spring（2）

作者：HuixiaZhang
原文链接：https://ld246.com/article/1538105701688
来源网站：链滴

1、注解开发实现ioc以及DI注入

1）注解实现ioc

需要导入aop的坐标

<context:component-scan base-package="cn.itcast.spring.demo8"/> 扫描包下所有带有组件解的类

@Component等价于@Controller(web)等价于@Service(service)等价于@Repository(dao) 表明前类是组件类

2）注解实现di

```
@Value("张三") 直接给普通类型去赋值，掌握，后面会结合properties文件来使用
@Autowired+@Qualifier=@Resource(jdk自带)
注意：使用注解实现di的时候，引用类型也要交给spring管理
```

3）spring和junit4集成的测试

```
@RunWith(SpringJUnit4ClassRunner.class)// spring整合junit4
@ContextConfiguration(locations = "classpath:applicationContext.xml")
```

4）通过xml和注解能不能混合使用？是可以的

总结xml和注解混搭DI注意点

第一种：applicationContext.xml

```
<bean id="b" class="cn.itheima.test.di2.B"/>
<bean id="a" class="cn.itheima.test.di2.A">
    <property name="b" ref="b"></property>
</bean>
```

A类中

```
private B b;
public void setB(B b) {
    this.b = b;
}
```

第二种：applicationContext.xml

```
<bean id="b" class="cn.itheima.test.di2.B"/>
<bean id="a" class="cn.itheima.test.di2.A">
</bean>
```

在A类中

```
@Autowired
private B b;
```

第三种：applicationContext.xml开启注解的扫描

```
<context:component-scan base-package="cn.itheima.test.di2"></context:component-sca
>
@Component
public class B {
}
@Component
public class A {
    @Autowired
    private B b;
}
```

注意：

A对象由spring容器创建，依赖B对象，B不被spring容器管理，A依赖注入B时，会报错
A对象手动new出来，依赖B对象，B被spring容器管理，A依赖注入B时，注入的B为null

1、aop面向切面编程介绍

增强类 目标类

A {

a (公共的逻辑)

b (业务逻辑)

c (公共的逻辑)

}

当一个方法中执行需要先后执行a,b,c逻辑的时候，我们能不能把a,c逻辑放在一个类中，
给其他类共用？能不能动态组合a+b+c逻辑，这就是面向切面编程思想的来源

底层使用动态代理：jdk和cglib，目标对象有接口，优先采用jdk，没有接口则采用cglib
1）jdk实现动态代理(基于接口实现代理对象)
接口 CustomerService
实现类 CustomerServiceImpl 目标类
代理类 CustomerProxy 目标对象的方法执行前后都可以增强

实现类与代理类，兄弟关系

2）cglib实现动态代理(Spring提供，基于继承父类而实现代理对象，代理对象继承目标对象)
实现类 CustomerServiceImpl 目标类
代理类 CustomerProxy 目标对象的方法执行前后都可以增强

实现类（父）与代理类（子），父子关系
总结：spring默认采用jdk动态代理，一个类，有接口，优先使用jdk；
无接口，通过配置proxy-target-class="true"采用cglib

2、aop各种概念介绍

1）核心概念：

(1)目标类(target): 要被增强的类

(2)代理类(proxy): 使用动态代理产生目标类的代理

(3)切入点(pointcut):目标类中需要增强的方法，这些方法都称为切入点

(4)通知(advice): 增强类中定义的方法，这些方法用于增强目标方法

(5)切面(aspect): 切入点+通知

2）其他概念：
(1)连接点(joinpoint):目标类中的所有方法 连接点包含切入点
(2)织入(weaving): 将通知方法加到目标方法中的过程 spring aop整个过程就是织入
(3)引入(introduction): 在目标类引入新的属性或者新的方法 了解一下就行

3、aop之aspectj各种通知

前置通知 before　目标方法被调用之前，就执行该前置通知方法

后置通知 after-returning　目标方法return返回之后，就执行该返回通知方法

最终通知 after　目标方法被调用完之后，不关心返回结果，就执行该最终通知方法

环绕通知 around　包裹了目标方法，在目标方法之前和在目标方法之后整个过程，经常使用ProceedoinPoint.proceed()来执行目标方法

异常通知 after-throwing　当目标方法在执行异常的时候，就会执行该异常通知方法

4、切入点表达式，execution(表达式) within

表达式完整格式：访问修饰符 返回类型 包名.类名.方法名(方法参数)　注意这里的访问修饰符可省略

* .(..) 匹配所有类所有方法　（第一个代表返回类型）

* (..) 匹配所有类所有方法

* cn.itcast.User.(..)　匹配User类下面的所有方法

* cn.itcast...*(..) 匹配itcast包下所有子包下的所有方法

实际开发中，根据情况改变切入点表达式的包和方法

5、总结两种方式实现AOP编程

xml方式

目标类　连接点　切入点

代理类　通知

切面=切入点+通知

```
目标类
<bean id="a"  class="xxx.A"/>

代理类
<bean id="b"  class="xxx.B"/>

<aop:config>
   切入点
   <aop:pointcut id="pointcut1" expression="execution(* cn.itcast.service.*.add*(..))" />

   <aop:aspect ref="b">
      <aop:before pointcut-ref="pointcut1" method="aaa" />
      <aop:after-returning pointcut-ref="pointcut2" method="bbb" />
      <aop:aroud pointcut-ref="pointcut3" method="ccc" />
      <aop:after-throwing pointcut-ref="pointcut4" method="ddd" />
      <aop:after pointcut-ref="pointcut5" method="eee" />
   </aop:aspect>
</aop:config>
```

注解方式

```
目标类　连接点　切入点
代理类　通知
切面=切入点+通知

applicationContext.xml
<context:component-scan base-package="cn.itcast.spring.demo4"/>
```

```
<aop:aspectj-autoproxy/>

目标类
@Component  @Controller @Service @Repository

代理类
@Component  @Aspect

@Before(value="execution(* cn.itcast.service.*.add*(..))")
public void a(JoinPoint jp) {

}

@AfterReturning(value="execution(* cn.itcast.service.*.delete*(..))",returning="abc")
public void b(JoinPoint jp,Object abc) {

}

@Around(value="execution(* cn.itcast.service.*.update*(..))")
public void c(ProceedingJoinPoint pjp) {

}

@AfterThrowing(value="execution(* cn.itcast.service.*.find*(..))",throwing="throwable")
public void d(Throwable throwable) {

}

@After(value="execution(* cn.itcast.service.*.batch*(..))")
public void e(JoinPoint jp) {

}
```