# 简介

**设计模式目的**：为了可重用代码，保证代码的可靠性，更容易被他人理解。

**设计模式的六大原则**：

总原则：开闭原则，即对扩展开放，对修改关闭。

1 单一职责原则：每个类应该实现单一的职责，否则应该把类拆分。

2 里氏替换原则：任何基类可以出现的地方，子类一定可以出现。它是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

3 依赖倒转原则：这是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。

4 接口隔离原则：使用多个隔离的接口，比使用单个接口要好。每个接口不存在子类用不到却必须实现的方法，否则要将接口拆分。

5 迪米特法则（最少知道原则）：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统的功能模块相对独立。

6 合成复用原则：尽量使用合成/聚合方式，而不是使用继承。

设计模式分为三大类：

**创建型模式**：（5）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 （简单工厂模式）

**结构型模式**：（7）代理模式 装饰器模式 适配器模式 外观模式 组合模式 享元模式 桥接模式

**行为型模式**：（11）观察者模式 责任链模式 模板方法模式 策略模式 迭代子模式 命令模式 状态模式 备忘录模式 访问者模式 中介者模式 解释器模式

其实还有两类：并发型模式和线程池模式

设计模式之间的关系见下图：



## 创建型模式

### 简单工厂模式

简单工厂模式并不属于23种模式中的一种，但是还是很有必要了解一下。

简单工厂模式：有一个工厂类，在工厂类中进行判断，创建需要的功能类。

不必使用具体的功能类去创建该类的实例，创建实例的操作交给工厂类去完成。但是，当需要增加一个新的功能类的时候，就需要在工厂类中增加一个判断。

### 工厂方法模式

定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法模式使一个类的实例化延迟到子类。

当新增加一个类的时候，不需要对工厂类进行修改，但是当新增一个功能类的时候，需要创建对应的工厂类。这样，就会创建过多的类，不如策略模式。

### 抽象工厂模式

提供一个创建一系列相关或者相互依赖对象的接口，而无需指定它们具体的类。

抽象工厂模式是工厂方法模式的升级版本。它用来创建一组相关或者相互依赖的对象。

与工厂方法模式的区别：工厂方法模式针对的是一个产品等级结构，而抽象工厂模式针对的是多个产品等级结构。通常，一个产品结构表现为抽象类或者接口，抽象工厂模式所提供的产品衍生自不同的接口或者抽象类。

**工厂方法模式**：

一个抽象产品类，可以派生出多个具体产品类。

一个抽象工厂类，可以派生出多个具体工厂类。

每个具体工厂类，只能创建一个具体产品类的实例。

**抽象工厂模式**：

多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。

一个抽象工厂类，可以派生出多个具体工厂类。

每个具体工厂类，可以创建多个具体产品类的实例。也就是创建一个产品家族下的多个产品。

例子：工厂可以生产鼠标和键盘，但是微软和罗技都有这两个产品。那么，微软和罗技就可以看成是两个产品族，分别由A工厂和B工厂生产各自的鼠标和键盘。A和B对应于抽象工厂，每个工厂生产的鼠标和键盘对应于工厂方法。

如果使用工厂方法模式，只要替换生成键盘的工厂方法就可以把键盘从罗技换到微软。

但是使用了抽象工厂模式，只要换家工厂，就可以同时换鼠标和键盘。如果需要的产品（鼠标 键盘..）有很多，使用抽象工厂模式一次性替换很方便。

### 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。

有懒汉和饿汉两种模式，同时要注意线程安全的写法。

应用：对于无状态的类使用单例模式，节省内存资源。

Servlet中的实例就是单例模式，但是是多线程。

Spring中创建的Bean对象默认是单例模式，这样就不用为每个请求创建一个实例对象，减少性能开销。但是Struts2中的Action是多例模式，针对每个请求都会创建一个实例，因此是线程安全的。

### 建造者模式

将一个复杂对象的创建与它的表示分离，使得同样的创建过程可以创建不同的表示。

需要建造者，还需要一个指挥者，负责整体的构建算法，也就是如何去组合产品。

应用：一个类的各个组成部分的具体实现类或者算法经常变化，但是将它们组合在一起的算法却相对稳定。提供一种封装机制将稳定的组合算法于易变的各个组成部分隔离开来。

### 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。

简单的说，就是从一个对象再创建另外一个可定制的对象，而且不需要知道任何创建的细节。

应用：如Object中的clone方法，需要该类实现Cloneable接口，注意这是浅表复制。如果想实现深表复制，可以将引用的对象也实现Cloneable接口，重写clone方法；或采用序列化，也就是采用流的方式读入当前对象的二进制输入，再写出二进制数据对应的对象。

/\* 写入当前对象的二进制流 \*/

ByteArrayOutputStream bos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bos);

oos.writeObject(this);

/\* 读出二进制流产生的新对象 \*/

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bis);

return ois.readObject();

## 结构型模式

### 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。

代理对象和被代理对象需要实现相同的接口，这样代理类才能完成代理工作。

### 装饰器模式

动态的给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更加灵活。

装饰对象和被装饰对象需要实现共同的接口，这样可以层层装饰。

在客户端创建被装饰的对象，然后作为构造参数传给装饰对象。

应用：当系统需要新的功能，向旧的类中添加新的代码。

Java的IO中，BufferedReader对fileReader就是一个装饰。

InputStreamReader是对inputStream进行适配，将字节流转换为字符流。

装饰模式与代理模式的区别：装饰模式关注于在一个对象上动态的添加方法，代理模式关注于控制对象的访问。对于客户端来说，代理模式中，客户端不需要知道被代理对象的信息，被代理的对象是在代理类中完成了一个实例的创建。而装饰模式中，需要在客户端中将原始对象作为一个参数传给装饰者模式。

### 适配器模式

将一个类的接口转换成用户希望的另外一种接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作。

适配器模式主要是希望复用一些现有的类，但接口与复用环境要求不一致的情况。

应用：系统的数据和行为都正确时，但是接口不符合，应该用适配器。

### 外观模式

为子系统的一组接口提供一致的界面，此模式定义了一个高层的接口，这个接口使得这一子系统更加容易使用。

应用：比如在MVC架构中，Action层 Service层和Dao层就是外观模式。Service层中的方法可能会需要多个Dao层中的方法结合使用，这样将这些方法封装起来，向Action层提供一系列简单的接口，使得Action层的代码更加简洁和清晰。

Tomcat中也使用了外观模式，Tomcat中的每个组件都要相互通信，但是不能将自己内部的数据过多的暴露给其他组件，使用外观模式进行隔离数据。实际上传递的是RequestFacade和ResponseFacade对象，只提供外部程序感兴趣的方法。

外观模式与代理模式的区别：

代理模式是代理一个单一的对象，而外观模式代表一个子系统。

### 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。

对于客户端来说，无需区分是操作的是树枝对象还是树叶对象。

应用：文件系统

### 享元模式

运用共享技术有效的支持大量细粒度的对象。

实现对象的共享，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。

FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，则返回已经存在的对象，如果没有，则创建一个新的对象。

应用：String类型就是享元模式，对象一旦创建，就不能改变，存放于常量池中。

数据库连接池，url driverClassName username password dbname这些对于每个连接来说都是一样的，所以适合用享元模式处理。通过连接池的管理，实现了数据库连接的共享，不需要每一次都创建新的连接，节省了数据库重新创建的开销，提升了系统的性能。

### 桥接模式

将抽象部分与它的实现部分分离，使它们都可以独立的变化。

应用：在JDBC中，使用了桥接模式。对于应用程序而言，只要选择不同的驱动方式，就可以让程序操作不同的数据库，而无需更改应用程序，对于数据库而言，为数据库实现不同的驱动程序，并不会影响应用程序。

## 行为型模式

### 观察者模式

也是发布—订阅模式，是一种一对多的依赖关系。让多个观察者对象同时监听某一个主题对象，这个主题对象发生变化时，会通知所有观察者对象，使得它们可以自动更新自己。

将一个系统分割成一系列相互合作的类有不好的作用，那就是需要维护相关对象间的一致性。不希望为了维护一致性而使各类紧密耦合，这样会给维护 扩展 重用带来不变。

应用：util库中有Observale和Observer接口，被观察对象集成Observable类，Watcher对象实现Observer接口。

Spring中的事件驱动模型是观察者模式的一个典型应用。

public abstract class ApplicationEvent extends EventObject {

private static final long serialVersionUID = 7099057708183571937L;

private final long timestamp;

public ApplicationEvent(Object source) {

super(source);

this.timestamp = System.currentTimeMillis();

}

public final long getTimestamp() {

return this.timestamp;

}

}

ApplicationEvent继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过source得到事件源.该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件.

public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

void onApplicationEvent(E event);

}

ApplicationListener继承自jdk的EventListener,所有的监听器都要实现这个接口,这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过不同对Event类的判断来进行相应的处理.当事件触发时所有的监听器都会收到消息,如果你需要对监听器的接收顺序有要求,可是实现该接口的一个实现SmartApplicationListener,通过这个接口可以指定监听器接收事件的顺序.

事件机制需要事件源 事件 事件监听器。ApplicationEvent相当于事件， ApplicationListener相当于事件监听器，事件源是ApplicationContext。ApplicationContext是spring中的全局容器，负责读取bean的配置文档，管理bean的加载，维护bean之间的依赖关系。ApplicationContext作为一个事件源，需要显示的调用publishEvent方法，传入一个ApplicationEvent的实现类对象作为参数，每当ApplicationContext发布ApplicationEvent时，所有的ApplicationListener就会被自动触发。

ApplicationContext接口实现了ApplicationEventPublisher接口，里面有一个发布事件的方法：

public interface ApplicationEventPublisher {

void publishEvent(ApplicationEvent event);

}

我们常用的ApplicationContext都继承了AbstractApplicationContext,像我们平时常见的ClassPathXmlApplicationContext、XmlWebApplicationContex也都是继承了它,AbstractApplicationcontext是ApplicationContext接口的抽象实现类,在该类中实现了publishEvent方法。

public void publishEvent(ApplicationEvent event) {

Assert.notNull(event, "Event must not be null");

if (logger.isTraceEnabled()) {

logger.trace("Publishing event in " + getDisplayName() + ": " + event);

}

getApplicationEventMulticaster().multicastEvent(event);

if (this.parent != null) {

this.parent.publishEvent(event);

}

}

在这个方法中,我们看到了一个getApplicationEventMulticaster().这就要牵扯到另一个类ApplicationEventMulticaster.

ApplicationEventMulticaster属于事件广播器，作用就是把ApplicationContext发布的Event广播给所有的监听器。

在AbstractApplicationcontext中有一个applicationEventMulticaster的成员变量,提供了监听器Listener的注册方法.

public abstract class AbstractApplicationContext extends DefaultResourceLoader

implements ConfigurableApplicationContext, DisposableBean {

　　private ApplicationEventMulticaster applicationEventMulticaster;

　　protected void registerListeners() {

// Register statically specified listeners first.

for (ApplicationListener<?> listener : getApplicationListeners()) {

getApplicationEventMulticaster().addApplicationListener(listener);

}

// Do not initialize FactoryBeans here: We need to leave all regular beans

// uninitialized to let post-processors apply to them!

String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);

for (String lisName : listenerBeanNames) {

getApplicationEventMulticaster().addApplicationListenerBean(lisName);

}

}

}

### 责任链模式

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合。将这些对象链接成一条链，并沿着这条链传递该请求，直到一个对象处理为止。

好处：请求是沿着链传递，直至有一个具体的处理者对象对其进行处理。这使得接收者和发送者都没有对方明确的信息，并且链中的对象也不知道链的结构。职责链可以简化对象之间的相互连接，仅需保持一个指向其后继者的调用，而不需要保持它所有的候选接收者的引用。

应用：Tomcat中的Filter使用了责任链模式。

Tomcat的容器设置也是责任链模式。Engine-Host-Context-Wrapper都是通过一个链传递请求。

### 模板方法模式

定义了一个操作中算法的骨架，而将一些步骤延迟到子类。模板方法使得子类可以不改变一个算法的结构就可以重新定义该算法的某些特定的步骤。

将不变的行为搬移到超类中，去除子类中的重复代码来体现它的优势。

应用：HttpServlet提供一个service方法，这个方法调用了7个do方法中的一个或者几个，完成对客户端的响应。这些do方法要求HttpServlet具体子类去实现。

### 策略模式

定义了算法家族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化不会影响到使用算法的用户。

这些算法都是完成相同的工作，只是实现不同。

应用：需要在不同的时间点，应用不用的业务规则。（比如打折促销各种优惠手段）

### 迭代子模式

提供一种方法顺序的访问一个聚合对象中各个元素，而不是暴露该对象的内部表示。

应用：当需要访问一个聚集对象时，而且不管这些对象是什么就需要遍历的时候。

Java中的集合。

### 命令模式

将一个请求封装为一个对象，从而使你可以用不同的请求对客户 进行参数化，对请求排队或者记录请求日志时，以及支持撤销的操作。

命令模式把发出命令的责任和执行命令的责任分割开来，委派给不同的对象。

优点：比较容易的设计一个请求队列。在需要的情况下，可以比较容易的将命令记入日志。

允许接收请求的一方决定是否要否决请求。比较容易的实现对请求的撤销和重做。

### 状态模式

当一个对象的内存状态改变时，允许改变其行为。这个对象看起来是像改变了类。

当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到不同状态的一系列类当中，可以把复杂的判断逻辑简化。

应用：当一个对象的行为取决于它的状态时，并且它必须在运行时刻根据它的状态改变它的行为时，可以考虑使用。

### 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。这样以后就可将该对象恢复到原先保存的状态。

应用：适应于功能比较复杂，但需要维护或记录属性历史的类。（游戏进度）

### 访问者模式

表示一个作用于某对象结构中的各元素的操作。它使你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。

应用：适应于数据结构相对稳定的系统。它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对的自由的演化。目的是把处理从数据结构中分离出来。如果系统中有比较稳定的数据结构，又有易于变化的算法，用访问者模式。

优点：增加新的操作很容易，只需要增加一个新的访问者。

### 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示的相互调用，从而使其耦合松散，而且可以独立的改变它们之间的交互。

中介对象需要知道所有的具体同事类，从具体同事类中接收消息，向具体同事发出命令。

具体同事类，每个具体同事只知道自己的行为，而不了解其他同事类的情况，但它们都认识中介对象。

优点：中介者模式减少了各个Colleage的耦合，使得可以独立的改变和复用各个Colleague类和Mediator。

缺点：交互的复杂性变为了中介者的复杂性，这使得中介者变得比任何一个类都复杂。

应用：应用于一组对象以定义良好但是复杂的方式进行通信的场合。以及想定制一个分布在多个类中而又不想生成太多子类的场合。

中介者模式与代理模式：

代理模式是一对一，这个代理只能代表一个对象。只能代理一方，比如PB是B的代理，A能够通过PB访问B，但是B不能通过PB访问A。（比如手机代理，我们只能通过手机代理去买手机）

中介者模式是多对多，这些被管理的对象之间都可以通信，它们的业务关系应该是交互在一起的。A可以通过中介访问B，B也能够通过中介访问A。（比如房屋中介，中介者有房源的信息也有客户的信息，可以双向进行通信）

### 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，这个解释器使用该表示来解释语言中的句子。

应用：如果一种特定类型的问题发生的频率足够高，那么就有可能值得将该问题的实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。