SPRING容器与SPRINGMVC容器的区别与联系

在spring整体框架的核心概念中，容器的核心思想是管理Bean的整个生命周期。但在一个项目中，Spring容器往往不止一个，最常见的场景就是在一个项目中引入Spring和SpringMVC这两个框架，其本质就是两个容器：Spring是根容器，SpringMVC是其子容器。关于这两个容器的创建、联系及区别也正是本文所关注的问题。

一、引子

Spring和SpringMVC作为Bean管理容器和MVC层的默认框架，已被众多web应用采用。但是在实际应用中，初级开发者常常会因对Spring和SpringMVC的配置失当导致一些奇怪的异常现象，比如Controller的方法无法拦截、Bean被多次加载等问题，这种情况发生的根本原因在于开发者对Spring容器和SpringMVC容器之间的关系了解不够深入，这也正是本文要阐述的问题。

二、Spring容器、SpringMVC容器与ServletContext之间的关系

在Web容器中配置Spring时，你可能已经司空见惯于web.xml文件中的以下配置代码，下面我们以该代码片段为基础来了解Spring容器、SpringMVC容器与ServletContext之间的关系。要想理解这三者的关系，需要先熟悉Spring是怎样在web容器中启动起来的。Spring的启动过程其实就是其Spring IOC容器的启动过程。特别地，对于web程序而言，IOC容器启动过程即是建立上下文的过程。

```
<web-app>

    ...

    <!-- 利用Spring提供的ContextLoaderListener监听器去监听ServletContext对象的创建，并初始化WebApplicationContext对象 -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <!-- Context Configuration locations for Spring XML files(默认查找/WEB-INF/applicationContext.xml) -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>

    <!-- 配置Spring MVC的前端控制器: DispatchcerServlet -->
    <servlet>
        <servlet-name>SpringMVC</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>SpringMVC</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    ...

</web-app>
```

1、Spring的启动过程

(1). 对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境；

(2). 在web.xml中会提供有contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用。在这个方法中，spring会初始化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext。WebApplicationContext是一个接口类，确切的说，其实际的实现类是XmlWebApplicationContext，它就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的<context-param>标签指定。在这个IoC容器初始化完毕后，Spring以WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE为属性Key，将其存储到ServletContext中，便于获取；

(3). ContextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，以最常见的DispatcherServlet为例，这个servlet实际上是一个标准的前端控制器，用以转发、匹配、处理每个servlet请求。DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文，用以持有spring mvc相关的bean。特别地，在建立DispatcherServlet自己的IoC上下文前，会利用WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE先从ServletContext中获取之前的根上下文(即WebApplicationContext)作为自己上下文的parent上下文。有了这个parent上下文之后，再初始化自己持有的上下文。这个DispatcherServlet初始化自己上下文的工作在其initStrategies方法中可以看到，大概的工作就是初始化处理器映射、视图解析等。这个servlet自己持有的上下文默认实现类也是mlWebApplicationContext。初始化完毕后，spring以与servlet的名字相关(此处不是简单的以servlet名为Key，而是通过一些转换，具体可自行查看源码)的属性为属性Key，也将其存到ServletContext中，以便后续使用。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文(第2步中初始化的上下文)定义的那些bean。

2、Spring容器与SpringMVC的容器联系与区别

ContextLoaderListener中创建Spring容器主要用于整个Web应用程序需要共享的一些组件，比如DAO、数据库的ConnectionFactory等；而由DispatcherServlet创建的SpringMVC的容器主要用于和该Servlet相关的一些组件，比如Controller、ViewResovler等。它们之间的关系如下：

(1). 作用范围

子容器(SpringMVC容器)可以访问父容器(Spring容器)的Bean，父容器(Spring容器)不能访问子容器(SpringMVC容器)的Bean。也就是说，当在SpringMVC容器中getBean时，如果在自己的容器中找不到对应的bean，则会去父容器中去找，这也解释了为什么由SpringMVC容器创建的Controller可以获取到Spring容器创建的Service组件的原因。

(2). 具体实现

在Spring的具体实现上，子容器和父容器都是通过ServletContext的setAttribute方法放到ServletContext中的。但是，ContextLoaderListener会先于DispatcherServlet创建ApplicationContext，DispatcherServlet在创建ApplicationContext时会先找到由ContextLoaderListener所创建的ApplicationContext，再将后者的ApplicationContext作为参数传给DispatcherServlet的ApplicationContext的setParent()方法。也就是说，子容器的创建依赖于父容器的创建，父容器先于子容器创建。在Spring源代码中，你可以在FrameServlet.Java中找到如下代码：

```
wac.setParent(parent);
```

其中，wac即为由DisptcherServlet创建的ApplicationContext，而parent则为有ContextLoaderListener创建的ApplicationContext。
此后，框架又会调用ServletContext的setAttribute()方法将wac加入到ServletContext中。

三、Spring容器和SpringMVC容器的配置

在Spring整体框架的核心概念中，容器是核心思想，就是用来管理Bean的整个生命周期的，而在一个项目中，容器不一定只有一个，Spring中可以包括多个容器，而且容器间有上下层关系，目前最常见的一种场景就是在一个项目中引入Spring和SpringMVC这两个框架，其实就是两个容器：Spring是根容器，SpringMVC是其子容器。在上文中，我们提到，SpringMVC容器可以访问Spring容器中的Bean，Spring容器不能访问SpringMVC容器的Bean。但是，若开发者对Spring容器和SpringMVC容器之间的关系了解不够深入，常常会因配置失当而导致同时配置Spring和SpringMVC时出现一些奇怪的异常，比如Controller的方法无法拦截、Bean被多次加载等问题。

在实际工程中，一个项目中会包括很多配置，根据不同的业务模块来划分，我们一般思路是各负其责，明确边界，即：Spring根容器负责所有其他非controller的Bean的注册，而SpringMVC只负责controller相关的Bean的注册，下面我们演示这种配置方案。

---

(1). Spring容器配置

Spring根容器负责所有其他非controller的Bean的注册：

```
<!-- 启用注解扫描，并定义组件查找规则 ，除了@controller，扫描所有的Bean -->
    <context:component-scan base-package="cn.edu.tju.rico">
        <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Controller" />
    </context:component-scan>
```

(2). SpringMVC容器配置

SpringMVC只负责controller相关的Bean的注册，其中@ControllerAdvice用于对控制器进行增强，常用于实现全局的异常处理类：

```
<!-- 启用注解扫描，并定义组件查找规则 ，mvc层只负责扫描@Controller、@ControllerAdvice -->
    <!-- base-package 如果多个，用“,”分隔 -->
    <context:component-scan base-package="cn.edu.tju.rico"
        use-default-filters="false">
        <!-- 扫描@Controller -->
        <context:include-filter type="annotation"
            expression="org.springframework.stereotype.Controller" />
        <!--控制器增强，使一个Contoller成为全局的异常处理类，类中用@ExceptionHandler方法注解的方法可以处理所有Controller发生的异常 -->
```

```
        <context:include-filter type="annotation"
            expression="org.springframework.web.bind.annotation.ControllerAdvice" />
    </context:component-scan>
```

在<context:component-scan>中可以添加use-default-filters，Spring配置中的use-default-filters用来指示是否自动扫描带有@Component、@Repository、@Service和@Controller的类。默认为true，即默认扫描。如果想要过滤其中这四个注解中的一个，比如@Repository，可以添加<context:exclude-filter />子标签，如下：

```
   <context:component-scan base-package="cn.edu.tju.rico" scoped-proxy="targetClass" use-default-filters="true">
    <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

而<context:include-filter/>子标签是用来添加扫描注解的：

```
<context:component-scan base-package="cn.edu.tju.rico" scoped-proxy="targetClass" use-default-filters="false">
    <context:include-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

四、Spring容器和SpringMVC容器的配置失当带来的问题

1、问题描述

在一个项目中，想使用Spring AOP在Controller中切入一些逻辑，但发现不能切入到Controller的中，但可以切入到Service中。最初的配置情形如下：

1). Spring的配置文件application.xml包含了开启AOP自动代理、Service扫描配置以及Aspect的自动扫描配置，如下所示：

```
<aop:aspectj-autoproxy/>
<context:component-scan base-package="cn.edu.tju.rico">
```

2). Spring MVC的配置文件spring-mvc.xml主要内容是Controller层的自动扫描配置。

```
<context:component-scan base-package="cn.edu.tju.rico.controller" />
```

3). 增强代码为如下:

```
@Component
@Aspect
public class SecurityAspect {

    private static final String DEFAULT_TOKEN_NAME = "X-Token";

    private TokenManager tokenManager;

    @Resource(name = "tokenManager")
    public void setTokenManager(TokenManager tokenManager) {
        this.tokenManager = tokenManager;
    }

    @Around("@annotation(org.springframework.web.bind.annotation.RequestMapping)")
    public Object execute(ProceedingJoinPoint pjp) throws Throwable {
        // 从切点上获取目标方法
        MethodSignature methodSignature = (MethodSignature) pjp.getSignature();
        Method method = methodSignature.getMethod();
        // 若目标方法忽略了安全性检查，则直接调用目标方法
        if (method.isAnnotationPresent(IgnoreSecurity.class)) {
            System.out
                    .println("method.isAnnotationPresent(IgnoreSecurity.class) : "
                            + method.isAnnotationPresent(IgnoreSecurity.class));
            return pjp.proceed();
        }
        // 从 request header 中获取当前 token
```

```
        String token = WebContext.getRequest().getHeader(DEFAULT_TOKEN_NAME);
        // 检查 token 有效性
        if (!tokenManager.checkToken(token)) {
            String message = String.format("token [%s] is invalid", token);
            throw new TokenException(message);
        }
        // 调用目标方法
        return pjp.proceed();
    }
}
```

4). 需要被代理的Controller如下:

```
@RestController
@RequestMapping("/tokens")
public class TokenController {

    private UserService userService;
    private TokenManager tokenManager;

    public UserService getUserService() {
        return userService;
    }

    @Resource(name = "userService")
    public void setUserService(UserService userService) {
        this.userService = userService;
    }

    public TokenManager getTokenManager() {
        return tokenManager;
    }

    @Resource(name = "tokenManager")
    public void setTokenManager(TokenManager tokenManager) {
        this.tokenManager = tokenManager;
    }

    @RequestMapping(method = RequestMethod.POST)
    @IgnoreSecurity
    public Response login(@RequestParam("uname") String uname,
            @RequestParam("passwd") String passwd) {
        boolean flag = userService.login(uname, passwd);
        if (flag) {
            String token = tokenManager.createToken(uname);
            System.out.println("**** Token **** : " + token);
            return new Response().success("Login Success...");
        }
        return new Response().failure("Login Failure...");
    }

    @RequestMapping(method = RequestMethod.DELETE)
    @IgnoreSecurity
    public Response logout(@RequestParam("uname") String uname) {
        tokenManager.deleteToken(uname);
        return new Response().success("Logout Success...");
    }
}
```

在运行过程中，发现这样配置并没有起作用，AOP配置不生效，没有生成TokenController的代理。

2、解决方案

由上一节可知，Spring容器先于SpringMVC容器进行创建，并且SpringMVC容器的创建依赖于Spring容器。在SpringMVC容器创建TokenController时，由于其没有启用AOP代理，并且父容器的配置与子容器配置的独立性，导致SpringMVC容器没有为TokenController生成代理，所以没有生效。我们只需要在SpringMVC的配置文件中添加Aspect的自动扫描配置即可：

```
<aop:aspectj-autoproxy/>
<context:component-scan base-package="com.hodc.sdk.controller" />
```

分类: spring

和碗说再见
关注 - 1
粉丝 - 4
+加关注

« 上一篇： 理解java的三大特性之多态
» 下一篇： Spring的事务配置详解

发表于 2018-05-11 15:25 和碗说再见 阅读(1863) 评论(1) 编辑 收藏

评论

#1楼
lz写的很清楚，赞！
不知道springboot中，算不算是统一成一个spring容器了？