# spring装配

## 注解自动化装配

### 创建可被发现的bean

在类上添加@Component注解,这就告诉spring该类会作为组件类。组件的名为类名首字母小写。如果想自定义组件名，可以使用@Component注解的value属性指定组件名。

### 启用组件扫描

#### javaConfig开启

创建一个java类，在该类上添加@Configuration注解和@ComponentScan注解。

@Configuration注解指明该java类是一个配置类

@ComponentScan注解表明扫描以该类为基础包的包中的组件。

**该注解默认扫描和该配置类同包的组件，如果想手动指定基础包可以使用属性basePackages或baseClasses指定基础包**

#### XML开启

在xml配置文件中添加<context:component-scan base-package="包"/>标签即可开启扫描以base-package属性所指定的包下的组件。

### 测试组件扫描

#### JUnit集成Spring

使用@RunWith(SpringJUnit4ClassRunner.class)告知JUnit创建Spring上下文

使用@ContextConfiguration(classes=..)指定spring配置文件

## 注解实现依赖注入

### 注解的位置

可以用在成员变量上、可以用在setter方法上、可以用在构造方法上。

如果是用在构造器或setter方法上，那么spring会尝试满足方法参数上所生命的依赖。

### 注解的选择

有两个注解可以实现注入依赖的功能：@Autowired和@Inject注解

这两个注解基本上是一样的,@Inject是Java规范中的一个注解

## Java代码装配

创建一个JavaConfig类,在该类上添加@Configuration注解,该注解表明该类是一个配置类,然后开启注解扫描,在上一节中已经说明怎么开启,然后在该配置类中添加一个方法,这个方法会创建所需类型的实例,并返回。然后在该方法上添加@Bean注解，这样就将该方法的返回值注册到Spring容器中了。默认情况下，bean的ID与带有@Bean注解的方法名是一样的，如果想指定一个ID，那么可以使用@Bean注解的name属性指定一个不同的名字。

|  |
| --- |
| **@Bean**  **public CompactDisc sgtPeppers(){**  **return new CompactDisc();**  **}** |

手动指定ID:

|  |
| --- |
| **@Bean(name="testBean")**  **public CompactDisc sgtPeppers(){**  **return new CompactDisc();**  **}** |

## Java代码实现依赖注入

在javaConfig实现依赖注入有两种方法

一种是在需要依赖其他bean的地方调用声明该bean的方法,比如我们需要依赖上面的CompactDisc实例,那么可以直接引用sgtPeppers()方法,这样就可以获取到spring中保存的该方法返回值的实例。此处需要注意,当调用sgtPeppers()方法时并不会每次都创建一个新的实例,spring会拦截添加了@Bean注解的方法的调用

|  |
| --- |
| **@Bean**  **public CDPlayer cdPlayer(){**  **return new CDPlayer(sgtPeppers());**  **}** |

另一种是在创建bean的方法参数里添加需要引入的类

|  |
| --- |
| **@Bean**  **public CDPlayer cdPlayer(CompactDisc compactDisc){**  **return new CDPlayer(compactDisc);**  **}** |

## XML实现装配

在spring的xml配置文件中使用beans作为根标签,并声明必要的xsd文件,然后在beans标签下创建bean子标签。

|  |
| --- |
| **<bean class="类全限定名"/>** |

注:此处没有指定ID,默认的ID为"全限定名#0",这样在别处如果用ID引用该bean时,可以使用该ID引用。如果需要指定ID,可以在bean标签中使用id属性指定

|  |
| --- |
| **<bean id="myBean" class="类全限定名"/>** |

# Spring配置加载

## 利用Servlet3.0规范实现Spring配置加载

在servlet3.0环境中,容器会通过java SPI 机制发现javax.servlet.ServletContainerInitializer接口的实现类,如果能发现的话,就会用它来配置Servlet容器.

spring利用这一特性,提供了这个接口的实现,名为SpringServletContainerInitializer,这个类反过来又会查找实现WebApplicationInitializer的类并将配置的任务交给它们来完成,AbstractAnnotationConfigDispatcherServletInitializer类实现了WebApplicationInitializer.所以我们继承AbstractAnnotationConfigDispatcherServletInitializer并重写其中的方法,即可实现Spring的初始化,这样就不需要在web.xml中配置DispatcherServlet和ContextLoaderListener了.

## ContextLoaderListener中加载

## DispatcherServlet中加载

# spring初始化bean的过程

* 初始化容器,包括初始化配置信息
* 当获取bean时先做实例化bean前的处理
* 实例化bean
* 实例化bean后的处理
* 属性注入前的处理
* 属性注入
* 处理bean所实现的各种aware接口
* 初始化(init方法)前的处理
* 执行init方法
* 初始化后的处理

[ 启动容器 ]初始化容器

1 调用BeanFactoryPostProcessor工厂后处理器的postProcessBeanFactory()对BeanDefinition对象进行后处理

这是ApplicationContext启动时，refresh方法中执行的，加工处理BeanDefinition(spring提供了一个CustomEditorConfigurer实现了BeanFactoryPostProcessor，用来实现对自定义属性编辑器的注册。之所以通过它来实现注册是因为工厂后处理器可以访问到beanFactory容器，从而获取用户自定义的BeanDefinition从而实例化并注册)

[ 通过调用getBean()获取某个bean（ApplicationContext或者BeanFactory）]

2 调用InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation()方法

如果容器注册了InstantiationAwareBeanPostProcessor接口，在实例化bean之前，将调用接口的postProcessBeforeInstantiation()

3 实例化

根据配置情况调用Bean的构造函数或工厂方法实例化Bean

4 调用InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation()方法

如果容器注册了InstantiationAwareBeanPostProcessor接口，在实例化bean之后，将调用接口的postProcessAfterInstantiation()

5 调用InstantiationAwareBeanPostProcessor的postProcessPropertyValues()方法

如果<bean>配置了<property>属性，容器在第六步着手将配置值设置到bean对应的属性当中，不过在设置每个属性之前将先调用InstantiationAwareBeanPostProcessor的postProcessPropertyValues()方法

6 设置属性值

调用bean的属性设置方法设置属性

7 调用BeanNameAware的setBeanName()方法

如果bean实现了org.springframework.beans.factory.BeanNameAware接口，将调用setBeanName()接口方法，将配置文件中该bean对应的名称设置到bean中

8 调用BeanFactoryAware的setBeanFactory()方法

如果bean实现了org.springframework.beans.factory.BeanFactoryAware接口，将调用setBeanFactory()接口方法，将beanFactory容器设置到bean中

9 调用ApplicationContextAware的setApplicationContext()方法

如果bean实现了org.springframework.context.ApplicationContextAware接口，会调用该接口setApplicationContext方法

10 调用BeanPostProcessor的postProcessBeforeInitialization()方法

（容器初始化时注册，在此调用/ 而若只应用BeanFactory，则要在客户应用代码中手动注册）

如果BeanFactory装配了org.springframework.beans.factory.config.BeanPostProcessor后处理器，将会调用BeanPostProcessor的Object postProcessBeforeInitialization(Object bean,String beanName)接口方法对bean进行加工操作。入参bean是当前处理的bean，而beanName则是当前bean的配置名，返回的对象为加工处理后的bean。用户可以使用该方法对bean进行特殊处理，甚至改变bean的行为。它为容器提供了bean进行后续加工的切入点，Spring容器所提供的各种功能（如AOP，动态代理等）都通过BeanPostProcessor实施。

11 调用InitializingBean的afterPropertiesSet()方法

如果bean实现了InitializingBean接口，将调用该接口的afterPropertiesSet()。

12 调用通过init-method属性配置的初始化方法

如果bean通过配置文件中init-method属性定义了初始化方法，会执行这个方法。

13 调用BeanPostProcessor的postProcessAfterInitialization()方法

如果在bean配置中指定bean的作用范围scope="prototype"，会将bean返回给调用者，调用者负责bean后续生命周期的管理，spring不再管理这些bean的生命周期。如果作用范围为scope="singleton"，则将bean放入Spring的Ioc容器的缓存池中，并将bean引用返回给调用者，Spring继续对这些bean进行后续的生命周期管理

(singleton)Spring缓存中准备就绪的bean

(prototype)准备就绪的bean

# BeanPostProcessor和BeanFactoryPostProcessor

BeanFactoryPostProcessor是用于对BeanFactory做处理的,而BeanPostProcessor是对Bean进行处理的.他们提供的api为:

# spring标签和注解

## <context:annotation-config/>和<context:component-scan/>

如果使用<context:component-scan/>的话,<context:annotation-config/>标签就可以不用了

## @Controller和@RestController

@RestController是@Controller和@ResponseBody的结合

## @ConfigurationProperties

该注解用于代替@Value注解,当一个类中有多个参数需要注入式,该注解比较便捷

注意:如果使用该注解需要两个条件,

1、使用@EnableConfigurationProperties注解开启该功能

2、将添加该注解的类纳入spring管理，有多种方式可以实现将bean纳入spring管理

* 1. 添加component注解
  2. 使用@Bean注解
  3. 在xml中定义bean
  4. 使用@EnableConfigurationProperties的属性引入该类（仅适用此处讨论的注解）

## @PropertySource

该注解用于代替PropertyPlaceholderConfigurer,使用注解的形式引入资源文件

## @ContextConfiguration

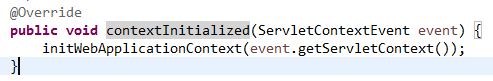
在测试时通过该注解指定配置文件的位置

# Spring配置解析

## @Configuration解析过程

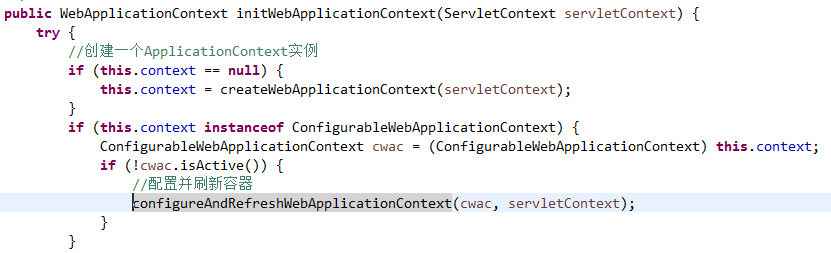
本次分析从spring容器的启动开始

**ContextLoaderListener监听到web项目初始化完毕之后执行contextInitialized()方法**



**initWebApplicationContext实现**

在ContextLoader类中



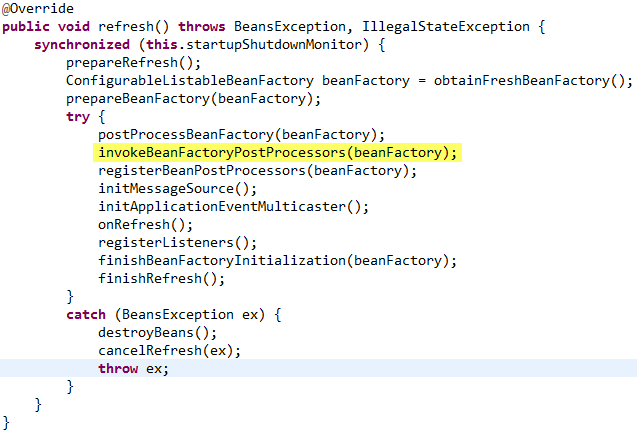
**configureAndRefreshWebApplicationContext()实现**

在ContextLoader类中

****

**refresh方法**

在AbstractApplicationContext中,在该方法中会加载spring的配置信息,并加载bean的定义信息,注册监听器,执行工厂后处理bean,@Configuration注解就是在工厂后处理bean中解析的(ConfigurationClassPostProcessor类),

****

**invokeBeanFactoryPostProcessors()方法**

该方法在AbstractApplicationContext中

|  |
| --- |
| **protected** **void** invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory) {  PostProcessorRegistrationDelegate.*invokeBeanFactoryPostProcessors*(beanFactory, getBeanFactoryPostProcessors());  } |

**invokeBeanFactoryPostProcessors()方法**

该方法在PostProcessorRegistrationDelegate类中

|  |
| --- |
| **public** **static** **void** invokeBeanFactoryPostProcessors(){  String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.**class**, **true**, **false**);  List<BeanDefinitionRegistryPostProcessor> priorityOrderedPostProcessors = **new** ArrayList<BeanDefinitionRegistryPostProcessor>();  **for** (String ppName : postProcessorNames) {  **if** (beanFactory.isTypeMatch(ppName, PriorityOrdered.**class**)) {  priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.**class**));  processedBeans.add(ppName);  }  }  OrderComparator.*sort*(priorityOrderedPostProcessors);  registryPostProcessors.addAll(priorityOrderedPostProcessors);  //在该方法中遍历bean的定义信息,找到所有的带有Configuration注解的类,并进行进一步的解析,比如解析Import注解,解析Bean注解 *invokeBeanDefinitionRegistryPostProcessors*(priorityOrderedPostProcessors, registry);  } |

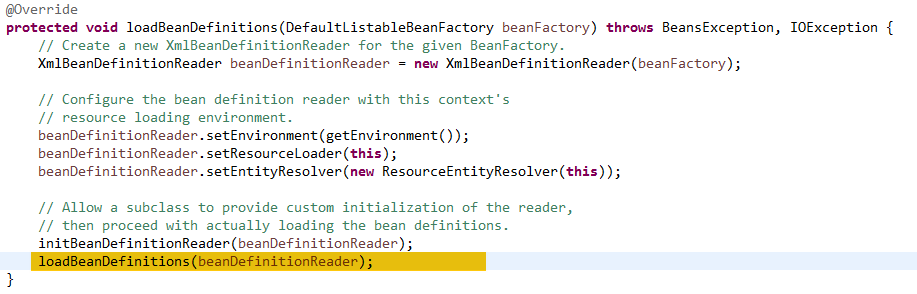
## Spring命名空间解析器

|  |
| --- |
| http://www.springframework.org/schema/p  org.springframework.beans.factory.xml.SimplePropertyNamespaceHandler  http://www.springframework.org/schema/mvc  org.springframework.web.servlet.config.MvcNamespaceHandler  http://www.springframework.org/schema/util  org.springframework.beans.factory.xml.UtilNamespaceHandler  http://www.springframework.org/schema/jee  org.springframework.ejb.config.JeeNamespaceHandler  http://www.springframework.org/schema/aop  org.springframework.aop.config.AopNamespaceHandler  http://www.springframework.org/schema/oxm  org.springframework.oxm.config.OxmNamespaceHandler  http://www.springframework.org/schema/jdbc  org.springframework.jdbc.config.JdbcNamespaceHandler  http://www.springframework.org/schema/cache  org.springframework.cache.config.CacheNamespaceHandler  http://mybatis.org/schema/mybatis-spring  org.mybatis.spring.config.NamespaceHandler  http://www.springframework.org/schema/c  org.springframework.beans.factory.xml.SimpleConstructorNamespaceHandler  http://www.springframework.org/schema/tx  org.springframework.transaction.config.TxNamespaceHandler  http://www.springframework.org/schema/integration  org.springframework.integration.config.xml.IntegrationNamespaceHandler  http://www.springframework.org/schema/integration/kafka  org.springframework.integration.kafka.config.xml.KafkaNamespaceHandler  http://www.springframework.org/schema/task  org.springframework.scheduling.config.TaskNamespaceHandler  http://www.springframework.org/schema/lang  org.springframework.scripting.config.LangNamespaceHandler  http://www.springframework.org/schema/context  org.springframework.context.config.ContextNamespaceHandler |

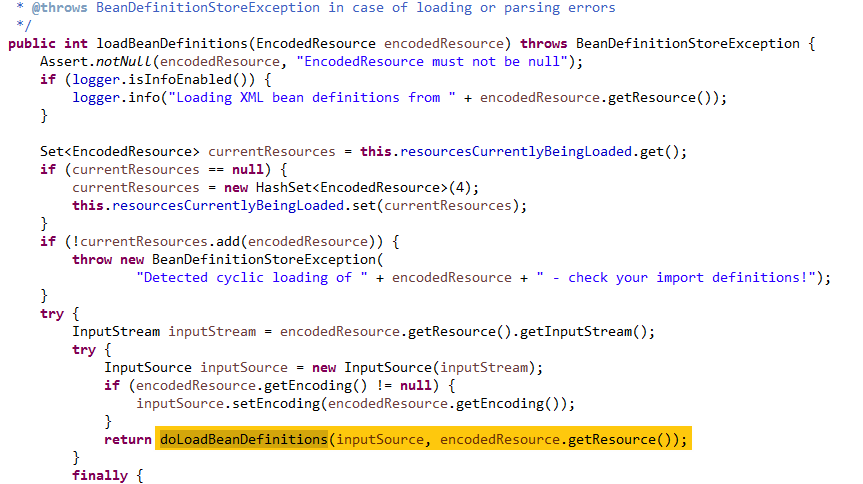
## Spring配置文件标签的加载和解析

XmlWebApplicationContext中的loadBeanDefinitions方法先读取配置文件,然后解析xml文件,解析xml的过程中遍历每一个标签以及子标签,然后根据标签名获取NameSpaceHandler,由对应的Handler处理具体的标签.这些NameSpaceHandler存放在jar包中的META-INF下的spring.handlers文件中,如下:(键为名称空间名,值为handler名)

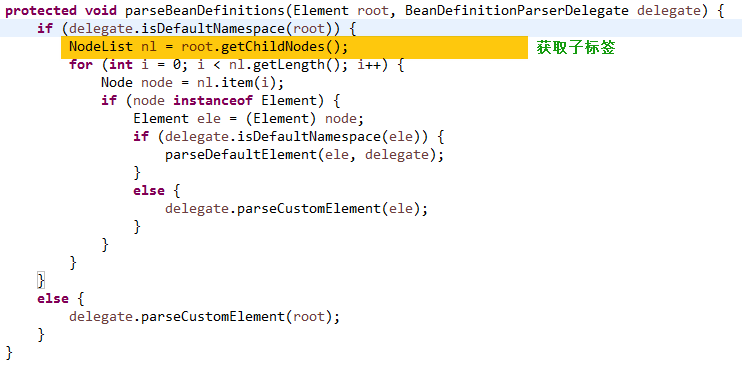




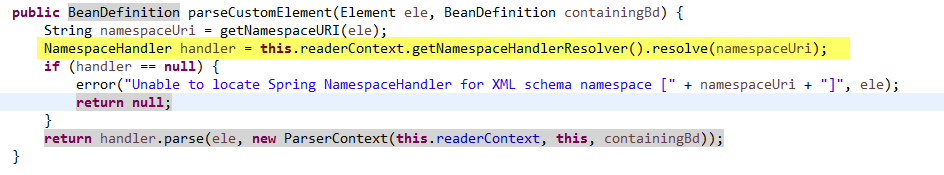
XmlWebApplicationContext ↑



XmlBeanDefinitionReader ↑



DefaultBeanDefinitionDocumentReader ↑

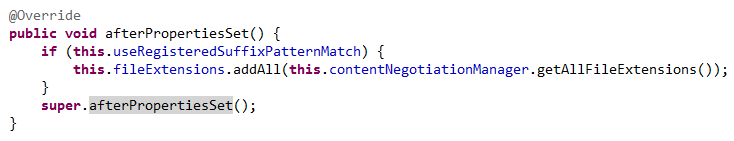


BeanDefinitionParserDelegate ↑

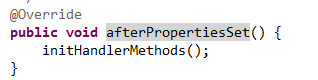
遍历每一个标签的时候在BeanDefinitionParserDelegate中的parseCustomElement方法中获取标签对应的NameSpaceHandler,由专门的NameSpaceHandler来执行具体的解析.

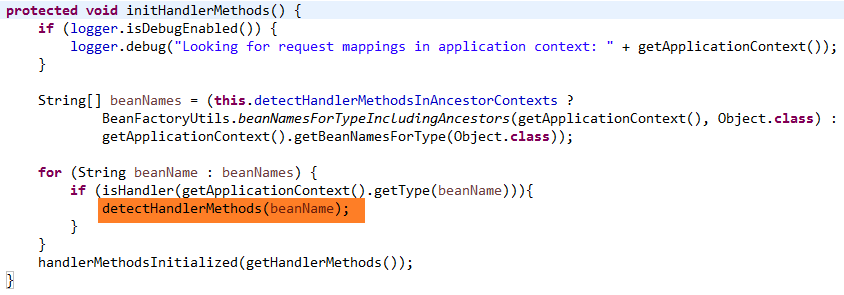
## RequestMapping注解的解析过程

RequestMappingHandlerMapping实现了InitializingBean接口,并实现了afterPropertiesSet方法(4.2之前是通过实现ApplicationContextAware中的setApplicationContext方法中实现的.DefaultAnnotationHandlerMapping也是通过实现ApplicationContextAware中的setApplicationContext方法来实现的)



RequestMappingHandlerMapping中的afterPropertiesSet方法 ↑







AbstractHandlerMethodMapping ↑

# spring内置组件

## ApplicationContextAware

通过继承ApplicationContextAware可以获取ApplicationContext实例的引用

只需要给出ApplicationContextAware的实现类,并将实现类注册到spring容器中,那么spring在对该实现类时会将applicationContext实例的引用设置到该实现类的对象中,在程序中可以通过该实现类的实例对ApplicationContext实例进行访问了

# Spring MVC

## HandlerExecutionChain的作用

HandlerExecutionChain中封装了HandlerMethod和HandlerInterceptor集合,在DispatcherServlet中调用getHandler方法,传入一个request对象可以获取对应的HandlerExecutionChain,getHandler方法中会调用HandlerMapping的getHandler方法获取HandlerExecutionChain,而HandlerMapping中的getHandler会根据request中的请求路径查找相应的HandlerMethod,然后调用getHandlerExecutionChain方法根据获取到的HandlerMethod创建一个HandlerExecutionChain,如果我们需要定义自己的HandlerExecutionChain,可以重写getHandlerExecutionChain方法,然后根据需要创建自己的HandlerExecutionChain方法.

为什么要自定义HandlerExecutionChain,可以这样分析,HandlerExecutionChain和HandlerMethod和Interceptor有关系,通过自定义HandlerExecutionChain可以对Method和Interceptor方法进行操作,比如获取methodName

## DispatcherServlet

### 重写DispatcherServlet中的doDispatcher

重写doDispatcher方法可以在一次请求中添加自己的逻辑

### DispatcherServlet的初始化过程

DispatcherServlet

FrameworkServlet

HttpServletBean

HttpServlet

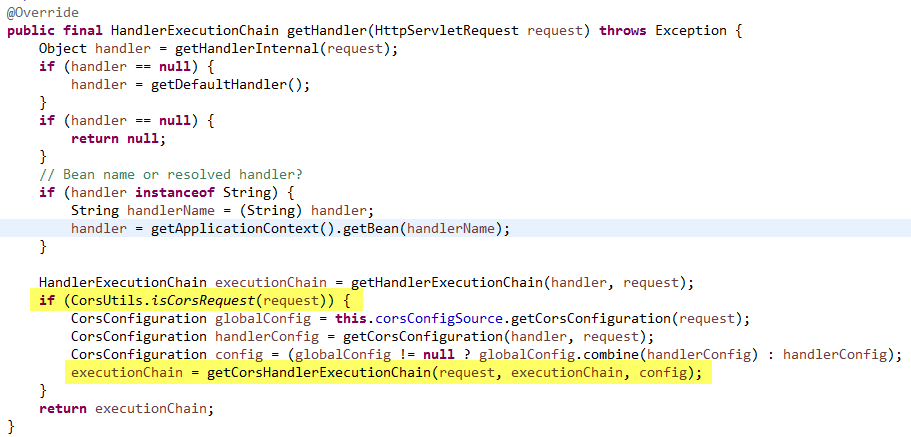
httpServlet继承GenericServlet,在该类中实现了init(ServletConfig)方法,并调用自己的init方法

HttpServletBean中重写了init方法,并调用自己的initServletBean方法

重写了initServletBean方法,在该方法中调用initApplicationContext方法,在init..Context方法中调用了对容器进行初始化,并添加了一个ContextRefreshListener监听器,在该监听器中执行本类的onApplicationEvent方法,调用onfresh方法

## spring4.2对跨域请求的支持

从4.2开始spring开始支持跨域请求,使用方法是在RequestMapping注解上添加@CrossOrigin注解,这样spring在获取HandlerMethod时会做特殊处理



AbstractHandlerMapping

## View Controller

我们可以直接指定路径与视图的映射关系,这种情况适用于静态场景,比如说现在有一个jsp页面,请求该jsp文件时不需要执行任何逻辑,我们就不需要在controller中通过配置RequestMapping来搭建请求路径与该jsp文件之间的映射,我们可以直接通过ViewController完成该功能.

**比如将"/"映射到"home"视图上.**

**方法一:**

|  |
| --- |
| <mvc:view-controller path="/" view-name="home"> |

**方法二:**

|  |
| --- |
| @Configuration  @EnableWebMvc  **public** **class** WebConfig **extends** WebMvcConfigurerAdapter {  @Override  **public** **void** addViewControllers(ViewControllerRegistry registry) {  registry.addViewController("/").setViewName("home");  }  } |

## Content Negotiation Manager

ContentNegotiationManager用于判断一个请求的ContentType,如json,xml,html

Spring中的一些组件根据ContentType不同会做出不同的处理,例如不同的ContentType需要不同的视图解析器给客户端做出不同的响应.有的需要返回html格式的响应,有的需要返回json,有的需要返回xml格式等等.

## WebDataBinder

Special **DataBinder** for data binding from **web request parameters** to JavaBean objects.

## WebBindingInitializer

WebBindingInitializer是用来初始化WebDataBinder的

# spring-data-jpa

## 引入依赖的途径

使用maven引入spring-data-jpa的方法是:

第一步：引入spring data release train bom

|  |
| --- |
| <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-releasetrain</artifactId>  <version>1.3.0.RELEASE</version>  <type>pom</type>  </dependency>  </dependencies>  </dependencyManagement> |

第二步：引入spring-data-jpa

|  |
| --- |
| <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-jpa</artifactId>  </dependency> |

## 接口介绍

### Repository

* Repository是spring data jpa的核心接口,
* 该接口需要两个类型参数,被管理的类的类型和被管理类中ID字段的类型.
* 该接口作为标记接口,用于捕获要持久化的类(通过类型参数获取)和该接口的扩展类

#### @ RepositoryDefinition

使用@RepositoryDefinition注解接口效果和继承Repository接口相同,都可以告知spring为该类生成代理类

### CRUDRepository

CRUDRepository在Repository接口的基础上定义了基本的CRUD操作方法

|  |
| --- |
| @NoRepositoryBean  **public** **interface** CrudRepository<T, ID **extends** Serializable> **extends** Repository<T, ID> {  <S **extends** T> S save(S entity);  <S **extends** T> Iterable<S> save(Iterable<S> entities);  T findOne(ID id);  **boolean** exists(ID id);  Iterable<T> findAll();  Iterable<T> findAll(Iterable<ID> ids);  **long** count();  **void** delete(ID id);  **void** delete(T entity);  **void** delete(Iterable<? **extends** T> entities);  **void** deleteAll();  } |

### PagingAndSortingRepository

在CRUDRepository的基础上PagingAndSortingRepository添加了分页和排序的相关功能

|  |
| --- |
| @NoRepositoryBean  **public** **interface** PagingAndSortingRepository<T, ID **extends** Serializable> **extends** CrudRepository<T, ID> {  Iterable<T> findAll(Sort sort);  Page<T> findAll(Pageable pageable);  } |

### 派生查询

派生count查询

|  |
| --- |
| public interface UserRepository extends CrudRepository<User, Long> {  Long countByLastname(String lastname);  } |

派生删除查询

|  |
| --- |
| public interface UserRepository extends CrudRepository<User, Long> {  Long deleteByLastname(String lastname);  List<User> removeByLastname(String lastname);  } |

## 定义查询方法

定义查询方法需要四步:

1. 定义一个接口继承Repository或它的子接口

|  |
| --- |
| interface PersonRepository extends Repository<Person, Long> { … } |

1. 在接口中按照规则定义一些方法

|  |
| --- |
| interface PersonRepository extends Repository<Person, Long> {  List<Person> findByLastname(String lastname);  } |

1. 在spring中开启JpaRepotitory功能

|  |
| --- |
| @EnableJpaRepositories  class Config {} |

1. 在需要用到Repository的地方进行注入

|  |
| --- |
| public class SomeClient {  @Autowired  private PersonRepository repository;  public void doSomething() {  List<Person> persons = repository.findByLastname("Matthews");  }  } |

## 定义查询方法的规则

### 方法名规则

|  |
| --- |
| public interface PersonRepository extends Repository<User, Long> {  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  // Enables the distinct flag for the query  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  // Enabling ignoring case for an individual property  List<Person> findByLastnameIgnoreCase(String lastname);  // Enabling ignoring case for all suitable properties  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  // Enabling static ORDER BY for a query  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  } |

### 属性表达式

加入你有个Person类,这个类有个Address变量,Address中有个zipcode变量,那么可以这样写

|  |
| --- |
| List<Person> findByAddressZipCode(ZipCode zipCode); |

这相当与x.address.zipCode

更好的写法是

|  |
| --- |
| List<Person> findByAddress\_ZipCode(ZipCode zipCode); |

### 特殊查询参数

Pageable and Sort

|  |
| --- |
| Page<User> findByLastname(String lastname, Pageable pageable);  Slice<User> findByLastname(String lastname, Pageable pageable);  List<User> findByLastname(String lastname, Sort sort);  List<User> findByLastname(String lastname, Pageable pageable); |

### 限制查询结果数量

Limiting the result size of a query with Top and First

User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);

# Spring Data Rest

Spring Data Rest是一个可以将Spring Data repository以restFull形式暴露出去的工具

## 开启spring data rest功能的方法

### 常规spring 项目

在常规spring项目中要想开启spring data rest,需要向spring中注册一个RepositoryRestMvcConfiguration实例,可以通过在Configuration类上添加@Import注解引入,也可以通过@bean注解直接声明

### spring boot项目

spring boot项目中只需要添加关于spring data rest的依赖,spring boot将会自动配置spring data rest.具体负责自动配置的类是RepositoryRestMvcAutoConfiguration

|  |
| --- |
| @Configuration  @ConditionalOnWebApplication  @ConditionalOnMissingBean(RepositoryRestMvcConfiguration.**class**)  @ConditionalOnClass(RepositoryRestMvcConfiguration.**class**)  @AutoConfigureAfter({ HttpMessageConvertersAutoConfiguration.**class**,  JacksonAutoConfiguration.**class** })  @EnableConfigurationProperties(RepositoryRestProperties.**class**)  @Import(**RepositoryRestMvcConfiguration.class**)  **public** **class** RepositoryRestMvcAutoConfiguration {  @Bean  **public** SpringBootRepositoryRestConfigurer springBootRepositoryRestConfigurer() {  **return** **new** SpringBootRepositoryRestConfigurer();  }  } |

## 暴露哪些Repository的逻辑

在项目中可能有很多Repository,但有些我们可能不想暴露出去,这时候需要利用暴露Repository的逻辑来保护一些资源

**这部分逻辑需要进一步测试**

### Default

默认情况下将会暴露所有repository接口下的资源,但可以通过@RestResource注解中的属性排除某一个repository资源

### All

暴露所有repository接口和通过注解(@RepositoryDefinition)声明的资源

### annotation

只有被@RestResource注解标注的repositories会被暴露

## 默认的暴露资源的路径规则

将Repository管理的domain作为资源以RestFull的形式暴露出去,将会暴露两个接口:

Repository所管理的domain名字的小写复数形式作为资源路径

以CarRepository<Car,Long>为例,spring data rest将会暴露两个接口

/cars和/cars/{id}

前者返回资源列表,后者返回某一个具体的资源

## 基础路径及定制方法

默认情况下Spring data rest 会以/为所暴露资源的基础路径

在spring boot项目中可以通过spring.data.rest.basePath=/api属性来手动指定基础路径

在常规spring项目中,你需要注册一个RepositoryRestConfigurer实例来自定义基础路径

## Collection Resource

集合资源

spring data rest会为Repository暴露一个Collection Resource,路径定义规则是以Repository所管理的domain的类名的小写复数形式命名,该资源路径可以通过@RepositoryRestResource注解指定

支持GET和POST请求

GET请求会返回资源列表

POST请求会依据请求体添加一个资源

## Item Resource

spring data rest会为Repository暴露一系列Item Resource作为Collection Resource的子资源

路径规则为domain的类名的小写复数/{id}

|  |
| --- |
| 405 Method Not Allowed - if the save(…) methods was not exported(through @RestResource(exported = false)) or is not present in the repository at all. |

支持GET、PUT和DELETE请求

GET请求会返回资源信息

PUT请求会修改指定的资源

DELETE请求会删除指定的资源

## association resource

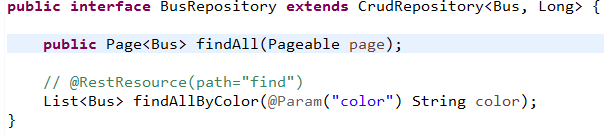
|  |
| --- |
| Spring Data REST exposes sub-resources of every item resource for each of the associations the item resource has. The name and path of the of the resource defaults to the name of the association property and can be customized using @RestResource on the association property. |

如果一个资源下有子资源，那么子资源的路径就是属性名

## search resource

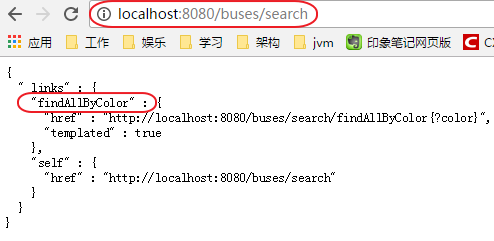
该功能是为了发现某个资源下的所有可用操作,只支持GET请求

例如有一个资源BusRepository<Bus,Long>,



可以通过下面的路径发现该资源的所有可用的query method resources

http://localhost8080/buses/search



## query method resources

|  |
| --- |
| The query method resource executes the query exposed through an individual query method on the repository interface. |

只支持GET请求

## 注解

### @RestResource

在Repository子接口上或其方法上使用,用来定制暴露的路径等信息

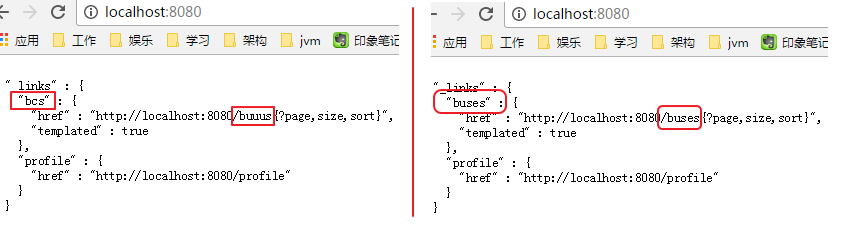
### @RepositoryRestResource

在spring data rest2.0以后用来替代@RestResource

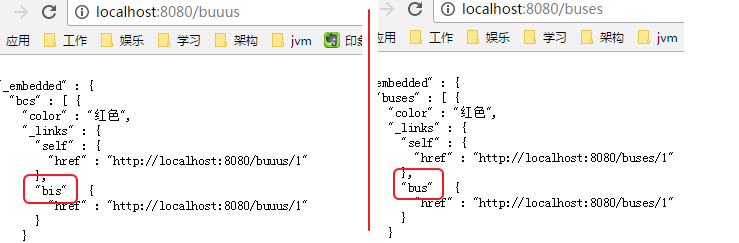
**@RepositoryRestResource对前台展示的影响**

|  |
| --- |
| @RepositoryRestResource(path = "buuus", collectionResourceRel = "bcs", itemResourceRel = "bis")  **public** **interface** BusRepository **extends** CrudRepository<Bus, Long> {} |

图左侧为使用@RepositoryRestResource,右侧为不使用



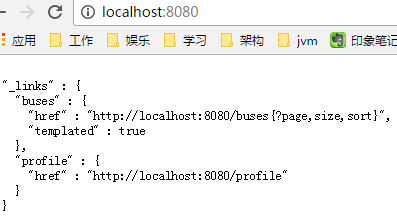
=========



## tips

通过访问spring data rest项目的根目录可以获取所有被暴露的资源的链接

例如:访问http://localhost:8080将会得到下面的结果



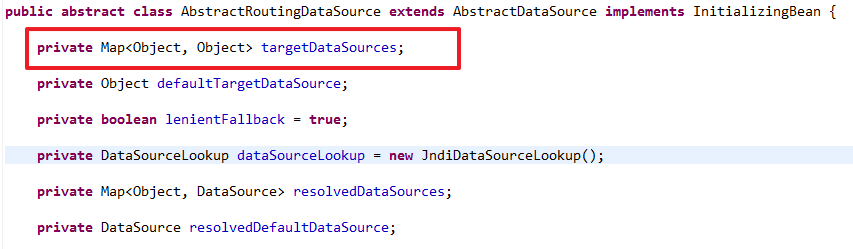
# DataSource

## AbstractRoutingDataSource

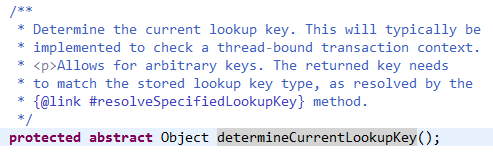
该类可以实现分库分表

spring支持动态切换连接池,通过AbstractRoutingDataSource实现

下图中红圈内的变量用于存放数据源和key之间的映射关系



下图中的方法是决定用哪个key去查找已经定义的数据源



**应用思路:**

1. 定义一个类,在其中声明一个成员变量ThreadLocal keyContainer
2. 定义一个切面,拦截所有Service方法,在通知中通过方法名决定获取对应(对应规则可以自己定义)的key,然后放到上一步定义的keyContainer中
3. 定义AbstractRoutingDataSource实现类,实现determineCurrentLookupKey方法

在determineCurrentLookupKey方法中从keyContainer中获取当前线程要使用的key