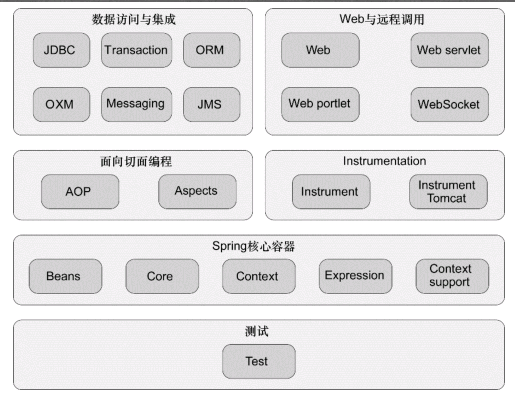
# Spring

## Spring的模块



1. Spring核心容器：核心容器是Spring框架的重要组成部分，也可以说是Spring框架的基础。他在整个框架中的作用是负责管理对象的创建，管理，配置等等的操作。其主要包含spring-core，spring-beans，spring-context，spring-expression，spring-context-support组件。

Spring的其他模块都是构建在核心容器之上的

1. 面向切面编程（aop）：这个模块是spring应用系统中开发切面的基础，利用面向切面编程，可以帮助应用对象解耦。例如，将日志，事务与具体的业务逻辑解耦。其主要包含spring-aop，spring-aspects组件。
2. Instrumentation：该模块提供了为JVM添加代理的功能，它为tomcat提供了一个织入代理，能够为tomcat传递文件，就像这些文件能够被类加载器加载的一样，该模块包含spring-instrument，spring-instrument-tomcat组件，使用较少，不必过分关注。
3. 数据访问与集成：Spring框架为了简化数据访问的操作，包装了很多关于数据访问的操作，提供了相应的模板，消除使用jdbc过程中的样板式代码。同时还提供了使用ORM框架的能力，可以与很多流行的ORM框架进行整合，如hibernate，mybatis等等的著名框架。还实现了数据事务的能力，能够支持事务。包含spring-jdbc，spring-tx，spring-orm，spring-oxm，spring-jms，spring-messaging组件。
4. Web和远程调用：Spring框架支持Web开发，以及与其他应用远程交互调用的方案。包含spring-web，spring-webmvc，spring-websocket，spring-webmvc-portlet组件。
5. Spring测试：Spring框架提供了测试的模块，可以实现单元测试，集成测试等等的测试流程，整合了JUnit或者TestNG测试框架。包含spring-test组件。

其他功能应用

Spring web flow

Spring web service

Spring security

Spring Integration

Spring data

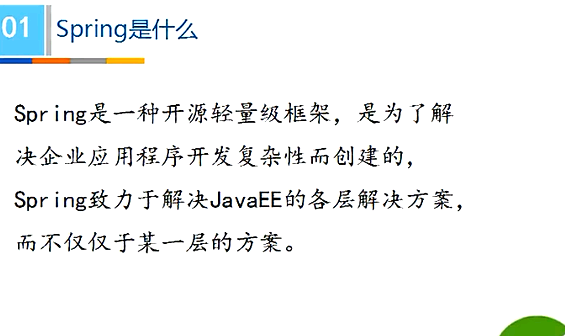
Spring boot:大量依赖于自动配置技术，能够消除大部分的spring配置

## 简化开发

Spring是一个开源框架，为了解决企业级应用开发的复杂性而创建的；核心目的就是**简化开发**；

为简化开发spring采取了4中策略

1. 基于pojo的轻量级和最小侵入性编程()
2. 通过依赖注入和面向接口实现松耦合
3. 基于切面和惯例进行声明式编程
4. 通过切面和模板减少样板式代码



### 最小侵入

pojo就是简单的java对象，有属性getter和setter方法；在基于spring构件的应用中，它的类通常在没有任何痕迹表名你使用了spring，最坏的场景就是使用spring的配置或者注解，pojo类的结构不会依赖spring，spring的非依赖模式意味着一个类在spring应用和非spring应用中都具有相同的作用

### 依赖注入-实现松耦合

任何具有实际意义的应用都是会由两个或者更多的类组成，这些类相互之间进行协作来完成业务逻辑；在非spring应用中，每个对象负者管理和自己相互协助的对象（它所依赖的对象）的引用，这样会导致高度耦合和难以测试的代码

在spring中通过di，对象的依赖关系由系统中的负责协调的对象的第三方组件在创建对象的时候进行设定（通过spring创建对象，并且完成依赖对象的引用赋值）；di让相互协作的对象的关系松耦合

装配：建立引用组件之间协作的行为

依赖注入的方式

1. setXXX方法注入属性注入，即通过setXxx()方法注入Bean的属性值或依赖对象，由于属性注入方式具有可选择性和灵活性高的优点，因此属性注入是实际应用中最常采用的注入方式。
2. 通过构造函数来完成注入，通过构造函数注入bean的属性值或者依赖对象，在配置文件中配置该类的bean，并配置构造器，在配置构造器中用到了<constructor-arg>节点，该节点有四个属性：

index是索引，指定注入的属性位置，从0开始；

type是指该属性所对应的类型；

ref 是指引用的依赖对象；

value 当注入的不是依赖对象，而是基本数据类型时，就用value；

1. 注解注入

@Resource

@Autowired

装配方式

1. 基于xml配置的装配
2. 基于java类的配置
3. 隐式的bean发现机制和自动装配

Spring从两个角度来实现自动化装配：

组件扫描(ComponentScan)：自动发现应用上下文中所创建的bean

自动装配(Autowired):自动满足bean之间的依赖

基于xml配置和java类配置这2中方式带来的收益是相同的；

Java配置类的优势在于初始化bean时通过代码方式控制bean初始化的整体逻辑

Xml实现自动装配

装配分为四种：byName, byType, constructor, autodetect。

具体选择哪一种装配方式，需要配置<bean>标签的autowire属性，如果没有配置，默认是byName类型，就是会根据属性的名字来进行自动装配。

1）byName就是会将与属性的名字一样的bean进行装配。

2）byType就是将同属性一样类型的bean进行装配。

3）constructor就是通过构造器来将类型与参数相同的bean进行装配。

4）autodetect是constructor与byType的组合，会先进行constructor，如果不成功，再进行byType。

No;不自动装配

注解实现自动装配，注解方式：：@Autowired，@Qualifier，@Named，@Resource，@Inject。

@Autowired注解是byType类型的

@Inject是byType类型的，是java ee提供的

@Qualifier注解使用byName进行装配

@Autowired与@Qualifier是spring提供的，@Inject与@Named是java ee的

@Resouce注解， 这个注解也是java ee的，也是byName类型的

自动检测配置，也是springmvc中最牛的一项功能：只要一个配置<context:component-scan base-package="">，base-package属性指定要自动检测扫描的包。该配置会自动扫描指定的包及其子包下面被构造型注解标注的类，并将这些类注册为spring bean，这样就不用在配置文件一个一个地配置成bean标签。构造型注解包括：@Controller，@Component，@Service，@Repositoryt标注的自定义注解。

控制反转：不再需要自己创建对象，只需要通过配置或者注解标记出需要创建的那些实例bean，那么spring启动时就会自动去创建那些实例，并且完成依赖的注入；将实例化的bean根据scope属性来决定是否缓存bean

### 基于切面进行声明式编程

面向切面编程可以将应用的功能分离出来形成可重用的组件

系统由许多不同的组件组成，每一个组件负责一块特定的功能，除了实现自身核心的功能之外，这些组件还需要一些额外的功能，比如日志，事务管理，安全认证等业务，需要将这些额外的功能添加到系统核心业务中去，这些业务就被称为**系统横切关注点**（除开系统本身的核心业务之外的其他的服务型的逻辑功能），因为它们会跨越系统的多个组件。

将这些关注点分散到各个组件中去，就会让代码更加复杂：实现关注点的代码将会重复出现，组件也会因为这些代码变得混乱

Aop使得这些服务模块化，并以申明式的方式将它们应用到需要的组件中去；让各个业务模块只需要关注自身，不需要了解系统服务的实现

### 基于切面消除样板式代码

通常为了实现通用和简单的任务，不得不重复的写一下同样的代码，比如使用jdbc访问数据库查询数据，加载驱动，获取连接对象，获取预处理对象，执行查询，关闭连接，释放资源，在执行不同查询的时候，会出现大量重复的jdbc操作只有少量的代码和具体的业务相关，其他的代码都是样板式代码；spring通过模板封装的方式来消除样板式代码，比如spring的JdbcTemplate操作数据库就避免了jdbc样板式代码

## Spring容器

在基于spring应用中，对象存放在spring容器中，spring容器负者创建，装配，配置对象，并管理他们的整个生命周期

Spring有多个容器，spring自带多个容器的实现，可以分为2类：

BeanFactory：bean工厂，最简单的spring容器，提供了基本的di支持

ApplicationContext：应用上下文，基于BeanFactory构建，并提供了应用框架级别的服务，例如从属性文件中解析文本信息以及发布应用事件给感兴趣的事件监听者

### Spring的应用上下文

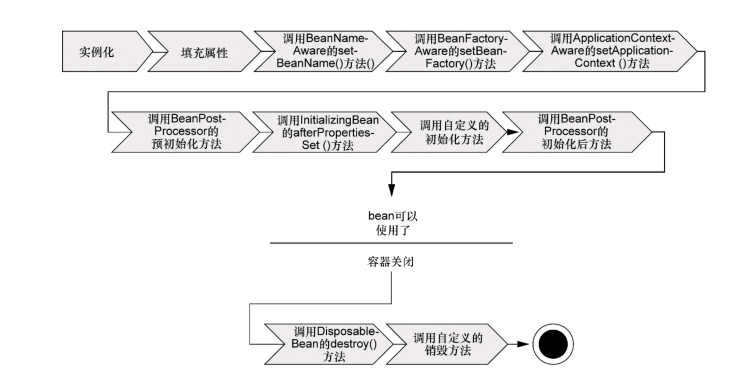
1. AnnotationConfigApplicationContext:从一个或者多个基于java的配置类中加载spring应用上下文
2. AnnotationConfigWebApplicationContext：从一个或者多个基于java的配置中加载spring web应用的上下文
3. ClassPathXmlApplicationContext：从类路径下的一个或者多个Xml文件中加载spring应用上下文
4. FileSystemXmlApplicationContext：从文件系统下的一个或者多个xml文件中加载spring应用上下文
5. XmlWebApplicationContext：从web应用下的一个或者多个xml配置文件中加载上下文

Spring中定义bean的方式

1. 基于Xml的配置，xml中定义bean标签，指定创建bean对应的类型
2. 基于注解，在类上使用@Component注解，
3. 基于使用@Configuration修饰的java配置类，使用@Bean注解来创建bean

当引用上下文初始化完成后，通过getBean方法从spring容器中取出bean

### Spring bean的生命周期



1. spring对bean进行实例化
2. spring将值和bean的引用注入到bean对应的属性中
3. 如果bean实现了BeanNameaware接口，那么就执行setBeanName方法
4. 如果bean实现了BeanFactoryAware接口，那么就是执行setBeanFactory方法
5. 如果实现了ApplicationContextAware接口，那么就是执行setApplicationContext方法
6. 如果bean实现了BeanPostProcessor接口，那么就执行postPropossBeforeInitialization方法
7. 如果bean实现了InitializaingBean接口，那么就执行afterPropertiesSet方法
8. 如果配置了init-method指定的初始化方法，就执行这个方法
9. 如果bean实现了BeanPostProcessor接口，那么就执行postPropossAfterInitialization方法

此时bean已经创建并且存放在应用上下文中，可以被使用了

Bean销毁，如果bean实现了DisposableBean接口，spring销毁时就会执行destroy方法，

如果配置了destroy-method指定的方法，那么就要执行这个方法

## Bean

Spring配置的可选方案

1. 在xml中配置
2. 在java中进行配置
3. 隐私的bean发现机制和自动装配

Spring的配置方式是可以相互搭配的，在一个应用中可以选择一些bean使用xml来配置，另一些使用java类来配置

### 自动化装配bean

Spring实现自动化装配bean

组件扫描：spring会自动发现应用上下文中所创建的bean

自动装配:spring自动实现bean之间满足的依赖

组件扫描和自动装配一起可以将显示配置降到最低

**@Component**

组件扫描就是扫描使用了@Component注解的类；这个注解表名该类会作为组件类，并且告知spring为这个类创建bean, 当@Component没有添加属性时，创建当前bean对应的id就是这个类名，首字母小写；也可以指定bean的id,如@Component(“xxx”)

默认情况下组件扫描是没有开启的，需要显示配置开启扫描组件类

开发组件扫描的方式

1.@ComponentScan注解

如果注解没有指定扫描的路径，那么就默认扫描当前类所在的包路径;

@ComponentScan的属性

basePackages:可以设置一个包路径，也可以设置多个；@ComponentScan(basePackages=”xxx”),

@ComponentScan(basePackages={“…”,”…”})

basePackageClasses:指定类或者接口所在的路径，@ComponentScan(basePackageClasses =A.class), @ComponentScan(basePackageClasses ={ A.class , B.class })

2.Xml中配置<context:component-scan base-package=”..”/>，指定扫描包路径

@Autowired

Spring的自动装配就是让spring自动满足bean依赖的一种方式，在满足依赖过程中，会在spring应用上下文中寻找匹配某个bean需要的目标bean，然后进行装配，spring使用注解@Autowired来完成装配

@Autowired注解可以用来类的任何地方（构造函数，setter方法，普通方法，成员变量），当没有满足的匹配的bean时，spring会抛出异常，为了避免异常抛出，可以在autowired注解中使用required属性，设置required=false,那么在没有找到匹配的bean时就不会抛出异常

组件扫描和自动装配在大多数情况下是很好的实现spring自动化配置的方式，但是有些时候自动化配置是不行的，比如讲第三方组件引入到应用中时，这是就没有办法通过@Component注解和@Autowired来实现自动装配，那么这就需要显示配置

### Java类装配bean

创建配置类

定义一个类，在这个类上使用@Configuration注解，那么这个类就是spring的java配置类

@Bean注解：这个注解作用在方法上，告诉spring这个方法会返回一个对象，该对象就是要注册到spring应用上下文中的bean,在方法体中实现创建bean的过程，bean的id默认就是方法名称，也可以通过name属性来指定bean的id

@Bean

Public A a(){

Return new A();

}

Bean注入

1. bean都配置在同一个类中

注入是直接调用对应创建bean的方法就完成注入

1. bean配置在不同类中或者注入其他方式创建的bean

通过方法参数传入要注入的bean，参数名称和bean的id一样，在同一个类中的bean也可以用这种方式

### Xml配置bean

声明bean

在配置文件中创建bean标签<bean />通过class属性来指定所属的类，如果没有配置id属性，那么就使用bean的权限定名来命名，名称后面跟着#0，后面的数字是用来计数的

注入bean

1. 构造器注入：使用标签<constractor-arg>表示bean的构造函数，子标签<propertier/>对应要注入的bean或者属性
2. 设置属性

在bean中直接定义子标签<properties/>来指定注入的属性或者bean,在bean类中要有对应的setter方法

在进行显示配置的时候javaConfig是更好的方案，因为它更加强大，类型安全并且对重构友好，因为是java代码，就像应用程序中的其他java代码一样

Xml创建bean是通过反射，效率比直接new要低一些

## Spring的其他装配

### profile

spring为环境相关的bean提供了根据环境决定哪些bean创建，哪些bean不创建的方案；spring引入了bean profile的功能，要使用profile，首先要将不同的bean定义整理到一个或多个profile之中，在将应用部署到每个环境时，确保对应的profile被激活

在javaConfig中使用注解@Profile(“”)来指定当前的bean属性那个profile



Profile注解用在了类上，那么这个类中的bean只有在对应的profile配置时才能被激活（也就是创建bean）

@Profile注解也可以直接使用在bean的方法上

在xml中配置

Spring的root标签beans具有属性profile，可以用来配置，表示当前配置文件中的所有bean就是要在当前profile属性执行被激活时才能被创建

也可以在bean上配置，bean有属性profile，激活指定的bean

激活Profile的方式

激活profile需要设置2个属性，spring.profiles.active和spring.profiles.default;如果设置了spring.profiles.active,那么它的值就会用来激活对应的profile，没有设置的话就检查spring.profiles.default的值来激活，如果都没有设置的话那么就没有要被激活的profile，那么创建的profile的bean就没有用了

设置这2个参数的方式

1. 作为dispatcherservlet的初始化参数
2. 作为web应用的上下文参数
3. 作为环境变量
4. 作为jvm的系统参数
5. 在集成测试类上，使用@ActiveProfiles注解设置

### 条件化bean

当希望一个bean在应用的类路径下包含特定的库时才创建，或者系统一个bean在另一个bean创建之后才创建，或者在某个环境下才创建bean,在spring4之前很难实现，但是在spring4引入了@Conditional注解，它可以用到带有@Bean注解的方法上，如果给定的条件计算为true，那么就会创建这个bean,否则就会忽略

@Conditional直接的参数是实现condition接口的类的class对象，首先通过condition的方法matches判断是否满足条件，如果满足就创建bean,不满足就不创建

@FunctionalInterface

**public** **interface** Condition {

**boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata);

}

ConditionContext的方法

BeanDefinitionRegistry getRegistry();返回BeanDefinitionRegistry，可以检查bean是否bean被定义

ConfigurableListableBeanFactory getBeanFactory();返回ConfigurableListableBeanFactory，可以检查bean是否存在

Environment getEnvironment();返回环境配置，检查是否有某些环境配置

ResourceLoader getResourceLoader();返回加载的资源

ClassLoader getClassLoader();返回类加载器

AnnotatedTypeMetadata ：用来检查@Bean注解的方法是是否还有其他的注解

### 首选bean

装配时的歧义

当一个接口有多个实现类时，通过接口类型去扫描，那么就会返回多个结果spring就会抛出异常NoUniqueBeanDefinitionException;spring提供了几种方案来解决这样的问题

1. [将首选的bean使用@Primary](mailto:1.将首选的bean使用@Primary)
2. [使用注解@Qualifier(“..”)](mailto:使用注解@Qualifier()来指定装配的bean,参数就是装配的beanid

### Bean的作用域

单例(singleton):整个应用中，只有一个bean实例（默认的作用域）

原型(prototype)：每次获取bean实例时，都会重新创建一个新的bean实例

会话(session)：在web应用中，为每个会话创建一个实例

请求(request)：在web应用中，为每个请求创建一个实例

作用域的配置

在javaconfig中，使用@Scope注解来指定作用域（@Component也是使用这个注解），在Xml中，使用bean的实现scope来指定作用域；在不指定作用域的时候都是单例的

会话和请求的作用域

在web应用中，能够实例化在会话和请求范围内共享的bean,那么将会是更有意义；比如在商城应用中，可能存在一个购物车的bean；如果是单例，那么所有用户都会往同一个购物车里面加东西，如果是原型，那么在一个地方添加了，在另一个地方继续添加，2次的购物车对象就不是同一个了；就购物车而言使用会话作为作用域是最合适的

### 运行时值注入

注入就是将一个bean装配到另一个bean中，值注入就是将一个值注入到bean的属性或者构造函数参数中，spring提供2种方式在运行时进行值注入

1属性占位符

@PropertySource,读取类路径下制定的properties文件

Spring支持将属性值定义在外部文件中，然后使用占位符将进行注入；占位符的使用形式使”${xxx}”，花括号内就是配置文件中属性名称，注入使用@Value注解；为了使用占位符必须配置PropertySourcesPlaceholderConfigurer的bean,因为这就是用来解析占位符的；在配置文件中使用<context-property-placeholder/>

2.Spring表达式语言（SpEL）

特性

1. 使用bean的id来引用bean
2. 调用方法和访问对象的属性
3. 对值进行算术，关系和逻辑运算
4. 正则表达式
5. 集合操作

SpEL的形式”#{….}”，

## Spring Aop

横切关注点：与应用的业务逻辑相分离，但是会直接嵌入到业务逻辑中，把横切关注点与业务逻辑分离是面向切面编程要解决的问题

核心业务基本上都需要一些通用的辅助功能，比如事务，安全；实现这些通过功能常见的是继承和委托，如果使用相同的基类，继承会导致一个脆弱的对象体现，使用委托需要对委托对象进行复杂调用

切面提供了取代继承和托的另一种可选方案，在许多场景下更加简洁；使用面向切面编程时，仍需要定义通用功能，但是可以通过声明的方式定义这个功能要以何种方式在何处应用，不会修改受应用的类，横切关注点可以被模块化为特殊的类，这些类就被称为切面，这样做的好处就是将每个关注点都集中于一个地方而不是分散到代码中，代码更加简洁

### AOP的术语

通知（advice）：在目标方法上增强的功能或者操作，比如安全，事务，日志

Spring定义了5中通知类型

前置通知：在目标方法执行之前执行的方法

后置通知：在目标方法执行之后执行的方法

返回通知，目标方法执行成功之后调用的方法

异常通知，在目标方法抛出异常后调用的方法

环绕通知，包裹目标方法，在环绕通知方法中执行目标方法，可以在执行前后自定义行为（也就是在环绕通知方法中同时实现前置通知和后置通知）

连接点

执行通知方法的触发点，比如调用目标方法前，后或者抛出异常这些行为都是连接点

切入点

就是要进行功能增强或者增加额外操作的方法（也就是核心业务执行的方法），通常使用明确的类和方法名，或者正则表达式定义所有匹配的类和方法名称来指定切点

切面

通知和切入点的结合，通知说明了干什么和什么时候干，切入点说明的在什么地方干（指的是目标方法）

引入

允许现有的类添加新方法或者属性，就是把切点定义通知方法用到目标类中

目标

就是指目标类，就是被通知的对象，也就是真正的业务逻辑

织入

把切面应用到目标对象并创建新的代理对象的过程，切面在指定的连接点被织入到对象中，在目标对象的生命周期中可以有多个点进行织入

编译期：切面在目标类编译时被织入，这种方式需要特殊的编译器，AspectJ的织入编译器就是以这种方式织入到切面的

类加载期：切面在被类加载器加载到jvm时被织入，

运行期：切面在运行到某个时刻被织入，springaop就是使用这种方式来实现的

### Spring中实现aop的方法

1. Xml配置

定义一起切面类，类中实现的通知方法（定义环绕通知就不用定义其他4个通知，没有定义环绕通知就定义另外几个通知）

前置通知

**public** **void** before(JoinPoint jp)

{

System.*out*.println("请求参数" + jp.getArgs());

System.*out*.println("before");

}

成功执行返回通知

参数名称要和配置文件的中Aop:after-returning的returning属性值一样

**public** **void** afterReturning(Object obj)

{ }

环绕通知

**public** **void** around(ProceedingJoinPoint j)

{

**try**

{

// 调用之前

j.proceed();// 执行proceed方法就是调用目标方法

// j.proceed(arg0)代参数的方法

// 调用之后

} **catch** (Throwable e)

{

e.printStackTrace();

}

}

在配置文件中配置这个切面类的bean，定义aop-config标签

定义aop:aspect子标签，标签属性ref值为切面类bean的id

在aop:aspect标签中定义子标签

Aop:pointcut:切入点属性为expression，

Aop:before:前置通知

Aop:after:后置通知

Aop:after-returning:成功执行后通知，returning属性指定返回结果的引用名，这名词和方法接收的要一样

Aop:after-throwing:执行异常通知：throwing属性指定抛出的异常，名词和方法接收的要一样

aop:around：环绕通知

aop:advisor

1. Aspect注解

在类上使用@Aspect注解表示当前类是切面，使用@Component表示spring能够自动创建当前类的bean

@Pointcut使用在一个方法上，注解设置的值是一个切点表达式,后面通知方法的注解就可以直接使用当前的方法名来引入切入点

在前置通知方法上使用@Before，后置通知方法使用@After，后置执行成功方法@ AfterReturning，后置执行异常方法@AfterThrowing

环绕通知@Around

在java配置类中使用@EnableAspectJAutoProxy，这个注解用来开启自动代理功能，AspectJ注解才能被使用

//@Aspect表名当前类是一个切面

@Aspect

@Component

**public** **class** Audience

{

// @Pointcut设置的值是一个切点表达式

// 定义命名的切点，在之后的aspectj注解的时候参数可以直接使用这个方法

// 定义切点一次之后就直接应用

@Pointcut("execution(\* spring.aop.DealSomeThing.func1(..))")

**public** **void** common()

{}

// 定义切点execution(\* spring.aop.DealSomeThing.\*(..))

// execution(\*

// spring.aop.DealSomeThing.func1(..)||spring.aop.DealSomeThing.func2(..))

// execution:表示方法执行时触发

// \*表示任意返回类型

// spring.aop.DealSomeThing方法所属的类

// 最后一个\*:通配符表示类中所有的方法，也可以直接写某个方法名

// ..表示方法任意的参数

// 连接符||-》or &&->and !->not

// @Before("execution(\* spring.aop.DealSomeThing.func1(..))")

@Before("common()")

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

{

System.*out*.println("------before-------");

}

@After("execution(\* spring.aop.DealSomeThing.func1(..))")

**public** **void** after()

{

System.*out*.println("------after-------");

}

@AfterThrowing(value = "execution(\* spring.aop.DealSomeThing.\*(..))", argNames = "e", throwing = "e")

**public** **void** afterThrowing(Exception e)

{

System.*out*.println("------afterThrowing-------");

}

@AfterReturning(value = "execution(\* spring.aop.DealSomeThing.\*(..))", argNames = "retVal", returning = "retVal")

**public** **void** afterRetruning(Object retVal)

{

System.*out*.println("------afterRetruning-------");

}

@Around(value = "common()")

**public** Object checkPermissionAround(ProceedingJoinPoint joinPoint)

{

**try**

{

Object[] args = joinPoint.getArgs();

joinPoint.proceed(args);

} **catch** (Throwable e)

{

}

**return** **null**;

}

}

环绕通知

环绕通知是最强大的通知方法，它能够让你编写的逻辑将被通知的目标方法完全包装起来，它实际上就像是在一个通知方法中同时编写前置通知和后置通知

@Around注解表示当前方法将会被作为环绕通知方法，这个方法要接受ProceedingJoinPoint对象作为参数，通过这个参数获取要执行的目标方法的参数，调用proceed方法就是执行目标方法

### SpringAop实现原理

使用的设计模式是代理模式

Aspect和advicor的区别

TODO

## Spring事务

事务的隔离级别

Isolation 属性一共支持五种事务设置，具体介绍如下：

DEFAULT 使用数据库设置的隔离级别 ( 默认 ) ，由 DBA 默认的设置来决定隔离级别 .

READ\_UNCOMMITTED 会出现脏读、不可重复读、幻读 ( 隔离级别最低，并发性能高 )

READ\_COMMITTED 会出现不可重复读、幻读问题（锁定正在读取的行）

REPEATABLE\_READ 会出幻读（锁定所读取的所有行）

SERIALIZABLE 保证所有的情况不会发生（锁表）

事务的传播行为

事务的7种传播级别：

1） PROPAGATION\_REQUIRED ，默认的spring事务传播级别，使用该级别的特点是，如果上下文中已经存在事务，

那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

2）PROPAGATION\_SUPPORTS ，从字面意思就知道，supports，支持，该传播级别的特点是，如果上下文存在事务，

则支持事务加入事务，如果没有事务，则使用非事务的方式执行。所以说，并非所有的包在transactionTemplate.execute中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3）PROPAGATION\_MANDATORY ， 该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！

配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4）PROPAGATION\_REQUIRES\_NEW ，从字面即可知道，new，每次都要一个新事务，该传播级别的特点是，

每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。

这是一个很有用的传播级别，举一个应用场景：现在有一个发送100个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送100封红包，然后再记录发送日志，发送日志要求100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。

怎么处理整个业务需求呢？就是通过这个PROPAGATION\_REQUIRES\_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5）PROPAGATION\_NOT\_SUPPORTED ，这个也可以从字面得知，not supported ，不支持，

当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。

所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环1000次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6）PROPAGATION\_NEVER ，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，

而PROPAGATION\_NEVER传播级别要求上下文中不能存在事务，一旦有事务，就抛出runtime异常，强制停止执行！这个级别上辈子跟事务有仇。

7）PROPAGATION\_NESTED ，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，

则嵌套事务执行，如果不存在事务，则新建事务。

Spring中的事务使用方式有声明式事务和编程式事务

声明式事务的实现：1xml配置 2使用注解

1xml配置

在xml文件中配置事务管理器bean，DataSourceTransactionManager

配置

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

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

</bean>

<!-- transaction-manager="transactionManager"可以不用配置，前提事务管理器的id必须定义transactionManager -->

<!-- spring默认找id=transactionManager的事务管理器 -->

<!-- 事务属性 -->

<tx:advice id=*"txAdvice"*>

<tx:attributes>

<!-- 如果希望get方法或者find方法是只读事务 -->

<!-- 这一事务属性表明，如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。 -->

<!-- 如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行 -->

<!-- 如果执行的方法有事务就是只读事务，如果没有事务就是没有事务 -->

<!-- <tx:method name="get\*" read-only="true" propagation="SUPPORTS" /> -->

<!-- <tx:method name="find\*" read-only="true" propagation="SUPPORTS" /> -->

<!-- 其他事务就使用默认配置 ,默认的配置必须放到最后面 -->

<!-- REQUIRED表示当前如果有事务就加入，没有就新建一个事务

SUPPORTS当前事务存在就加入，不存在就创建一个空事务-->

<tx:method name=*"\*"* propagation=*"REQUIRED"* />

</tx:attributes>

</tx:advice>

注解

开启注解

<tx:annotation-driven transaction-manager/>

使用@Transactional注解，这个注解可以用在方法上

属性

Propagation事务的传播行为

Isolation事务的隔离级别

Timeout事务起始时间

Readonly，设置事务是否只读 true表示只读，在事务设置为只读后，如果进行写操作，会抛出异常

**Spring事务架构中默认只在抛出运行时异常时和unchecked异常是标记事务回滚，也就是抛出Runtimeexception及其子类时，回滚事务；当事务的方法中抛出checkedexception时将不会回滚事务(抛出Exception就不会使用到事务)**

**Rollbackfor:实现让checkedexception异常回滚；使用方式@Transactional(rollbackFor=Exception.class)**

**Unrollbackfor:让非检查异常不回滚事务，使用方式@Transactional(Unrollbackfor =RuntimeException.class)**

编程式事务

定义DataSourceTransactionalManager的bean,获取事务管理器实例

获取事务状态，调用事务管理器的getTransaction方法获取事务状态，方法的参数是TransactionDefinition的实现类，可以为null，为null的话就内部默认使用DefaultTransactionDefinition来传入属性，调用getTransaction方法后就开启了事务

提交事务调用rollBack方法

## Springmvc

Spring mvc基于模型-视图-控制器模式实现的，用于构件灵活和松耦合的web应用

### Mvc模式

M-model:模型，应用中用来处理数据逻辑的部分，通常模型对象负责从数据库中存取数据，处理数据相关业务逻辑（对应service层，dao层）

V-view:视图，应用中处理数据显示的部分，通常视图对象依据模型数据创建，视图就是用户看到并且与之交互的界面

C-controller:控制器，应用中处理用户交互的部分，通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据；控制器接收用户的输入并调整模型和视图去完成响应的用户需求

### Springmvc的请求流程

1.用户发送请求至前端控制器DispatcherServlet

2.DispatcherServlet收到请求调用处理器映射器HandlerMapping。

3.处理器映射器根据请求url找到具体的处理器，

生成处理器执行链HandlerExecutionChain(包括处理器对象和处理器拦截器)一并返回给DispatcherServlet。

4.DispatcherServlet根据处理器Handler获取处理器适配器HandlerAdapter执行HandlerAdapter处理一系列的操作，

如：参数封装，数据格式转换，数据验证等操作

5.执行处理器Handler(Controller，也叫页面控制器)。

6.Handler执行完成返回ModelAndView

7.HandlerAdapter将Handler执行结果ModelAndView返回到DispatcherServlet

8.DispatcherServlet将ModelAndView传给ViewReslover视图解析器

9.ViewReslover解析后返回具体View

10.DispatcherServlet对View进行渲染视图（即将模型数据model填充至视图中）。

11.DispatcherServlet响应用户。

### 接收请求参数

查询参数

@RequestParam(value=”name”,defaultValue=”tom”)String name

defaultValue默认值

路径变量

1./xxx/xx?id=123

使用@RequestParam注解

2./xxx/xx/123

在@RequestMapping中使用占位符，用大括号{}括起来

@RequestMapping(“/xxx/xx/{id}”)

在方法参数中使用@pathVariable(“id”) long id来获取变量值，参数中的变量名要和RequestMapping中一样

在接收视图传来的非表单参数时，只要方法中的参数名称和表单提交时的属性名称一致，可以不使用注解直接就能获取

### 表单校验

可以在接收表单后，逐个判断属性是否合理，但是会增加许多if语句，让代码结果更加复杂

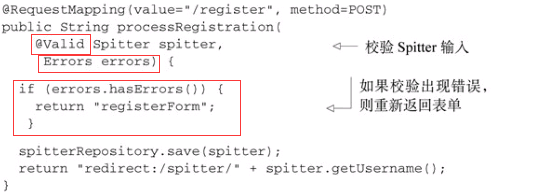
Springmvc提供了java校验api的支持，不需要额外的配置

注解列表





在接收参数时使用@Valid注解，告知spring当前对象需要验证，增加一个Errors对象，在方法中调用hasErrors方法，判断是否有不通过的属性，然后跳转到对应的视图



### 处理异常

Spring提供了多种方法将异常转换为响应

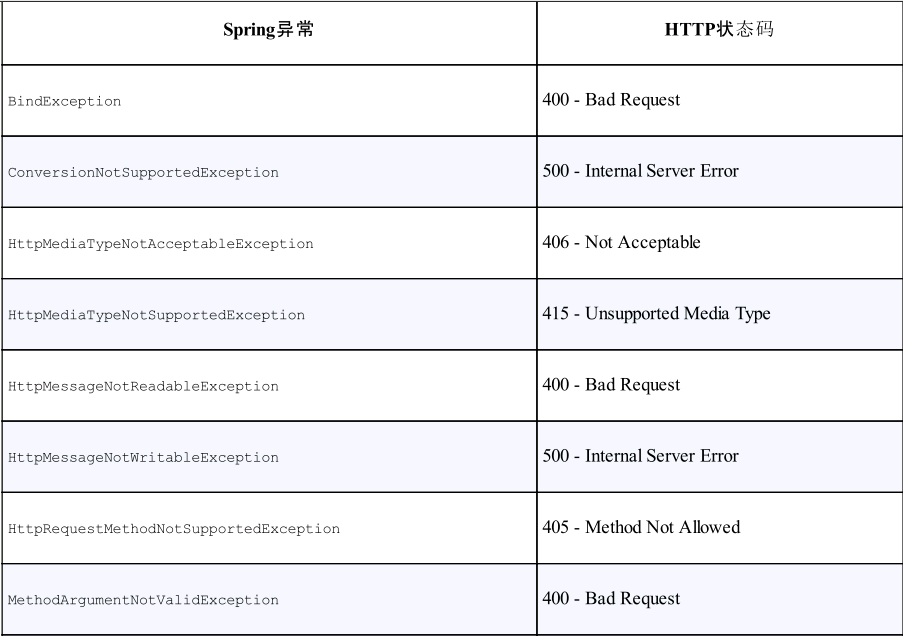
将特定的异常自动映射为指定的http状态码

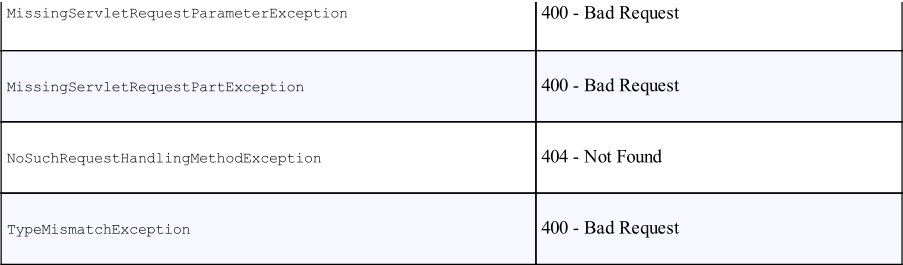
异常上可以添加@ResponseStatus注解，将其映射为某个http状态码

在方法上添加@ExceptionHandler注解来处理异常

将异常映射为状态码

Spring异常和状态码的映射表





定义一个异常，再异常上使用@ResponseStatus将异常和http状态码绑定

@ResponseStatus(value = HttpStatus.*NOT\_FOUND*, reason = "name is null")

**public** **class** NameNullException **extends** RuntimeException

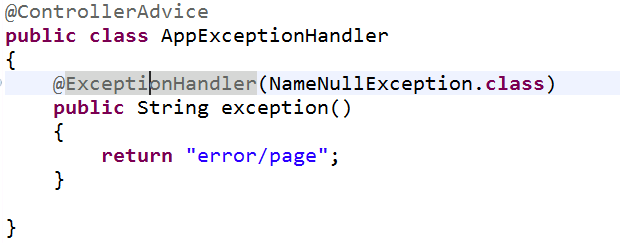
{

}

Value为异常，reason为异常说明

编写异常处理方法

定义一个异常返回类，类上使用@ControllerAdvice注解（@ControllerAdvice这个注解会将@ExceptionHandler标注的方法应用到所有带有@RequestMapping的方法中，@ControllerAdvice自带Component注解），在这个类中定义不同异常的处理方法，方法上使用@ExceptionHandler关联指定的异常



## Spring mvc rest

### rest

Rest：Representational State Transfer

Representational:rest资源可以用各种形式来进行表述，包括xml，json甚至html

State：使用rest时，我们更关注的是资源的状态而不是对资源采取的行为

Transfer：rest涉及到转移资源数据，它以某种表示性形式从一个应用转移到另一个应用

Rest就是将资源的状态以最适合客户端或者服务端的形式从服务器端转移到客户端；rest是面向资源的，强调描述应用程序的事物和名词

Rest中会有行为，它们是通过http方法来定义的，也就是get,post,put,delete,patch以及其他http方法构成的rest中的动作，这些通常会和crud进行匹配

C:post

Read:get

Update:put或者patch

Delete：delete

虽然http方法和crud动作是映射的，但是并不是严格限制的，put可以用来创建新的资源，post可以用来更新资源

### Spring对rest的支持

Spring支持以下方式创建rest资源（spring4.0版本开始）

控制器可以处理所有的http方法

借助@Pathvariable注解，控制器能够处理参数化的url

借助spring的视图和视图解析器，资源能够以多种形式进行表述，将数据渲染为xml，json等

可以使用ContentNegotiatingViewResolver来选择最合适的客户端表述

使用@ResponseBody注解和HttpMethodConverter的各种实现，能够替换基于视图渲染的方式

使用@RequestBody注解和HttpMethodConverter的各种实现，将传入的http参数转换为java对象

Spring提供了2中方式将java表述形式转换为发送给客户端的表述形式

1. 内容协商，选择一个视图，它能够将模型渲染为呈现给客户端的表述形式
2. 消息转换器：通过消息转换器来将返回的对象转换为客户端的表述形式

**协商资源**

Spring的ContentNegotiatingViewResolver是一个特殊的视图解析器，它考虑到客户端需要的内容类型，ContentNegotiatingViewResolver的工作涉及两个步骤：

1. 确定请求媒体类型

首先查看url文件的扩展名，如果url结尾有扩展名，ContentNegotiatingViewResolver就会基于扩展名来确定所需的类型：.Json，那么所需的内容就是application/json，如果是.xml,那么客户端请求类型就是application/xml

如果不能根据文件扩展名得到媒体类型那么就是从accept头信息查找媒体类型

如果accept头信息中也没有媒体类型，那么就默认使用“/”作为内容类型，这就意味着客户端必须接收任何形式的表述（数据格式）

默认媒体类型的选择

ContentNegotiationManager:修改媒体类型的选择行为，

指定默认类型

通过请求参数指定内容类型

忽略请求的accept头部信息

将请求的扩展名映射为特定的媒体类型

ContentNegotiatingViewResolver的优势和局限

构件rest资源表述层时，控制器代码无需修改

ContentNegotiatingViewResolver作为视图解析器，只能决定资源改如何渲染到客户端，并没有涉及到客户端要发送什么样的表述给控制器

1. 找到合适请求媒体类型的视图解析器

**消息转换器**

消息转换提供了一种更为直接的方式，它能够将控制器产生的数据转换为服务于客户端的表述形式，当使用消息转换功能时，dispatchservlet不在将模型数据发送到视图，这里就没有模型和视图，只有控制器产生的数据，以及消息转换器转换数据之后产生的资源表述。

Spring自带各种消息转换器，这些都是自动注册的

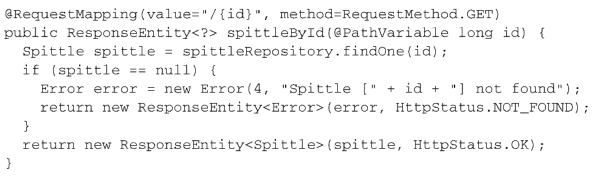
当处理方法返回java对象时，这个对象会放在模型中并在视图中渲染使用，但是如果使用了消息转换功能，那么需要告诉spring正常的模型/视图流程，并使用消息转换器，最简单的方式就是使用@ResponseBody注解

### rest的异常处理

让响应体中包含错误信息，spring提供了多种处理方式

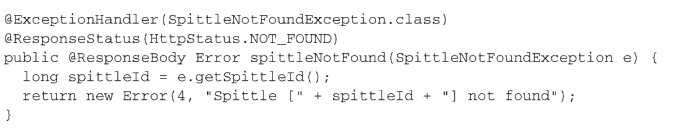
1. 使用@ResponseStatus注解来指定状态码
2. 返回ResponseEntry对象，该对象包含更多响应相关的信息
3. 使用异常处理器

**使用ResponseEntry**



**使用异常处理器**

定义异常处理器类，在类中指定异常的处理方法，使用@ResponseStatus指定异常编码，抛出异常的时候方法返回异常方法的执行结果



Spring boot

Spring bean的属性

1. bean的唯一标识符，在spring容器中不能存在两个相同的id
2. class全类名
3. name 别名，可以定义多个，中间使用逗号分开；id可以不用指定，在没有设置id的时候，如果配置了name,那么就将name的值作为beanname的值，id的值就是name,如果没有设置name，那么就用全类名作为beanname，id为全类名；当存在多个相同class的匿名beans时，id根据spring读取配置文件的顺序生成id,

com.bean.BeanName

com.bean.BeanName#0

com.bean.BeanName#1

name的值使用逗号隔开，那么就相当于存在多个标志符，比如bean name=”a,b,c” context.getBean(“a”); context.getBean(“b”); context.getBean(“c”);id和name同时存在是，两个都可以通过getBean的值获取对象

1. abstarct，设置bean是否为抽象类，默认为false，如果设为true，将不会实例化
2. autowire-candidate，默认为true，如果为false，那么当前bean不能自动装配到其他bean中
3. autowire：设置自动装配方法

byname,通过熟悉名称来自动装配，属性指的是id或者name

byType,根据属性类型来找到和配置文件中配置的class属性的全类名一直的bean来自动装配，rugged找到多个类型一致的bean，则抛出异常，一个也没有找到则不装配，也不抛出异常，

constructor：根据构造器中参数类型来自动装配，如果找打多个类一致的bean，则抛出异常，如果一个都没有找到，那么不装配，并抛出异常

autodetect是constructor与byType的组合，会先进行constructor，如果不成功，再进行byType。

1. depends-on，一个bean实例化的过程要依赖于另一个bean的初始化，
2. init-method:创建bean周调用的方法，该方法必须是一个无参方法
3. destroy-method，销毁bean之前执行的方法，方法参数最多只能有一个，并且参数类型只能为布尔类型（spring提供了close()方法来关闭容器，销毁bean）
4. factory-bean:创建bean的工厂类对象，class属性失效
5. factory-method：指定创建bean的工厂方法
6. lazy-init:是否延迟加载，如果为true，则在第一次使用bean是才会去创建bean实例（调用getBean方法），如果为false，那么会在初始化spring容器时，实例化bean对象
7. parent，指定bean的父类，class属性失效
8. primary 当一个bean出现多个候选者时，设置primary为true后，优先使用这个bean来自动装配
9. scope：设置bean的作用域，默认为singleton，还有property，request，session

Spring bean的子标签

1. meta，元数据声明的key并不会在bean中体现，只是一个额外的声明，当需要使用配置的内容时，通过beandefinition的getAttribute获取
2. lookup-method:子元素获取器注入，把一个方法声明为返回某种类型的bean
3. replace-method:可以在运行时调用新的方法替换现有的方法，还能动态更新方法的逻辑，

<replace-method name=”被替换的方法” replacer=替换的bean>

替换的bean必须实现MethodReplacer接口，重写reimplement方法

1. constructor-arg:构造参数，内部index属性执行顺序注入，没有配置就按从上到下的属性
2. property:注入成员变量的值

context标签

<context:annotation-config/>开启注解

<context:component-scan/>指定注解所在的包

使用注解之前要开启自动扫描功能，其中base-package为需要扫描的包(含子包)。

<context:component-scan base-package="cn.test"/>  
@Configuration把一个类作为一个IoC容器，它的某个方法头上如果注册了@Bean，就会作为这个Spring容器中的Bean。  
@Scope注解 作用域  
@Lazy(true) 表示延迟初始化  
@Service用于标注业务层组件、   
@Controller用于标注控制层组件（如struts中的action）  
@Repository用于标注数据访问组件，即DAO组件。  
@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。  
@Scope用于指定scope作用域的（用在类上）  
@PostConstruct用于指定初始化方法（用在方法上）  
@PreDestory用于指定销毁方法（用在方法上）  
@DependsOn：定义Bean初始化及销毁时的顺序  
@Primary：自动装配时当出现多个Bean候选者时，被注解为@Primary的Bean将作为首选者，否则将抛出异常  
@Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：  
@Autowired @Qualifier("personDaoBean") 存在多个实例配合使用  
@Resource默认按名称装配，当找不到与名称匹配的bean才会按类型装配。  
@PostConstruct 初始化注解  
@PreDestroy 摧毁注解 默认 单例  启动就加载  
@Async异步方法调用

@value注解：读取配置文件的属性，可以有@value(“${}”)通过多了符号引入参数和@value(“#{}”)通过井号引入参数

@value(“#{beanName.属性}”)，属性在bean中应该是可以访问的，最好是public

@value(“${}”),用于获取配置文件中的属性

Spring加载property配置文件，

<context:property-placeholder location=”…”/>

如果要加载多个properties文件，在location中使用逗号隔开

其他属性

1. ignore-resource-not-found,找不到配置文件，是否忽略，默认为false，即不忽略，找不到抛出异常，设置为true，找不到配置文件也不抛出异常
2. ignore-unresolvable，是否忽略解析不到的属性，默认false，解析不到就抛出异常，设置为true，解析不到就将属性作为值赋值给变量

@PostConstruct，@PreDestroy两个注解通过CommonAnnotationBeanPostProcessor注册，

@PostConstruct用于修饰一个非静态的void方法，修饰方法在构造函数之后运行，并且只会执行一次，在web服务中@PostContruct修饰的方法，在服务器加载servlet的时候执行，在init方法之前执行，修饰的方法不能有参数

@PreDestroy作用在容器中一个实例生命周期结束前要执行的方法上，作用的方法不能有参数，并且不能是静态的

Spring循环依赖

就是循环引用，也就是2个或者2个以上的bean相互持有对方，最终形成了闭环，比如A依赖于B,B依赖于C,c依赖于A,对象间的相互依赖形成一个死循环

循环依赖的形式

1. 构造函数参数循环依赖，此依赖无法解决，只能抛出异常
2. Setter属性循环依赖（只有在单例的情况下才会尝试循环依赖，原型模式下抛出异常）

创建bean实例后，将调用addSingletonFactory方法将当前单例工厂存放，在singletonFactories中(sigletonFactory缓存单例工厂结构是HashMap)

将对象缓存在singletonObject缓存单例实例化对象，设置依赖属性时，如果依赖属性实例还没有不存在，那么就先创建该实例，创建后因为要依赖原实例，那么就从缓存中取出依赖实例，注入到这个实例中，实例初始化完成后，就将该实例注入到最开始要创建的实例中

原型模式下不缓存创建的实例对象，因此无法提前暴露一个创建中的Bean

FactoryBean和BeanFactory

Spring提供的一个FactoryBean的工厂类接口，可以通过实现该接口定制实例化bean的逻辑，通常用来创建比较复杂的bean,一般的bean直接配置即可，但是如果一个bean的创建过程涉及到很多其他的bean和复杂的逻辑，xml配置比较困难，这是可以考虑用factoryBean,比如mybatis-spring中的sqlsessionfactorybean;

当配置文件中bean的class配置的类实现了FactoryBean接口时，通过getBean方法返回的不是FactoryBean的实例，而是getObject方法返回的对象，也就是返回getObject方法中创建的对象

FactoryBean是个bean，在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，是一个可以生产对象和装饰对象的工厂bean，由spring管理后，生产的对象是由getObject()方法决定的（从容器中获取到的对象不是“ FactoryBeanTest ” 对象）。

BeanFactory是个bean 工厂，是一个工厂类(接口)， 它负责生产和管理bean的一个工厂是ioc 容器最底层的接口，是个ioc容器，是spring用来管理和装配普通bean的ioc容器（这些bean成为普通bean）。

Spring扩展

自定义环境变量验证

自定义属性编辑器

继承java.beans.PropertyEditorSupport，重写setAsText方法，将方法的参数转换后，调用setValue方法来设置值

**public** **class** MyPropertiesEditor **extends** PropertyEditorSupport

{

@Override

**public** **void** setAsText(String text) **throws** IllegalArgumentException

{

SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd");

**try**

{

**this**.setValue(sdf.parse(text));

} **catch** (ParseException e)

{

System.*out*.println("不是日期字符串");

e.printStackTrace();

}

}

}

在配置文件中引入自定义属性，

定义org.springframework.beans.factory.config.CustomEditorConfigurer bean，配置customEditors属性，将自定义的编辑器引入设置到 customEditors中即可

<bean class=*"org.springframework.beans.factory.config.CustomEditorConfigurer"*>

<property name=*"customEditors"*>

<map>

<entry key=*"java.util.Date"* value=*"spring.editor.MyPropertiesEditor"*>

</entry>

</map>

</property>

</bean>

BeanPostProcessor：bean后置处理器，bean初始化前后进行拦截，

方法postProcessBeforeInitialization,初始化方法调用之前调用（一般是initmethod方法，或者@PostConstruct注解的方法，在构造函数之后执行）

方法postProcessAfterInitialization,初始化方法调用之后调用

BeanFactoryPostProcessor：用来处理修改beandefinition信息的后置处理器，在bean还没初始化，只是创建好BeanDefinition的时候执行接口的方法postProcessBeanFactory；在初始化bean之前最后一次提供修改的操作，一般用来修改BeanDefinition或bean的属性

BeanDefinitionRegistryPostProcessor：继承BeanFactoryPostProcessor，增加了postProcessBeanDefinitionRegistry方法，用来注册BeanDefinition，修改BeanDefinition

Spring事件监听

Spring java配置

基于java类配置的优势在于可以通过代码的方式控制bean的初始化的整体逻辑，在bean实例化bean的逻辑比较复杂时，比较适合用java类配置