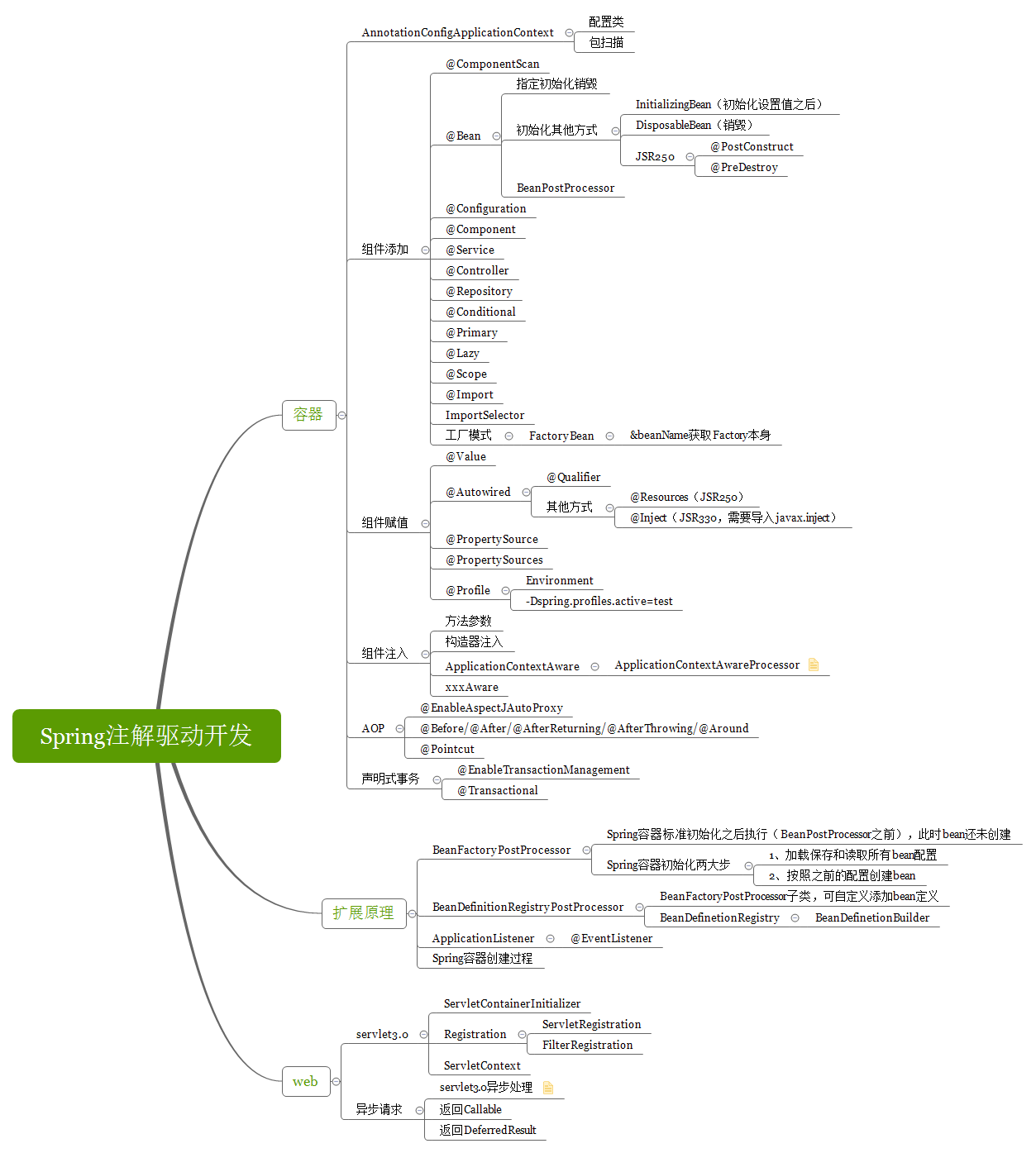
Spring注解开发

1. **基本介绍**

****

更多知识见同级目录下《Spring实战（第4版）》

1. **装配Bean**
2. **自动化装配（包扫描）**

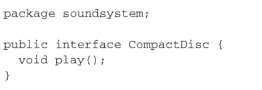
Spring 从两个角度来实现自动化装配：

* 组件扫描（ component scanning ）： Spring 会自动发现应用上下文中所创建的 bean 。
* 自动装配（ autowiring ）： Spring 自动满足 bean 之间的依赖。

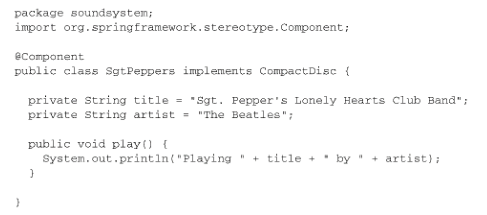
组件扫描和自动装配组合在一起就能发挥出强大的威力，它们能够将你的显式配置降低到最少。

自动装配会扫描带@Component、@Service、@Controller、@Repository的组件。

定义接口

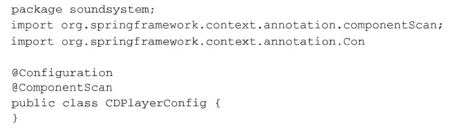


定义接口实现类



SgtPeppers 类上使用了 @Component 注解。这个简单的注解表明该类会作为组件类，并告知 Spring 要为这个类创建 bean 。没有必要显式配置 SgtPeppersbean ，因为这个类使用了 @Component 注解，所以 Spring 会为你把事情处理妥当。

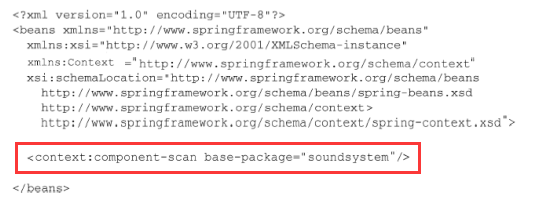
不过，组件扫描默认是不启用的。我们还需要显式配置一下 Spring ，从而命令它去寻找带有 @Component 注解的类，并为其创建 bean 。



现在我们只需观察一下 CDPlayerConfig 类并没有显式地声明任何 bean ，只不过它使用了 @ComponentScan 注解，这个注解能够在 Spring 中启用组件扫描。

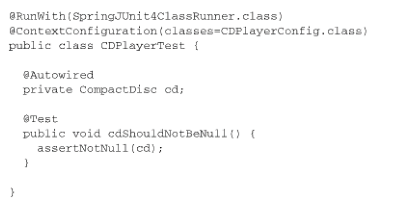
如果没有其他配置的话， @ComponentScan 默认会扫描与配置类相同的包。因为 CDPlayerConfig 类位于 soundsystem 包中，因此 Spring将会扫描这个包以及这个包下的所有子包，查找带有 @Component 注解的类。这样的话，就能发现 CompactDisc ，并且会在 Spring 中自动为其创建一个 bean 。

如果你更倾向于使用 XML 来启用组件扫描的话，那么可以使用 Spring context 命名空间的 <context:component-scan> 元素。



尽管我们可以通过 XML 的方案来启用组件扫描，但是在后面的讨论中，我更多的还是会使用基于 Java 的配置。

测试：



1. **@ComponentScans高级功能**

|  |
| --- |
| //配置类==配置文件  @Configuration //告诉Spring这是一个配置类  @ComponentScans(  value = {  @ComponentScan(value="com.atguigu",includeFilters = {  /\* @Filter(type=FilterType.ANNOTATION,classes={Controller.class}),  @Filter(type=FilterType.ASSIGNABLE\_TYPE,classes={BookService.class}),\*/  @Filter(type=FilterType.***CUSTOM***,classes={MyTypeFilter.**class**})  },useDefaultFilters = **false**)  }  )  //@ComponentScan value:指定要扫描的包  //excludeFilters = Filter[] ：指定扫描的时候按照什么规则排除那些组件  //includeFilters = Filter[] ：指定扫描的时候只需要包含哪些组件  //FilterType.ANNOTATION：按照注解  //FilterType.ASSIGNABLE\_TYPE：按照给定的类型；  //FilterType.ASPECTJ：使用ASPECTJ表达式  //FilterType.REGEX：使用正则指定  //FilterType.CUSTOM：使用自定义规则  **public** **class** MainConfig {  //给容器中注册一个Bean;类型为返回值的类型，id默认是用方法名作为id  @Bean("person")  **public** Person person01(){  **return** **new** Person("lisi", 20);  }  } |

MyTypeFilter类，自定义的过滤规则

|  |
| --- |
| **public** **class** MyTypeFilter **implements** TypeFilter {  /\*\*  \* metadataReader：读取到的当前正在扫描的类的信息  \* metadataReaderFactory:可以获取到其他任何类信息的  \*/  **public** **boolean** match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory)  **throws** IOException {  // **TODO** Auto-generated method stub  //获取当前类注解的信息  AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();  //获取当前正在扫描的类的类信息  ClassMetadata classMetadata = metadataReader.getClassMetadata();  //获取当前类资源（类的路径）  Resource resource = metadataReader.getResource();    String className = classMetadata.getClassName();  System.***out***.println("--->"+className);  **if**(className.contains("er")){  **return** **true**;  }  **return** **false**;  }  } |

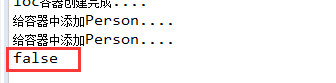
1. **@Scope控制bean作用范围**

|  |
| --- |
| //默认是单实例的  /\*\*  \* ConfigurableBeanFactory#SCOPE\_PROTOTYPE  \* **@see** ConfigurableBeanFactory#SCOPE\_SINGLETON  \* **@see** org.springframework.web.context.WebApplicationContext#SCOPE\_REQUEST request  \* **@see** org.springframework.web.context.WebApplicationContext#SCOPE\_SESSION sesssion  \* **@return**\  \* **@Scope**:调整作用域  \* prototype：多实例的：ioc容器启动并不会去调用方法创建对象放在容器中。  \* 每次获取的时候才会调用方法创建对象；  \* singleton：单实例的（默认值）：ioc容器启动会调用方法创建对象放到ioc容器中。  \* 以后每次获取就是直接从容器（map.get()）中拿，  \* request：同一次请求创建一个实例  \* session：同一个session创建一个实例  \*  \*/  @Scope(scopeName="prototype")  @Lazy  @Bean("person")  **public** Person person(){  System.***out***.println("给容器中添加Person....");  **return** **new** Person("张三", 25);  } |

测试：

|  |
| --- |
| @Test  **public** **void** test02(){  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConfig2.**class**);  System.***out***.println("ioc容器创建完成....");  Object bean = applicationContext.getBean("person");  Object bean2 = applicationContext.getBean("person");  System.***out***.println(bean == bean2);  } |

结果：



1. **@Lazy懒加载**

懒加载：主要针对单实例bean

\* 单实例bean：默认在容器启动的时候创建对象；

\* 懒加载：容器启动不创建对象。第一次使用(获取)Bean创建对象，并初始化；

|  |
| --- |
| @Lazy  @Bean("person")  **public** Person person(){  System.***out***.println("给容器中添加Person....");  **return** **new** Person("张三", 25);  } |

1. **@Conditional**

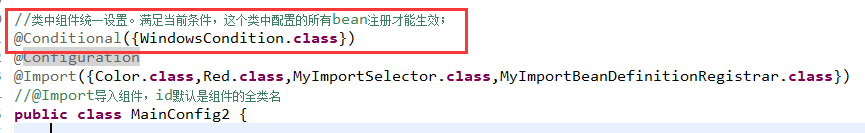
按条件加载bean，spring4.0新添的功能，在springboot中大量使用了该注解。

|  |
| --- |
| /\*\*  \* **@Conditional**({Condition}) ： 按照一定的条件进行判断，满足条件给容器中注册bean  \*  \* 如果系统是windows，给容器中注册("bill")  \* 如果是linux系统，给容器中注册("linus")  \*/    @Bean("bill")  **public** Person person01(){  **return** **new** Person("Bill Gates",62);  }    @Conditional(LinuxCondition.**class**)  @Bean("linus")  **public** Person person02(){  **return** **new** Person("linus", 48);  } |

自定义的条件规则类

|  |
| --- |
| //判断是否linux系统  **public** **class** LinuxCondition **implements** Condition {  /\*\*  \* ConditionContext：判断条件能使用的上下文（环境）  \* AnnotatedTypeMetadata：注释信息  \*/  **public** **boolean** matches(ConditionContext context, AnnotatedTypeMetadata metadata) {  // TODO是否linux系统  //1、能获取到ioc使用的beanfactory  ConfigurableListableBeanFactory beanFactory = context.getBeanFactory();  //2、获取类加载器  ClassLoader classLoader = context.getClassLoader();  //3、获取当前环境信息  Environment environment = context.getEnvironment();  //4、获取到bean定义的注册类  BeanDefinitionRegistry registry = context.getRegistry();    String property = environment.getProperty("os.name");    //可以判断容器中的bean注册情况，也可以给容器中注册bean  **boolean** definition = registry.containsBeanDefinition("person");  **if**(property.contains("linux")){  **return** **true**;  }  **return** **false**;  }  } |

当然我们也可以放到类上统一设置



1. **@Import系列注解**

**@Import**

@Import导入组件，id默认是组件的全类名。

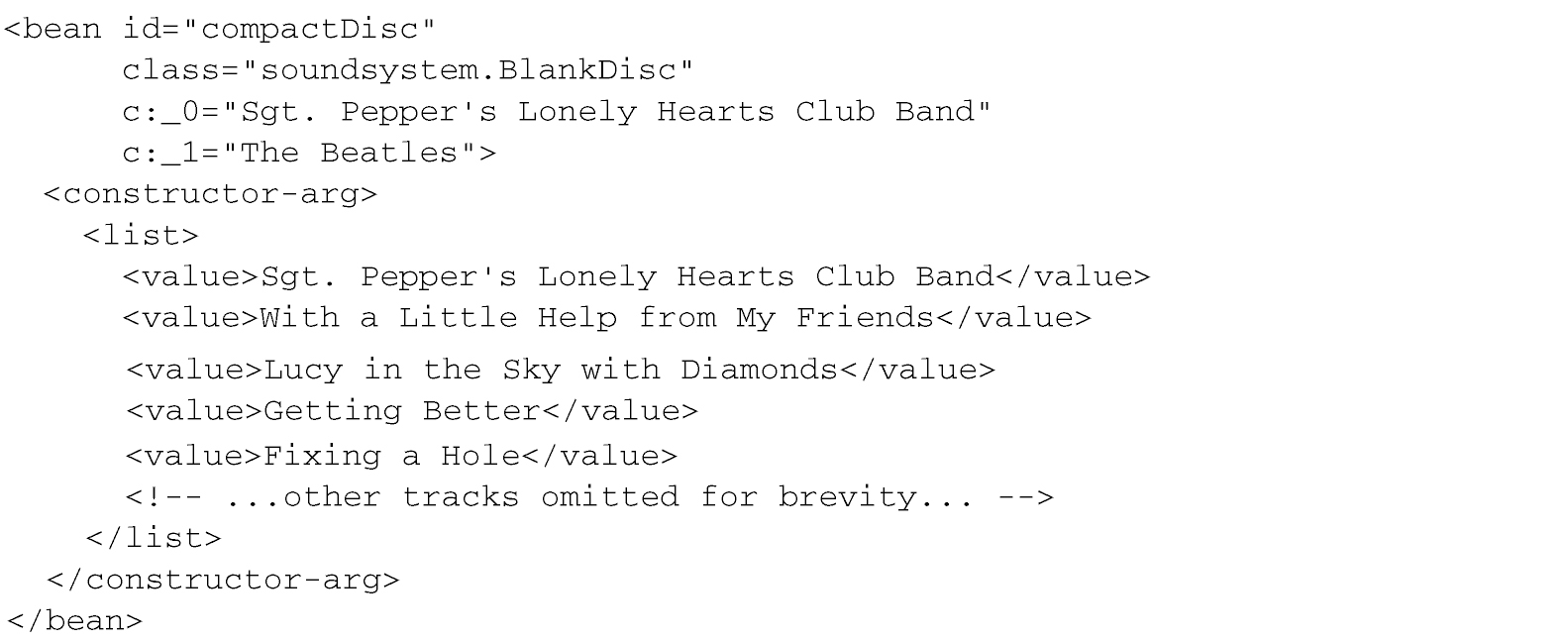
|  |
| --- |
| **public** **class** Color {  **private** Car car;  **public** Car getCar() {  **return** car;  }  **public** **void** setCar(Car car) {  **this**.car = car;  }  @Override  **public** String toString() {  **return** "Color [car=" + car + "]";  }  } |



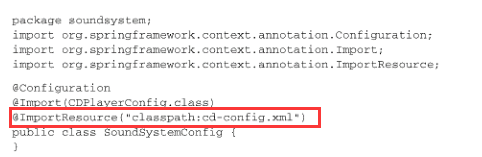
此时发现Color类就装配到bean容器中了。

**@ImportResource**

装配xml配置中的bean



@ImportResource 注解，假设 BlankDisc 定义在名为 cd-config.xml 的文件中，该文件位于根类路径下，那么可以修改 SoundSystemConfig ，让它使用 @ImportResource 注解。



**ImportSelector**

ImportSelector:返回需要导入的组件的全类名数组

自定义的选择器

|  |
| --- |
| //自定义逻辑返回需要导入的组件  **public** **class** MyImportSelector **implements** ImportSelector {  //返回值，就是到导入到容器中的组件全类名  //AnnotationMetadata:当前标注@Import注解的类的所有注解信息  **public** String[] selectImports(AnnotationMetadata importingClassMetadata) {  // **TODO** Auto-generated method stub  //importingClassMetadata  //方法不要返回null值  **return** **new** String[]{"com.atguigu.bean.Blue","com.atguigu.bean.Yellow"};  }  } |

使用



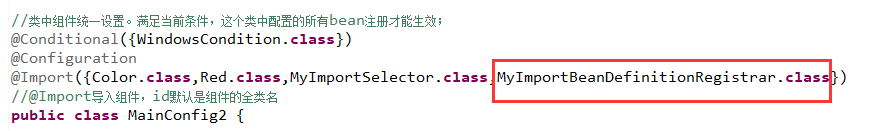
**ImportBeanDefinitionRegistrar**

ImportBeanDefinitionRegistrar:手动注册bean到容器中

新建MyImportBeanDefinitionRegistrar类

|  |
| --- |
| **public** **class** MyImportBeanDefinitionRegistrar **implements** ImportBeanDefinitionRegistrar {  /\*\*  \* AnnotationMetadata：当前类的注解信息  \* BeanDefinitionRegistry:BeanDefinition注册类；  \* 把所有需要添加到容器中的bean；调用  \* BeanDefinitionRegistry.registerBeanDefinition手工注册进来  \*/  **public** **void** registerBeanDefinitions(AnnotationMetadata importingClassMetadata, BeanDefinitionRegistry registry) {  **boolean** definition = registry.containsBeanDefinition("com.atguigu.bean.Red");  **boolean** definition2 = registry.containsBeanDefinition("com.atguigu.bean.Blue");  **if**(definition && definition2){  //指定Bean定义信息；（Bean的类型，Bean。。。）  RootBeanDefinition beanDefinition = **new** RootBeanDefinition(RainBow.**class**);  //注册一个Bean，指定bean名  registry.registerBeanDefinition("rainBow", beanDefinition);  }  }  } |

使用



启动测试会发现rainBow就注册到了bean容器中

1. **FactoryBean**

使用Spring提供的 FactoryBean（工厂Bean）;

\* 1）、默认获取到的是工厂bean调用getObject创建的对象

\* 2）、要获取工厂Bean本身，我们需要给id前面加一个&

创建一个Spring定义的FactoryBean

|  |
| --- |
| **public** **class** ColorFactoryBean **implements** FactoryBean<Color> {  //返回一个Color对象，这个对象会添加到容器中  @Override  **public** Color getObject() **throws** Exception {  System.***out***.println("ColorFactoryBean...getObject...");  **return** **new** Color();  }  @Override  **public** Class<?> getObjectType() {  **return** Color.**class**;  }  //是单例？  //true：这个bean是单实例，在容器中保存一份  //false：多实例，每次获取都会创建一个新的bean；  @Override  **public** **boolean** isSingleton() {  **return** **false**;  }  } |

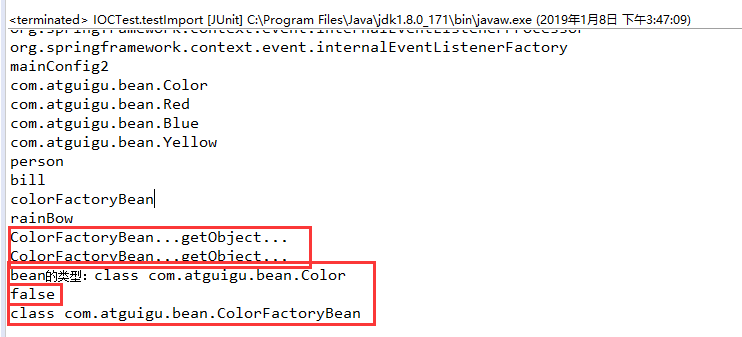
在配置类MainConfig2中添加代码

|  |
| --- |
| @Bean  **public** ColorFactoryBean colorFactoryBean(){  **return** **new** ColorFactoryBean();  } |

测试

|  |
| --- |
| @Test  **public** **void** testImport(){  String[] definitionNames = applicationContext.getBeanDefinitionNames();  **for** (String name : definitionNames) {  System.***out***.println(name);  }  //工厂Bean获取的是调用getObject创建的对象  Object bean2 = applicationContext.getBean("colorFactoryBean");  Object bean3 = applicationContext.getBean("colorFactoryBean");  System.***out***.println("bean的类型："+bean2.getClass());  System.***out***.println(bean2 == bean3);  Object bean4 = applicationContext.getBean("&colorFactoryBean");  System.***out***.println(bean4.getClass());  } |

结果



1. **Bean的生命周期**
2. **概要**

bean的生命周期：

\* bean创建---初始化----销毁的过程

\* 容器管理bean的生命周期；

我们可以自定义初始化和销毁方法；容器在bean进行到当前生命周期的时候来调用我们自定义的初始化和销毁方法。



**Spring执行流程**

\* 遍历得到容器中所有的BeanPostProcessor；挨个执行beforeInitialization，

\* 一但返回null，跳出for循环，不会执行后面的BeanPostProcessor.postProcessorsBeforeInitialization

\*

\* BeanPostProcessor原理

\* populateBean(beanName, mbd, instanceWrapper);给bean进行属性赋值

\* initializeBean

\* {

\* applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

\* invokeInitMethods(beanName, wrappedBean, mbd);执行自定义初始化

\* applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);

\*}

如何自定义初始化和销毁方法？

1）、在xml中配置bean的时候可以设置属性来定义，init-method和destroy-method

2）、注解方式

3）、通过让Bean实现InitializingBean（定义初始化逻辑），DisposableBean（定义销毁逻辑）;

4）、可以使用JSR250规范中的；

\* @PostConstruct：在bean创建完成并且属性赋值完成；来执行初始化方法。直接标注到方法上就行

\* @PreDestroy：在容器销毁bean之前通知我们进行清理工作

5）、BeanPostProcessor【interface】：bean的后置处理器，在bean初始化前后进行一些处理工作；

\* postProcessBeforeInitialization:在初始化之前工作

\* postProcessAfterInitialization:在初始化之后工作

1. **注解方式（初始化和销毁）**

构造方法（对象创建）

\* 单实例：在容器启动的时候创建对象

\* 多实例：在每次获取的时候创建对象

初始化：

\* 对象创建完成，并赋值好，调用初始化方法。。。

销毁：

\* 单实例：容器关闭的时候调用销毁方法

\* 多实例：容器不会管理这个bean；容器不会调用销毁方法；

Car类

|  |
| --- |
| **public** **class** Car {  **public** Car(){  System.***out***.println("car constructor...");  }  **public** **void** init(){  System.***out***.println("car ... init...");  }  **public** **void** detory(){  System.***out***.println("car ... detory...");  }  } |

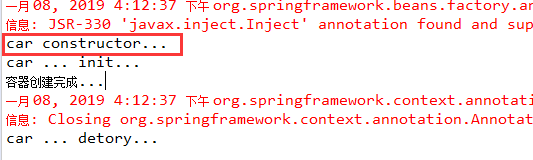
配置类

|  |
| --- |
| @ComponentScan("com.atguigu.bean.Car")  @Configuration  **public** **class** MainConfigOfLifeCycle {  @Scope("prototype")  @Bean(initMethod="init",destroyMethod="detory")  **public** Car car(){  **return** **new** Car();  }  } |

测试

|  |
| --- |
| @Test  **public** **void** test01(){  //1、创建ioc容器  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConfigOfLifeCycle.**class**);  System.***out***.println("容器创建完成...");    applicationContext.getBean("car");  //关闭容器  applicationContext.close();  } |

单实例结果



1. **BeanPostProcessor后置处理器**

新建后置处理器MyBeanPostProcessor放到bean文件夹下

|  |
| --- |
| /\*\*  \* 后置处理器：初始化前后进行处理工作  \* 将后置处理器加入到容器中  \*/  @Component  **public** **class** MyBeanPostProcessor **implements** BeanPostProcessor {  @Override  **public** Object postProcessBeforeInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println("postProcessBeforeInitialization..."+beanName+"=>"+bean);  **return** bean;  }  @Override  **public** Object postProcessAfterInitialization(Object bean, String beanName) **throws** BeansException {  System.***out***.println("postProcessAfterInitialization..."+beanName+"=>"+bean);  **return** bean;  }  } |

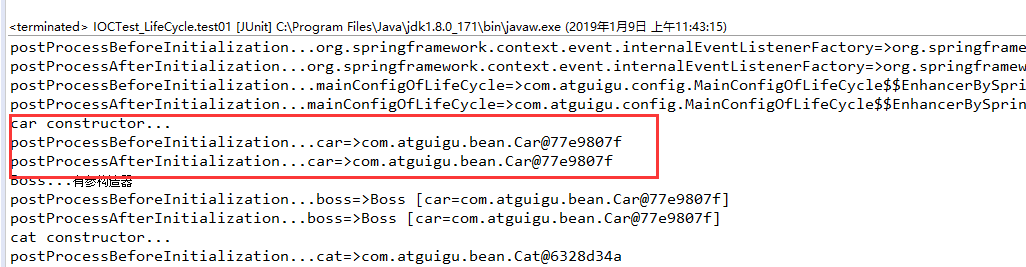
配置类

|  |
| --- |
| @ComponentScan("com.atguigu.bean")  @Configuration  **public** **class** MainConfigOfLifeCycle {  } |

测试类

|  |
| --- |
| @Test  **public** **void** test01(){  //1、创建ioc容器  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConfigOfLifeCycle.**class**);  System.***out***.println("容器创建完成...");  //关闭容器  applicationContext.close();  } |

结果



**Spring底层对 BeanPostProcessor 的使用；**

bean赋值，注入其他组件，@Autowired，生命周期注解功能，@Async,xxx 等等都会有相关的BeanPostProcessor对其进行处理

1. **获取ApplicationContext**

新建Dog类

|  |
| --- |
| @Component  **public** **class** Dog **implements** ApplicationContextAware {  **private** ApplicationContext applicationContext;    **public** Dog(){  System.***out***.println("dog constructor...");  }  **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {  **this**.applicationContext = applicationContext;  }  } |

启动程序，此时就能在Dog类中获取到ApplicationContext容器

1. **属性赋值**

新建Person类，省略set和get方法

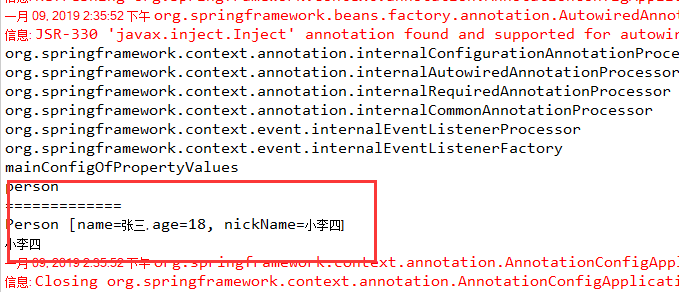
|  |
| --- |
| **public** **class** Person {  //使用@Value赋值的3种方式  //1、基本数值  //2、可以写SpEL； #{}  //3、可以写${}；取出配置文件【properties】中的值（在运行环境变量里面的值）  @Value("张三")  **private** String name;  @Value("#{20-2}")  **private** Integer age;  @Value("${person.nickName}")  **private** String nickName;  } |

配置类，引入Person的配置文件

|  |
| --- |
| //使用@PropertySource读取外部配置文件中的k/v保存到运行的环境变量中;加载完外部的配置文件以后使用${}取出配置文件的值  @PropertySource(value={"classpath:/person.properties"})  @Configuration  **public** **class** MainConfigOfPropertyValues {  @Bean  **public** Person person(){  **return** **new** Person();  }  } |

测试类

|  |
| --- |
| AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConfigOfPropertyValues.**class**);  @Test  **public** **void** test01(){  printBeans(applicationContext);  System.***out***.println("=============");    Person person = (Person) applicationContext.getBean("person");  System.***out***.println(person);    ConfigurableEnvironment environment = applicationContext.getEnvironment();  String property = environment.getProperty("person.nickName");  System.***out***.println(property);  applicationContext.close();  } |



1. **自动装配**
2. **@Autowired**

\* **1**、@Autowired：自动注入：

\* 1）、默认优先按照类型去容器中找对应的组件:applicationContext.getBean(BookDao.class);找到就赋值

\* 2）、如果找到多个相同类型的组件，再将属性的名称作为组件的id去容器中查找

\* applicationContext.getBean("bookDao")

\* 3）、@Qualifier("bookDao")：使用@Qualifier指定需要装配的组件的id，而不是使用属性名

\* 4）、自动装配默认一定要将属性赋值好，没有就会报错；

\* 可以使用@Autowired(required=false);

\* 5）、@Primary：让Spring进行自动装配的时候，默认使用首选的bean；

\* 也可以继续使用@Qualifier指定需要装配的bean的名字

\* BookService{

\* @Autowired

\* BookDao bookDao;

\* }

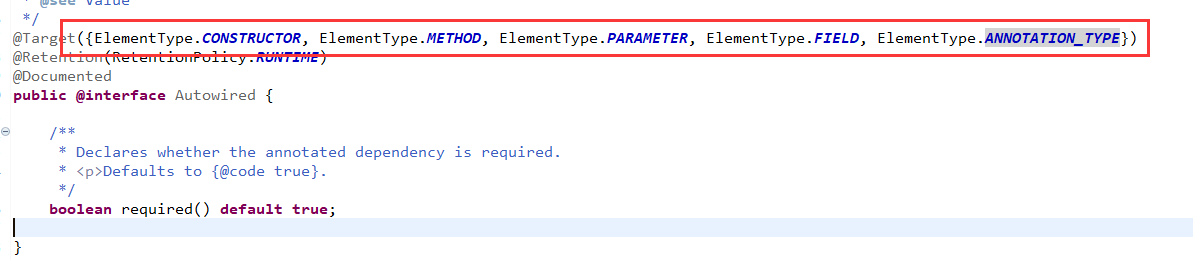
\* **2**、 @Autowired:构造器，参数，方法，属性；都是从容器中获取参数组件的值

\* 1）、[标注在方法位置]：@Bean+方法参数；参数从容器中获取;默认不写@Autowired效果是一样的；都能自动装配

\* 2）、[标在构造器上]：如果组件只有一个有参构造器，这个有参构造器的@Autowired可以省略，参数位置的组件还是可以自动从容器中获取

\* 3）、放在参数位置：

源码：



@Autowired是根据反射进行装配属性的。

|  |
| --- |
| /\*\*  \* 处理在字段上的注解  \* @param bean 处理的bean  \*/  public void fieldAnnotation(Object bean){  try {  //获取其全部的字段描述  Field[] fields = bean.getClass().getFields();  for(Field f : fields){  if(f!=null && f.isAnnotationPresent(ZxfResource.class)){  ZxfResource resource = f.getAnnotation(ZxfResource.class);  String name ="";  Object value = null;  if(resource.name()!=null&&!"".equals(resource.name())){  name = resource.name();  value = sigletions.get(name);  }else{  for(String key : sigletions.keySet()){  //判断当前属性所属的类型是否在配置文件中存在  if(f.getType().isAssignableFrom(sigletions.get(key).getClass())){  //获取类型匹配的实例对象  value = sigletions.get(key);  break;  }  }  }  //允许访问private字段  f.setAccessible(true);  //把引用对象注入属性  f.set(bean, value);  }  }  } catch (Exception e) {  log.info("字段注解解析异常..........");  }  } |

新建类BookDao，此处会注入一个bookDao的bean

|  |
| --- |
| //名字默认是类名首字母小写  @Repository  **public** **class** BookDao {  **private** String lable = "1";  **public** String getLable() {  **return** lable;  }  **public** **void** setLable(String lable) {  **this**.lable = lable;  }  @Override  **public** String toString() {  **return** "BookDao [lable=" + lable + "]";  }  } |

配置类，此处也会在bean容器中注入一个bookDao2的bean

|  |
| --- |
| @Configuration  @ComponentScan({"com.atguigu.service","com.atguigu.dao",  "com.atguigu.controller","com.atguigu.bean"})  **public** **class** MainConifgOfAutowired {  @Bean("bookDao2")  **public** BookDao bookDao(){  BookDao bookDao = **new** BookDao();  bookDao.setLable("2");  **return** bookDao;  }  } |
|  |
|  |

使用类BookService

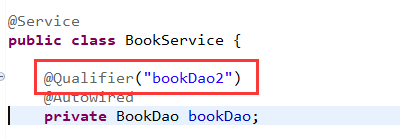
|  |
| --- |
| @Service  **public** **class** BookService {  @Autowired  **private** BookDao bookDao;    **public** **void** print(){  System.***out***.println(bookDao);  }  @Override  **public** String toString() {  **return** "BookService [bookDao=" + bookDao + "]";  }  } |

测试类

|  |
| --- |
| @Test  **public** **void** test01(){  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConifgOfAutowired.**class**);    BookService bookService = applicationContext.getBean(BookService.**class**);  System.***out***.println(bookService);  } |

此处用的是bookDao的bean

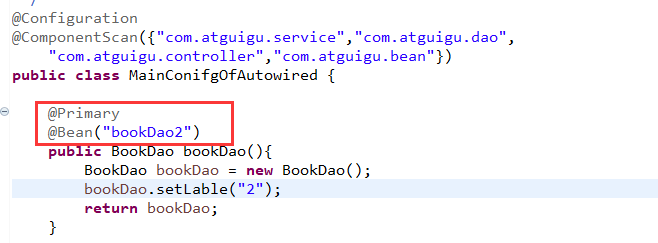
如果我们想指定使用某个bean，可以使用@Qualifier注解，此注解要加到@Autowired之上



通过如上设置，启动测试，则就能使用bookDao2的bean了

当然我们也可以通过

@Primary可以设置优先的bean



如果我们即设置了@Primary也设置了@Qualifier，@Qualifier的优先级更高

如果我们没有注入bean，但是又不想程序报错，此时可以设置

@Autowired(required=false)

1. **@Resource**

Spring还支持使用@Resource(JSR250), java规范的注解

@Resource:

\* 可以和@Autowired一样实现自动装配功能；默认是按照组件名称进行装配的；

\* 没有能支持@Primary功能没有支持@Autowired（reqiured=false）;

@Resource可以指定name属性来装配指定的bean，类似于@Qualifier注解功能

|  |
| --- |
| @Resource(name="bookDao2")  **private** BookDao bookDao; |

1. **@Inject**

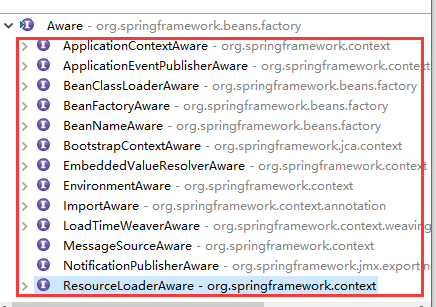
@Inject需要导入javax.inject的包，和Autowired的功能一样。支持@Primary功能，但是没有required=false的功能；

导包

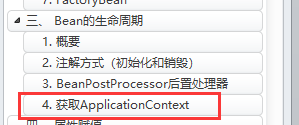
|  |
| --- |
| <dependency>  <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>  <version>1</version>  </dependency> |

1. **自定义组件使用Spring容器底层组件**

自定义组件想要使用Spring容器底层的一些组件（ApplicationContext，BeanFactory，xxx）；只需要自定义组件实现xxxAware；在创建对象的时候，会调用接口规定的方法注入相关组件；Aware，把Spring底层一些组件注入到自定义的Bean中；每一个xxxAware都有相应的xxxProcessor来进行处理；具体如下这么多Aware



示例参见



1. **Profile**

Spring为我们提供的可以根据当前环境，动态的激活和切换一系列组件的功能；

比如我们项目有开发环境、测试环境、生产环境等的配置信息，这些信息是不一样的，我们在开发环境通过profile激活开发环境配置，在正式环境激活正式环境配置信息。

@Profile：指定组件在哪个环境的情况下才能被注册到容器中，不指定，任何环境下都能注册这个组件

\* 1）、加了环境标识的bean，只有这个环境被激活的时候才能注册到容器中。默认是default环境

\* 2）、写在配置类上，只有是指定的环境的时候，整个配置类里面的所有配置才能开始生效

\* 3）、没有标注环境标识的bean在，任何环境下都是加载的；

创建配置类

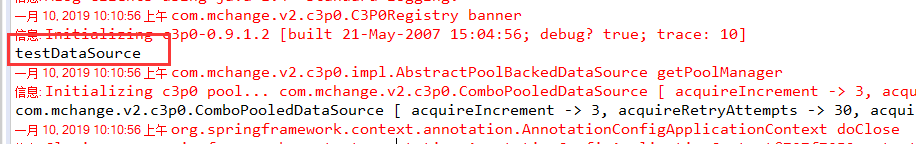
|  |
| --- |
| @PropertySource("classpath:/dbconfig.properties")  @Configuration  **public** **class** MainConfigOfProfile **implements** EmbeddedValueResolverAware{    @Value("${db.user}")  **private** String user;  **private** StringValueResolver valueResolver;  **private** String driverClass;  @Profile("test")  @Bean("testDataSource")  **public** DataSource dataSourceTest(@Value("${db.password}")String pwd) **throws** Exception{  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(user);  dataSource.setPassword(pwd);  dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test");  dataSource.setDriverClass(driverClass);  **return** dataSource;  }    @Profile("dev")  @Bean("devDataSource")  **public** DataSource dataSourceDev(@Value("${db.password}")String pwd) **throws** Exception{  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser(user);  dataSource.setPassword(pwd);  dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/ssm\_crud");  dataSource.setDriverClass(driverClass);  **return** dataSource;  }  **public** **void** setEmbeddedValueResolver(StringValueResolver resolver) {  **this**.valueResolver = resolver;  driverClass = valueResolver.resolveStringValue("${db.driverClass}");  }  } |

此处省略dbconfig.properties配置内容

测试类：

|  |
| --- |
| @Test  **public** **void** test01(){  AnnotationConfigApplicationContext applicationContext =  **new** AnnotationConfigApplicationContext();  //1、创建一个applicationContext  //2、设置需要激活的环境  applicationContext.getEnvironment().setActiveProfiles("dev");  //3、注册主配置类  applicationContext.register(MainConfigOfProfile.**class**);  //4、启动刷新容器  applicationContext.refresh();    String[] namesForType = applicationContext.getBeanNamesForType(DataSource.**class**);  **for** (String string : namesForType) {  System.***out***.println(string);  }    DataSource bean = applicationContext.getBean(DataSource.**class**);  System.***out***.println(bean);  applicationContext.close();  } |

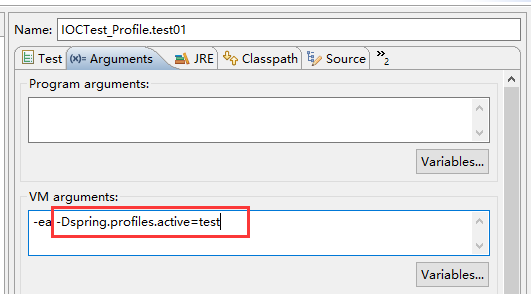
结果：



如上是代码激活，我们也可以使用命令激活

使用命令行动态参数: 在虚拟机参数位置加载 -Dspring.profiles.active=test

Eclipse激活：



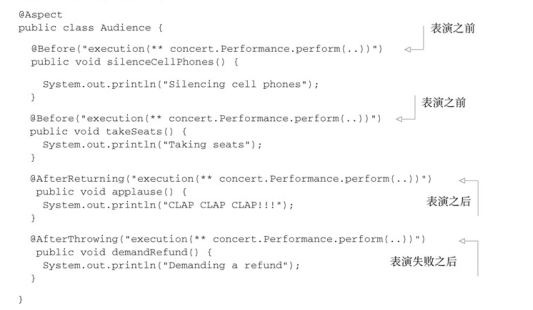
1. **AOP**
2. **概要**

AOP：【动态代理】

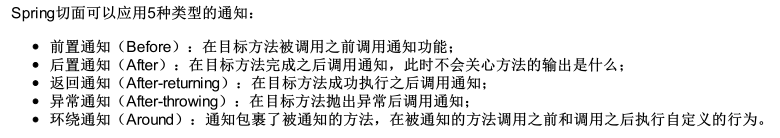
\* 指在程序运行期间动态的将某段代码切入到指定方法指定位置进行运行的编程方式；

**切面**：如果要重用通用功能的话，最常见的面向对象技术是继承（ inheritance ）或委托（ delegation ）。但是，如果在整个应用中都使用相同的基类，继承往往会导致一个脆弱的对象体系；而使用委托可能需要对委托对象进行复杂的调用。切面提供了取代继承和委托的另一种可选方案，而且在很多场景下更清晰简洁。在使用面向切面编程时，我们仍然在一个地方定义通用功能，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，而无需修改受影响的类。横切关注点可以被模块化为特殊的类，这些类被称为切面（ aspect ）。

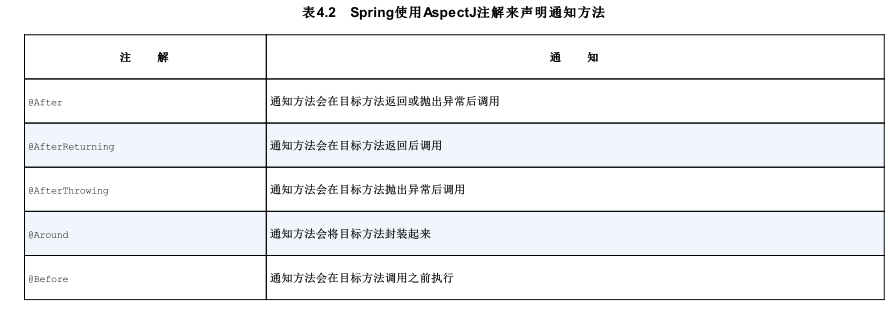
定义切面:(指如下整个类)



**通知（ Advice ）**：切面的工作被称为通知，比如事务控制，添加日志等等



定义通知：



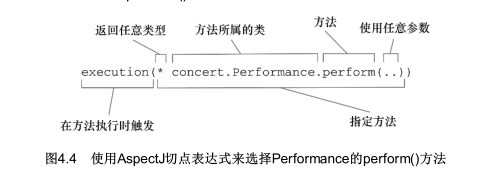
**连接点（ Join point ）**：连接点是在应用执行过程中能够插入切面的一个点。这个点可以是调用方法时、抛出异常时、甚至修改一个字段时。切面代码可以利用这些点插入到应用的正常流程之中，并添加新的行为。

连接点可以说是切点的全集。切点是连接点的子集。也可以理解为，连接点是我们没有定义那个select开头规则时，满足条件的全部的方法。

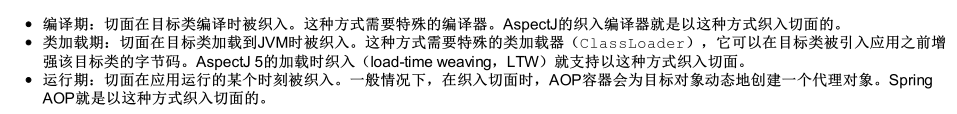
连接点是一个虚拟的概念，可以理解为所有满足切点扫描条件的所有的时机。

**切点（ Poincut ）**：一个切面并不需要通知应用的所有连接点。切点有助于缩小切面所通知的连接点的范围。切点的定义会匹配通知所要织入的一个或多个连接点。我们通常使用明确的类和方法名称，或是利用正则表达式定义所匹配的类和方法名称来指定这些切点。有些 AOP 框架允许我们创建动态的切点，可以根据运行时的决策（比如方法的参数值）来决定是否应用通知。

定义切点：



**织入（ Weaving ）：**织入是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目标对象中。在目标对象的生命周期里有多个点可以进行织入：



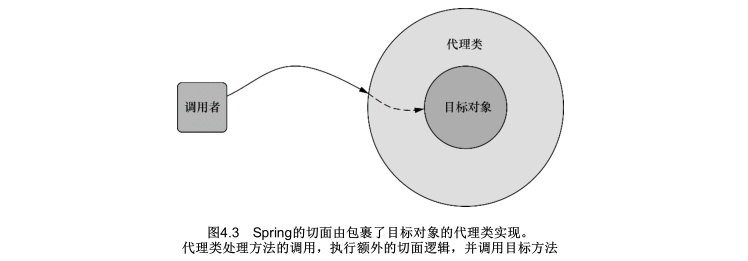
1. **Spring对AOP的支持**

Spring 提供了 4 种类型的 AOP 支持：

* 基于代理的经典 Spring AOP ；
* 纯 POJO 切面；
* @AspectJ 注解驱动的切面；
* 注入式 AspectJ 切面（适用于 Spring 各版本）。

前三种都是 Spring AOP 实现的变体， Spring AOP 构建在动态代理基础之上，因此， Spring 对 AOP 的支持局限于方法拦截。

通过在代理类中包裹切面， Spring 在运行期把切面织入到 Spring 管理的 bean 中。如图 4.3 所示，代理类封装了目标类，并拦截被通知方法的调用，再把调用转发给真正的目标 bean 。当代理拦截到方法调用时，在调用目标 bean 方法之前，会执行切面逻辑。



直到应用需要被代理的 bean 时， Spring 才创建代理对象。如果使用的是 ApplicationContext 的话，在 ApplicationContext 从 BeanFactory 中加载所有 bean 的时候， Spring 才会创建被代理的对象。因为 Spring 运行时才创建代理对象，所以我们不需要特殊的编译器来织入 Spring AOP 的切面。

Spring 基于动态代理，所以 Spring 只支持方法连接点。

1. **示例**

实现步骤：



基本类MathCalculator，也就是被增强类

|  |
| --- |
| **public** **class** MathCalculator {  **public** **int** div(**int** i,**int** j){  System.***out***.println("MathCalculator...div...");  **return** i/j;  }  } |

切面类LogAspects

|  |
| --- |
| /\*\*  \* 切面类  \* **@Aspect**： 告诉Spring当前类是一个切面类  \*/  @Aspect  **public** **class** LogAspects {  //抽取公共的切入点表达式  //1、本类引用  //2、其他的切面引用  // @Pointcut("execution(public int com.atguigu.aop.MathCalculator.div(int,int))")  @Pointcut("execution(public int com.atguigu.aop.MathCalculator.\*(..))")  **public** **void** pointCut(){};    //@Before在目标方法之前切入；切入点表达式（指定在哪个方法切入）  @Before("pointCut()")  **public** **void** logStart(JoinPoint joinPoint){  Object[] args = joinPoint.getArgs();  System.***out***.println(""+joinPoint.getSignature().getName()+"运行。。。@Before:参数列表是：{"+Arrays.*asList*(args)+"}");  }    @After("com.atguigu.aop.LogAspects.pointCut()")  **public** **void** logEnd(JoinPoint joinPoint){  System.***out***.println(""+joinPoint.getSignature().getName()+"结束。。。@After");  }    //JoinPoint一定要出现在参数表的第一位  @AfterReturning(value="pointCut()",returning="result")  **public** **void** logReturn(JoinPoint joinPoint,Object result){  System.***out***.println(""+joinPoint.getSignature().getName()+"正常返回。。。@AfterReturning:运行结果：{"+result+"}");  }    @AfterThrowing(value="pointCut()",throwing="exception")  **public** **void** logException(JoinPoint joinPoint,Exception exception){  System.***out***.println(""+joinPoint.getSignature().getName()+"异常。。。异常信息：{"+exception+"}");  }  } |

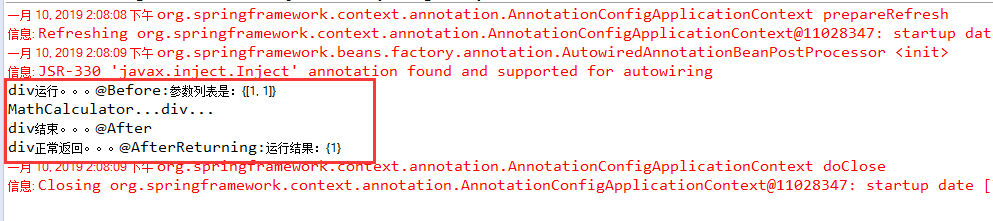
配置类

|  |
| --- |
| @EnableAspectJAutoProxy //开启基于注解的aop模式  @Configuration  **public** **class** MainConfigOfAOP {    //业务逻辑类加入容器中  @Bean  **public** MathCalculator calculator(){  **return** **new** MathCalculator();  }  //切面类加入到容器中  @Bean  **public** LogAspects logAspects(){  **return** **new** LogAspects();  }  } |

测试类

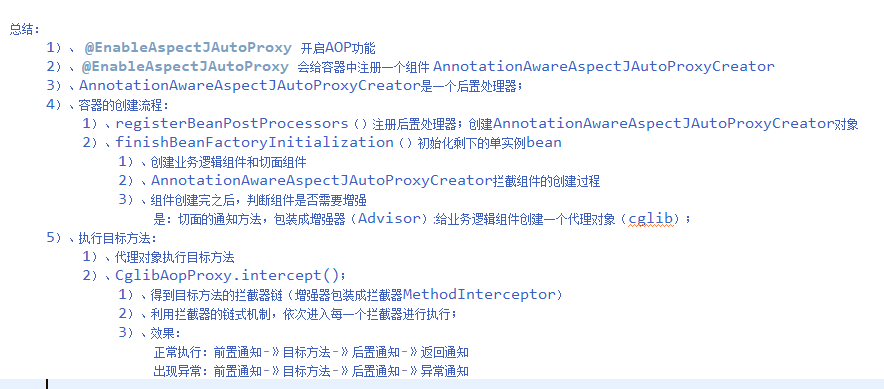
|  |
| --- |
| @Test  **public** **void** test01(){  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(MainConfigOfAOP.**class**);  //1、不要自己创建对象  // MathCalculator mathCalculator = new MathCalculator();  // mathCalculator.div(1, 1); mathCalculator对象其实是个代理对象，不是MathCalculator的原对象  MathCalculator mathCalculator = applicationContext.getBean(MathCalculator.**class**);  mathCalculator.div(1, 1);  applicationContext.close();  } |

结果：



1. **Aop原理**

Aop执行步骤流程

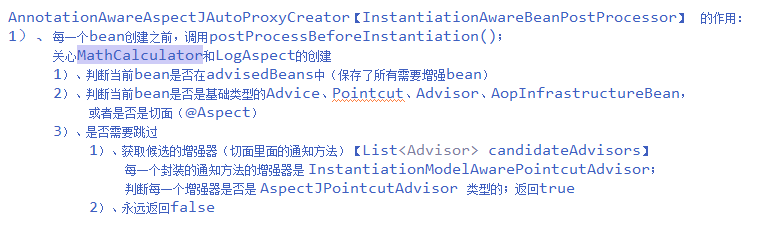


原理大致如下：











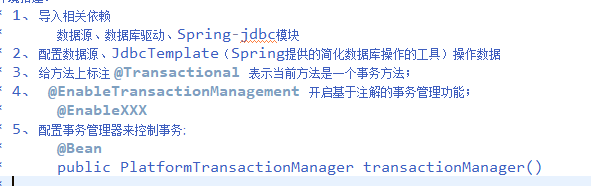


1. **声明式事务**
2. **原理**

|  |
| --- |
| \* 原理：  \* 1）、@EnableTransactionManagement  \* 利用TransactionManagementConfigurationSelector给容器中会导入组件  \* 导入两个组件  \* AutoProxyRegistrar  \* ProxyTransactionManagementConfiguration  \* 2）、AutoProxyRegistrar：  \* 给容器中注册一个 InfrastructureAdvisorAutoProxyCreator 组件；  \* InfrastructureAdvisorAutoProxyCreator：？  \* 利用后置处理器机制在对象创建以后，包装对象，返回一个代理对象（增强器），代理对象执行方法利用拦截器链进行调用；  \*  \* 3）、ProxyTransactionManagementConfiguration 做了什么？  \* 1、给容器中注册事务增强器；  \* 1）、事务增强器要用事务注解的信息，AnnotationTransactionAttributeSource解析事务注解  \* 2）、事务拦截器：  \* TransactionInterceptor；保存了事务属性信息，事务管理器；  \* 他是一个 MethodInterceptor；  \* 在目标方法执行的时候；  \* 执行拦截器链；  \* 事务拦截器：  \* 1）、先获取事务相关的属性  \* 2）、再获取PlatformTransactionManager，如果事先没有添加指定任何transactionmanger  \* 最终会从容器中按照类型获取一个PlatformTransactionManager；  \* 3）、执行目标方法  \* 如果异常，获取到事务管理器，利用事务管理回滚操作；  \* 如果正常，利用事务管理器，提交事务 |

1. **示例**

步骤：



配置类

|  |
| --- |
| @EnableTransactionManagement  @ComponentScan("com.atguigu.tx")  @Configuration  **public** **class** TxConfig {    //数据源  @Bean  **public** DataSource dataSource() **throws** Exception{  ComboPooledDataSource dataSource = **new** ComboPooledDataSource();  dataSource.setUser("root");  dataSource.setPassword("123456");  dataSource.setDriverClass("com.mysql.jdbc.Driver");  dataSource.setJdbcUrl("jdbc:mysql://localhost:3306/test");  **return** dataSource;  }    @Bean  **public** JdbcTemplate jdbcTemplate() **throws** Exception{  //Spring对@Configuration类会特殊处理；给容器中加组件的方法，多次调用都只是从容器中找组件  JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource());  **return** jdbcTemplate;  }    //注册事务管理器在容器中  @Bean  **public** PlatformTransactionManager transactionManager() **throws** Exception{  **return** **new** DataSourceTransactionManager(dataSource());  }  } |

UserDao类

|  |
| --- |
| @Repository  **public** **class** UserDao {  @Autowired  **private** JdbcTemplate jdbcTemplate;    **public** **void** insert(){  String sql = "INSERT INTO `user`(id,name,age,addr) VALUES(?,?,?,?)";  String username = UUID.*randomUUID*().toString().substring(0, 5);  jdbcTemplate.update(sql,3, username,19,"dgthgf");    }  } |

UserService类

|  |
| --- |
| @Service("userService01")  **public** **class** UserService {  @Autowired  **private** UserDao userDao;    @Transactional  **public** **void** insertUser(){  userDao.insert();  //otherDao.other();xxx  System.***out***.println("插入完成...");  **int** i = 10/0;  }  } |

测试类

|  |
| --- |
| @Test  **public** **void** test01(){  AnnotationConfigApplicationContext applicationContext =  **new** AnnotationConfigApplicationContext(TxConfig.**class**);    UserService userService = applicationContext.getBean(UserService.**class**);    userService.insertUser();  applicationContext.close();  } |

Service执行的时候抛出异常，插入数据则回滚，不会成功插入。

1. **扩展原理**
2. **BeanFactoryPostProcessor**

beanFactory的后置处理器

在BeanFactory标准初始化之后调用，来定制和修改BeanFactory的内容；

**执行时机**：所有的bean定义已经保存加载到beanFactory，但是bean的实例还未创建

**原理**：

1）、ioc容器创建对象

2）、调用invokeBeanFactoryPostProcessors(beanFactory);

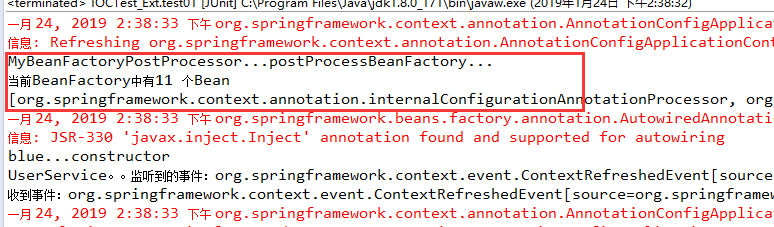
3）、如何找到所有的BeanFactoryPostProcessor并执行他们的方法？

直接在BeanFactory中找到所有类型是BeanFactoryPostProcessor的组件，并执行他们的方法

在初始化创建其他组件前面执行

|  |
| --- |
| @Component  **public** **class** MyBeanFactoryPostProcessor **implements** BeanFactoryPostProcessor {  @Override  **public** **void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {  System.***out***.println("MyBeanFactoryPostProcessor...postProcessBeanFactory...");  **int** count = beanFactory.getBeanDefinitionCount();  String[] names = beanFactory.getBeanDefinitionNames();  System.***out***.println("当前BeanFactory中有"+count+" 个Bean");  System.***out***.println(Arrays.*asList*(names));  }  } |

测试：



1. **BeanDefinitionRegistryPostProcessor**

BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子类。

**执行时机**：在所有bean定义信息将要被加载，bean实例还未创建的；因此他在BeanFactoryPostProcessor执行之前执行。

优先于BeanFactoryPostProcessor执行；

**作用**：利用BeanDefinitionRegistryPostProcessor给容器中再额外添加一些组件；

**原理**：

1）、ioc创建对象

2）、refresh()-》invokeBeanFactoryPostProcessors(beanFactory);

3）、从容器中获取到所有的BeanDefinitionRegistryPostProcessor组件。

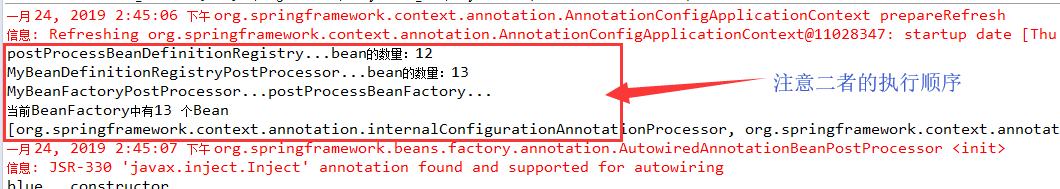
4）、依次触发所有的postProcessBeanDefinitionRegistry()方法

5）、再来触发postProcessBeanFactory()方法BeanFactoryPostProcessor；

6）、再来从容器中找到BeanFactoryPostProcessor组件；然后依次触发postProcessBeanFactory()方法

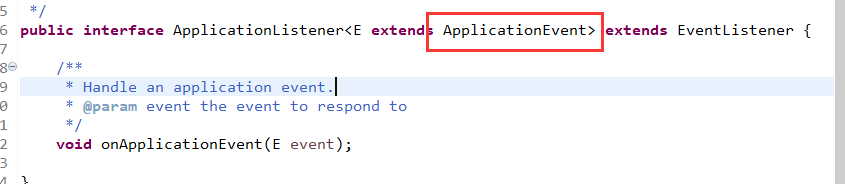
|  |
| --- |
| @Component  **public** **class** MyBeanDefinitionRegistryPostProcessor **implements** BeanDefinitionRegistryPostProcessor{  **public** **void** postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) **throws** BeansException {  System.***out***.println("MyBeanDefinitionRegistryPostProcessor...bean的数量："+beanFactory.getBeanDefinitionCount());  }  //BeanDefinitionRegistry Bean定义信息的保存中心，以后BeanFactory就是按照BeanDefinitionRegistry里面保存的每一个bean定义信息创建bean实例；  **public** **void** postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) **throws** BeansException {  // **TODO** Auto-generated method stub  System.***out***.println("postProcessBeanDefinitionRegistry...bean的数量："+registry.getBeanDefinitionCount());  //方式1：  //RootBeanDefinition beanDefinition = new RootBeanDefinition(Blue.class);  //方式2：  AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.*rootBeanDefinition*(Blue.**class**).getBeanDefinition();  //向IOC容器注入一个bean  registry.registerBeanDefinition("hello", beanDefinition);  }  } |

测试：



1. **ApplicationListener**

**作用**：监听容器中发布的事件。完成事件驱动模型开发；



是一个接口，监听ApplicationEvent及其子类的事件

编写自己的监听器

步骤1：写一个监听器（ApplicationListener实现类）来监听某个事件（ApplicationEvent及其子类）

步骤2：把监听器加入到容器；

|  |
| --- |
| @Component  **public** **class** MyApplicationListener **implements** ApplicationListener<ApplicationEvent> {  @Override  **public** **void** onApplicationEvent(ApplicationEvent event) {  System.***out***.println("收到事件：" + event);  }  } |

步骤3：只要容器中有相关事件的发布，我们就能监听到这个事件；

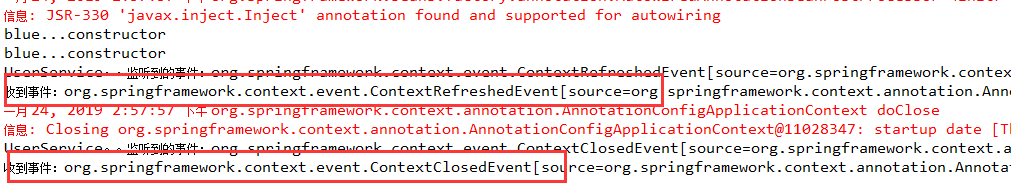
测试代码

|  |
| --- |
| @Test  **public** **void** test01(){  AnnotationConfigApplicationContext applicationContext = **new** AnnotationConfigApplicationContext(ExtConfig.**class**);  applicationContext.close();  } |

发现有两个事件，一个是容器刷新事件，一个是容器关闭事件，这是spring内置的事件。

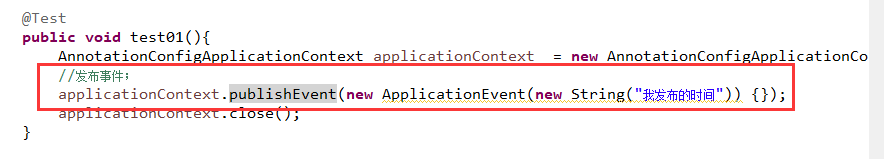
1）、ContextRefreshedEvent：容器刷新完成（所有bean都完全创建）会发布这个事件；

2）、ContextClosedEvent：关闭容器会发布这个事件；



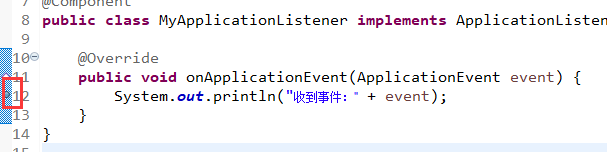
我们自己发布事件

applicationContext.publishEvent()

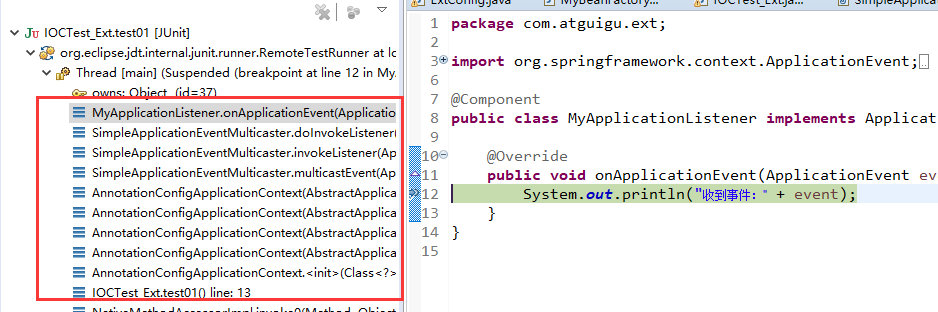


**原理**

在自定义的listener代码中打断点



Debug方式启动，红圈就是堆栈的执行流程



启动spring，容器完成刷新的事件执行流程

1）、refresh()

2）、finishRefresh();容器刷新完成会发布ContextRefreshedEvent事件

3）、publishEvent(new ContextRefreshedEvent(this));

4）、获取事件多播器（派发器）

getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);

5）、invokeListener(listener, event);

6）、执行回调函数listener.onApplicationEvent(event);

整体执行流程：

1）、refresh()

2）、initApplicationEventMulticaster();

初始化事件多播器ApplicationEventMulticaster；

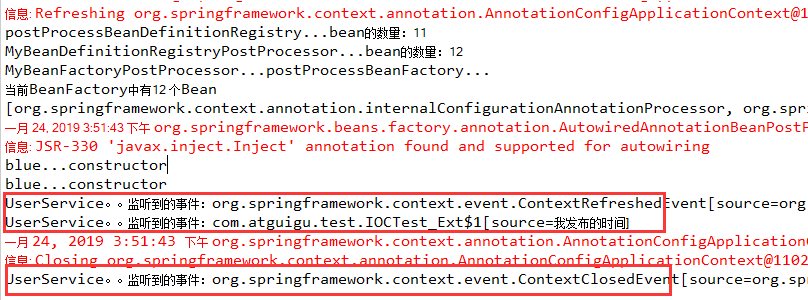
3）、registerListeners();从容器中拿到所有的监听器，把他们注册到applicationEventMulticaster中；

**@EventListener**

一般我们不编写ApplicationListener的子类，这是比较麻烦的事，通常都是在任意的类的方法上添加@EventListener注解。

|  |
| --- |
| @Service  **public** **class** UserService {    @EventListener(classes={ApplicationEvent.**class**})  **public** **void** listen(ApplicationEvent event){  System.***out***.println("UserService。。监听到的事件："+event);  }  } |

启动测试，效果跟实现ApplicationListener是一样的



**原理**：使用EventListenerMethodProcessor处理器来解析方法上的@EventListener；

SmartInitializingSingleton类的方法afterSingletonsInstantiated（）

finishBeanFactoryInitialization(beanFactory);初始化剩下的单实例bean；

1. **Servlet3.0**
2. **简介**

参考文档：

https://www.ibm.com/developerworks/cn/java/j-lo-servlet30/index.html

新特性：

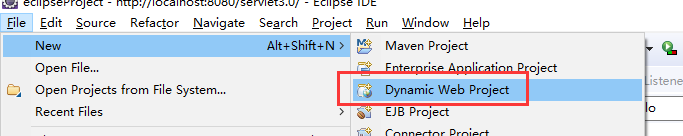
1）、异步处理支持：有了该特性，Servlet 线程不再需要一直阻塞，直到业务处理完毕才能再输出响应，最后才结束该 Servlet 线程。在接收到请求之后，Servlet 线程可以将耗时的操作委派给另一个线程来完成，自己在不生成响应的情况下返回至容器。针对业务处理较耗时的情况，这将大大减少服务器资源的占用，并且提高并发处理速度。

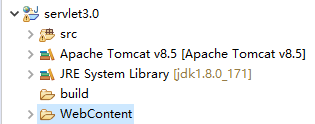
2）、新增的注解支持：该版本新增了若干注解，用于简化 Servlet、过滤器（Filter）和监听器（Listener）的声明，这使得 web.xml 部署描述文件从该版本开始不再是必选的了。

3）、可插性支持：熟悉 Struts2 的开发者一定会对其通过插件的方式与包括 Spring 在内的各种常用框架的整合特性记忆犹新。将相应的插件封装成 JAR 包并放在类路径下，Struts2 运行时便能自动加载这些插件。现在 Servlet 3.0 提供了类似的特性，开发者可以通过插件的方式很方便的扩充已有 Web 应用的功能，而不需要修改原有的应用。

1. **示例**

步骤1：新建动态web项目servlet3.0



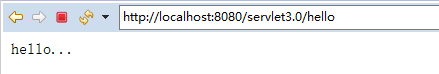


步骤2：新建HelloServlet

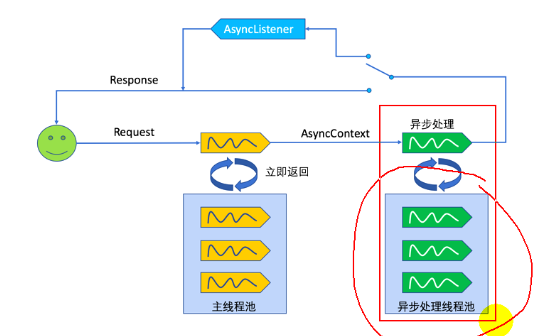
|  |
| --- |
| @WebServlet("/hello")  **public** **class** HelloServlet **extends** HttpServlet {  @Override  **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  //super.doGet(req, resp);  System.***out***.println(Thread.*currentThread*()+" start...");  **try** {  sayHello();  } **catch** (Exception e) {  e.printStackTrace();  }  resp.getWriter().write("hello...");  System.***out***.println(Thread.*currentThread*()+" end...");  }    **public** **void** sayHello() **throws** Exception{  System.***out***.println(Thread.*currentThread*()+" processing...");  Thread.*sleep*(3000);  }  } |

此处添加了注解@WebServlet

将项目加入到容器，启动容器测试



1. **Servlet3.0异步处理**



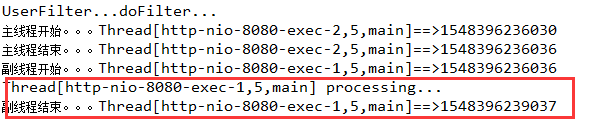
当一个请求过来的时候，主线程接受到请求，如果处理时间比较长的话，主线程则一直阻塞等待处理完成后才能返回，当请求比较少的时候，没关系，但是当用户请求过多的时候，主线程一直等待，无法接收新用户请求，此时我们可以考虑异步处理，主线程接收到请求，将具体的处理事务给子线程处理，自己可以马上空闲，继续接收新的用户请求，等子线程处理完成后再将处理结果返回给用户。

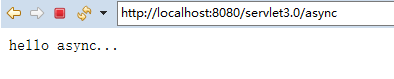
一般Tomcat是有主线程池来对线程进行管理，我们也可以对子线程进行线程池管理，当然springMVC有对子线程处理的线程池。

新建异步处理的servlet

|  |
| --- |
| @WebServlet(value="/async",asyncSupported=**true**)  **public** **class** HelloAsyncServlet **extends** HttpServlet {  @Override  **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  //1、支持异步处理asyncSupported=true  //2、开启异步模式  System.***out***.println("主线程开始。。。"+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  AsyncContext startAsync = req.startAsync();    //3、业务逻辑进行异步处理;开始异步处理  startAsync.start(**new** Runnable() {  @Override  **public** **void** run() {  **try** {  System.***out***.println("副线程开始。。。"+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  sayHello();  //获取到异步上下文  AsyncContext asyncContext = req.getAsyncContext();  //4、获取响应  ServletResponse response = asyncContext.getResponse();  response.getWriter().write("hello async...");  System.***out***.println("副线程结束。。。"+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  //处理完调用,必须最后调用，要不然会出错  startAsync.complete();  } **catch** (Exception e) {  e.printStackTrace();  }  }  });  System.***out***.println("主线程结束。。。"+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  }  **public** **void** sayHello() **throws** Exception{  System.***out***.println(Thread.*currentThread*()+" processing...");  Thread.*sleep*(3000);  }  } |

测试：





1. **ServletContainerInitializer**

Servlet容器启动会扫描，当前应用里面每一个jar包的ServletContainerInitializer的实现。

要想servlet容器加载到jar的入口，首先我必须提供ServletContainerInitializer的实现类；其次必须绑定在，META-INF/services/javax.servlet.ServletContainerInitializer文件中，内容就是ServletContainerInitializer实现类的全类名；

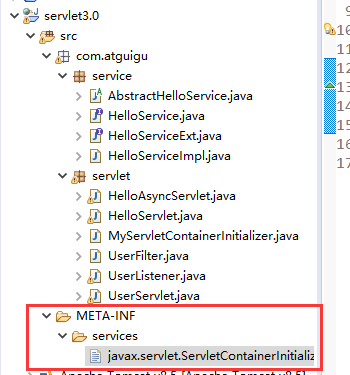
新建MyServletContainerInitializer类，它是ServletContainerInitializer的实现类

|  |
| --- |
| **public** **class** MyServletContainerInitializer **implements** ServletContainerInitializer {  /\*\*  \* 应用启动的时候，会运行onStartup方法；  \*  \* Set<Class<?>> arg0：感兴趣的类型的所有子类型；  \* ServletContext arg1:代表当前Web应用的ServletContext；一个Web应用一个ServletContext；  \*  \* 1）、使用ServletContext注册Web组件（Servlet、Filter、Listener）  \* 2）、使用编码的方式，在项目启动的时候给ServletContext里面添加组件；  \* 必须在项目启动的时候来添加；  \* 1）、ServletContainerInitializer得到的ServletContext；  \* 2）、ServletContextListener得到的ServletContext；  \*/  @Override  **public** **void** onStartup(Set<Class<?>> arg0, ServletContext sc) **throws** ServletException {  //注册组件 ServletRegistration  ServletRegistration.Dynamic servlet = sc.addServlet("userServlet", **new** UserServlet());  //配置servlet的映射信息  servlet.addMapping("/user");  } |

新建UserServlet

|  |
| --- |
| **public** **class** UserServlet **extends** HttpServlet {  @Override  **protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  resp.getWriter().write("tomcat...");  }  } |

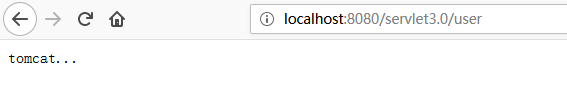
新建文件javax.servlet.ServletContainerInitializer文件，具体位置如下所示：



文件内容，将MyServletContainerInitializer类配置到该文件

|  |
| --- |
| com.atguigu.servlet.MyServletContainerInitializer |

启动容器测试



1. **注册三大组件（listener、filter、servlet）**

由于在上面已经注册了servlet，因此在这不在不再展示代码，只展示增量代码

新建UserListener

|  |
| --- |
| /\*\*  \* 监听项目的启动和停止  \* **@author** lfy  \*  \*/  **public** **class** UserListener **implements** ServletContextListener {  //监听ServletContext销毁  @Override  **public** **void** contextDestroyed(ServletContextEvent arg0) {  System.***out***.println("UserListener...contextDestroyed...");  }  //监听ServletContext启动初始化  @Override  **public** **void** contextInitialized(ServletContextEvent arg0) {  ServletContext servletContext = arg0.getServletContext();  System.***out***.println("UserListener...contextInitialized...");  }  } |

新建UserFilter类

|  |
| --- |
| **public** **class** UserFilter **implements** Filter {  @Override  **public** **void** destroy() {  }  @Override  **public** **void** doFilter(ServletRequest arg0, ServletResponse arg1, FilterChain arg2)  **throws** IOException, ServletException {  // 过滤请求  System.***out***.println("UserFilter...doFilter...");  //放行  arg2.doFilter(arg0, arg1);    }  @Override  **public** **void** init(FilterConfig arg0) **throws** ServletException {  }  } |

在MyServletContainerInitializer中注册listener和filter，代码也是放在onStartup方法中

|  |
| --- |
| //注册Listener  sc.addListener(UserListener.**class**);    //注册Filter FilterRegistration  FilterRegistration.Dynamic filter = sc.addFilter("userFilter", UserFilter.**class**);  //配置Filter的映射信息  filter.addMappingForUrlPatterns(EnumSet.*of*(DispatcherType.***REQUEST***), **true**, "/\*"); |

1. **@HandlesTypes**

容器启动的时候会将@HandlesTypes指定的这个类型下面的子类（实现类，子接口等）传递过来；

新建接口HelloService

|  |
| --- |
| **public** **interface** HelloService {} |

新建HelloService的子类AbstractHelloService

|  |
| --- |
| **public** **abstract** **class** AbstractHelloService **implements** HelloService {  } |

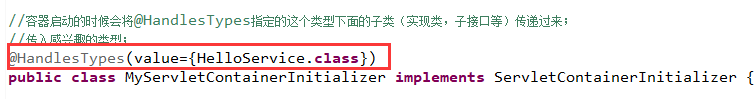
新建HelloService的子类HelloServiceExt

|  |
| --- |
| **public** **interface** HelloServiceExt **extends** HelloService {  } |

新建HelloService的子类HelloServiceImpl

|  |
| --- |
| **public** **class** HelloServiceImpl **implements** HelloService {  } |

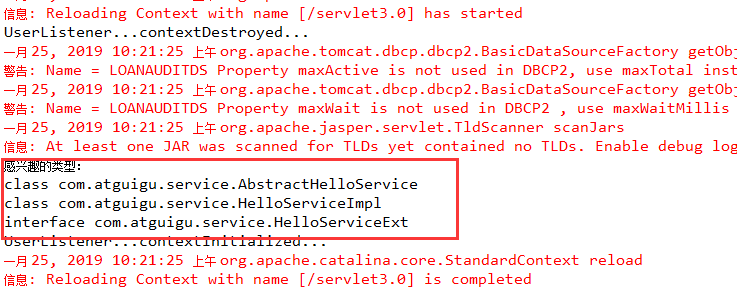
修改MyServletContainerInitializer类，添加注解@HandlesTypes



在方法onStartup中添加如下代码

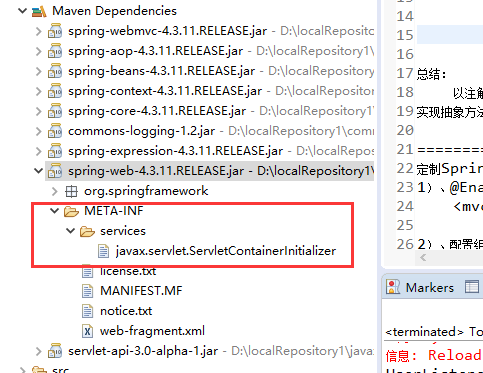
|  |
| --- |
| System.***out***.println("感兴趣的类型：");  **for** (Class<?> claz : arg0) {  System.***out***.println(claz);  } |

启动容器



1. **整合springMVC原理**

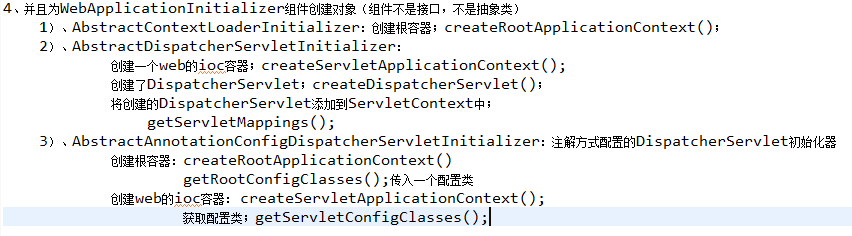
在spring-web的jar包下我们可以看到如下文件



web容器在启动的时候，会扫描每个jar包下的META-INF/services/javax.servlet.ServletContainerInitializer

然后加载这个文件指定的类SpringServletContainerInitializer

spring的应用一启动会加载感兴趣的WebApplicationInitializer接口的下的所有组件；



总结：

以注解方式来启动SpringMVC；继承

AbstractAnnotationConfigDispatcherServletInitializer；

实现抽象方法指定DispatcherServlet的配置信息；

1. **Springmvc的异步处理**

**Callable**

**流程**：

1、控制器返回Callable

2、Spring异步处理，将Callable 提交到 TaskExecutor 使用一个隔离的线程进行执行

3、DispatcherServlet和所有的Filter退出web容器的线程，但是response 保持打开状态；

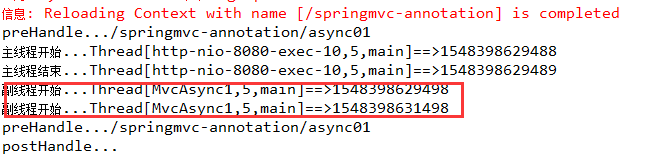
4、Callable返回结果，SpringMVC将请求重新派发给容器，恢复之前的处理；

5、根据Callable返回的结果。SpringMVC继续进行视图渲染流程等（从收请求-视图渲染）。

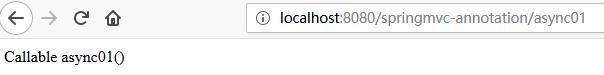
新建AsyncController类

|  |
| --- |
| @Controller  **public** **class** AsyncController {  /\*\*  \* preHandle.../springmvc-annotation/async01  主线程开始...Thread[http-bio-8081-exec-3,5,main]==>1513932494700  主线程结束...Thread[http-bio-8081-exec-3,5,main]==>1513932494700  =========DispatcherServlet及所有的Filter退出线程============================    ================等待Callable执行==========  副线程开始...Thread[MvcAsync1,5,main]==>1513932494707  副线程开始...Thread[MvcAsync1,5,main]==>1513932496708  ================Callable执行完成==========    ================再次收到之前重发过来的请求========  preHandle.../springmvc-annotation/async01  postHandle...（Callable的之前的返回值就是目标方法的返回值）  afterCompletion...    异步的拦截器:  1）、原生API的AsyncListener  2）、SpringMVC：实现AsyncHandlerInterceptor；  \* **@return**  \*/  @ResponseBody  @RequestMapping("/async01")  **public** Callable<String> async01(){  System.***out***.println("主线程开始..."+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());    Callable<String> callable = **new** Callable<String>() {  @Override  **public** String call() **throws** Exception {  System.***out***.println("副线程开始..."+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  Thread.*sleep*(2000);  System.***out***.println("副线程开始..."+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  **return** "Callable async01()";  }  };    System.***out***.println("主线程结束..."+Thread.*currentThread*()+"==>"+System.*currentTimeMillis*());  **return** callable;  }  } |

测试结果：

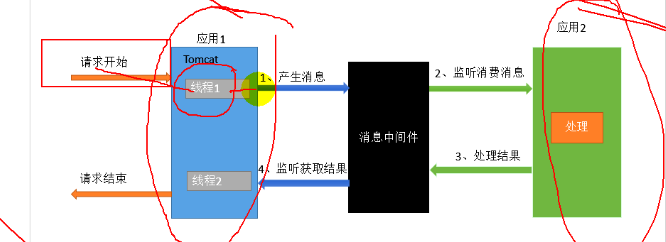


执行了两次，是因为spring将请求重新派发给容器



**DeferredResult**

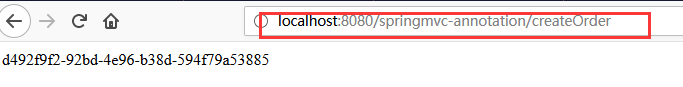
有的时候，web服务器在接收到请求后，并不会对请求做任何处理，而是将这个请求操作发送到消息队列，真正处理该请求的服务器会从消息队列里面取消息，然后进行处理，处理完成后返回结果。



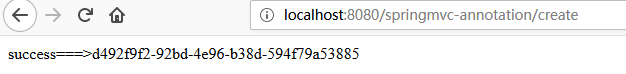
|  |
| --- |
| @ResponseBody  @RequestMapping("/createOrder")  **public** DeferredResult<Object> createOrder(){  //设置3秒，也就是创建订单必须在3秒内处理成功，如果没有处理成功，则返回客户端create fail...  DeferredResult<Object> deferredResult = **new** DeferredResult<>((**long**)3000, "create fail...");  DeferredResultQueue.*save*(deferredResult);  **return** deferredResult;  }    @ResponseBody  @RequestMapping("/create")  **public** String create(){  //创建订单  String order = UUID.*randomUUID*().toString();  DeferredResult<Object> deferredResult = DeferredResultQueue.*get*();  deferredResult.setResult(order);  **return** "success===>"+order;  } |

测试：

首先发送如下请求,顺序很重要



然后必须在3秒内发送如下请求



这样createOrder请求才能执行成功。

1. **aaa**