# **什么是设计模式**

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。

使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。简单说：

**模式：在某些场景下，针对某类问题的某种通用的解决方案。**

场景：项目所在的环境

问题：约束条件，项目目标等

解决方案：通用、可复用的设计，解决约束达到目标。

# **设计模式7大原则**

## **2.0. 七大原则总结**

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：

◆　开闭原则是总纲，它告诉我们要对扩展开放(对提供者而言)，对修改关闭(对使用者而言)。

◆　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。

## 2.1**单一职责原则（Single Responsibility Principle，SRP）**

### **含义**

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。

### **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。

### **实现方式**

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。

## 2.2**接口隔离原则（Interface Segregation Principle，ISP）**

### **含义**

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。

### **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：

◆　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。

◆　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。

◆　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。

◆　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。

◆　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。

### **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。

## 2.3 **依赖倒置原则（Dependence Inversion Principle，DIP）**

### **含义**

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。其核心思想是：要面向接口编程，不要面向实现编程。

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。

### **作用**

◆　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。

### **实现方式**

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：

◆　每个类尽量提供接口或抽象类，或者两者都具备。

◆　变量的声明类型尽量是接口或者是抽象类。

◆　任何类都不应该从具体类派生。

◆　使用继承时尽量遵循里氏替换原则。

## 2.4**里氏替换原则（Liskov Substitution Principle，LSP）**

### **含义**

继承必须确保超类所拥有的性质在子类中仍然成立。

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。里氏替换原则是继承复用的基础，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。

### **作用**

◆　里氏替换原则是实现开闭原则的重要方式之一。

◆　它克服了继承中重写父类造成的可复用性变差的缺点。

◆　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。

### **实现方式**

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。

## 2.5 **开闭原则（Open Closed Principle，OCP）**

### **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。

### **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：

◆　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。

◆　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。

◆　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。

### **实现方式**

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。

## 2.6**迪米特法则（Law of Demeter，LoD）**

### **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。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。

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

### **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：

◆　降低了类之间的耦合度，提高了模块的相对独立性。

◆　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，在釆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。

### **实现方式**

从迪米特法则的定义和特点可知，它强调以下2点：

◆　从依赖者的角度来说，只依赖应该依赖的对象。

◆　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：

◆　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。

◆　在类的结构设计上，尽量降低类成员的访问权限。

◆　在类的设计上，优先考虑将一个类设置成不变类。

◆　在对其他类的引用上，将引用其他对象的次数降到最低。

◆　不暴露类的属性成员，而应该提供相应的访问器（set方法和get 方法）。

◆　谨慎使用序列化（Serializable）功能。

## 2.7**合成复用原则（Composite Reuse Principle，CRP）**

### **含义**

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。

### **作用**

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：

◆　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。

◆　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。

◆　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：

◆　它维持了类的封装性。因为成分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。

◆　新旧类之间的耦合度低。这种复用所需的依赖较少，新对象存取成分对象的唯一方法是通过成分对象的接口。

◆　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分对象类型相同的对象。

### **实现方式**

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。

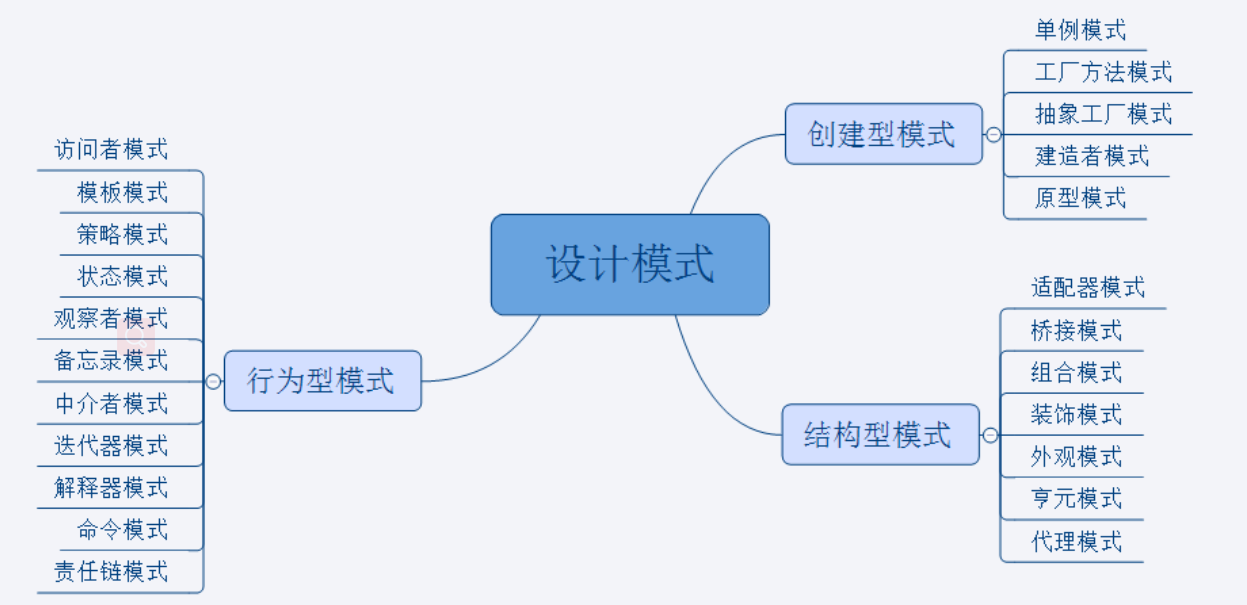
# **设计模式的三个分类**

**创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。**

**结构型模式：把类或对象结合在一起形成一个更大的结构(功能组合)。**

**行为型模式：类和对象如何交互，及划分责任和算法(访问交互)。**

**如下图所示：**



# 设计模式34个

## 工厂(方法)模式

工厂方法(接口)：定义一个创建对象的接口，让子类决定实例化那个类。

简单工厂(接口实例)：一个工厂类根据传入的参量决定创建出那一种产品类的实例 。

### 定义

作为抽象工厂模式的孪生兄弟，工厂方法模式定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个，也就是说工厂方法模式让实例化推迟到子类。

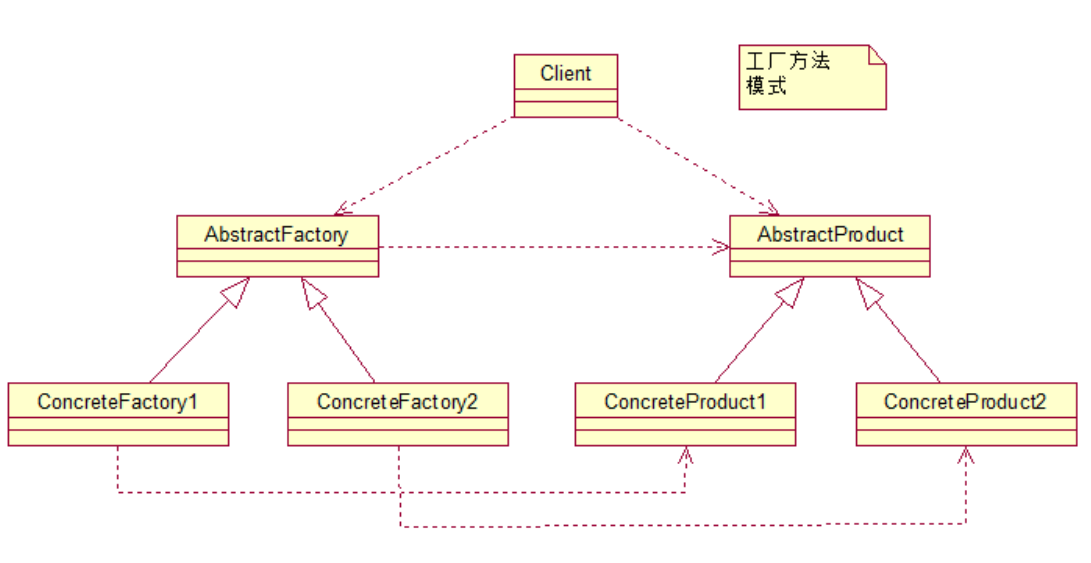
优点：

1. 工厂方法模式非常符合“开闭原则”，当需要增加一个新的产品时，我们只需要增加一个具体的产品类和与之对应的具体工厂即可，无须修改原有系统。
2. 同时在工厂方法模式中用户只需要知道生产产品的具体工厂即可，无须关心产品的创建过程，甚至连具体的产品类名称都不需要知道。

缺点：

1. 虽然他很好的符合了“开闭原则”，但是由于每新增一个新产品时就需要增加两个类，这样势必会导致系统的复杂度增加。
2. 客户端使用抽象的接口来创建一个相关的产品，需要关心实际产出的具体产品是什么, 可与和具体的产品耦合度比较高。

### 1.2 UML 类图



在工厂方法模式中用户只需要知道生产产品的具体工厂即可

客户端使用抽象的接口来创建一个相关的产品，需要关心实际产出的具体产品是什么

### 1.3 小栗子

#### 小栗子-简单工厂模式

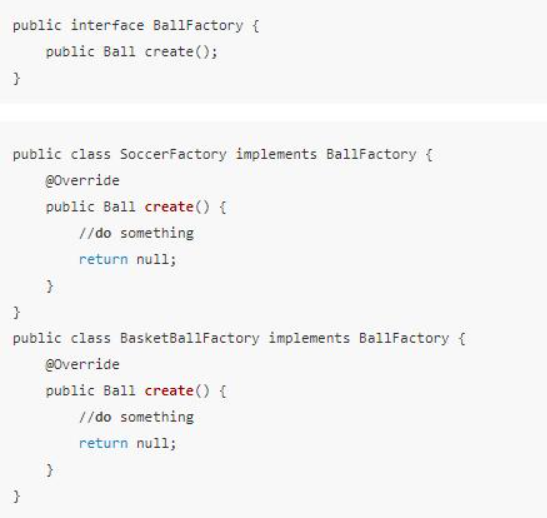
****简单工厂模式****

这个用的比较少，就是有个工厂，告诉你我要什么东西，你造好了给我就行。比如说：



#### 小栗子-工厂模式

这个其实和简单工厂模式差不太多，就是将工厂继续拆分，比如说刚刚EasyBallFactory是一个总工厂，我们现在拆分成SoccerFactory和BasketBallFactory分别生产足球和篮球。某个工厂内部可以根据需求生产不同的产品，比如说soccer可以生产不同大小的出来。



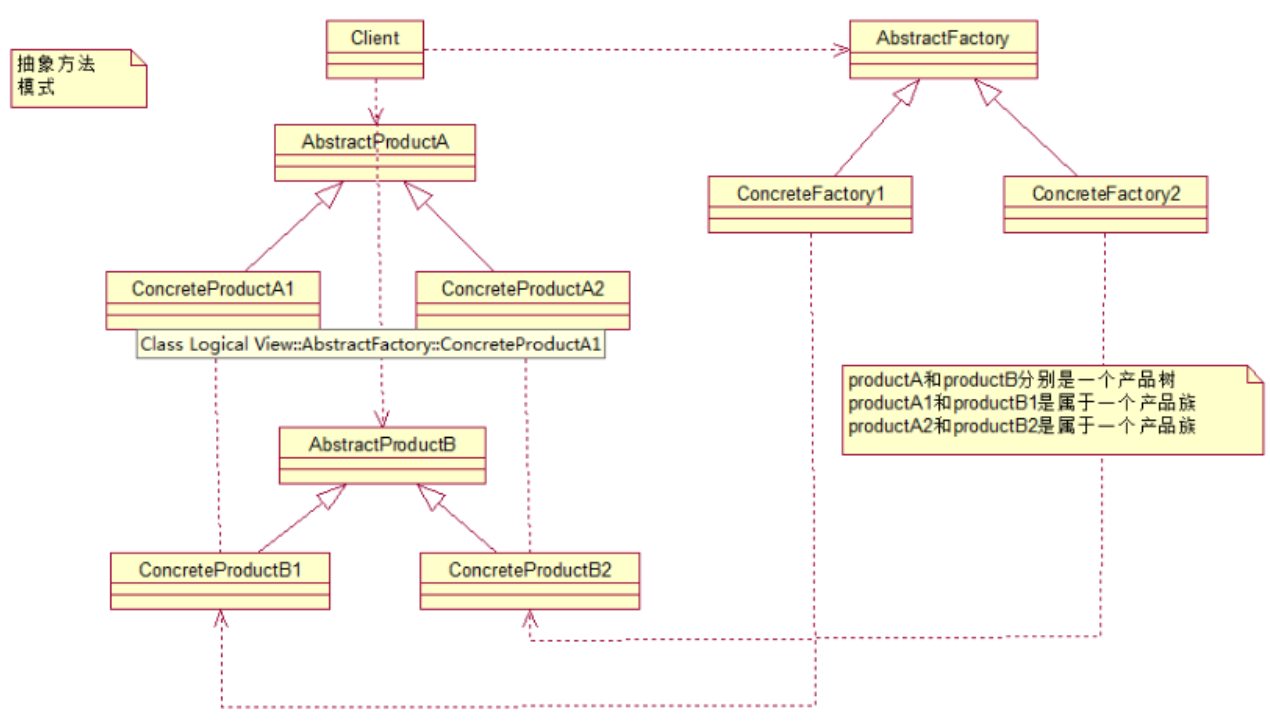
## 抽象工厂模式

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。

### 定义

所谓抽象工厂模式就是提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类。他允许客户端使用抽象的接口来创建一组相关的产品，而不需要关心实际产出的具体产品是什么。这样一来，客户就可以从具体的产品中被解耦。它的优点是隔离了具体类的生成，使得客户端不需要知道什么被创建了，而缺点就在于新增新的行为会比较麻烦，因为当添加一个新的产品对象时，需要更改接口及其下所有子类。

### UML类图



ConcreteProductA1 和 ConcreteProductB1是一个家族的产品，他们之间是相关或者依赖关系。

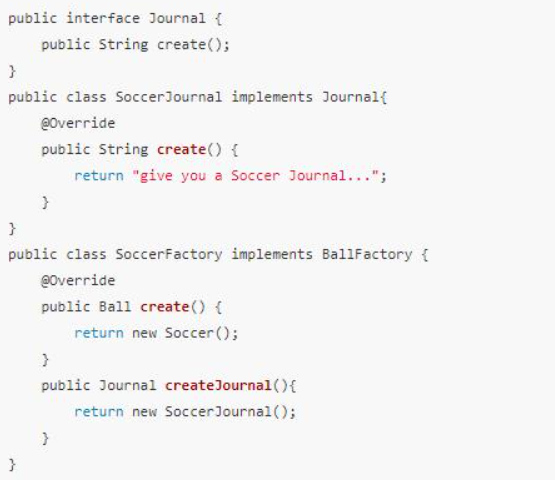
ConcreteFactory1可以同时生成ConcreteProductA1和ConcreteProductB1两个产品。(在内部提供两个方法，分别生产A1和B1)

### 2.3 小栗子

#### 小栗子-01

抽象工厂模式主要设计产品组的概念，就是某一个工厂生产出配套的一系列产品。例如，在生产足球的同时，SoccerFactory还可以生产与之配套的足球杂志。

(BallFactory是球厂)



## 单例模式

单例模式：某个类只能有一个实例，提供一个全局的访问点。

### 3.1 定义

单例模式，它的定义就是确保某一个类只有一个实例，并且提供一个全局访问点。

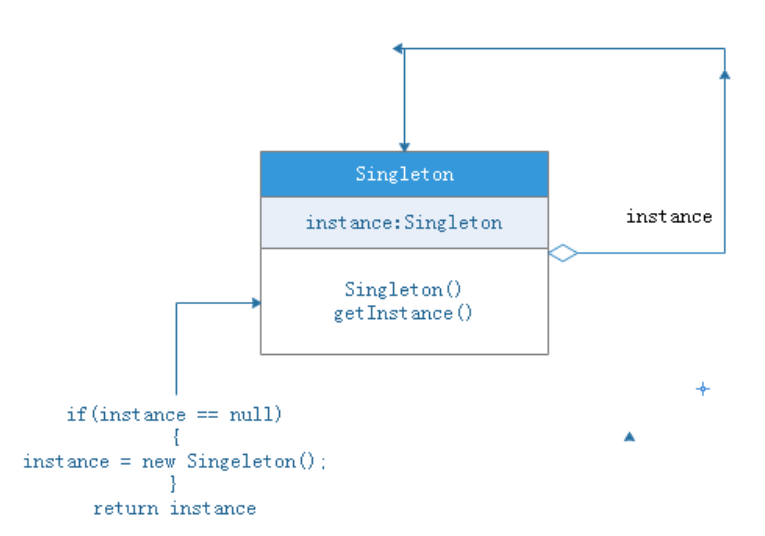
单例模式具备典型的3个特点：1、只有一个实例。 2、自我实例化。 3、提供全局访问点。

 因此当系统中只需要一个实例对象或者系统中只允许一个公共访问点，除了这个公共访问点外，不能通过其他访问点访问该实例时，可以使用单例模式。

单例模式的主要优点就是节约系统资源、提高了系统效率，同时也能够严格控制客户对它的访问。也许就是因为系统中只有一个实例，这样就导致了单例类的职责过重，违背了“单一职责原则”，同时也没有抽象类，所以扩展起来有一定的困难。

### 3.2 UML类图

其UML结构图非常简单，就只有一个类：



### 3.3 小栗子

#### 小栗子-01

单例模式有很多种形式，最佳实践应该是两重判断，保证只new出来一个。单例可以说是非常普遍的设计模式了。单例就是指在服务容器的生命周期中只能有这么一个。比如说Servlet、Spring中注入的Bean等等都是单例的。



## 建造者模式

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

### 定义

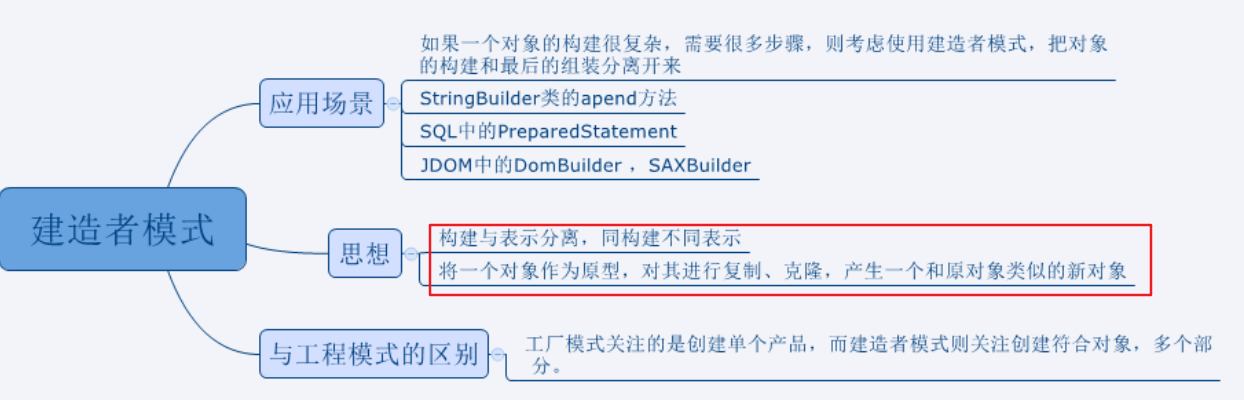
对于建造者模式而言，它主要是将一个复杂对象的构建与表示分离，使得同样的构建过程可以创建不同的表示。适用于那些产品对象的内部结构比较复杂。

优点：建造者模式将复杂产品的构建过程封装分解在不同的方法中，使得创建过程非常清晰，能够让我们更加精确的控制复杂产品对象的创建过程，同时它隔离了复杂产品对象的创建和使用，使得相同的创建过程能够创建不同的产品。

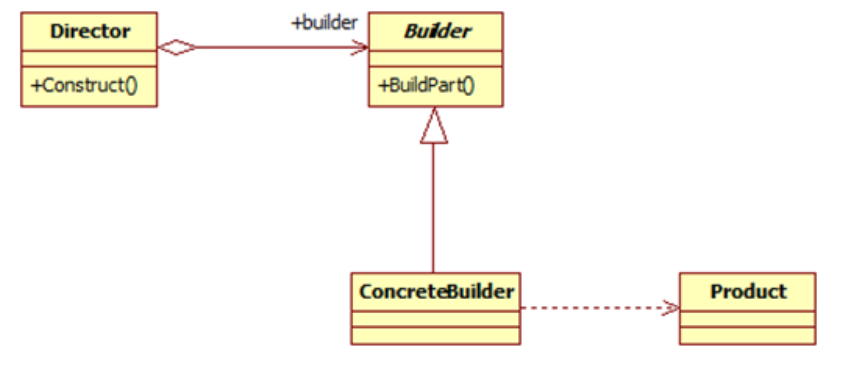
缺点：但是如果某个产品的内部结构过于复杂，将会导致整个系统变得非常庞大，不利于控制，同时若几个产品之间存在较大的差异，则不适用建造者模式，毕竟这个世界上存在相同点大的两个产品并不是很多，所以它的使用范围有限。

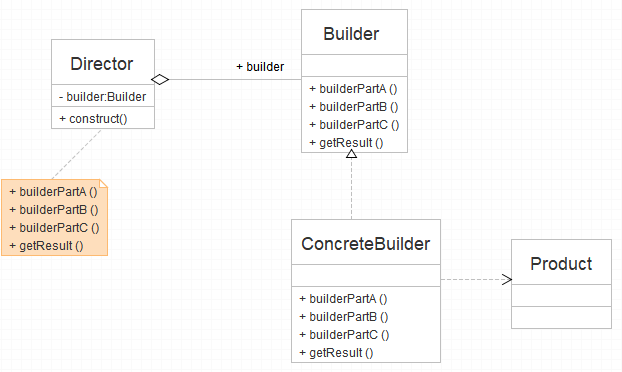
* 与工程模式的区别：

工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。



### UML类图

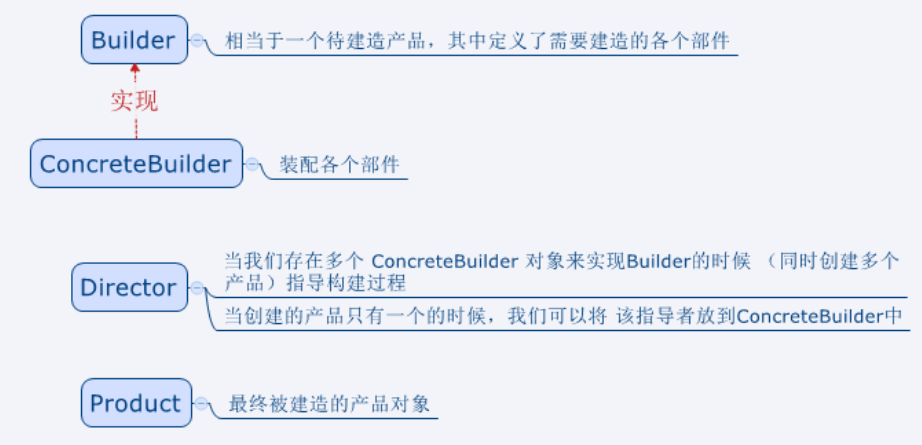




### 4.3模式结构

建造者模式的四个角色

1. Product(产品角色) ：一个具体的产品对象
2. Builder(抽象建造者): 创建一个Product对象的各个部件指定的接口。
3. ConcreteBuilder(具体的建造者)：实现接口，构建和装配各个部件。
4. Director(指挥者)：构建一个使用Builder接口的对象。它主要是用于创建一个复杂的对象。它主要有两个作用：a. 隔离了客户与对象的生产过程。 b. 负责控制产品对象的生产过程。



### 4.4 建造者模式的小栗子

#### 小栗子-01

建立一个建造者模式的小栗子。

我们建造一个属性为Person的产品，同时创建Man 和Women.

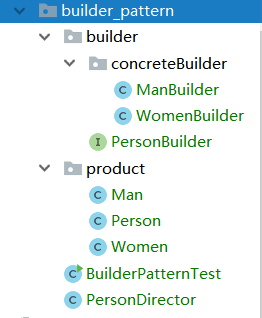
1.创建接口PersonBuilder，其中包含需要创建的 head body foot   即  Builder

2.创建实现类WomenBuilder和ManBuilder来实现PersonBuilder接口中的方法，即 ConcreteBuilder

3.创建PersonDirector 来指定你要创建的是Women还是Man,即Director

4.由于最后生成的产品是Person，所以我们要创建实体类Person,

5.同时我们可以创建普通类Man和Women,来继承Person, 增强拓展性，方便PersonDirector进行管理。



1. 创建产品类Person

|  |
| --- |
| **public class** Person {  **private** String **head** ;  **private** String **body** ;  **private** String **foot** ;   Setter 和 Getter ，还有有toString()方法进行了省略 } |

1. 创建Man和Women类来继承Person

|  |
| --- |
| **public class** Man **extends** Person {  **public** Man() {  System.***out***.println(**"开始建造男神"**);  } }  **public class** Women **extends** Person {  **public** Women() {  System.***out***.println(**"开始建造女神"**);  } } |

3.创建接口PersonBuilder，其中包含需要创建的 head body foot   即  Builder

|  |
| --- |
| **public interface** PersonBuilder {  **void** buildHead();  **void** buildBody();  **void** buildFoot();   Person builderPerson(); } |

4.创建实现类WomenBuilder和ManBuilder来实现PersonBuilder接口中的方法，即 ConcreteBuilder

|  |
| --- |
| **public class** ManBuilder **implements** PersonBuilder {  Person **person**;   **public** ManBuilder() { **person**=**new** Man(); }  @Override  **public void** buildHead() { **person**.setHead(**"建造男神的头"**); }  @Override  **public void** buildBody() { **person**.setBody(**"建造男神的身体"**); }  @Override  **public void** buildFoot() { **person**.setFoot(**"建造男神的脚"**); }  @Override  **public** Person builderPerson() {  **return person**;  } }  **public class** WomenBuilder **implements** PersonBuilder {  Person **person**;   **public** ManBuilder() { **person**=**new** Man(); }  @Override  **public void** buildHead() { **person**.setHead(**"建造女神的头"**); }  @Override  **public void** buildBody() { **person**.setBody(**"建造女神的身体"**); }  @Override  **public void** buildFoot() { **person**.setFoot(**"建造女神的脚"**); }  @Override  **public** Person builderPerson() {  **return person**;  } } |

1. 创建PersonDirector 来指定你要创建的是Women还是Man,即Director

|  |
| --- |
| **public class** PersonDirector {  */\*\**  ***建造者模式与工程模式的区别:工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。*** *\* 构造人  \** ***@param pb*** *builder  \** ***@return*** *\*/* **public** Person constructPerson(PersonBuilder pb) {  pb.buildHead();  pb.buildBody();  pb.buildFoot();  **return** pb.builderPerson();  } } |

6.建个测试类测试一下。

|  |
| --- |
| **public class** BuilderPatternTest {  **public static void** main(String[] args) {  PersonDirector pd=**new** PersonDirector();  Person manPerson=pd.constructPerson(**new** ManBuilder());  Person womenPerson=pd.constructPerson(**new** WomenBuilder());   System.***out***.println(manPerson);  System.***out***.println(womenPerson);  } } |

#### 小栗子-02

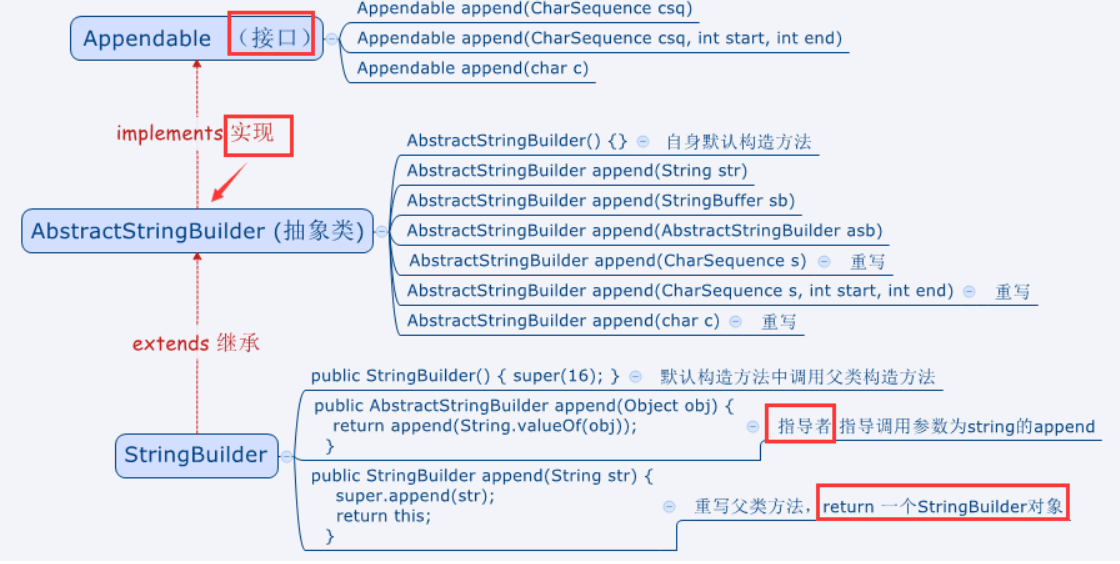
将一个复杂对象分布创建。如果一个超大的类的属性特别多，我们可以把属性分门别类，不同属性组成一个稍微小一点的类，再把好几个稍微小点的类窜起来。比方说一个电脑，可以分成不同的稍微小点的部分CPU、主板、显示器。CPU、主板、显示器分别有更多的组件，不再细分。





SpringBoot实现了0配置，几乎所有的配置都写到了java代码中，大量的配置不得不让配置类采用建造者模式，这样层次比较清晰。

#### 小栗子- StringBuilder类的apend方法



#### 小栗子-SQL-中的PrepareStatement

#### 小栗子-JDOM中的DomBuilder,SAXBuilder

## 原型模式

原型模式：通过复制现有的实例来创建新的实例。

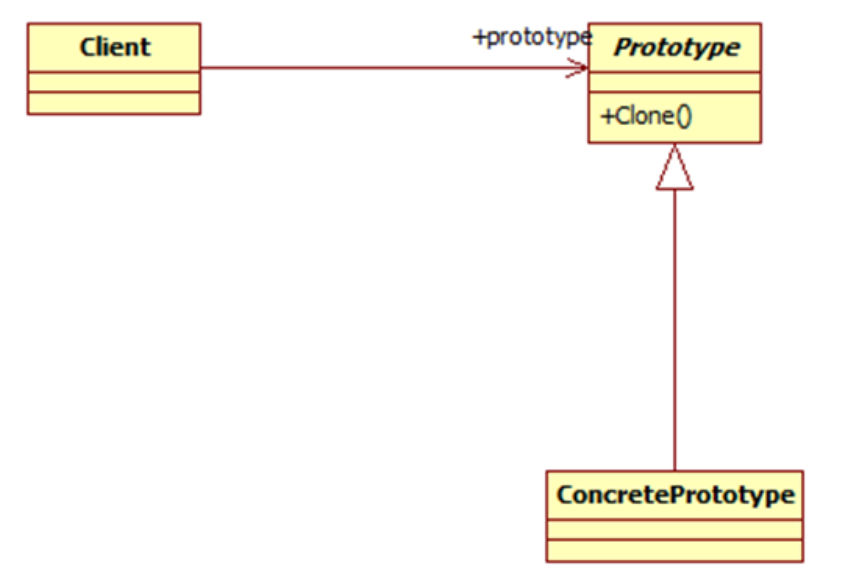
### 5.1定义

在我们应用程序可能有某些对象的结构比较复杂，但是我们又需要频繁的使用它们，如果这个时候我们来不断的新建这个对象势必会大大损耗系统内存的，这个时候我们需要使用原型模式来对这个结构复杂又要频繁使用的对象进行克隆。所以原型模式就是用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。

它主要应用于那些创建新对象的成本过大时的情景。它的主要优点就是简化了新对象的创建过程，提高了效率，同时原型模式提供了简化的创建结构。

### 5.2 UML类图

Prototype(原型模式)



### ****5.3模式结构****

原型模式包含如下角色：  
Prototype：抽象原型类  
ConcretePrototype：具体原型类  
Client：客户类

### 5.4 小栗子

#### 小栗子-01

原型模式用的比较少，用于创建重复对象。需要实现Cloneable。 可以选择重写clone()方法。

clone分为浅克隆和深克隆：浅克隆只是克隆引用，对象还是一个；深克隆是对象也新创建了一个。如下：



## 适配器模式

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

### 定义

在我们的应用程序中我们可能需要将两个不同接口的类来进行通信，在不修改这两个接口的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。

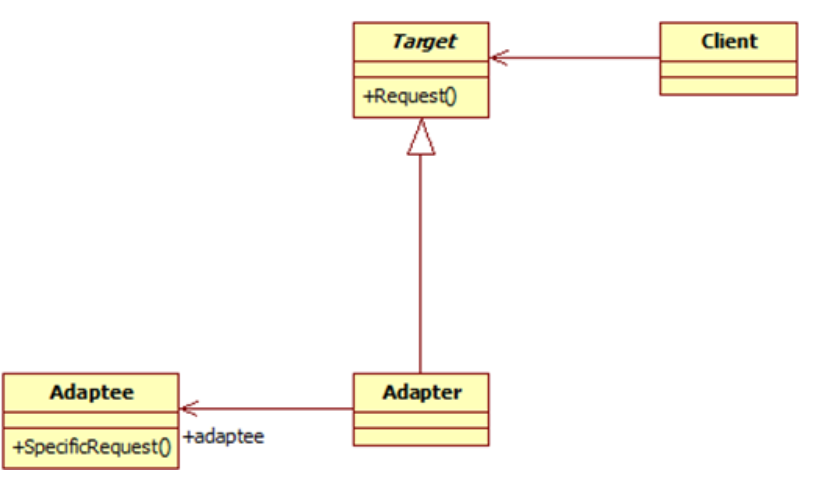
作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

适配器，顾名思义，是

a.让两个不兼容的东西可以一起工作。例如插座的电源是220V，手机直接给他220V 50HZ的交流电我相信一般都会直接炸了（除了诺基亚...）手机充电器就进行了适配，将电压变小，交流电变成直流电。除了这种需要改变属性的操作（比较好说，不举例子了），

B.适配器还用于在接口继承方面。假设一个顶级接口有一大堆方法需要实现类实现，我新写了个类只是想选择的实现一两个接口，那其他的方法我是不是都需要实现一下，即使是空实现（单纯实现，不进行任何逻辑操作），这是我们就需要一个适配器类，空实现那些方法，我的新类只需要继承这个适配器类就好了，要是想实现某个方法，只需要重写掉配置类中对应的方法就好。这种模式基本都会用到，毕竟谁的代码还没个顶级接口啊。

### UML类图



### 模式结构

适配器模式包含如下角色：

Target：目标抽象类

Adapter：适配器类

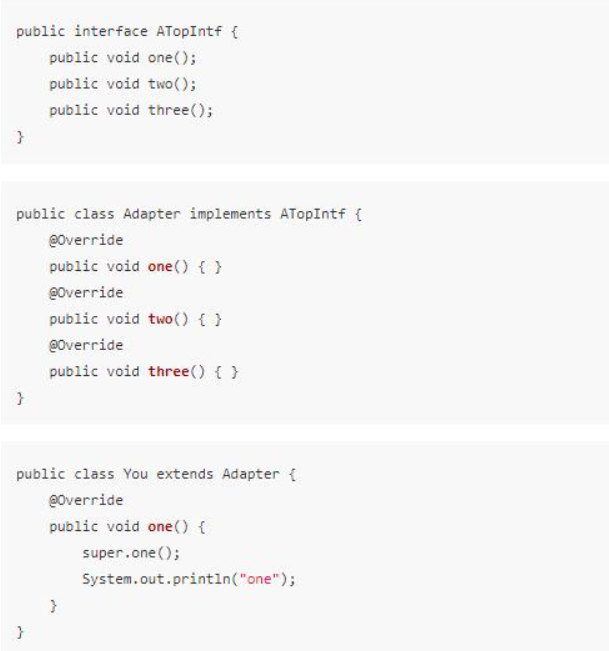
Adaptee：适配者类

Client：客户类

### 7.3 小栗子

#### 小栗子-01

下面是b. 适配器在接口继承方面的应用示例



## 桥接模式

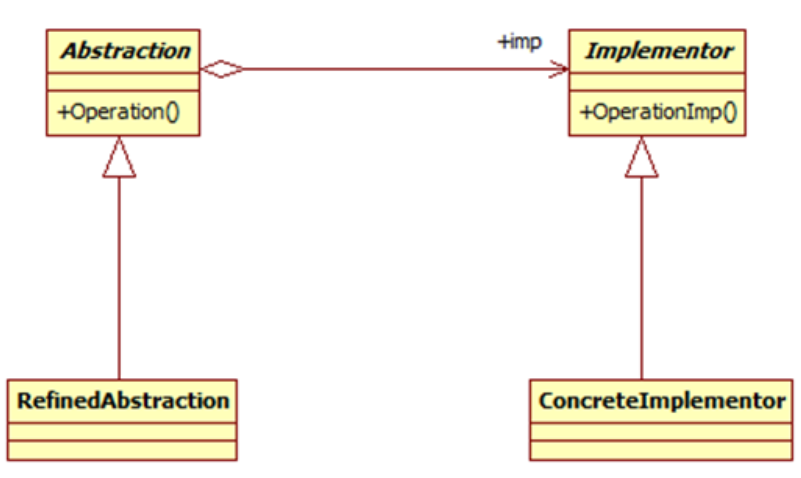
桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

### 8.1定义

如果说某个系统能够从多个角度来进行分类，且每一种分类都可能会变化，那么我们需要做的就是将这多个角度分离出来，使得他们能独立变化，减少他们之间的耦合，这个分离过程就使用了桥接模式。所谓桥接模式就是将抽象部分和实现部分隔离开来，使得他们能够独立变化。

桥接模式将继承关系转化成关联关系，封装了变化，完成了解耦，减少了系统中类的数量，也减少了代码量。

### 8.2 UML类图



### 8.3 模式结构

桥接模式包含如下角色：

Abstraction：抽象类

RefinedAbstraction：扩充抽象类（但是实现类可以不进行改变）

Implementor：实现类接口

ConcreteImplementor：具体实现类

### 8.4 小栗子

#### 小栗子-01

就是用于抽象化和实现化的解耦。又是解耦，貌似设计模式就是教我们如何优雅的解耦。提高了代码的拓展性，并且可以实现代码的动态切换。 最开始的Ball、Soccer、BasketBall接着用，增加新的类。

其中create(){} 使用了桥接模式。 BallCut聚合了 Ball对象。





## 过滤器模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 组合模式

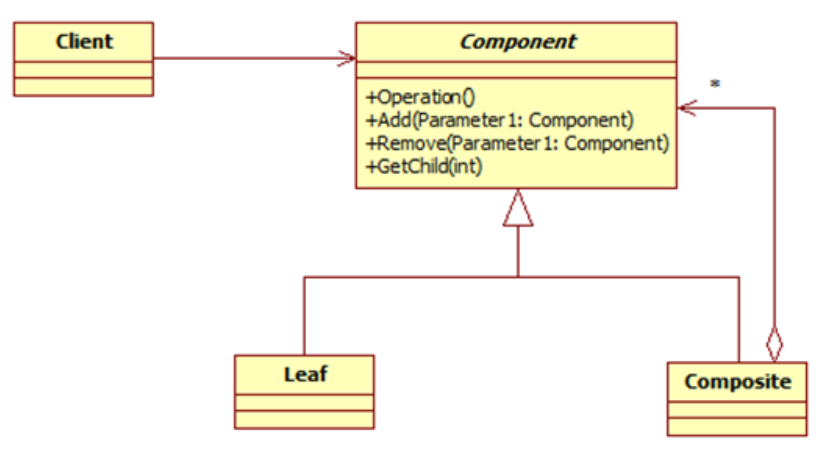
组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。

### 10.1定义

组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。它定义了如何将容器对象和叶子对象进行递归组合，使得客户在使用的过程中无须进行区分，可以对他们进行一致的处理。在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因。

虽然组合模式能够清晰地定义分层次的复杂对象，也使得增加新构件也更容易，但是这样就导致了系统的设计变得更加抽象，如果系统的业务规则比较复杂的话，使用组合模式就有一定的挑战了。

### 10.2UML类图



Composite：复合的,合成

Component: 组成部分

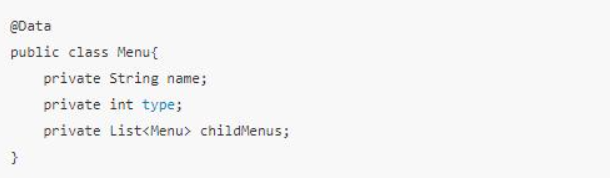
### 10.3模式结构

**模式结构**  
组合模式包含如下角色：  
Component: 抽象构件  
Leaf: 叶子构件  
Composite: 容器构件  
Client: 客户类

### 10.4小栗子

#### 小栗子-01

组合模式是将存在某种包含关系的数据组织在一起，典型的例子就是树状结构。例如菜单功能，一个菜单除了自己该有的属性，还可能包含子菜单，创建的时候可以使用递归的方法。



## 装饰器模式

装饰模式：动态的给对象添加新的功能。

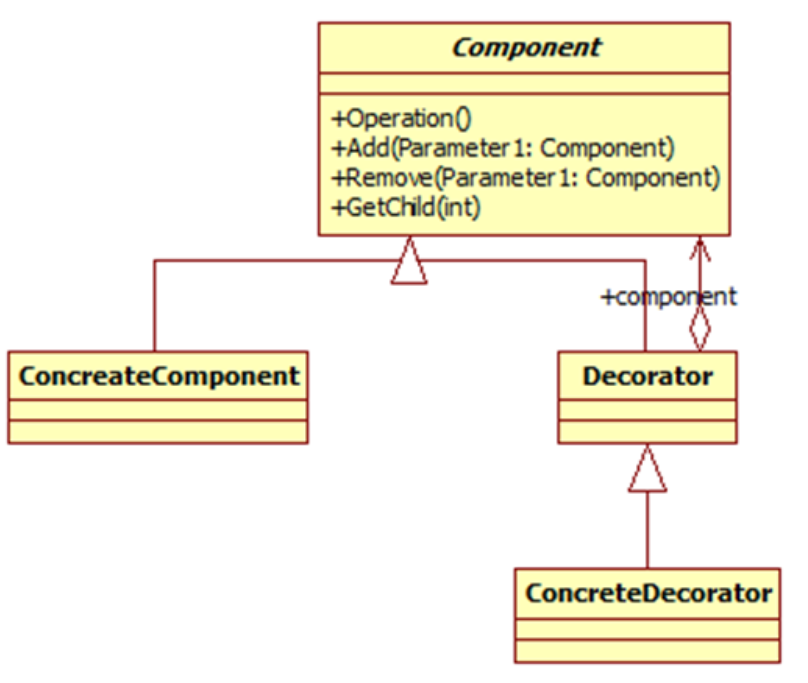
### 11.1定义

我们可以通过继承和组合的方式来给一个对象添加行为，虽然使用继承能够很好拥有父类的行为，但是它存在几个缺陷：一、对象之间的关系复杂的话，系统变得复杂不利于维护。二、容易产生“类爆炸”现象。三、是静态的。在这里我们可以通过使用装饰者模式来解决这个问题。

装饰者模式，动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更加有弹性的替代方案。虽然装饰者模式能够动态将责任附加到对象上，但是他会产生许多的细小对象，增加了系统的复杂度。

一个装饰类，在原来类的基础上增加一点功能。是不是和代理模式很像，我甚至可以将整个代码搬过来照样可以说的通的。这两个模式意思上有点差别，代理模式是原对象做不了那件事，必须让代理对象去做，主导侧重于代理对象，比如说买车。装饰模式是说，就是让原对象直接去做这件事，只是功能上增强一点，主导在于原对象。比如说炒菜的时候撒点盐。

### 11.2 UML类图



### 11.3模式结构

装饰模式包含如下角色：  
Component: 抽象构件  
ConcreteComponent: 具体构件  
Decorator: 抽象装饰类  
ConcreteDecorator: 具体装饰类

从UML类图可以看到，Decorator 继承了Component，同时内部聚合了Component；

(和代理模式很相似)

### 11.4小栗子

#### 小栗子-01

#### 小栗子- FilterInputStream -java

**装饰着模式：**FilterInputStream 继承 InputStream的同时，也聚合了一个 InputStream实例。

Component: 抽象构件 InputStream   
ConcreteComponent: 具体构件 FileInputStream  
Decorator: 抽象装饰类 FilterInputStream

ConcreteDecorator: 具体装饰类 BufferedInputStream

1. 抽象装饰类 FilterInputStream

**//这个类最能体现装饰者的结构**

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

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

**public int** read(**byte** b[], **int** off, **int** len) **throws** IOException {  
 **return in**.read(b, off, len);  
 }

其他功能省略

}

2. 具体装饰类 BufferedInputStream

**public  
class** BufferedInputStream **extends** FilterInputStream {

**protected volatile byte buf**[];

**protected int pos**; //当前读取字节位置

**protected int count**; //缓存buff中实际拥有的字节数量

**public** BufferedInputStream(InputStream in, **int** size) {  
 **super**(in); // 保存 具体构件 InputStream  
 **buf** = **new byte**[size];  
 }

// 装饰者调用 具体构建的功能

**private void** fill() **throws** IOException {

InputStream input = **super**.**in**;

input.read(**buf**, **pos**, buffer.**length** - **pos**);

}

**public synchronized int** read() **throws** IOException {  
 **if** (**pos** >= **count**) {  
 fill();  
 **if** (**pos** >= **count**)  
 **return** -1;  
 }  
 **return buf**[**pos**++] & 0xff;  
 }

1. 测试类 具体构件 FileInputStream 抽象构件 InputStream

**public static void** main(String[] args) **throws** IOException {

*//****TODO 装饰者设计模式*** FileInputStream fileInputStream = **new** FileInputStream(**""**);  
 *// 缓存字节流 public BufferedInputStream(InputStream in, int size) size是缓存的字节大小* BufferedInputStream bufferedInputStream = **new** BufferedInputStream(fileInputStream,1024);  
  
 *//先进行缓存的填充，然后从缓存中读取一个字节进行返回* **int** read = bufferedInputStream.read();  
}

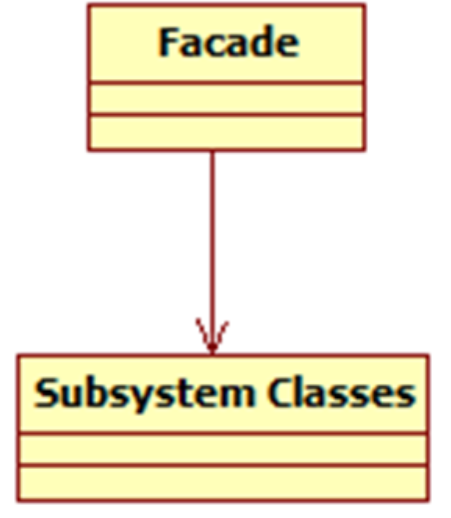
## 外观模式

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

### 12.1定义

我们都知道类与类之间的耦合越低，那么可复用性就越好，如果两个类不必彼此通信，那么就不要让这两个类发生直接的相互关系，如果需要调用里面的方法，可以通过第三者来转发调用。外观模式非常好的诠释了这段话。外观模式提供了一个统一的接口，用来访问子系统中的一群接口。它让一个应用程序中子系统间的相互依赖关系减少到了最少，它给子系统提供了一个简单、单一的屏障，客户通过这个屏障来与子系统进行通信。通过使用外观模式，使得客户对子系统的引用变得简单了，实现了客户与子系统之间的松耦合。但是它违背了“开闭原则”，因为增加新的子系统可能需要修改外观类或客户端的源代码。

### 12.2 UML类图



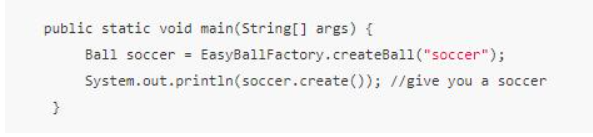
### 12.3 模式结构

外观模式包含如下角色：  
Facade(外观): 外观角色  
SubSystem:子系统角色

### 12.4 小栗子

#### 小栗子-01

又称门面模式，就是一个门面，一个操作无需让对象知道其内部实现的复杂度，尽量让用户感知到是非常简单的。这就是为什么我们controller层尽量（或者说一定）少些业务逻辑，让controller层只是起到一个传参和通用性参数校验的功能，剩下的全交给service去做吧。我们还需要在代码中不断将“长得”特别长的代码封装成一个方法，“让处处都有好看的外观”。看一下我们曾写过的代码，这里只起到了传参的作用，究竟这个足球是怎么创建出来的，客户端不必担心。



## 亨元模式

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

### 13.1定义

在一个系统中对象过多会使得内存占用过多，特别是那些大量重复的对象，这就是对系统资源的极大浪费。享元模式对对象的重用提供了一种解决方案，它使用共享技术对相同或者相似对象实现重用。

* 享元模式就是运行共享技术有效地支持大量细粒度对象的复用。

系统使用少量对象,而且这些都比较相似，状态变化小，可以实现对象的多次复用。

* 这里有一点要注意：享元模式要求能够共享的对象必须是细粒度对象。

享元模式通过共享技术使得系统中的对象个数大大减少了，同时享元模式使用了内部状态和外部状态，同时外部状态相对独立，不会影响到内部状态，所以享元模式能够使得享元对象在不同的环境下被共享。同时正是分为了内部状态和外部状态，享元模式会使得系统变得更加复杂，同时也会导致读取外部状态所消耗的时间过长。

享元模式尽可能的让用户复用已经有的对象，从而避免造成反复创建对象的资源浪费。首先就会想到数据库连接池还有String常量池，延伸一下，几乎所有和缓存有关的代码，多少都会用到享元模式。享元模式要求大部分的对象可以外部化。

这边要说两个概念，享元模式对象的属性可以分为两个部分:内部状态和外部状态。

* 内部状态是指不会随环境而改变的值，比如说个人信息。
* 外部状态是指随环境改变的值，不能进行共享的信息，如某大学生选修的课程。

### 13.2 UML类图



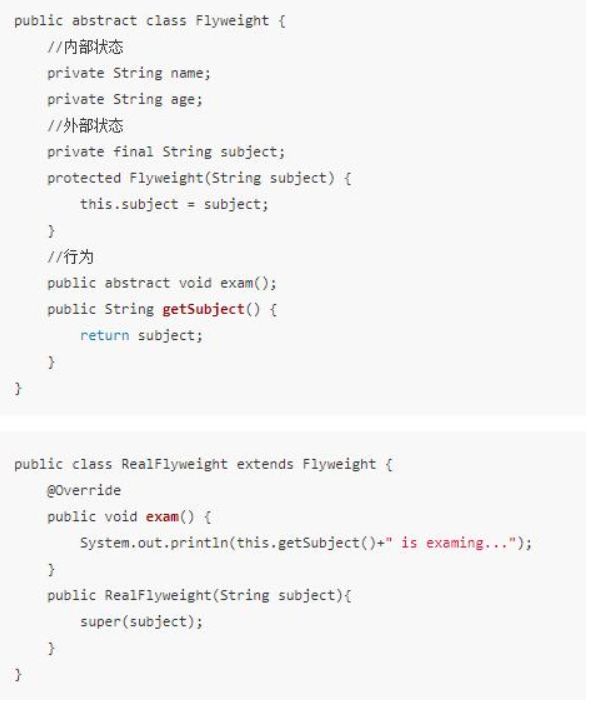
### 13.3模式结构

享元模式包含如下角色：  
Flyweight: 抽象享元类  
ConcreteFlyweight: 具体享元类  
UnsharedConcreteFlyweight: 非共享具体享元类  
FlyweightFactory: 享元工厂类

### 13.4小栗子

#### 小栗子-01

使用外部状态来作为key来查找是否有已创建的可共享的细粒度对象。



## 代理模式

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

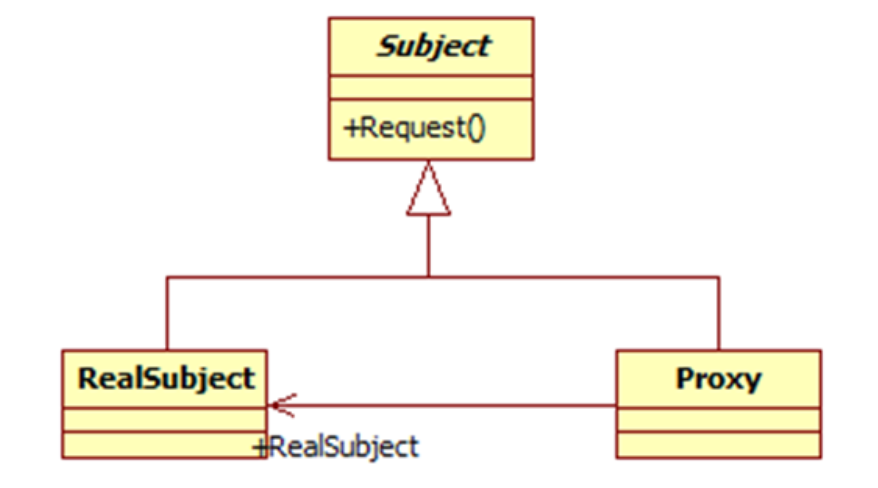
### 14.1定义

 代理模式就是给一个对象提供一个代理，并由代理对象控制对原对象的引用。它使得客户不能直接与真正的目标对象通信。

代理对象是目标对象的代表，其他需要与这个目标对象打交道的操作都是和这个代理对象在交涉。

代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了保护了目标对象的作用，同时也在一定程度上面减少了系统的耦合度。

### 14.2 UML类图



### 14.3模式结构

代理模式包含如下角色：  
 Subject: 抽象主题角色(接口)  
 Proxy: 代理主题角色  
 RealSubject: 真实主题角色

### 14.4小栗子

#### 小栗子-01

Spring的AOP用的是动态代理，何为动态不看了，用过Spring的小伙伴都知道吧。单纯看一下最基础代理模式是什么样的。代理就是，一个对象辅助另一个对象去做某件事，同时还可以增加一点辅助功能。例如，你买车，的确是你花钱把车买到了，但是你不可能直接去和厂家谈吧，你应该通过4S店购买，同时4S店帮助你入保险扣税等操作，最终你才得到了你想要的车。





## 责任链模式

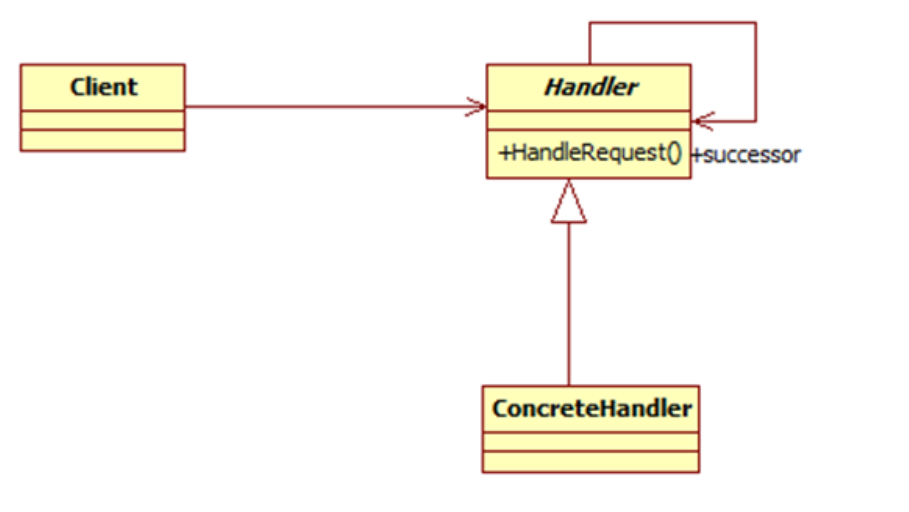
责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

### 15.1定义

职责链模式描述的请求如何沿着对象所组成的链来传递的。它将对象组成一条链，发送者将请求发给链的第一个接收者，并且沿着这条链传递，直到有一个对象来处理它或者直到最后也没有对象处理而留在链末尾端。

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止，这就是职责链模式。在职责链模式中，使得每一个对象都有可能来处理请求，从而实现了请求的发送者和接收者之间的解耦。同时职责链模式简化了对象的结构，它使得每个对象都只需要引用它的后继者即可，而不必了解整条链，这样既提高了系统的灵活性也使得增加新的请求处理类也比较方便。但是在职责链中我们不能保证所有的请求都能够被处理，而且不利于观察运行时特征。

### 15.2 UML类图



### 15.3 模式结构

职责链模式包含如下角色：  
Handler: 抽象处理者  
ConcreteHandler: 具体处理者  
Client: 客户类

### 15.4小栗子

#### 小栗子-01

责任链模式为请求创建一个接收者对象的链，对发送者和接受者进行解耦合。filter链就是责任链模式。



## 命令模式

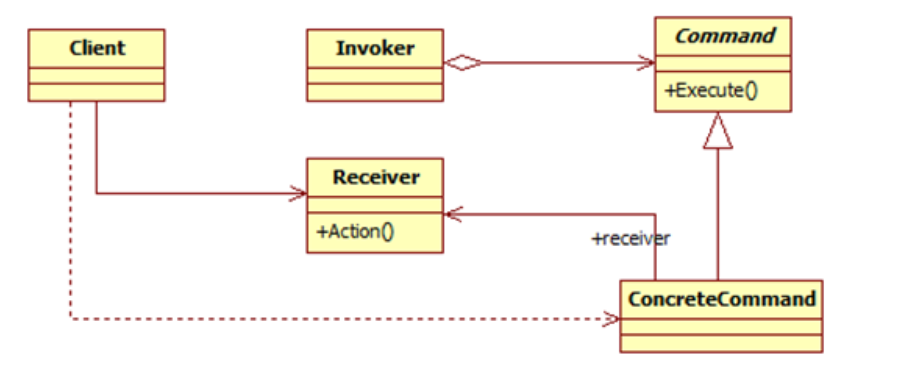
命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

### 16.1定义

 有些时候我们想某个对象发送一个请求，但是我们并不知道该请求的具体接收者是谁，具体的处理过程是如何的，们只知道在程序运行中指定具体的请求接收者即可，对于这样将请求封装成对象的我们称之为命令模式。所以命令模式将请求封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。同时命令模式支持可撤销的操作。

命令模式可以将请求的发送者和接收者之间实现完全的解耦，发送者和接收者之间没有直接的联系，发送者只需要知道如何发送请求命令即可，其余的可以一概不管，甚至命令是否成功都无需关心。同时我们可以非常方便的增加新的命令，但是可能就是因为方便和对请求的封装就会导致系统中会存在过多的具体命令类。

### 16.2 UML类图



### 16.3模式结构

命令模式包含如下角色：  
Command: 抽象命令类  
ConcreteCommand: 具体命令类  
Invoker: 调用者  
Receiver: 接收者  
Client:客户类

### 16.4小栗子

#### 小栗子-01

命令模式是将请求以命令的形式包裹在对象中，并传递给对象，调用对象寻找到处理该命令的合适的对象，并将该命令传递给相应的对象，该对象执行。简单点说就是不同请求都封装成一个对象，不同的请求调用不同的执行者。



## 迭代器模式

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

### 17.1定义

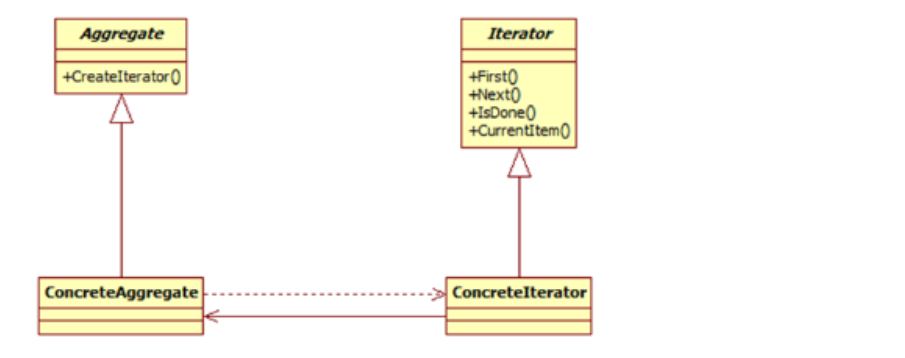
对于迭代在编程过程中我们经常用到，能够游走于聚合内的每一个元素，同时还可以提供多种不同的遍历方式，这就是迭代器模式的设计动机。在我们实际的开发过程中，我们可能会需要根据不同的需求以不同的方式来遍历整个对象，但是我们又不希望在聚合对象的抽象接口中充斥着各种不同的遍历操作，于是我们就希望有某个东西能够以多种不同的方式来遍历一个聚合对象，这时迭代器模式出现了。

何为迭代器模式？所谓迭代器模式就是提供一种方法顺序访问一个聚合对象中的各个元素，而不是暴露其内部的表示。迭代器模式是将迭代元素的责任交给迭代器，而不是聚合对象，我们甚至在不需要知道该聚合对象的内部结构就可以实现该聚合对象的迭代。

通过迭代器模式，使得聚合对象的结构更加简单，它不需要关注它元素的遍历，只需要专注它应该专注的事情，这样就更加符合单一职责原则了。

提供一个方法，可以顺序访问一个对象内部的各个元素，不需要知道内部构造。现在基本很少自己实现迭代器了，基本成熟的框架或者强大的JDK都会给出访问的方法，比如说java中iterator。这样做主要是进一步封装对象内部的结构，让行为和结构想耦合。这个不举例子了，用过iterator这个的小伙伴应该都清楚，就是不停的next，去访问下一个元素。

### 17.2 UML类图



### 17.3模式结构

迭代器模式包含如下角色：  
Iterator: 抽象迭代器  
ConcreteIterator: 具体迭代器  
Aggregate: 抽象聚合类  
ConcreteAggregate: 具体聚合类

### 17.4小栗子

#### 小栗子-01

## 中介者模式

中介者模式：用一个中介对象来封装一系列的对象交互。

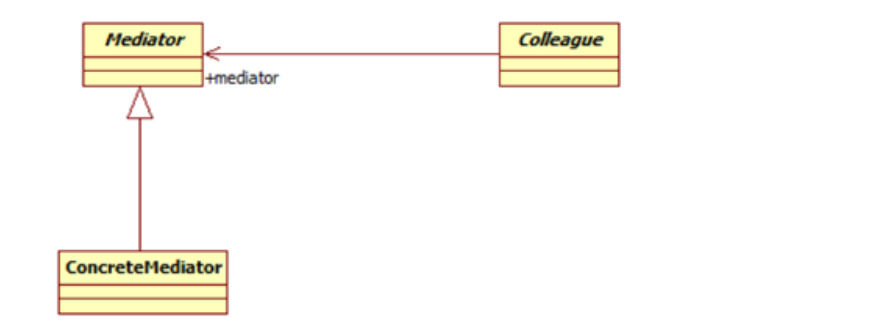
### 18.1定义

租房各位都有过的经历吧！在这个过程中中介结构扮演着很重要的角色，它在这里起到一个中间者的作用，给我们和房主互相传递信息。在外面软件的世界里同样需要这样一个中间者。在我们的系统中有时候会存在着对象与对象之间存在着很强、复杂的关联关系，如果让他们之间有直接的联系的话，必定会导致整个系统变得非常复杂，而且可扩展性很差！在前面我们就知道如果两个类之间没有不必彼此通信，我们就不应该让他们有直接的关联关系，如果实在是需要通信的话，我们可以通过第三者来转发他们的请求。同样，这里我们利用中介者来解决这个问题。

所谓中介者模式就是用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。在中介者模式中，中介对象用来封装对象之间的关系，各个对象可以不需要知道具体的信息通过中介者对象就可以实现相互通信。它减少了对象之间的互相关系，提供了系统可复用性，简化了系统的结构。

 在中介者模式中，各个对象不需要互相知道了解，他们只需要知道中介者对象即可，但是中介者对象就必须要知道所有的对象和他们之间的关联关系，正是因为这样就导致了中介者对象的结构过于复杂，承担了过多的职责，同时它也是整个系统的核心所在，它有问题将会导致整个系统的问题。所以如果在系统的设计过程中如果出现“多对多”的复杂关系群时，千万别急着使用中介者模式，而是要仔细思考是不是您设计的系统存在问题。

### 18.2 UML类图



### 18.3模式结构

Mediator: 抽象中介者  
ConcreteMediator: 具体中介者  
Colleague: 抽象同事类  
ConcreteColleague: 具体同事类

### 18.4小栗子

#### 小栗子-01

降低对象或者说事物之间通讯的复杂性，降低耦合。比如说分布式系统中，不是需要实时反馈的操作，我们无需直接对接，只需将想做的事告诉中间件，中间件告诉另外一个系统。比如说，访问（用户点击）一条新闻操作，同时需要记录是谁访问了什么新闻，同时给新闻浏览次数加1，还要实时更新用户喜好...总之要更新n个数据库表，甚至还要操作像ES,Mongo等多个中间件数据。但是对于用户来说，我只是做了一个点击操作，希望得到的结果就是看条新闻啊，你这么多操作，搞这么慢，用户体验很差啊，而且并发量也很低，那不如做成两个小系统，A系统，拉取新闻，推送，并组装一个信息扔给MQ中间件，ok，结束，用户看到新闻。然后B系统监听，得到这个消息，进行各种更新，这里，这个中间件就是我们的中介。再比如说，MVC中的控制层就是展示层和模型层的中介。再比如说，下面这个聊天室：



## 观察者模式

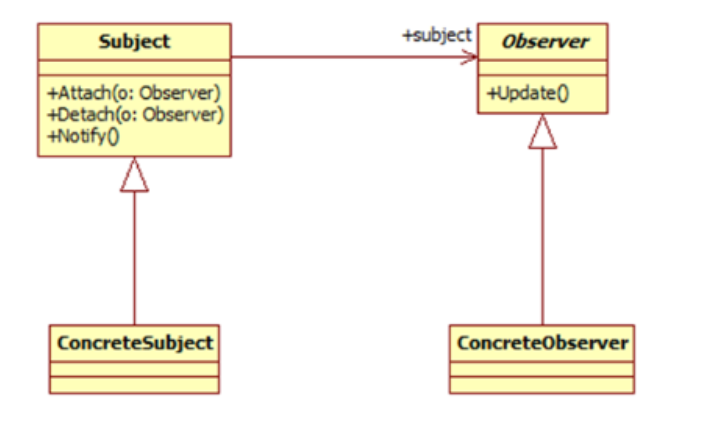
观察者模式：对象间的一对多的依赖关系。

### 19.1定义

何谓观察者模式？观察者模式定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

在这里，发生改变的对象称之为观察目标，而被通知的对象称之为观察者。一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，所以么可以根据需要增加和删除观察者，使得系统更易于扩展。所以观察者提供了一种对象设计，让主题和观察者之间以松耦合的方式结合。

### 19.2 UML类图



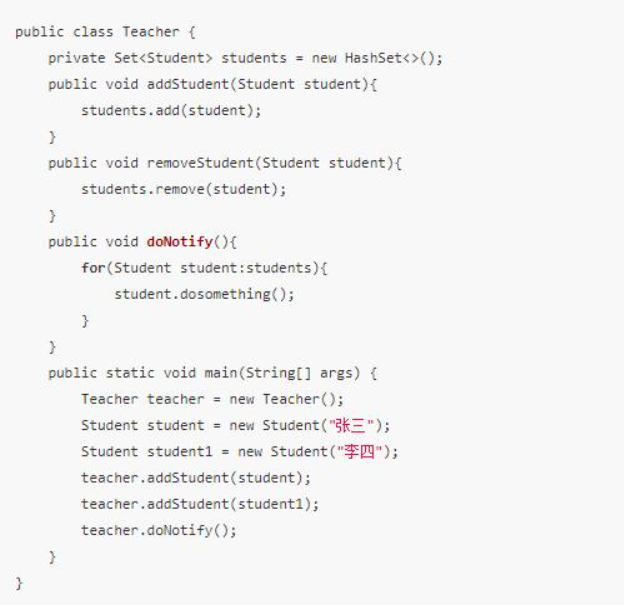
### 19.3模式结构

 观察者模式包含如下角色：  
Subject: 目标  
ConcreteSubject: 具体目标  
Observer: 观察者  
ConcreteObserver: 具体观察者

### 19.4小栗子

#### 小栗子-01

定义了一种一对多的依赖关系，当一个对象（被观察者）状态改变的时候，所有依赖于该对象的观察者都会被通知，从而进行相关操作。很多中间件都依赖于观察者模式，例如RabbitMQ,还有那些事件驱动模型（好像node就是）。下面举个例子，被观察者是监考老师，考试时间结束，通知所有观察者学生上交试卷。Java学习圈子



## 状态模式

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

### 20.1定义

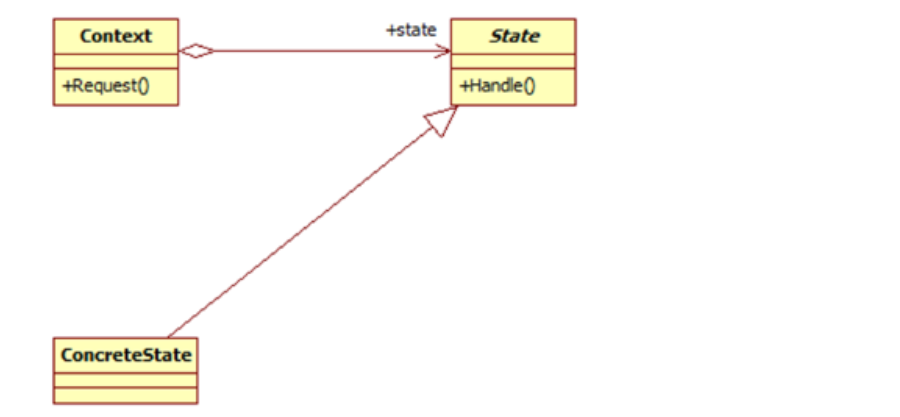
在很多情况下我们对象的行为依赖于它的一个或者多个变化的属性，这些可变的属性我们称之为状态，也就是说行为依赖状态，即当该对象因为在外部的互动而导致他的状态发生变化，从而它的行为也会做出相应的变化。对于这种情况，我们是不能用行为来控制状态的变化，而应该站在状态的角度来思考行为，即是什么状态就要做出什么样的行为。这个就是状态模式。

所以状态模式就是允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

在状态模式中我们可以减少大块的if…else语句，它是允许态转换逻辑与状态对象合成一体，但是减少if…else语句的代价就是会换来大量的类，所以状态模式势必会增加系统中类或者对象的个数。

同时状态模式是将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为。但是这样就会导致系统的结构和实现都会比较复杂，如果使用不当就会导致程序的结构和代码混乱，不利于维护。

### 20.2 UML类图



### 20.3 模式结构

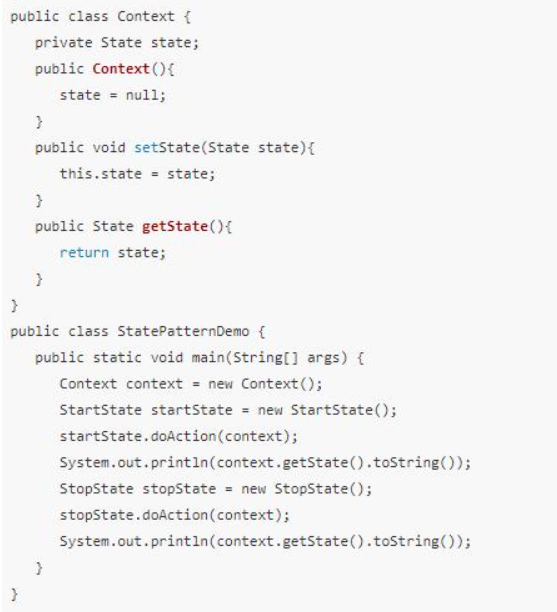
 状态模式包含如下角色：  
Context: 环境类  
State: 抽象状态类  
ConcreteState: 具体状态类

### 20.4小栗子

#### 小栗子-01

简单来说，就是一个对象有不同的状态，根据状态不同，可能有不同的行为。





## 空对象模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 策略模式

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

### 22.1定义

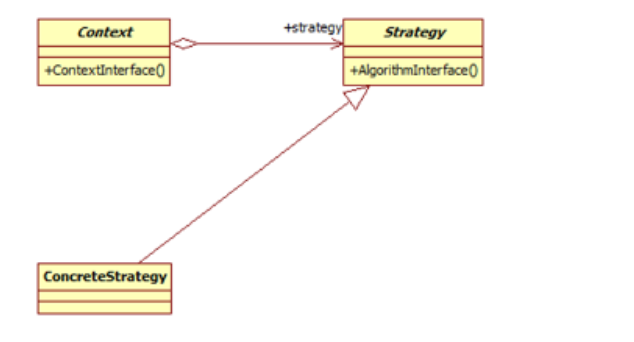
我们知道一件事可能会有很多种方式来实现它，但是其中总有一种最高效的方式，在软件开发的世界里面同样如此，我们也有很多中方法来实现一个功能，但是我们需要一种简单、高效的方式来实现它，使得系统能够非常灵活，这就是策略模式。

所以策略模式就是定义了算法族，分别封装起来，让他们之前可以互相转换，此模式然该算法的变化独立于使用算法的客户。

在策略模式中它将这些解决问题的方法定义成一个算法群，每一个方法都对应着一个具体的算法，这里的一个算法我就称之为一个策略。虽然策略模式定义了算法，但是它并不提供算法的选择，即什么算法对于什么问题最合适这是策略模式所不关心的，所以对于策略的选择还是要客户端来做。客户必须要清楚的知道每个算法之间的区别和在什么时候什么地方使用什么策略是最合适的，这样就增加客户端的负担。

同时策略模式也非常完美的符合了“开闭原则”，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。但是一个策略对应一个类将会是系统产生很多的策略类。

### 22.2 UML类图



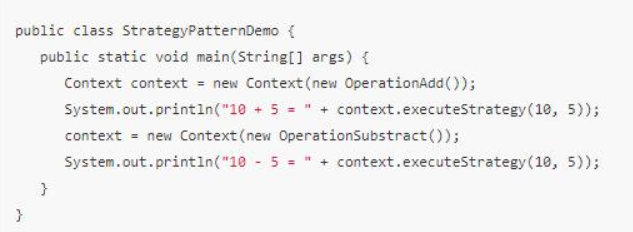
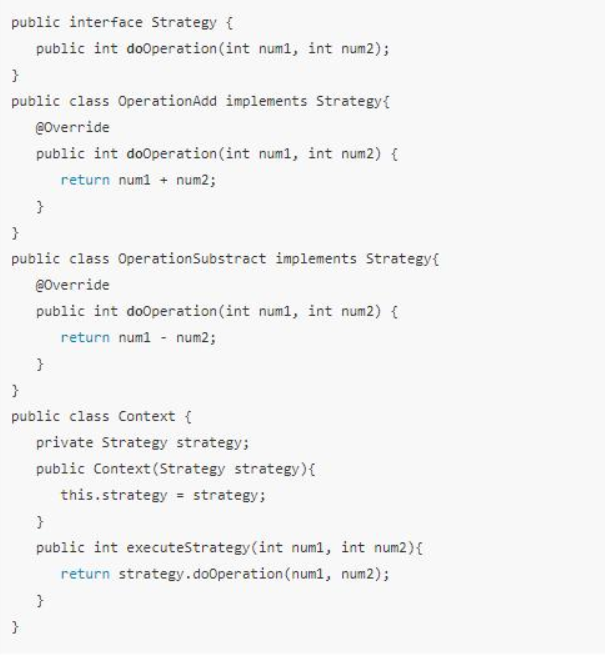
### 22.3模式结构

策略模式包含如下角色：  
Context: 环境类  
Strategy: 抽象策略类  
ConcreteStrategy: 具体策略类

### 22.4小栗子

#### 小栗子-01

定义一组算法, 将每个算法都封装起来, 并且使它们之间可以互换。可以说是一组算法的封装，根据客户端给出的不同要求，进行不同的运算。比如下面这个简易计算器。



## 模板模式

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

### 23.1定义

有些时候我们做某几件事情的步骤都差不多，仅有那么一小点的不同，在软件开发的世界里同样如此，如果我们都将这些步骤都一一做的话，费时费力不讨好。所以我们可以将这些步骤分解、封装起来，然后利用继承的方式来继承即可，当然不同的可以自己重写实现嘛！这就是模板方法模式提供的解决方案。

所谓模板方法模式就是在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

模板方法模式就是基于继承的代码复用技术的。在模板方法模式中，我们可以将相同部分的代码放在父类中，而将不同的代码放入不同的子类中。也就是说我们需要声明一个抽象的父类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法让子类来实现剩余的逻辑，不同的子类可以以不同的方式来实现这些逻辑。所以模板方法的模板其实就是一个普通的方法，只不过这个方法是将算法实现的步骤封装起来的。

### 23.2 UML类图



### 23.3 模式结构

模板方法模式包含如下角色：  
AbstractClass: 抽象类   
ConcreteClass: 具体子类

### 23.4 小栗子

#### 小栗子-01

一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。SpringBoot为用户封装了很多继承代码，都用到了模板方式，例如那一堆XXXtemplate。



## 访问者模式

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

意图：主要将数据结构与数据操作分离。

主要解决：稳定的数据结构和易变的操作耦合问题。

### 25.1定义

访问者模式俗称23大设计模式中最难的一个。除了结构复杂外，理解也比较难。

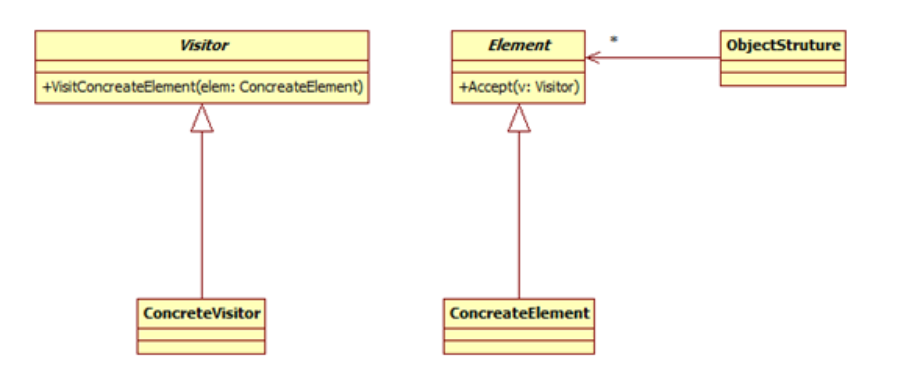
在我们软件开发中我们可能会对同一个对象有不同的处理，如果我们都做分别的处理，将会产生灾难性的错误。对于这种问题，访问者模式提供了比较好的解决方案。

访问者模式即表示一个作用于某对象结构中的各元素的操作，它使我们可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

访问者模式的目的是封装一些施加于某种数据结构元素之上的操作，一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。为不同类型的元素提供多种访问操作方式，且可以在不修改原有系统的情况下增加新的操作方式。同时我们还需要明确一点那就是访问者模式是适用于那些数据结构比较稳定的，因为他是将数据的操作与数据结构进行分离了，如果某个系统的数据结构相对稳定，但是操作算法易于变化的话，就比较适用适用访问者模式，因为访问者模式使得算法操作的增加变得比较简单了。

|  |
| --- |
| 当对特定角色进行访问的时候，需要通过访问者进行访问。一个对象不太方便被你直接访问的时候，你需要将自己的引用交给访问者，通过访问者去访问该对象。比如说，化学课，想看一个细胞结构，由于肉眼无法直接看到微观世界的玩意，需要通过显微镜间接访问。 |

### 25.2 UML类图



|  |
| --- |
| 介绍 **意图：**主要将数据结构与数据操作分离。  **主要解决：**稳定的数据结构和易变的操作耦合问题。  **何时使用：**需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，使用访问者模式将这些封装到类中。  **如何解决：**在被访问的类里面加一个对外提供接待访问者的接口。  **关键代码：**在数据基础类里面有一个方法接受访问者，将自身引用传入访问者。  **应用实例：**您在朋友家做客，您是访问者，朋友接受您的访问，您通过朋友的描述，然后对朋友的描述做出一个判断，这就是访问者模式。**拦截器与过滤器**  **优点：** 1、符合单一职责原则。 2、优秀的扩展性。 3、灵活性。  **缺点：** 1、具体元素对访问者公布细节，违反了迪米特原则。 2、具体元素变更比较困难。 3、违反了依赖倒置原则，依赖了具体类，没有依赖抽象。  **使用场景：** 1、对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作。 2、需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。  **注意事项：**访问者可以对功能进行统一，可以做报表、UI、**拦截器与过滤器**。 |

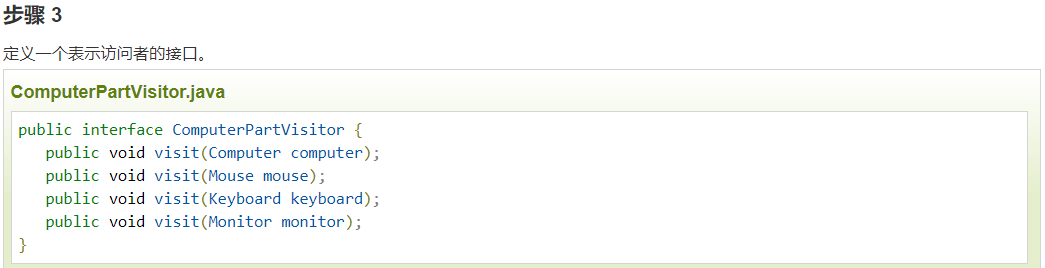
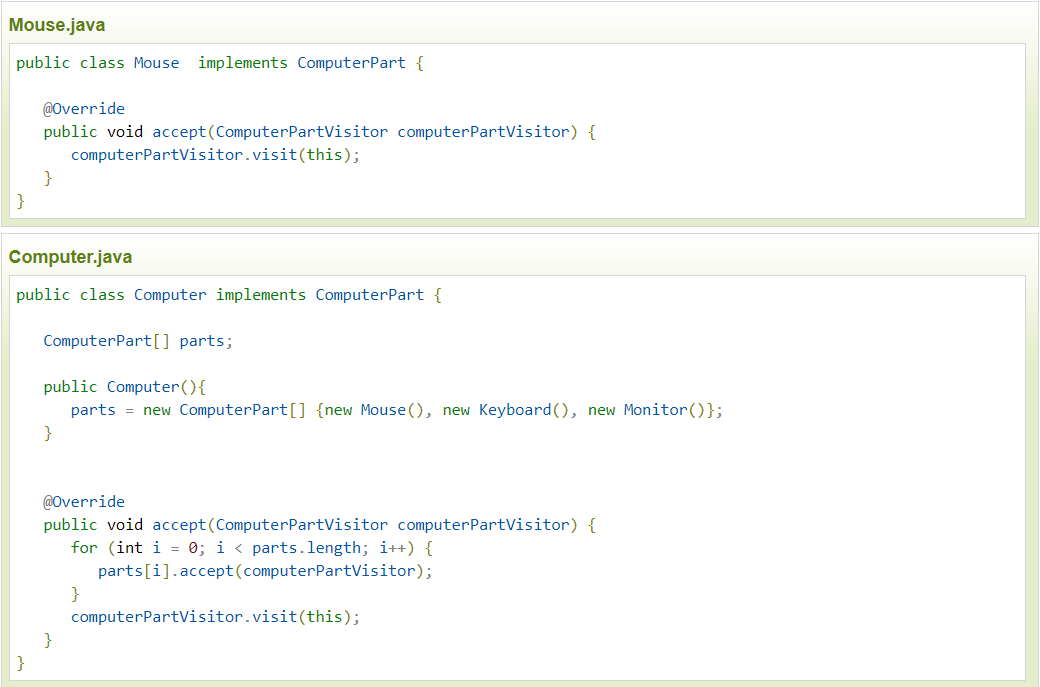
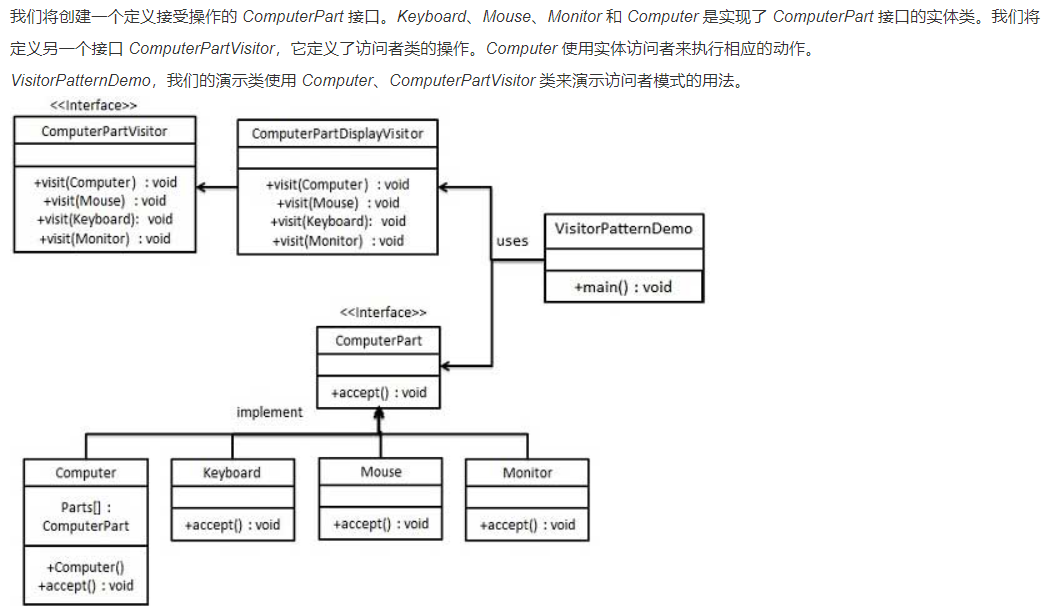
### 25.3 模式结构

访问者模式包含如下角色：  
Vistor: 抽象访问者  
ConcreteVisitor: 具体访问者

Element: 抽象元素  
ConcreteElement: 具体元素   
ObjectStructure: 对象结构

### 25.4 小栗子

#### 小栗子-01



## MVC模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 业务代表模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 组合实体模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 数据访问对象模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 前端控制器模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 拦截过滤器模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 服务定位器模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 传输对象模式

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01

## 33.解释器模式

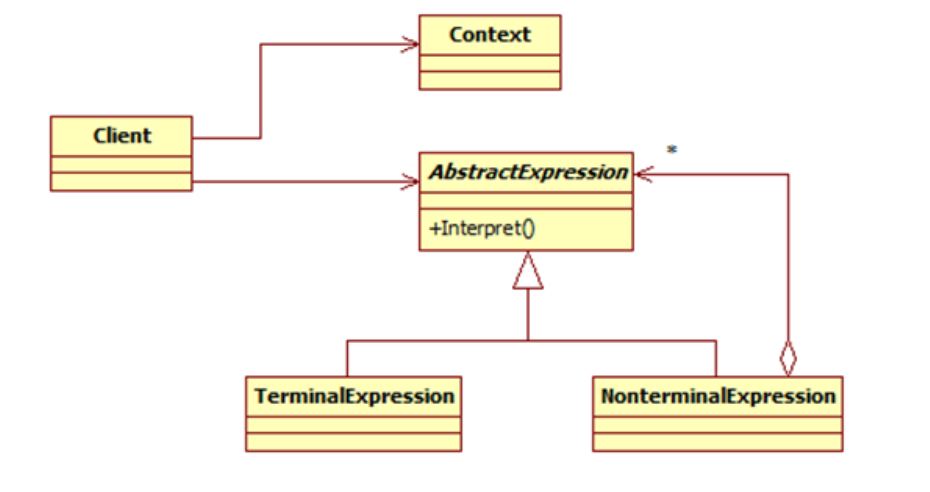
解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

### 33.1定义

所谓解释器模式就是定义语言的文法，并且建立一个解释器来解释该语言中的句子。解释器模式描述了如何构成一个简单的语言解释器，主要应用在使用面向对象语言开发的编译器中。它描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。

|  |
| --- |
| 构建一种翻译方式，将某种语言或描述翻译成我们很好理解的语言或者描述。这里很好理解的意思是看得懂，看的快。本来我也想举什么编译器这种高大上的，将底层语言甚至机械语言和我们使用的高级编程语言。后来想了想，其实Map就可以看作一个很好的编译器，key你可以存放一个非常小的字符串，value理论上你可以存放任何东西，所以代码就不写了。 |

### 33.2UML类图



### 33.3模式结构

解释器模式包含如下角色：  
AbstractExpression: 抽象表达式  
TerminalExpression: 终结符表达式  
NonterminalExpression: 非终结符表达式  
Context: 环境类  
Client: 客户类

### 33.4小栗子

#### 小栗子-01

## 34.备忘录模式

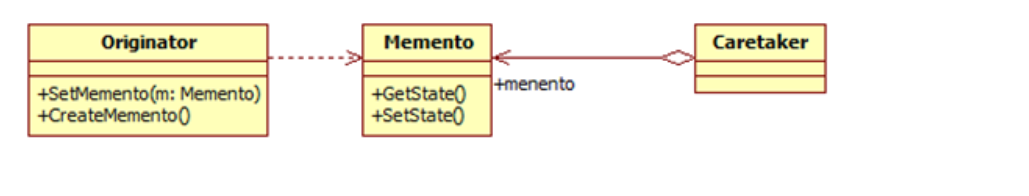
备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

### 34.1定义

后悔药人人都想要，但是事实却是残酷的，根本就没有后悔药可买，但是也不仅如此，在软件的世界里就有后悔药！备忘录模式就是一种后悔药，它给我们的软件提供后悔药的机制，通过它可以使系统恢复到某一特定的历史状态。

所谓备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。它实现了对信息的封装，使得客户不需要关心状态保存的细节。保存就要消耗资源，所以备忘录模式的缺点就在于消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

### 34.2 UML类图



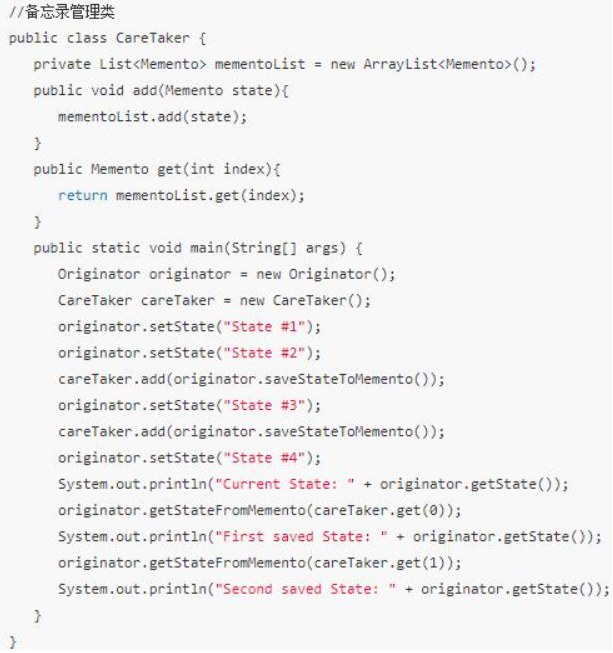
### 34.3模式结构

备忘录模式包含如下角色：  
Originator: 原发器  
Memento: 备忘录  
Caretaker: 负责人

### 34.4小栗子

#### 小栗子-01

相当于做一个快照，在不破坏对象本身结构的情况下，记录对象的一个状态，合适的时候可以恢复到这种状态。数据库做事务回滚的时候就用了这种方式。这里需要注意的是，对象不与备忘录本身耦合，而是跟备忘录管理类耦合（就是List<备忘录>），这个好理解，毕竟快照不止一个嘛。Java学习圈子



## 35.设计模式资源

### 定义

### UML类图

### 模式结构

### 小栗子

#### 小栗子-01