# 常见的设计模式

**1) 单例模式。**

单例模式是一种常用的软件设计模式。

在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数的控制并节约系统资源。

应用场景：如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

**2) 工厂模式。**

工厂模式主要是为创建对象提供了接口。

应用场景如下：

a、 在编码时不能预见需要创建哪种类的实例。

b、 系统不应依赖于产品类实例如何被创建、组合和表达的细节。

**3) 策略模式。**

策略模式：定义了算法族，分别封装起来，让它们之间可以互相替换。此模式让算法的变化独立于使用算法的客户。

应用场景如下。

a、 一件事情，有很多方案可以实现。

b、我可以在任何时候，决定采用哪一种实现。

c.、未来可能增加更多的方案。

d、 策略模式让方案的变化不会影响到使用方案的客户。

举例业务场景如下。

系统的操作都要有日志记录，通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面。日志写到数据库与文件中是两种算法，但调用方不关心，只负责写就是。

**4) 观察者模式。**

观察者模式又被称作发布/订阅模式，定义了对象间一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

应用场景如下：

a、对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

b、对象仅需要将自己的更新通知给其他对象而不需要知道其他对象的细节。

**5) 迭代器模式。**

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，而又不暴露该对象的内部表示。

应用场景如下：

当你需要访问一个聚集对象，而且不管这些对象是什么都需要遍 历的时候，就应该考虑用迭代器模式。其实stl容器就是很好的迭代器模式的例子。

**6) 模板方法模式。**

模板方法模式定义一个操作中的算法的骨架，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步骤。

## 案例

# Spring boot相关设计模式

## 1.工厂模式

### 1.1 实现方式

Spring中的BeanFactory就是简单工厂模式的体现，根据传入beanName来获得Bean对象，先从缓存中取，缓存中没有再创建。

### 1.2实现原理

bean容器的启动阶段：

读取bean的xml配置文件，将bean元素分别转换成一个个的BeanDefinition对象。

然后通过BeanDefinitionRegistry将这些bean注册到beanFactory中，保存在它的一个ConcurrentHashMap中。

将BeanDefinition注册到了beanFactory之后，在这里Spring为我们提供了一个扩展的接口，允许我们通过实现接口BeanFactoryPostProcessor 在此处来插入我们定义的代码。典型的例子就是：PropertyPlaceholderConfigurer，我们一般在配置数据库的dataSource时使用到的占位符的值，就是它注入进去的。

### 1.3 设计意义

松耦合。可以将原来硬编码的依赖，通过Spring这个beanFactory这个工厂来注入依赖，也就是说原来只有依赖方和被依赖方，现在我们引入了第三方——spring这个beanFactory，由它来解决bean之间的依赖问题，达到了松耦合的效果。

bean的额外处理。通过Spring接口的暴露，在实例化bean的阶段我们可以进行一些额外的处理，这些额外的处理只需要让bean实现对应的接口即可，那么spring就会在bean的生命周期调用我们实现的接口来处理该bean。

## 单例模式

spring对单例的实现：spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是任意的java对象

## 适配器模式

### 3.1实现方式

SpringMVC中的适配器HandlerAdatper。

### 3.2实现原理

### HandlerAdatper根据Handler规则执行不同的Handler。

### 3.3实现过程：

DispatcherServlet根据HandlerMapping返回的handler，向HandlerAdatper发起请求，处理Handler。

HandlerAdapter根据规则找到对应的Handler并让其执行，执行完毕后Handler会向HandlerAdapter返回一个ModelAndView，最后由HandlerAdapter向DispatchServelet返回一个ModelAndView。

### 3.4实现意义：

HandlerAdatper使得Handler的扩展变得容易，只需要增加一个新的Handler和一个对应的HandlerAdapter即可。

因此Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

## 4.装饰器模式

### 4.1实现原理

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

### 4.2 实现过程

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

## 5.代理模式

### 5.1实现方式：

AOP底层，就是动态代理模式的实现。

### 5.2动态代理：

在内存中构建的，不需要手动编写代理类。

****5.3静态代理：****

需要手工编写代理类，代理类引用被代理对象。

### 5.4实现原理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

## 6策略模式

### 6.1实现方式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源。

Resource 接口介绍

Resource接口是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。

Resource 接口主要提供了如下几个方法：

getInputStream()：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。

exists()：返回 Resource 所指向的资源是否存在。

isOpen()：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束应该显式关闭，以防止资源泄漏。

getDescription()：返回资源的描述信息，通常用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL。

getFile：返回资源对应的 File 对象（通常不用）。

getURL：返回资源对应的 URL 对象（通常不用）。

Resource 接口本身没有提供访问任何底层资源的实现逻辑，针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑。

Spring 为 Resource 接口提供了如下实现类：

UrlResource：访问网络资源的实现类。

ClassPathResource：访问类加载路径里资源的实现类。

FileSystemResource：访问文件系统里资源的实现类。

ServletContextResource：访问相对于 ServletContext 路径里的资源的实现类

InputStreamResource：访问输入流资源的实现类。

ByteArrayResource：访问字节数组资源的实现类。

这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。