**Spring框架中常用的设计模式详解**

**1）工厂设计模式（简单工厂和工厂方法）**

Spring使用工厂模式可以通过BeanFactory或ApplicationContext创建bean对象。

两者对比：

 BeanFactory ：延迟注入(使用到某个 bean 的时候才会注入),相比于BeanFactory来说会占用更少的内存，程序启动速度更快。

 ApplicationContext ：容器启动的时候，不管你用没用到，一次性创建所有 bean 。BeanFactory 仅提供了最基本的依赖注入支持，ApplicationContext 扩展了 BeanFactory ,除了有BeanFactory的功能还有额外更多功能，所以一般开发人员使用ApplicationContext会更多。

**2)单例设计模式**

Spring中bean的默认作用域就是singleton。除了singleton作用域，Spring bean还有下面几种作用域：

 prototype : 每次请求都会创建一个新的 bean 实例。

 request : 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。

 session : 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。

 global-session： 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话。

Spring实现单例的方式：

xml格式：<bean id="userService" class="top.snailclimb.UserService" scope="singleton"/>

注解：@Scope(value = "singleton")

Spring通过ConcurrentHashMap实现单例注册表的特殊方式实现单例模式。Spring实现单例的核心代码如下：

// 通过 ConcurrentHashMap（线程安全） 实现单例注册表

private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(64);

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(beanName, "'beanName' must not be null");

synchronized (this.singletonObjects) {

// 检查缓存中是否存在实例

Object singletonObject = this.singletonObjects.get(beanName);

if (singletonObject == null) {

//...省略了很多代码

try {

singletonObject = singletonFactory.getObject();

}

//...省略了很多代码

// 如果实例对象在不存在，我们注册到单例注册表中。

addSingleton(beanName, singletonObject);

}

return (singletonObject != NULL\_OBJECT ? singletonObject : null);

}

}

//将对象添加到单例注册表

protected void addSingleton(String beanName, Object singletonObject) {

synchronized (this.singletonObjects) {

this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL\_OBJECT));

}

}

}

**3）代理设计模式**

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用JDK Proxy去进行代理了，这时候Spring AOP会使用Cglib，这时候Spring AOP会使用Cglib生成一个被代理对象的子类来作为代理。

### 4）模板方法设计模式

模板方法模式是一种行为设计模式，它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在不改变一个算法的结构即可重定义该算法的默写特定步骤的实现方式。例如：

public abstract class Template {

//这是我们的模板方法

public final void TemplateMethod(){

PrimitiveOperation1();

PrimitiveOperation2();

PrimitiveOperation3();

}

protected void PrimitiveOperation1(){

//当前类实现

}

//被子类实现的方法

protected abstract void PrimitiveOperation2();

protected abstract void PrimitiveOperation3();

}

public class TemplateImpl extends Template {

@Override

public void PrimitiveOperation2() {

//当前类实现

}

@Override

public void PrimitiveOperation3() {

//当前类实现

}

}

Spring中jdbcTemplate、hibernateTemplate等以Template结尾的对数据库操作的类，它们就使用到模板模式。一般情况下，我们都是使用继承的方式来实现模板模式，但是Spring并没有使用这种方式，而是使用Callback模式与模板方法配合，既达到了代码复用的效果，同时增加了灵活性。

**5）观察者设计模式**

观察者设计模式是一种对象行为模式。它表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变时，这个对象锁依赖的对象也会做出反应。Spring事件驱动模型就是观察者模式很经典的应用。

事件角色：ApplicationEvent（org.springframework.context包下）充当事件的角色，这是一个抽象类。

事件监听者角色：ApplicationListener充当了事件监听者的角色，它是一个接口，里面只定义了一个onApplicationEvent（）方法来处理ApplicationEvent。

事件发布者角色：ApplicationEventPublisher充当了事件的发布者，它也是个接口。

**实现方式：spring的事件驱动模型使用的是 观察者模式 ，Spring中Observer模式常用的地方是listener的实现。**

**具体实现：**

**事件机制的实现需要三个部分,事件源,事件,事件监听器**

ApplicationEvent抽象类[事件]

继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过构造器参数source得到事件源.

该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件.

Spring事件流程总结：

 定义一个事件: 实现一个继承自 ApplicationEvent，并且写相应的构造函数；

 定义一个事件监听者：实现 ApplicationListener 接口，重写 onApplicationEvent() 方法；

 使用事件发布者发布消息: 可以通过 ApplicationEventPublisher 的 publishEvent() 方法发布消息。

/ 定义一个事件,继承自ApplicationEvent并且写相应的构造函数

public class DemoEvent extends ApplicationEvent{

private static final long serialVersionUID = 1L;

private String message;

public DemoEvent(Object source,String message){

super(source);

this.message = message;

}

public String getMessage() {

return message;

}

// 定义一个事件监听者,实现ApplicationListener接口，重写 onApplicationEvent() 方法；

@Component

public class DemoListener implements ApplicationListener<DemoEvent>{

//使用onApplicationEvent接收消息

@Override

public void onApplicationEvent(DemoEvent event) {

String msg = event.getMessage();

System.out.println("接收到的信息是："+msg);

}

}

// 发布事件，可以通过ApplicationEventPublisher 的 publishEvent() 方法发布消息。

@Component

public class DemoPublisher {

@Autowired

ApplicationContext applicationContext;

public void publish(String message){

//发布事件

applicationContext.publishEvent(new DemoEvent(this, message));

}

}

**6）适配器设计模式**

适配器设计模式将一个接口转换成客户希望的另一个接口，适配器模式使得接口不兼容的那些类可以一起工作，其别名为包装器。在Spring MVC中，DispatcherServlet根据请求信息调用HandlerMapping，解析请求对应的Handler，解析到对应的Handler（也就是我们常说的Controller控制器）后，开始由HandlerAdapter适配器处理。为什么要在Spring MVC中使用适配器模式？Spring MVC中的Controller种类众多不同类型的Controller通过不同的方法来对请求进行处理，有利于代码的维护拓展。

**实现方式：SpringMVC中的适配器HandlerAdatper。**

**实现原理：HandlerAdatper根据Handler规则执行不同的Handler。**

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。

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

**7）装饰者设计模式**

装饰者设计模式可以动态地给对象增加些额外的属性或行为。相比于使用继承，装饰者模式更加灵活。Spring 中配置DataSource的时候，DataSource可能是不同的数据库和数据源。我们能否根据客户的需求在少修改原有类的代码下切换不同的数据源？这个时候据需要用到装饰者模式。

**8）策略设计模式**

Spring 框架的资源访问接口就是基于策略设计模式实现的。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接口本身没有提供访问任何底层资源的实现逻辑，针对不同的额底层资源，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类型。

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。