# 1. 面向对象设计原则

## <1. > 开闭原则

### 1. 定义

开闭原则（Open Closed Principle，OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：**软件实体应当对扩展开放，对修改关闭**（Software entities should be open for extension，but closed for modification），这就是开闭原则的经典定义。

### 2. 含义

开闭原则的含义是：**当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。**

### 3.作用

开闭原则是面向对象程序设计的终极目标，它使软件实体拥有一定的适应性和灵活性的同时具备稳定性和延续性。具体来说，其作用如下。

1. 对软件测试的影响 软件遵守开闭原则的话，软件测试时只需要对扩展的代码进行测试就可以了，因为原有的测试代码仍然能够正常运行。
2. 可以提高代码的可复用性 粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。
3. 可以提高软件的可维护性 遵守开闭原则的软件，其稳定性高和延续性强，从而易于扩展和维护。

### 4.实现方法

可以通过“抽象约束、封装变化”来实现开闭原则，即通过接口或者抽象类为软件实体定义一个相对稳定的抽象层，而将相同的可变因素封装在相同的具体实现类中。 因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节可以从抽象派生来的实现类来进行扩展，当软件需要发生变化时，只需要根据需求重新派生一个实现类来扩展就可以了。

## <2. > 里氏替换原则

### 1.定义

里氏替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的里斯科夫（Liskov）女士在 1987 年的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》（Data Abstraction and Hierarchy）里提出来的，她提出：**继承必须确保超类所拥有的性质在子类中仍然成立**（Inheritance should ensure that any property proved about supertype objects also holds for subtype objects）。

### 2.含义

里氏替换原则主要阐述了有关继承的一些原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中蕴含的原理。里氏替换原是继承复用的基础，它反映了基类与子类之间的关系，是对开闭原则的补充，是对实现抽象化的具体步骤的规范。

## 3.作用

里氏替换原则的主要作用如下。

1. 里氏替换原则是实现开闭原则的重要方式之一。
2. 它克服了继承中重写父类造成的可复用性变差的缺点。
3. 它是动作正确性的保证。即类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

## 4.实现方法

里氏替换原则通俗来讲就是：**子类可以扩展父类的功能，但不能改变父类原有的功能。也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。** 如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。 如果程序违背了里氏替换原则，则继承类的对象在基类出现的地方会出现运行错误。这时其修正方法是：取消原来的继承关系，重新设计它们之间的关系。

# <3. >依赖倒置原则

## 1.定义

依赖倒置原则（Dependence Inversion Principle，DIP）是 Object Mentor 公司总裁罗伯特·马丁（Robert C.Martin）于 1996 年在 C++ Report 上发表的文章。 依赖倒置原则的原始定义为：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象（High level modules shouldnot depend upon low level modules.Both should depend upon abstractions.Abstractions should not depend upon details. Details should depend upon abstractions）。其核心思想是：要面向接口编程，不要面向实现编程。

## 2.含义

**依赖倒置原则是实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合。** 由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。 使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给它们的实现类去完成。

## 3.作用

依赖倒置原则的主要作用如下。

- 依赖倒置原则可以降低类间的耦合性。
- 依赖倒置原则可以提高系统的稳定性。
- 依赖倒置原则可以减少并行开发引起的风险。
- 依赖倒置原则可以提高代码的可读性和可维护性。

## 4.实现方法

**依赖倒置原则的目的是通过要面向接口的编程来降低类间的耦合性，**所以我们在实际编程中只要遵循以下4点，就能在项目中满足这个规则。

1. 每个类尽量提供接口或抽象类，或者两者都具备。
2. 变量的声明类型尽量是接口或者是抽象类。
3. 任何类都不应该从具体类派生。
4. 使用继承时尽量遵循里氏替换原则。

# <4. >单一职责原则

## 1.定义

单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。这里的职责是指类变化的原因，**单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分**（There should never be more than one reason for a class to change）。

## 2.含义

该原则提出对象不应该承担太多职责，如果一个对象承担了太多的职责，至少存在以下两个缺点：

1. 一个职责的变化可能会削弱或者抑制这个类实现其他职责的能力；
2. 当客户端需要该对象的某一个职责时，不得不将其他不需要的职责全都包含进来，从而造成冗余代码或代码的浪费。

## 3.作用(优点)

单一职责原则的核心就是控制类的粒度大小、将对象解耦、提高其内聚性。如果遵循单一职责原则将有以下优点。

- 降低类的复杂度。一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多。
- 提高类的可读性。复杂性降低，自然其可读性会提高。
- 提高系统的可维护性。可读性提高，那自然更容易维护了。
- 变更引起的风险降低。变更是必然的，如果单一职责原则遵守得好，当修改一个功能时，可以显著降低对其他功能的影响。

## 4.实现方法

单一职责原则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，再封装到不同的类或模块中。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重构经验。

**单一职责同样也适用于方法。一个方法应该尽可能做好一件事情。如果一个方法处理的事情太多，其颗粒度会变得很粗，不利于重用。**

# <5.>接口隔离原则

## 1.定义

接口隔离原则（Interface Segregation Principle，ISP）**要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。**

2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：**客户端不应该被迫依赖于它不使用的方法**（Clients should not be forced to depend on methods they do not use）。**该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上**（The dependency of one class to another one should depend on the smallest possible interface）。

## 2.含义

以上两个定义的含义是：**要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。**

**接口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想**，但两者是不同的：

- **单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离。**
- **单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。**

## 3.作用（优点）

**接口隔离原则是为了约束接口、降低类对接口的依赖性，**遵循接口隔离原则有以下 5 个优点。

1. 将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。
2. 接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。
3. 如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。
4. 使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。
5. 能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

## 4.实现方式

在具体应用接口隔离原则时，应该根据以下几个规则来衡量。

- 接口尽量小，但是要有限度。一个接口只服务于一个子模块或业务逻辑。
- 为依赖接口的类定制服务。只提供调用者需要的方法，屏蔽不需要的方法。
- 了解环境，拒绝盲从。每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。
- 提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。

# <6. >迪米特法则

## 1. 定义

**迪米特法则（Law of Demeter，LoD）又叫作最少知识原则**（Least Knowledge Principle，LKP），产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

## 2.含义

迪米特法则的定义是：**只与你的直接朋友交谈，不跟“陌生人”说话**（Talk only to your immediate friends and not to strangers）。其含义是：**如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。**

注：迪米特法则中的“朋友”是指：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。

## 3.作用（优点）

**迪米特法则要求限制软件实体之间通信的宽度和深度**，正确使用迪米特法则将有以下两个优点。

1. **降低了类之间的耦合度，提高了模块的相对独立性。**
2. **由于亲合度降低，从而提高了类的可复用率和系统的扩展性。**

但是，**过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。**所以，**在采用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。**

## 4.实现方式

从迪米特法则的定义和特点可知，它强调以下两点：

1. 从依赖者的角度来说，只依赖应该依赖的对象。
2. 从被依赖者的角度说，只暴露应该暴露的方法。

所以，在运用迪米特法则时要注意以下 6 点。

1. 在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标。
2. 在类的结构设计上，尽量降低类成员的访问权限。
3. 在类的设计上，优先考虑将一个类设置成不变类。
4. 在对其他类的引用上，将引用其他对象的次数降到最低。
5. 不暴露类的属性成员，而应该提供相应的访问器（set 和 get 方法）。
6. 谨慎使用序列化（Serializable）功能。

# <7.>合成复用原则

## 1.定义

合成复用原则（Composite Reuse Principle，CRP）**又叫组合/聚合复用原则**（Composition/Aggregate Reuse Principle，CARP）。**它要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。**

## 2.含义

如果要使用继承关系，则必须严格遵循里氏替换原则。合成复用原则同里氏替换原则相辅相成的，两者都是开闭原则的具体实现规范。

## 3.作用（优点）

**通常类的复用分为继承复用和合成复用两种**，继承复用虽然有简单和易实现的优点，但它也存在以下缺点。

1. **继承复用破坏了类的封装性。因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。**
2. **子类与父类的耦合度高。父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。**
3. **它限制了复用的灵活性。从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。**

采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点。

1. **它维持了类的封装性。因为成员对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。**

2. **新旧类之间的耦合度低。这种复用所需的依赖较少，新对象存取成员对象的唯一方法是通过成员对象的接口。**
3. **复用的灵活性高。这种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对象。**

### 4.实现方法

合成复用原则是通过将已有的对象纳入新对象中，作为新对象的成员对象来实现的，新对象可以调用已有对象的功能，从而达到复用。

## 1.1总结

结合前几节的内容，我们一共介绍了 7 种设计原则，它们分别为开闭原则、里氏替换原则、依赖倒置原则、单一职责原则、接口隔离原则、迪米特法则和本节所介绍的合成复用原则。

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。

# 2.模式的特点和分类

## 1.创建型模式

**创建型模式的主要关注点是“怎样创建对象？”**，它的主要特点是“**将对象的创建与使用分离**”。这样可以降低系统的耦合度，使用者不需要关注对象的创建细节，对象的创建由相关的工厂来完成。就像我们去商场购买商品时，不需要知道商品是怎么生产出来一样，因为它们由专门的厂商生产。

```
创建型模式分为以下几种。
    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。
    2.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。
    3.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。
    4.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。
    5.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。
  以上 5 种创建型模式，除了工厂方法模式属于类创建型模式，其他的全部属于对象创建型模式
```

## 2.结构型模式

结构型模式描述如何将类或对象按某种布局组成更大的结构。它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，后者采用组合或聚合来组合对象。

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”，所以对象结构型模式比类结构型模式具有更大的灵活性。

结构型模式分为以下 7 种：

```
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。
2. 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。
3. 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。
4. 装饰（Decorator）模式：动态地给对象增加一些职责，即增加其额外的功能。
5. 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。
6. 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。
7. 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。
```

以上 7 种结构型模式，除了适配器模式分为类结构型模式和对象结构型模式两种，其他的全部属于对象结构型模式。

# 3.行为型模式

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。

```
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。
2.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。
3.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。
4.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。
5.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。
6.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。
7.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。
8.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。
9.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。
10.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。
11.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。
```

以上 11 种行为型模式，除了模板方法模式和解释器模式是类行为型模式，其他的全部属于对象行为型模式。

# 3. 23种设计模式的功能

```
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。
```

```
2. 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。
3. 工厂方法（Factory Method）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。
4. 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。
5. 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。
6. 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。
7. 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。
8. 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，从而降低　　了抽象和实现这两个可变维度的耦合度。
9. 装饰（Decorator）模式：动态的给对象增加一些职责，即增加其额外的功能。
10. 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。
11. 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。
12. 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。
13. 模板方法（TemplateMethod）模式：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。
14. 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。
15. 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。
16. 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。
17. 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。
18. 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。
19. 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。
20. 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。
21. 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。
22. 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。
23. 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。
```

# 3.1单例模式

## 1.定义

单例（Singleton）模式的定义：**指一个类只有一个实例，且该类能自行创建这个实例的一种模式。**例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，或出现各个窗口显示内容的不一致等错误。

## 2.特点

1. 单例类只有一个实例对象；
2. 该单例对象必须由单例类自行创建；
3. 单例类对外提供一个访问该单例的全局访问点；

## 3.结构

单例模式的主要角色如下。

- 单例类：包含一个实例且能自行创建这个实例的类。
- 访问类：使用单例的类。

## 4.实现方法

Singleton 模式通常有两种实现形式。

### 4.1懒汉式单例

该模式的特点是类加载时没有生成单例，只有当第一次调用 getInstance 方法时才去创建这个单例。代码如下：

```
public class SingletonLazy
{
    public static void main(String[] args)
    {
        President zt1=President.getInstance();
        zt1.getName();    //输出总统的名字
        President zt2=President.getInstance();
        zt2.getName();    //输出总统的名字
        if(zt1==zt2)
        {
           System.out.println("他们是同一人！");
        }
        else
        {
           System.out.println("他们不是同一人！");
        }
    }
}
class President
{
    private static volatile President instance=null;    //保证instance在所有线程中同步
    //private避免类在外部被实例化
    private President()
    {
        System.out.println("产生一个总统！");
    }
    public static synchronized President getInstance()
    {
        //在getInstance方法上加同步
        if(instance==null)
        {
               instance=new President();
        }
        else
        {
           System.out.println("已经有一个总统，不能产生新总统！");
        }
        return instance;
    }
    public void getName()
```

```
    {
        System.out.println("我是美国总统：特朗普。");
    }
}
```

```
产生一个总统!
我是美国总统：特朗普。
已经有一个总统，不能产生新总统!
我是美国总统：特朗普。
他们是同一人!
```

**注意：如果编写的是多线程程序，则不要删除上例代码中的关键字 volatile 和 synchronized，否则将存在线程非安全的问题。如果不删除这两个关键字就能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源，这是懒汉式单例的缺点。**

## 4.2饿汉式单例

该模式的特点是类一旦加载就创建一个单例，保证在调用 getInstance 方法之前单例已经存在了。

```
import java.awt.*;
import javax.swing.*;
public class SingletonEager
{
    public static void main(String[] args)
    {
        JFrame jf=new JFrame("饿汉单例模式测试");
        jf.setLayout(new GridLayout(1,2));
        Container contentPane=jf.getContentPane();
        Bajie obj1=Bajie.getInstance();
        contentPane.add(obj1);
        Bajie obj2=Bajie.getInstance();
        contentPane.add(obj2);
        if(obj1==obj2)
        {
            System.out.println("他们是同一人！");
        }
        else
        {
            System.out.println("他们不是同一人！");
        }
        jf.pack();
        jf.setVisible(true);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
class Bajie extends JPanel
{
    private static Bajie instance=new Bajie();
    private Bajie()
    {
        JLabel l1=new JLabel(new ImageIcon("src/Bajie.jpg"));
        this.add(l1);
```

```
    }
    public static Bajie getInstance()
    {
        return instance;
    }
}
```

**饿汉式单例在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以是线程安全的，可以直接用于多线程而不会出现问题。**

## 5.应用场景

前面分析了单例模式的结构与特点，以下是它通常适用的场景的特点。

- 在应用场景中，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。
- 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。
- 当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。

# 3.2原型模式

## 1.定义

原型（Prototype）模式的定义如下：**用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。**在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。例如，Windows 操作系统的安装通常较耗时，如果复制就快了很多。在生活中复制的例子非常多，这里不一一列举了。

## 2.结构

由于 Java 提供了对象的 clone() 方法，所以 用 Java 实现原型模式很简单。

1. 抽象原型类：规定了具体原型对象必须实现的接口。
2. 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。
3. 访问类：使用具体原型类中的 clone() 方法来复制新的对象。

## 3.实现

**原型模式的克隆分为浅克隆和深克隆，Java 中的 Object 类提供了浅克隆的 clone() 方法，具体原型类只要实现 Cloneable 接口就可实现对象的浅克隆，这里的 Cloneable 接口就是抽象原型类。**

```
public class ProtoTypeCitation
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        citation obj1=new citation("张三","同学：在2016学年第一学期中表现优秀，被评为三好学生。","韶关学院");
        obj1.display();
        citation obj2=(citation) obj1.clone();
        obj2.setName("李四");
        obj2.display();
```

```
    }
}
//奖状类
class citation implements Cloneable
{
    String name;
    String info;
    String college;
    citation(String name,String info,String college)
    {
        this.name=name;
        this.info=info;
        this.college=college;
        System.out.println("奖状创建成功！");
    }
    void setName(String name)
    {
        this.name=name;
    }
    String getName()
    {
        return(this.name);
    }
    void display()
    {
        System.out.println(name+info+college);
    }
    public Object clone() throws CloneNotSupportedException
    {
        System.out.println("奖状拷贝成功！");
        return (citation)super.clone();
    }
}
```

程序运行结果如下：

```
奖状创建成功！
张三同学：在2016学年第一学期中表现优秀，被评为三好学生。韶关学院
奖状拷贝成功！
李四同学：在2016学年第一学期中表现优秀，被评为三好学生。韶关学院
```

## 4.应用场景

原型模式通常适用于以下场景。

- 对象之间相同或相似，即只是个别的几个属性不同的时候。
- 对象的创建过程比较麻烦，但复制比较简单的时候。

# 3.3工厂方法模式

## 1.定义

工厂方法（FactoryMethod）模式的 定义：**定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。**

我们把被创建的对象称为“产品”，把创建产品的对象称为“工厂”。如果要创建的产品不多，只要一个工厂类就可以完成，这种模式叫“简单工厂模式”，它不属于 GoF 的 23 种经典设计模式，它的缺点是增加新产品时会背“开闭原则”。

本节介绍的“工厂方法模式”是对简单工厂模式的进一步抽象化，其好处是可以使系统在不修改原来代码的情况下引进新的产品，即满足开闭原则。

## 2.作用（优点）

工厂方法模式的主要优点有：

- 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；
- 在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则；

其缺点是：每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

## 3.结构

**工厂方法模式由抽象工厂、具体工厂、抽象产品和具体产品等4个要素构成。**本节来分析其基本结构和实现方法。

工厂方法模式的主要角色如下。

1. 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。
2. 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。
3. 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。
    1. 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。

## 4.应用场景

工厂方法模式通常适用于以下场景。

- 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。

# 3.4抽象工厂模式

## 1.定义

抽象工厂（AbstractFactory）模式的定义：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。**抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族。**

## 2.条件

使用抽象工厂模式一般要满足以下条件。

- 系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品。
- 系统一次只可能消费其中某一族产品，即同族的产品一起使用。

## 3.作用（优点）

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。
- 当增加一个新的产品族时不需要修改原代码，满足开闭原则。

其缺点是：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。

## 4.结构

抽象工厂模式同工厂方法模式一样，**也是由抽象工厂、具体工厂、抽象产品和具体产品等 4 个要素构成，**但抽象工厂中方法个数不同，抽象产品的个数也不同。现在我们来分析其基本结构和实现方法。

抽象工厂模式的主要角色如下。

1. 抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含多个创建产品的方法 newProduct()，可以创建多个不同等级的产品。
2. 具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。
3. 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，抽象工厂模式有多个抽象产品。
4. 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 同具体工厂之间是多对一的关系。

## 5.应用场景

抽象工厂模式通常适用于以下场景：

1. 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。
2. 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。
3. 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。

## 6.扩展

抽象工厂模式的扩展有一定的“开闭原则”倾斜性：

1. **当增加一个新的产品族时只需增加一个新的具体工厂，不需要修改原代码，满足开闭原则。**
2. **当产品族中需要增加一个新种类的产品时，则所有的工厂类都需要进行修改，不满足开闭原则。**

另一方面，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。

# 3.5建造者模式

## 1.定义

建造者（Builder）模式的定义：**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，这样的设计模式被称为建造者模式。**它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

## 2.优缺点

该模式的主要优点如下：

1. 各个具体的建造者相互独立，有利于系统的扩展。
2. 客户端不必知道产品内部组成的细节，便于控制细节风险。

其缺点如下：

1. 产品的组成部分必须相同，这限制了其使用范围。
2. 如果产品的内部变化复杂，该模式会增加很多的建造者类。

**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，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。**

**建造者（Builder）模式由产品、抽象建造者、具体建造者、指挥者等 4 个要素构成**，现在我们来分析其基本结构和实现方法。

## 3.结构

建造者（Builder）模式的主要角色如下。

1. 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滅部件。
2. 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()。
3. 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。
4. 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。

## 4.应用场景

建造者（Builder）模式创建的是复杂对象，其产品的各个部分经常面临着剧烈的变化，但将它们组合在一起的算法却相对稳定，所以它通常在以下场合使用。

- 创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的。
- 创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

建造者（Builder）模式在应用过程中可以根据需要改变，如果创建的产品种类只有一种，只需要一个具体建造者，这时可以省略掉抽象建造者，甚至可以省略掉指挥者角色。

# 3.6代理模式

## 1.定义

代理模式的定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

## 2.优缺点

代理模式的主要优点有：

- 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；

其主要缺点是：

- 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；

## 3.结构

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问。

代理模式的主要角色如下。

1. 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。
2. 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。
3. 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。

## 4.应用场景

前面分析了代理模式的结构与特点，现在来分析以下的应用场景。

- 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。

# 3.7适配器模式

## 1.定义

适配器模式（Adapter）的定义如下：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。**

## 2.优缺点

该模式的主要优点如下。

- 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。

其缺点是：对类适配器来说，更换适配器的实现过程比较复杂。

## 3.结构

**类适配器模式可采用多重继承方式实现**，如 C++可定义一个适配器类来同时继承当前系统的业务接口和现有组件库中已经存在的组件接口；**Java不支持多继承，但可以定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件。**

对象适配器模式可采用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。现在来介绍它们的基本结构。

适配器模式（Adapter）包含以下主要角色。

1. 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。
2. 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。
3. 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。

## 4.实现

```
package adapter;
//目标接口
interface Target
{
    public void request();
}
//适配者接口
class Adaptee
{
    public void specificRequest()
    {
        System.out.println("适配者中的业务代码被调用！");
    }
}
//类适配器类
class ClassAdapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest();
    }
}
//客户端代码
public class ClassAdapterTest
{
    public static void main(String[] args)
    {
```

```
        System.out.println("类适配器模式测试：");
        Target target = new ClassAdapter();
        target.request();
    }
}

程序的运行结果如下：
类适配器模式测试：
适配者中的业务代码被调用！
```

## 5.应用场景

适配器模式（Adapter）通常适用于以下场景。

- 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。

# 3.8桥接模式

## 1.定义

桥接（Bridge）模式的定义如下：**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，从而降低了抽象和实现这两个可变维度的耦合度。**

## 2.优缺点

桥接（Bridge）模式的优点是：

- 由于抽象与实现分离，所以扩展能力强；
- 其实现细节对客户透明。

缺点是：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，这增加了系统的理解与设计难度。

## 3.结构

可以将抽象化部分与实现化部分分开，取消二者的继承关系，改用组合关系。

桥接（Bridge）模式包含以下主要角色。

1. 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。
2. 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。
3. 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。
4. 具体实现化（Concrete Implementor）角色：给出实现化角色接口的具体实现。

## 4.实现

```
package bridge;
public class BridgeTest
{
    public static void main(String[] args)
    {
```

```
        Implementor imple=new ConcreteImplementorA();
        Abstraction abs=new RefinedAbstraction(imple);
        abs.Operation();
    }
}
//实现化角色
interface Implementor
{
    public void OperationImpl();
}
//具体实现化角色
class ConcreteImplementorA implements Implementor
{
    public void OperationImpl()
    {
        System.out.println("具体实现化(Concrete Implementor)角色被访问" );
    }
}
//抽象化角色
abstract class Abstraction
{
   protected Implementor imple;
   protected Abstraction(Implementor imple)
   {
       this.imple=imple;
   }
   public abstract void Operation();
}
//扩展抽象化角色
class RefinedAbstraction extends Abstraction
{
   protected RefinedAbstraction(Implementor imple)
   {
       super(imple);
   }
   public void Operation()
   {
       System.out.println("扩展抽象化(Refined Abstraction)角色被访问" );
       imple.OperationImpl();
   }
}

程序的运行结果如下:

扩展抽象化(Refined Abstraction)角色被访问
具体实现化(Concrete Implementor)角色被访问
```

## 5.应用场景

桥接模式通常适用于以下场景。

1. 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。
2. 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。

3. 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。

# 3.9装饰模式

## 1.定义

装饰（Decorator）模式的定义：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。

## 2.优缺点

装饰（Decorator）模式的主要优点有：

- 采用装饰模式扩展对象的功能比采用继承方式更加灵活。
- 可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。

其主要缺点是：装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

## 3.结构

通常情况下，扩展一个类的功能会使用继承方式来实现。但继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀。如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。下面来分析其基本结构和实现方法。

装饰模式主要包含以下角色。

1. 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。
2. 具体构件（Concrete Component）角色：实现抽象构件，通过装饰角色为其添加一些职责。
3. 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。
4. 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。

## 4.实现

```
package decorator;
public class DecoratorPattern
{
    public static void main(String[] args)
    {
        Component p=new ConcreteComponent();
        p.operation();
        System.out.println("---------------------------------");
        Component d=new ConcreteDecorator(p);
        d.operation();
    }
}
//抽象构件角色
interface  Component
{
    public void operation();
}
//具体构件角色
```

```
class ConcreteComponent implements Component
{
    public ConcreteComponent()
    {
        System.out.println("创建具体构件角色");
    }
    public void operation()
    {
        System.out.println("调用具体构件角色的方法operation()");
    }
}
//抽象装饰角色
class Decorator implements Component
{
    private Component component;
    public Decorator(Component component)
    {
        this.component=component;
    }
    public void operation()
    {
        component.operation();
    }
}
//具体装饰角色
class ConcreteDecorator extends Decorator
{
    public ConcreteDecorator(Component component)
    {
        super(component);
    }
    public void operation()
    {
        super.operation();
        addedFunction();
    }
    public void addedFunction()
    {
        System.out.println("为具体构件角色增加额外的功能addedFunction()");
    }
}

程序运行结果如下:
创建具体构件角色
调用具体构件角色的方法operation()
---------------------------------
调用具体构件角色的方法operation()
为具体构件角色增加额外的功能addedFunction()

```

## 5.应用场景

前面讲解了关于装饰模式的结构与特点，下面介绍其适用的应用场景，装饰模式通常在以下几种情况使用。

- 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。

装饰模式在 Java语言中的最著名的应用莫过于 Java I/O 标准库的设计了。例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

下面代码是为 FileReader 增加缓冲区而采用的装饰类 BufferedReader 的例子：

```
BufferedReader in=new BufferedReader(new FileReader("filename.txtn));
String s=in.readLine();
```

# 3.10 外观模式

## 1.定义

外观（Facade）模式的定义：**是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。**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的细节，这样会大大降低应用程序的复杂度，提高了程序的可维护性。

## 2.优缺点

**外观（Facade）模式是“迪米特法则”的典型应用**，它有以下主要优点。

1. 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。
2. 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。
3. 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。

外观（Facade）模式的主要缺点如下。

1. 不能很好地限制客户使用子系统类。
2. 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”。

## 3.结构

外观（Facade）模式的结构比较简单，主要是定义了一个高层接口。它包含了对各个子系统的引用，客户端可以通过它访问各个子系统的功能。现在来分析其基本结构和实现方法。

外观（Facade）模式包含以下主要角色。

1. 外观（Facade）角色：为多个子系统对外提供一个共同的接口。
2. 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。
3. 客户（Client）角色：通过一个外观角色访问各个子系统的功能。

## 4.实现

```
package facade;
public class FacadePattern
```

```
{
    public static void main(String[] args)
    {
        Facade f=new Facade();
        f.method();
    }
}
//外观角色
class Facade
{
    private SubSystem01 obj1=new SubSystem01();
    private SubSystem02 obj2=new SubSystem02();
    private SubSystem03 obj3=new SubSystem03();
    public void method()
    {
        obj1.method1();
        obj2.method2();
        obj3.method3();
    }
}
//子系统角色
class SubSystem01
{
    public  void method1()
    {
        System.out.println("子系统01的method1()被调用! ");
    }
}
//子系统角色
class SubSystem02
{
    public  void method2()
    {
        System.out.println("子系统02的method2()被调用! ");
    }
}
//子系统角色
class SubSystem03
{
    public  void method3()
    {
        System.out.println("子系统03的method3()被调用! ");
    }
}

程序运行结果如下:
子系统01的method1()被调用!
子系统02的method2()被调用!
子系统03的method3()被调用!
```

## 5.应用场景

通常在以下情况下可以考虑使用外观模式。

1. 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。
2. 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。
3. 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。

# 3.11享元模式

## 1.定义

享元（Flyweight）模式的定义：运用共享技术来有效地支持大量细粒度对象的复用。它通过共享已经存在的又橡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。

## 2.优缺点

享元模式的主要优点是：**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。**

其主要缺点是：

1. 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。
2. 读取享元模式的外部状态会使得运行时间稍微变长。

## 3.结构

享元模式中存在以下两种状态：

1. **内部状态，即不会随着环境的改变而改变的可共享部分；**
2. **外部状态，指随环境改变而改变的不可以共享的部分。享元模式的实现要领就是区分应用中的这两种状态，并将外部状态外部化。下面来分析其基本结构和实现方法。**

享元模式的主要角色有如下。

1. 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。
2. 具体享元（Concrete Flyweight）角色：实现抽象享元角色中所规定的接口。
3. 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。
4. 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，享元工厂检查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。

## 4.实现

```
package flyweight;
import java.util.HashMap;
public class FlyweightPattern
{
    public static void main(String[] args)
    {
        FlyweightFactory factory=new FlyweightFactory();
        Flyweight f01=factory.getFlyweight("a");
        Flyweight f02=factory.getFlyweight("a");
```

```
        Flyweight f03=factory.getFlyweight("a");
        Flyweight f11=factory.getFlyweight("b");
        Flyweight f12=factory.getFlyweight("b");
        f01.operation(new UnsharedConcreteFlyweight("第1次调用a。"));
        f02.operation(new UnsharedConcreteFlyweight("第2次调用a。"));
        f03.operation(new UnsharedConcreteFlyweight("第3次调用a。"));
        f11.operation(new UnsharedConcreteFlyweight("第1次调用b。"));
        f12.operation(new UnsharedConcreteFlyweight("第2次调用b。"));
    }
}
//非享元角色
class UnsharedConcreteFlyweight
{
    private String info;
    UnsharedConcreteFlyweight(String info)
    {
        this.info=info;
    }
    public String getInfo()
    {
        return info;
    }
    public void setInfo(String info)
    {
        this.info=info;
    }
}
//抽象享元角色
interface Flyweight
{
    public void operation(UnsharedConcreteFlyweight state);
}
//具体享元角色
class ConcreteFlyweight implements Flyweight
{
    private String key;
    ConcreteFlyweight(String key)
    {
        this.key=key;
        System.out.println("具体享元"+key+"被创建！");
    }
    public void operation(UnsharedConcreteFlyweight outState)
    {
        System.out.print("具体享元"+key+"被调用，");
        System.out.println("非享元信息是:"+outState.getInfo());
    }
}
//享元工厂角色
class FlyweightFactory
{
    private HashMap<String, Flyweight> flyweights=new HashMap<String, Flyweight>();
    public Flyweight getFlyweight(String key)
    {
```

```
        Flyweight flyweight=(Flyweight)flyweights.get(key);
        if(flyweight!=null)
        {
            System.out.println("具体享元"+key+"已经存在，被成功获取！");
        }
        else
        {
            flyweight=new ConcreteFlyweight(key);
            flyweights.put(key, flyweight);
        }
        return flyweight;
    }
}
```

程序运行结果如下：

```
具体享元a被创建!
具体享元a已经存在，被成功获取!
具体享元a已经存在，被成功获取!
具体享元b被创建!
具体享元b已经存在，被成功获取!
具体享元a被调用，非享元信息是:第1次调用a。
具体享元a被调用，非享元信息是:第2次调用a。
具体享元a被调用，非享元信息是:第3次调用a。
具体享元b被调用，非享元信息是:第1次调用b。
具体享元b被调用，非享元信息是:第2次调用b。
```

## 5.应用场景

前面分析了享元模式的结构与特点，下面分析它适用的应用场景。享元模式是通过减少内存中对象的数量来节省内存空间的，所以以下几种情形适合采用享元模式。

1. 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。
2. 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。
3. 由于享元模式需要额外维护一个保存享元的数据结构，所以应当在有足够多的享元实例时才值得使用享元模式。

在实际使用过程中，有时候会稍加改变，即存在两种特殊的享元模式：单纯享元模式和复合享元模式：

(1) 单纯享元模式，这种享元模式中的所有的具体享元类都是可以共享的，不存在非共享的具体享元类。

(2) 复合享元模式，这种享元模式中的有些享元对象是由一些单纯享元对象组合而成的，它们就是复合享元对象。虽然复合享元对象本身不能共享，但它们可以分解成单纯享元对象再被共享。

# 3.12组合模式

## 1.定义

组合（Composite）模式的定义：有时又叫作部分-整体模式，它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整体”的关系，使用户对单个对象和组合对象具有一致的访问性。

## 2.优缺点

组合模式的主要优点有：

1. 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；
2. 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”；

其主要缺点是：

1. 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；
2. 不容易限制容器中的构件；
3. 不容易用继承的方法来增加构件的新功能；

## 3.结构

组合模式包含以下主要角色。

1. 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。
2. 树叶构件（Leaf）角色：是组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中 声明的公共接口。
3. 树枝构件（Composite）角色：是组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明的接口，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。

组合模式分为透明式的组合模式和安全式的组合模式。

(1) 透明方式：在该方式中，由于抽象构件声明了所有子类中的全部方法，所以客户端无须区别树叶对象和树枝对象，对客户端来说是透明的。但其缺点是：树叶构件本来没有 Add()、Remove() 及 GetChild() 方法，却要实现它们（空实现或抛异常），这样会带来一些安全性问题。

(2) 安全方式：在该方式中，将管理子构件的方法移到树枝构件中，抽象构件和树叶构件没有对子对象的管理方法，这样就避免了上一种方式的安全性问题，但由于叶子和分支有不同的接口，客户端在调用时要知道树叶对象和树枝对象的存在，所以失去了透明性。

## 4.实现

```
package composite;
import java.util.ArrayList;
public class CompositePattern
{
    public static void main(String[] args)
    {
        Component c0=new Composite();
        Component c1=new Composite();
        Component leaf1=new Leaf("1");
        Component leaf2=new Leaf("2");
        Component leaf3=new Leaf("3");
        c0.add(leaf1);
        c0.add(c1);
        c1.add(leaf2);
        c1.add(leaf3);
        c0.operation();
    }
```

```
}
//抽象构件
interface Component
{
    public void add(Component c);
    public void remove(Component c);
    public Component getChild(int i);
    public void operation();
}
//树叶构件
class Leaf implements Component
{
    private String name;
    public Leaf(String name)
    {
        this.name=name;
    }
    public void add(Component c){ }
    public void remove(Component c){ }
    public Component getChild(int i)
    {
        return null;
    }
    public void operation()
    {
        System.out.println("树叶"+name+": 被访问! ");
    }
}
//树枝构件
class Composite implements Component
{
    private ArrayList<Component> children=new ArrayList<Component>();
    public void add(Component c)
    {
        children.add(c);
    }
    public void remove(Component c)
    {
        children.remove(c);
    }
    public Component getChild(int i)
    {
        return children.get(i);
    }
    public void operation()
    {
        for(Object obj:children)
        {
            ((Component)obj).operation();
        }
    }
}

```

```
程序运行结果如下：
树叶1：被访问！
树叶2：被访问！
树叶3：被访问！
```

## 5.应用场景

前面分析了组合模式的结构与特点，下面分析它适用的以下应用场景。

1. 在需要表示一个对象整体与部分的层次结构的场合。
2. 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。

# 3.13模板方法

## 1.定义

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。

## 2.优缺点

该模式的主要优点如下。

1. 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。
2. 它在父类中提取了公共的部分代码，便于代码复用。
3. 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。

该模式的主要缺点如下。

1. 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。
2. 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。

## 3.结构

**模板方法模式需要注意抽象类与具体子类之间的协作。它用到了虚函数的多态性技术以及“不用调用我，让我来调用你”的反向控制技术。**现在来介绍它们的基本结构。

模板方法模式包含以下主要角色。

(1) 抽象类（Abstract Class）：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：在抽象类中申明，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（Concrete Class）：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。

## 4.实现

```
package templateMethod;
public class TemplateMethodPattern
{
    public static void main(String[] args)
    {
        AbstractClass tm=new ConcreteClass();
        tm.TemplateMethod();
    }
}
//抽象类
abstract class AbstractClass
{
    public void TemplateMethod() //模板方法
    {
        SpecificMethod();
        abstractMethod1();
         abstractMethod2();
    }
    public void SpecificMethod() //具体方法
    {
        System.out.println("抽象类中的具体方法被调用...");
    }
    public abstract void abstractMethod1(); //抽象方法1
    public abstract void abstractMethod2(); //抽象方法2
}
//具体子类
class ConcreteClass extends AbstractClass
{
    public void abstractMethod1()
    {
        System.out.println("抽象方法1的实现被调用...");
    }
    public void abstractMethod2()
    {
        System.out.println("抽象方法2的实现被调用...");
    }
}

程序的运行结果如下:
抽象类中的具体方法被调用...
抽象方法1的实现被调用...
抽象方法2的实现被调用...
```

## 5.应用场景

模板方法模式通常适用于以下场景。

1. 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。

2. 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。
3. 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。

# 3.14策略模式

## 1.定义

策略（Strategy）模式的定义：**该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。**

## 2.优缺点

策略模式的主要优点如下。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

其主要缺点如下。

1. 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。
2. 策略模式造成很多的策略类。

## 3.结构

策略模式是准备一组算法，并将这组算法封装到一系列的策略类里面，作为一个抽象策略类的子类。策略模式的重心不是如何实现算法，而是如何组织这些算法，从而让程序结构更加灵活，具有更好的维护性和扩展性。

策略模式的主要角色如下。

1. 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。
2. 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。
3. 环境（Context）类：持有一个策略类的引用，最终给客户端调用。

## 4.实现

```
package strategy;
public class StrategyPattern
{
    public static void main(String[] args)
    {
        Context c=new Context();
        Strategy s=new ConcreteStrategyA();
        c.setStrategy(s);
        c.strategyMethod();
```

```
        System.out.println("-----------------");
        s=new ConcreteStrategyB();
        c.setStrategy(s);
        c.strategyMethod();
    }
}
//抽象策略类
interface Strategy
{
    public void strategyMethod();    //策略方法
}
//具体策略类A
class ConcreteStrategyA implements Strategy
{
    public void strategyMethod()
    {
        System.out.println("具体策略A的策略方法被访问！");
    }
}
//具体策略类B
class ConcreteStrategyB implements Strategy
{
  public void strategyMethod()
  {
      System.out.println("具体策略B的策略方法被访问！");
  }
}
//环境类
class Context
{
    private Strategy strategy;
    public Strategy getStrategy()
    {
        return strategy;
    }
    public void setStrategy(Strategy strategy)
    {
        this.strategy=strategy;
    }
    public void strategyMethod()
    {
        strategy.strategyMethod();
    }
}
```

程序运行结果如下：

```
具体策略A的策略方法被访问！
-----------------
具体策略B的策略方法被访问！
```

## 5.应用场景

策略模式在很多地方用到，如 Java SE 中的容器布局管理就是一个典型的实例，Java SE 中的每个容器都存在多种布局供用户选择。在程序设计中，通常在以下几种情况中使用策略模式较多。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

# 3.15命令模式

## 1.定义

命令（Command）模式的定义如下：**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。**

## 2.优缺点

命令模式的主要优点如下。

1. 降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦。
2. 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，它满足“开闭原则”，对扩展比较灵活。
3. 可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令。
4. 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。

其缺点是：可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。

## 3.结构

可以将系统中的相关操作抽象成命令，使调用者与实现者相关分离，其结构如下。

命令模式包含以下主要角色。

1. 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()。
2. 具体命令角色（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。
3. 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。
4. 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。

## 4.实现

```
package command;
public class CommandPattern
{
    public static void main(String[] args)
    {
        Command cmd=new ConcreteCommand();
        Invoker ir=new Invoker(cmd);
```

```
        System.out.println("客户访问调用者的call()方法...");
        ir.call();
    }
}
//调用者
class Invoker
{
    private Command command;
    public Invoker(Command command)
    {
        this.command=command;
    }
    public void setCommand(Command command)
    {
        this.command=command;
    }
    public void call()
    {
        System.out.println("调用者执行命令command...");
        command.execute();
    }
}
//抽象命令
interface Command
{
    public abstract void execute();
}
//具体命令
class ConcreteCommand implements Command
{
    private Receiver receiver;
    ConcreteCommand()
    {
        receiver=new Receiver();
    }
    public void execute()
    {
        receiver.action();
    }
}
//接收者
class Receiver
{
    public void action()
    {
        System.out.println("接收者的action()方法被调用...");
    }
}

程序的运行结果如下:
客户访问调用者的call()方法...
调用者执行命令command...
接收者的action()方法被调用...
```

## 5.应用场景

命令模式通常适用于以下场景。

1. 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。
2. 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。
3. 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。
4. 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。

# 3.16责任链模式

## 1.定义

责任链（Chain of Responsibility）模式的定义：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。**责任链模式也叫职责链模式。**

在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，所以责任链将请求的发送者和请求的处理者解耦了。

## 2.优缺点

责任链模式是一种对象行为型模式，其主要优点如下。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

其主要缺点如下。

1. 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。
2. 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。
3. 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。

## 3.结构

通常情况下，可以通过数据链表来实现职责链模式的数据结构。

职责链模式主要包含以下角色。

1. 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。
2. 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。

3. 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。

## 4.实现

```
package chainOfResponsibility;
public class ChainOfResponsibilityPattern
{
    public static void main(String[] args)
    {
        //组装责任链
        Handler handler1=new ConcreteHandler1();
        Handler handler2=new ConcreteHandler2();
        handler1.setNext(handler2);
        //提交请求
        handler1.handleRequest("two");
    }
}
//抽象处理者角色
abstract class Handler
{
    private Handler next;
    public void setNext(Handler next)
    {
        this.next=next;
    }
    public Handler getNext()
    {
        return next;
    }
    //处理请求的方法
    public abstract void handleRequest(String request);
}
//具体处理者角色1
class ConcreteHandler1 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("one"))
        {
            System.out.println("具体处理者1负责处理该请求！");
        }
        else
        {
            if(getNext()!=null)
            {
                getNext().handleRequest(request);
            }
            else
            {
                System.out.println("没有人处理该请求！");
            }
        }
```

```
    }
}
//具体处理者角色2
class ConcreteHandler2 extends Handler
{
    public void handleRequest(String request)
    {
        if(request.equals("two"))
        {
            System.out.println("具体处理者2负责处理该请求！");
        }
        else
        {
            if(getNext()!=null)
            {
                getNext().handleRequest(request);
            }
            else
            {
                System.out.println("没有人处理该请求！");
            }
        }
    }
}
程序运行结果如下：
具体处理者2负责处理该请求！
```

## 5.应用场景

前边已经讲述了关于责任链模式的结构与特点，下面介绍其应用场景，责任链模式通常在以下几种情况使用。

1. 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。
2. 可动态指定一组对象处理请求，或添加新的处理者。
3. 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。

## 6.扩展

职责链模式存在以下两种情况。

1. 纯的职责链模式：一个请求必须被某一个处理者对象所接收，且一个具体处理者对某个请求的处理只能采用以下两种行为之一：自己处理（承担责任）；把责任推给下家处理。
2. 不纯的职责链模式：允许出现某一个具体处理者对象在承担了请求的一部分责任后又将剩余的责任传给下家的情况，且一个请求可以最终不被任何接收端对象所接收。

# 3.17状态模式

状态模式的解决思想是：**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。**

## 1.定义

状态（State）模式的定义：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。

## 2.优缺点

状态模式是一种对象行为型模式，其主要优点如下。

1. 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”。
2. 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。
3. 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。

状态模式的主要缺点如下。

1. 状态模式的使用必然会增加系统的类与对象的个数。
2. 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。

## 3.结构

**状态模式把受环境改变的对象行为包装在不同的状态对象里，其意图是让一个对象在其内部状态改变的时候，其行为也随之改变。**现在我们来分析其基本结构和实现方法。

状态模式包含以下主要角色。

1. 环境（Context）角色：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。
2. 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。
3. 具体状态（Concrete State）角色：实现抽象状态所对应的行为。

## 4.实现

```
package state;
public class StatePatternClient
{
    public static void main(String[] args)
    {
        Context context=new Context();    //创建环境
        context.Handle();    //处理请求
        context.Handle();
        context.Handle();
        context.Handle();
    }
}
//环境类
class Context
{
    private State state;
    //定义环境类的初始状态
    public Context()
    {
        this.state=new ConcreteStateA();
    }
```

```
    //设置新状态
    public void setState(State state)
    {
        this.state=state;
    }
    //读取状态
    public State getState()
    {
        return(state);
    }
    //对请求做处理
    public void Handle()
    {
        state.Handle(this);
    }
}
//抽象状态类
abstract class State
{
    public abstract void Handle(Context context);
}
//具体状态A类
class ConcreteStateA extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 A.");
        context.setState(new ConcreteStateB());
    }
}
//具体状态B类
class ConcreteStateB extends State
{
    public void Handle(Context context)
    {
        System.out.println("当前状态是 B.");
        context.setState(new ConcreteStateA());
    }
}

程序运行结果如下:

当前状态是 A.
当前状态是 B.
当前状态是 A.
当前状态是 B.
```

## 5.应用场景

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。

# 3.18观察者模式

## 1.定义

观察者（Observer）模式的定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

## 2.优缺点

观察者模式是一种对象行为型模式，其主要优点如下。

1. 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。
2. 目标与观察者之间建立了一套触发机制。

它的主要缺点如下。

1. 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。
2. 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。

## 3.结构

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。

观察者模式的主要角色如下。

1. 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。
2. 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。
3. 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。
4. 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。

## 4.实现

```
package observer;
import java.util.*;
public class ObserverPattern
{
    public static void main(String[] args)
    {
        Subject subject=new ConcreteSubject();
        Observer obs1=new ConcreteObserver1();
        Observer obs2=new ConcreteObserver2();
        subject.add(obs1);
        subject.add(obs2);
        subject.notifyObserver();
```

```
    }
}
//抽象目标
abstract class Subject
{
    protected List<Observer> observers=new ArrayList<Observer>();
    //增加观察者方法
    public void add(Observer observer)
    {
        observers.add(observer);
    }
    //删除观察者方法
    public void remove(Observer observer)
    {
        observers.remove(observer);
    }
    public abstract void notifyObserver(); //通知观察者方法
}
//具体目标
class ConcreteSubject extends Subject
{
    public void notifyObserver()
    {
        System.out.println("具体目标发生改变...");
        System.out.println("--------------");

        for(Object obs:observers)
        {
            ((Observer)obs).response();
        }

    }
}
//抽象观察者
interface Observer
{
    void response(); //反应
}
//具体观察者1
class ConcreteObserver1 implements Observer
{
    public void response()
    {
        System.out.println("具体观察者1作出反应! ");
    }
}
//具体观察者1
class ConcreteObserver2 implements Observer
{
    public void response()
    {
        System.out.println("具体观察者2作出反应! ");
    }
```

```
}

程序运行结果如下：

具体目标发生改变...
--------------
具体观察者1作出反应!
具体观察者2作出反应!
```

## 5.应用场景

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。

1. 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。
2. 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。

## 6.扩展

在 Java中，通过 java.util.Observable 类和 java.util.Observer 接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

### 1. Observable类

Observable 类是抽象目标类，它有一个 Vector 向量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法。

1. void addObserver(Observer o) 方法：用于将新的观察者对象添加到向量中。
2. void notifyObservers(Object arg) 方法：调用向量中的所有观察者对象的 update。方法，通知它们数据发生改变。通常越晚加入向量的观察者越先得到通知。
3. void setChange() 方法：用来设置一个 boolean 类型的内部标志位，注明目标对象发生了变化。当它为真时，notifyObservers() 才会通知观察者。

### 2. Observer 接口

Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用 void update(Observable o,Object arg) 方法，进行相应的工作。

# 3.19中介者模式

## 1.定义

中介者（Mediator）模式的定义：**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。**

## 2.优缺点

中介者模式是一种对象行为型模式，其主要优点如下。

1. 降低了对象之间的耦合性，使得对象易于独立地被复用。
2. 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。

其主要缺点是：当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

## 3.结构

中介者模式实现的关键是找出“中介者”，下面对它的结构和实现进行分析。

中介者模式包含以下主要角色。

1. 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。
2. 具体中介者（ConcreteMediator）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。
3. 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。
4. 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。

## 4.实现

```
package mediator;
import java.util.*;
public class MediatorPattern
{
    public static void main(String[] args)
    {
        Mediator md=new ConcreteMediator();
        Colleague c1,c2;
        c1=new ConcreteColleague1();
        c2=new ConcreteColleague2();
        md.register(c1);
        md.register(c2);
        c1.send();
        System.out.println("--------------");
        c2.send();
    }
}
//抽象中介者
abstract class Mediator
{
    public abstract void register(Colleague colleague);
    public abstract void relay(Colleague cl); //转发
}
//具体中介者
class ConcreteMediator extends Mediator
{
    private List<Colleague> colleagues=new ArrayList<Colleague>();
    public void register(Colleague colleague)
    {
        if(!colleagues.contains(colleague))
        {
            colleagues.add(colleague);
            colleague.setMedium(this);
        }
    }
    public void relay(Colleague cl)
    {
```

```
        for(Colleague ob:colleagues)
        {
            if(!ob.equals(cl))
            {
                ((Colleague)ob).receive();
            }
        }
    }
}
//抽象同事类
abstract class Colleague
{
    protected Mediator mediator;
    public void setMedium(Mediator mediator)
    {
        this.mediator=mediator;
    }
    public abstract void receive();
    public abstract void send();
}
//具体同事类
class ConcreteColleague1 extends Colleague
{
    public void receive()
    {
        System.out.println("具体同事类1收到请求。");
    }
    public void send()
    {
        System.out.println("具体同事类1发出请求。");
        mediator.relay(this); //请中介者转发
    }
}
//具体同事类
class ConcreteColleague2 extends Colleague
{
    public void receive()
    {
        System.out.println("具体同事类2收到请求。");
    }
    public void send()
    {
        System.out.println("具体同事类2发出请求。");
        mediator.relay(this); //请中介者转发
    }
}
```

程序的运行结果如下:

```
具体同事类1发出请求。
具体同事类2收到请求。
-------------
具体同事类2发出请求。
具体同事类1收到请求。
```

## 5.应用场景

前面分析了中介者模式的结构与特点，下面分析其以下应用场景。

- 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。

# 3.20迭代器模式

经常要访问一个聚合对象中的各个元素，如“数据结构”中的链表遍历，通常的做法是将链表的创建和遍历都放在同一个类中，但这种方式不利于程序的扩展，如果要更换遍历方法就必须修改程序源代码，这违背了 “开闭原则”。

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：

1. 暴露了聚合类的内部表示，使其数据不安全；
2. 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。

## 1.定义

迭代器（Iterator）模式的定义：提供一个对象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。

## 2.优缺点

迭代器模式是一种对象行为型模式，其主要优点如下。

1. 访问一个聚合对象的内容而无须暴露它的内部表示。
2. 遍历任务交由迭代器完成，这简化了聚合类。
3. 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。
4. 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
5. 封装性良好，为遍历不同的聚合结构提供一个统一的接口。

其主要缺点是：增加了类的个数，这在一定程度上增加了系统的复杂性。

## 3.结构

迭代器模式是通过将聚合对象的遍历行为分离出来，抽象成迭代器类来实现的，其目的是在不暴露聚合对象的内部结构的情况下，让外部代码透明地访问聚合的内部数据。现在我们来分析其基本结构与实现方法。

迭代器模式主要包含以下角色。

1. 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。
2. 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。
3. 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。
4. 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。

## 4.实现

```
package iterator;
import java.util.*;
public class IteratorPattern
{
    public static void main(String[] args)
    {
        Aggregate ag=new ConcreteAggregate();
        ag.add("中山大学");
        ag.add("华南理工");
        ag.add("韶关学院");
        System.out.print("聚合的内容有: ");
        Iterator it=ag.getIterator();
        while(it.hasNext())
        {
            Object ob=it.next();
            System.out.print(ob.toString()+"\t");
        }
        Object ob=it.first();
        System.out.println("\nFirst: "+ob.toString());
    }
}
//抽象聚合
interface Aggregate
{
    public void add(Object obj);
    public void remove(Object obj);
    public Iterator getIterator();
}
//具体聚合
class ConcreteAggregate implements Aggregate
{
    private List<Object> list=new ArrayList<Object>();
    public void add(Object obj)
    {
        list.add(obj);
    }
    public void remove(Object obj)
    {
        list.remove(obj);
    }
    public Iterator getIterator()
    {
        return(new ConcreteIterator(list));
    }
}
//抽象迭代器
interface Iterator
{
    Object first();
    Object next();
    boolean hasNext();
```

```
}
//具体迭代器
class ConcreteIterator implements Iterator
{
    private List<Object> list=null;
    private int index=-1;
    public ConcreteIterator(List<Object> list)
    {
        this.list=list;
    }
    public boolean hasNext()
    {
        if(index<list.size()-1)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public Object first()
    {
        index=0;
        Object obj=list.get(index);;
        return obj;
    }
    public Object next()
    {
        Object obj=null;
        if(this.hasNext())
        {
            obj=list.get(++index);
        }
        return obj;
    }
}

程序运行结果如下:
聚合的内容有:中山大学      华南理工      韶关学院
First:中山大学
```

## 5.应用场景

迭代器模式通常在以下几种情况使用。

1. 当需要为聚合对象提供多种遍历方式时。
2. 当需要为遍历不同的聚合结构提供一个统一的接口时。
3. 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。

由于聚合与迭代器的关系非常密切，所以大多数语言在实现聚合类时都提供了迭代器类，因此大数情况下使用语言中已有的聚合类的迭代器就已经够了。

# 3.21访问者模式

## 1.定义

访问者（Visitor）模式的定义：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。

## 2.优缺点

访问者(Visitor)模式是一种行为型模式，其主要优点如下：

1. 扩展性好。能够在不修改对象结构的元素情况下，为对象结构中的元素添加新的功能。
2. 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，从而提高系统的服用程度。
3. 灵活性好。访问者模式将数据结构与作用的数据结构上的操作解耦，使得操作集合可以相对的演化而不影响系统的数据结构。
4. 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，使得每一个访问者的功能都比较的单一。

访问者(Visitor)模式的主要缺点：

1. 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”。
2. 破坏封装。访问者模式中具体元素对访问者公布具体细节，这破坏了对象的封装性。
3. 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。

## 3.结构

访问者（Visitor）模式实现的关键是如何将作用于元素的操作分离出来封装成独立的类。

访问者模式包含以下主要角色。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

## 4.实现

```
1  package visitor;
2  import java.util.*;
3  public class VisitorPattern
4  {
5      public static void main(String[] args)
6      {
```

```
        ObjectStructure os=new ObjectStructure();
        os.add(new ConcreteElementA());
        os.add(new ConcreteElementB());
        Visitor visitor=new ConcreteVisitorA();
        os.accept(visitor);
        System.out.println("------------------------");
        visitor=new ConcreteVisitorB();
        os.accept(visitor);
    }
}
//抽象访问者
interface Visitor
{
    void visit(ConcreteElementA element);
    void visit(ConcreteElementB element);
}
//具体访问者A类
class ConcreteVisitorA implements Visitor
{
    public void visit(ConcreteElementA element)
    {
        System.out.println("具体访问者A访问-->"+element.operationA());
    }
    public void visit(ConcreteElementB element)
    {
        System.out.println("具体访问者A访问-->"+element.operationB());
    }
}
//具体访问者B类
class ConcreteVisitorB implements Visitor
{
    public void visit(ConcreteElementA element)
    {
        System.out.println("具体访问者B访问-->"+element.operationA());
    }
    public void visit(ConcreteElementB element)
    {
        System.out.println("具体访问者B访问-->"+element.operationB());
    }
}
//抽象元素类
interface Element
{
    void accept(Visitor visitor);
}
//具体元素A类
class ConcreteElementA implements Element
{
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
    public String operationA()
```

```
    {
        return "具体元素A的操作。";
    }
}
//具体元素B类
class ConcreteElementB implements Element
{
    public void accept(Visitor visitor)
    {
        visitor.visit(this);
    }
    public String operationB()
    {
        return "具体元素B的操作。";
    }
}
//对象结构角色
class ObjectStructure
{
    private List<Element> list=new ArrayList<Element>();
    public void accept(Visitor visitor)
    {
        Iterator<Element> i=list.iterator();
        while(i.hasNext())
        {
            ((Element) i.next()).accept(visitor);
        }
    }
    public void add(Element element)
    {
        list.add(element);
    }
    public void remove(Element element)
    {
        list.remove(element);
    }
}

程序的运行结果如下:
具体访问者A访问-->具体元素A的操作。
具体访问者A访问-->具体元素B的操作。
------------------------
具体访问者B访问-->具体元素A的操作。
具体访问者B访问-->具体元素B的操作。
```

## 5.应用场景

通常在以下情况可以考虑使用访问者（Visitor）模式。

1. 对象结构相对稳定，但其操作算法经常变化的程序。
2. 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。
3. 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。

# 3.22备忘录模式

## 1.定义

备忘录（Memento）模式的定义：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。

## 2.优缺点

备忘录模式是一种对象行为型模式，其主要优点如下。

- 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。

其主要缺点是：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。

## 3.结构

备忘录模式的核心是设计备忘录类以及用于管理备忘录的管理者类。

备忘录模式的主要角色如下。

1. 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。
2. 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。
3. 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。

## 4.实现

```
package memento;
public class MementoPattern
{
    public static void main(String[] args)
    {
        Originator or=new Originator();
        Caretaker cr=new Caretaker();
        or.setState("S0");
        System.out.println("初始状态:"+or.getState());
        cr.setMemento(or.createMemento()); //保存状态
        or.setState("S1");
        System.out.println("新的状态:"+or.getState());
        or.restoreMemento(cr.getMemento()); //恢复状态
        System.out.println("恢复状态:"+or.getState());
    }
}
//备忘录
class Memento
{
    private String state;
```

```
    public Memento(String state)
    {
        this.state=state;
    }
    public void setState(String state)
    {
        this.state=state;
    }
    public String getState()
    {
        return state;
    }
}
//发起人
class Originator
{
    private String state;
    public void setState(String state)
    {
        this.state=state;
    }
    public String getState()
    {
        return state;
    }
    public Memento createMemento()
    {
        return new Memento(state);
    }
    public void restoreMemento(Memento m)
    {
        this.setState(m.getState());
    }
}
//管理者
class Caretaker
{
    private Memento memento;
    public void setMemento(Memento m)
    {
        memento=m;
    }
    public Memento getMemento()
    {
        return memento;
    }
}


程序运行的结果如下:
初始状态:S0
新的状态:S1
恢复状态:S0
```

## 5.应用场景

1. 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。
2. 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。

# 3.23解释器模式

## 1.定义

解释器（Interpreter）模式的定义：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。

这里提到的文法和句子的概念同编译原理中的描述相同，“文法”指语言的语法规则，而“句子”是语言集中的元素。例如，汉语中的句子有很多，“我是中国人”是其中的一个句子，可以用一棵语法树来直观地描述语言中的句子。

## 2.优缺点

解释器模式是一种类行为型模式，其主要优点如下。

1. 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。
2. 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。

解释器模式的主要缺点如下。

1. 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。
2. 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。
3. 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。

## 3.结构

解释器模式包含以下主要角色。

1. 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()。
2. 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。
3. 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。
4. 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。
5. 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。

## 4.实现

```
1  package interpreterPattern;
```

```
import java.util.*;
/*文法规则
  <expression> ::= <city>的<person>
  <city> ::= 韶关|广州
  <person> ::= 老人|妇女|儿童
*/
public class InterpreterPatternDemo
{
    public static void main(String[] args)
    {
        Context bus=new Context();
        bus.freeRide("韶关的老人");
        bus.freeRide("韶关的年轻人");
        bus.freeRide("广州的妇女");
        bus.freeRide("广州的儿童");
        bus.freeRide("山东的儿童");
    }
}
//抽象表达式类
interface Expression
{
    public boolean interpret(String info);
}
//终结符表达式类
class TerminalExpression implements Expression
{
    private Set<String> set= new HashSet<String>();
    public TerminalExpression(String[] data)
    {
        for(int i=0;i<data.length;i++)set.add(data[i]);
    }
    public boolean interpret(String info)
    {
        if(set.contains(info))
        {
            return true;
        }
        return false;
    }
}
//非终结符表达式类
class AndExpression implements Expression
{
    private Expression city=null;
    private Expression person=null;
    public AndExpression(Expression city,Expression person)
    {
        this.city=city;
        this.person=person;
    }
    public boolean interpret(String info)
    {
        String s[]=info.split("的");
```

```
        return city.interpret(s[0])&&person.interpret(s[1]);
    }
}
//环境类
class Context
{
    private String[] citys={"韶关","广州"};
    private String[] persons={"老人","妇女","儿童"};
    private Expression cityPerson;
    public Context()
    {
        Expression city=new TerminalExpression(citys);
        Expression person=new TerminalExpression(persons);
        cityPerson=new AndExpression(city,person);
    }
    public void freeRide(String info)
    {
        boolean ok=cityPerson.interpret(info);
        if(ok) System.out.println("您是"+info+"，您本次乘车免费！");
        else System.out.println(info+"，您不是免费人员，本次乘车扣费2元！");
    }
}

程序运行结果如下：

您是韶关的老人，您本次乘车免费!
韶关的年轻人，您不是免费人员，本次乘车扣费2元!
您是广州的妇女，您本次乘车免费!
您是广州的儿童，您本次乘车免费!
山东的儿童，您不是免费人员，本次乘车扣费2元!
```

## 5.应用场景

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。

1. 当语言的文法较为简单，且执行效率不是关键问题时。
2. 当问题重复出现，且可以用一种简单的语言来进行表达时。
3. 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。

注意：解释器模式在实际的软件开发中使用比较少，因为它会引起效率、性能以及维护等问题。如果碰到对表达式的解释，在 Java中可以用 Expression4J 或 Jep 等来设计。

参考文献：图说设计模式 : https://design-patterns.readthedocs.io/zh_CN/latest/

Java设计模式 : http://c.biancheng.net/design_pattern/