Java 设计模式

# Part 1. 设计模式起源和诞生

## 定义

设计模式(Design Patterns)是一套被反复使用、多数人知晓、经过分类的、代码设计经验的总结。

为了代码可重用性，让代码更容易被他人理解，保证代码可靠性。设计模式使代码编写真正工程化；设计模式是软件工程的及时脉络，如同大厦的结果一样。

## 1.2 起源

与很多软件工程技术一样，模式起源于建筑领域，软件工程只有短短的几十年，已经拥有几千年的底蕴的建筑工程相比，后者有怠惰指的学习和借鉴的地方。

哈佛大学的建筑学欧式克里斯托弗▪亚历山大，是建筑学领域的模式之父。他与其研究团队用了约20年的时间，对住宅和周边环境进行了大量的调查研究，发现人们对舒适住宅和城市华景存在一些共同的认同规律，将它们归纳成253个模式。对每一个模式都从前提条件、目标问题、解决方法三个方面进行了描述，并给出了从徐秀分析到结构设计再到经典实例的过程模型。

所以，对模式的定义可以抽象为在特定环境下，人们为了解决某类重复出现问题二总结归纳出来的有效解决方案。

## 诞生

GoF将模式的概念引入软件工程领域，这标志着软件模式的诞生。软件模式并非仅限于设计模式，还包括架构模式、分析模式和过程模式等，实际上，在软件开发生命周期的每一个阶段都存在杂合一些被认同的模式。

软件模式主要由四部分构成，包括待解决问题、约束条件、解决方案、优点。

软件模式与具体的应用领域无关，也就是是说无论从事的是移动开发、桌面开发、WEB开发还是嵌入式软件的开发，都可以使用软件模式。

在软件模式中，设计模式是研究最为深入的分支，它融合了众多专家的设计经验，已经在成千上万的软件中得到应用。1995年，GoF将收集和整理好的23中设计模式会变成了一本名叫《设计模式》的书，该书的出版也标志着设计模式正式成为面向对象开发的一个重要研究分支。

## 设计模式分类

总体来说设计模式分为三大类：

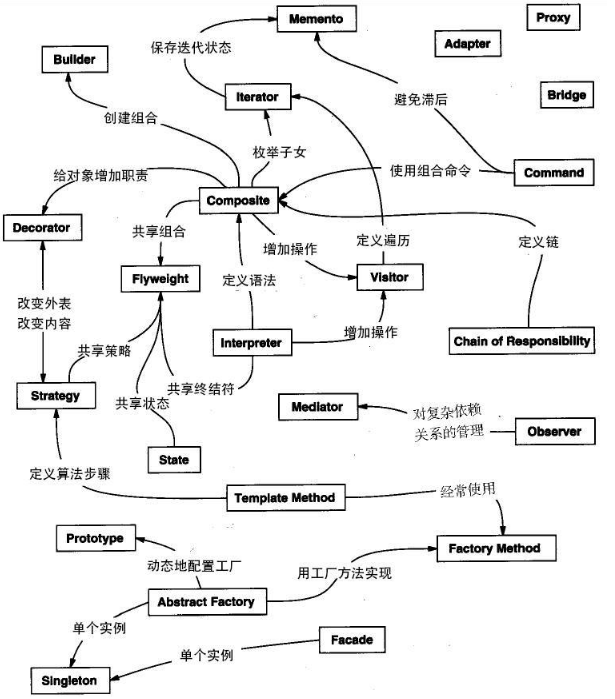
创建者模式：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式；

结构型模式：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式；

行为型模式：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介模式、解释其模式；

其他模式：并发型模式、线程池模式。

如下图所示设计模式之间的关系：



# Part 2. 设计模式的六大原则

* 单一职责原则(Single responsibility principle)
* 里氏替换原则(Liskov substitution principle)
* 依赖倒转原则(Dependency inversion principle)
* 接口隔离原则(Interface segregation principle)
* 迪米特法则(又称为最少知道原则，Demeter principle)
* 开闭原则(Open-Close principle)

## 2.1 单一职责原则(Single responsibility principle)

一个类，只有一个引起它变化的原因。应该只有一个职责。每一个职责都是变化的一个轴线，如果一个类有一个以上的职责，这些职责就耦合在了一起。这回导致脆弱的设计。当一个职责放生变化时，可能会影响其他的职责。另外，多个职责耦合在一起，会影响复用性。例如:要实现逻辑和界面的分离。

## 2.2 里氏替换原则(Liskov substitution principle)

在面向对象的程序设计中，里氏替换原则(Liskov Substitution principle)是对子类型的特别定义。它由芭芭拉▪利斯科夫(Barbara Liskov)在1987年在一次会议上名为“数据的抽象与层次”的演说中首次提出。

里氏替换原则的内容可以描述为：派生类(子类)对象能够替换其基类(超类)对象被使用。以上内容有罗伯特▪马丁(Robert Martin)对原文的解读。

## 2.3 依赖倒转原则(Dependency inversion principle)

在面向对象编程领域中依赖反转原则(Dependency inversion principle)是一种特定的解耦(传统的依赖关系创建在高层次上，而具体的策略设置则应用在低层次的模块上)形式，使得高层次的模块不依赖于低层次的模块的实现细节，依赖关系呗颠倒(反转)，从而使得低层次模块依赖于高层次模块的需求抽象。

该原则规定：

* 高层次的模块不应该依赖于低层次的模块，两者都应该依赖于抽象接口
* 抽象接口不应该依赖于具体实现。而具体实现则应该依赖于抽象接口。

该原则颠倒了一部分人对于面向对象设计的认识方式。如高层次和低层次对象都应该依赖于相同的抽象接口，如下图所示：

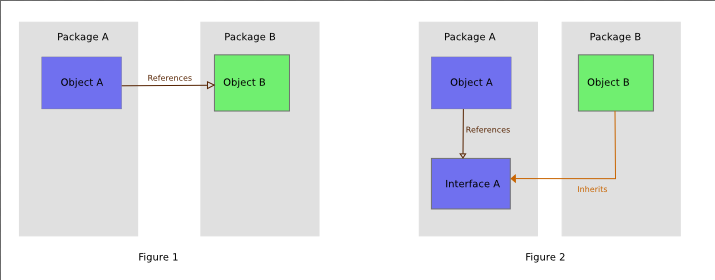


图1中，高层对象A依赖于低层对象B的实现；图2中，把高层对象A对象对低层对象的需求抽象为一个接口A，低层对象B实现了接口A，这就是依赖反转。

## 2.4 接口隔离原则(interface-segregation principles)

接口隔离(ISP)指明客户(client)应该不依赖于它不使用的方法。接口隔离原则拆分非常庞大臃肿的接口为更小和更具体的接口，这样客户将会只需要知道他们感兴趣的方法。这种缩小的接口也被称为角色接口(role interfaces)。接口隔离原则的目的是系统解开耦合，从而容易重构，、更改和重新部署。接口隔离原则是在SOLID(面向对象设计)中五个面向对象设计(OOD)的原则之一，类似于在GRASP(面向对象设计)中的高内聚性。

## 2.5 迪米特法则

一个对象应该对其他对象报纸最少的了解。

类与类之间的关系月密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

迪米特法则又称为最少知道原则，最早是在1987年由美国Northeaastern University的Ian Holland提出。通俗来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：只与直接的朋友通信。首先来解释一下什么是直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，李瑶、关联、组合、聚合等。其中我们称出现成员变量、方法参数、方法返回值中的类为直接朋友，而出现在局部变量中的类则不是直接朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

## 2.6 开闭原则(Open-Close principle)

在面向对象编程领域中，开闭原则规定“软件中的对象(类、模块、函数等)应该对于扩展是开放的，但是对于修改是封闭的，这意味着一个实体是允许在不改变它的源代码的前提下变更它的行为。该特性在产品化的环境中是特别有价值的，在这种环境中，改变源代码需要代码审查、单元测试以及诸如此类的用法确保产品使用质量的过程。遵循这种原则的代码在扩展时并不发生改变，因此无需上述的过程。

开闭原则的命名被应用在两种方式上。这两种方式都使用了继承来解决明显的困境，但是他们的目的，技术以及结果是不同的。

### 2.6.1 梅耶开闭原则

波特兰▪迈耶一般被认为是最早提出开闭原则这一术语的人，在他1988年发行的《面向对象软件构造》中提出。这一想法认为一旦完成，一个类的实现只应该因错误而修改，新的或改变的特性应该通过新建不同的类实现。新建的类可以通过集成的方式来重用原类的代码。衍生的子类可以或不可以拥有和猿类相同的接口。

梅耶的定义提倡实现继承。具体实现可以通过继承方式来重用，但是接口规格不必如此。已存在的实现对于修改是封闭的，但是新的实现不必实现原有接口。

### 2.6.2 多态开闭原则

在20世纪90年代，开闭原则被广泛的重新定义由于抽象化接口的使用，在这中间实现可以被改变，多种实现可以被创建，并且多态化的替换不同的实现。

相比梅耶的使用方式，多态开闭原则的定义倡导对抽象基类的继承。接口规定可以通过继承来重用，但是实现不必重用。已存在的接口对于修改是封闭的，并且新的实现必须、至少、实现那个接口。

罗伯特▪ C ▪马丁1996年发表的文章《开闭原则》是使用这种方法的启发式著作。在2001年，Craig Larman把开闭原则关联到了Alistair Cockburn的名为受护的变量的模式以及David Parnas关于信息隐藏的讨论。

# Part 3. 23种设计模式

## 3.1 创建型模式

### 3.1.1 抽象工厂模式(Abstract Factory)

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

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

#### 3.1.1.1 介绍

**意图：**提供一个创建一些列相关或相互依赖对象的接口，而无需指定他们具体的类；

**主要解决：**主要解决接口选择的问题；

**何时使用：**系统的产品有多于一个的产品族，而系统只消费其中某一族产品；

**如何解决：**在一个产品族里面，定义多个产品；

**关键代码：**在一个工厂里聚合多种同类产品；

**应用实例：**

具体产品：商务装、时尚装，商务男装、商务女装，时尚男装、时尚女装，上装、下装

工厂：每个衣柜(具体工厂)只能存放一种服装，所有的衣柜(抽象工厂)抽象为抽象工厂

**优点：**当一个产品族中的多个对象被设计成一起工作时，他能保证客户端是中只是用同一个产品族中的对象。

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

**使用场景：**

▪ QQ换皮肤，一整套一起换；

▪ 生成不同操作系统的程序。

**实现：**

类图如下所示：





代码实现：<https://github.com/CoderFreeMan/abstract_factory.git>

### 3.1.2 建造模式(Builder Pattern)

生成器模式(Builder Pattern)是一种创建型设计模式，又名：建造模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来(抽象类别)，使这个抽象过程的不同实现方法可以构造出不同(属性)的对象。

#### 3.1.2.1 适用性

在以下情况使用生成器模式：

▪ 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时；

▪ 当构造过程必须允许被构造的对象有不同的表示时。

#### 3.1.2.2 组成部分

▪ Builder：为创建一个Product对象的各个部件指定抽象接口；

▪ ConcreteBuilder：实现Builder的接口以构造和装配该产品的各个部件；定义并明确它所创建的表示；提供一个检索产品的接口；

▪ Director：构造一个使用Builder接口的对象；

▪ Product：表示被构造的复杂对象。ConcreateBuilder创建该产品的内部表示并定义它的装配过程；包含定义组成部件的类，包括将这些部件装配成最终产品的接口。

#### 3.1.2.3 协作

客户创建Director对象，并用它所想要的Builder对象进行配置

▪ 一旦产品部件被生成，导向器就会通知生成器；

▪ 生成器处理导向器的请求，并将部件添加到该产品中；

▪ 客户从生成器中检索产品。

### 3.1.3 工厂方法模式(Factory Method pattern)

工厂方法模式是Java中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

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

#### 3.1.3.1 介绍

**意图：**定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过长延迟到子类进行；

**主要解决：**主要解决接口选择的问题；

**何时使用：**我们明确地计划不同条件下创建不同实例时；

**如何解决：**让其子类实现工厂接口，返回的也是一个抽象的产品；

**关键代码：**创建过程在其子类执行；

**应用实例：**

▪ 你需要一辆车，可以直接从工厂里边提货，而不用去关心这辆汽车是怎么造出来的，以及这个汽车里面的具体实现；

▪ Hibernate 换数据库只需换方言和驱动就可以。

**优点：**

▪ 一个调用者想创建一个对象，只要知道其名称就可以了；

▪ 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以；

▪ 屏蔽产品的具体实现，调用者只关心产品的接口。

**缺点：**每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。

**使用场景：**

▪ 日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方；

▪ 数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据可能有变化时；

▪ 设计一个连接服务器的框架，需要三个协议，“POP3”、“IMAP”、“HTTP”，可以把这三个座位产品类，共同实现一个接口。

**注意事项：**座位一种创建模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是负载对象适合使用工厂模式，而简单对象，特别是只需要通过new就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

#### 3.1.3.2实现

类图如下所示：

IHuman

+cry():void

<<Interface>>

YellowHuman

+cry():void

BlackHuman

+cry():void

implements

implements

HumanFactory

+getIHuman():IHuman

creates

**代码实现：**[**https://github.com/CoderFreeMan/simple\_factory.git**](https://github.com/CoderFreeMan/simple_factory.git)

### 3.1.4 原型模式(Prototype pattern)

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

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

#### 3.1.4.1 介绍

**意图：**用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象；

**主要解决：**在运行期间建立和删除原型；

**何时使用：**

▪ 当一个系统应用该独立于它的产品创建，构成和表示时；

▪ 当要实例化的类是在运行时刻指定时，例如，通过动态装载；

▪ 为了避免创建一个与产品类层次平行的工厂类层次时；

▪ 当一个类的实例只能有几个不同状态组合中的一种时，建立相应数目的原型并克隆他们可能比每次用合适的窗台手工实例化该类更方便一些。

**如何解决：**利用已有的一个圆形对象，快速地生成和圆心对象一样的实例；

**关键代码：**

▪ 实现克隆操作，在Java继承Cloneable，重写clone()；

▪ 原型模式同样用于隔离类对象的使用和具体类型(易变类)之间的耦合关系，它同样要求这些“易变类”拥有稳定的接口。

**应用实例：**

▪ 细胞分裂；

▪ Java中的Object clone(0方法；

**优点：**

▪ 提高性能；

▪ 逃避构造方法的约束；

**缺点：**

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

▪ 必须实现Cloneable接口

**使用场景：**

▪ 资源优化场景；

▪ 类初始化需要消耗非常多的资源，这个资源包括数据、硬件资源等；

▪ 性能和安全要求的场景；

▪ 通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式；

▪ 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值是，可以考虑使用原型模式拷贝多个对象供调用者使用；

▪ 在实际项目中，原型模式很是单独出现，一般是和工厂方法模式一起出现，通过clone的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与Java融为浑然一体，大家可以随手拿来使用；

**注意事项：**

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝事项Cloneable，重写，深拷贝是通过事项Serializable读取二进制流。

#### 3.1.4.2 实现

<https://github.com/CoderFreeMan/prototype_pattern.git>

### 3.1.5 单例模式(Singleton pattern)

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

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

注意：

● 单例类只能有一个实例(线程安全的)；

● 单例类必须自己创建自己的唯一实例；

● 单例类必须给所有其他对象提供这一范例。

#### 3.1.5.1 介绍

**意图：**保证一个类仅有一个实例，并提供一个访问它的全局访问点。

**主要解决问题：**一个全局使用的类频繁地创建于销毁。

**如何使用：**控制实例数目，节省系统资源。

**解决方法：**判断系统是否已经有这个实例，如果有则返回，如果没有则创建。

**关键代码：**私有的构造方法。

**应用实例：**

▪ 一个国家只能有一个主席；

▪ 文件处理时通过唯一的实例来进行操作；

▪ 一台电脑不能让两台打印机打印同一个文件。

**优点：**

**▪** 内存中只有一个实例，节约内存开销，尤其是频繁的创建和销毁的实例；

▪ 避免对资源的多重占用(比如文件操作)。

**缺点：**

**▪** 没有借口，不能被继承，与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外部是怎样来实例化。

**使用场景：**

**▪** 生产唯一序列号；

▪ WEB中的计数器，不用每次都在数据库里+1，用单例先缓存起来；

▪ 创建的一个对象需要消耗的资源过多，比如I/O与数据库的连接等。

**注意事项：**

getInstance()方法中需要使用使用同步锁synchronized(Singleton.class)防止多线程同时进入造成instance被多次实例化。

#### 3.1.5.2 实现

类图如下所示：

SingletonPatternDemo

+main

SingleObject

-instance:SingleObject

-SingleObject

+getInstance():SingleObject

+showMessage():void

asks

returns

代码实例：<https://github.com/CoderFreeMan/singleton_pattern.git>

## 3.2 结构型模式

### 3.2.1 适配器模式(Adapter pattern)

### 3.2.2 桥接模式(Bridge pattern)

### 3.2.3 组合模式(Composite pattern)

### 3.2.4 装饰模式(Decorator pattern)

### 3.2.5 外观模式(Façade pattern)

### 3.2.6 享元模式(Flyweight pattern)

### 3.2.7 代理模式(Proxy pattern)

## 3.3 行为型模式

### 3.3.1 责任链模式(Chain-of-responsibility pattern)

### 3.3.2 命令模式(Command pattern)

### 3.3.3 翻译模式(Interpreter pattern)

### 3.3.4 迭代器模式(Iterator pattern)

### 3.3.5 仲裁器模式(Mediator pattern)

### 3.3.6 回忆模式(Memento pattern)

### 3.3.7 观察者模式(Observer pattern)

### 3.3.8 状态机模式(State pattern)

### 3.3.9 策略模式(Strategy pattern)

### 3.3.10 模板方法模式(Template method pattern)

### 3.3.11 参观者模式(Visitor)