Spring系列产品

[1. Spring IOC 1](#_101)

[1.1. 组件 1](#_104)

[1.1.1. @Component 1](#_105)

[1.1.2. @Controller 1](#_106)

[1.1.3. @Service 1](#_107)

[1.1.4. @Repository 1](#_108)

[1.2. BeanFactory和FactoryBean 1](#_109)

[1.2.1. 前者IOC 容器 1](#_110)

[1.2.2. 后者特殊的Bean，具体如ProxyFactoryBean 1](#_111)

[1.3. 原理：建立BeanName到BeanDefinition的映射 1](#_112)

[1.4. 作用域 1](#_113)

[1.4.1. singleton 1](#_114)

[1.4.2. prototype 1](#_115)

[1.4.3. session 1](#_116)

[1.4.4. request 1](#_117)

[1.5. BeanFactoryPostProcessor和BeanPostProcesser 1](#_132)

[1.6. Bean生命周期 2](#_133)

[1.7. BeanFactory和FactoryBean 2](#_143)

[2. Spring AOP 2](#_102)

[2.1. 通知 2](#_118)

[2.1.1. 前置通知 2](#_119)

[2.1.2. 正常返回通知 2](#_120)

[2.1.3. 异常返回通知 2](#_121)

[2.1.4. 最终通知 3](#_122)

[2.1.5. 环绕通知 3](#_123)

[2.2. 生成代理 3](#_124)

[2.2.1. JDK动态代理 3](#_125)

[2.2.2. CGLib 4](#_126)

[2.3. 原理 4](#_144)

[3. Spring 事务 5](#_103)

[3.1. 编程式事务 5](#_127)

[3.1.1. TransactionTemplate 5](#_129)

[3.2. 声明式事务 5](#_128)

[3.2.1. 通过aop实现 5](#_130)

[3.2.2. 事务传播级别 5](#_131)

[4. 常见问题 6](#_138)

[4.1. @Autowired和@Resource 6](#_139)

[4.2. 设计模式 6](#_140)

[5. SpringMVC 6](#_134)

[5.1. 参数解析 6](#_135)

[5.2. DispatcherServlet初始化 7](#_137)

[5.3. 用户请求处理过程 7](#_136)

[5.3.1. HandlerAdapter 8](#_141)

# Spring IOC

## 组件

### @Component

### @Controller

### @Service

### @Repository

## BeanFactory和FactoryBean

### 前者IOC 容器

### 后者特殊的Bean，具体如ProxyFactoryBean

## 原理：建立BeanName到BeanDefinition的映射

## 作用域

### singleton

### prototype

### session

### request

## BeanFactoryPostProcessor和BeanPostProcesser

BeanFactoryPostProcessor实现类：在bean创建前读取配置元信息，修改任意bean。

执行时机：在IOC容器加载了bean定义后，在bean实例化之前执行的。

BeanPostProcessor实现类：在该bean初始化方法前，执行自己的处理逻辑。

执行时机：在IOC容器加载bean定义，并且实例化bean之后执行。

作用域为singleton但是lazy-init为false的普通bean。

## Bean生命周期

1. 实例化Bean对象（用构造方法），需要注意的是BeanFactory是懒加载，而ApplicationContext则是饿加载。

2. 设置对象属性（依赖注入），实质是还是使用反射：

构造器注入construct-arg

setter注入property

注解注入

3. 检查Aware接口并设置相关依赖。保扩setBeanName()，setBeanFactory()和setApplicationContext()方法。

4. BeanPostProcesser 前置处理。

5. 检查是否是InitializingBean以决定是否调用afterProperties方法。

6. 检查是否有自定义的init-method。

7. 调用BeanPostProcessor接口的postProcessAfterInitialization方法。

8. 应用上下文销毁，调用DisposableBean的destroy方法。

9. 调用自定义的销毁方法。

## BeanFactory和FactoryBean

BeanFactory是IOC容器或对象工厂。在Spring中，所有的Bean都是由BeanFactory来管理的。

FactoryBean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean，即实现FactoryBean的getObject方法。在IOC容器中通过Bean Id得到其getObject的产物。如ProxyFactoryBean，通过这个FactoryBean生成对应的代理对象。

# Spring AOP

## 通知

### 前置通知

在切入点前面执行，前置通知不会影响连接点的执行，除非前置通知出现异常。

### 正常返回通知

在正常返回后执行

### 异常返回通知

在连接点抛出异常后执行

### 最终通知

不管是正常执行完成，还是抛出异常，都会执行

### 环绕通知

最强大的通知类型。

使用ProceedingJoinPoint.process去控制。

## 生成代理

### JDK动态代理

1. 基于接口

2. Proxy.newProxyInstance

3. 实现InvocationHandler的invoke方法

public class JDKProxy implements InvocationHandler {

/\*\* 需要代理的目标对象 \*/

private Object targetObject;

/\*\* 将目标对象传入进行代理 \*/

public Object newProxy(Object targetObject) {

this.targetObject = targetObject; //返回代理对象

return Proxy.newProxyInstance(targetObject.getClass().getClassLoader(), targetObject.getClass().getInterfaces(), this);

}

/\*\* invoke方法 \*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable { // 一般我们进行逻辑处理的函数比如这个地方是模拟检查权限

checkPopedom();

// 设置方法的返回值

Object ret = null;

// 调用invoke方法，ret存储该方法的返回值

ret = method.invoke(targetObject, args);

return ret;

}

/\*\* \* 模拟检查权限的例子 \*/

private void checkPopedom() {

System.out.println("======检查权限checkPopedom()======");

}

}

### CGLib

1. 基于父类

2. enhancer.create()

3. 实现MethodInterceptor的intercept方法

public class CGLibProxy implements MethodInterceptor {

/\*\* CGLib需要代理的目标对象 \*/

private Object targetObject;

public Object createProxyObject(Object obj) {

this.targetObject = obj;

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(obj.getClass());

enhancer.setCallback(this);

Object proxyObj = enhancer.create();

// 返回代理对象

return proxyObj;

}

@Override

public Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) throws Throwable {

Object obj = null;

// 过滤方法

if ("addUser".equals(method.getName())) {

// 检查权限

checkPopedom();

}

obj = method.invoke(targetObject, args);

return obj;

}

private void checkPopedom() {

System.out.println("======检查权限checkPopedom()======");

}

}

## 原理

Spring代理对象的核心类是AbstractAutoProxyCreator,实现了BeanPostProcessor接口。

BeanPostProcessor具有两个接口方法，postProcessBeforeInitialization和postProcessAfterInitialization。

在postProcessAfterInitialization中通过wrapIfNecessary方法，查询当前bean的所有aop增强，最终通过AOPUtils工具类实现...最后则是通过ProxyFactory.getProxy()来创建代理对象。

ProxyFactory的父类构造器实例化了DefaultAopProxyFactory类，对于接口或者目标类就是代理对象，使用JDK动态代理；否则使用CGLib代理。

# Spring 事务

## 编程式事务

### TransactionTemplate

## 声明式事务

声明式事务的实现就是通过环绕增强的方式，在目标方法执行之前开启事务，在目标方法执行之后提交或者回滚事务

### 通过aop实现

Spring的事务控制器也是根据AOP实现的，通过TransactionProxy获得Service层对象（实际上是使用ProxyFactory生成的代理对象）

调用代理对象的方法时，先使用TransactionInterceptor拦截器进行拦截，通过这个拦截器的环绕式实现事务的开启与提交。

### 事务传播级别

**[1]死活不要事务**

PROPAGATION\_NEVER：没有就非事务执行，有就抛出异常

PROPAGATION\_NOT\_SUPPORTED：没有就非事务执行，有就事务挂起，然后非事务执行

**[2]可有可无的**

PROPAGATION\_SUPPORTS：有就用，没有就算了

**[3]必须有事务的**

PROPAGATION\_REQUIRED：默认传播级别，如果没有，就新建一个事务；如果有，就加入当前事务。

PROPAGATION\_REQUIRED\_NEW：如果没有，就新建一个事务；如果有，就把事务挂起，执行自己的事务。

PROPAGATION\_NESTED：如果没有，就新建一个事务；如果有，就嵌套。

PROPAGATION\_MANDATORY：如果没有，就抛出异常；如果有，就使用当前异常。

# 常见问题

## @Autowired和@Resource

**@Autowired**：默认按照类型装配，当找到多个匹配实例时，再通过name确定Bean。当然存在多个Bean的时候也可以使用@Primary和@Priority进一步判断，还可以和@Qualifier配合使用进行精确查找。

**@Resource**：使用时需要写上name = Bean id(by name)或type = BeanClass.class(by class)，等价于@Autowired和@Qualifier。

## 设计模式

**Spring IOC中用到的设计模式**

工厂模式：需要用到的Bean可以从BeanFactory或ApplicationContext中取出。

单例模式：Spring中bean的默认作用域就是singleton（单例）

**Spring AOP中用到的设计模式**

代理模式：AOP就是基于动态代理的

**JDBC Template中用到的**

模板方法：定义了操作的骨架，将一些步骤延迟到子类

**其他**

观察者模式：Spring中的ApplicationEvent和ApplicationListener<>，使用ApplicationEventPublishAware发布事件

# SpringMVC

## 参数解析

1. SpringMVC初始化时会设置一些默认的参数解析器。当SpringMVC接收到请求后首先根据url查找对应的HandlerMethod。

2. 遍历HandlerMethod中的MethodParameter数组。

3. 根据MethodParameter的类型来查找和确认使用哪个参数解析器。

4. 解析参数，从Request中解析出MethodParameter对应的参数，这里解析出来的都是String。

5. 转换参数，将String转化为具体需要的类型，包括基本类型、对象等。

## DispatcherServlet初始化

1. 从DispatcherServlet（前端控制器）的init方法开始，首先读取配置文件进行初始化，把ContextLoaderListener建立起来的WebApplicationContext作为父上下文，然后初始化所有的Controller层的Bean放在IOC容器中备用。

2. DispatcherServlet的init方法第二步是初始化各种组件，以HandlerMapping为例，会加载ApplicationContext中的HandlerMapping，其中有RequestMappingHandlerMapping对象。

3. RequestMappingHandlerMapping会扫描所有自定义的拦截器对象，并且会扫描Controller获取其中标有@RequestMapping的方法，存入urlLookup和mappingLookup中。

ContextLoaderListener在启动Web容器中，会加载整个Spring容器相关的bean配置管理

## 用户请求处理过程

有以下步骤：

1. DispatcherServlet遍历HandlerMapping，找到对应的HandlerMapping，并得到HandlerExecutionChain（内部包含了拦截器）

2. 调用拦截器的preHandle方法

3. 使用HandlerExecutionChain中的Handler遍历HandlerAdapter集合，找到支持此Handler的HandlerAdapter

4. 使用HandlerAdapter得到ModelAndView。

解析方法的参数@RequestBody

处理方法的返回值@ResponseBody

5. 调用拦截器的postHandler

6. 异常处理，前面4个步骤可能会发生异常

7. ViewResolver根据ModelAndView进行解析，得到View

8. 调用拦截器的afterCompletion方法。

9. 响应给客户端。

### HandlerAdapter

HandlerAdapter是处理适配器，它的作用是调用具体的方法对用户发来的请求来进行处理，Spring MVC就是通过HandlerAdapter来实际调用处理函数。

HttpRequestHandlerAdapter

SimpleControllerHandlerAdapter:支持实现了controller接口的控制器

AnnotationMethodHandlerAdapter