# Spring学习笔记

## SpringIoc

### 1.1下载jar

https://maven.springframework.org/release/org/springframework/spring/

spring-framework-4.3.9.RELEASE-dist.zip

开发spring至少需要使用的jar(5个+1个):

spring-aop.jar 开发AOP特性时需要的JAR

spring-beans.jar 处理Bean的jar <bean>

spring-context.jar 处理spring上下文的jar <context>

spring-core.jar spring核心jar

spring-expression.jar spring表达式

三方提供的日志jar

commons-logging.jar 日志

### 1.2编写配置文件

ApplicationContext conext =new ClassPathXmlApplicationContext("applicationContext.xml") ;

//执行从springIOC容器中获取一个 id为student的对象

Student student = (Student)conext.getBean("student") ;

### 1.3依赖注入3种方式

1.set注入：通过setXxx()赋值

赋值，默认使用的是 set方法();

依赖注入底层是通过反射实现的。

<bean id=*"student"* class=*"com.spring.demo.student"*>

<property name=*"age"* value=*"1"*></property>

<property name=*"name"* value=*"ygf"*></property>

<property name=*"flag"* value=*"true"*></property>

</bean>

2.构造器注入：通过构造方法赋值

<constructor-arg value=*"24"* name=*"age"*></constructor-arg>

<constructor-arg value=*"shibc"* name=*"name"*></constructor-arg>

需要注意：如果 <constructor-arg>的顺序 与构造方法参数的顺序不一致，则需要通过type或者index或name指定。

3.p命名空间注入

引入p命名空间

xmlns:p="http://www.springframework.org/schema/p"

<bean id=*"teacher"* class=*"org.lanqiao.entity.Teacher"* p:age=*"25"* p:name=*"ww"* ></bean>

简单类型：

p:属性名="属性值"

引用类型（除了String外）：

p:属性名-ref="引用的id"

注意多个 p赋值的时候 要有空格。

注意：

无论是String还是Int/short/long，在赋值时都是 value="值" ，

因此建议 此种情况 需要配合 name\type进行区分

### 1.4注入各种集合数据类型

List Set map properties

#### List

<property name=*"listElement"*>

<list>

<value>足球</value>

<value>篮球足球</value>

<value>乒乓球</value>

</list>

</property>

#### Set

<property name=*"setElement"*>

<set>

<value>足球2</value>

<value>篮球足球2</value>

<value>乒乓球2</value>

</set>

</property>

#### Array

<property name=*"arrayElement"*>

<array>

<value>足球1</value>

<value>篮球足球1</value>

<value>乒乓球1</value>

</array>

</property>

#### Map

<property name=*"mapElement"*>

<map>

<entry>

<key>

<value>foot</value>

</key>

<value>足球3</value>

</map>

</property>

注意: set、list、数组 各自都有自己的标签<set> <list> <array>，但是也可以混着用

给对象类型赋值null ：

<property name="name" >

<null/> -->注意 没有<value>

</property>

赋空值 ""

<property name="name" >

<value></value>

</property>

在ioc中定义bean的前提：该bean的类 必须提供了 无参构造

自动装配（只适用于 ref类型 ）：

约定优于配置

自动装配：

<bean ... class="org.lanqiao.entity.Course" autowire="byName|byType|constructor|no" > byName本质是byId

byName: 自动寻找：其他bean的id值=该Course类的属性名

byType: 其他bean的类型(class) 是否与 该Course类的ref属性类型一致 （注意，此种方式 必须满足：当前Ioc容器中 只能有一个Bean满足条件 ）

constructor： 其他bean的类型(class) 是否与 该Course类的构造方法参数 的类型一致；此种方式的本质就是byType

可以在头文件中 一次性将该ioc容器的所有bean 统一设置成自动装配：

<beans xmlns="http://www.springframework.org/schema/beans"

...

default-autowire="byName">

自动装配虽然可以减少代码量，但是会降低程序的可读性，使用时需要谨慎。

使用注解定义bean：通过注解的形式 将bean以及相应的属性值 放入ioc容器

<context:component-scan base-package="org.lanqiao.dao">

</context:component-scan>Spring在启动的时候，会根据base-package在 该包中扫描所有类，查找这些类是否有注解@Component("studentDao"),如果有，则将该类 加入spring Ioc容器。

@Component细化：

dao层注解：@Repository

service层注解：@Service

控制器层注解：@Controller

### 1.5使用注解实现事务（声明式事务）

1. jar包

spring-tx-4.3.9.RELEASE

ojdbc.jar

commons-dbcp.jar 连接池使用到数据源

commons-pool.jar 连接池

spring-jdbc-4.3.9.RELEASE.jar

aopalliance.jar

1. 配置
2. 增加事务tx的命名空间

<!-- 配置数据库相关 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*>

<property name=*"driverClassName"* value=*"oracle.jdbc.OracleDriver"*></property>

<property name=*"url"* value=*"jdbc:oracle:thin:@127.0.0.1:1521:ORCL"*></property>

<property name=*"username"* value=*"scott"*></property>

<property name=*"password"* value=*"tiger"*></property>

<property name=*"maxActive"* value=*"10"*></property>

<property name=*"maxIdle"* value=*"6"*></property>

</bean>

<!-- 配置事务管理器txManager -->

<bean id=*"txManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 增加对事务的支持 -->

<tx:annotation-driven transaction-manager=*"txManager"* />

将需要 成为事务的方法 前增加注解：

@Transactional(readOnly=false,propagation=Propagation.REQUIRED)

## SpringAop

切入点和切面

### 2.1通知类型

**前置通知[Before advice]**：在连接点前面执行，前置通知不会影响连接点的执行，除非此处抛出异常。   
**后置**通知**[After returning advice]**：在连接点正常执行完成后执行，如果连接点抛出异常，则不会执行。   
**异常通知[After throwing advice]**：在连接点抛出异常后执行。

**最终通知[After (finally) advice]**：在连接点执行完成后执行，不管是正常执行完成，还是抛出异常，都会执行返回通知中的内容。   
**环绕通知[Around advice]**：环绕通知围绕在连接点前后，比如一个方法调用的前后。这是最强大的通知类型，能在方法调用前后自定义一些操作。环绕通知还需要负责决定是继续处理join point(调用ProceedingJoinPoint的proceed方法)还是中断执行。

### 2.2实现接口实现通知

前置通知实现接口:MethodBeforeAdvice

//普通类 ->前置通知

public class LogBefore implements MethodBeforeAdvice{

//前置通知的具体内容

@Override

public void before(Method method, Object[] args, Object target) throws Throwable {

System.out.println("前置通知...");

}

配置:

<!-- =========连接线的一方========= -->

<bean id="logBefore" class="org.lanqiao.aop.LogBefore">

</bean>

<!-- 将addStudent() 和 通知 进行关联 -->

<aop:config>

<!-- 配置切入点 （在哪里执行通知 ） -->

<!-- =========连接线的另一方========= -->

<aop:pointcut expression="execution(public void org.lanqiao.service.impl.StudentServiceImpl.deleteStudentByNo(int)) or execution(public \* org.lanqiao.service.impl.StudentServiceImpl.addStudent(..))" id="poioncut"/>

<!-- advisor:相当于 链接切入点 和切面的线 -->

<!-- =========连接线========= -->

<aop:advisor advice-ref="logBefore" pointcut-ref="poioncut"/>

</aop:config>

后置通知实现接口:AfterReturningAdvice

配置同上

异常返回通知实现接口:ThrowsAdvice

最终实现接口:

环绕通知实现接口:MethodInterceptor

**public** **class** LogAround **implements** MethodInterceptor{

@Override

**public** Object invoke(MethodInvocation invocation) **throws** Throwable {

Object result = **null** ;

**try** {

//方法体2...

System.***out***.println("用环绕通知实现的[前置通知]...");

// invocation.proceed() 之前的代码：前置通知

result = invocation.proceed() ;//控制着目标方法的执行 ，addStudent()

//result 就是目标方法addStudent()方法的返回值

// invocation.proceed() 之后的代码：后置通知

System.***out***.println("用环绕通知实现的[后置通知]...:");

System.***out***.println("-----------------目标对象target"+invocation.getThis()+",调用的方法名："+invocation.getMethod().getName()+",方法的参数个数："+invocation.getArguments().length+",返回值："+result);

} **catch** (Exception e) {

//方法体3...

//异常通知

System.***out***.println("用环绕通知实现的[异常通知]...");

}

**return** result;

}

}

配置

<!-- =========连接线的一方========= -->

<bean id="logaround" class="com.shibc.aop.xml.LogAround"></bean> <!-- 将addStudent() 和 通知 进行关联 -->

<aop:config>

<!-- 配置切入点 （在哪里执行通知 ） -->

<!-- =========连接线的另一方========= -->

<aop:pointcut expression="execution(public void org.lanqiao.service.impl.StudentServiceImpl.deleteStudentByNo(int)) or execution(public \* org.lanqiao.service.impl.StudentServiceImpl.addStudent(..))" id="poioncut"/>

<!-- advisor:相当于 链接切入点 和切面的线 -->

<!-- =========连接线========= -->

<aop:advisor advice-ref="logaround" pointcut-ref="poioncut"/>

</aop:config>

### 2.3通过注解实现通知

配置:

<!-- 开启注解对AOP的支持 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

@Component

@Aspect

**public** **class** LogAspectAnnotation {

@Before("execution(public \* com.shibc.service.impl.IStudentServiceImpl.addStudent(..))")

**public** **void** befor(JoinPoint jp ) {

System.***out***.println("《注解形式-前置通知》：目标对象："+jp.getTarget()+",方法名："+jp.getSignature().getName() +",参数列表："+ jp.getArgs().length );

}

@AfterReturning("execution(public \* com.shibc.service.impl.IStudentServiceImpl.queryStudent(..))")

**public** **void** after(JoinPoint jp) {

System.***out***.println("《注解形式-前置通知》：目标对象："+jp.getTarget()+",方法名："+jp.getSignature().getName() +",参数列表："+ jp.getArgs().length );

}

@Around("execution(public \* com.shibc.service.impl.IStudentServiceImpl.deleteStudent(..))")

**public** **void** around(ProceedingJoinPoint jp){

System.***out***.println("注解形式【环绕】方法之前：前置通知");

**try** {

jp.proceed();

System.***out***.println("注解形式【环绕】方法之前：后置通知");

} **catch** (Throwable e) {

System.***out***.println("注解形式【环绕】方法之前：异常通知");

}**finally** {

System.***out***.println("注解形式【环绕】方法之前：最终通知");

}

}

}

配置:

<!-- 开启注解对AOP的支持 -->

<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

### 2.4通过配置实现通知

1. 普通类

**public** **class** LogSchema {

//后置通知方法 :JoinPoint适用于注解

**public** **void** afterReturning(JoinPoint jp,Object returnValue) **throws** Throwable {

System.***out***.println("》》》》》》》》》》》后置通知：目标对象："+jp.getThis()+",调用的方法名："+jp.getSignature().getName()+",方法的参数个数："+jp.getArgs().length+"，方法的返回值："+returnValue);

}

**public** **void** before() {

System.***out***.println("》》》》》》》》》》》前置通知...");

}

**public** **void** whenException(JoinPoint jp,NullPointerException e) {

System.***out***.println(">>>>>>>>>>>>>>>>异常：" +e.getMessage());

}

//注意：环绕通知 会返回目标方法的返回值，因此返回值为Object

**public** Object around(ProceedingJoinPoint jp) {

System.***out***.println("''''''''''''''''''环绕通知：前置通知");

Object result = **null** ;

**try** {

result = jp.proceed() ;//执行方法

System.***out***.println("'''''''''"+jp.getSignature().getName()+","+result);

System.***out***.println("''''''''''''''''''环绕通知：后置通知");

}**catch**(Throwable e) {

System.***out***.println("''''''''''''''''''环绕通知：异常通知");

}

**return** result ;

}

}

1. 配置

<!-- 基于Schema配置 -->

<bean id=*"logSchema"* class=*"com.shibc.aop.xml.LogSchema"*></bean>

<aop:config>

切入点（连接线的一端：业务类的具体方法）

<aop:pointcut expression=*"execution(public \* com.shibc.service.impl.IStudentServiceImpl.addStudent(..))"* id=*"pcShema"*/>

（连接线的另一端：通知 类）

<aop:advisor advice-ref=*"logSchema"* pointcut-ref=*"pcShema"* />

schema方式

<aop:aspect ref=*"logSchema"*>

连接线：连接 业务 addStudent 和 通知before

<aop:before method=*"before"* pointcut-ref=*"pcShema"*/>

连接线：连接 业务 addStudent 和 通知afterReturning

<aop:after-returning method=*"afterReturning"* returning=*"returnValue"* pointcut-ref=*"pcShema"*/>

<aop:after-throwing method=*"whenException"* pointcut-ref=*"pcShema"* throwing=*"e"*/>

环绕

<aop:around method=*"around"* pointcut-ref=*"pcShema"* />

</aop:aspect>

</aop:config>

### spring进阶

Spring IoC容器？

两种形式：

xml配置文件：applicationContext.xml

存bean: <bean id class>

取bean:

ApplicationContext context= new ClassPathXmlApplicationContext("applicationContext.xml");

context.getBean();

注解：带有@Configuration注解的类（配置类）

注意：两种形式获取的Ioc容器是 独立的

存bean

XXX

取bean

ApplicationContext context = new AnnotationConfigApplicationContext(MyConfig.class) ;

bean: 类

a. 非三层组件 ：Student\Teacher\配置信息 : @Bean

b. 三层组件 ：Controller、Service、Dao ：@Component(@Controller、@Service、@Repository)

三层组件

a.加入@Component等注解

b.配置，让容器识别注解

i.XML形式：通过扫描器 将@Component等注解所在包 扫描

<context:component-scan base-package="com.yanqun" >

</context:component-scan>

ii.注解

@Configuration

@ComponentScan(value="com.yanqun")

public class MyConfig {...}

component-scan指定规则：

FilterType ：

ANNOTATION：指定类型 (@Compent @Controller @Service @Repository)

ASSIGNABLE\_TYPE :自定义类的类中选 StudentDao

ASPECTJ,

REGEX,

CUSTOM：自定义形式 ：a.@ComponentScan.Filter(type= FilterType.CUSTOM,classes ={ MyFilter.class })},useDefaultFilters = false)

b.MyFilter implements TypeFilter 重写match()方法，如果返回值true

---

@Component(@Controller、@Service、@Repository)：三层组件

非三层组件

---

--bean的作用域

scope: singleton| prototype

执行时机（产生bean的时机）：

singleton：容器在初始化时，就会创建对象（唯一的一个）；以后再getBean时，不再产生新的bean。singleton也支持延迟加载（懒加载）：在第一次使用时产生。 @Lazy

prototype：容器在初始化时，不创建对象；只是在每次使用时（每次从容器获取对象时 ，context.getBean(Xxxx)）,再创建对象;并且 每次getBean()都会创建一个新的对象。

--条件注解 Spring Boot

可以让某一个Bean 在某些条件下 加入Ioc容器，其他情况下不加IoC容器。

a.准备 bean

b.增加条件Bean：给每个Bean设置条件 ，必须实现Condition接口

c.根据条件，加入IoC容器

--回顾给IoC加入Bean的方法

注解 ：全部在@Congiration配置中设置：

三层组件： 扫描器 + 三层注解

非三层组件： ① @Bean+返回值

②@import

③FactoryBean(工厂Bean)

@import使用：

①直接编写到@Import中，并且id值 是全类名

②自定义ImportSelector接口的实现类，通过selectimports方法实现（方法的返回值 就是要纳入IoC容器的Bean） 。 并且 告知程序 自己编写的实现类。 @Import({Orange.class,MyImportSelector.class})

③编写ImportBeanDefinitionRegistrar接口的实现类，重写方法

@Import({Orange.class,MyImportSelector.class，ImportBeanDefinitionRegistrar.class})

--FactoryBean(工厂Bean)

1.准备bean。实现类和重写方法 2.注册bean。注册到@Bean中

注意：需要通过&区分 获取的对象是哪一个 ： 不加&,获取的是最内部真实的Apple；

如果加了&，获取的 是FacotryBean

--IoC容器：初始化容器、..使用容器、销毁容器

---Bean的生命周期：创建(new ...)、初始化（赋初值）、 ....、销毁 (servlet)

方法一: Student.java

适用于：@Bean+返回值方式

init destroy

xml:

init-method="myInit" destroy-method="myDestroy"

注解：

@Bean(value="stu",initMethod = "myInit",destroyMethod = "myDestroy")

IoC容器在初始化时，会自动创建对象(构造方法) ->init ->.....->当容器关闭时 调用destroy...

方法二：

三层注解 （功能性注解、MyIntToStringConverter.java）：@Controller、@Service、@Repository、@Component

-->三层注解（功能性注解【三层、功能性类】）

三层组件： 扫描器 + 三层注解（4个）

JAVA规范 ：JSR250

1.将响应组件 加入 @Component注解、 给初始化方法加@PostConstruct、给销毁方法加@PreDestroy

@PostConstruct：相当于方法一的init

@PreDestroy：相当于方法一的destroy

如果要获取@Component注解中的bean，那么该Bean的名字就是@Component（value="xxx"）的value值

方法三：两个接口

接口：适用于三层组件（扫描器+三层组件）

InitializingBean初始化

DisposableBean 销毁

初始化：只需要 实现InitializingBean中的afterPropertiesSet()方法

销毁：实现DisposableBean 中的destroy()方法

问题：要在SPring IOC容器中操作：操作方式 对象：Bean+返回 ，三层组件

->如果是注解形式 ， 随便写一个方法 ，然后加上相应注解即可

如果是接口形式，必须 实现接口中规定的方法

方法四：（给容器中的所有Bean加初始化、销毁）一个接口

接口：适用于三层组件

接口BeanPostProcessor：拦截了所有中容器的Bean

---自动装配 : 三层组件(4个注册+扫描器)

@Autowired

Controller->Service->Dao

三层组件

通过@Autowired从Ioc容器中 根据类型自动注入（没有调用setXxx()方法）

-如果@Autowired在属性前标注，则不调用setXxx；如果标注在setXxx前面 ，则调用setXxx

-不能放在方法的参数前

@Autowired

private Xxx xx;

public void aa()

{

}

@Autowired

public void setXxx(xx xx)

{

}

Bean+返回值：

@Autowired 在方法的参数前（也可以省略）、方法前 （构造方法：特殊，如果只有一个有参构造方法，则构造方法前的@Autowired也可以省略）

public void setXxx(@Autowired xx xx)

{

}

之前：@Autowired 根据类型匹配：

三层注入方式/@Bean+返回值

1.如果有多个类型相同的，匹配哪个？

报错。 /默认值@primary

2.能否根据名字匹配？

可以，结合 @Qualifier("stuDao2")使用。

3.如果有0个类型相同，默认报错；可以修改成不注入（null），@Autowired(required=false)

自动注入方式一： @Autowired (Spring) ，默认根据类型

自动注入方式二 @Resource（JSR250），默认根据名字 （如果 有名字，根据名字匹配；如果没有名字，先根据名字查找，如果没找到，再根据类型查找）；也可以通过name或type属性 指定根据名字 或类型找。

自动注入方式一： @Inject（JSR330），额外引入javax.inject.jar，默认根据类型匹配

---利用Spring底层组件进行开发 (三层组件)

能够供我们使用的组件，都是Aware的子接口，即XxxxAware

以ApplicationContextAware为例:实现步骤

a.实现ApplicationContextAware，

b.重写其中的方法，都包含了一个对象。只需要将该对象 赋值到属性中即可

有什么用：例如ApplicationContextAware，可以通过该接口 获取到Ioc容器对象。

执行时间： 如果在main（）中new Ioc容器： 先执行ApplicationContextAware实现类中的方法，通过该方法传入IoC容器 供我们自己使用； 然后再将该容器通过new返回给用户

BeanNameAware：

--环境切换：@Profile

Spring:切换环境

激活方式一：

-Dspring.profiles.active=@Profile环境名

-Dspring.profiles.active=myApple

数据库环境：

@Profile

127.0.0.1 scott tiger

@Profile

192.168...

激活方式二：

硬编码

坑：错误写法

ApplicationContext context = new AnnotationConfigApplicationContext(MyConfig.class) ;

ConfigurableEnvironment environment = (ConfigurableEnvironment)context.getEnvironment();

environment.setActiveProfiles("myBanana");

其中AnnotationConfigApplicationContext中有一个refresh()操作：会将我们设置的一些参数还原

没激活 |->进行激活 ->刷新 ->没激活

流程调整：

没激活->进行激活 | ->刷新

什么时候设置 保存点|： 配置类的编写处

IoC容器在使用时必须refresh() ;如果是有参构造，内部已经刷新；如果无参构造，需要手工刷新。

---Spring重要组件

接口BeanPostProcessor：拦截了所有中容器的Bean，并且可以进行bean的初始化 、销毁

创建->初始化->...-》销毁

BeanPostProcessor

BeanFactoryPostProcessor：拦截了容器

BeanDefinitionRegistryPostProcessor：即将被加载之前（解析之前，称为BeanDefination对象之前）

BeanDefinitionRegistryPostProcessor(a) -》加载bean->BeanFactoryPostProcessor(b)->实例化bean->BeanPostProcessor

同一个方法 在不同地方（类、接口）的出现时机问题：a继承b，因此a中必然包含b中的方法(记c )：虽然a和b中都有c，但是 因此c出现的时机不同， 则c的执行顺序也不同： 如果是在a中出现，则先执行；如果是在b中执行 则后执行

在同一个地方（类、接口），的不同方法的出现时机问题

监听器：

可以监听事件 ，监听的对象必须是 ApplicationEvent自身或其子类/子接口

方式一：

1必须实现ApplicationListener接口，

方式二：注解

(语法上 可以监听任意事件，但建议 ApplicationEvent自身或其子类/子接口)

Spring：要让SPring识别自己，必须加入IOc容器（Bean+返回值| 注解+扫描器）

自定被监听事件

a.自定义类 实现ApplicationEvent接口（自定义事件）

b.发布事件

context.publishEvent(自定义事件);