浙江大学

题目	软件工程设计模式研究报告
小组成员	武靖超 郭炅 李国昌 金雨若 金俊婕
学院	计算机科学与技术学院
时间	2020 年 5 月 10 日

目录

1 最新文献回顾

软件工程中的设计模式研究最早可追溯到 GoF 所著的《设计模式：可重用的面向对象软件的元素》，将设计模式定义为针对常见和可重复的软件设计问题的解决方案，软件开发人员可以使用相同或经过微调的方式解决频繁发生的问题。近年来研究者们在设计模式领域做了大量的工作。

有的研究者关注设计模式的有效性。针对设计模式编码大量信息的特征，有研究者提出使用特征图算法对源代码的设计实例进行检测 [1]，可以有效揭示隐藏在源代码中的有用信息。有研究者基于图论提出了一种两阶段检测设计模式的方法 [2]，第一阶段将模式和语义转换为语义图，第二阶段应用语义匹配算法获得候选实例，最终得到模式的行为特征。

有的研究者关注设计模式的自动识别。有研究者在 UML 和文本分类方法的基础上改进了自动选择设计模式的技术，通过提出新的框架和基于无监督学习的评估模型改进自动化技术 [3]。有研究者开发了完全基于机器学习技术的识别方法 [4]，针对现有的识别工具构建数据集，通过自动检测选择设计模式，提高程序的可维护性和可靠性。

有研究者关注模式实例的查找。针对从源代码中发现设计模式实例的现有方法的低效，有研究者提出了一种基于有序序列快速检索候选模式实例的方法 [5]，大大降低搜索空间，提升响应速度。有研究者将基于可视语言解析和模型检查的静态分析与基于源代码工具的动态分析相结合 [6]，提出了一种用于在源代码中检测模式实例的工具，可以获得更好的正确性与完整性。

有研究者关注设计模式的教学。针对学生，为软件工程设计模式构建了一个交互式学习环境 [7]，用于帮助学生改进解决方案并加深对设计模式的理解。

2 系统体系分析

本项目围绕着疫情服务展开，是一个疫情监管与服务平台，民众可以在该平台上获得准确的疫情相关信息以及在疫情期间所需要的大部分服务。

本项目也是围绕着数据的增删改查的数据中心模型，项目的主要操作是有关数据的操作。项目中国际舆情中的新闻列表，邻里拼团中的订单、物品，防疫查询中的门诊点、药店等都可以用面向对象的思想进行建模。由于我们使用了前后端分离的架构进行项目编写，这在不知不觉中其实已经用到许多设计模式。

3 GoF 设计模式应用–创建型模式

设计模式所提倡的一个原则是“相比于继承，我们更优先使用组合”。正是在这一原则的指导下，我们可以采用一些典型的设计模式去替代我们原有的一些设计理念。许多设计模式都遵循着这一法则，最近的许多研究也是基于此法则逐渐发展延申。

第三——五章我们主要沿着 GoF 所著的设计模式书籍《设计模式：可复用面向对象软件的基础》来展开我们的设计模式应用研究。

3.1 工厂模式

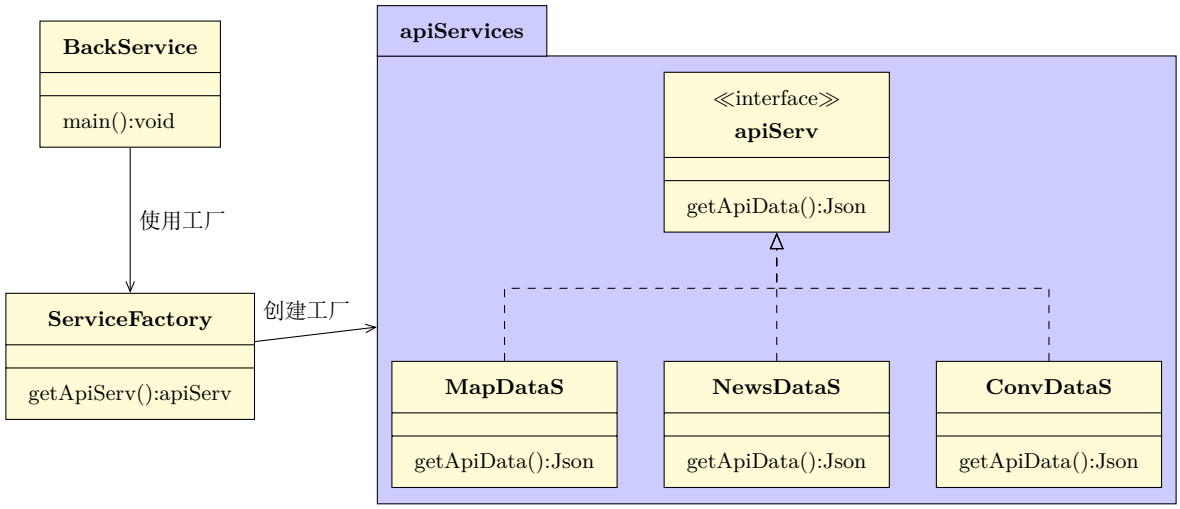
工厂模式（Factory Pattern）是最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

3.1.1 介绍

- 意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。
- 主要解决：主要解决接口选择的问题。
- 如何使用：我们明确地计划不同条件下创建不同实例时。
- 如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。
- 关键代码：创建过程在其子类执行。

3.1.2 举例



在我们的工程中，虽然使用了 spring boot，服务端的整体架构相对固定，但是仍然可以对一些数据服务进行抽象化处理，比如，可以创建一个获取数据的 *Service* 接口和实现 *Service* 接口的实体类，工厂类 *ServiceFactory*，预期可以使用 *ServiceFactory* 获取 *Service* 对象，该对象向 *ServiceFactory* 传递信息，以便获取其所需对象的类型。

通过这种方式，在 *Controller* 层，可以调用统一的 *getData* 接口获取数据，方便对为数众多的获取数据的函数调用进行管理，唯一增加的成本是需要声明获取对应的服务类即可。

3.1.3 优缺点分析

使用工厂模式对工程代码进行抽象化处理，其优点是十分明确的：

1 一个调用者想创建一个对象，只要知道其名称就可以了。

2 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以

3 屏蔽产品的具体实现，调用者只关心产品的接口。

当然也存在一定的缺点：每次增加一个获取数据的服务时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

3.2 抽象工厂模式

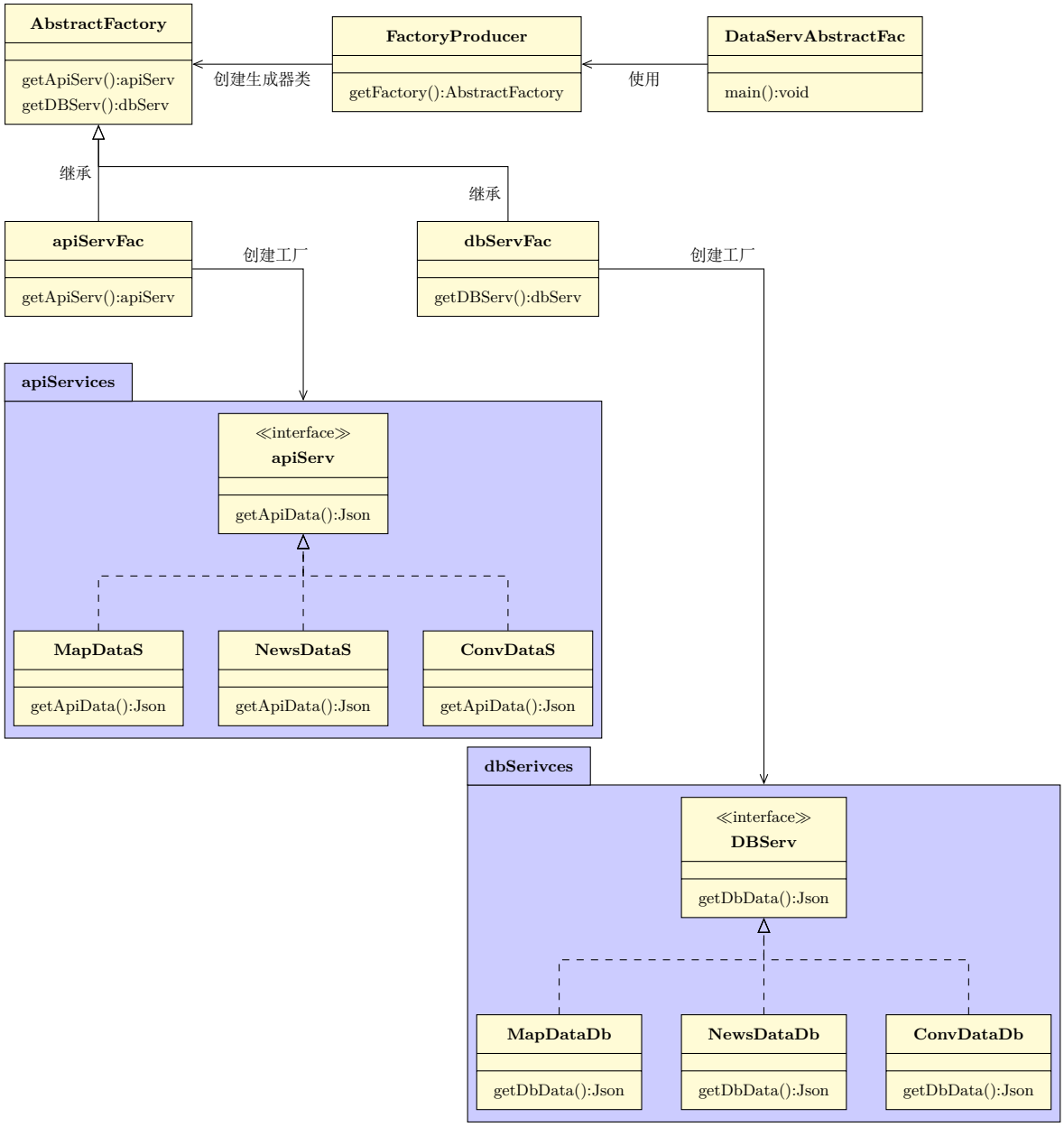
抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

3.2.1 介绍

- 意图：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。
- 主要解决：主要解决接口选择的问题。
- 如何使用：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品。
- 如何解决：在一个产品族里面，定义多个产品。
- 关键代码：在一个工厂里聚合多个同类产品。

3.2.2 举例



由于作业在服务器端使用持久化存储技术，实际上对客户端的数据请求存在两种响应方式，其一为从服务器端存储的数据库内返回数据，另一种是从远程外部 API 处请求数据，因此，对于两种不同的数据来源，由于请求数据的执行逻辑相差极大，实际是两种执行模式，需要构建两种请求函数，接口也不相同，从工厂模式的角度来说，需要构建两种不同的工厂产生两种不同的类。

使用抽象工厂模式，可以定义一个抽象工厂类，基于一个工厂生成器产生两种不同的工厂，可以极大的统一代码结构，针对不同的数据类型，只需要增加底部子类即可。

3.2.3 优缺点分析

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

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

3.3 单例模式

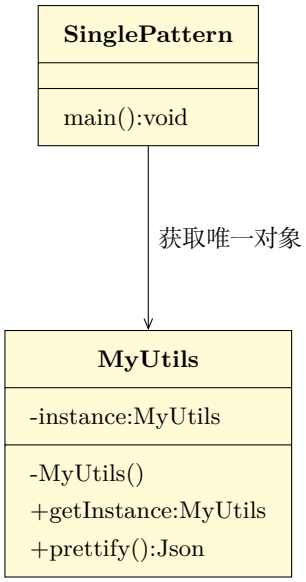
单例模式（Singleton Pattern）是最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

3.3.1 介绍

- 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 主要解决：一个全局使用的类频繁地创建与销毁。
- 如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。
- 关键代码：构造函数是私有的。

3.3.2 举例



在服务器端，为了对从外部 API 或者内部数据库获取的数据进行 json 美化，维护了一个 $MyUtils$ 类，该类内部存在一个方法 $prettify()$，该方法可以对所有的 json 数据进行格式化，是服务器向客户端返回数据前的最后一步。

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

3.3.3 优缺点分析

优点：

a. 在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例（比如管理学院首页页面缓存）。

b. 避免对资源的多重占用（比如写文件操作）。

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

3.4 建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

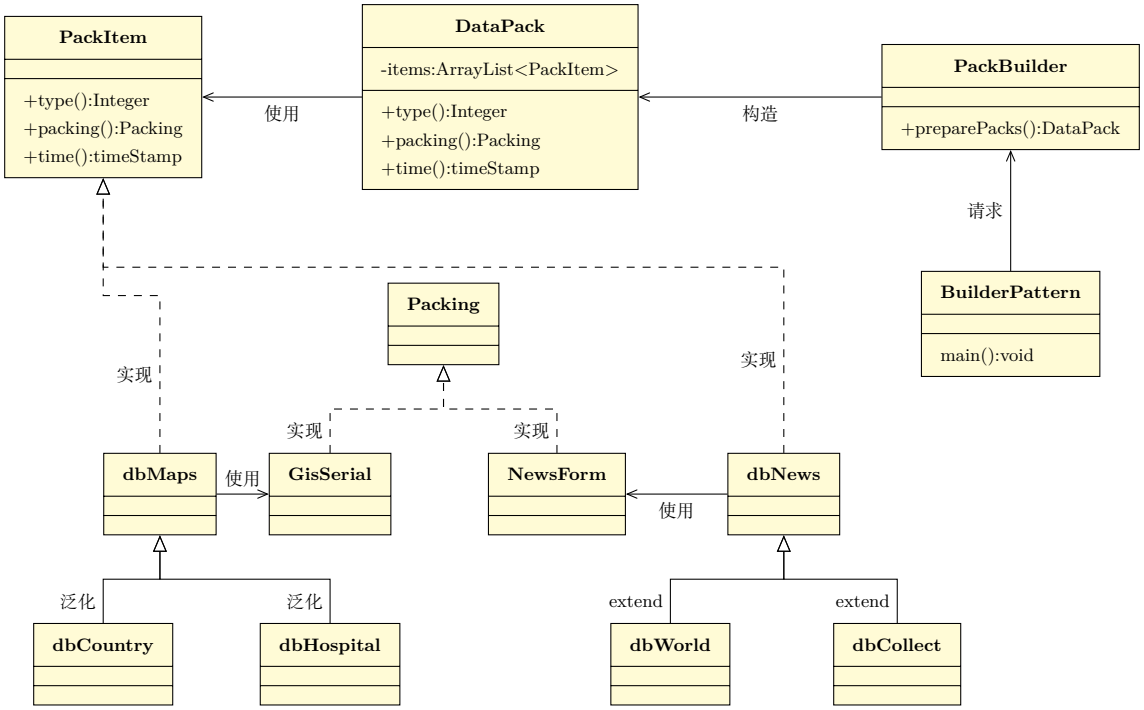
3.4.1 介绍

- 意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。
- 主要解决：主要解决在软件系统中，有时候面临着”一个复杂对象”的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。
- 何时使用：一些基本部件不会变，而其组合经常变化的时候。
- 如何解决：将变与不变分离开。
- 关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

3.4.2 举例

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

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



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

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

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

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

3.4.3 优缺点分析

优点：建造者独立，易扩展；便于控制细节风险。

缺点：需要生成的对象具有复杂的内部结构；需要生成的对象内部属性本身相互依赖

3.5 原型模式

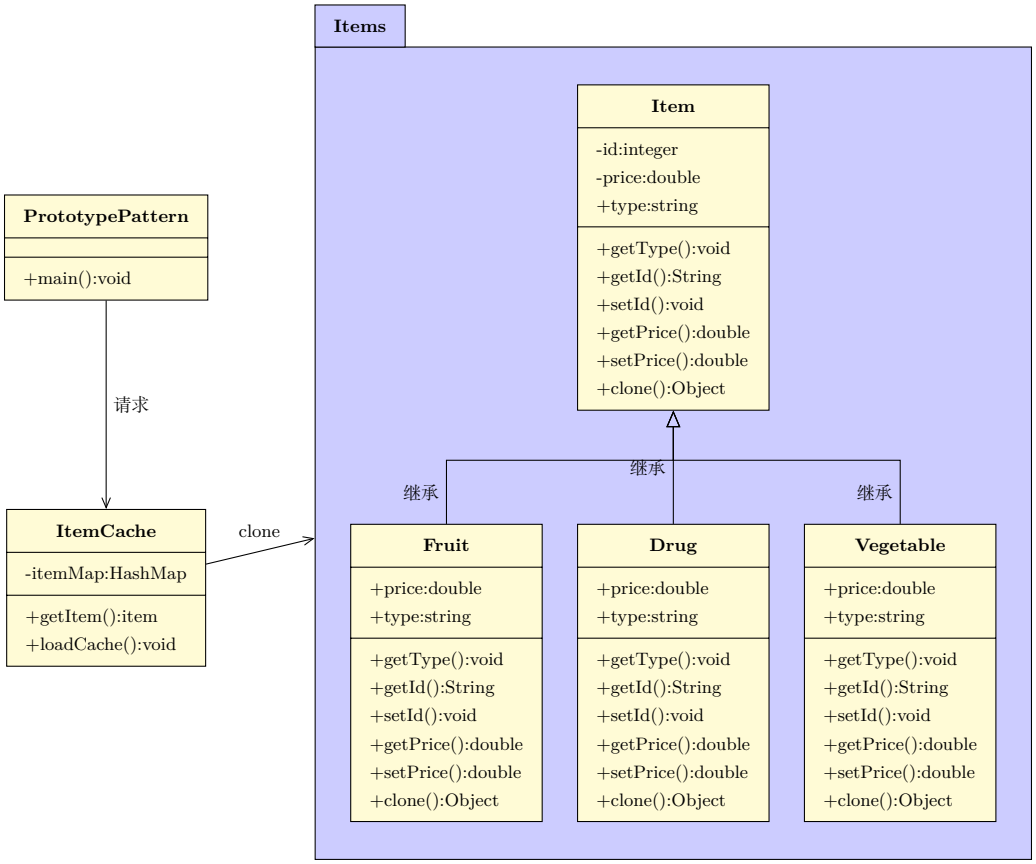
原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

3.5.1 介绍

- 意图：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
- 主要解决：在运行期建立和删除原型。
- 何时使用：1、当一个系统应该独立于它的产品创建，构成和表示时。2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。3、为了避免创建一个与产品类层次平行的工厂类层次时。4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。
- 如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。
- 关键代码：1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()。2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些” 易变类” 拥有稳定的接口。

3.5.2 举例



举例，工程涉及在线订单商品业务，因此需要维护众多的商品信息，但是这些商品的基本信息字段是一致的，每一个实例只能是一种商品，同时具体是哪种商品是经由客户端动态指定的，手动维护所有的可能性并实例化相对冗余，使用原型模式可以大幅度降低操作难度

有三种不同的商品，显然每个商品实例只能是一种商品，但是每个实例都需要维护 $id, type$ 和 $price$ 字段，因此必须考虑使用继承基类的方式进行处理，使用原型模式可以规避掉每次冗余的实例初始化过程，而只需要进行 clone 即可。

客户端的订单是动态变化的，不能保证每次的商品种类是相同的但是维护所有的可能性或者每次均创建一个新的实例，相对麻烦。使用 clone，处理成本会降低许多。

3.5.3 优缺点分析

优点：

a. 性能提高

b. 逃避构造函数的约束

缺点：

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

b. 必须实现 Cloneable 接口。

4 GoF 设计模式应用-结构性模型

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

结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对象组合方式具有更大的灵活性，而这种机制用静态类组合是不可能实现的。[8]

4.1 适配器（Adaptor）

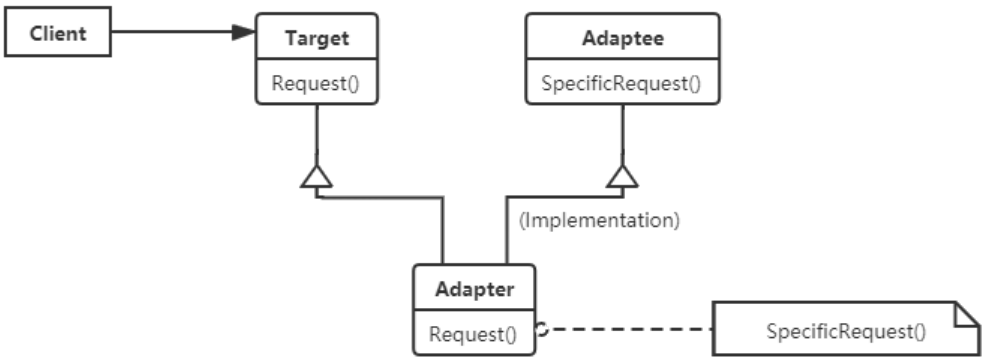
4.1.1 介绍

适配器模式旨在将一个类的接口转换成客户希望的另一个接口，使得一些原本由于接口不兼容而无法一起工作的类能够在一起工作。

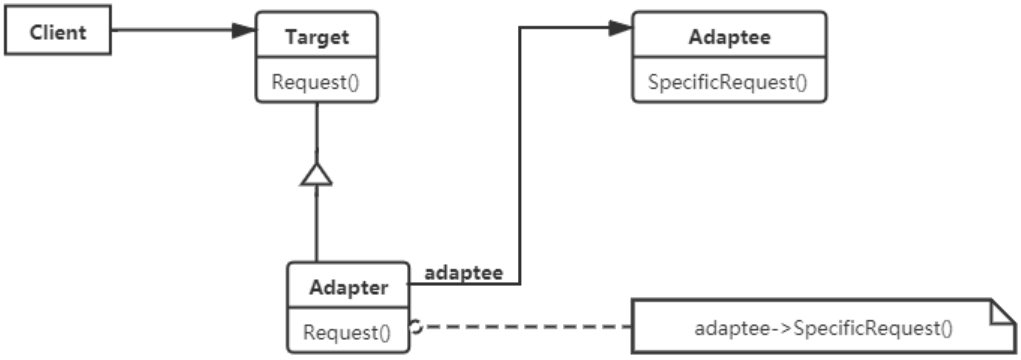
以下情况可使用 Adapter 模式

- 想使用一个已经存在的类，而它的接口不符合需求。
- 像创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即接口可能不兼容的类）协同合作。
- （仅适用于对象 Adapter）想使用一些已存在的子类，但不可能对每一个都进行子类化一匹配它们的接口。对象适配器可以适配它的父类接口。

类适配器使用多重继承对一个接口与另一个接口进行匹配。



对象适配器依赖于对象组合。



在上图中，我们可以看到一共有 4 类参与者：

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

4.1.2 举例

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

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

4.1.3 优缺点分析

类适配器和对象适配器有不同的权衡。

类适配器

优点：使得 Adapter 可以重定义 Adaptee 的部分行为；仅仅引入了一个对象，并不需要额外的指针以间接得到 Adaptee。

缺点：想要匹配一个类以及所有它的子类时，类 Adapter 将不能胜任工作。

对象适配器

优点：允许一个 Adapter 与多个 Adaptee—即 Adaptee 本身以及它的所有子类同时工作。

缺点：重定义 Adaptee 的行为比较困难。

4.2 桥接（Bridge）

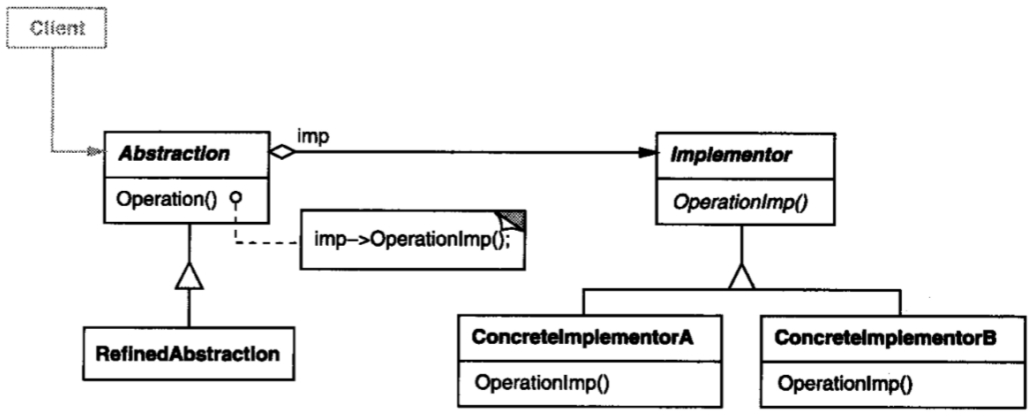
4.2.1 介绍

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

以下一些情况适用 Bridge 模式:

- 不希望在抽象和它的实现部分之间有一个固定的绑定关系。
- 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。
- 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
- 在 C++ 中想对客户完全隐藏抽象的实现部分。
- 在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

桥接模式的结构图如下：



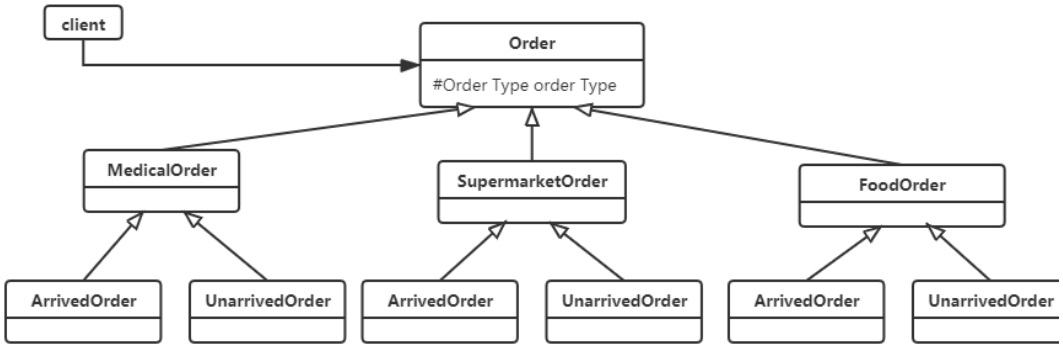
在上图中，我们可以看到一共有 4 类参与者：

- Abstraction ——定义抽象类的接口；维护一个指向 Implementor 类型对象的指针。
- 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。一般来讲，Implementor 接口仅提供基本操作，而 Abstraction 则定义了基于这些基本操作的较高层次的操作。
- ConcreteImplementor ——实现 Implementor 接口并定义它的具体实现。

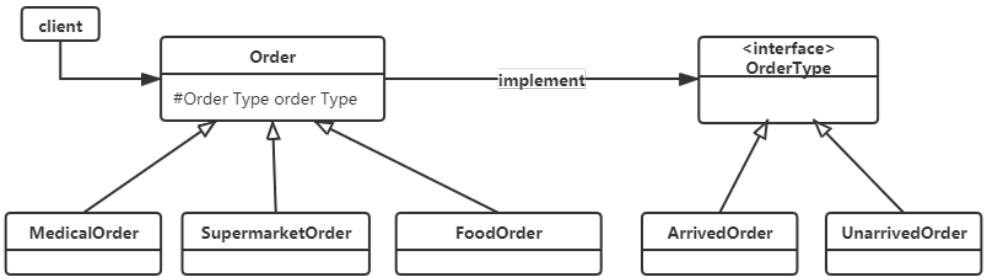
4.2.2 举例

在我们的系统中 Order 类既可以按送达情况分为未送达和已送达，也可以根据订单类型分为医用物品、超市物品和水果蔬菜。这种特性意味着我们可以用桥接模式去实现。

下图所示是未使用桥接模式的一种可能的实现方法，可以看到，由于送达情况和订单类型两个维度相互交错，所以继承体系十分复杂。



下图所示为重构为桥接模式后的类图。我们把豪华/普通这一属性分离出来，单独使用一个接口去表示它。从图中可以发现，原先的耦合度被降低，继承体系的复杂度也得到降低。



4.2.3 优缺点分析

优点:

- 分离接口及其实现部分
- 提高可扩充性
- 实现细节对客户透明

缺点:

- 桥接模式的使用会增加系统的理解与设计难度，由于关联关系建立在抽象层，要求开发者一开始就针对抽象层进行设计与编程。
- 桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性，如何正确识别两个独立维度也需要一定的经验积累。

4.3 组合（Composite）

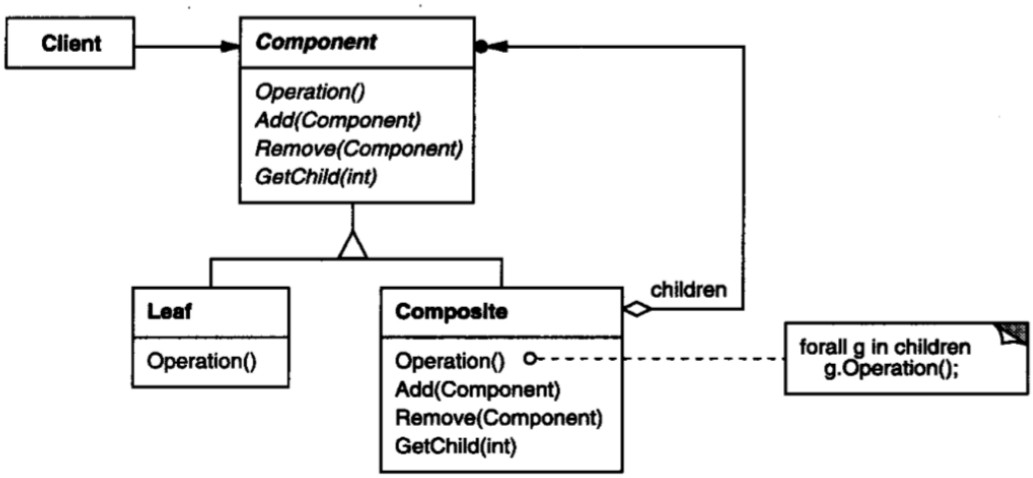
4.3.1 介绍

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

以下一些情况适用 Composite 模式:

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

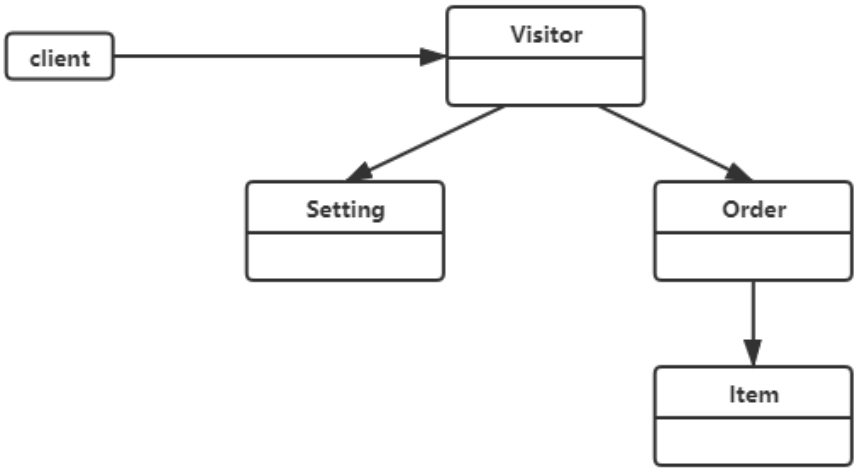
组合模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

- Component ——为组合中的对象声明接口; 在适当的情况下，实现所有类共有接口的缺省行为; 声明一个接口用于访问和管理 Component 的子组件。
- Leaf ——在组合中表示叶节点对象，叶节点没有子节点; 在组合中定义图元对象的行为。
- Composite ——定义有子部件的那些部件的行为; 存储子部件; 在 Component 接口中实现与子部件有关的操作。
- Client ——通过 Component 接口操纵组合部件的对象。

4.3.2 举例



在本系统中，Setting 类和 Order 类是 Visitor 类的一部分，而 Item 类又是 Order 类的一部分。其中 Visitor 类是指使用小程序且可以发起或参与拼团的用户；Setting 类用于保存用户的个人信息；Order 类用于记录用户拼团生成的订单；Item 类表示一件商品在数据库中的信息。

用户使用 Component 类接口与组合结构中的对象进行交互。如果接收者是一个叶节点，则直接处理请求。如果接收者是 Composite，它通常将请求发送给它的子部件，在转发请求之前与/或之后可能执行一些辅助操作。

4.3.3 优缺点分析

优点:

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

缺点:

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

4.4 外观（Facade）

4.4.1 介绍

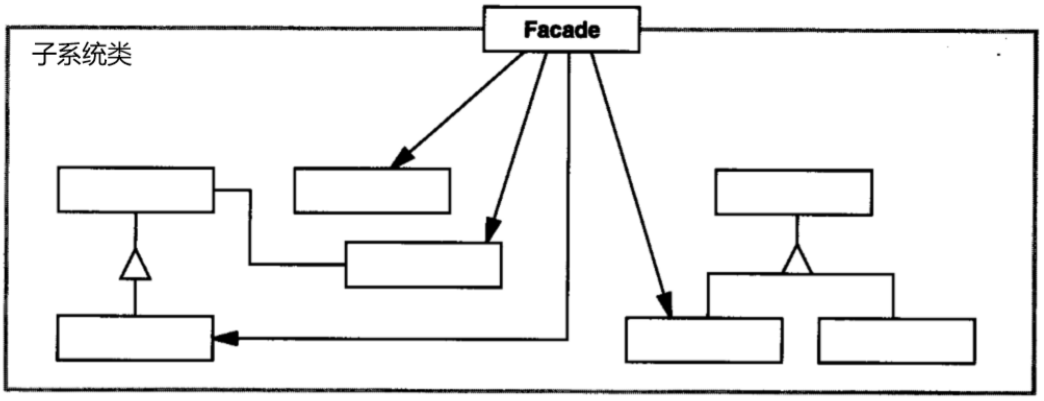
外观意在提供为为子系统中的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

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

以下一些情况适用 Facade 模式:

- 要为一个复杂子系统提供一个简单接口时。
- 客户程序与抽象类的实现部分之间存在着很大的依赖。
- 构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。。

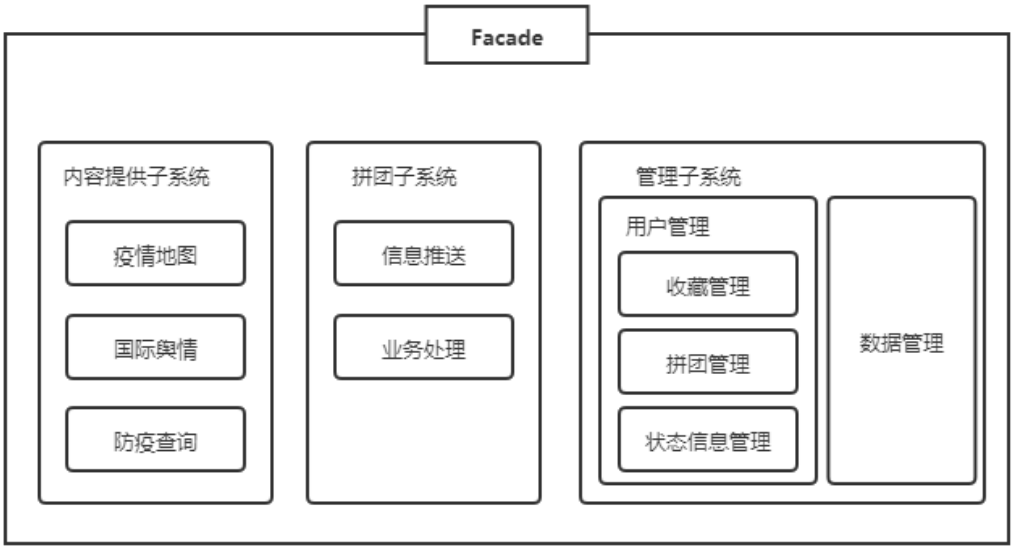
外观模式的结构图如下:



在上图中，我们可以看到一共有 2 类参与者:

- Facade ——知道哪些子系统类负责处理请求；将客户的请求代理给适当的子系统对象
- Subsystem classes ——实现子系统的功能；处理由 Facade 对象指派的任务；没有 facade 的任何相关信息；即没有指向 facade 的指针。

4.4.2 举例



在本系统中，一共可分为 3 个子系统，包括内容提供系统、拼团系统、管理系统。

我们也为这三个子系统设计了一个统一的接口供客户使用，使得子系统间的通信和相互依赖关系达到最小。

客户程序通过发送请求给 Facade 的方式与子系统通讯，Facade 将这些消息转发给适当的子系统对象。尽管是子系统中的有关对象在做实际工作，但 Facade 模式本身也必须将它的接口转换成子系统的接口。

4.4.3 优缺点分析

优点:

- 对客户屏蔽子系统组件，因而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。
- 实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。
- 如果应用需要，它并不限制它们使用子系统类。因此你可以在系统易用性和通用性之间加以选择。

缺点:

- 不符合开闭原则，如果要修改某一个子系统的功能，通常外观类也要一起修改。
- 没有办法直接阻止外部不通过外观类访问子系统的功能，因为子系统类中的功能必须是公开的。

5 GoF 设计模式应用-行为模式

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

行为型模式分为类行为模式和对象行为模式，前者采用继承机制来在类间分派行为，后者采用组合或聚合在对象间分配行为。由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象行为模式比类行为模式具有更大的灵活性。
结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对象组合方式具有更大的灵活性，而这种机制用静态类组合是不可能实现的。

5.1 模板方法（Template Method）

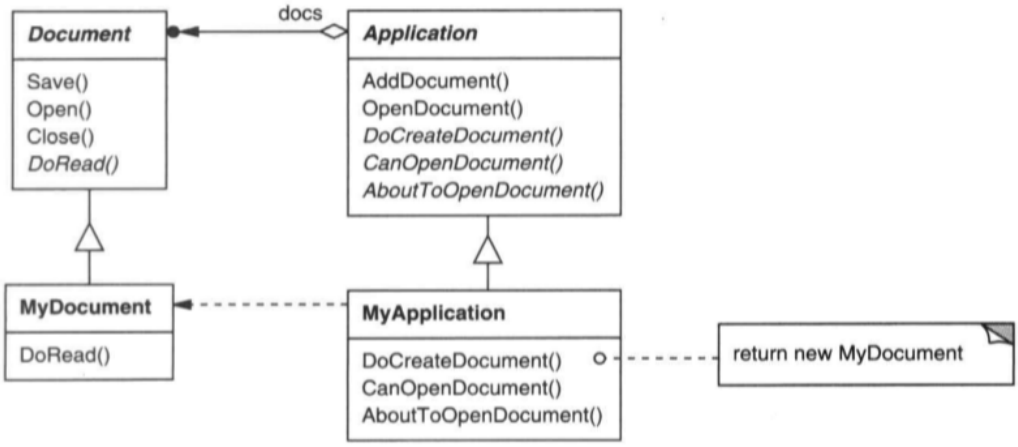
5.1.1 介绍

模板方法（Template Method）模式的定义如下：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

以下情况可使用模板方法模式

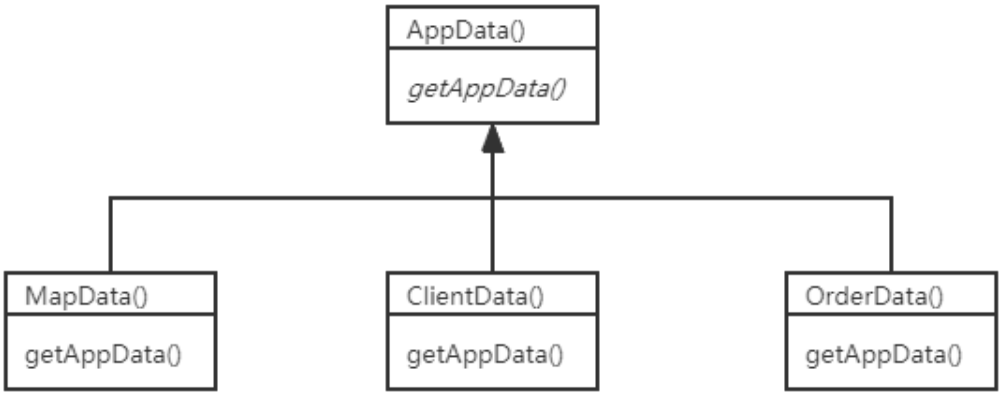
- 算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。
- 当多个子类存在公共的行为时，可以将其提取出来并集中到一个公共父类中以避免代码重复。首先，要识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

在模板方法模式中，基本方法包含：抽象方法、具体方法和钩子方法，正确使用“钩子方法”可以使得子类控制父类的行为。如下面例子中，可以通过在具体子类中重写钩子方法 HookMethod1() 和 HookMethod2() 来改变抽象父类中的运行结果。



5.1.2 举例

在我们的系统中基本上很多工作都可以采用模板方法来实现，例如，我们可以构建数据获取的普遍模板，在他的子类实例中去实现方法。



5.1.3 优缺点分析

优点:

- 它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
- 它在父类中提取了公共的部分代码，便于代码复用。
- 部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

缺点:

- 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象。

- 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。

5.2 策略（Strategy）

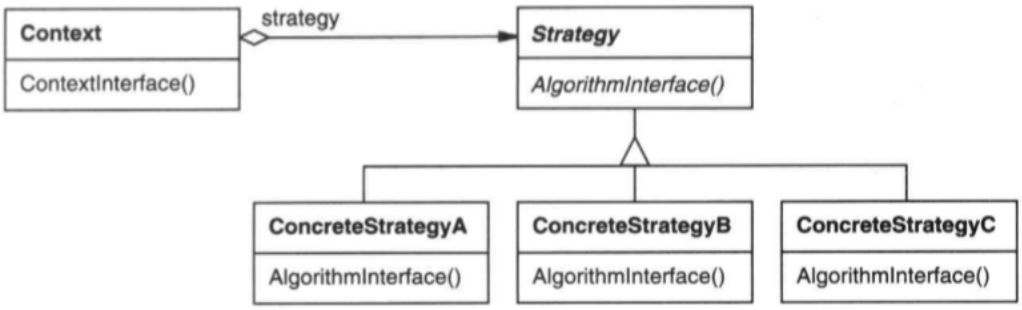
5.2.1 介绍

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

以下一些情况适用 Strategy 模式:

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

策略模式的结构图如下：

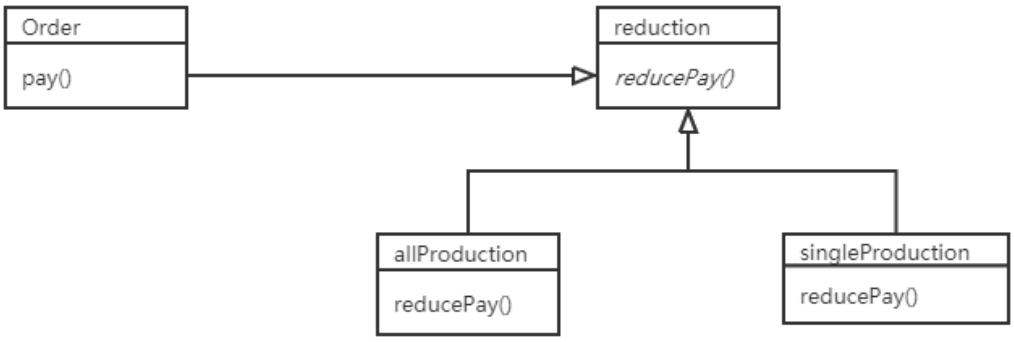


在上图中，我们可以看到一共有 3 类参与者：

- 抽象策略（Strategy）类：定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。
- 具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，提供具体的算法实现。
- 环境（Context）类：持有一个策略类的引用，最终给客户端调用。

5.2.2 举例

在我们的下单系统中，主要的优惠策略有满减和单品优惠两种。所以，我们可以使用策略模式来封装这两种优惠计算方式。



5.2.3 优缺点分析

优点:

- 策略模式提供了对 “开闭原则” 的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。
- 使用策略模式可以避免多重条件选择语句。多重条件选择语句不易维护，它把采取哪一种算法或行为的逻辑与算法或行为本身的实现逻辑混合在一起，将它们全部硬编码 (Hard Coding) 在一个庞大的多重条件选择语句中，比直接继承环境类的办法还要原始和落后。
- 策略模式提供了一种算法的复用机制。由于将算法单独提取出来封装在策略类中，因此不同的环境类可以方便地复用这些策略类。

缺点:

- 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。
- 策略模式将造成系统产生很多具体策略类，任何细小的变化都将导致系统要增加一个新的具体策略类。

5.3 命令（Command）

5.3.1 介绍

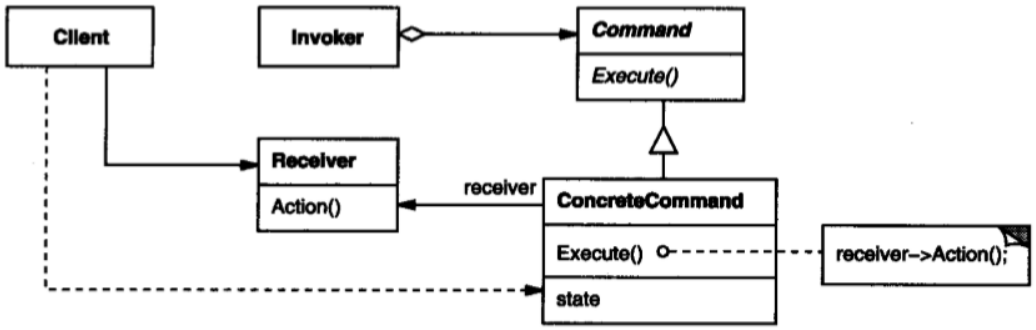
将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。命令模式是一种对象行为型模式，其别名为动作 (Action) 模式或事务 (Transaction) 模式。

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

以下一些情况适用 Command 模式:

- 当系统需要将请求调用者与请求接收者解耦时，命令模式使得调用者和接收者不直接交互。
- 当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能。
- 当系统需要执行一组操作时，命令模式可以定义宏命令来实现该功能。
- 方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复。

命令模式的结构图如下:

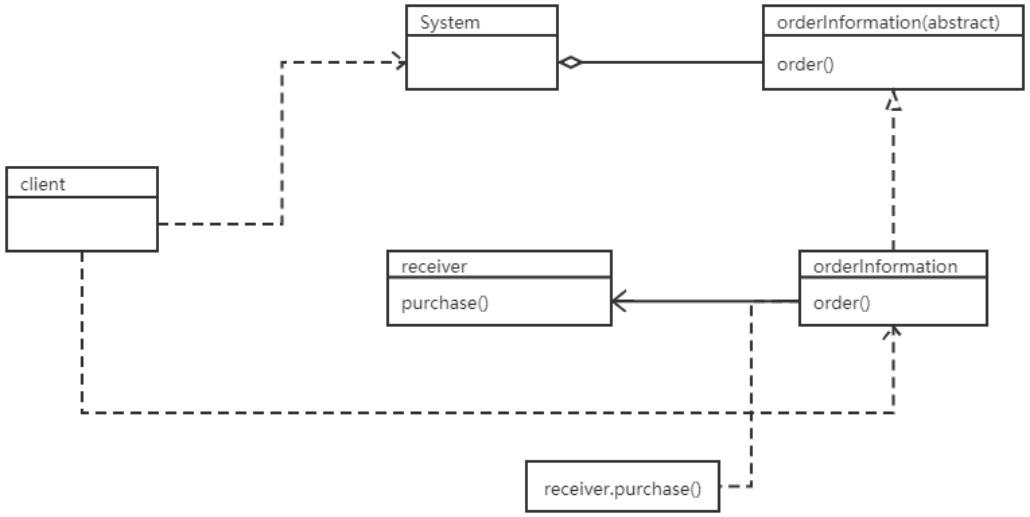


在上图中，我们可以看到一共有 4 类参与者:

- Command（抽象命令类）: 抽象命令类一般是一个抽象类或接口，在其中声明了用于执行请求的 execute() 等方法，通过这些方法可以调用请求接收者的相关操作。
- ConcreteCommand（具体命令类）: 具体命令类是抽象命令类的子类，实现了在抽象命令类中声明的方法，它对应具体的接收者对象，将接收者对象的动作绑定其中。在实现 execute() 方法时，将调用接收者对象的相关操作 (Action)。
- Invoker（调用者）: 调用者即请求发送者，它通过命令对象来执行请求。一个调用者并不需要在设计时确定其接收者，因此它只与抽象命令类之间存在关联关系。在程序运行时可以将一个具体命令对象注入其中，再调用具体命令对象的 execute() 方法，从而实现间接调用请求接收者的相关操作。
- Receiver（接收者）: 接收者执行与请求相关的操作，它具体实现对请求的业务处理。

5.3.2 举例

在本系统中，点单操作涉及到客户下单，系统收集客户下单信息，将处理好的订单总和统计给采购方，协助采购方进行购买。如果单纯以每笔订单的形式发送给采购方，采购方可能接受到大量重复的信息。所以系统进行初步整合减少耦合。



5.3.3 优缺点分析

优点:

- 降低系统的耦合度。由于请求者与接收者之间不存在直接引用，因此请求者与接收者之间实现完全解耦，相同的请求者可以对应不同的接收者，同样，相同的接收者也可以供不同的请求者使用，两者之间具有良好的独立性。
- 新的命令可以很容易地加入到系统中。由于增加新的具体命令类不会影响到其他类，因此增加新的具体命令类很容易，无须修改原有系统源代码，甚至客户类代码，满足“开闭原则”的要求。
- 可以比较容易地设计一个命令队列或宏命令（组合命令）。

缺点:

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

5.4 观察者（Observer）

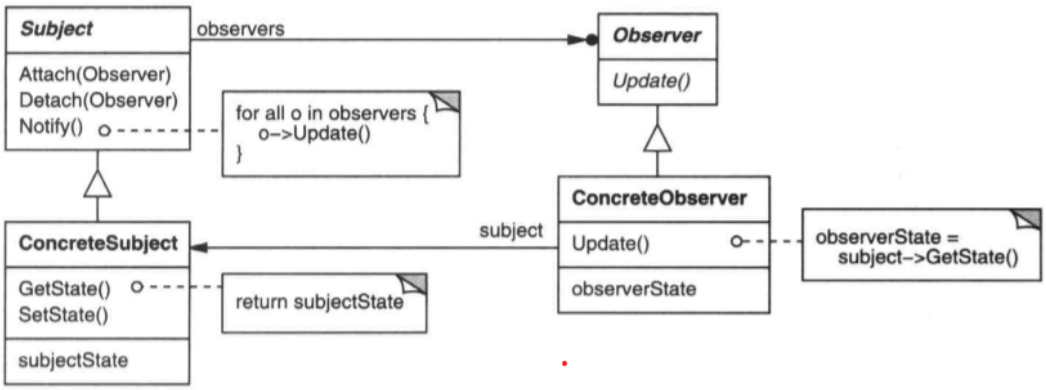
5.4.1 介绍

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

以下一些情况适用 Observer 模式:

- 要为一个复杂子系统提供一个简单接口时。
- 客户程序与抽象类的实现部分之间存在着很大的依赖。
- 构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。。

观察者模式的结构图如下:

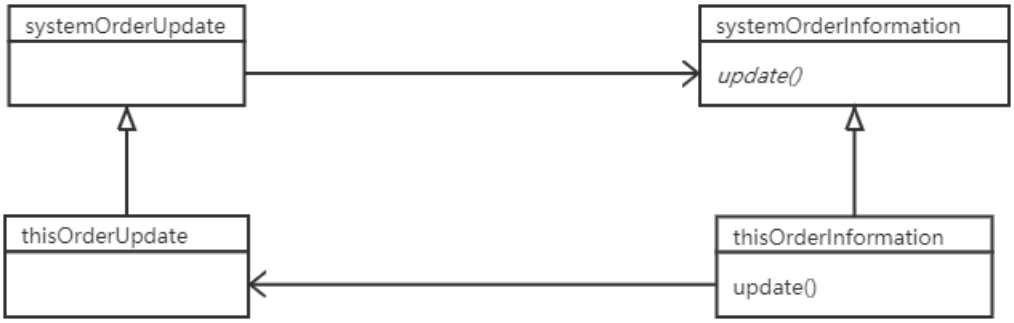


在上图中，我们可以看到一共有 4 类参与者:

- Subject（被观察者或目标，抽象主题）: 被观察的对象。当需要被观察的状态发生变化时，需要通知队列中所有观察者对象。Subject 需要维持（添加，删除，通知）一个观察者对象的队列列表。
- ConcreteSubject（具体被观察者或目标，具体主题）: 被观察者的具体实现。包含一些基本的属性状态及其他操作。
- Observer（观察者）: 接口或抽象类。当 Subject 的状态发生变化时 Observer 对象将通过一个 callback 函数得到通知。
- ConcreteObserver（具体观察者）: 观察者的具体实现。得到通知后将完成一些具体的业务逻辑处理。

5.4.2 举例

在本系统中，因为系统需要检测是否有新的订单产生，并将产生的订单信息和之前的订单信息做整合处理告知代购者，所以我们可以引入观察模式添加这一触发机制。



5.4.3 优缺点分析

优点:

- 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。
- 目标与观察者之间建立了一套触发机制。

缺点:

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

5.5 责任链（Chain of Responsibility）

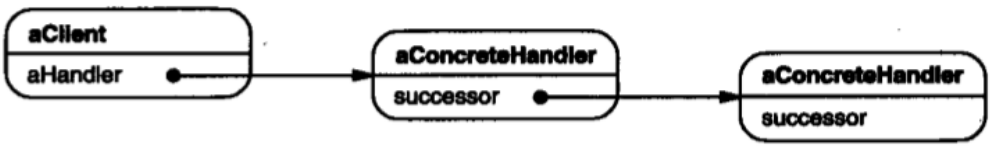
5.5.1 介绍

为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。这就是责任链（Chain of Responsibility）模式的定义。

以下一些情况适用 Chain of Responsibility 模式:

- 有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定。
- 可动态指定一组对象处理请求，或添加新的处理者。
- 在不明确指定请求处理者的情况下，向多个处理者中的一个提交请求。

责任链模式的结构图如下:

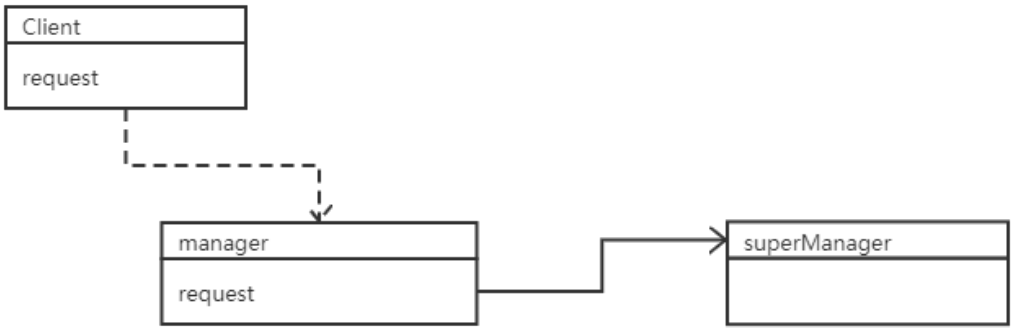


在上图中，我们可以看到一共有 2 类参与者：

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

5.5.2 举例

在本系统中，关于权限以及任务处理一共分为三种用户：普通用户，管理员以及超级管理员。所以我们可以采用责任链的方式对请求处理进行约束。



5.5.3 优缺点分析

优点:

- 降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
- 增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。
- 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。
- 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if $\cdots$ else 语句。
- 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

缺点：

- 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
- 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。

- 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

5.6 状态（State）

5.6.1 介绍

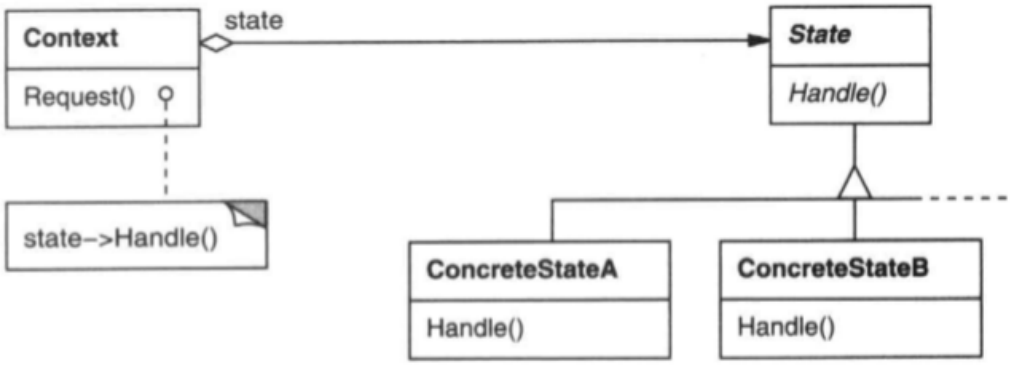
在软件开发过程中，应用程序中的有些对象可能会根据不同的情况做出不同的行为，我们把这种对象称为有状态的对象，而把影响对象行为的一个或多个动态变化的属性称为状态。当有状态的对象与外部事件产生互动时，其内部状态会发生改变，从而使得其行为也随之发生改变。如人的情绪有高兴的时候和伤心的时候，不同的情绪有不同的行为，当然外界也会影响其情绪变化。

当控制一个对象状态转换的条件表达式过于复杂时，把相关“判断逻辑”提取出来，放到一系列的状态类当中，这样可以把原来复杂的逻辑判断简单化。这就是状态模式的思想。

以下一些情况适用 State 模式:

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

状态模式的结构图如下：



在上图中，我们可以看到一共有 3 类参与者：

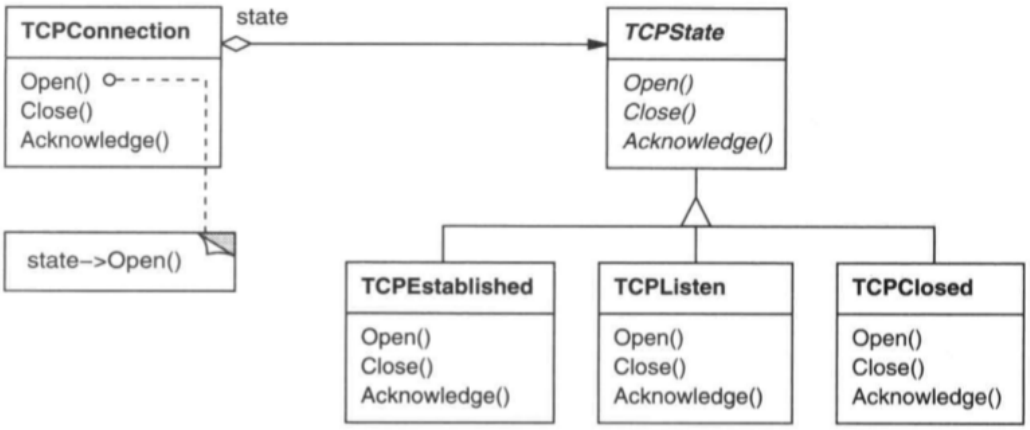
- Context（环境）：也称为上下文，它定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理。
- State（抽象状态）：定义一个接口，用以封装环境对象中的特定状态所对应的行为。
- ConcreteState（具体状态）：实现抽象状态所对应的行为。

5.6.2 举例

在本系统中，没有发现适用于此类设计模式的应用场景，这里以 TCP 连接做举例。

一个 TCPConnection 对象的状态处于若干不同状态之一: 连接已建立（Established)、正在监听(Listening)、连接已关闭 (Closed)。当一个 TCPConnection 对象收到其他对象的请求时, 它根据自身的当前状态作出不同的反应。

在这一模式的关键思想下,引入了一个称为 TCPState 的抽象类来表示网络的连接状态。TCPState 类为各表示不同的操作状态的子类声明了一个公共接口。TCPState 的子类实现与特定状态相关的行为。



5.6.3 优缺点分析

优点:

- 状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。
- 减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。
- 有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。

缺点:

- 状态模式的使用必然会增加系统的类与对象的个数。
- 状态模式的结构与实现都较为复杂，如果使用不当会导致程序结构和代码的混乱。

5.7 迭代器（Iterator）

5.7.1 介绍

既然将遍历方法封装在聚合类中不可取，那么聚合类中不提供遍历方法，将遍历方法由用户自己实现是否可行呢？答案是同样不可取，因为这种方式会存在两个缺点:

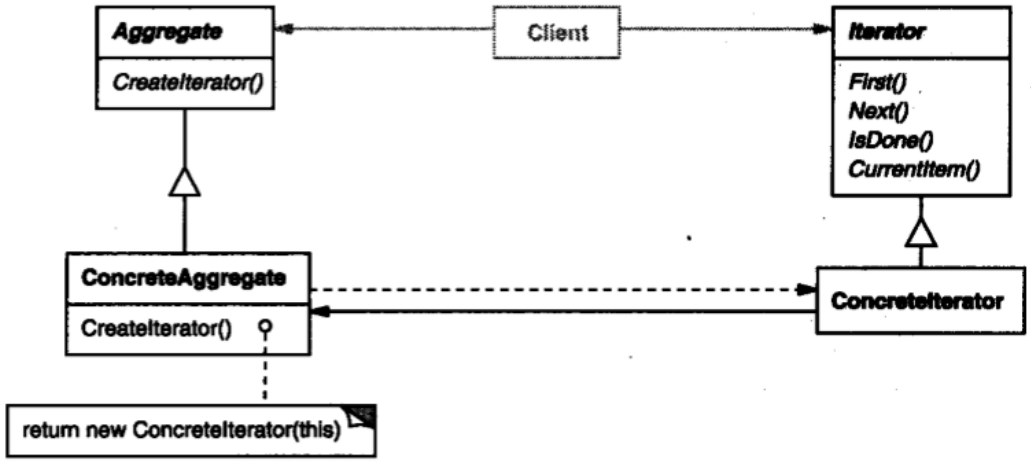
- 暴露了聚合类的内部表示，使其数据不安全.
- 增加了客户的负担。

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

以下一些情况适用 Iterator 模式:

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

迭代器模式的结构图如下:



在上图中，我们可以看到一共有 4 类参与者:

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

5.7.2 举例

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

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

5.7.3 优缺点分析

优点:

- 访问一个聚合对象的内容而无须暴露它的内部表示。
- 遍历任务交由迭代器完成，这简化了聚合类。
- 它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。
- 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
- 封装性良好，为遍历不同的聚合结构提供一个统一的接口。

缺点:

- 增加了类的个数，这在一定程度上增加了系统的复杂性。

5.8 访问者（Visitor）

5.8.1 介绍

在现实生活中，有些集合对象中存在多种不同的元素，且每种元素也存在多种不同的访问者和处理方式。

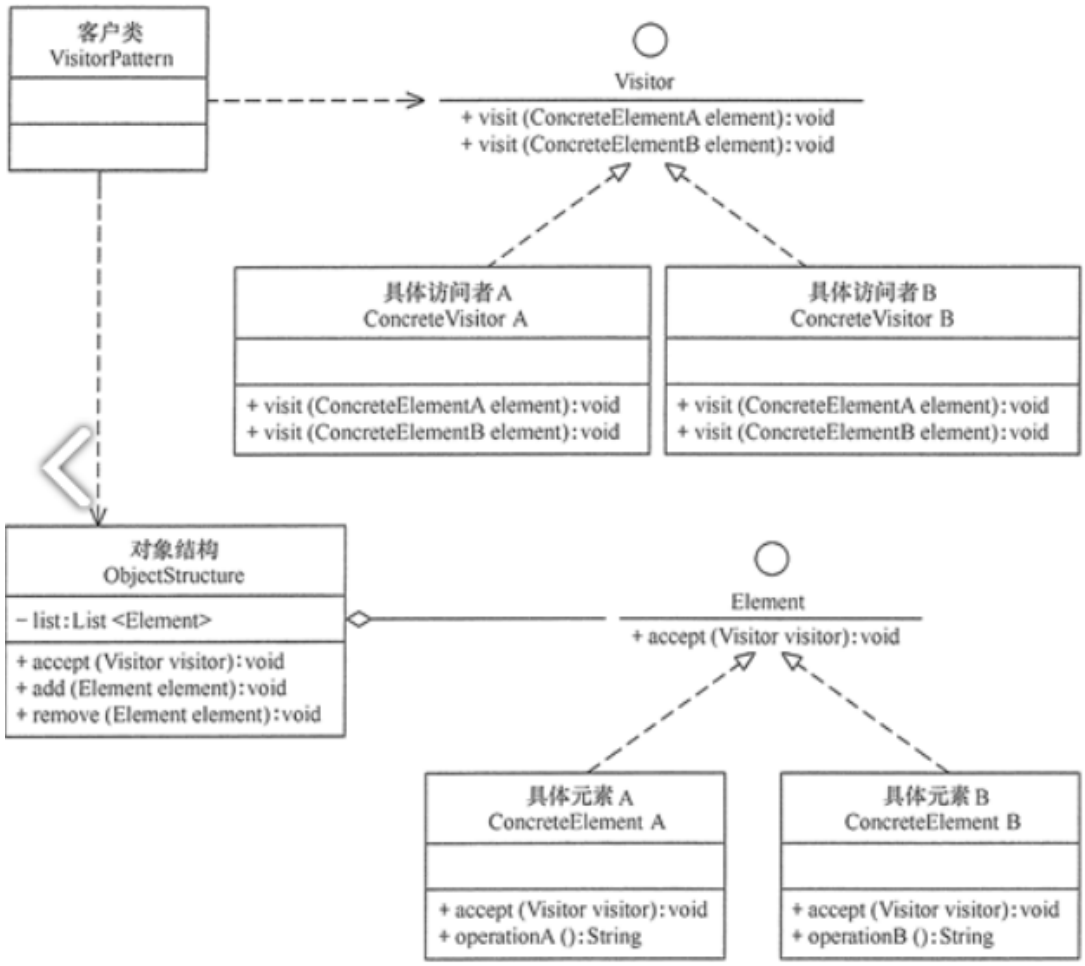
这样的例子还有很多，例如，电影或电视剧中的人物角色，不同的观众对他们的评价也不同；还有顾客在商场购物时放在“购物车”中的商品，顾客主要关心所选商品的性价比，而收银员关心的是商品的价格和数量。

这些被处理的数据元素相对稳定而访问方式多种多样的数据结构，如果用“访问者模式”来处理比较方便。访问者模式将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

以下一些情况适用 Visitor 模式:

- 对象结构相对稳定，但其操作算法经常变化的程序。
- 对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构。
- 对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作。

访问者模式的结构图如下:



在上图中，我们可以看到一共有 5 类参与者:

- Visitor（抽象访问者）: 定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit() ，该操作中的参数类型标识了被访问的具体元素。
- ConcreteVisitor(具体访问者): 实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。
- Element(抽象元素）：声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。

- ConcreteElement(具体元素)：实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this) ，另外具体元素中可能还包含本身业务逻辑的相关操作。
- ObjectStructure(对象结构)：是一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由 List、Set、Map 等聚合类实现。

5.8.2 举例

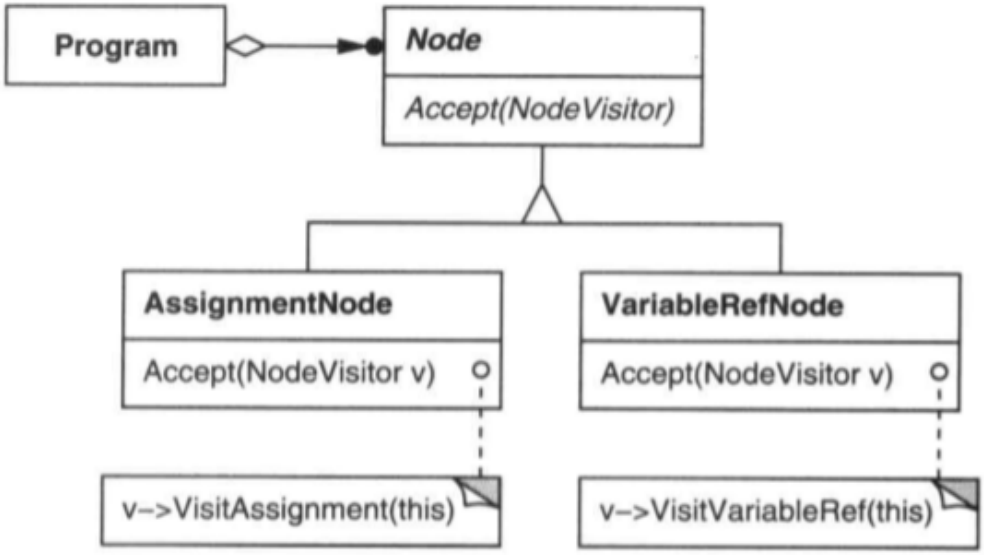
考虑一个编译器，它将源程序表示为一个抽象语法树。该编译器需在抽象语法树上实施某些操作以进行“静态语义”分析，例如检查是否所有的变量都已经被定义了。它也需要生成代码。因此它可能要定义许多操作以进行类型检查、代码优化、流程分析，检查变量是否在使用前被赋初值，等等。此外，还可使用抽象语法树进行优美格式打印、程序重构、code instrumentation 以及对程序进行多种度量。

这些操作大多要求对不同的节点进行不同的处理。例如对代表赋值语句的结点的处理就不同于对代表变量或算术表达式的结点的处理。因此有用于赋值语句的类，有用于变量访问的类，还有用于算术表达式的类，等等。结点类的集合当然依赖于被编译的语言，但对于一个给定的语言其变化不大。但是将所有这些操作分散到各种结点类中会导致整个系统难以理解、难以维护和修改。

我们可以将每一个类中相关的操作包装在一个独立的对象（称为一个 Visitor）中，并在遍历抽象语法树时将此对象传递给当前访问的元素。当一个元素“接受”该访问者时，该元素向访问者发送一个包含自身类信息的请求。该请求同时也将该元素本身作为一个参数。然后访问者将为该元素执行该操作—这一操作以前是在该元素的类中的。

一个不使用访问者的编译器可能会通过在它的抽象语法树上调用 TypeCheck 操作对一个过程进行类型检查。每一个结点将对调用它的成员的 TypeCheck 以实现自身的 TypeCheck。如果该编译器使用访问者对一个过程进行类型检查，那么它将会创建一个 TypeCheckingVisitor 对象，并以这个对象为一个参数在抽象语法树上调用 A c c e p t 操作。每一个结点在实现 Accept 时将会回调访问者：一个赋值结点调用访问者的 VisitAssignm ent 操作，而一个变量引用将调用 VisitVariableReference。以前类 AssignmentNode 的 TypeCheck 操作现在成为 TypeCheckingVisitor 的 VisitAssignment 操作。[8]

为使访问者不仅仅只做类型检查，我们需要所有抽象语法树的访问者有一个抽象的父类 NodeVisitor。NodeVisitor 必须为每一个结点类定义一个操作。一个需要计算程序度量的应用将定义 NodeVisitor 的新的子类，并且将不再需要在结点类中增加与特定应用相关的代码。



5.8.3 优缺点分析

优点:

- 扩展性好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。
- 复用性好。可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。
- 灵活性好。访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构。
- 符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。

缺点:

- 增加新的元素类很困难。在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”。
- 破坏封装。访问者模式中具体元素对访问者公布细节，这破坏了对象的封装性。
- 违反了依赖倒置原则。访问者模式依赖了具体类，而没有依赖抽象类。

5.9 备忘录（Memento）

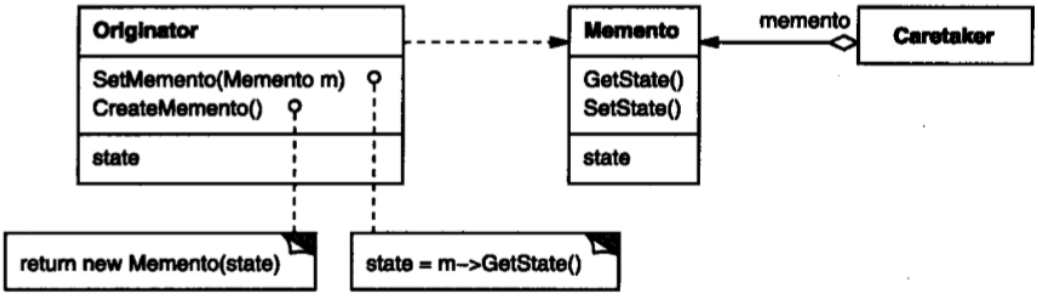
5.9.1 介绍

备忘录（Memento）在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。

以下一些情况适用 Memento 模式:

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

备忘录模式的结构图如下:



在上图中，我们可以看到一共有 3 类参与者:

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

5.9.2 举例

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

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

ConstraintSolver 可作为一个原发器。下面的事件序列描述了取消操作的过程:

- 作为移动操作的一个副作用, 编辑器向 ConstraintSolver 请求一个备忘录
- ConstraintSolver 创建并返回一个备忘录, 在这个例子中该备忘录是 SolverState 类的一个实例。SolverState 备忘录包含一些描述 ConstraintSolver 的内部等式和变量当前状态的数据结构。

- 此后当用户取消移动操作时, 编辑器将 SolverState 备忘录送回给 ConstraintSolver。
- 根据 SolverState 备忘录中的信息，ConstraintSolver 改变它的内部结构以精确地将它的等式和变量返回到它们各自先前的状态。

5.9.3 优缺点分析

优点:

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

缺点：

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

5.10 解释器（Interpreter）

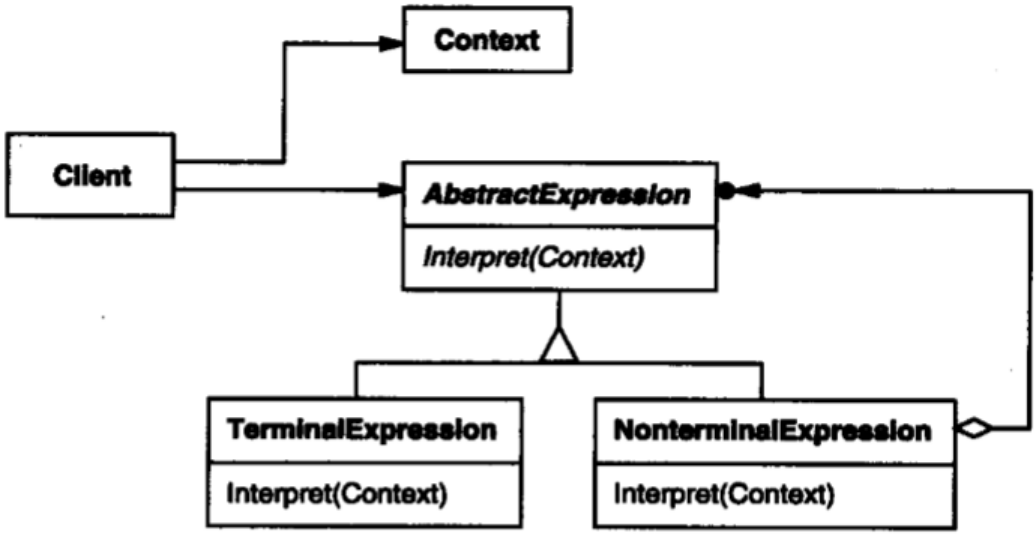
5.10.1 介绍

解释器（Interpreter）给分析对象定义一个语言，并定义该语言的文法表示，再设计一个解析器来解释语言中的句子。也就是说，用编译语言的方式来分析应用中的实例。这种模式实现了文法表达式处理的接口，该接口解释一个特定的上下文。

以下一些情况适用 Interpreter 模式:

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

解释器模式的结构图如下：



在上图中，我们可以看到一共有 5 类参与者：

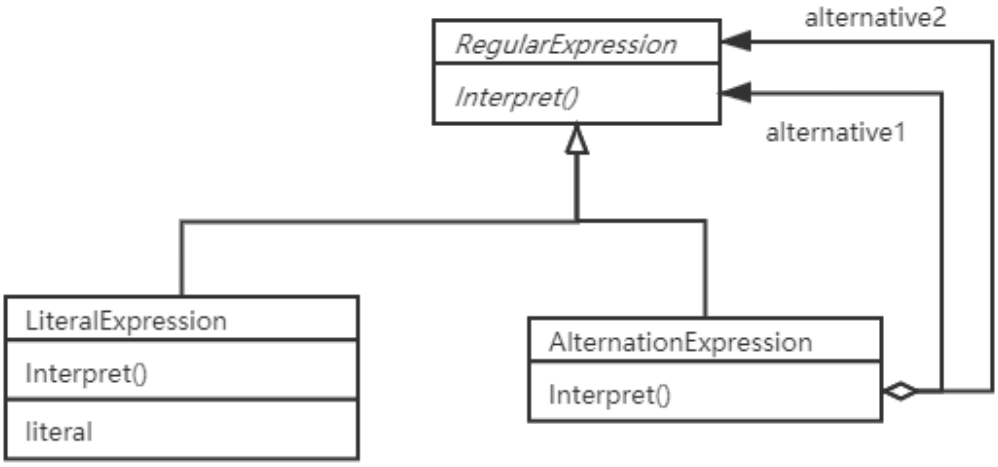
- AbstractExpression（抽象表达式）：定义解释器的接口，约定解释器的解释操作，主要包含解释方法 interpret()。
- TerminalExpression（终结符表达式）：是抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应。
- NonterminalExpression（非终结符表达式）：也是抽象表达式的子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式。
- Context（环境）：通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值。
- Client（客户端）：主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树，然后调用解释器的解释方法，当然也可以通过环境角色间接访问解释器的解释方法。

5.10.2 举例

解释器模式描述了如何为正则表达式定义一个文法, 如何表示一个特定的正则表达式, 以及如何解释这个正则表达式。考虑这个文法定义：

- expression ::= literal | alternation | '(' expression ')'
- alternation ::= expression '|' expression
- literal ::= 'a' | 'b' | 'c'

符号 expression 是开始符号,literal 是定义简单字的终结符。解释器模式使用类来表示每一条文法规则。在规则右边的符号是这些类的实例变量。上面的文法用五个类表示: 一个抽象类 RegularExpression 和它的子类 LiteralExpression、AlternationExpression 定义的变量代表子表达式。



5.10.3 优缺点分析

优点:

- 扩展性好。由于在解释器模式中使用类来表示语言的文法规则，因此可以通过继承等机制来改变或扩展文法。
- 容易实现。在语法树中的每个表达式节点类都是相似的，所以实现其文法较为容易。

缺点:

- 执行效率较低。解释器模式中通常使用大量的循环和递归调用，当要解释的句子较复杂时，其运行速度很慢，且代码的调试过程也比较麻烦。
- 会引起类膨胀。解释器模式中的每条规则至少需要定义一个类，当包含的文法规则很多时，类的个数将急剧增加，导致系统难以管理与维护。
- 可应用的场景比较少。在软件开发中，需要定义语言文法的应用实例非常少，所以这种模式很少被使用到。

5.11 中介者（Mediator）

5.11.1 介绍

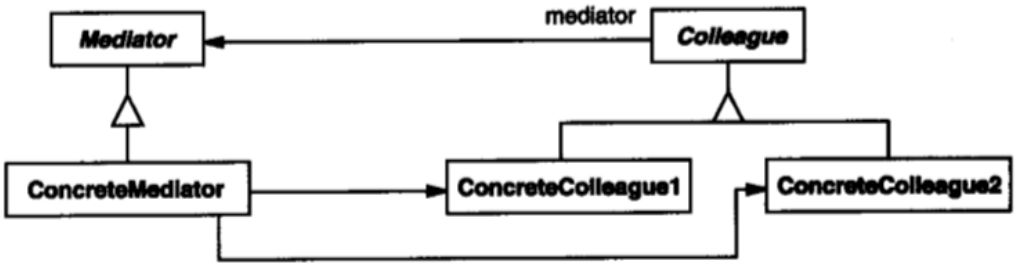
在现实生活中，常常会出现好多对象之间存在复杂的交互关系，这种交互关系常常是“网状结构”，它要求每个对象都必须知道它需要交互的对象。

中介者模式就是定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

以下一些情况适用 Mediator 模式:

- 当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用时。
- 当想创建一个运行于多个类之间的对象，又不想生成新的子类时。

中介者模式的结构图如下:

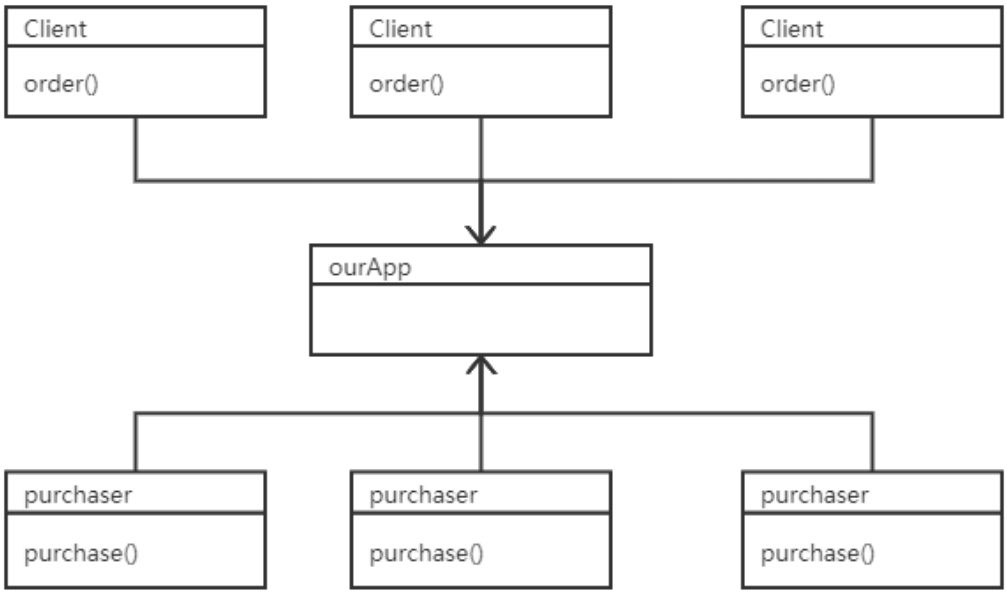


在上图中，我们可以看到一共有 4 类参与者:

- Mediator（抽象中介者）: 它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。
- ConcreteMediator（具体中介者）: 实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。
- Colleague（抽象同事类）: 定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。
- ConcreteColleague（具体同事类）: 是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。

5.11.2 举例

在本系统中，没有使用中介者的具体例子，但是本系统本身就是中介者模式的一种实例。本系统作为采购方代购和客户下单的中间环节，帮助进行订单统计与信息交互。



5.11.3 优缺点分析

优点:

- 降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
- 增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。
- 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。
- 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if $\cdots$ else 语句。
- 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

缺点:

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

参考文献

[1] H. Thaller, L. Linsbauer, and A. Egyed, "Feature maps: A comprehensible software representation for design pattern detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 207–217, IEEE, 2019.

[2] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[3] S. Hussain, J. Keung, M. K. Sohail, A. A. Khan, and M. Ilahi, "Automated framework for classification and selection of software design patterns," *Applied Soft Computing*, vol. 75, pp. 1–20, 2019.

[4] S. Chaturvedi, A. Chaturvedi, A. Tiwari, and S. Agarwal, "Design pattern detection using machine learning techniques," in *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, pp. 1–6, IEEE, 2018.

[5] D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, "Efficiently detecting structural design pattern instances based on ordered sequences," *Journal of Systems and Software*, vol. 142, pp. 35–56, 2018.

[6] D. Reimanis and C. Izurieta, "Behavioral evolution of design patterns: Understanding software reuse through the evolution of pattern behavior," in *International Conference on Software and Systems Reuse*, pp. 77–93, Springer, 2019.

[7] T. Reischmann and H. Kuchen, "An interactive learning environment for software engineering design patterns," in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, pp. 1–2, 2018.

[8] R. J. Erich Gamma, Richard Helm and J. Vlissides in *Design Patterns Elements of Reusable Object Oriented Software*, p. 94, Addison-Wesley, 1995.