# 设计模式

根据设计模式的参考书中所提到的，总共有 23 种设计模式。这些模式可以分为三大类：创建型模式（Creational Patterns）、结构型模式（Structural Patterns）、行为型模式（Behavioral Patterns）

## 创建型模式

这些设计模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用新的运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

* 工厂模式（Factory Pattern）
* 抽象工厂模式（Abstract Factory Pattern）
* 单例模式（Singleton Pattern）
* 建造者模式（Builder Pattern）
* 原型模式（Prototype Pattern）

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

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

单例模式：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。

建造者模式：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

原型模式：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。

## 结构型模式

这些设计模式关注类和对象的组合。继承的概念被用来组合接口和定义组合对象获得新功能的方式。

* 适配器模式（Adapter Pattern）
* 桥接模式（Bridge Pattern）
* 过滤器模式（Filter、Criteria Pattern）
* 组合模式（Composite Pattern）
* 装饰器模式（Decorator Pattern）
* 外观模式（Facade Pattern）
* 享元模式（Flyweight Pattern）
* 代理模式（Proxy Pattern）

适配器模式：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

桥接模式：将抽象部分与实现部分分离，使它们都可以独立的变化。

过滤器模式：这种模式允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来

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。组合模式使得用户对单个对象和组合对象的使用具有一致性。

装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

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

享元模式：运用共享技术有效地支持大量细粒度的对象。

代理模式：为其他对象提供一种代理以控制对这个对象的访问。

## **行为型模式**

这些设计模式特别关注对象之间的通信。

* 责任链模式（Chain of Responsibility Pattern）
* 命令模式（Command Pattern）
* 解释器模式（Interpreter Pattern）
* 迭代器模式（Iterator Pattern）
* 中介者模式（Mediator Pattern）
* 备忘录模式（Memento Pattern）
* 观察者模式（Observer Pattern）
* 状态模式（State Pattern）
* 空对象模式（Null Object Pattern）
* 策略模式（Strategy Pattern）
* 模板模式（Template Pattern）
* 访问者模式（Visitor Pattern）

责任链模式：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。

命令模式：将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。

解释器模式：给定一个语言，定义它的文法表示，并定义一个解释器，这个解释器使用该标识来解释语言中的句子。

迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

中介者模式：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。

备忘录模式：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。

状态模式：定义对象间的一种一对多的依赖关系，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。

空对象模式：在空对象模式中，我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

策略模式：定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。

模板模式：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

访问者模式：主要将数据结构与数据操作分离。

## **J2EE 模式**

这些设计模式特别关注表示层。这些模式是由 Sun Java Center 鉴定的。

* MVC 模式（MVC Pattern）
* 业务代表模式（Business Delegate Pattern）
* 组合实体模式（Composite Entity Pattern）
* 数据访问对象模式（Data Access Object Pattern）
* 前端控制器模式（Front Controller Pattern）
* 拦截过滤器模式（Intercepting Filter Pattern）
* 服务定位器模式（Service Locator Pattern）
* 传输对象模式（Transfer Object Pattern）

## **MVC模式：**

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

* **Model（模型）** - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。
* **View（视图）** - 视图代表模型包含的数据的可视化。
* **Controller（控制器）** - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

作者：lorio  
链接：https://www.zhihu.com/question/27738109/answer/100241918  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

GoF (Gang of Four，四人组， 《Design Patterns: Elements of Reusable Object-Oriented Software》/《设计模式》一书的作者：Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides)并没有把MVC提及为一种设计模式，而是把它当做“一组用于构建用户界面的类集合”。**在他们看来，MVC其实是其它三个经典的设计模式的演变：观察者模式(Observer)(Pub/Sub), 策略模式(Strategy)和组合模式(Composite)**。根据MVC在框架中的实现不同可能还会用到工厂模式(Factory)和装饰器(Decorator)模式。我在另一本免费的书“JavaScript Design Patterns For Beginners”中讲述了这些模式，如果你有兴趣可以阅读更多信息。

正如我们所讨论的，models表示应用的数据，而views处理屏幕上展现给用户的内容。为此，MVC在核心通讯上基于推送/订阅模型(惊讶的是 在很多关于MVC的文章中并没有提及到)。当一个model变化时它对应用其它模块发出更新通知(“publishes”)，订阅者 (subscriber)——通常是一个Controller，然后更新对应的view。观察者——这种自然的观察关系促进了多个view关联到同一个 model。

对于感兴趣的开发人员想更多的了解解耦性的MVC(根据不同的实现)，这种模式的目标之一就是在一个主题和它的观察者之间建立一对多的关系。当这个 主题改变的时候，它的观察者也会得到更新。Views和controllers的关系稍微有点不同。Controllers帮助views对不同用户的输 入做不同的响应，是一个非常好的策略模式列子。

## **前端控制器模式**

前端控制器模式（Front Controller Pattern）是用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

* **前端控制器（Front Controller）** - 处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 [web](http://caibaojian.com/c/web" \o "网站建设) 的应用程序，也可以是基于桌面的应用程序。
* **调度器（Dispatcher）** - 前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序。
* **视图（View）** - 视图是为请求而创建的对象。



## 设计模式的六大原则

**1、开闭原则（Open Close Principle）**

开闭原则的意思是：**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

**5、迪米特法则，又称最少知道原则（Demeter Principle）**

最少知道原则是指：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

**6、合成复用原则（Composite Reuse Principle）**

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承。