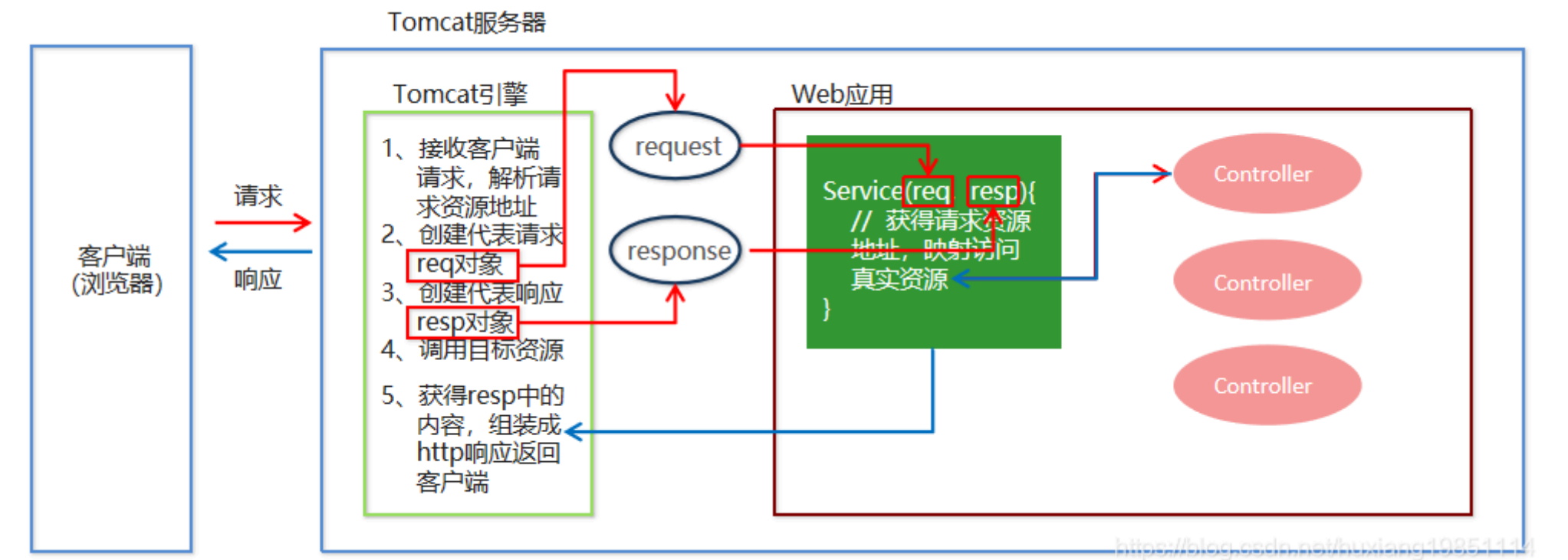


SpringMVC源码分析

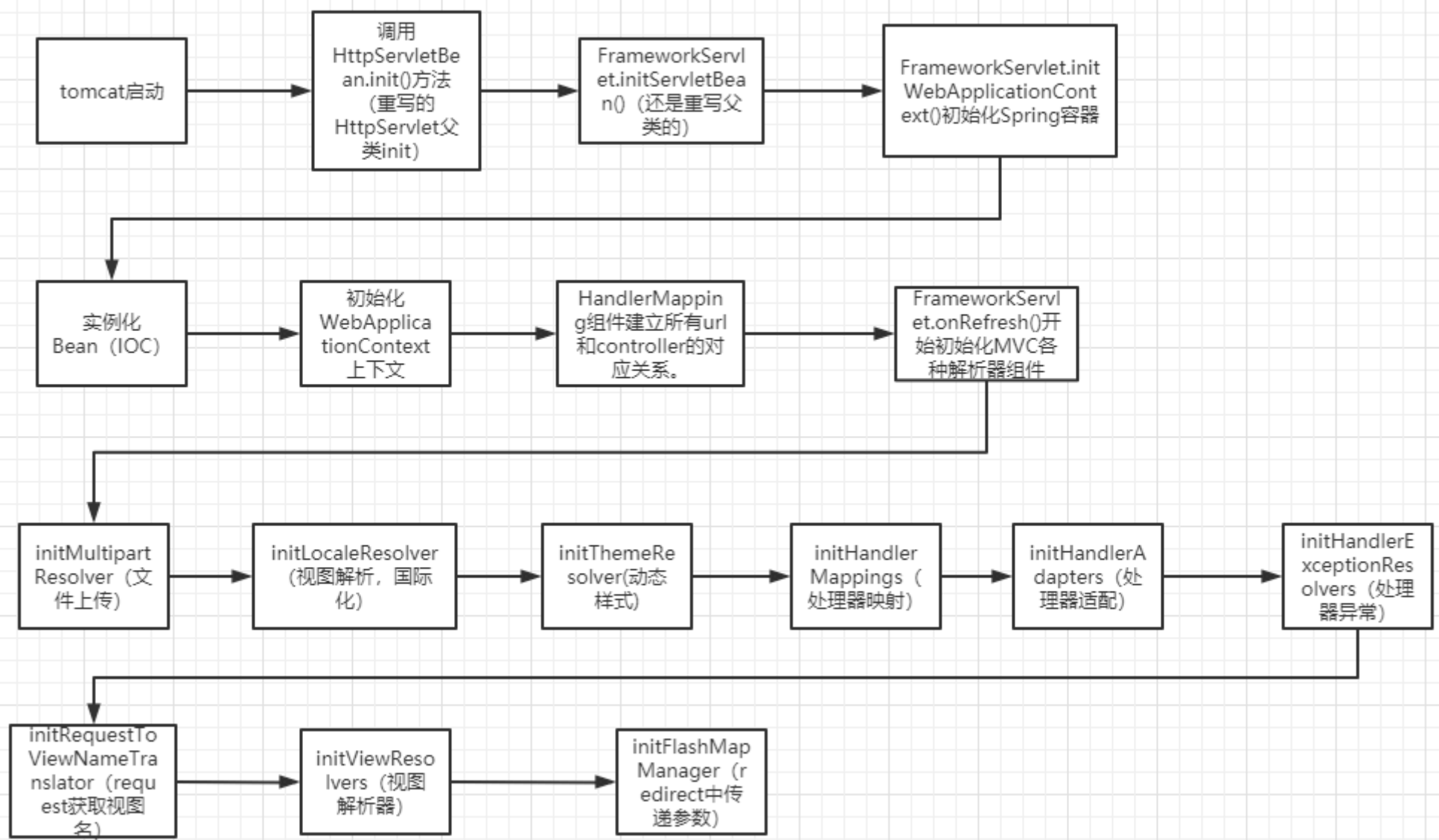
目录

- 1、SpringMVC架构图
- 2、SpringMVC初始化流程图
- 3、SpringMVC执行流程图
- 4、SpringMVC组件解析
- 5、SpringMVC的工作机制
- 6、源码分析
 - 6.1 初始化流程
 - 6.2 请求执行流程
- 7、谈谈SpringMVC的优化

1、SpringMVC 架构图

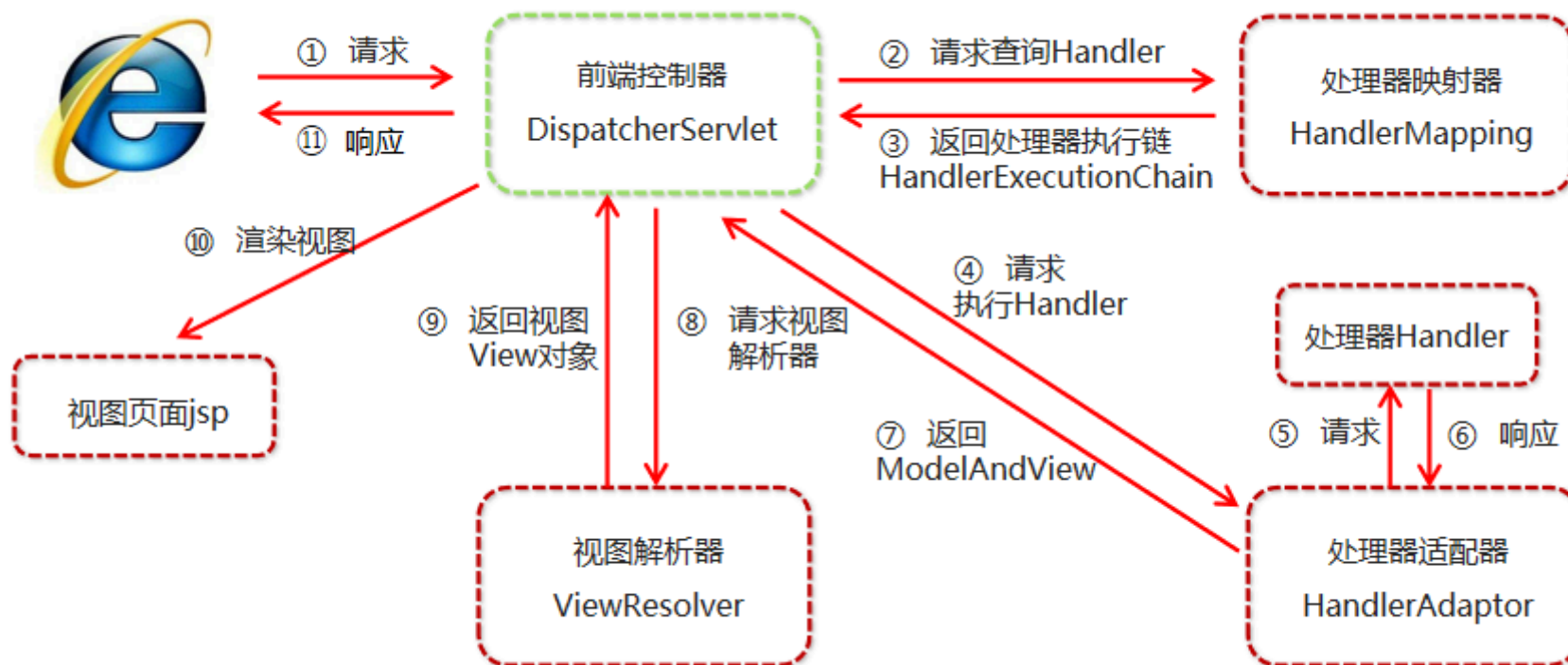


2、SpringMVC初始化流程图



<https://blog.csdn.net/huxiang19851114>

3、SpringMVC执行流程图



<https://blog.csdn.net/huxiang19851114>

- 用户发送请求至前端控制器DispatcherServlet。
- DispatcherServlet收到请求调用HandlerMapping处理器映射器。
- 处理器映射器找到具体的处理器(可以根据xml配置、注解进行查找), 生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
- DispatcherServlet调用HandlerAdapter处理器适配器。
- HandlerAdapter经过适配调用具体的处理器(Controller, 也叫后端控制器)。
- Controller执行完成返回ModelAndView。
- HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。
- DispatcherServlet将ModelAndView传给ViewReslover视图解析器。
- ViewReslover解析后返回具体View。

- DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。
- DispatcherServlet响应用户。

4、SpringMVC组件解析

前端控制器：DispatcherServlet 用户请求到达前端控制器，它就相当于 MVC 模式中的 C，DispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，DispatcherServlet 的存在降低了组件之间的耦合性。

处理器映射器：HandlerMapping HandlerMapping负责根据用户请求找到Handler处理器，SpringMVC提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

处理器适配器：HandlerAdapter 通过 HandlerAdapter对处理器进行执行适配，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。因为SpringMVC中的Handler可以是任意的形式，只要能处理请求就ok，但是 **Servlet** 需要的处理方法的结构却是固定的，都是以request和response为参数的方法。如何让固定的Servlet处理方法调用灵活的Handler来进行处理呢？这就是HandlerAdapter要做的事情。

处理器：Handler 它就是我们开发中要编写的具体业务处理接口。由 DispatcherServlet把用户请求转发到 Handler。由Handler对具体的用户请求进行处理。Handler的概念，也就是处理器。它直接应对着MVC中的C也就是Controller层，它的具体表现形式有很多，可以是类，也可以是方法。在Controller层中@RequestMapping标注的所有方法都可以看成是一个Handler，只要可以实际处理请求就可以是Handler。

视图解析器：View Resolver View Resolver负责将处理结果生成View 视图，View Resolver首先根据逻辑视图名解析成物理视图名，即具体的页面地址，再生成 View视图对象（包括响应信息等），最后对View进行渲染将处理结果通过页面展示给用户。

视图：View SpringMVC 框架提供了很多的 View 视图类型的支持，包括：JstlView、FreeMarkerView等。最常用的视图就是JSP。一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由程序员根据业务需求开发具体的页面

5、SpringMVC的工作机制

在容器初始化时会建立所有url和Controller的对应关系，保存到Map<url,Controller>中。Tomcat启动时会通知Spring初始化容器(加载Bean的定义信息和初始化所有单例Bean)，然后SpringMVC会遍历容器中的Bean，获取每一个Controller中的所有方法访问的url,然后将url和Controller保存到一个Map中；

这样就可以根据Request快速定位到Controller，因为最终处理Request的是Controller中的方法，Map中只保留了url和Controller中的对应关系，所以要根据Request的url进一步确认Controller中的Method，这一步工作的原理就是拼接Controller的url(Controller上@RequestMapping的值)和方法的url(method上@RequestMapping的值)，与Request的url进行匹配,找到匹配的那个方法；

确定处理请求的Method后，接下来的任务就是参数绑定，把Request中参数绑定到方法的形式参数上，这一步是整个请求处理过程中最复杂的一个步骤。SpringMVC提供了两种Request参数与方法形参的绑定方法：

- ① 通过注解进行绑定 @RequestParam
- ② 通过参数名称进行绑定.

使用注解进行绑定，我们只要在方法参数前面声明@RequestParam("a")，就可以将Request中参数“a”的值绑定到方法的该参数上。使用参数名称进行绑定的前提是必须要获取方法中参数的名称，Java反射只提供了获取方法的参数的类型，并没有提供获取参数名称的方法。SpringMVC解决这个问题的方法是用asm框架读取字节码文件，来获取方法的参数名称。asm框架是一个字节码操作框架，关于asm更多介绍可以参考它的官网。个人建议，使用注解来完成参数绑定，这样就可以省去asm框架的读取字节码的操作，这也算性能提升吧。

6、源码 分析

6.1 初始化流程

6.1.1 配置加载分类

Spring Web容器监听器（可能在有些项目里面service和controller分开了）

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext03.xml</param-value>
</context-param>
```

```

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

```

先加载ContextLoaderListener监听,然后调用了父类的模板方法initWebApplicationContext():

```

/*****ContextLoaderListener.java*****/
public void contextInitialized(ServletContextEvent event) {
    initWebApplicationContext(event.getServletContext());
}

/*****ContextLoader.java*****/
public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
    //.....略

    try {
        //创建web上下文，以保证它在ServletContext关闭时可用
        if (this.context == null) {
            this.context = createWebApplicationContext(servletContext);
        }
        if (this.context instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;
            if (!cwac.isActive()) { //isActive()在刷新容器时变为true,在prepareRefresh()方法里，不是刷新完才设置
                // The context has not yet been refreshed -> provide services such as
                // setting the parent context, setting the application context id, etc
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit parent ->
                    // determine parent for root web application context, if any.
                    ApplicationContext parent = loadParentContext(servletContext);
                    cwac.setParent(parent);
                }
                //配置了<context-param>和监听时，先加载IOC配置
                configureAndRefreshWebApplicationContext(cwac, servletContext);
            }
        }
        //将上下文设置到servlet容器中，当加载spring-mvc文件的时候直接使用即可
        servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);

        //.....略
    }

    protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {
        //.....略
        //其实这个地方就是加载Spring容器配置
        wac.refresh();
    }
}

```

Spring MVC前端控制器

```

<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

DispatcherServlet是MVC的心脏，继承了FrameworkServlet，FrameworkServlet又继承了HttpServletBean，HttpServletBean又继承了HttpServlet并且实现了Servlet的init方法。从顶层往子类往下看，一系列的模板方法!（父类调用，子类实现）

```
/******HttpServletBean.java******/
/**
 * 重写了父类HttpServlet的init方法，进行配置文件的读取和MVC的初始化
 */
public final void init() throws ServletException {

    //获取配置文件信息，准备设置Bean的属性对象PropertyValue
    PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
    if (!pvs.isEmpty()) {
        try {
            BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
            ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
            bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
            initBeanWrapper(bw);
            //有点长，直接跟进去到BeanWrapper包装接口实现类BeanWrapperImpl中看setValue
            //将配置文件设置到上下文中，以备后面解析
            bw.setPropertyValues(pvs, true);
        }
        catch (BeansException ex) {
            if (logger.isDebugEnabled()) {
                logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);
            }
            throw ex;
        }
    }

    // 进入正题：初始化ServletBean
    initServletBean();
}
```

FrameworkServlet继承了HttpServletBean，又重写了initServletBean方法：

```
/******FrameworkServlet.java******/
protected final void initServletBean() throws ServletException {
    //.....略

    try {
        //重点在这里，初始化web 容器上下文信息
        this.webApplicationContext = initWebApplicationContext();
        //空壳方法，可用于拓展
        initFrameworkServlet();
    }
    //.....略
}

/******FrameworkServlet.java******/
protected WebApplicationContext initWebApplicationContext() {
    //获取Spring容器初始化时候的上下文信息
    WebApplicationContext rootContext =
        WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;

    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        wac = this.webApplicationContext;
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
            if (!cwac.isActive()) {
                // The context has not yet been refreshed -> provide services such as
                // setting the parent context, setting the application context id, etc
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit parent -> set
                    // the root application context (if any; may be null) as the parent
                    cwac.setParent(rootContext);
                }
            }
        }
    }
}
```

```

        }
        //这个地方避免自定义上下文监听没有进行IOC容器初始化，再一次初始化
        configureAndRefreshWebApplicationContext(cwac);
    }
}
}
if (wac == null) {
    //判断是否已经初始化过了web容器上下文
    wac = findWebApplicationContext();
}
if (wac == null) {
    // 开始初始化web容器上下文，过程跟IOC一样，调用了AbstractApplicationContext refresh方法
    //然后在后置处理器中进行url和controller的绑定
    //传入rootContext，在此基础上继续加载MVC资源
    wac = createWebApplicationContext(rootContext);
}

if (!this.refreshEventReceived) {
    synchronized (this.onRefreshMonitor) {
        //开始初始化MVC九大组件
        onRefresh(wac);
    }
}

if (this.publishContext) {
    // Publish the context as a servlet context attribute.
    String attrName = getServletContextAttributeName();
    getServletContext().setAttribute(attrName, wac);
}

return wac;
}

```

6.1.2 加载SpringMVC容器

```

/*****FrameworkServlet.java*****/
protected WebApplicationContext createWebApplicationContext(@Nullable ApplicationContext parent) {
    Class<?> contextClass = getContextClass();
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
        throw new ApplicationContextException(
            "Fatal initialization error in servlet with name '" + getServletName() +
            "': custom WebApplicationContext class [" + contextClass.getName() +
            "] is not of type ConfigurableWebApplicationContext");
    }
    ConfigurableWebApplicationContext wac =
        (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);

    wac.setEnvironment(getEnvironment());
    //将IOC容器作为Web容器的父容器
    wac.setParent(parent);
    //获取到web配置信息
    String configLocation = getContextConfigLocation();
    if (configLocation != null) {
        wac.setConfigLocation(configLocation);
    }
    //调用IOC容器初始化流程，实例化并依赖注入，初始化及回调后置处理器
    configureAndRefreshWebApplicationContext(wac);

    return wac;
}

```

```

/*****FrameworkServlet.java*****/
protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac) {
    if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
        // The application context id is still set to its original default value
        // -> assign a more useful id based on available information
    }
}

```



```

    if (this.contextId != null) {
        wac.setId(this.contextId);
    }
    else {
        // Generate default id...
        wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
            ObjectUtils.getDisplayString(getServletContext().getContextPath()) + '/' + getServletName());
    }
}
//把之前ContextLoaderListener加载的web容器信息设置到WebApplicationContext
wac.setServletContext(getServletContext());
wac.setServletConfig(getServletConfig());
wac.setNamespace(getNamespace());
//请记住这个地方，Spring在上下文中添加了一个监听事件SourceFilteringListener，当IOC初始化完成后通过广播事件触发该监听，进行MVC九大组件的初始化
wac.addApplicationListener(new SourceFilteringListener(wac, new ContextRefreshListener()));

// The wac environment's #initPropertySources will be called in any case when the context
// is refreshed; do it eagerly here to ensure servlet property sources are in place for
// use in any post-processing or initialization that occurs below prior to #refresh
ConfigurableEnvironment env = wac.getEnvironment();
if (env instanceof ConfigurableWebEnvironment) {
    ((ConfigurableWebEnvironment) env).initPropertySources(getServletContext(), getServletConfig());
}
//啥也没干
postProcessWebApplicationContext(wac);
applyInitializers(wac);
//接下来就是IOC的流程了
wac.refresh();
}

```

后面大部分操作其实跟IOC一样，我们只看几个特殊的地方：

1、Service注入到Controller



```

@Controller
public class IndexController {

    @Autowired
    private UserService userService;

    @RequestMapping("/index")
    public String index(){
        return "index";
    }
}

```

怎么注入进来的

<https://blog.csdn.net/huxiang19851114>

首先我们不能像之前IOC普通自动装配了，因为service和controller分属两个不同的上下文，万一service没有实例化，需要切换上下文-----》相当于切换了容器，这个不能违背的基本原则！但是这两个容器又一个特点：父子关系！

```
beanDefinitionNames = {ArrayList@4199} size = 24
> 0 = "indexController"
> 1 = "userController"
> 2 = "org.springframework.context.annotation.internalConfigurationAnnotationProcessor"
> 3 = "org.springframework.context.annotation.internalAutowiredAnnotationProcessor"
> 4 = "org.springframework.context.annotation.internalCommonAnnotationProcessor"
> 5 = "org.springframework.context.event.internalEventListenerProcessor"
> 6 = "org.springframework.context.event.internalEventListenerFactory"
> 7 = "org.springframework.web.servlet.view.InternalResourceViewResolver#0"
> 8 = "mvcContentNegotiationManager"
> 9 = "org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"
> 10 = "mvcCorsConfigurations"
> 11 = "org.springframework.format.support.FormattingConversionServiceFactoryBean#0"
> 12 = "org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"
> 13 = "mvcUriComponentsContributor"
> 14 = "org.springframework.web.servlet.handler.MappedInterceptor#0"
> 15 = "org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver#0"
> 16 = "org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandler#0"
> 17 = "org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver#0"
> 18 = "org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"
> 19 = "org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"
> 20 = "org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"
> 21 = "mvcHandlerMappingIntrospector"
> 22 = "org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler#0"
> 23 = "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping#0"
manualSingletonNames = {LinkedHashSet@4200} size = 7
```

MVC属性注入使用了后置处理器，像我们这个例子，使用的Autowired注解，所以对象的后置处理器类为：
AutowiredAnnotationBeanPostProcessor

```

/*****AutowiredAnnotationBeanPostProcessor.java*****/
public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String beanName) {
    //根据bean name, bean对象全限定类名解析出来属性, 封装成一个元数据对象, 包括: 目标对象类, 注入元素, 检查元素
    InjectionMetadata metadata = findAutowiringMetadata(beanName, bean.getClass(), pvs);
    try {
        //通过元素据的方式注入, 参数: bean实例, bean名称, 属性列表(其实没用到)
        metadata.inject(bean, beanName, pvs);
    }
    catch (BeanCreationException ex) {
        throw ex;
    }
    catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Injection of autowired dependencies failed", ex);
    }
    return pvs;
}

/*****InjectionMetadata.java*****/
public void inject(Object target, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {
    Collection<InjectedElement> checkedElements = this.checkedElements;
    Collection<InjectedElement> elementsToIterate =
        (checkedElements != null ? checkedElements : this.injectedElements);
    if (!elementsToIterate.isEmpty()) {
        for (InjectedElement element : elementsToIterate) {
            if (logger.isTraceEnabled()) {
                logger.trace("Processing injected element of bean '" + beanName + "': " + element);
            }
            //一堆判断后拿到注入元素对象(userService属性)
            element.inject(target, beanName, pvs);
        }
    }
}

/*****AutowiredAnnotationBeanPostProcessor.java*****/
protected void injectme(Object bean, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {
    //得到属性对象: private com.ydt.spring03.service.UserService
    com.ydt.spring03.controller.IndexController.userService

```



```

Field field = (Field) this.member;
Object value;
if (this.cached) {
    value = resolvedCachedArgument(beanName, this.cachedFieldValue);
}
else {
    //定义一个依赖描述对象，存储依赖属性field和是否必须依赖
    DependencyDescriptor desc = new DependencyDescriptor(field, this.required);
    //设置被依赖对象controller全限定类名
    desc.setContainingClass(bean.getClass());
    Set<String> autowiredBeanNames = new LinkedHashSet<>(1);
    Assert.state(beanFactory != null, "No BeanFactory available");
    TypeConverter typeConverter = beanFactory.getTypeConverter();
    try {
        //通过parenBeanFactory获取到之前IOC容器初始化的userServiceImpl实体对象
        value = beanFactory.resolveDependency(desc, beanName, autowiredBeanNames, typeConverter);
    }
    catch (BeansException ex) {
        throw new UnsatisfiedDependencyException(null, beanName, new InjectionPoint(field), ex);
    }
    synchronized (this) {
        if (!this.cached) {
            if (value != null || this.required) {
                this.cachedFieldValue = desc;
                registerDependentBeans(beanName, autowiredBeanNames);
                if (autowiredBeanNames.size() == 1) {
                    String autowiredBeanName = autowiredBeanNames.iterator().next();
                    if (beanFactory.containsBean(autowiredBeanName) &&
                        beanFactory.isTypeMatch(autowiredBeanName, field.getType())) {
                        //将注入的属性对象userServiceImpl缓存起来
                        this.cachedFieldValue = new ShortcutDependencyDescriptor(
                            desc, autowiredBeanName, field.getType());
                    }
                }
            }
            else {
                this.cachedFieldValue = null;
            }
            this.cached = true;
        }
    }
}
if (value != null) {
    ReflectionUtils.makeAccessible(field);
    //反射设置属性值，完成service 到control的注入
    field.set(bean, value);
}
}
}

```

属性注入调后置处理器代码非常复杂，最后是调到了[AbstractBeanFactory#getType](#)方法里，去获取属性对应的bean，如果在当前容器没获取到bean,就从父容器获取。debug也耐心，一步一步看调用结果，不然都不知道在哪个方法调到了[AbstractBeanFactory#getType](#)


```

    }
}
else {
    delegate.parseCustomElement(root);
}
}
}

```

通过NamespaceHandler策略模式，找到NamespaceHandler下接口实现MvcNamespaceHandler

```

/*****MvcNamespaceHandler.java*****/
public void init() {
    //相同的套路，非默认Spring标签就根据命名空间来找： *NamespaceHandler
    registerBeanDefinitionParser("annotation-driven", new AnnotationDrivenBeanDefinitionParser());
    //.....
}

```

进入AnnotationDrivenBeanDefinitionParser解析器，找到解析方法：

```

/*****AnnotationDrivenBeanDefinitionParser.java*****/
public BeanDefinition parse(Element element, ParserContext context) {
    //.....
    //处理器映射器 RequestMappingHandlerMapping Bean Definition
    RootBeanDefinition handlerMappingDef = new RootBeanDefinition(RequestMappingHandlerMapping.class);
    handlerMappingDef.setSource(source);
    handlerMappingDef.setRole(BeaDefinition.ROLE_INFRASTRUCTURE);
    handlerMappingDef.getPropertyValues().add("order", 0);
    handlerMappingDef.getPropertyValues().add("contentNegotiationManager", contentNegotiationManager);

    if (element.hasAttribute("enable-matrix-variables")) {
        Boolean enableMatrixVariables = Boolean.valueOf(element.getAttribute("enable-matrix-variables"));
        handlerMappingDef.getPropertyValues().add("removeSemicolonContent", !enableMatrixVariables);
    }

    configurePathMatchingProperties(handlerMappingDef, element, context);
    readerContext.getRegistry().registerBeanDefinition(HANDLER_MAPPING_BEAN_NAME, handlerMappingDef);

    RuntimeBeanReference corsRef = MvcNamespaceUtils.registerCorsConfigurations(null, context, source);
    handlerMappingDef.getPropertyValues().add("corsConfigurations", corsRef);

    RuntimeBeanReference conversionService = getConversionService(element, source, context);
    RuntimeBeanReference validator = getValidator(element, source, context);
    RuntimeBeanReference messageCodesResolver = getMessageCodesResolver(element);
    //绑定Bean Definition，用于适配器处理映射关系查找
    RootBeanDefinition bindingDef = new RootBeanDefinition(ConfigurableWebBindingInitializer.class);
    bindingDef.setSource(source);
    bindingDef.setRole(BeaDefinition.ROLE_INFRASTRUCTURE);
    bindingDef.getPropertyValues().add("conversionService", conversionService);
    bindingDef.getPropertyValues().add("validator", validator);
    bindingDef.getPropertyValues().add("messageCodesResolver", messageCodesResolver);

    //.....
    //处理器适配器Bean Definition
    RootBeanDefinition handlerAdapterDef = new RootBeanDefinition(RequestMappingHandlerAdapter.class);
    handlerAdapterDef.setSource(source);
    handlerAdapterDef.setRole(BeaDefinition.ROLE_INFRASTRUCTURE);
    handlerAdapterDef.getPropertyValues().add("contentNegotiationManager", contentNegotiationManager);
    handlerAdapterDef.getPropertyValues().add("webBindingInitializer", bindingDef);
    handlerAdapterDef.getPropertyValues().add("messageConverters", messageConverters);
    addRequestBodyAdvice(handlerAdapterDef);
    addResponseBodyAdvice(handlerAdapterDef);

    if (element.hasAttribute("ignore-default-model-on-redirect")) {
        Boolean ignoreDefaultModel = Boolean.valueOf(element.getAttribute("ignore-default-model-on-redirect"));
        handlerAdapterDef.getPropertyValues().add("ignoreDefaultModelOnRedirect", ignoreDefaultModel);
    }
    if (argumentResolvers != null) {

```

```

        handlerAdapterDef.getPropertyValues().add("customArgumentResolvers", argumentResolvers);
    }
    if (returnValueHandlers != null) {
        handlerAdapterDef.getPropertyValues().add("customReturnValueHandlers", returnValueHandlers);
    }
    if (asyncTimeout != null) {
        handlerAdapterDef.getPropertyValues().add("asyncRequestTimeout", asyncTimeout);
    }
    if (asyncExecutor != null) {
        handlerAdapterDef.getPropertyValues().add("taskExecutor", asyncExecutor);
    }

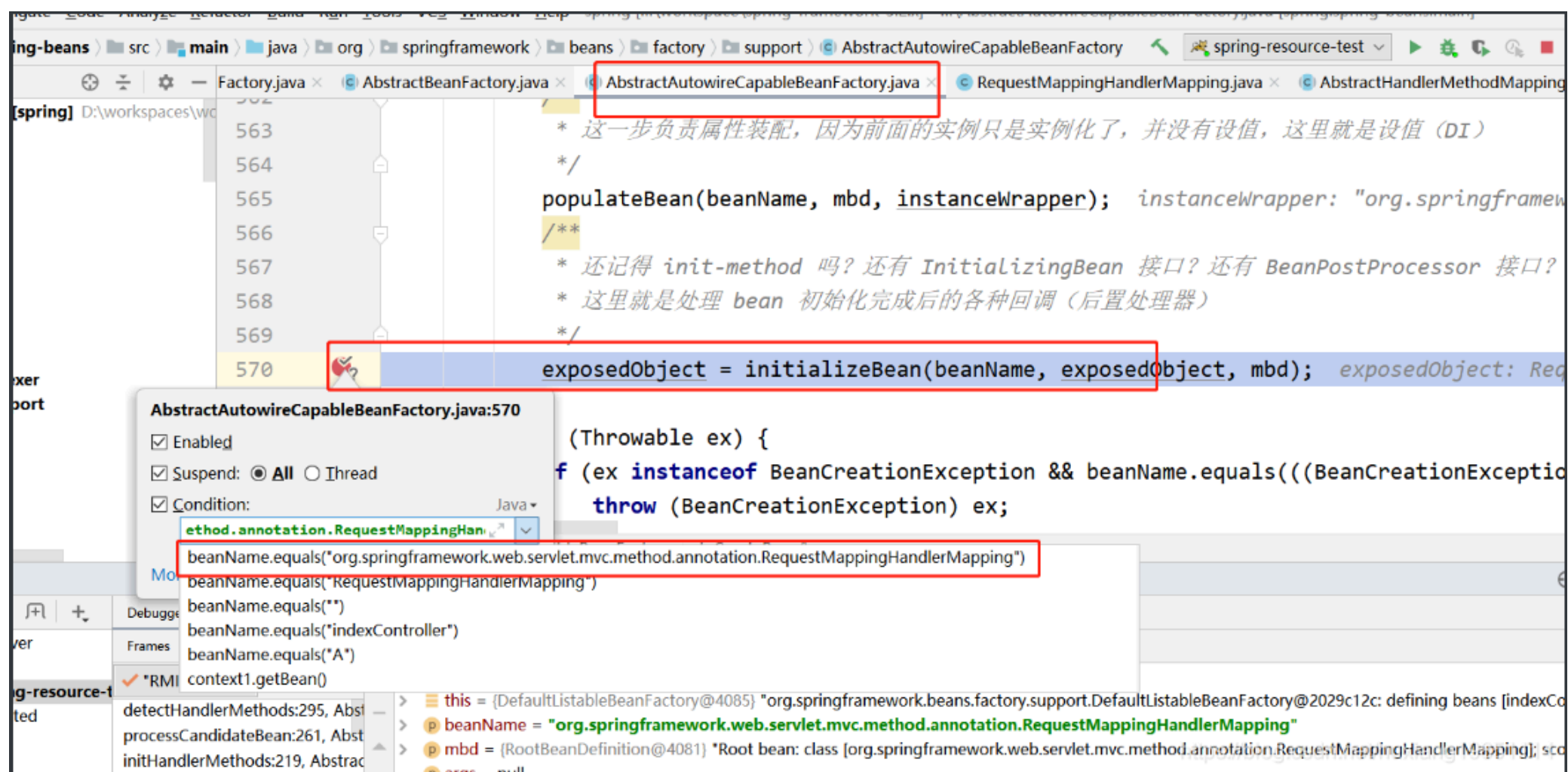
    handlerAdapterDef.getPropertyValues().add("callableInterceptors", callableInterceptors);
    handlerAdapterDef.getPropertyValues().add("deferredResultInterceptors", deferredResultInterceptors);
    //.....

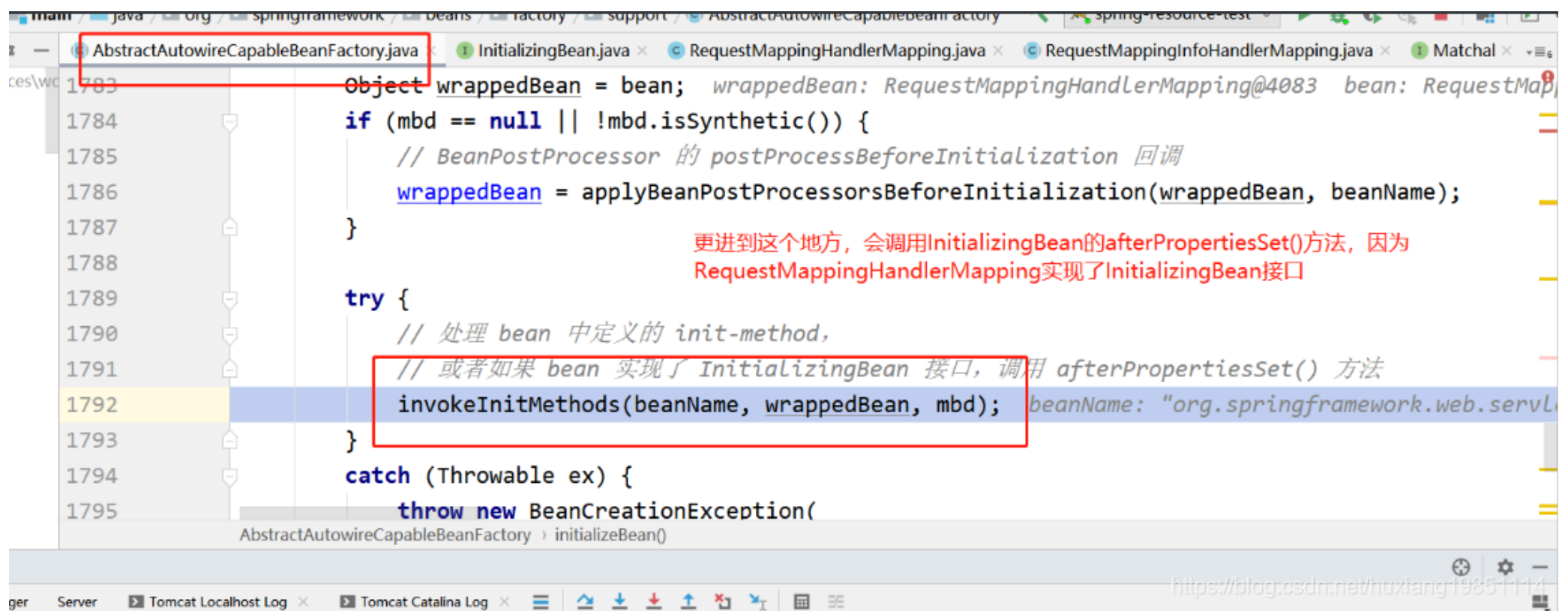
    context.registerComponent(new BeanComponentDefinition(handlerMappingDef, HANDLER_MAPPING_BEAN_NAME));
    context.registerComponent(new BeanComponentDefinition(handlerAdapterDef, HANDLER_ADAPTER_BEAN_NAME));
    //.....
    // Ensure BeanNameUrlHandlerMapping (SPR-8289) and default HandlerAdapters are not "turned off"
    //注册MVC组件（处理器映射器，处理器适配器）
    MvcNamespaceUtils.registerDefaultComponents(context, source);
    context.popAndRegisterContainingComponent();
    return null;
}

```

接下来的事情我们都很清楚了，无非是根据Bean Definition实例化Bean，属性设置，后置处理器回调

而我们对于HandlerMapping，HandlerAdapter处理就在后置处理器中，我们以RequestMappingHandlerMapping为例，将断点打在此处：





一直跟进到具体回调方法：

```

/*****AbstractHandlerMethodMapping.java*****/
//重写的afterPropertiesSet方法
public void afterPropertiesSet() {
    initHandlerMethods();
}

protected void initHandlerMethods() {
    //遍历所有的Bean Name，进行mapping映射设置
    for (String beanName : getCandidateBeanNames()) {
        if (!beanName.startsWith(SCOPED_TARGET_NAME_PREFIX)) {
            //跟进去瞅瞅
            processCandidateBean(beanName);
        }
    }
    handlerMethodsInitialized(getHandlerMethods());
}

protected void processCandidateBean(String beanName) {
    //.....
    if (beanType != null && isHandler(beanType)) {
        //继续跟进去
        detectHandlerMethods(beanName);
    }
}

protected void detectHandlerMethods(Object handler) {
    Class<?> handlerType = (handler instanceof String ?
        obtainApplicationContext().getType((String) handler) : handler.getClass());

    if (handlerType != null) {
        Class<?> userType = ClassUtils.getUserClass(handlerType);
        Map<Method, T> methods = MethodIntrospector.selectMethods(userType,
            (MethodIntrospector.MetadataLookup<T>) method -> {
                try {
                    return getMappingForMethod(method, userType);
                }
                catch (Throwable ex) {
                    throw new IllegalStateException("Invalid mapping on handler class [" +
                        userType.getName() + "]: " + method, ex);
                }
            });
        if (logger.isTraceEnabled()) {
            logger.trace(formatMappings(userType, methods));
        }
        methods.forEach((method, mapping) -> {
            Method invocableMethod = AopUtils.selectInvocableMethod(method, userType);
            //注册处理器方法，其实就是将往HandlerMapping中设置url和controller的映射

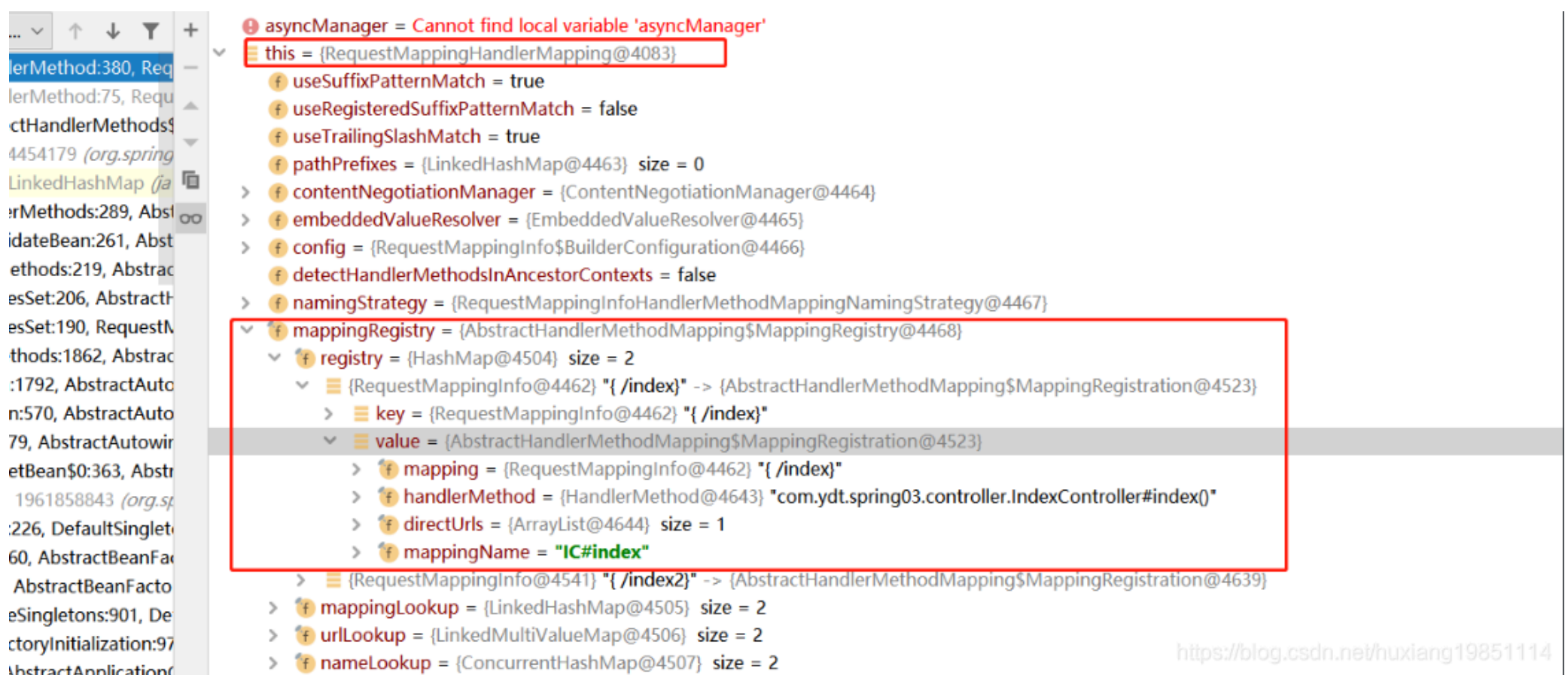
```



```

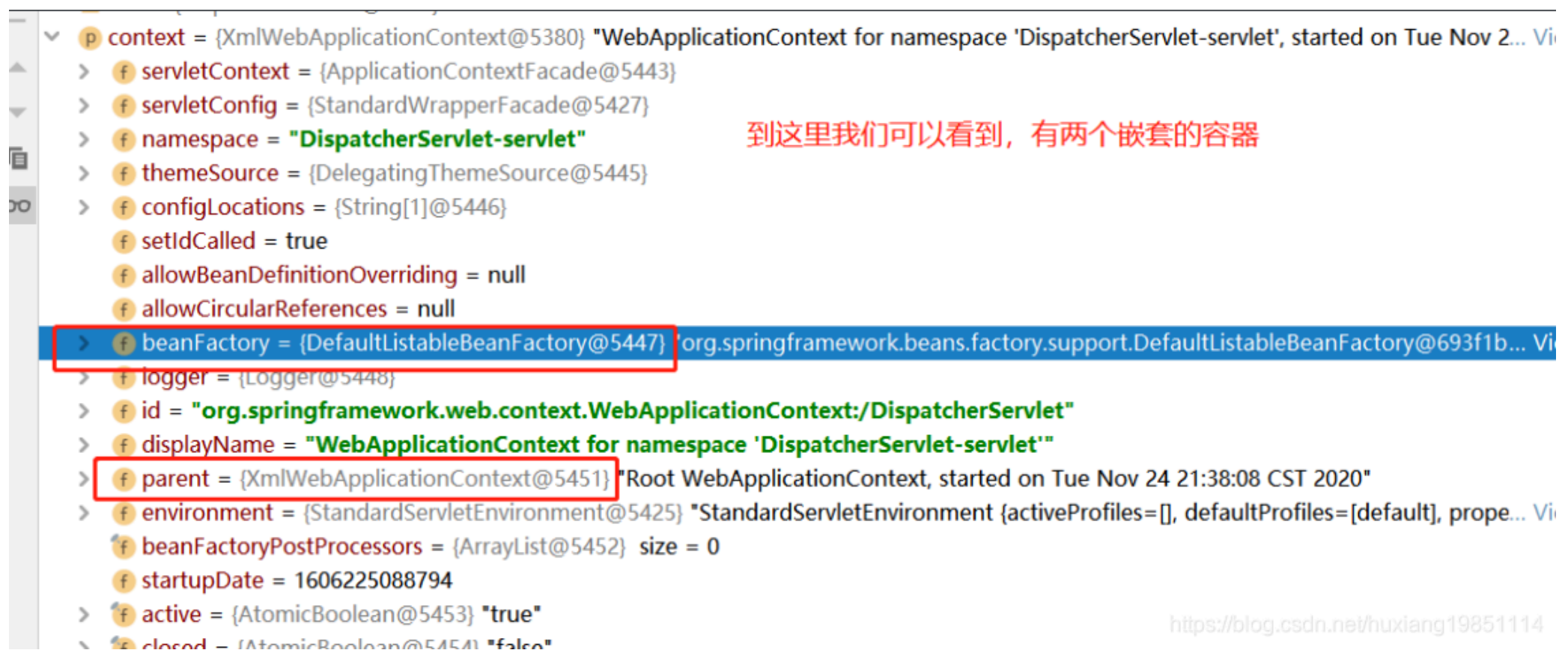
        registerHandlerMethod(handler, invocableMethod, mapping);
    });
}
}

```



<https://blog.csdn.net/huxiang19851114>

6.1.3 初始化MVC组件



<https://blog.csdn.net/huxiang19851114>

当IOC容器实例完成初始化并设置属性依赖后，回调用finishRefresh方法进行广播，执行publishEvent方法触发监听事件，还记得前面注册到Web容器中的SourceFilteringListener监听器吗？这里会触发onApplicationEvent方法：

```

/*****SourceFilteringListener.java*****/
public void onApplicationEvent(ApplicationEvent event) {
    if (event.getSource() == this.source) {
        onApplicationEventInternal(event);
    }
}

protected void onApplicationEventInternal(ApplicationEvent event) {
    if (this.delegate == null) {
        throw new IllegalStateException(
            "Must specify a delegate object or override the onApplicationEventInternal method");
    }
    //使用监听器适配器对象（因为有太多的监听类型）进行事件回调处理
    //---》请一直跟进到FrameworkServlet，会发现调用了其onRefresh方法进行MVC组件的初始化
}

```

```
        this.delegate.onApplicationEvent(event);
    }
}
```

DispatchServlet继承了FrameworkServlet，重写了onRefresh方法，跟到 initStrategies方法：

```
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context); //文件上传
    initLocaleResolver(context); //国际化
    initThemeResolver(context); //动态样式
    initHandlerMappings(context); //处理器映射器
    initHandlerAdapters(context); //处理器适配器
    initHandlerExceptionResolvers(context); //处理器异常解析器
    initRequestToViewNameTranslator(context); //请求视图名转换器
    initViewResolvers(context); //视图解析器
    initFlashMapManager(context); //每一个FlashMap保存着一套Redirect转发所传递的参数
}
}
```

文件上传解析器

```
private void initMultipartResolver(ApplicationContext context) {
    try {
        //初始化获取文件上传解析器，其实就是一个Bean
        /*<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">*/
        this.multipartResolver = context.getBean(MULTIPART_RESOLVER_BEAN_NAME, MultipartResolver.class);
        //*****
    }
}
```

国际化解析器

```
private void initLocaleResolver(ApplicationContext context) {
    try {
        //国际化解析器，其实还是一个Bean
        /*<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
        this.localeResolver = context.getBean(LOCALE_RESOLVER_BEAN_NAME, LocaleResolver.class);
        //*****
    }
}
```

动态样式模板解析器

```
private void initThemeResolver(ApplicationContext context) {
    try {
        //动态样式模板解析器，还是一个Bean
        /*<bean id="themeResolver" class="org.springframework.web.servlet.theme.SessionThemeResolver"/>
        this.themeResolver = context.getBean(THEME_RESOLVER_BEAN_NAME, ThemeResolver.class);
        //*****
    }
}
```

处理器映射器

```
private void initHandlerMappings(ApplicationContext context) {
    this.handlerMappings = null;

    if (this.detectAllHandlerMappings) { //处理所有的处理器映射
        // Find all HandlerMappings in the ApplicationContext, including ancestor contexts.
        // 查找ApplicationContext中的所有HandlerMapping
        Map<String, HandlerMapping> matchingBeans =
            BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);
        //拿到注册的Map<url,controller>处理器映射集合，存入handlerMappings集合
        if (!matchingBeans.isEmpty()) {
            //处理器映射器放的东西包括（control对象或者control方法和url的映射，还有过滤器调用链）
            this.handlerMappings = new ArrayList<>(matchingBeans.values());
            // 按照优先级排序。
            AnnotationAwareOrderComparator.sort(this.handlerMappings);
        }
    }
}
```

```

    }
    //.....
}

```

处理器适配器

```

private void initHandlerAdapters(ApplicationContext context) {
    this.handlerAdapters = null;

    if (this.detectAllHandlerAdapters) {
        // Find all HandlerAdapters in the ApplicationContext, including ancestor contexts.
        Map<String, HandlerAdapter> matchingBeans =
            BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerAdapter.class, true, false);
        if (!matchingBeans.isEmpty()) {
            this.handlerAdapters = new ArrayList<>(matchingBeans.values());
            // We keep HandlerAdapters in sorted order.
            AnnotationAwareOrderComparator.sort(this.handlerAdapters);
        }
    }
    //.....
}

```

其他的就不一一列举了，所谓的初始化MVC控件只是将这些控件对象放入前端控制器的组件属性中，基本上都是集合类型，这些控件在容器初始化的时候都已经通过后置处理器完成了，他们都实现了InitializingBean接口

6.2 请求执行流程

6.2.1 请求入口

FrameworkServlet重写了父类HttpServletBean的doGet，doPost方法

```

/*****FrameworkServlet.java*****/
protected final void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    //.....
    try {
        //集中访问入口
        doService(request, response);
    }
    //.....
}

```

DispatchServlet又重写了父类doService方法:

```

/*****DispatchServlet.java*****/
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
    //.....
    try {
        //集中分发访问入口
        doDispatch(request, response);
    }
    //.....
}

```

6.2.2 请求匹配映射分发

这也是整个MVC核心方法：

```

/*****DispatchServlet.java*****/
/** 中央控制器,控制请求的转发 */
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HttpServletRequest processedRequest = request;
    HandlerExecutionChain mappedHandler = null;
    boolean multipartRequestParsed = false;

    WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

    try {
        ModelAndView mv = null;
        Exception dispatchException = null;

        try {
            // 1.检查是否是文件上传的请求
            processedRequest = checkMultipart(request);
            multipartRequestParsed = (processedRequest != request);

            // 2.取得处理当前请求的controller,这里也称为handler,处理器,第一个步骤的意义就在这里体现了.
            // 这里并不是直接返回controller,而是返回的HandlerExecutionChain请求处理器链对象,该对象封装了handler和
interceptors.

            mappedHandler = getHandler(processedRequest);
            // 如果handler为空,则返回404
            if (mappedHandler == null) {
                noHandlerFound(processedRequest, response);
                return;
            }

            //3. 获取处理request的处理器适配器handler adapter
            HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

            // Process last-modified header, if supported by the handler.
            String method = request.getMethod();
            boolean isGet = "GET".equals(method);
            if (isGet || "HEAD".equals(method)) {
                long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {
                    return;
                }
            }
            // 4.拦截器的预处理方法
            if (!mappedHandler.applyPreHandle(processedRequest, response)) {
                return;
            }

            // 5.实际的处理器处理请求,返回结果视图对象
            mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

            if (asyncManager.isConcurrentHandlingStarted()) {
                return;
            }
            // 结果视图对象的处理
            applyDefaultViewName(processedRequest, mv);
            // 6.拦截器的后处理方法
            mappedHandler.applyPostHandle(processedRequest, response, mv);
        }
        catch (Exception ex) {
            dispatchException = ex;
        }
        catch (Throwable err) {
            // As of 4.3, we're processing Errors thrown from handler methods as well,
            // making them available for @ExceptionHandler methods and other scenarios.
            dispatchException = new NestedServletException("Handler dispatch failed", err);
        }
        // 请求成功响应之后的方法
        processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
    }
}
```

```

    }
    catch (Exception ex) {
        triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
    }
    catch (Throwable err) {
        triggerAfterCompletion(processedRequest, response, mappedHandler,
            new NestedServletException("Handler processing failed", err));
    }
    finally {
        if (asyncManager.isConcurrentHandlingStarted()) {
            // Instead of postHandle and afterCompletion
            if (mappedHandler != null) {
                mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
            }
        }
        else {
            // Clean up any resources used by a multipart request.
            if (multipartRequestParsed) {
                cleanupMultipart(processedRequest);
            }
        }
    }
}
}

```

getHandler(processedRequest)方法实际上就是从HandlerMapping中找到url和controller的对应关系。这也就是之前在初始化MVC组件建立Map<url,Controller>的意义。我们知道，最终处理request的是controller中的方法，我们现在只是知道了controller，还要进一步确认controller中处理request的方法。

6.2.3 获取Handler和拦截器链

```

/*****DispatchServlet.java*****/
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        /*处理器映射器（包含handler和intercept）
        包含三种：
            RequestMappingHandlerMapping 最常见模式
            BeanNameUrlHandlerMapping 以 "/" 开头的 bean <bean id="/sayByeBye.do" class="com.ydt.spring03.controller.OrderController">
</bean>
            SimpleUrlHandlerMapping :<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                <property name="mappings">
                    <props>
                        <prop key="sayByeBye2.do">orderController</prop>
                    </props>
                </property>
            </bean>

        */
        for (HandlerMapping mapping : this.handlerMappings) {
            HandlerExecutionChain handler = mapping.getHandler(request);
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}

public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    //先根据请求获取对应的handler
    Object handler = getHandlerInternal(request);
    if (handler == null) {
        handler = getDefaultHandler();
    }
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?

```



```

        if (handler instanceof String) {
            String handlerName = (String) handler;
            handler = obtainApplicationContext().getBean(handlerName);
        }
        //获取处理器执行链（handler，intercept）
        HandlerExecutionChain executionChain = getHandlerExecutionChain(handler, request);

        //.....
    }

/*****RequestMappingInfoHandlerMapping.java*****/
protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws Exception {
    request.removeAttribute(PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE);
    try {
        //调用父类AbstractHandlerMethodMapping模板方法
        return super.getHandlerInternal(request);
    }
    finally {
        ProducesRequestCondition.clearMediaTypesAttribute(request);
    }
}

/*****AbstractHandlerMethodMapping.java*****/
protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws Exception {
    //根据request解析得到请求路径：/index.do
    String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);
    request.setAttribute(LOOKUP_PATH, lookupPath);
    this.mappingRegistry.acquireReadLock();
    try {
        //根据请求路径和request获取HandlerMethod对象，里面包括bean name，method，参数等
        HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath, request);
        //再一次通过bean name从容器中取出对应的controller对象
        return (handlerMethod != null ? handlerMethod.createWithResolvedBean() : null);
    }
    finally {
        this.mappingRegistry.releaseReadLock();
    }
}

protected HandlerMethod lookupHandlerMethod(String lookupPath, HttpServletRequest request) throws Exception {
    List<Match> matches = new ArrayList<>();
    List<T> directPathMatches = this.mappingRegistry.getMappingsByUrl(lookupPath);
    if (directPathMatches != null) {
        addMatchingMappings(directPathMatches, matches, request);
    }
    if (matches.isEmpty()) {
        //通过遍历所有的mapping key，获取到所有匹配/index.*的映射，添加到matches中
        addMatchingMappings(this.mappingRegistry.getMappings().keySet(), matches, request);
    }

    if (!matches.isEmpty()) {
        //只取第一个，这也是为什么我们写多个相同controller接口时，只有一个生效的原因
        Match bestMatch = matches.get(0);
        if (matches.size() > 1) {
            //这个地方就是对多个匹配的controller的情况下，给用户一个异常提示
        }
        request.setAttribute(BEST_MATCHING_HANDLER_ATTRIBUTE, bestMatch.handlerMethod);
        handleMatch(bestMatch.mapping, lookupPath, request);
        //返回HandlerMethod对象
        return bestMatch.handlerMethod;
    }
    else {
        return handleNoMatch(this.mappingRegistry.getMappings().keySet(), lookupPath, request);
    }
}

//获取执行链，这个地方主要是获取拦截器，也就是之前MVC组件初始化的时候的adaptedInterceptors，默认会有区域切换和类型转换的拦截器
protected HandlerExecutionChain getHandlerExecutionChain(Object handler, HttpServletRequest request) {

```

```

        //拿到handler（controller或者method）
        HandlerExecutionChain chain = (handler instanceof HandlerExecutionChain ?
            (HandlerExecutionChain) handler : new HandlerExecutionChain(handler));

        //访问路径
        String lookupPath = this.urlPathHelper.getLookupPathForRequest(request, LOOKUP_PATH);
        for (HandlerInterceptor interceptor : this.adaptedInterceptors) {
            //判断过滤器是否为mapping过滤器,如果是普通Bean过滤器，直接往执行链中扔
            if (interceptor instanceof MappedInterceptor) {
                MappedInterceptor mappedInterceptor = (MappedInterceptor) interceptor;
                //判断访问路径是否在过滤器中被包含或者被排除
                if (mappedInterceptor.matches(lookupPath, this.pathMatcher)) {
                    chain.addInterceptor(mappedInterceptor.getInterceptor());
                }
            }
            else {
                chain.addInterceptor(interceptor);
            }
        }
        return chain;
    }
}
/*（1）过滤器：

```

依赖于servlet容器。在实现上基于函数回调，可以对几乎所有请求进行过滤，但是缺点是一个过滤器实例只能在容器初始化时调用一次。使用过滤器的目的是用来做一些过滤操作，获取我们想要获取的数据，比如：在过滤器中修改字符编码；在过滤器中修改HttpServletRequest的一些参数，包括：过滤低俗文字、危险字符等

（2）拦截器：

依赖于web框架，在SpringMVC中就是依赖于SpringMVC框架。在实现上基于Java的反射机制，属于面向切面编程（AOP）的一种运用。由于拦截器是基于web框架的调用，因此可以使用Spring的依赖注入（DI）进行一些业务操作，同时一个拦截器实例在一个controller生命周期之内可以多次调用。但是缺点是只能对controller请求进行拦截，对其他的一些比如直接访问静态资源的请求则没办法进行拦截处

6.2.4 获取请求处理适配器

Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替Controller执行相应的方法。这样在扩展Controller 时，只需要增加一个适配器类就完成了SpringMVC的扩展了

```

/*****DispatcherServlet.java*****/
protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {
    if (this.handlerAdapters != null) {
        /*根据handler（参数）获取处理器适配器
        包含三种：
            RequestMappingHandlerAdapter 注解处理器适配器（最常用）
            HttpRequestHandlerAdapter 支持handler是Servlet的适配模式,也就是实现了HttpRequestHandler的handler
            SimpleControllerHandlerAdapter 非注解处理器适配器,支持所有实现了 Controller 接口的 Handler 控制器
        */
        for (HandlerAdapter adapter : this.handlerAdapters) {
            if (adapter.supports(handler)) {
                return adapter;
            }
        }
    }
    //.....
}

```

6.2.5 MVC过滤器前置处理

```

boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = 0; i < interceptors.length; i++) {
            HandlerInterceptor interceptor = interceptors[i];
            //如果前置处理完成并返回true，触发执行最终处理
            if (!interceptor.preHandle(request, response, this.handler)) {
                triggerAfterCompletion(request, response, null);
                return false;
            }
            this.interceptorIndex = i;
        }
    }
}

```

```

    }
}
return true;
}

```

6.2.5 反射调用返回结果视图

根据url确定Controller中处理请求的方法，然后通过反射获取该方法上的注解和参数，解析方法和参数上的注解，最后反射调用方法获取ModelAndView结果视图。因为上面采用注解url形式说明的，所以我们这里继续以注解处理器适配器来说明。调用的就是HandlerAdapter的handle()，适配器子类RequestMappingHandlerAdapter.handle()中的核心逻辑由invokeHandlerMethod(request, response, handler)实现。

```

/*****RequestMappingHandlerAdapter.java*****/

```

```

protected ModelAndView handleInternal(HttpServletRequest request,
                                     HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    ModelAndView mav;
    checkRequest(request);

    // 涉及到Session的情况下对于handler的调用
    if (this.synchronizeOnSession) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
    }
    else {
        // No HttpSession available -> no mutex necessary
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }
}
else {
    // 进入我们的对应的适配器了，跟进去
    mav = invokeHandlerMethod(request, response, handlerMethod);
}

//.....
}

```

```

protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
                                     HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        //.....一大堆的设置：参数，返回视图，包装好的执行方法.....
        //核心调度方法，使用反射调用目标对象
        invocableMethod.invokeAndHandle(webRequest, mavContainer);
        if (asyncManager.isConcurrentHandlingStarted()) {
            return null;
        }
        //通过mavContainer进行视图处理
        return getModelAndView(mavContainer, modelFactory, webRequest);
    }
    finally {
        webRequest.requestCompleted();
    }
}

```

```

//无非是得到Model和View

```

```

private ModelAndView getModelAndView(ModelAndViewContainer mavContainer,
                                     ModelFactory modelFactory, NativeWebRequest
webRequest) throws Exception {

```

```

    modelFactory.updateModel(webRequest, mavContainer);

```

```

                                     ModelFactory modelFactory, NativeWebRequest

```

```

        if (mavContainer.isRequestHandled()) {
            return null;
        }
        ModelMap model = mavContainer.getModel();
        ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model, mavContainer.getStatus());
        if (!mavContainer.isViewReference()) {
            mav.setView((View) mavContainer.getView());
        }
        if (model instanceof RedirectAttributes) {
            Map<String, ?> flashAttributes = ((RedirectAttributes) model).getFlashAttributes();
            HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);
            if (request != null) {
                RequestContextUtils.getOutputFlashMap(request).putAll(flashAttributes);
            }
        }
        return mav;
    }

//中间是一堆的MVC边角料功能，我们跟着主线一直跟进到
//*****InvocableHandlerMethod.java*****/
protected Object doInvoke(Object... args) throws Exception {
    ReflectionUtils.makeAccessible(getBridgedMethod());
    try {
        //底层反射调用目标对象方法（这个地方使用了桥接方法，主要是兼容JDK1.5之前不支持泛型的问题）
        return getBridgedMethod().invoke(getBean(), args);
    }
    //.....
}

```

通过上面的代码，已经可以找到处理request的Controller中的方法了，现在看如何解析该方法上的参数,并调用该方法。也就是执行方法这一步。执行方法这一步最重要的就是获取方法的参数，然后我们就可以反射调用方法了。

SpringMVC中提供两种request参数到方法中参数的绑定方式：

① 通过注解进行绑定 @RequestParam

② 通过参数名称进行绑定 使用注解进行绑定，我们只要在方法参数前面声明@RequestParam("a")，就可以将request中参数a的值绑定到方法的该参数上。使用参数名称进行绑定的前提是必须要获取方法中参数的名称，Java反射只提供了获取方法的参数的类型，并没有提供获取参数名称的方法。SpringMVC解决这个问题的方法是用asm框架读取字节码文件,来获取方法的参数名称。个人建议，使用注解来完成参数绑定，这样就可以省去asm框架的读取字节码的操作。

```

//*****InvocableHandlerMethod.java*****/
protected Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer,
    Object... providedArgs) throws Exception {

    MethodParameter[] parameters = getMethodParameters();
    if (ObjectUtils.isEmpty(parameters)) {
        return EMPTY_ARGS;
    }

    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        MethodParameter parameter = parameters[i];
        parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
        args[i] = findProvidedArgument(parameter, providedArgs);
        if (args[i] != null) {
            continue;
        }
        if (!this.resolvers.supportsParameter(parameter)) {
            throw new IllegalStateException(formatArgumentError(parameter, "No suitable resolver"));
        }
        try {
            args[i] = this.resolvers.resolveArgument(parameter, mavContainer, request, this.dataBinderFactory);
        }
        catch (Exception ex) {
            // Leave stack trace for later, exception may actually be resolved and handled...
        }
    }
}

```

```

        if (logger.isDebugEnabled()) {
            String exMsg = ex.getMessage();
            if (exMsg != null && !exMsg.contains(parameter.getExecutable().toGenericString())) {
                logger.debug(formatArgumentError(parameter, exMsg));
            }
        }
        throw ex;
    }
}
return args;
}
}

```

到这里,方法的参数值列表也获取到了，就可以直接进行方法的调用了。整个请求过程中最复杂的一步就是在这里了。ok，到这里整个请求处理过程的关键步骤都分析完了。理解了SpringMVC中的请求处理流程,整个代码还是比较清晰的。

7、谈谈SpringMVC的优化

上面我们已经对SpringMVC的工作原理和源码进行了分析,在这个过程发现了几个优化点：

- 1、Controller如果能保持单例，尽量使用单例,这样可以减少创建对象和回收对象的开销。也就是说，如果Controller的类变量和实例变量可以以方法形参声明的尽量以方法的形参声明，不要以类变量和实例变量声明，这样可以避免线程安全问题。
- 2、处理request的方法中的形参务必加上@RequestParam注解，这样可以避免SpringMVC使用asm框架读取class文件获取方法参数名的过程。即便SpringMVC对读取出的方法参数名进行了缓存，如果不要读取class文件当然是更加好。
- 3、阅读源码的过程中,发现SpringMVC并没有对处理url的方法进行缓存，也就是说每次都要根据请求url去匹配Controller中的方法url，如果把url和方法的关系缓存起来，会不会带来性能上的提升呢？有点恶心的是，负责解析url和方法对应关系的ServletHandlerMethodResolver是一个private的内部类，不能直接继承该类增强代码，必须要改代码后重新编译。当然，如果缓存起来，必须要考虑缓存的线程安全问题。

父子容器详解

分类专栏：


Spring学习专栏

文章标签：

spring

rpc

java

 Spring学习专栏

专栏收录该内容

10 订阅 52 篇文章

订阅专栏

又一次被面试官带到坑里面了。

面试官：springmvc用过么？

我：用过啊，经常用呢

面试官：springmvc中为什么需要用父子容器？

我：嗯。。。没听明白你说的什么。

面试官：就是controller层交给一个spring容器加载，其他的service和dao层交给另外一个spring容器加载，web.xml中有这块配置，这两个容器组成了父子容器的关系。

我：哦，原来是这块啊，我想起来了，我看大家都这么用，所以我也这么用

面试官：有没有考虑过为什么？

我：我在网上看大家都这么用，所以我也这么用了，具体也不知道为什么，不过用起来还挺顺手的

面试官：如果只用一个容器可以么，所有的配置都交给一个spring容器加载？

我：应该不行吧！

面试官：确定不行么？

我：让我想一会。。。。。我感觉是可以的，也可以正常运行。

面试官：那我们又回到了开头的问题，为什么要用父子容器呢？

我：我叫你哥好么，别这么玩我了，被你绕晕了？

面试官：好吧，你回去试试看吧，下次再来告诉我，出门右转，不送！

我：脸色变绿了，灰头土脸的走了。

回去之后，我好好研究了一番，下次准备再去给面试官一点颜色看看。

主要的问题

- 1. 什么是父子容器？
- 2. 为什么需要用父子容器？
- 3. 父子容器如何使用？

下面我们就来探讨探讨。

我们先来看一个案例

系统中有2个模块：module1和module2，两个模块是独立开发的，module2会使用到module1中的一些类，module1会将自己打包为jar提供给module2使用，我们来看一下这2个模块的代码。

模块1

放在module1包中，有3个类

Service1

```
package com.javacode2018.lesson002.demo17.module1;

import org.springframework.stereotype.Component;
```

```
    @Component
public class Service1 {
    public String m1() {
        return "我是module1中的Service1中的m1方法";
    }
}
```

Service2

```
package com.javacode2018.lesson002.demo17.module1;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Service2 {

    @Autowired
    private com.javacode2018.lesson002.demo17.module1.Service1 service1; //@1

    public String m1() { //@2
        return this.service1.m1();
    }

}
```

上面2个类，都标注了@Component注解，会被spring注册到容器中。
@1: Service2中需要用到Service1，标注了@Autowired注解，会通过spring容器注入进来
@2: Service2中有个m1方法，内部会调用service的m1方法。

来个spring配置类：Module1Config

```
package com.javacode2018.lesson002.demo17.module1;

import org.springframework.context.annotation.ComponentScan;

@ComponentScan
public class Module1Config {
}
```

上面使用了@ComponentScan注解，会自动扫描当前类所在的包中的所有类，将标注有@Component注解的类注册到spring容器，即Service1和Service2会被注册到spring容器。

再来看模块2

放在module2包中，也是有3个类，和模块1中的有点类似。

Service1

模块2中也定义了一个Service1，内部提供了一个m2方法，如下：

```
package com.javacode2018.lesson002.demo17.module2;

import org.springframework.stereotype.Component;

@Component
public class Service1 {
    public String m2() {
        return "我是module2中的Service1中的m2方法";
    }
}
```

```
| }
```

Service3

```
package com.javacode2018.lesson002.demo17.module2;

import com.javacode2018.lesson002.demo17.module1.Service2;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Service3 {
    //使用模块2中的Service1
    @Autowired
    private com.javacode2018.lesson002.demo17.module2.Service1 service1; //@1
    //使用模块1中的Service2
    @Autowired
    private com.javacode2018.lesson002.demo17.module1.Service2 service2; //@2

    public String m1() {
        return this.service2.m1();
    }

    public String m2() {
        return this.service1.m2();
    }
}
```

@1: 使用module2中的Service1

@2: 使用module1中的Service2

先来思考一个问题

上面的这些类使用spring来操作会不会有问题？会有什么问题？

这个问题还是比较简单的，大部分人都可以看出来，会报错，因为两个模块中都有Service1，被注册到spring容器的时候，bean名称会冲突，导致注册失败。

来个测试类，看一下效果

```
package com.javacode2018.lesson002.demo17;

import com.javacode2018.lesson001.demo21.Config;
import com.javacode2018.lesson002.demo17.module1.Module1Config;
import com.javacode2018.lesson002.demo17.module2.Module2Config;
import org.junit.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class ParentFactoryTest {

    @Test
    public void test1() {
        //定义容器
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
        //注册bean
        context.register(Module1Config.class, Module2Config.class); //@1
        //启动容器
        context.refresh();
    }
}
```

@1: 将Module1Config、Module2Config注册到容器，spring内部会自动解析这两个类上面的注解，即：@ComponentScan注解，然后会进行包扫描，将标注了@Component的类注册到spring容器。

运行test1输出

下面是部分输出：

```
Caused by: org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'service1' for
```

service1这个bean的名称冲突了。

那么我们如何解决？

对module1中的Service1进行修改？这个估计是行不通的，module1是别人以jar的方式提供给我们的，源码我们是无法修改的。

而module2是我们自己的开发的，里面的东西我们可以随意调整，那么我们可以去修改一下module2中的Service1，可以修改一下类名，或者修改一下这个bean的名称，此时是可以解决问题的。

不过大家有没有想过一个问题：如果我们的模块中有很多类都出现了这种问题，此时我们一个个去重构，还是比较痛苦的，并且代码重构之后，还涉及到重新测试的问题，工作量也是蛮大的，这些都是风险。

而spring中的父子容器就可以很好的解决上面这种问题。

什么是父子容器

创建spring容器的时候，可以给当前容器指定一个父容器。

BeanFactory的方式

```
//创建父容器parentFactory
DefaultListableBeanFactory parentFactory = new DefaultListableBeanFactory();
//创建一个子容器childFactory
DefaultListableBeanFactory childFactory = new DefaultListableBeanFactory();
//调用setParentBeanFactory指定父容器
childFactory.setParentBeanFactory(parentFactory);
```

ApplicationContext的方式

```
//创建父容器
AnnotationConfigApplicationContext parentContext = new AnnotationConfigApplicationContext();
//启动父容器
parentContext.refresh();

//创建子容器
AnnotationConfigApplicationContext childContext = new AnnotationConfigApplicationContext();
//给子容器设置父容器
childContext.setParent(parentContext);
//启动子容器
childContext.refresh();
```

上面代码还是比较简单的，大家都可以看懂。

我们需要了解父子容器的特点，这些是比较关键的，如下。

父子容器特点

- 1. 父容器和子容器是相互隔离的，他们内部可以存在名称相同的bean
- 2. 子容器可以访问父容器中的bean，而父容器不能访问子容器中的bean
- 3. 调用子容器的getBean方法获取bean的时候，会沿着当前容器开始向上面的容器进行查找，直到找到对应的bean为止
- 4. 子容器中可以通过任何注入方式注入父容器中的bean，而父容器中是无法注入子容器中的bean，原因是第2点

使用父子容器解决开头的问题

关键代码

```
@Test
public void test2() {
    //创建父容器
    AnnotationConfigApplicationContext parentContext = new AnnotationConfigApplicationContext();
    //向父容器中注册Module1Config配置类
    parentContext.register(Module1Config.class);
    //启动父容器
    parentContext.refresh();

    //创建子容器
    AnnotationConfigApplicationContext childContext = new AnnotationConfigApplicationContext();
    //向子容器中注册Module2Config配置类
    childContext.register(Module2Config.class);
    //给子容器设置父容器
    childContext.setParent(parentContext);
    //启动子容器
    childContext.refresh();

    //从子容器中获取Service3
    Service3 service3 = childContext.getBean(Service3.class);
    System.out.println(service3.m1());
    System.out.println(service3.m2());
}
```

运行输出

```
我是module1中的Service1中的m1方法
我是module2中的Service1中的m2方法
```

这次正常了。

父子容器使用注意点

我们使用容器的过程中，经常会使用到的一些方法，这些方法通常会在下面的两个接口中

```
org.springframework.beans.factory.BeanFactory
org.springframework.beans.factory.ListableBeanFactory
```

这两个接口中有很多方法，这里就不列出来了，大家可以去看一下源码，这里要说的是使用父子容器的时候，有些需要注意的地方。

BeanFactory接口，是spring容器的顶层接口，这个接口中的方法是支持容器嵌套结构查找的，比如我们常用的getBean方法，就是这个接口中定义的，调用getBean方法的时候，会从沿着当前容器向上查找，直到找到满足条件的bean为止。

而ListableBeanFactory这个接口中的方法是不支持容器嵌套结构查找的，比如下面这个方法

```
String[] getBeanNamesForType(@Nullable Class<?> type)
```

获取指定类型的所有bean名称，调用这个方法的时候只会返回当前容器中符合条件的bean，而不会去递归查找其父容器中的bean。

来看一下案例代码，感受一下：

```
@Test
public void test3() {
    //创建父容器parentFactory
    DefaultListableBeanFactory parentFactory = new DefaultListableBeanFactory();
    //向父容器parentFactory注册一个bean[userNme->"路人甲Java"]
    parentFactory.registerBeanDefinition("userNme",
```



```
        BeanDefinitionBuilder.  
            genericBeanDefinition(String.class).  
                addConstructorArgValue("路人甲Java").  
                getBeanDefinition());  
  
        //创建一个子容器childFactory  
        DefaultListableBeanFactory childFactory = new DefaultListableBeanFactory();  
        //调用setParentBeanFactory指定父容器  
        childFactory.setParentBeanFactory(parentFactory);  
        //向子容器parentFactory注册一个bean[address->"上海"]  
        childFactory.registerBeanDefinition("address",  
            BeanDefinitionBuilder.  
                genericBeanDefinition(String.class).  
                addConstructorArgValue("上海").  
                getBeanDefinition());  
  
        System.out.println("获取bean【userName】： " + childFactory.getBean("userName"));/*@1  
  
        System.out.println(Arrays.asList(childFactory.getBeanNamesForType(String.class))); //@2  
    }
```

上面定义了2个容器

父容器：parentFactory，内部定义了一个String类型的bean：userName->路人甲Java

子容器：childFactory，内部也定义了一个String类型的bean：address->上海

@1：调用子容器的getBean方法，获取名称为userName的bean，userName这个bean是在父容器中定义的，而getBean方法是BeanFactory接口中定义的，支持容器层次查找，所以getBean是可以找到userName这个bean的

@2：调用子容器的getBeanNamesForType方法，获取所有String类型的bean名称，而getBeanNamesForType方法是ListableBeanFactory接口中定义的，这个接口中方法不支持层次查找，只会在当前容器中查找，所以这个方法只会返回子容器的address

我们来运行一下看看效果：

```
    获取bean【userName】：路人甲Java  
    [address]
```

结果和分析的一致。

那么问题来了：有没有方式解决ListableBeanFactory接口不支持层次查找的问题？

spring中有个工具类就是解决这个问题的，如下：

```
org.springframework.beans.factory.BeanFactoryUtils
```

这个类中提供了很多静态方法，有很多支持层次查找的方法，源码你们可以去细看一下，名称中包含有Ancestors的都是支持层次查找的。

在test2方法中加入下面的代码：

```
        //层次查找所有符合类型的bean名称  
        String[] beanNamesForTypeIncludingAncestors = BeanFactoryUtils.beanNamesForTypeIncludingAncestors(childFactory, String.class);  
        System.out.println(Arrays.asList(beanNamesForTypeIncludingAncestors));  
  
        Map<String, String> beansOfTypeIncludingAncestors = BeanFactoryUtils.beansOfTypeIncludingAncestors(childFactory, String.class);  
        System.out.println(Arrays.asList(beansOfTypeIncludingAncestors));
```

运行输出

```
    [address, userName]
```

查找过程是按照层次查找所有满足条件的bean。

回头看一下springmvc父子容器的问题

问题1：springmvc中只使用一个容器是否可以？

只使用一个容器是可以正常运行的。

问题2：那么springmvc中为什么需要用到父子容器？

通常我们使用springmvc的时候，采用3层结构，controller层，service层，dao层；父容器中会包含dao层和service层，而子容器中包含的只有controller层；这2个容器组成了父子容器的关系，controller层通常会注入service层的bean。

采用父子容器可以避免有些人在service层去注入controller层的bean，导致整个依赖层次是比较混乱的。

父容器和子容器的需求也是不一样的，比如父容器中需要有事务的支持，会注入一些支持事务的扩展组件，而子容器中controller完全用不到这些，对这些并不关心，子容器中需要注入一下springmvc相关的bean，而这些bean父容器中同样是不会用到的，也是不关心一些东西，将这些相互不关心的东西隔开，可以有效的避免一些不必要的错误，而父子容器加载的速度也会快一些。

总结

1. 本文需掌握父子容器的用法，了解父子容器的特点：子容器可以访问父容器中bean，父容器无法访问子容器中的bean
2. BeanFactory接口支持层次查找
3. ListableBeanFactory接口不支持层次查找
4. BeanFactoryUtils工具类中提供了一些非常实用的方法，比如支持bean层次查找的方法等等

案例源码

<https://gitee.com/javacode2018/spring-series>

路人甲java所有案例代码以后都会放到这个上面，大家watch一下，可以持续关注动态。