子类即可。重构后的结构如图2所示：

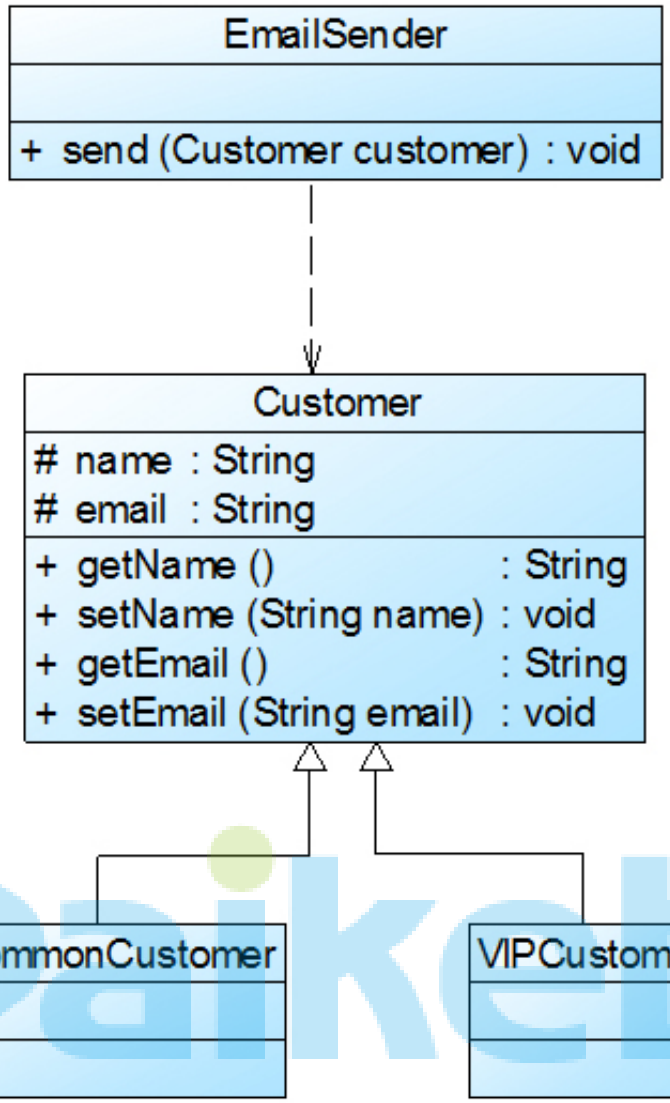


图2 重构后的结构图

里氏代换原则是实现开闭原则的重要方式之一。在本实例中，在传递参数时使用基类对象，除此以外，在定义成员变量、定义局部变量、确定方法返回类型时都可使用里氏代换原则。针对基类编程，在程序运行时再确定具体子类。

依赖倒转原则

如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要实现机制之一，它是系统抽象化的具体实现。依赖倒转原则是**Robert C. Martin**在1996年为“C++Reporter”所写的专栏Engineering Notebook的第三篇，后来加入到他在2002年出版的经典著作“**Agile Software Development, Principles, Patterns, and Practices**”一书中。依赖倒转原则定义如下：

> 依赖倒转原则(Dependency Inversion Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。

依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加的新方法。

在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中，这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，满足开闭原则的要求。

在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过**依赖注入(DependencyInjection, DI)**的方式注入到其他对象中，**依赖注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象**。常用的注入方式有三种，分别是：**构造注入，设值注入（Setter注入）和接口注入**。构造注入是指通过构造函数来传入具体类的对象，设值注入是指通过Setter方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象来覆盖父类对象。

下面通过一个简单实例来加深对依赖倒转原则的理解：

Sunny软件公司开发人员在开发某CRM系统时发现：该系统经常需要将存储在TXT或Excel文件中的客户信息转存到数据库中，因此需要进行数据格式转换。在客户数据操作类中将调用数据格式转换类的方法实现格式转换和数据库插入操作，初始设计方案结构如图1所示：

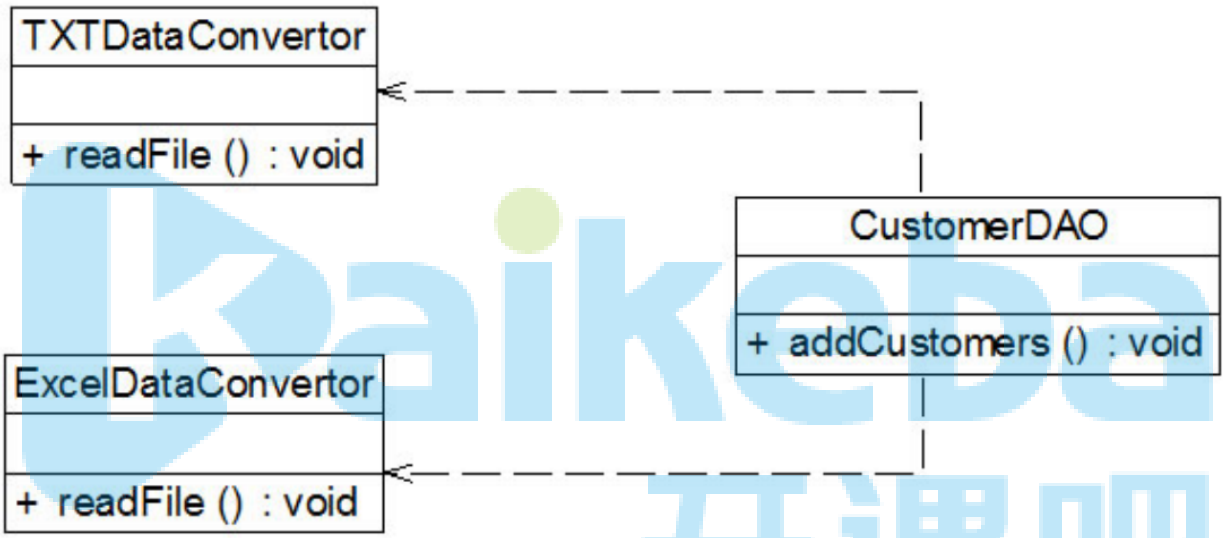


图1 初始设计方案结构图

在编码实现图1所示结构时，Sunny软件公司开发人员发现该设计方案存在一个非常严重的问题，由于每次转换数据时数据来源不一定相同，因此需要更换数据转换类，如有时候需要将TXTDataConvertor改为ExcelDataConvertor，此时，需要修改CustomerDAO的源代码，而且在引入并使用新的数据转换类时也不得不修改CustomerDAO的源代码，系统扩展性较差，违反了开闭原则，现需要对该方案进行重构。

在本实例中，由于CustomerDAO针对具体数据转换类编程，因此在增加新的数据转换类或者更换数据转换类时都不得不修改CustomerDAO的源代码。我们可以通过引入抽象数据转换类解决该问题，在引入抽象数据转换类DataConvertor之后，CustomerDAO针对抽象类DataConvertor编程，而将具体数据转换类名存储在配置文件中，符合依赖倒转原则。根据里氏代换原则，程序运行时，具体数据转换类对象将替换DataConvertor类型的对象，程序不会出现任何问题。更换具体数据转换类时无须修改源代码，只需要修改配置文件；如果需要增加新的具体数据转换类，只要将新增数据转换类作为DataConvertor的子类并修改配置文件即可，原有代码无须做任何修改，满足开闭原则。重构后的结构如图2所示：

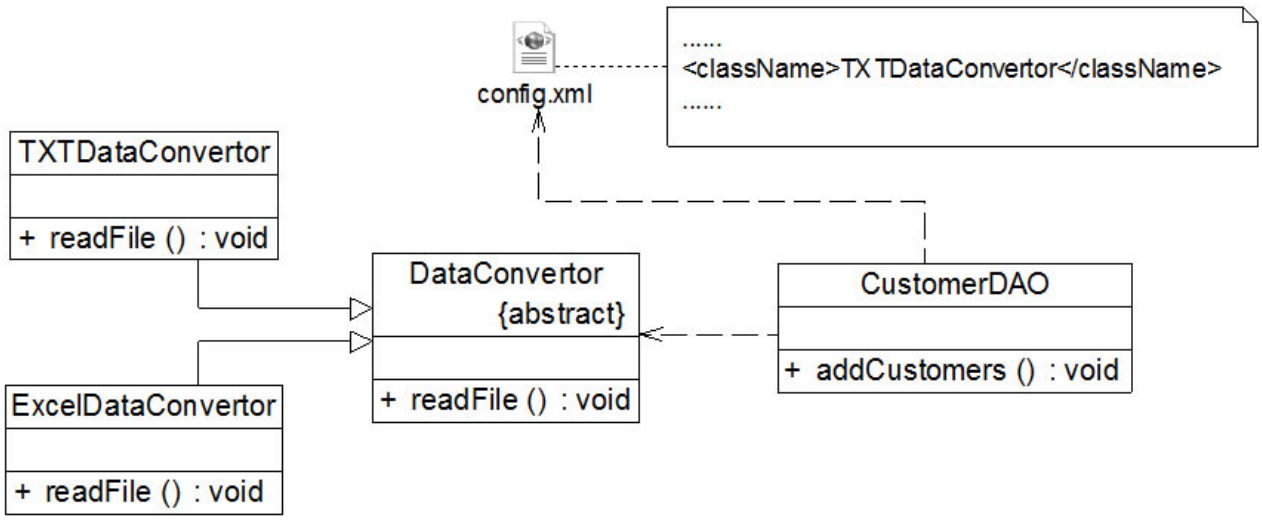


图2重构后的结构图

在上述重构过程中，我们使用了开闭原则、里氏代换原则和依赖倒转原则，在大多数情况下，这三个设计原则会同时出现，**开闭原则是目标，里氏代换原则是基础，依赖倒转原则是手段**，它们相辅相成，相互补充，目标一致，只是分析问题时所站角度不同而已。

接口分离原则

接口隔离原则定义如下：

> 接口隔离原则(Interface Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

根据接口隔离原则，当一个接口太大时，我们需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。**每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干。**这里的“接口”往往有两种不同的含义：一种是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另外一种是指某种语言具体的“接口”定义，有严格的定义和结构，比如Java语言中的interface。对于这两种不同的含义，ISP的表达方式以及含义都有所不同：

(1) 当把“接口”理解成一个类型所提供的所有方法特征的集合的时候，这就是一种逻辑上的概念，接口的划分将直接带来类型的划分。可以把接口理解成角色，一个接口只能代表一个角色，每个角色都有它特定的一个接口，此时，这个原则可以叫做“**角色隔离原则**”。

(2) 如果把“接口”理解成狭义的特定语言的接口，那么ISP表达的意思是指**接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。**在面向对象编程语言中，实现一个接口就需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“**定制服务**”，即为不同的客户端提供宽窄不同的接口。

下面通过一个简单实例来加深对接口隔离原则的理解：

Sunny软件公司开发人员针对某CRM系统的客户数据显示模块设计了如图1所示接口，其中方法dataRead()用于从文件中读取数据，方法transformToXML()用于将数据转换成XML格式，方法createChart()用于创建图表，方法displayChart()用于显示图表，方法createReport()用于创建文字报表，方法displayReport()用于显示文字报表。

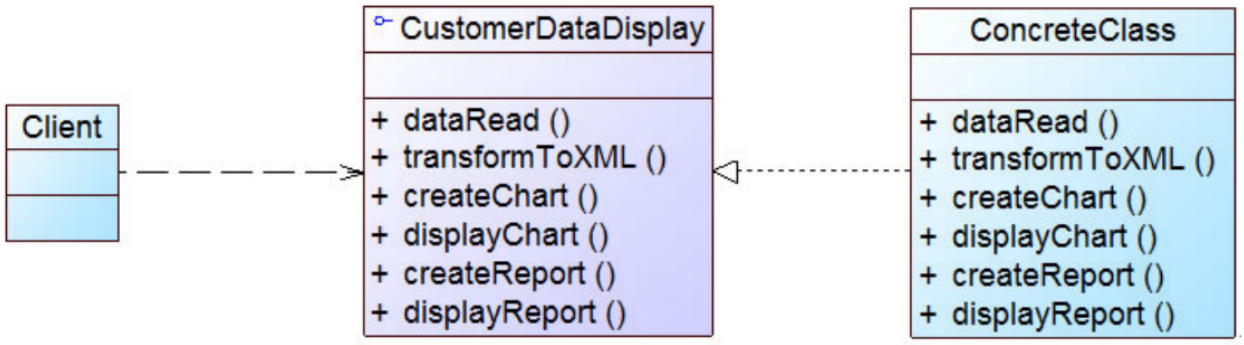


图1 初始设计方案结构图

在实际使用过程中发现该接口很不灵活，例如如果一个具体的数据显示类无须进行数据转换（源文件本身就是XML格式），但由于实现了该接口，将不得不实现其中声明的transformToXML()方法（至少需要提供一个空实现）；如果需要创建和显示图表，除了需实现与图表相关的方法外，还需要实现创建和显示文字报表的方法，否则程序编译时将报错。　　现使用接口隔离原则对其进行重构。

在图1中，由于在接口CustomerDataDisplay中定义了太多方法，即该接口承担了太多职责，一方面导致该接口的实现类很庞大，在不同的实现类中都不得不实现接口中定义的所有方法，灵活性较差，如果出现大量的空方法，将导致系统中产生大量的无用代码，影响代码质量；另一方面由于客户端针对大接口编程，将在一定程序上破坏程序的封装性，客户端看到了不应该看到的方法，没有为客户端定制接口。因此需要将该接口按照接口隔离原则和单一职责原则进行重构，将其中的一些方法封装在不同的小接口中，确保每一个接口使用起来都较为方便，并都承担某一单一角色，每个接口中只包含一个客户端（如模块或类）所需的方法即可。

通过使用接口隔离原则，本实例重构后的结构如图2所示：

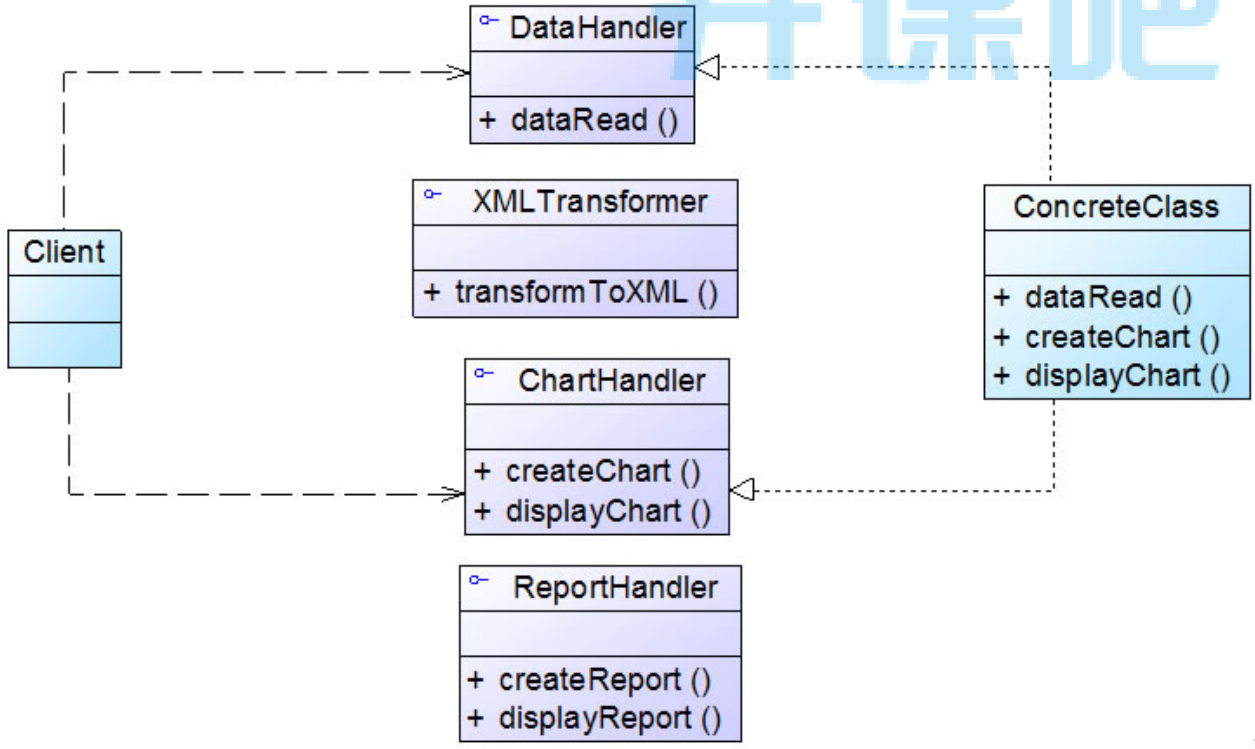


图2 重构后的结构图

在使用接口隔离原则时，我们需要注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来很不方便。一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强迫客户依赖于那些它们不用的方法。

合成复用原则

合成复用原则又称为组合/聚合复用原则(Composition/Aggregate Reuse Principle, CARP)，其定义如下：

> 合成复用原则(Composite Reuse Principle, CRP)：尽量使用对象组合，而不是继承来达到复用的目的。

合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能的目的。简言之：**复用时要尽量使用组合/聚合关系（关联关系），少用继承。**

在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又称“白箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；而且继承只能在有限的环境中使用（如类没有声明为不能被继承）。

由于组合或聚合关系可以将已有的对象（也可称为成员对象）纳入到新对象中，使之成为新对象的一部分，因此新对象可以调用已有对象的功能，这样做可以使得成员对象的内部实现细节对于新对象不可见，所以这种复用又称为“黑箱”复用，相对继承关系而言，其耦合度相对较低，成员对象的变化对新对象的影响不大，可以在新对象中根据实际需要有选择性地调用成员对象的操作；合成复用可以在运行时动态进行，新对象可以动态地引用与成员对象类型相同的其他对象。

一般而言，如果两个类之间是“Has-A”的关系应使用组合或聚合，如果是“Is-A”关系可使用继承。"Is-A"是严格的分类学意义上的定义，意思是一个类是另一个类的"一种"；而"Has-A"则不同，它表示某一个角色具有某一项责任。

案例讲解

Sunny软件公司开发人员在初期的CRM系统设计中，考虑到客户数量不多，系统采用MySQL作为数据库，与数据库操作有关的类如CustomerDAO类等都需要连接数据库，连接数据库的方法getConnection()封装在DBUtil类中，由于需要重用DBUtil类的getConnection()方法，设计人员将CustomerDAO作为DBUtil类的子类，初始设计方案结构如图1所示：

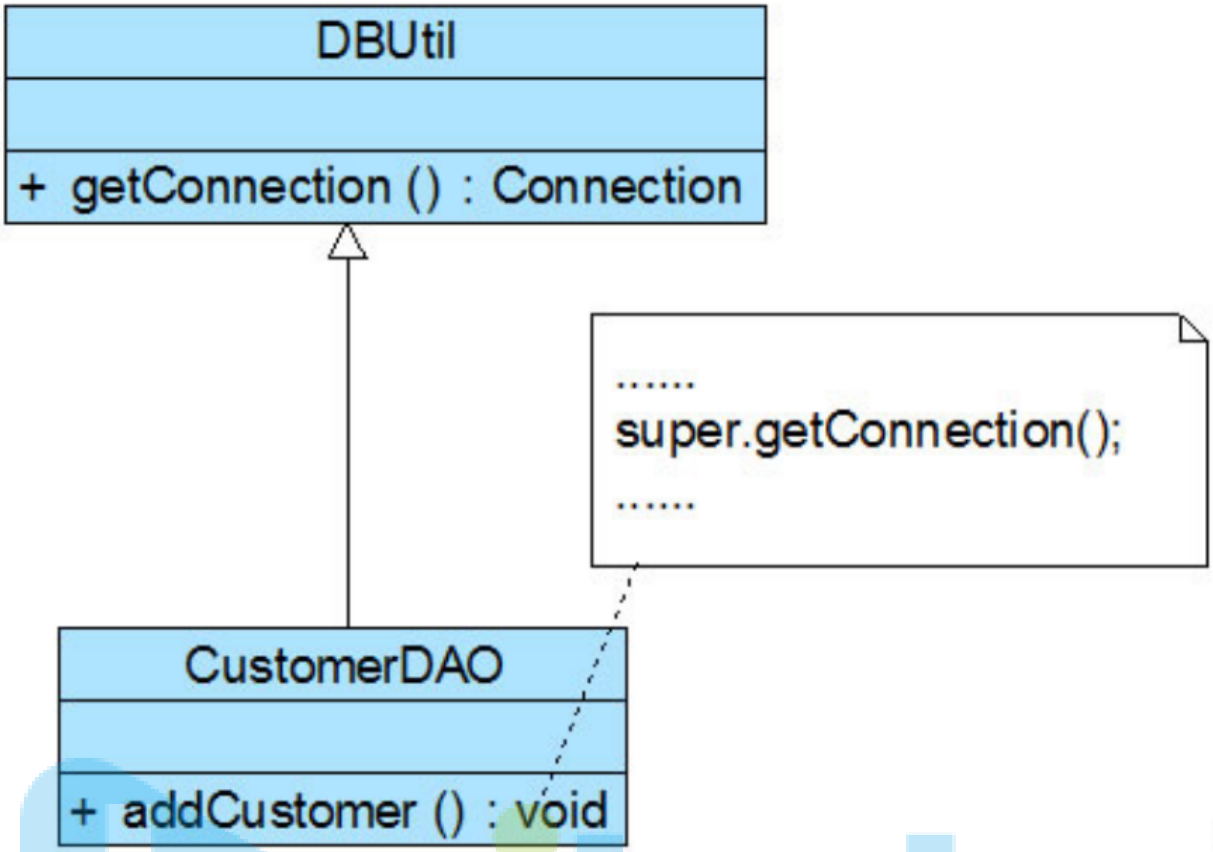


图1 初始设计方案结构图

随着客户数量的增加，系统决定升级为Oracle数据库，因此需要增加一个新的OracleDBUtil类来连接Oracle数据库，由于在初始设计方案中CustomerDAO和DBUtil之间是继承关系，因此在更换数据库连接方式时需要修改CustomerDAO类的源代码，将CustomerDAO作为OracleDBUtil的子类，这将违反开闭原则。【当然也可以修改DBUtil类的源代码，同样会违反开闭原则。】　　现使用合成复用原则对其进行重构。

根据合成复用原则，我们在实现复用时应该多用关联，少用继承。因此在本实例中我们可以使用关联复用来取代继承复用，重构后的结构如图2所示：

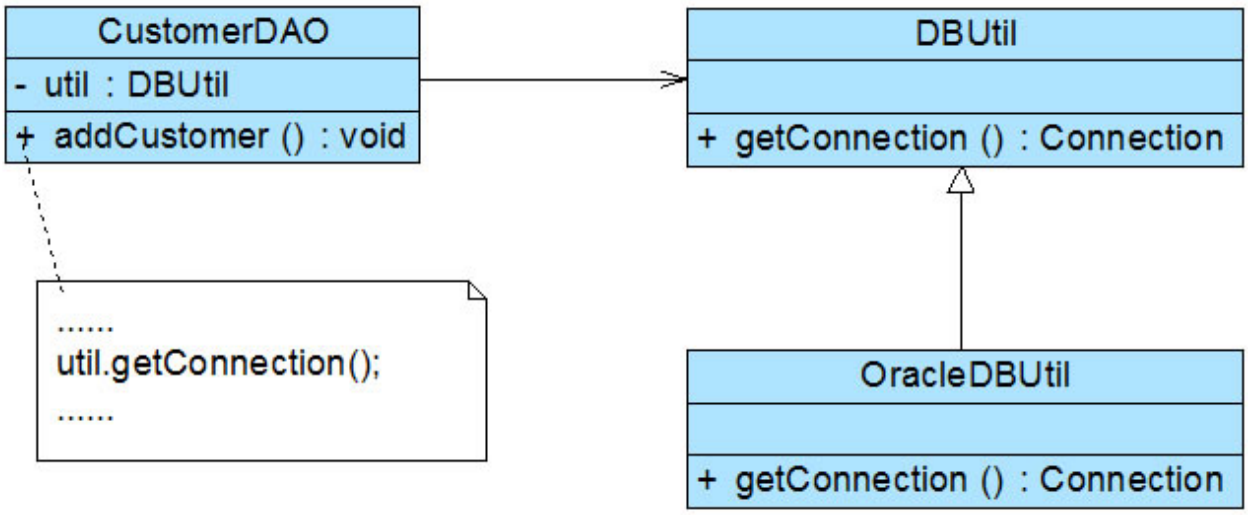


图2 重构后的结构图

在图2中，CustomerDAO和DBUtil之间的关系由继承关系变为关联关系，采用依赖注入的方式将DBUtil对象注入到CustomerDAO中，可以使用构造注入，也可以使用Setter注入。如果需要对DBUtil的功能进行扩展，可以通过其子类来实现，如通过子类OracleDBUtil来连接Oracle数据库。由于CustomerDAO针对DBUtil编程，根据里氏代换原则，DBUtil子类的对象可以覆盖DBUtil对象，只需在CustomerDAO中注入子类对象即可使用子类所扩展的方法。例如在CustomerDAO中注入OracleDBUtil对象，即可实现Oracle数据库连接，原有代码无须进行修改，而且还可以很灵活地增加新的数据库连接方式。

迪米特原则

迪米特法则来自于1987年美国东北大学(Northeastern University)一个名为“Demeter”的研究项目。迪米特法则又称为最少知识原则(LeastKnowledge Principle, LKP)，其定义如下：

```
迪米特法则(Law of  Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用。
```

如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。**迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。**

迪米特法则还有几种定义形式，包括：**不要和“陌生人”说话、只与你的直接朋友通信**等，在迪米特法则中，对于一个对象，其朋友包括以下几类：

(1) 当前对象本身(this)；

(2) 以参数形式传入到当前对象方法中的对象；

(3) 当前对象的成员对象；

(4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

(5) 当前对象所创建的对象。

任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。在应用迪米特法则时，一个对象只能与直接朋友发生交互，不要与“陌生人”发生直接交互，这样做可以降低系统的耦合度，一个对象的改变不会给太多其他对象带来影响。

迪米特法则要求我们在设计系统时，**应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用**。简言之，就是**通过引入一个合理的第三者来降低现有对象之间的耦合度**。

在将迪米特法则运用到系统设计中时，要注意下面的几点：**在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。**

案例讲解

有一个学校，下属有各个学院和总部，现要求打印出学校总部员工ID和学院员工ID。代码如下：

```
//学校总部员工类
public class Employee{
    private String id;
    public String getId(){
        return id;
    }

    public void setId(String id){
        this.id = id;
    }
}
```

```
//学院的员工类
public class CollegeEmployee{
    private String id;
    public String getId(){
        return id;
    }

    public void setId(String id){
        this.id = id;
    }
}
```

//没有违反迪米特法则

```
//管理学院员工的管理类
public class CollegeManager{
    public List<CollegeEmployee> listCollegeEmployee(){
        public List<CollegeEmployee> list = new ArrayList<>();

        for(int = 0; i < 10; i++){
            CollegeEmployee e = new CollegeEmployee();
            e.setId("学院员工id: " + id);
            list.add(e);
        }
        return list;
    }
}
```

```
//学校管理类
public class SchoolManager{
```

```
    //返回总部所有员工
    public List<Employee> listEmployee(){
        List<Employee> list = new ArrayList<>();
      for(int i = 0; i< 5; i++){
            Employee emp = new Employee();
            emp.setId("学校总部员工id: " + i);
            list.add(emp);
        }
        return list;
    }

    //输出学校总部和学院员工信息
    public void printEmployee(CollegeManager sub){
        List<CollegeEmployee> collegeEmpList = sub.listCollegeEmployee();
        System.out.println("------------学院员工----------------");
        for(CollegeEmployee e collegeEmpList){
            System.out.println(e.getId);
        }

        List<Employee> empList = this.listEmployee();
        System.out.println("------------学校总部员工----------------");
        for(Employee e empList){
            System.out.println(e.getId);
        }
    }
}
```

分析：

在类SchoolManager中，根据直接朋友的定义可知，类Employee和类CollegeManager是直接朋友，类Employee是以方法的返回值作为直接朋友，而类CollegeManager是以方法的参数作为直接朋友。

类CollegeEmployee不是SchoolManager的直接朋友，因为CollegeEmployee是以局部变量方式出现在SchoolManager，违反了迪米特法则。

按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合，对上面的SchoolManager、CollegeManager类进行改进，如下：

```
//管理学院员工的管理类
public class CollegeManager{
    public List<CollegeEmployee> listCollegeEmployee(){
        public List<CollegeEmployee> list = new ArrayList<>();

        for(int = 0; i < 10; i++){
            CollegeEmployee e = new CollegeEmployee();
            e.setId("学院员工id: " + id);
            list.add(e);
        }
```

```
        return list;
    }

    public void printCollegeEmp(){
        List<CollegeEmployee> collegeEmpList = this.listCollegeEmployee();
        System.out.println("------------学院员工----------------");
        for(CollegeEmployee e collegeEmpList){
            System.out.println(e.getId);
        }
    }
}
```

```
//学校管理类
public class SchoolManager{
    //返回总部所有员工
    public List<Employee> listEmployee(){
        List<Employee> list = new ArrayList<>();
      for(int i = 0; i< 5; i++){
            Employee emp = new Employee();
            emp.setId("学校总部员工id: " + i);
            list.add(emp);
        }
        return list;
    }

    //输出学校总部和学院员工信息
    public void printEmployee(CollegeManager sub){
        sub.printCollegeEmp();

        List<Employee> empList = this.listEmployee();
        System.out.println("------------学校总部员工----------------");
        for(Employee e empList){
            System.out.println(e.getId);
        }
    }
}
```

23种设计模式

设计模式（Design Pattern）是前辈们对代码开发经验的总结，是解决特定问题的一系列套路。它不是语法规定，而是一套用来提高代码可复用性、可维护性、可读性、稳健性以及安全性的解决方案。

1995 年，GoF（Gang of Four，四人组/四人帮）合作出版了《设计模式：可复用面向对象软件的基础》一书，共收录了 **23 种设计模式**，从此树立了软件设计模式领域的里程碑，人称「GoF设计模式」。

设计模式概述

- **设计模式（Design pattern）代表了最佳的实践**，通常被有经验的面向对象的软件开发人员所采用。
- 设计模式是软件开发人员在软件开发过程中面临的一般**问题的解决方案**。这些解决方案是众多软件开发人员经过相当长的一段时间的**试验和错误总结出来**的。
- 设计模式是一套被**反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。**使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。
- 设计模式不是一种方法和技术，而**是一种思想**。
- **设计模式和具体的语言无关**，学习设计模式就是要建立面向对象的思想，尽可能的面向接口编程，低耦合，高内聚，**使设计的程序可复用**。
- 学习设计模式能够促进对面向对象思想的理解，反之亦然。它们相辅相成。

设计模式的类型

总体来说，设计模式按照功能分为**三类23种**：

- **创建型（5种）**： 工厂模式、抽象工厂模式、单例模式（重点）、原型模式、构建者模式
- **结构型（7种）**：适配器模式、装饰模式、代理模式（重点）、外观模式、桥接模式、组合模式、享元模式
- **行为型（11种）**：模板方法模式、策略模式、观察者模式、中介者模式、状态模式、**责任链模式**、命令模式、迭代器模式、访问者模式、解释器模式、备忘录模式

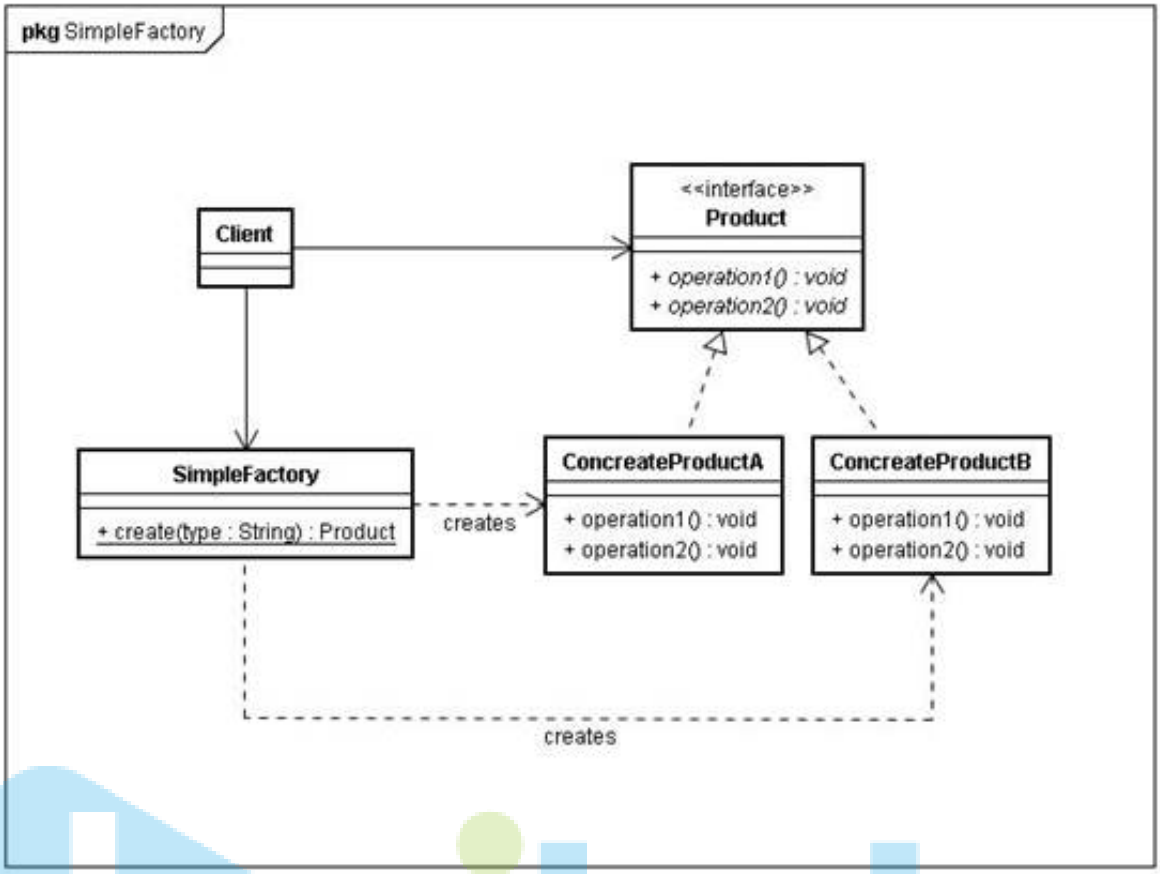
创建型设计模式

简单工厂模式

介绍

工厂类拥有一个工厂方法（create），接受了一个参数，通过不同的参数实例化不同的产品类。

图示



优缺点

- **优点：**
  - 很明显，简单工厂的特点就是“简单粗暴”，通过一个含参的工厂方法，我们可以实例化任何产品类，上至飞机火箭，下至土豆面条，无所不能。
  - 所以简单工厂有一个别名：上帝类。
- **缺点：**
  - 任何”东西“的子类都可以被生产，负担太重。当所要生产产品种类非常多时，工厂方法的代码量可能会很庞大。
  - 在**遵循开闭原则**（对拓展开放，对修改关闭）的条件下，简单工厂对于增加新的产品，无能为力。因为增加新产品只能通过修改工厂方法来实现。

工厂方法正好可以解决简单工厂的这两个缺点。

> 小提示：spring中是通过配置文件和反射解决了简单工厂中的缺点。

示例

- 普通-简单工厂类：

```
public class AnimalFactory {
```

```
  //简单工厂设计模式（负担太重、不符合开闭原则）
  public static Animal createAnimal(String name){
    if ("cat".equals(name)) {
      return new Cat();
    }else if ("dog".equals(name)) {
      return new Dog();
    }else if ("cow".equals(name)) {
      return new Cow();
    }else{
      return null;
    }
  }
}
```

- 静态方法工厂

```
//该简单工厂，也称为静态方法工厂
public class AnimalFactory2 {

  public static Dog createDog(){
    return new Dog();
  }

  public static Cat createCat(){
    return new Cat();
  }
}
```

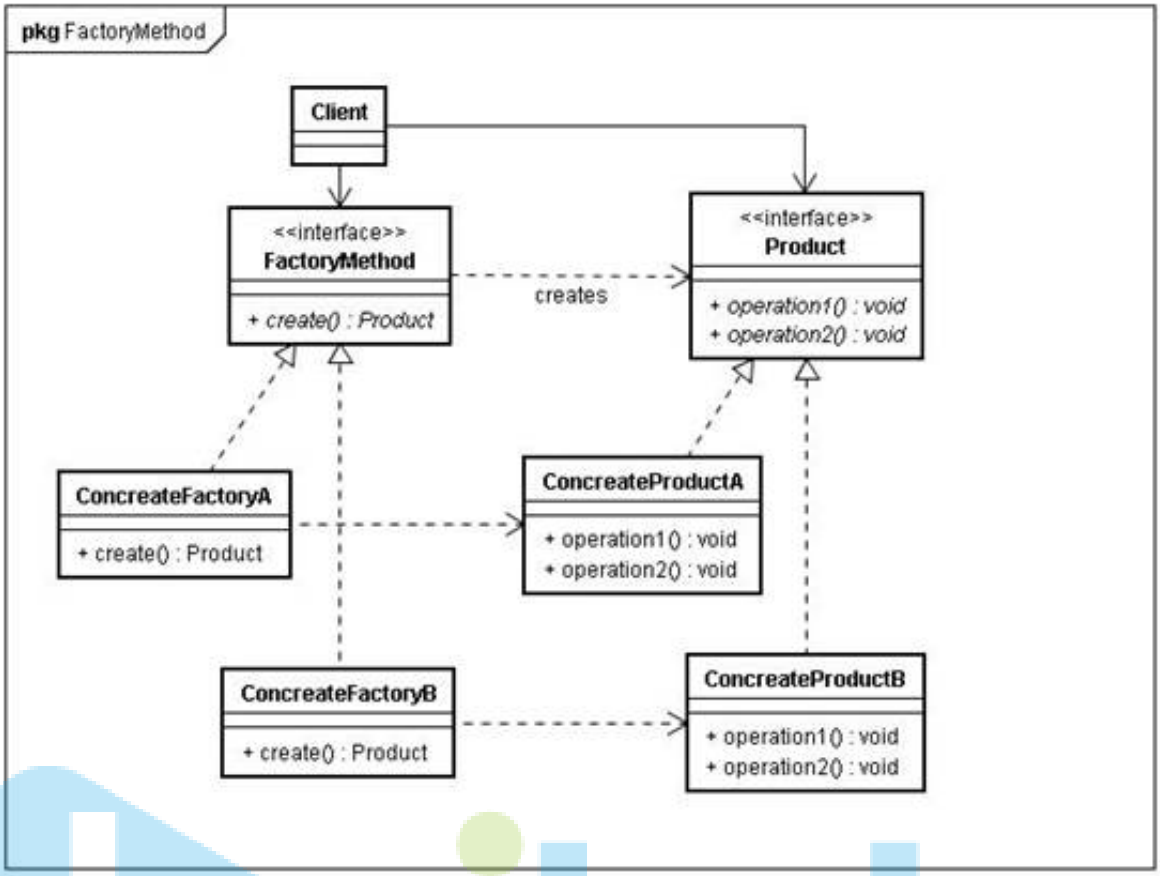

工厂方法模式

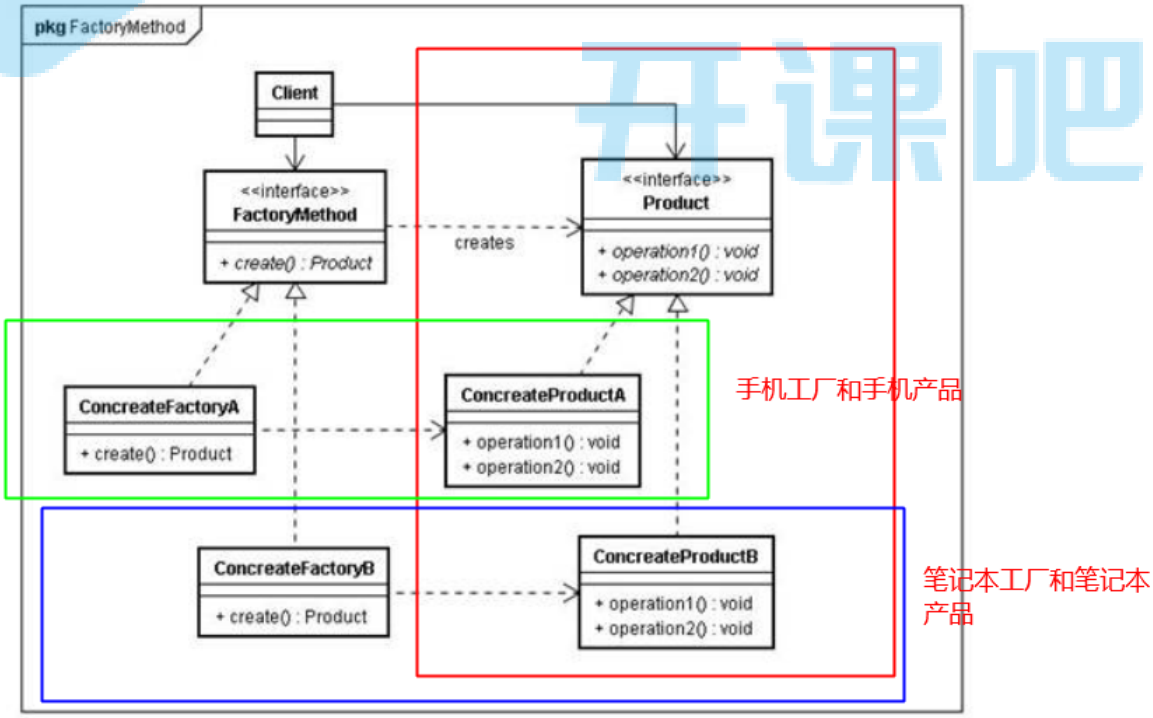
介绍

工厂方法是针对每一种产品提供一个工厂类。

通过不同的工厂实例来创建不同的产品实例。

图示





优缺点

优点：

- 工厂方法模式就很好的减轻了工厂类的负担，把某一类/某一种东西交由一个工厂生产；（对应简单工厂的缺点1）
- 同时增加某一类”东西“并不需要修改工厂类，只需要添加生产这类”东西“的工厂即可，使得工厂类符合开放-封闭原则。

缺点：

- 对于某些可以形成产品族（一组产品）的情况处理比较复杂。

示例

- 抽象出来的工厂对象

```
// 抽象出来的动物工厂----它只负责生产一种产品
public abstract class AnimalFactory {
  // 工厂方法
  public abstract Animal createAnimal();
}
```

- 具体的工厂对象1

```
// 具体的工厂实现类
public class CatFactory extends AnimalFactory {

  @Override
  public Animal createAnimal() {
    return new Cat();
  }
}
```

- 具体的工厂对象2

```
//具体的工厂实现类
public class DogFactory extends AnimalFactory {

  @Override
  public Animal createAnimal() {
    return new Dog();
  }
}
```

抽象工厂模式

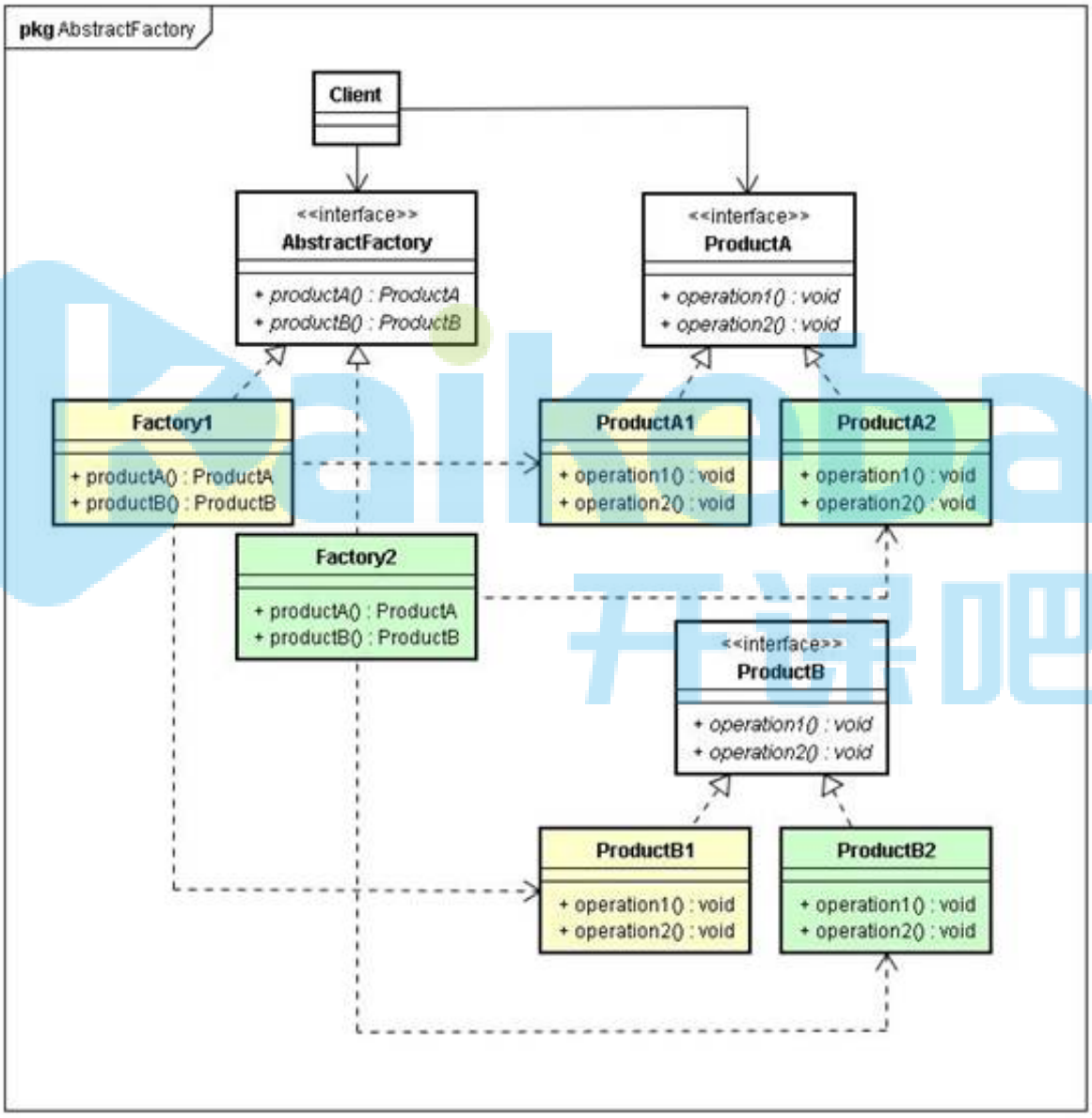
介绍

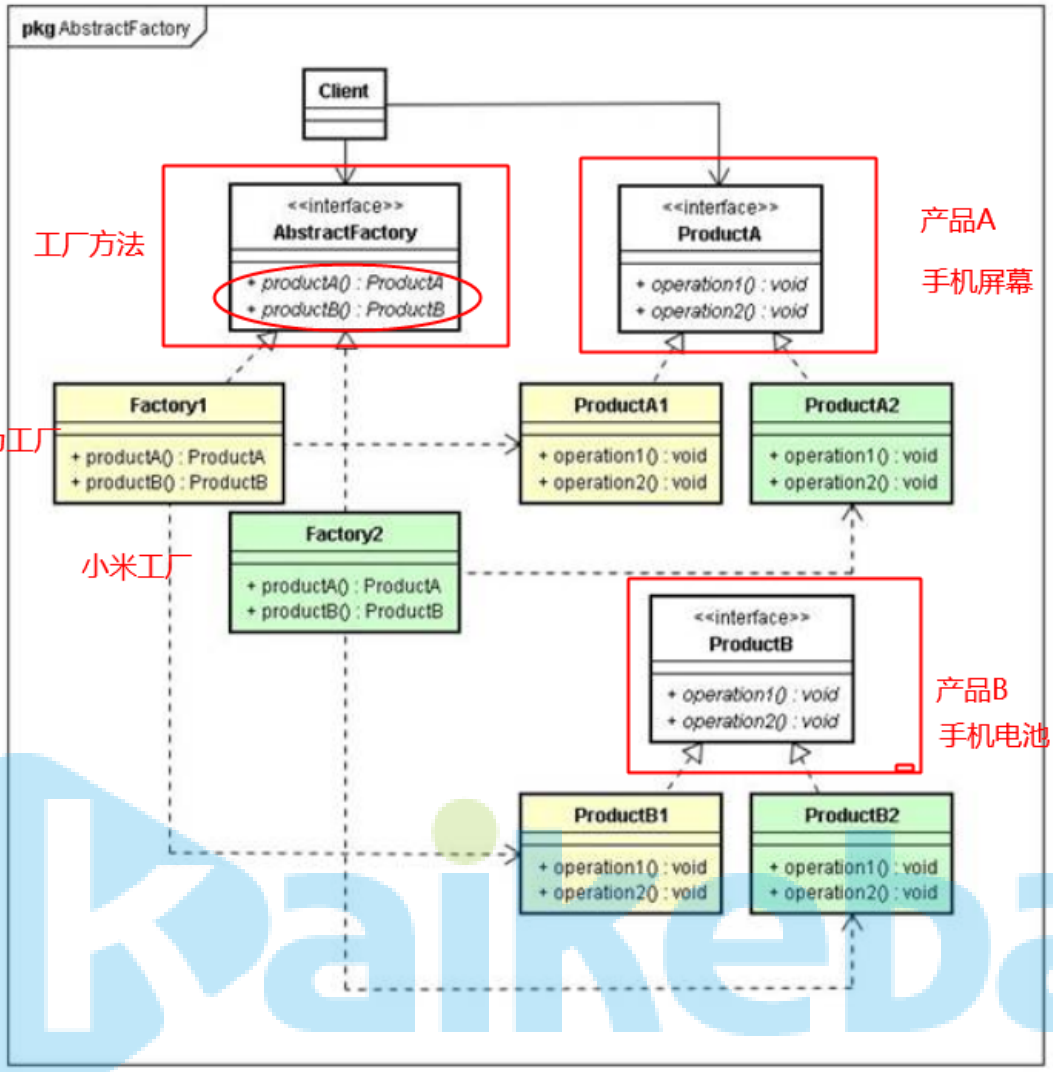
- 抽象工厂是应对**产品族**概念的。

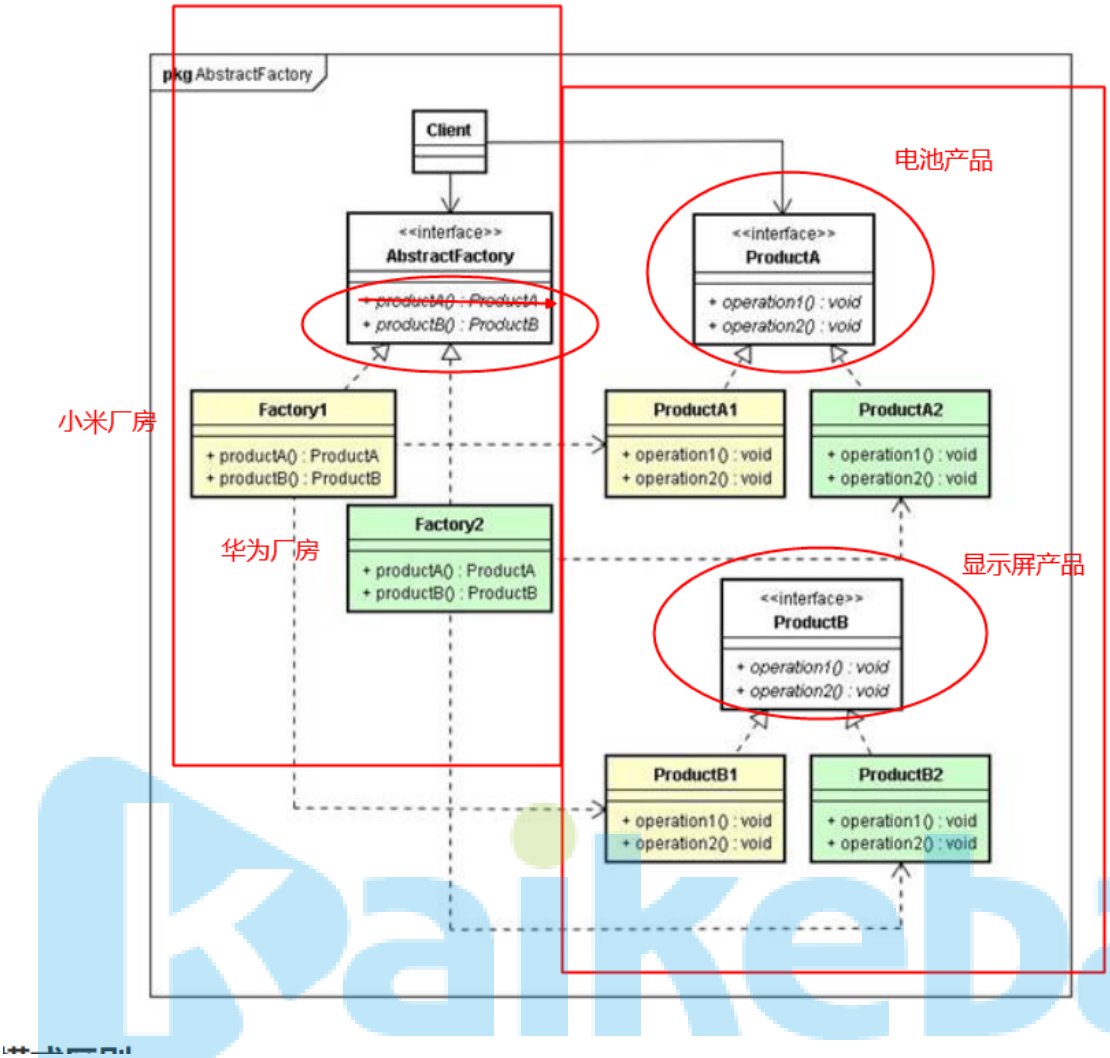
  > 例如，汽车可以分为轿车、SUV、MPV等，也分为奔驰、宝马等。我们可以将奔驰的所有车看作是一个产品族，而将宝马的所有车看作是另一个产品族。分别对应两个工厂，一个是奔驰的工厂，另一个是宝马的工厂。与工厂方法不同，奔驰的工厂不只是生产具体的某一个产品，而是一族产品（奔驰轿车、奔驰SUV、奔驰MPV）。“抽象工厂”的“抽象”指的是就是这个意思。

- 上边的工厂方法模式是一种极端情况的抽象工厂模式（即只生产一种产品的抽象工厂模式），而抽象工厂模式可以看成是工厂方法模式的一种推广。

图示




pkg AbstractFactory
Client
工厂方法
<<interface>>
AbstractFactory
+ productA() : ProductA
+ productB() : ProductB
<<interface>>
ProductA
+ operation1() : void
+ operation2() : void
产品A
手机屏幕
华为工厂
Factory1
+ productA() : ProductA
+ productB() : ProductB
ProductA1
+ operation1() : void
+ operation2() : void
ProductA2
+ operation1() : void
+ operation2() : void
小米工厂
Factory2
+ productA() : ProductA
+ productB() : ProductB
<<interface>>
ProductB
+ operation1() : void
+ operation2() : void
产品B
手机电池
ProductB1
+ operation1() : void
+ operation2() : void
ProductB2
+ operation1() : void
+ operation2() : void



工厂模式区别

- **简单工厂：** 使用一个工厂对象用来生产同一等级结构中的任意产品。（不支持拓展增加产品）
- **工厂方法：** 使用多个工厂对象用来生产同一等级结构中对应的固定产品。（支持拓展增加产品）
- **抽象工厂：** 使用多个工厂对象用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）

示例

待补充

单例模式（面试）

单例模式：
　懒汉式：延迟加载方式
　饿汉式：立即加载

面试绝对面的都是懒汉式的问题。

单例模式如果使用不当，就容易引起线程安全问题。
    * 饿汉式不存在线程安全问题，但是它一般不被使用，因为它会浪费内存的空间
    * 懒汉式会合理使用内存空间，只有第一次被加载的时候，才会真正去创建对象。但是这种方式存在线程安全问题。

懒汉式单例模式的实现方式有三种：
    * 双重检查锁方式（DCL）

介绍

单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。这样的模式有几个好处：

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

示例

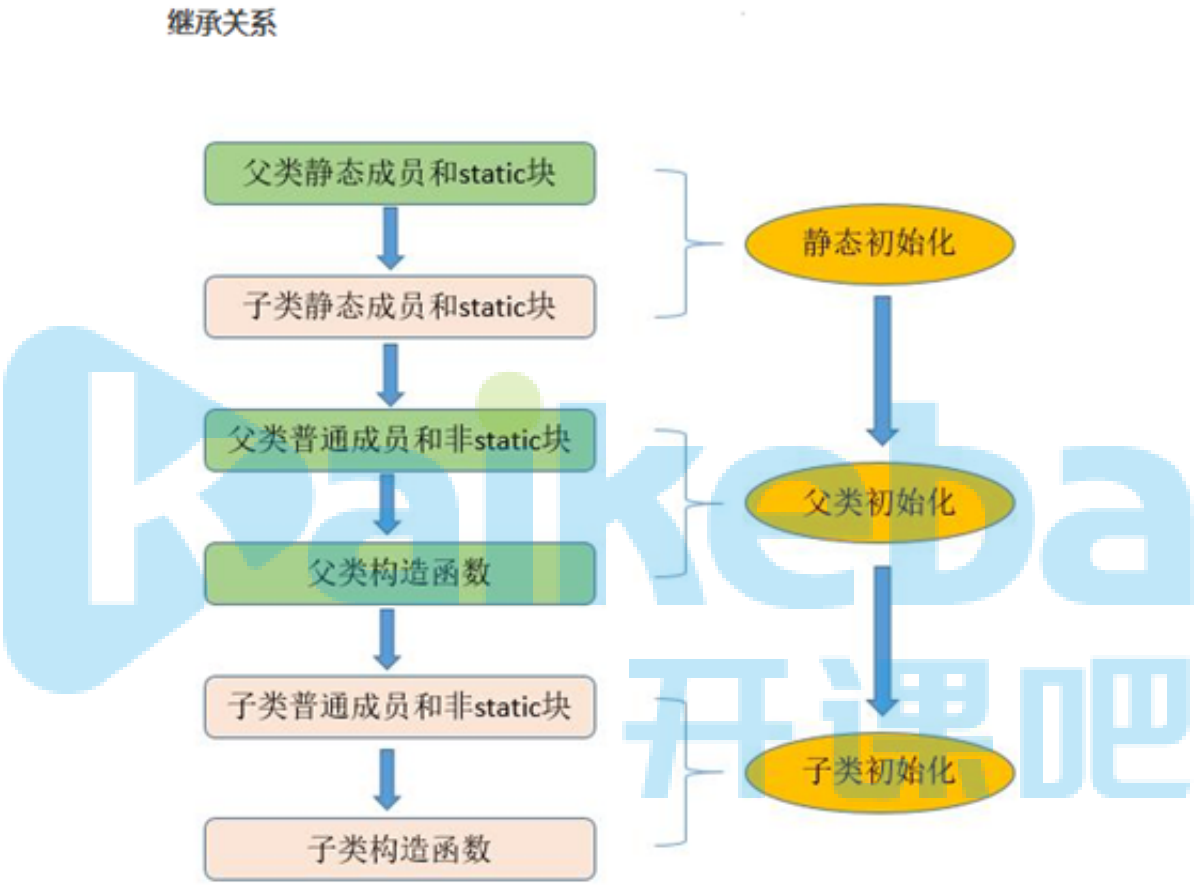
- 饿汉式单例

```
public class Student1 {
  // 2：成员变量初始化本身对象
  private static Student1 student = new Student1();

  // 1：构造私有
  private Student1() {
  }

  // 3：对外提供公共方法获取对象
  public static Student1 getSingletonInstance() {
    return student;
  }
}
```

- 懒汉式单例

```
public class Student5 {

  private Student5() {
  }
  /*
```

```
     * 此处使用一个内部类来维护单例 JVM在类加载的时候，是互斥的，所以可以由此保证线程安全问题
     */
    private static class SingletonFactory {
        private static Student5 student = new Student5();
    }
    /* 获取实例 */
    public static Student5 getSingletonInstance() {
        return SingletonFactory.student;
    }
}
```



Java单例的三种经典实现

双重检查锁（DCL）

```
public class DoubleCheckLockSingleton {
    private static volatile DoubleCheckLockSingleton instance;

    private DoubleCheckLockSingleton() {
        if(instance != null){
      // gun
        }
    }
```

```
    public static DoubleCheckLockSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckLockSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckLockSingleton();
                }
            }
        }
        return instance;
    }

    public void tellEveryone() {
        System.out.println("This is a DoubleCheckLockSingleton " +
this.hashCode());
    }
    private Object readResolve() {
    }
}
```

> 注意：
>
> volatile关键字在此处起了什么作用？
>
> 为何要执行两次`instance == null`判断？

静态内部类

```
public class StaticInnerHolderSingleton {
    // 静态内部类
    private static class SingletonHolder {
        private static final StaticInnerHolderSingleton INSTANCE =
                            new StaticInnerHolderSingleton();
    }

    private StaticInnerHolderSingleton() {}

    public static StaticInnerHolderSingleton getInstance() {
        return SingletonHolder.INSTANCE;
    }

    public void tellEveryone() {
        System.out.println("This is a StaticInnerHolderSingleton" +
this.hashCode());
    }
}
```

> 这种方式是通过什么机制保证**线程安全性**与**延迟加载**的？（注意，这是Java单例的两大要点，必须保证）

枚举

```
public enum EnumSingleton {
    INSTANCE;

    public void tellEveryone() {
        System.out.println("This is an EnumSingleton " + this.hashCode());
    }
}
```

> Java枚举的本质是什么?
>
> 这种方式又是通过什么机制保证线程安全性与延迟加载的?

在Java圣经《Effective Java》中，Joshua Bloch大佬如是说：

> **A single-element enum type is often the best way to implement a singleton.**

为什么说枚举是（一般情况下）最好的Java单例实现呢？他也做出了简单的说明：

> It is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks.

大意就是，枚举单例可以有效防御两种破坏单例（即让单例产生多个实例）的行为：反射攻击与序列化攻击。言外之意就是前两种单例方式都会被破坏。那么我们就拿平时最常用的双重检查锁方式开刀来试试看。

如何破坏一个单例

反射攻击

直接上代码：

```
public class SingletonAttack {
    public static void main(String[] args) throws Exception {
        reflectionAttack();
    }

    public static void reflectionAttack() throws Exception {
        //通过反射，获取单例类的私有构造器
        Constructor constructor =
DoubleCheckLockSingleton.class.getDeclaredConstructor();
        //设置私有成员的暴力破解
        constructor.setAccessible(true);
```

```
        // 通过反射去创建单例类的多个不同的实例
        DoubleCheckLockSingleton s1 =
(DoubleCheckLockSingleton)constructor.newInstance();
        // 通过反射去创建单例类的多个不同的实例
        DoubleCheckLockSingleton s2 =
(DoubleCheckLockSingleton)constructor.newInstance();
        s1.tellEveryone();
        s2.tellEveryone();
        System.out.println(s1 == s2);
    }
}
```

执行结果如下：

```
This is a DoubleCheckLockSingleton 1368884364
This is a DoubleCheckLockSingleton 401625763
false
```

这种方法非常简单暴力，通过反射侵入单例类的私有构造方法并强制执行，使之产生多个不同的实例，这样单例就被破坏了。

序列化攻击

这种攻击方式只对实现了Serializable接口的单例有效，但偏偏有些单例就是必须序列化的。现在假设DoubleCheckLockSingleton类已经实现了该接口，上代码：

```
public class SingletonAttack {
    public static void main(String[] args) throws Exception {
        serializationAttack();
    }

    public static void serializationAttack() throws Exception {
        // 对象序列化流去对对象进行操作
        ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream("serFile"));
        //通过单例代码获取一个对象
        DoubleCheckLockSingleton s1 = DoubleCheckLockSingleton.getInstance();
        //将单例对象，通过序列化流，序列化到文件中
        outputStream.writeObject(s1);

    // 通过序列化流，将文件中序列化的对象信息读取到内存中
        ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream(new File("serFile")));
        //通过序列化流，去创建对象
```

```
        DoubleCheckLockSingleton s2 =
(DoubleCheckLockSingleton)inputStream.readObject();
        s1.tellEveryone();
        s2.tellEveryone();

        System.out.println(s1 == s2);
    }
}
```

执行结果如下：

```
This is a DoubleCheckLockSingleton 777874839
This is a DoubleCheckLockSingleton 254413710
false
```

为什么会发生这种事？长话短说，在`ObjectInputStream.readObject()`方法执行时，其内部方法`readOrdinaryObject`()中有这样一句话：

```
//其中desc是类描述符
obj = desc.isInstantiable() ? desc.newInstance() : null;
```

也就是说，如果一个实现了Serializable/Externalizable接口的类可以在运行时实例化，那么就调用newInstance()方法，使用其默认构造方法反射创建新的对象实例，自然也就破坏了单例性。要防御序列化攻击，就得将instance声明为transient，并且在单例中加入以下语句：

```
private Object readResolve() {
    return instance;
}
```

这是因为在上述readOrdinaryObject()方法中，会通过卫语句`desc.hasReadResolveMethod()`检查类中是否存在名为readResolve()的方法，如果有，就执行`desc.invokeReadResolve(obj)`调用该方法。readResolve()会用自定义的反序列化逻辑覆盖默认实现，因此强制它返回instance本身，就可以防止产生新的实例。

枚举单例的防御机制

对反射的防御

我们直接将上述reflectionAttack()方法中的双重检查锁方式的类名改成EnumSingleton并执行，会发现报如下异常：

```
Exception in thread "main" java.lang.NoSuchMethodException:
me.lmagics.singleton.EnumSingleton.<init>()
    at java.lang.Class.getConstructor0(Class.java:3082)
    at java.lang.Class.getDeclaredConstructor(Class.java:2178)
    at
me.lmagics.singleton.SingletonAttack.reflectionAttack(SingletonAttack.java:35)
    at me.lmagics.singleton.SingletonAttack.main(SingletonAttack.java:19)
```

这是因为所有Java枚举都隐式继承自Enum抽象类，而Enum抽象类根本没有无参构造方法，只有如下一个构造方法：

```
    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }
```

那么我们就改成获取这个有参构造方法，即：

`Constructor constructor = EnumSingleton.class.getDeclaredConstructor(String.class, int.class);`

结果还是会抛出异常：

```
Exception in thread "main" java.lang.IllegalArgumentException: Cannot
reflectively create enum objects
    at java.lang.reflect.Constructor.newInstance(Constructor.java:417)
    at
me.lmagics.singleton.SingletonAttack.reflectionAttack(SingletonAttack.java:38)
    at me.lmagics.singleton.SingletonAttack.main(SingletonAttack.java:19)
```

来到Constructor.newInstance()方法中，有如下语句：

```
    if ((clazz.getModifiers() & Modifier.ENUM) != 0)
        throw new IllegalArgumentException("Cannot reflectively create enum
objects");
```

可见，JDK反射机制内部完全禁止了用反射创建枚举实例的可能性。

对序列化的防御

如果将serializationAttack()方法中的攻击目标换成EnumSingleton，那么我们就会发现s1和s2实际上是同一个实例，最终会打印出true。这是因为ObjectInputStream类中，对枚举类型有一个专门的readEnum()方法来处理，其简要流程如下：

- 通过类描述符取得枚举单例的类型EnumSingleton；

- 取得枚举单例中的枚举值的名字（这里是INSTANCE）；
- 调用Enum.valueOf()方法，根据枚举类型和枚举值的名字，获得最终的单例。

这种处理方法与readResolve()方法大同小异，都是以绕过反射直接获取单例为目标。不同的是，枚举对序列化的防御仍然是JDK内部实现的。

综上所述，枚举单例确实是目前最好的单例实现了，不仅写法非常简单，并且JDK能够保证其安全性，不需要我们做额外的工作。

> 反射攻击和反序列化问题是两种最常见的破坏单例模式的手段，前者利用反射机制修改构造函数的可见性然后强行创建一个新的实例，后者是将原先的对象序列化后再反序列化从而生成一个新的实例

原型模式 （面试）

原型模式就是用来进行对象复制的。省掉了堆内存一些复杂的处理流程。
对象复制分为两种情况：深拷贝和浅拷贝

浅拷贝：就是将对象中的简单类型属性和String类型属性进行复制。引用类型的属性复制的只是引用而不是对应的对象。

深拷贝：将对象中的所有信息，都完全复制一份，包括引用类型的对象，也是拷贝为一个新对象，引用地址是不同的。

介绍

> 原型模式虽然是创建型的模式，但是与工厂模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。

示例

- 先创建一个原型类：

```
public class Prototype implements Cloneable {

    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }
}
```

> 很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，此处不再深究。
>
> 在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：
>
> - **浅复制**：将一个对象复制后，**基本数据类型**的变量都会**重新创建**，而**引用类型**，指向的**还是原对象所指向的**。
> - **深复制**：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。**简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。**

- 写一个深浅复制的例子

```
public class Prototype implements Cloneable, Serializable {

    private static final long serialVersionUID = 1L;
    private String string;

    private SerializableObject obj;

    /* 浅复制 */
    public Object clone() throws CloneNotSupportedException {
        Prototype proto = (Prototype) super.clone();
        return proto;
    }

    /* 深复制 */
    public Object deepClone() throws IOException, ClassNotFoundException {

        /* 写入当前对象的二进制流 */
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);

        /* 读出二进制流产生的新对象 */
        ByteArrayInputStream bis = new
ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
```

```
    }

    public SerializableObject getObj() {
        return obj;
    }

    public void setObj(SerializableObject obj) {
        this.obj = obj;
    }

}

class SerializableObject implements Serializable {
    private static final long serialVersionUID = 1L;
}
```

原型模式的优点

1. **根据客户端要求实现动态创建对象，客户端不需要知道对象的创建细节，便于代码的维护和扩展。**
2. **使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。**所以在需要重复地创建相似对象时可以考虑使用原型模式。比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多。

原型模式的注意事项

1. **使用原型模式复制对象不会调用类的构造方法。**因为对象的复制是通过调用Object类的clone方法来完成的，它直接在内存中复制数据，因此不 会调用到类的构造方法。不但构造方法中的代码不会执行，甚至连访问权限都对原型模式无效。还记得单例模式吗？单例模式中，只要将构造方法的访问权限设置为 private型，就可以实现单例。但是clone方法直接无视构造方法的权限，所以，单例模式与原型模式是冲突的。
2. **在使用时要注意深拷贝与浅拷贝的问题。**clone方法只会拷贝对象中的基本的数据类型，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝。如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝。

构建者模式

介绍

建造者模式的定义是：**将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。**

建造者模式的角色定义，在建造者模式中存在以下4个角色：

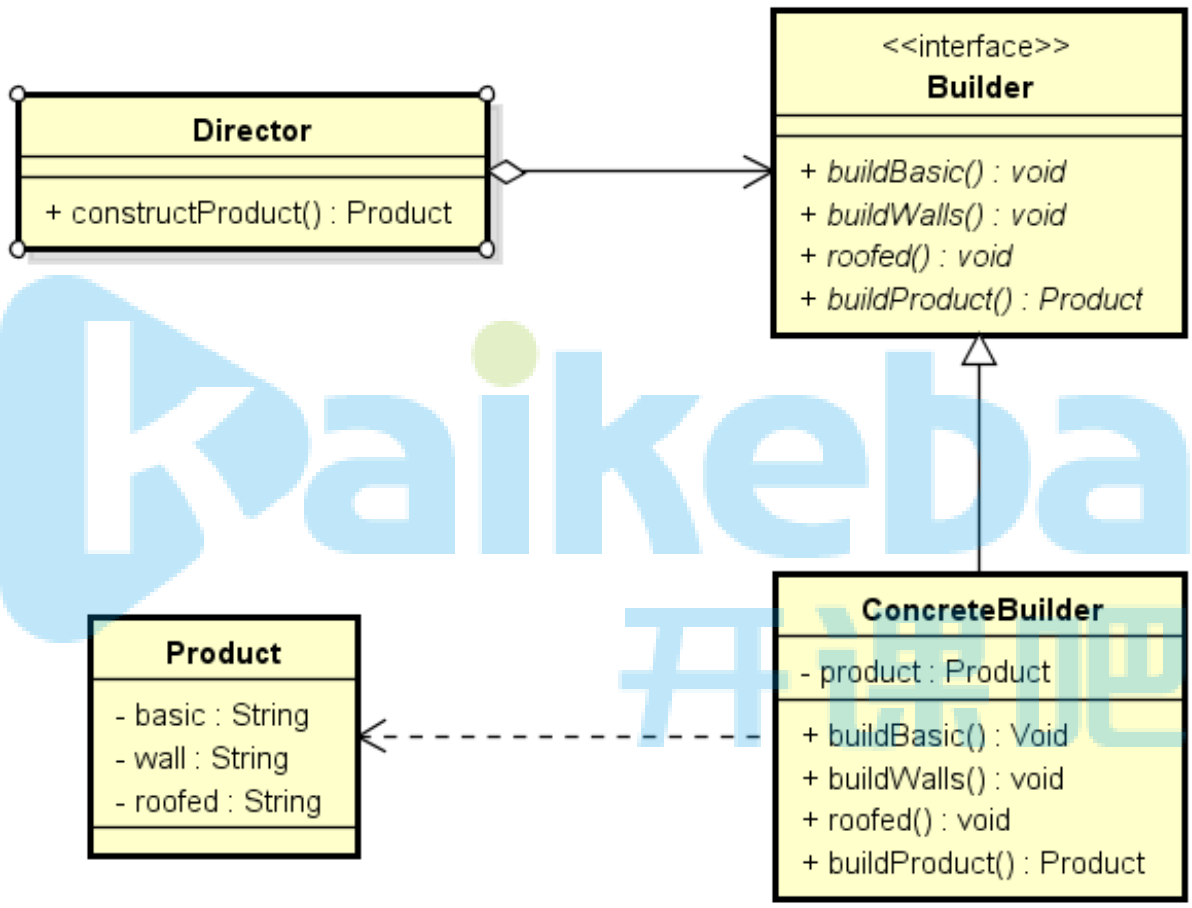
1. builder:为创建一个产品对象的各个部件指定抽象接口。
2. ConcreteBuilder:实现Builder的接口以构造和装配该产品的各个部件，定义并明确它所创建的表

示，并提供一个检索产品的接口。
3. Director:构造一个使用Builder接口的对象。
4. Product:表示被构造的复杂对象。ConcreteBuilder创建该产品的内部表示并定义它的装配过程，包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

工厂模式和构建者模式的区别

构建者模式和工厂模式很类似，区别在于构建者模式是一种个性化产品的创建。而工厂模式是一种标准化的产品创建。

图示



- 导演类：按照一定的顺序或者一定的需求去组装一个产品。
- 构造者类：提供对产品的不同个性化定制，最终创建出产品。
- 产品类：最终的产品

示例

- 构建者

```
// 构建器
public class StudentBuilder {

  // 需要构建的对象
  private Student student = new Student();
```

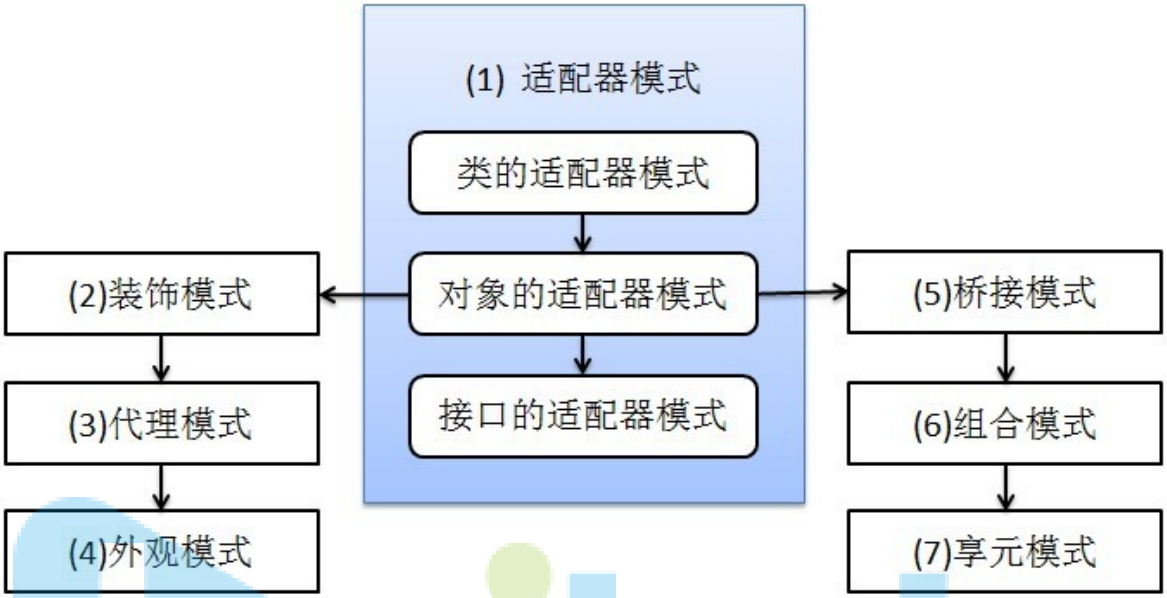
```
  public StudentBuilder id(int id) {
    student.setId(id);
    return this;
  }

  public StudentBuilder name(String name) {
    student.setName(name);
    return this;
  }

  public StudentBuilder age(int age) {
    student.setAge(age);
    return this;
  }

  public StudentBuilder father(String fatherName) {
    Father father = new Father();
    father.setName(fatherName);
    student.setFather(father);
    return this;
  }

  // 构建对象
  public Student build() {
    return student;
  }
}
```

- 导演类

```
// 导演类/测试类
public class BuildDemo {

  public static void main(String[] args) {

    StudentBuilder builder = new StudentBuilder();
    // 决定如何创建一个Student
    Student student =
builder.age(1).name("zhangsan").father("zhaosi").build();
    System.out.println(student);

  }
}
```

结构型设计模式

我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：**适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式**。其中对象的适配器模式是各种模式的起源，我们看下面的图：

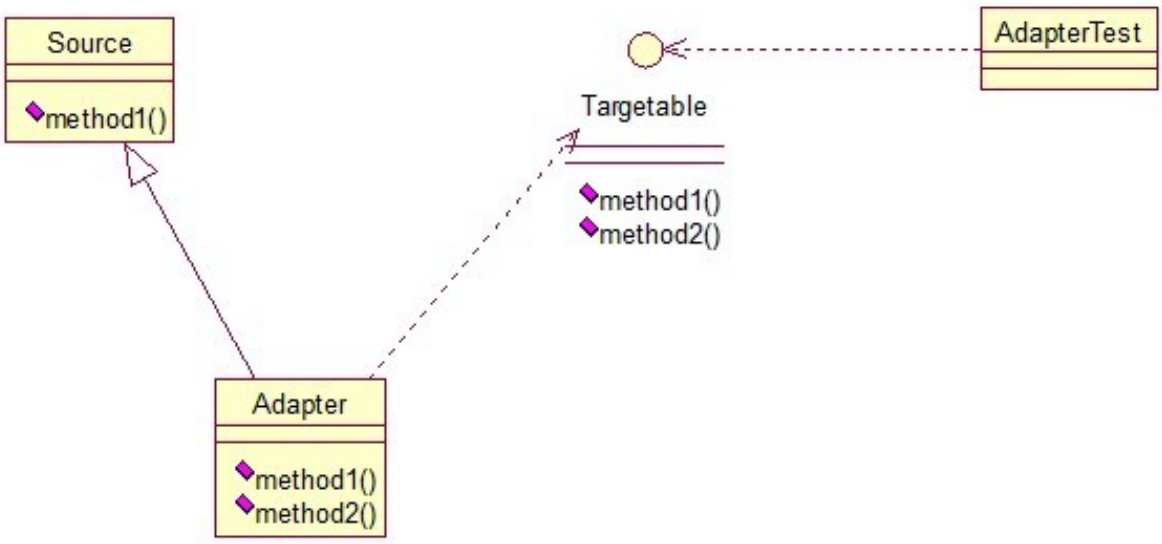


6、适配器模式（Adapter）

适配器模式

适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：**类的适配器模式、对象的适配器模式、接口的适配器模式**。

类的适配器模式



核心思想就是：有一个`Source`类，拥有一个方法，待适配，目标接口是`Targetable`，通过`Adapter`类，将`Source`的功能扩展到`Targetable`里，看代码：

```
public class Source {
     public void method1() {
         System.out.println("this is original method!");
     }
 }
```

```
 public interface Targetable {
     /* 与原类中的方法相同 */
     public void method1();
     /* 新类的方法 */
     public void method2();
 }
```

```
 public class Adapter extends Source implements Targetable {
     @Override
     public void method2() {
         System.out.println("this is the targetable method!");
     }
 }
```

`Adapter`类继承`Source`类，实现`Targetable`接口，下面是测试类：

```
 public class AdapterTest {
     public static void main(String[] args) {
         Targetable target = new Adapter();
         target.method1();
         target.method2();
     }
 }
```

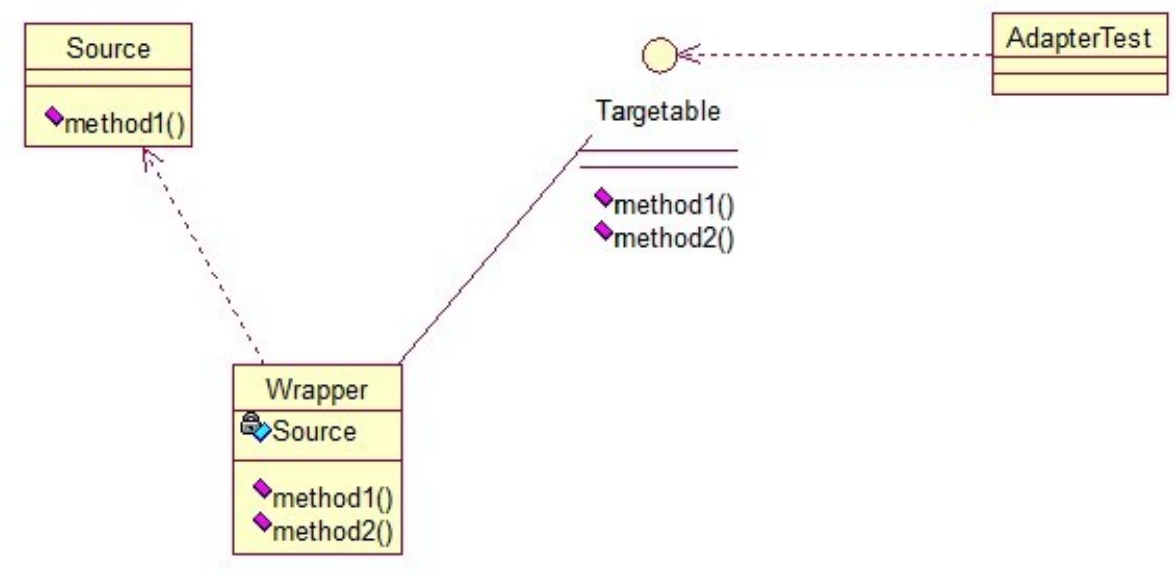
输出：

```
this is original method!
this is the targetable method!
```

这样`Targetable`接口的实现类就具有了`Source`类的功能。

对象的适配器模式

基本思路和类的适配器模式相同，只是将`Adapter`类作修改，这次不继承`Source`类，而是持有`Source`类的实例，以达到解决兼容性的问题。看图：



只需要修改`Adapter`类的源码即可：

```
public class Wrapper implements Targetable {
    private Source source;
    public Wrapper(Source source){
        super();
        this.source = source;
    }
    @Override
     public void method2() {
         System.out.println("this is the targetable method!");
     }
     @Override
     public void method1() {
         source.method1();
     }
 }
```

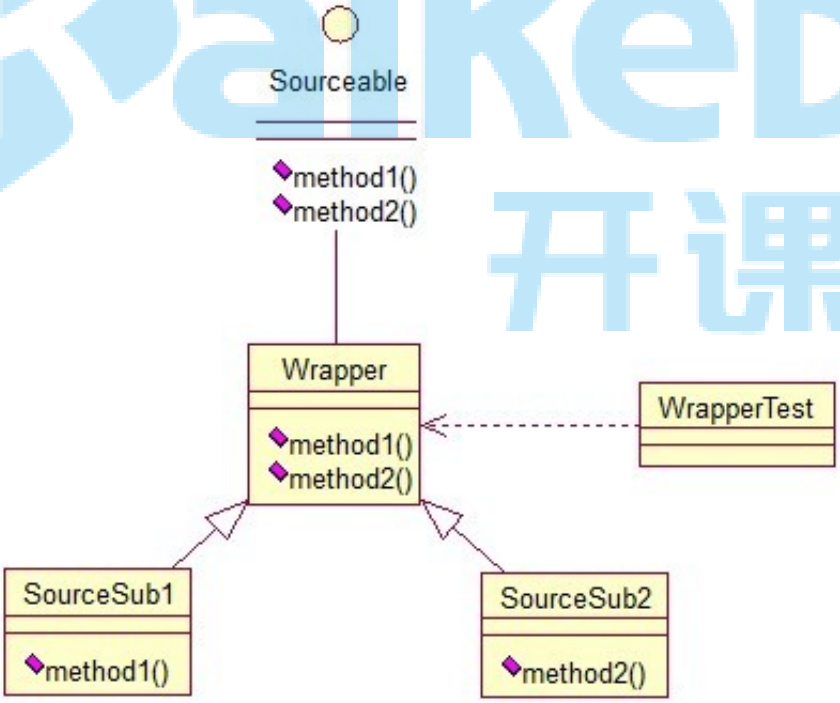
测试类：

```
public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

输出与第一种一样，只是适配的方法不同而已。

接口的适配器模式

第三种适配器模式是**接口的适配器模式**，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。

看代码：

```
public interface Sourceable {
    public void method1();
    public void method2();
}
```

抽象类Wrapper2：

```
public abstract class Wrapper implements Sourceable{

    public void method1(){}
    public void method2(){}
}
```

```
public class SourceSub1 extends Wrapper {
    public void method1(){
        System.out.println("the sourceable interface's first Sub1!");
    }
}
```

```
public class SourceSub2 extends Wrapper {
    public void method2(){
        System.out.println("the sourceable interface's second Sub2!");
    }
}
```

测试类：

```
public class WrapperTest {

    public static void main(String[] args) {
        Sourceable source1 = new SourceSub1();
        Sourceable source2 = new SourceSub2();

        source1.method1();
        source1.method2();
        source2.method1();
        source2.method2();
    }
}
```

测试输出：

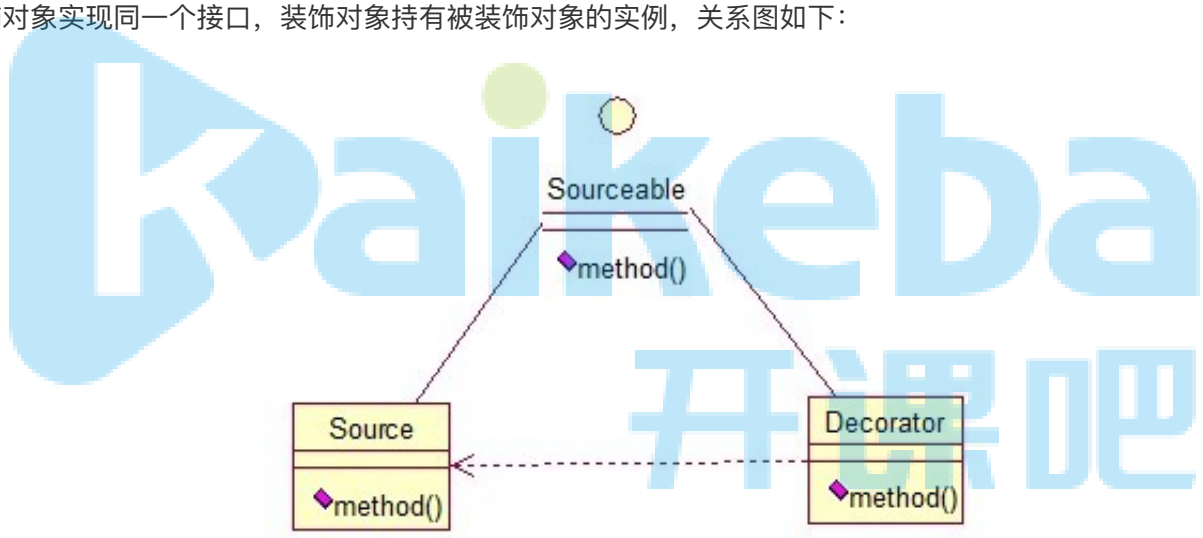
```
the sourceable interface's first Sub1!
the sourceable interface's second Sub2!
```

达到了我们的效果！

> 讲了这么多，总结一下三种适配器模式的应用场景：
>
> - **类的适配器模式**：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。
> - **对象的适配器模式**：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。
> - **接口的适配器模式**：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

装饰模式

顾名思义，装饰模式就是给一个对象增加（**装饰**）一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

```
public interface Sourceable {
    public void method();
}
```

```
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```
public class Decorator implements Sourceable {
    private Sourceable source;
    public Decorator(Sourceable source){
        super();
        this.source = source;
    }
    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}
```

测试类：

```
public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}
```

输出：

```
before decorator!
the original method!
after decorator!
```

装饰器模式的应用场景：

1. 需要扩展一个类的功能。
2. 动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：

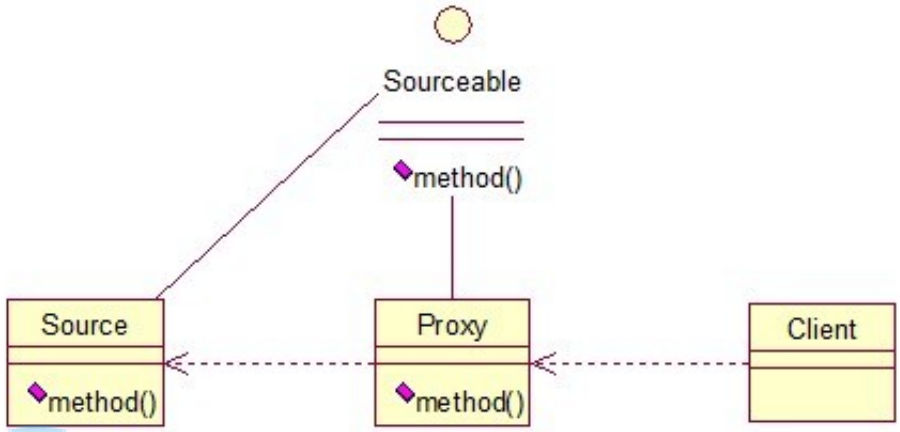
产生过多相似的对象，不易排错！

代理模式（面试）

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作。代理又分为动态代理和静态代理

静态代理

比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：



根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

```
public interface Sourceable {
    public void method();
}
```

```
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```
public class Proxy implements Sourceable {
    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
```

```
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类：

```
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
```

输出：

```
before proxy!
the original method!
after proxy!
```

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1. 修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。
2. 就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

动态代理

JDK动态代理

基于接口去实现的动态代理

```
public class JDKProxyFactory implements InvocationHandler {

  // 目标对象的引用
  private Object target;
```

```
  // 通过构造方法将目标对象注入到代理对象中
  public JDKProxyFactory(Object target) {
    super();
    this.target = target;
  }

  /**
   * @return
   */
  public Object getProxy() {

    // 如何生成一个代理类呢?
    // 1、编写源文件
    // 2、编译源文件为class文件
    // 3、将class文件加载到JVM中(ClassLoader)
    // 4、将class文件对应的对象进行实例化（反射）

    // Proxy是JDK中的API类
    // 第一个参数：目标对象的类加载器
    // 第二个参数：目标对象的接口
    // 第二个参数：代理对象的执行处理器
    Object object = Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),
        this);

    return object;
  }

  /**
   * 代理对象会执行的方法
   */
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    Method method2 = target.getClass().getMethod("saveUser", null);
    Method method3 =
Class.forName("com.sun.proxy.$Proxy4").getMethod("saveUser", null);
    System.out.println("目标对象的方法:" + method2.toString());
    System.out.println("目标接口的方法:" + method.toString());
    System.out.println("代理对象的方法:" + method3.toString());
    System.out.println("这是jdk的代理方法");
    // 下面的代码，是反射中的API用法
    // 该行代码，实际调用的是[目标对象]的方法
    // 利用反射，调用[目标对象]的方法
    Object returnValue = method.invoke(target, args);

    return returnValue;
  }
```

```
}
```

CGLib动态代理

是通过子类继承父类的方式去实现的动态代理，不需要接口。

```
public class CgLibProxyFactory implements MethodInterceptor {

  /**
   * @param clazz
   * @return
   */
  public Object getProxyByCgLib(Class clazz) {
    // 创建增强器
    Enhancer enhancer = new Enhancer();
    // 设置需要增强的类的类对象
    enhancer.setSuperclass(clazz);
    // 设置回调函数
    enhancer.setCallback(this);
    // 获取增强之后的代理对象
    return enhancer.create();
  }

  /***
   * Object proxy:这是代理对象，也就是[目标对象]的子类
   * Method method:[目标对象]的方法
   * Object[] arg:参数
   * MethodProxy methodProxy: 代理对象的方法
   */
  @Override
  public Object intercept(Object proxy, Method method, Object[] arg,
MethodProxy methodProxy) throws Throwable {
    // 因为代理对象是目标对象的子类
    // 该行代码，实际调用的是父类目标对象的方法
    System.out.println("这是cglib的代理方法");

    // 通过调用子类[代理类]的invokeSuper方法，去实际调用[目标对象]的方法
    Object returnValue = methodProxy.invokeSuper(proxy, arg);
    // 代理对象调用代理对象的invokeSuper方法，而invokeSuper方法会去调用目标类的invoke方
法完成目标对象的调用

    return returnValue;
  }
}
```

行为型设计模式

模板方法模式

父类+子类

父类去抽取共性的方法操作：一般父类去制定方法的操作步骤，比如说把大象装冰箱分几步

子类去实现复杂的特性的功能

策略模式

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

在策略模式中，我们创建表示各种策略的对象和一个行为随着策略对象改变而改变的 context 对象。策略对象改变 context 对象的执行算法。

介绍

意图：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

主要解决：在有多种算法相似的情况下，使用 **if...else** 所带来的复杂和难以维护。

何时使用：一个系统有许多许多类，而区分它们的只是他们直接的行为。

如何解决：将这些算法封装成一个一个的类，任意地替换。

关键代码：实现同一个接口。

应用实例： 1、诸葛亮的锦囊妙计，每一个锦囊就是一个策略。 2、旅行的出游方式，选择骑自行车、坐汽车，每一种旅行方式都是一个策略。 3、JAVA AWT 中的 LayoutManager。

优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

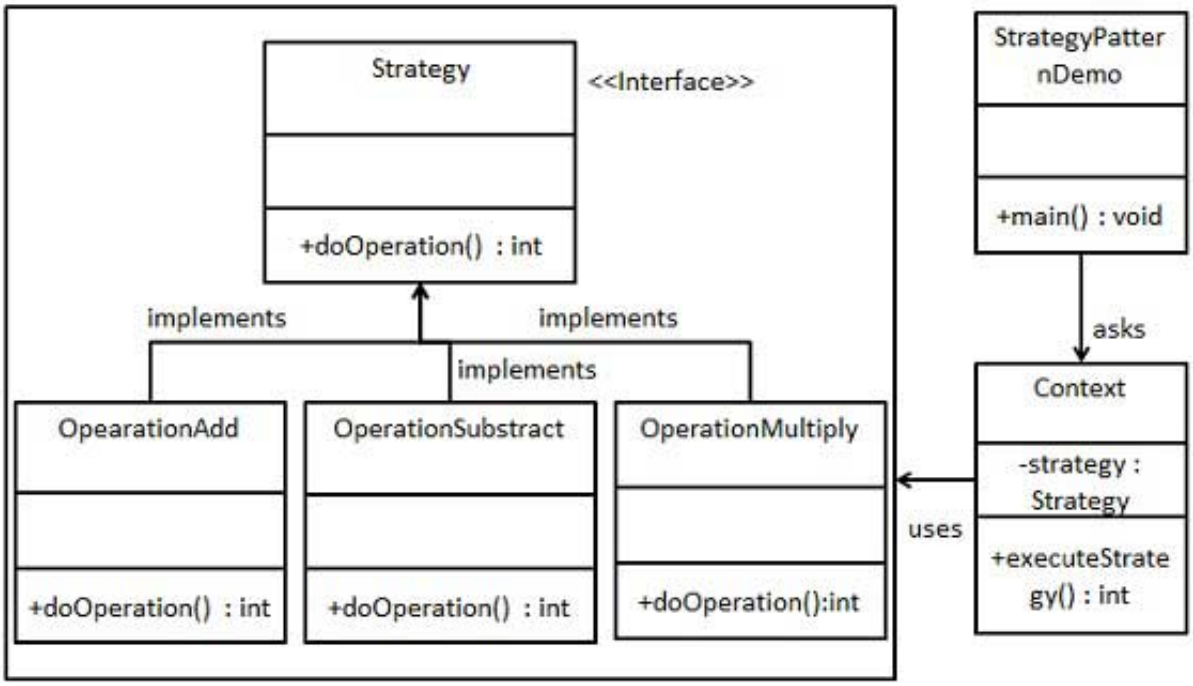
使用场景： 1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。 2、一个系统需要动态地在几种算法中选择一种。 3、如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。

注意事项：如果一个系统的策略多于四个，就需要考虑使用混合模式，解决策略类膨胀的问题。

实现

我们将创建一个定义活动的 *Strategy* 接口和实现了 *Strategy* 接口的实体策略类。*Context* 是一个使用了某种策略的类。

StrategyPatternDemo，我们的演示类使用 *Context* 和策略对象来演示 Context 在它所配置或使用的策略改变时的行为变化。



步骤1

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}
```

步骤2

```
public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}
```

```
public class OperationSubstract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}
```

```
public class OperationMultiply implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

步骤3

```
public class Context {
    private Strategy strategy;
    public Context(Strategy strategy){
        this.strategy = strategy;
    }
    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

步骤 4

使用 *Context* 来查看当它改变策略 *Strategy* 时的行为变化。

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationSubstract());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

步骤 5

执行程序，输出结果：

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

其他设计模式

MVC设计模式

委托设计模式