**Spring**

目录

[概述 1](#_Toc25452)

[如何下载Spring 2](#_Toc11106)

[Spring框架的组成部分 2](#_Toc27273)

[IOC 2](#_Toc6061)

[依赖注入 3](#_Toc362)

[bean的作用域 3](#_Toc7655)

[bean的自动装配 4](#_Toc29637)

[Spring中常用的注解 5](#_Toc1752)

[代理模式 6](#_Toc2949)

[静态代理 6](#_Toc4094)

[动态代理 6](#_Toc9859)

[AOP 9](#_Toc9430)

[AOP术语 9](#_Toc6954)

[AspectJ 11](#_Toc32630)

[Aspectj切入点语法 11](#_Toc27431)

[实现AOP的两种方法 12](#_Toc21254)

[在Spring中整合mybatis 12](#_Toc31881)

**概述**

Spring 春天。Spring框架的出现，解决了企业应用开发的复杂性。

Spring 优点

* Spring是一个开源、免费的框架（容器）
* Spring是一个轻量级的，非入侵式的框架
* Spring的核心: 控制反转(IOC)和面向切面编程(AOP)
* 支持事务的处理，对框架整合的支持

总结: Spring是一个轻量级的控制反转和面向切面编程的框架。

**如何下载Spring**

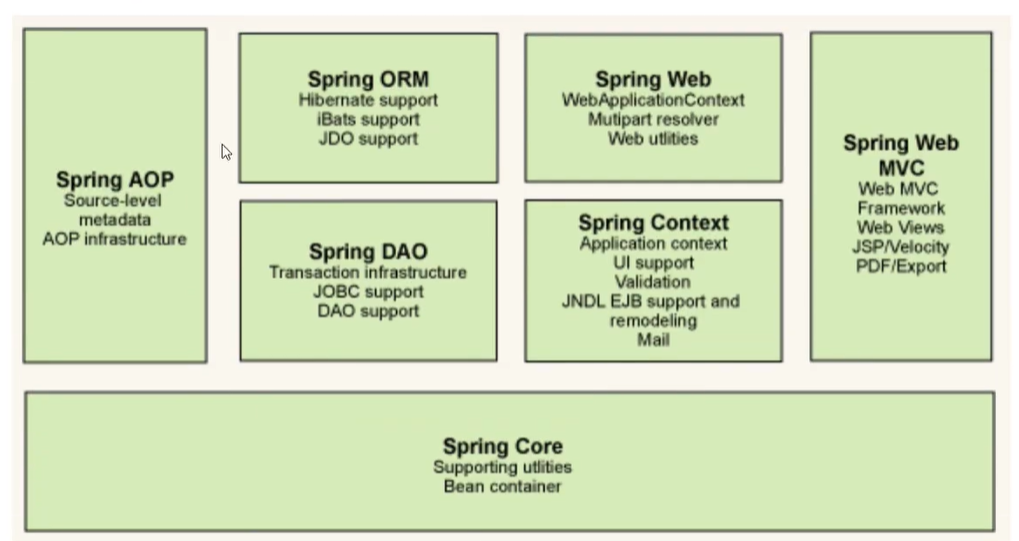
Maven仓库（https://mvnrepository.com/）

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-webmvc</artifactId>  
 <version>5.3.7</version>  
</dependency>

官网（<https://spring.io/projects/spring-framework#overview>）

Github（https://github.com/spring-projects/spring-framework）

**Spring框架的组成部分**



官网地址: <https://spring.io/projects/spring-framework#overview>

**IOC**

IOC（控制反转）：是一种思想，DI(依赖注入)是实现IOC的一种方式。IOC的核心思想，总结就是所有的java对象都由SpringIoC容器创建、管理和装配。

org.springframework.context.ApplicationContext接口代表 Spring IoC 容器，负责实例化、配置和组装 bean。容器通过读取配置元数据来获取有关要实例化、配置和组装哪些对象的指令。

**依赖注入**

依赖注入有两种方式:

- [基于构造函数的依赖注入](https://www.docs4dev.com/docs/zh/spring-framework/5.1.3.RELEASE/reference/core.html" \l "beans-constructor-injection)（通过bean类的构造器注入）

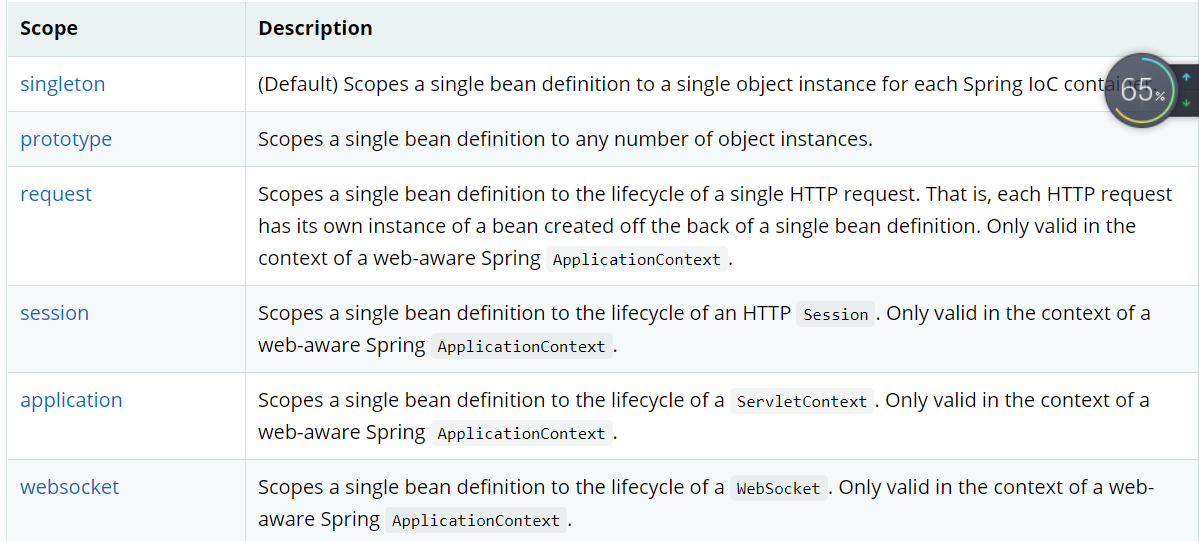
- 基于setter(set方法)的依赖注入（通过属性的set方法注入）(重点)

- 扩展注入

1. p命名空间注入，可以直接给bean的属性赋值。p命名空间注入本质还是setter注入，需要提供set方法。
2. c命名空间注入。c命名空间注入本质还是构造器注入，需要提供带参数的构造方法。

注: p和c命名空间注入，不能直接使用，需要导入约束。依赖注入就是将bean的注入到IoC容器中。可以理解为，将某个bean注入到IoC容器中后，这个bean就成为了Spring的一个组件，可以直接拿出来使用。

**bean的作用域**



- 单例模式（singleton），每次从容器get的时候，拿到的都是同一个对象

- 原型模式（prototype），每次从容器get的时候，都会产生一个新的对象

- request，session和application就是在Web开发时bean的作用域。

注: 如果没有显示的指明，默认是singleton。

**bean的自动装配**

在容器中注入一个bean，如果这个bean包含其它属性，除了手动的一个个为其装配属性外，还可以自动装配属性。自动装配是spring满足bean依赖的一种方法。有以下3中自动装配属性的方法。

- ByType自动装配

根据bean的属性类型，在IoC容器中查找，如果存在两个类型相同的

bean，则报错。

- ByName自动装配

根据bean的属性名，在IoC容器中查找，如果存在两个名字相同的bean，

则报错。

- 使用注解实现自动装配。

@Autowired注解。该注解默认通过ByType进行装配，如果容器中存在

多个满足类型的实例(bean)，就会再通过ByName进行装配，如果容器中

又存在多个满足Name的实例，则装配失败，抛出异常。所以当容器中存

在多个类型相同的实例时，可以使用@Qualifier注解来指明到底转配哪一

个实例。

@Resource注解。该注解默认通过ByName进行装配，如果在容器中找

不到，则再通过ByType装配。@Resource注解中有一个name属性，如

果容器中出现多个name相同的实例时，可以通过name属性指明到底转

配哪一个实例。注意: @Resource注解是JDK中自带的，但从JDK11开始，

该注解从JDK中移除。

**Spring中常用的注解**

@Autowired

@Qualifier

@Resource

@Component 该注解放在类上,说明这个类被spring管理了，spring容器会

将这个类部署成bean

@Service

@Repository

@Controller

注: @Service，@Repository和@Controller是@Component的衍生注解，它

们的功能一样，都是将某个类注入到spring容器，并装配bean。

@Scope 该注解用于指定bean的作用域

注: 在配置comtext命名空间和开启注解扫描后，才能使用Spring中的注解。

**代理模式**

代理模式的核心思想是借助代理类在不修改原有代码的基础上，拓展新的

功能。即通过代理类，在不修改目标方法的基础上，扩展该方法的功能。代理模式分为静态代理和动态代理。

**静态代理**

抽象角色（接口）

真实角色（接口的实现类）

代理角色（接口的调用者）

**动态代理**

动态生成的代理类是一个虚拟的类，不是直接写好的。动态代理分两类:

基于接口的动态代理和基于类的动态代理。常见的实现动态代理的方法有:

JDK动态代理：即使用jdk反射包下的接口和类实现动态代理。JDK动态代理

的是接口，且要求被代理的对象(目标对象)至少要实现一个接口。

cglib动态代理：cglib动态代理的类。

javassist动态代理，即使用javassist工具实现动态代理

这里主要学习JDK动态代理，在学习JDK实现动态代理前，需要先了解以

下3个重要的API:

- Method类，代表类中的方法，切确的说就是目标类中的方法。

作用: 通过Method可以执行某个目标类中的方法，也就是目标对象的方

法。

该类中有一个重要方法，如下所示:

public Object invoke(Object obj, Object... args); //Method对象使用该

方法，表示执行方法的调用。obi是目标对象，args是目标对象方法的参

数，即调用哪个目标对象的方法。invoke方法的返回值就是目标对象方法

的返回值。

- InvocationHandler接口，调用处理器。

该接口中只有一个invoke方法，在该方法中编写代理对象要执行的功能代

码，简单讲代理类要完成的功能就写在这个invoke方法中。该方法如下:

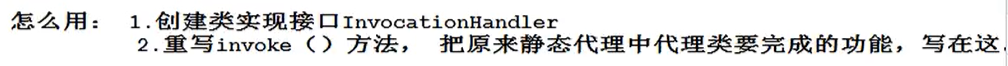
public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable;

该方法中的三个参数详解:

proxy 为jdk创建的代理对象，无需赋值

method 为目标类中的方法，jdk会提供这个method对象，无需赋值

args 为目标类中方法的参数，jdk会提供这个参数，无需赋值。 

注: InvocationHandler就代表动态生成的代理类要干什么。

- proxy类

该类中有一个静态的方法newProxyInstance，用于创建代理对象。该方法

如下:

public static Object newProxyInstance(ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h)

throws IllegalArgumentException;

该方法中的三个参数详解:

loader 是目标对象的类加载器。假如a是目标对象，通过

a.getClass().classLoader(); 就可以获取对象的类加载器。

Interfaces 是目标对象实现的接口

h 是InvocationHandler对象(调用处理器)，该对象会自动调用其invoke()

方法。前面提到过，invoke方法中就包含代理类要完成的功能。

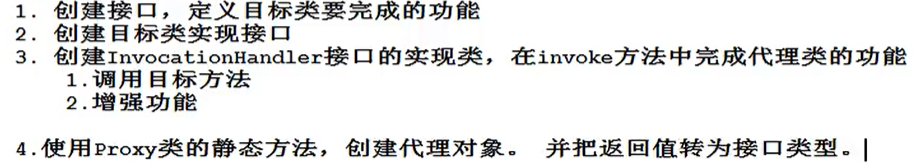
注: Method类中的invoke方法和InvocationHandler接口中的invoke方

法，不是同一个方法，别弄混淆。如果有接口，使用JDK动态代理，创建

该接口实现类的代理对象。如果没有接口，使用cglib动态代理，创建子

类的代理对象。

总结: 动态创建代理类的步骤



**AOP**

AOP(Aspect Oriented Programming)，面向切面编程。AOP底层就是通过动态代理来实现。所以AOP的核心思想就是在不修改原有代码的基础上，扩展新的功能。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

**AOP术语**

- 连接点 : 类中的哪些方法可以被增强(扩展功能)，可以被增强的方法被称

为连接点。

- 切入点: 实际被增强的方法，被称为切入点。切入点就是通知执行的位置。

- 通知（增强）: 通知就是新增的逻辑代码。简单讲就是新增的那个功能就

是通知。通知根据位置可以分为5种:

前置通知：在目标方法执行之前添加通知

后置通知：在目标方法执行之后添加通知

环绕通知：在目标方法执行前后都添加通知

异常通知：目标方法出现异常后执行该通知。

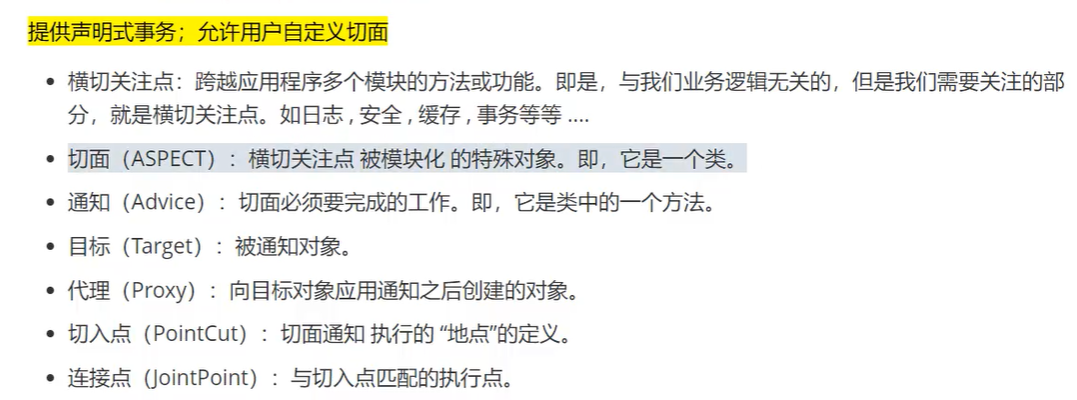
最终通知：当目标方法发生异常，后置通知是不会执行的。但目标方法是

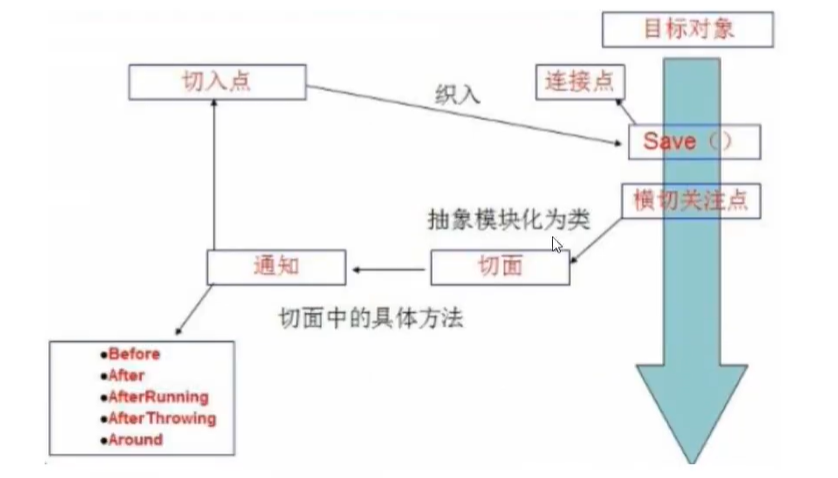
否发生异常，最终通知都会执行，类似于finally语句。

- 切面：将通知应用到切入点的过程就叫做切面。可见切面就是一序列操作

的过程，动作。

简单的图解如下:







**AspectJ**

Spring框架一般是使用AspectJ来实现AOP操作。AspectJ不是Spring框

架的组成部分，它是一个独立的AOP框架，因此在使用AspectJ之前，需

要导入和AspectJ相关的依赖。AspectJ和Spring结合使用，使AOP操作更

加方便、高效。

**Aspectj切入点语法**

语法格式: execution(修饰符列表 返回值类型 全类名.方法名(参数列表)

抛出的异常)。

解析: ”全类名.方法名(参数列表)” 表示要增强哪个类中的哪个方法，

如果方法中的参数任意，用 .. 来表示。

修饰符列表和抛出的异常可以省略。返回值不能省略，如果返回值任意，

用 \* 来表示。

eg : execution(\* com.holun.User.add(..)) 表示对com.holun.User类中的

add方法进行增强（拓展功能）。\* 表示add方法的返回值任意，.. 表示

add方法的参数任意。

execution(\* com.holun.User.\*(..)) 表示对com.holun.User类中的所有方

法进行增强（拓展功能）。

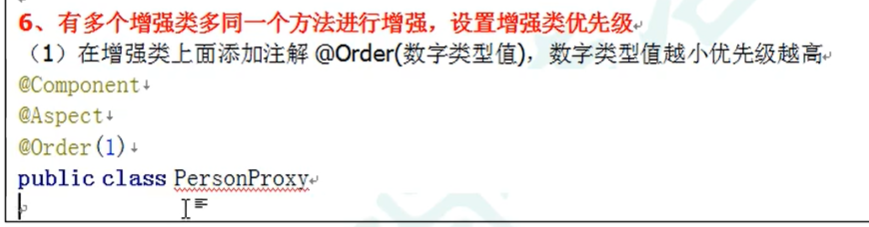


**实现AOP的两种方法**

可以基于XML配置文件实现AOP和基于注解实现AOP。

- 使用原生的Sping API完成切面

- 自定义切面



**在Spring中整合mybatis**

步骤：

1. 导入相关jai包(依赖)

junit

mybatis

mysql数据库驱动

spring相关的jar，比如Spring-jdbc

aspectJ (aop织入需要的依赖）

mybatis-spring

1. 编写配置文件

spring-dao.xml（该文件就相当于MybatisUtil）

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- 这里使用Spring内置的jdbc 使用Spring的数据源替换原先mybatis中配置的数据源。 -->  
 <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
 <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />  
 <property name="url" value="jdbc:mysql://localhost:3306/mydatabase2?useSSL=false&amp;serverTimezone=GMT%2B8" />  
 <property name="username" value="root" />  
 <property name="password" value="haolun" />  
 </bean>  
  
 <!--  
 在mybatis中是使用SqlSessionFactoryBuilder来创建 SqlSessionFactory  
 而在 MyBatis-Spring 中，是使用 SqlSessionFactoryBean来创建 SqlSessionFactory  
 -->  
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <!--注意：SqlSessionFactory 需要一个 DataSource（数据源）。这可以是任意的 DataSource-->  
 <property name="dataSource" ref="dataSource" />  
 <!-- 绑定mybatis核心配置文件 -->  
 <property name="configLocation" value="classpath:mybatis-config.xml" />  
 <!-- 指定Mapper(映射文件)的位置 -->  
 <property name="mapperLocations" value="classpath:mapper/\*.xml" />  
 </bean>  
  
 <!--  
 使用sqlSessionFactory创建SqlSessionTemplate, SqlSessionTemplate就是 mybatis中的SqlSession  
 SqlSessionTemplate（SqlSession模板）类实现了SqlSession接口  
 -->  
 <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate" >  
 <!-- 只能使用构造器注入sqlSessionFactory,因为SqlSessionTemplate类中没有set方法 -->  
 <constructor-arg index="0" ref="sqlSessionFactory" />  
 </bean>  
  
</beans>

mybatis-config.xml

<?xml version="1.0" encoding="UTF-8" ?>  
<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd">  
  
<configuration>  
  
 <typeAliases>  
 <!-- 给com.holun.entity包下的所有实体类起别名 -->  
 <package name="com.holun.entity" />  
 </typeAliases>  
  
</configuration>

applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <import resource="spring-dao.xml" />  
</beans>