# 设计模式

1. 单例
2. 原型
3. 工厂方法
4. 抽象工厂
5. 建造者
6. 代理
7. 适配器
8. 桥接
9. 装饰
10. 外观
11. 享元
12. 组合
13. 模板方法
14. 策略
15. 命令
16. 职责链
17. 状态
18. 观察者
19. 中介者
20. 迭代器
21. 访问者
22. 备忘录
23. 解释器

**创建型模式**：怎样创建对象？ 将对象的创建和使用分离

1. 单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。

特点：

单例类只有一个实例对象；

该单例对象必须由单例类自行创建；

单例类对外提供一个访问该单例的全局访问点

优点：

单例模式可以保证内存里只有一个实例，减少了内存的开销。

可以避免对资源的多重占用。

单例模式设置全局访问点，可以优化和共享资源的访问。

缺点：

单例模式一般没有接口，扩展困难。如果要扩展，则除了修改原来的代码，没有第二种 途径，违背开闭原则。

在并发测试中，单例模式不利于代码调试。在调试过程中，如果单例中的代码没有执行 完，也不能模拟生成一个新的对象。

单例模式的功能代码通常写在一个类中，如果功能设计不合理，则很容易违背单一职责 原则。

应用场景

需要频繁创建的一些类，使用单例可以降低系统的内存压力，减少 GC。

某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等。

某些类创建实例时占用资源较多，或实例化耗时较长，且经常使用。

某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

频繁访问数据库或文件的对象。

对于一些控制硬件级别的操作，或者从系统上来讲应当是单一控制逻辑的操作，如果有多个实例，则系统会完全乱套。

当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

1. 原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。

优点：

[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 自带的原型模式基于内存二进制流的复制，在性能上比直接 new 一个对象更加优 良。

可以使用深克隆方式保存对象的状态，使用原型模式将对象复制一份，并将其状态保存 起来，简化了创建对象的过程，以便在需要的时候使用（例如恢复到历史某一状态）， 可辅助实现撤销操作。

缺点：

需要为每一个类都配置一个 clone 方法

clone 方法位于类的内部，当对已有类进行改造的时候，需要修改代码，违背了开闭原 则。

当实现深克隆时，需要编写较为复杂的代码，而且当对象之间存在多重嵌套引用时，为 了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来会比较麻烦。因此， 深克隆、浅克隆需要运用得当。

包含以下主要角色：

抽象原型类：规定了具体原型对象必须实现的接口。

具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象。

访问类：使用具体原型类中的 clone() 方法来复制新的对象。

实现clonable接口

或者通过序列化

应用场景

对象之间相同或相似，即只是个别的几个属性不同的时候。

创建对象成本较大，例如初始化时间长，占用CPU太多，或者占用网络资源太多等，需要优化资源。

创建一个对象需要繁琐的数据准备或访问权限等，需要提高性能或者提高安全性。

系统中大量使用该类对象，且各个调用者都需要给它的属性重新赋值。

1. 工厂方法（FactoryMethod）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。

优点：

用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。

灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。

典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米 特法则、依赖倒置原则和里氏替换原则。

缺点：

类的个数容易过多，增加复杂度

增加了系统的抽象性和理解难度

抽象方法只能生产一种产品，此弊端可使用[抽象工厂模式](http://c.biancheng.net/view/1351.html" \t "http://c.biancheng.net/view/_blank)解决。

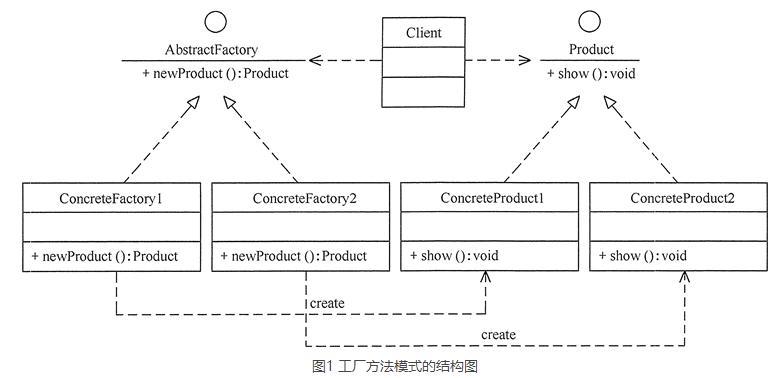
工厂方法模式的主要角色如下。

抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通过它访问具体工厂的 工厂方法 newProduct() 来创建产品。

具体工厂（ConcreteFactory）：主要是实现抽象工厂中的抽象方法，完成具体产品的创 建。

抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能。

具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接口，由具体工厂来创建， 它同具体工厂之间一一对应。



每需要创建一个新产品就添加一个具体工厂类，一个工厂只负责生产一种产品

应用场景

客户只知道创建产品的工厂名，而不知道具体的产品名。如 TCL 电视工厂、海信电视工厂等。

创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口。

客户不关心创建产品的细节，只关心产品的品牌

1. 抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

当系统中只存在一个等级结构的产品时，抽象工厂模式将退化到工厂方法模式。

抽象工厂模式除了具有工厂方法模式的优点外，

其他主要优点如下。

可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。

当需要产品族时，抽象工厂可以保证客户端始终只使用同一个产品的产品组。

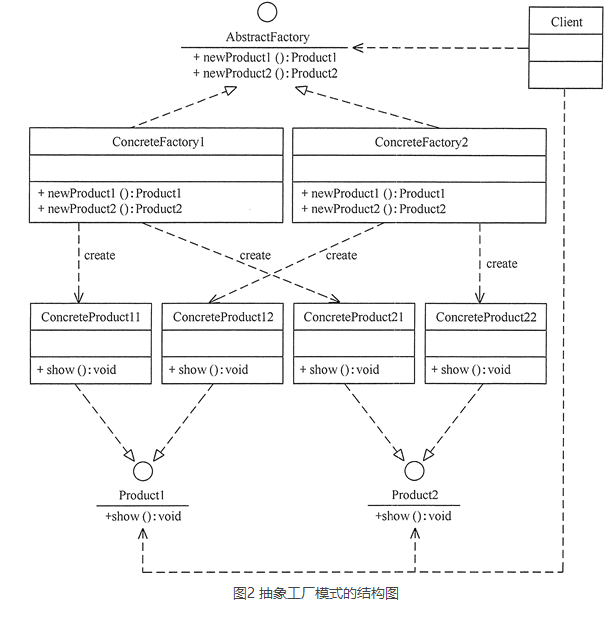
抽象工厂增强了程序的可扩展性，当增加一个新的产品族时，不需要修改原代码，满足开闭原则。

产品族：比如品牌A，B。

不同等级：比如鞋子，包包

缺点：

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。增加了系统的抽象性和理解难度



应用场景：

当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等。

系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋。

系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

1. 建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

指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

优点：

封装性好，构建和表示分离。

扩展性好，各个具体的建造者相互独立，有利于系统的解耦。

客户端不必知道产品内部组成的细节，建造者可以对创建过程逐步细化，而不对其它模 块产生任何影响，便于控制细节风险。

缺点：

产品的组成部分必须相同，这限制了其使用范围。

如果产品的内部变化复杂，如果产品内部发生变化，则建造者也要同步修改，后期维护 成本较大。

建造者（Builder）模式和工厂模式的关注点不同：

建造者模式注重零部件的组装过程，

而[工厂方法模式](http://c.biancheng.net/view/1348.html" \t "http://c.biancheng.net/view/_blank)更注重零部件的创建过程，但两者可以结合使用。

建造者模式更加注重方法的调用顺序，工厂模式注重创建对象。

创建对象的力度不同，建造者模式创建复杂的对象，由各种复杂的部件组成，工厂模式创建出来的对象都一样

关注重点不一样，工厂模式只需要把对象创建出来就可以了，而建造者模式不仅要创建出对象，还要知道对象由哪些部件组成。

建造者模式根据建造过程中的顺序不一样，最终对象部件组成也不一样。

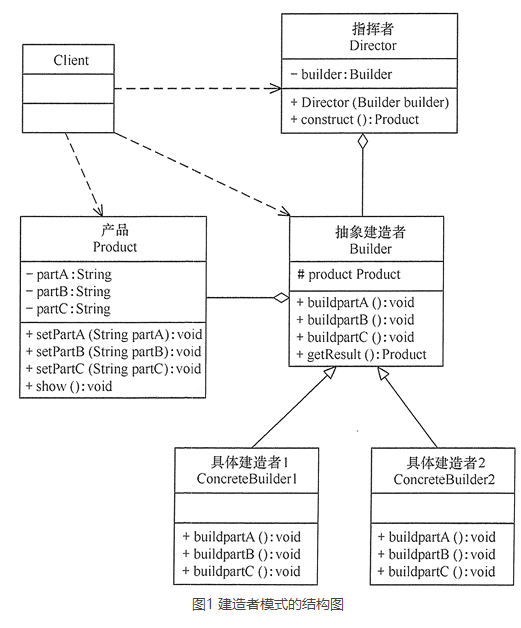
建造者（Builder）模式的主要角色如下。

产品角色（Product）：它是包含多个组成部件的复杂对象，由具体建造者来创建其各 个零部件。

抽象建造者（Builder）：它是一个包含创建产品各个子部件的抽象方法的接口，通常还 包含一个返回复杂产品的方法 getResult()。

具体建造者(Concrete Builder）：实现 Builder 接口，完成复杂产品的各个部件的具体创 建方法。

指挥者（Director）：它调用建造者对象中的部件构造与装配方法完成复杂对象的创建， 在指挥者中不涉及具体产品的信息。



应用场景

相同的方法，不同的执行顺序，产生不同的结果。

多个部件或零件，都可以装配到一个对象中，但是产生的结果又不相同。

产品类非常复杂，或者产品类中不同的调用顺序产生不同的作用。

初始化一个对象特别复杂，参数多，而且很多参数都具有默认值。**结构型模式：**将类或对象按某种布局组成更大的结构

分为类结构型模式和对象结构型模式，前者采用继承机制来组织接口和类，后者釆用组合或聚合来组合对象

1. 代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。（静态）

优点：

代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；

代理对象可以扩展目标对象的功能；

代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序 的可扩展性

缺点：

代理模式会造成系统设计中类的数量增加

在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢；

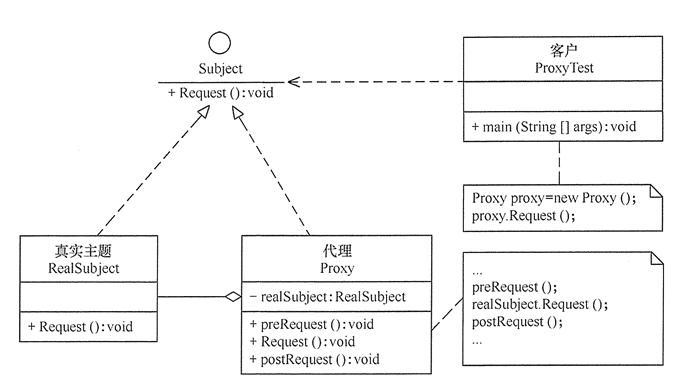
增加了系统的复杂度；

代理模式的主要角色如下。

抽象主题（Subject）类：通过接口或抽象类声明真实主题和代理对象实现的业务方法。

真实主题（Real Subject）类：实现了抽象主题中的具体业务，是代理对象所代表的真实 对象，是最终要引用的对象。

代理（Proxy）类：提供了与真实主题相同的接口，其内部含有对真实主题的引用，它 可以访问、控制或扩展真实主题的功能。



应用场景：

远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。

虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。

安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。

智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。

延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，Hibernate 中就存在属性的延迟加载和关联表的延时加载。

1. 适配器（Adapter）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能一起工作。

适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

优点。

客户端通过适配器可以透明地调用目标接口。

复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类。

将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。

在很多业务场景中符合开闭原则。

缺点：

适配器编写过程需要结合业务场景全面考虑，可能会增加系统的复杂性。

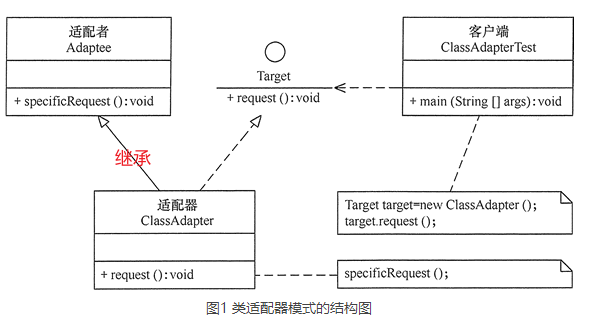
增加代码阅读难度，降低代码可读性，过多使用适配器会使系统代码变得凌乱。

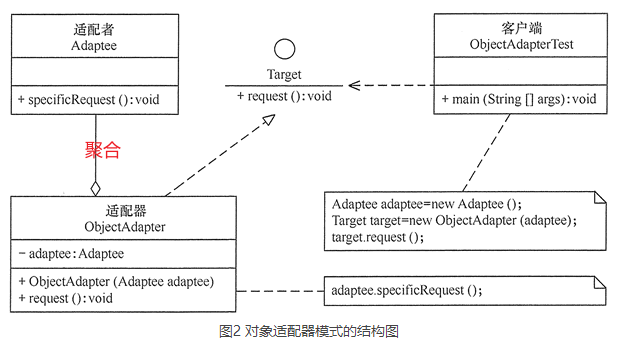
适配器模式（Adapter）包含以下主要角色。

目标（Target）接口：当前系统业务所期待的接口，它可以是抽象类或接口。

适配者（Adaptee）类：它是被访问和适配的现存组件库中的组件接口。

适配器（Adapter）类：它是一个转换器，通过继承或引用适配者的对象，把适配者接 口转换成目标接口，让客户按目标接口的格式访问适配者。





应用场景：

以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致。

使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

1. 桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现的，从而降低了抽象和实现这两个可变维度的耦合度。

场景：在现实生活中，某些类具有两个或多个维度的变化，如图形既可按形状分，又可按颜色分。如何设计类似于 Photoshop 这样的软件，能画不同形状和不同颜色的图形呢？如果用继承方式，m 种形状和 n 种颜色的图形就有 m×n 种，不但对应的子类很多，而且扩展困难。当一个类内部具备两种或多种变化维度时，使用桥接模式可以解耦这些变化的维度，使高层代码架构稳定。

优点：

抽象与实现分离，扩展能力强

符合开闭原则

符合合成复用原则

其实现细节对客户透明

缺点：

由于聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，能正确地识别出系统中两个独立变化的维度，这增加了系统的理解与设计难度

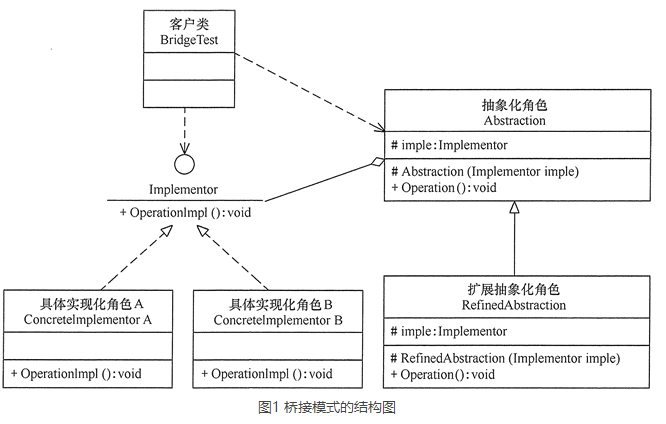
桥接（Bridge）模式包含以下主要角色。

抽象化（Abstraction）角色：定义抽象类，并包含一个对实现化对象的引用。

扩展抽象化（Refined Abstraction）角色：是抽象化角色的子类，实现父类中的业务方法， 并通过组合关系调用实现化角色中的业务方法。

实现化（Implementor）角色：定义实现化角色的接口，供扩展抽象化角色调用。

具体实现化（Concrete Implementor）角色：给出实现化角色接口的具体实现。



应用场景：

当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时。

当一个系统不希望使用继承或因为多层次继承导致系统类的个数急剧增加时。

当一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性时。

1. 装饰（Decorator）模式：指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。

优点：

装饰器是继承的有力补充，比继承灵活，在不改变原有对象的情况下，动态的给一个对 象扩展功能，即插即用

通过使用不用装饰类及这些装饰类的排列组合，可以实现不同效果

装饰器模式完全遵守开闭原则

缺点：

装饰器模式会增加许多子类，过度使用会增加程序得复杂性。

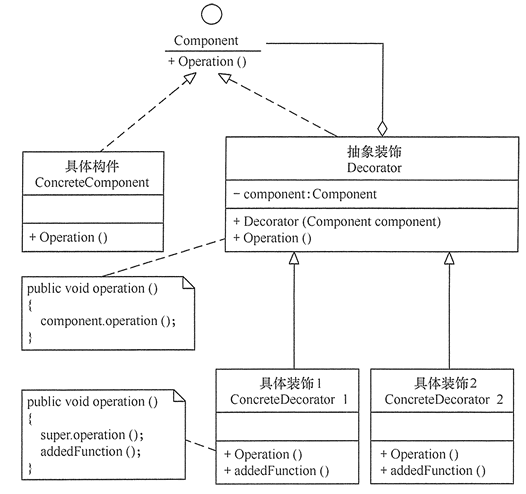
装饰器模式主要包含以下角色。

抽象构件（Component）角色：定义一个抽象接口以规范准备接收附加责任的对象。

具体构件（ConcreteComponent）角色：实现抽象构件，通过装饰角色为其添加一些职 责。

抽象装饰（Decorator）角色：继承抽象构件，并包含具体构件的实例，可以通过其子类 扩展具体构件的功能。

具体装饰（ConcreteDecorator）角色：实现抽象装饰的相关方法，并给具体构件对象添 加附加的责任。



应用场景：

当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类。

当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰器模式却很好实现。

当对象的功能要求可以动态地添加，也可以再动态地撤销时

1. 外观（Facade）模式：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体细节

优点。

降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。

对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容 易。

降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编 译一个子系统不会影响其他的子系统，也不会影响外观对象。

缺点。

不能很好地限制客户使用子系统类，很容易带来未知风险。

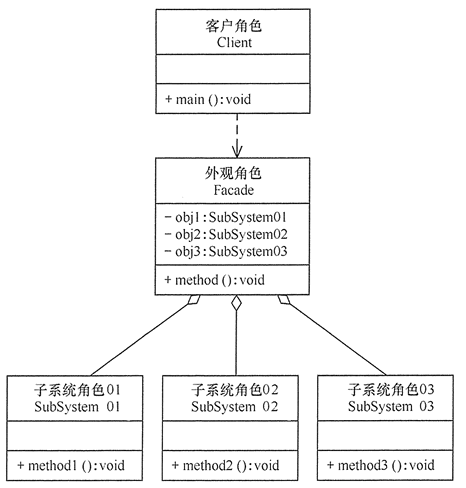
增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

外观（Facade）模式包含以下主要角色。

外观（Facade）角色：为多个子系统对外提供一个共同的接口。

子系统（Sub System）角色：实现系统的部分功能，客户可以通过外观角色访问它。

客户（Client）角色：通过一个外观角色访问各个子系统的功能。



应用场景：

对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系。

当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问。

当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

1. 享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。通过共享已经存在的对象来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率。

享元模式的定义提出了两个要求，细粒度和共享对象。因为要求细粒度，所以不可避免地会使对象数量多且性质相近，此时我们就将这些对象的信息分为两个部分：内部状态和外部状态。

内部状态指对象共享出来的信息，存储在享元信息内部，并且不回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，随环境的改变而改变，不可共享。

场景：

当系统中多处需要同一组信息时，可以把这些信息封装到一个对象中，然后对该对象进行缓存，这样，一个对象就可以提供给多出需要使用的地方，避免大量同一对象的多次创建，降低大量内存空间的消耗。  
  
 享元模式其实是[工厂方法模式](http://c.biancheng.net/view/1348.html" \t "http://c.biancheng.net/view/_blank)的一个改进机制，享元模式同样要求创建一个或一组对象，并且就是通过工厂方法模式生成对象的，只不过享元模式为工厂方法模式增加了缓存这一功能。

优点：

相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给 内存带来的压力。  
  
缺点：

为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性。

读取享元模式的外部状态会使得运行时间稍微变长。

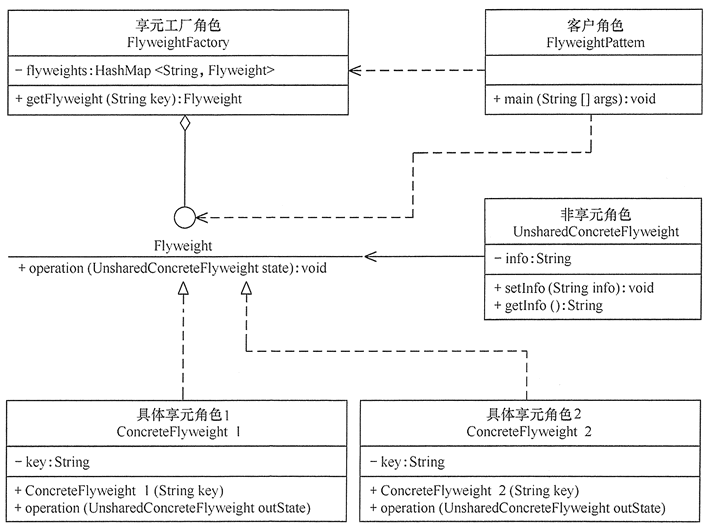
享元模式的主要角色有如下。

抽象享元角色（Flyweight）：是所有的具体享元类的基类，为具体享元规范需要实现的 公共接口，非享元的外部状态以参数的形式通过方法传入。

具体享元（Concrete Flyweight）角色：实现抽象享元角色中所规定的接口。

非享元（Unsharable Flyweight)角色：是不可以共享的外部状态，它以参数的形式注入具 体享元的相关方法中。

享元工厂（Flyweight Factory）角色：负责创建和管理享元角色。当客户对象请求一个 享元对象时，享元工厂检査系统中是否存在符合要求的享元对象，如果存在则提供给客 户；如果不存在的话，则创建一个新的享元对象。



应用场景

系统中存在大量相同或相似的对象，这些对象耗费大量的内存资源。

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

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

1. 组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

优点：

组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码；

更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”；

缺点：

设计较复杂，客户端需要花更多时间理清类之间的层次关系；

不容易限制容器中的构件；

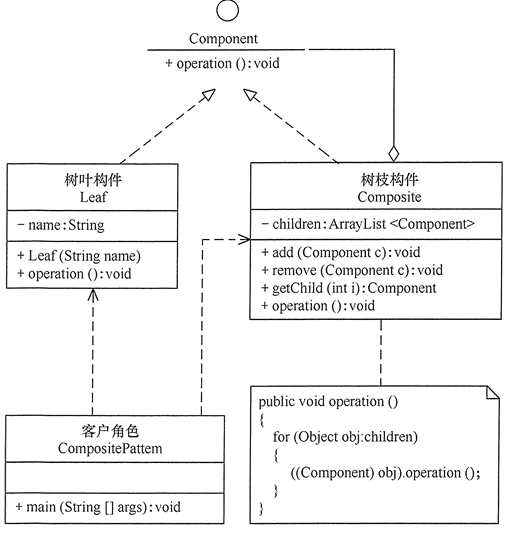
不容易用继承的方法来增加构件的新功能

组合模式包含以下主要角色。

抽象构件（Component）角色：它的主要作用是为树叶构件和树枝构件声明公共接口，并实现它们的默认行为。在透明式的组合模式中抽象构件还声明访问和管理子类的接口；在安全式的组合模式中不声明访问和管理子类的接口，管理工作由树枝构件完成。（总的抽象类或接口，定义一些通用的方法，比如新增、删除）

树叶构件（Leaf）角色：是组合中的叶节点对象，它没有子节点，用于继承或实现抽象构件。

树枝构件（Composite）角色 / 中间构件：是组合中的分支节点对象，它有子节点，用于继承和实现抽象构件。它的主要作用是存储和管理子部件，通常包含 Add()、Remove()、GetChild() 等方法。



使用场景：

在需要表示一个对象整体与部分的层次结构的场合。

要求对用户隐藏组合对象与单个对象的不同，用户可以用统一的接口使用组合结构中的所有对象的场合。

**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

1. 模板方法（Template Method）模式：定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在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。

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，间接地增加了系统实现的复杂度。

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

由于继承关系自身的缺点，如果父类添加新的抽象方法，则所有子类都要改一遍。

模板方法模式包含以下主要角色

1）抽象类/抽象模板（Abstract Class）

抽象模板类，负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下。  
  
① 模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。  
② 基本方法：是整个算法中的一个步骤，包含以下几种类型。

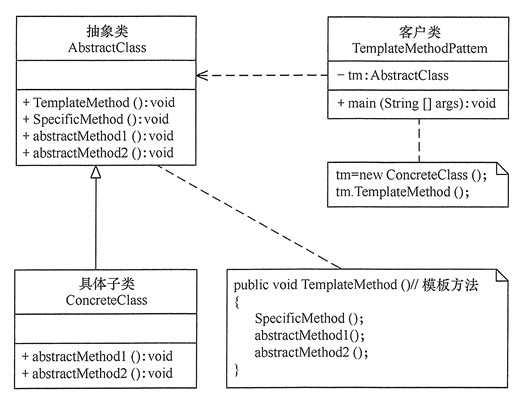
抽象方法：在抽象类中声明，由具体子类实现。

具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它。

钩子方法：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两 种。

2）具体子类/具体实现（Concrete Class）

具体实现类，实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。



模板方法模式通常适用于以下场景。

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。

1. 策略（Strategy）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，且算法的改变不会影响使用算法的客户。

优点：

多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句，如 if...else 语句、 switch...case 语句。

策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转 移到父类里面，从而避免重复的代码。

策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同 的。

策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算 法。

策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的 分离。

缺点：

客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类。

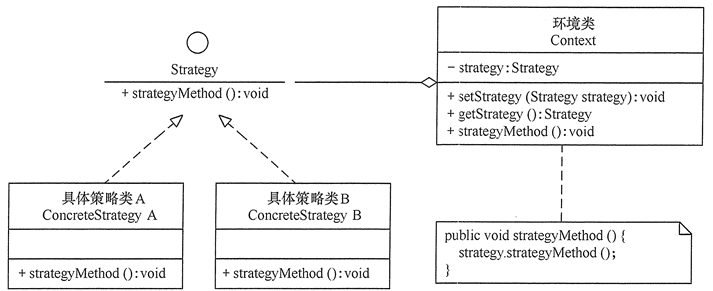
策略模式造成很多的策略类，增加维护难度。

策略模式的主要角色如下。

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。



通常在以下几种情况中使用策略模式较多。

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](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)。

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。

1. 命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理

优点。

通过引入中间件（抽象接口）降低系统的耦合度。

扩展性良好，增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类， 且满足“开闭原则”。

可以实现宏命令。命令模式可以与[组合模式](http://c.biancheng.net/view/1373.html" \t "http://c.biancheng.net/view/_blank)结合，将多个命令装配成一个组合命令，即 宏命令。

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](http://c.biancheng.net/view/1400.html" \t "http://c.biancheng.net/view/_blank)结合，实现命 令的撤销与恢复。

可以在现有命令的基础上，增加额外功能。比如日志记录，结合装饰器模式会更加灵活。

缺点：

可能产生大量具体的命令类。因为每一个具体操作都需要设计一个具体命令类，这会增 加系统的复杂性。

命令模式的结果其实就是接收方的执行结果，但是为了以命令的形式进行架构、解耦请 求与实现，引入了额外类型结构（引入了请求方与抽象命令接口），增加了理解上的困 难。不过这也是[设计模式](http://c.biancheng.net/design_pattern/" \t "http://c.biancheng.net/view/_blank)的通病，抽象必然会额外增加类的数量，代码抽离肯定比代码 聚合更加难理解。

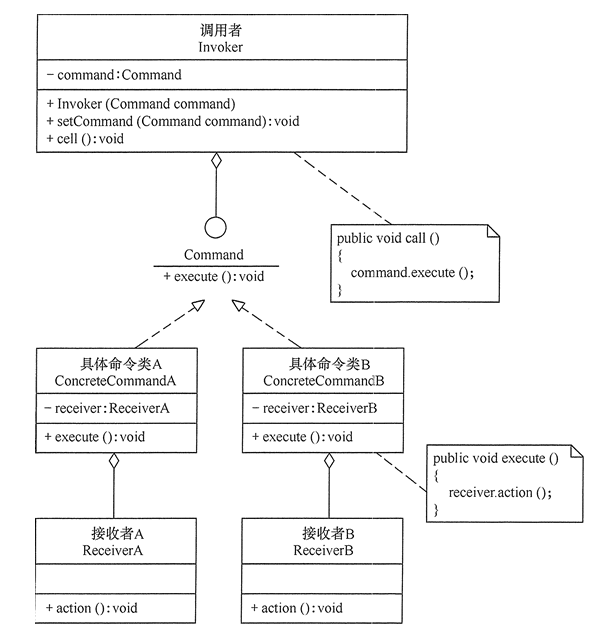
命令模式包含以下主要角色。

抽象命令类（Command）角色：声明执行命令的接口，拥有执行命令的抽象方法 execute()。

具体命令类（Concrete Command）角色：是抽象命令类的具体实现类，它拥有接收者 对象，并通过调用接收者的功能来完成命令要执行的操作。

实现者/接收者（Receiver）角色：执行命令功能的相关操作，是具体命令对象业务的真 正实现者。

调用者/请求者（Invoker）角色：是请求的发送者，它通常拥有很多的命令对象，并通 过访问命令对象来执行相关请求，它不直接访问接收者。



命令模式通常适用于以下场景。

请求调用者需要与请求接收者解耦时，命令模式可以使调用者和接收者不直接交互。

系统随机请求命令或经常增加、删除命令时，命令模式可以方便地实现这些功能。

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。

当系统需要支持命令的撤销（Undo）操作和恢复（Redo）操作时，可以将命令对象存 储起来，采用备忘录模式来实现。

1. 职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

优点。

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

增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。

增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或 者调动它们的次序，也可动态地新增或者删除责任。

责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持 其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。

责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成， 明确各类的责任范围，符合类的单一职责原则。

缺点。

不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定 会被处理，该请求可能一直传到链的末端都得不到处理。

对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。

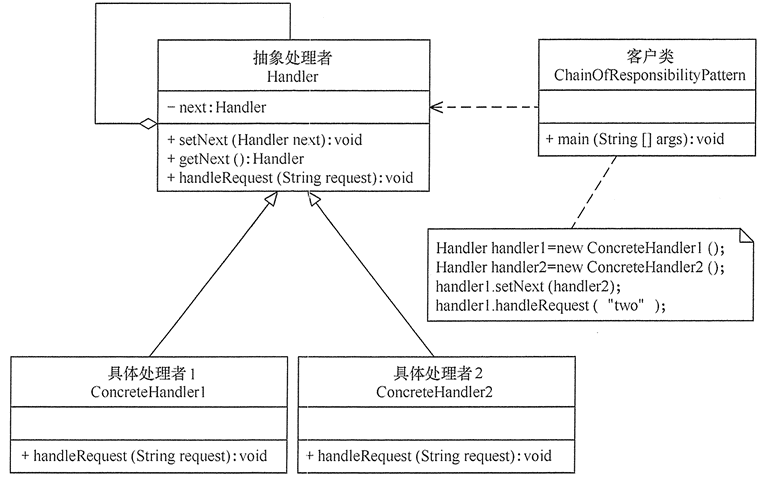
职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的 错误设置而导致系统出错，如可能会造成循环调用。

职责链模式主要包含以下角色。

抽象处理者（Handler）角色：定义一个处理请求的接口，包含抽象处理方法和一个后 继连接。

具体处理者（Concrete Handler）角色：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。

客户类（Client）角色：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。



责任链模式通常在以下几种情况使用。

多个对象可以处理一个请求，但具体由哪个对象处理该请求在运行时自动确定。

可动态指定一组对象处理请求，或添加新的处理者。

需要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

1. 状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为

优点。

结构清晰，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”。

将状态转换显示化，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。

状态类职责明确，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。

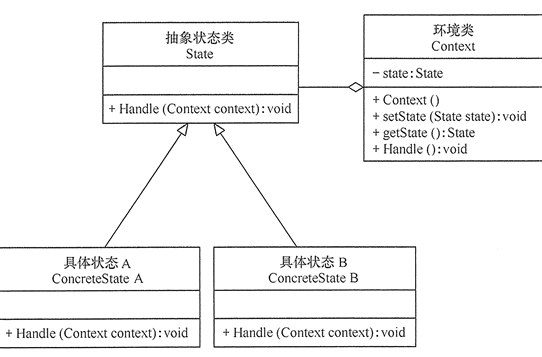
状态模式对开闭原则的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源码，否则无法切换到新增状态，而且修改某个状态类的行为也需要修改对应类的源码。

状态模式包含以下主要角色。

环境类（Context）角色：也称为上下文，它定义了客户端需要的接口，内部维护一个当前状态，并负责具体状态的切换。

抽象状态（State）角色：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，可以有一个或多个行为。

具体状态（Concrete State）角色：实现抽象状态所对应的行为，并且在需要的情况下进行状态切换。



通常在以下情况下可以考虑使用状态模式。

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。

1. 观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

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。符合依赖倒置原则。

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

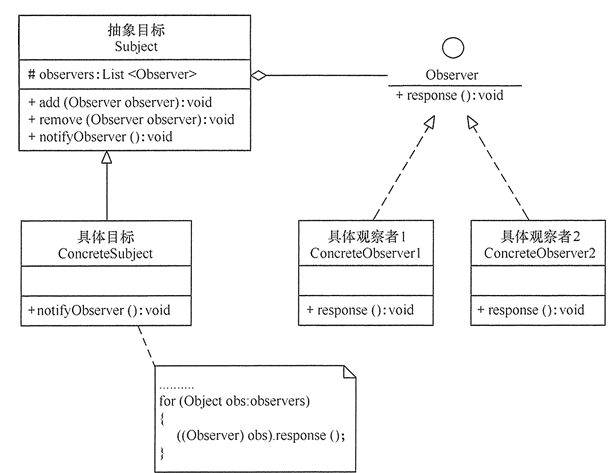
观察者模式的主要角色如下。

抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集 类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

具体主题（Concrete Subject）角色：也叫具体目标类，它实现抽象目标中的通知方法， 当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象 方法，当接到具体主题的更改通知时被调用。

具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得 到目标的更改通知时更新自身的状态。



通过前面的分析与应用实例可知观察者模式适合以下几种情形。

对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。

当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可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。

实现类似广播机制的功能，不需要知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播。

多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知。

1. 中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

优点。

类之间各司其职，符合迪米特法则。

降低了对象之间的耦合性，使得对象易于独立地被复用。

将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。

缺点：

中介者模式将原本多个对象直接的相互依赖变成了中介者和多个同事类的依赖关系。当同事类越多时，中介者就会越臃肿，变得复杂且难以维护。

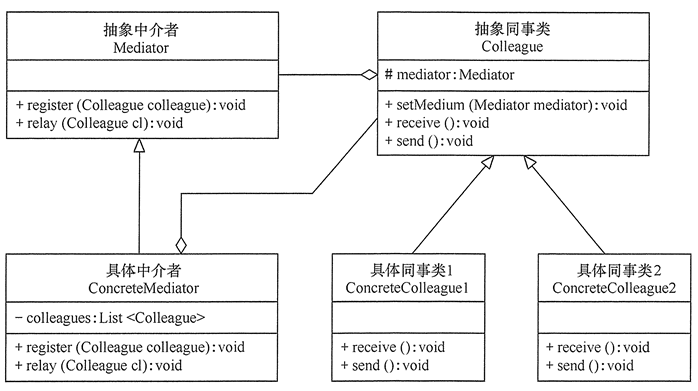
中介者模式包含以下主要角色。

抽象中介者（Mediator）角色：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。

具体中介者（Concrete Mediator）角色：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。

抽象同事类（Colleague）角色：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。

具体同事类（Concrete Colleague）角色：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。



应用场景：

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。

1. 迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

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。

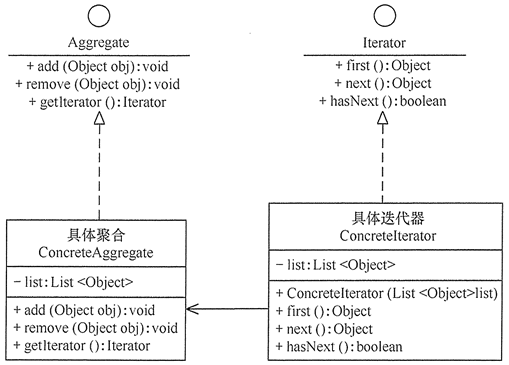
迭代器模式主要包含以下角色。

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

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

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

具体迭代器（Concretelterator）角色：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。



应用场景

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。

1. 访问者（Visitor）模式：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

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

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。

访问者模式包含以下主要角色。

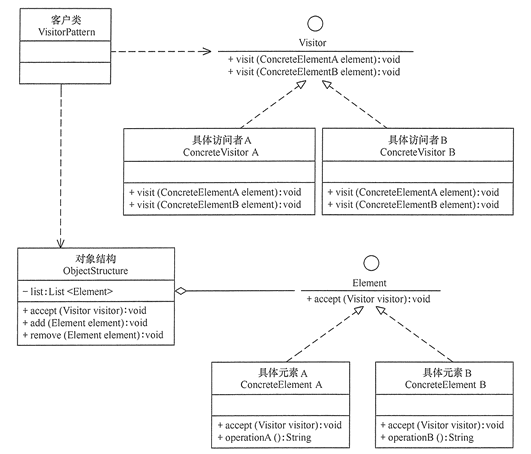
抽象访问者（Visitor）角色：定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit() ，该操作中的参数类型标识了被访问的具体元素。

具体访问者（ConcreteVisitor）角色：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。

抽象元素（Element）角色：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。

具体元素（ConcreteElement）角色：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。

对象结构（Object Structure）角色：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。



通常在以下情况可以考虑使用访问者（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。

1. 备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

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：

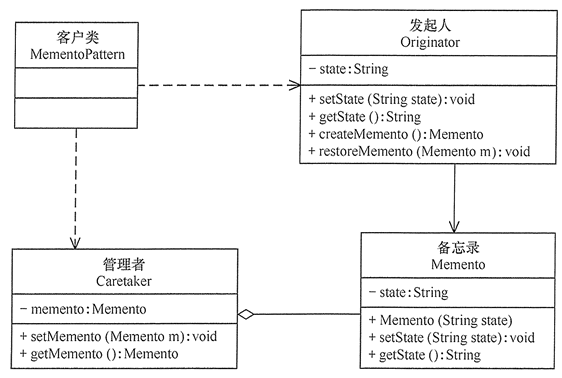
资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资 源。

备忘录模式的主要角色如下。

发起人（Originator）角色：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。

备忘录（Memento）角色：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。

管理者（Caretaker）角色：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。



该模式的以下应用场景。

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。

1. 解释器（Interpreter）模式：提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

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。

解释器模式包含以下主要角色。

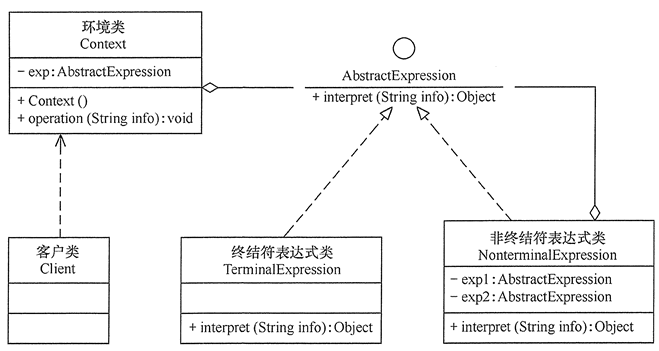
抽象表达式（Abstract Expression）角色：定义解释器的接口，约定解释器的解释操作，主要包含解释方法 interpret()。

终结符表达式（Terminal Expression）角色：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。

非终结符表达式（Nonterminal Expression）角色：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。

环境（Context）角色：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。

客户端（Client）：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。



前面介绍了解释器模式的结构与特点，下面分析它的应用场景。

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。