

09 | 分解Dispatcher：如何把专门的事情交给专门的部件去做？

2023-03-31 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。今天我们继续手写 MiniSpring。

经过上节课的工作，我们已经实现了 IoC 与 MVC 的结合，还定义了 Dispatcher 与 WebApplicationContext 两个相对独立又互相关联的结构。


这节课我们计划在已有的 ApplicationConfigWebApplicationContext 和 DispatcherServlet 基础上，把功能做进一步地分解，让 Dispatcher 只负责解析 request 请求，用 Context 专门用来管理各个 Bean。

两级 ApplicationContext

按照通行的 Web 分层体系，一个程序它在结构上会有 Controller 和 Service 两层。在我们的程序中，Controller 由 DispatcherServlet 负责启动，Service 由 Listener 负责启动。我们

计划把这两部分所对应的容器进行进一步地切割，拆分为 XmlWebApplicationContext 和 AnnotationConfigWebApplicationContext。

首先在 DispatcherServlet 这个类里，增加一个对 WebApplicationContext 的引用，命名为 parentApplicationContext。这样，当前这个类里就有了两个对 WebApplicationContext 的引用。

 复制代码

```
1 private WebApplicationContext webApplicationContext;  
2 private WebApplicationContext parentApplicationContext;
```

新增 parentApplicationContext 的目的是，把 Listener 启动的上下文和 DispatcherServlet 启动的上下文两者区分开来。按照时序关系，Listener 启动在前，对应的上下文我们把它叫作 parentApplicationContext。

我们调整一下 init() 方法。

 复制代码

```
1 public void init(ServletConfig config) throws ServletException {  
2     super.init(config);  
3     this.parentApplicationContext = (WebApplicationContext)  
4 this.getServletContext().getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION  
5 _CONTEXT_ATTRIBUTE);  
6     sContextConfigLocation =  
7 config.getInitParameter("contextConfigLocation");  
8  
9     URL xmlPath = null;  
10    try {  
11        xmlPath = this.getServletContext().getResource(sContextConfigLocation);  
12    } catch (MalformedURLException e) {  
13        e.printStackTrace();  
14    }  
15    this.packageNames = XmlScanComponentHelper.getNodeValue(xmlPath);  
16    this.webApplicationContext = new  
17 AnnotationConfigWebApplicationContext(sContextConfigLocation,  
18 this.parentApplicationContext);  
19    Refresh();  
20 }
```

初始化的时候先从 ServletContext 里拿属性

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, 得到的是前一步 Listener 存放在这里的那个 parentApplicationContext。然后通过 contextConfigLocation 配置文件, 创建一个新的 WebApplicationContext。

从上述代码, 我们可以发现, 里面构建了一个 AnnotationConfigWebApplicationContext 对象, 这个对象的构造函数需要两个参数, 一个是配置文件路径, 另一个是父上下文。但以前 AnnotationConfigWebApplicationContext 只有一个参数为 String 的构造函数。所以这里我们需要扩展改造一下, 把 DispatcherServlet 里一部分和扫描包相关的代码移到 AnnotationConfigWebApplicationContext 里。你可以看一下修改后的 AnnotationConfigWebApplicationContext 代码。

 复制代码

```
1 package com.minis.web;
2
3 import java.io.File;
4 import java.net.MalformedURLException;
5 import java.net.URL;
6 import java.util.ArrayList;
7 import java.util.List;
8 import javax.servlet.ServletContext;
9 import com.minis.beans.BeansException;
10 import com.minis.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor;
11 import com.minis.beans.factory.config.BeanDefinition;
12 import com.minis.beans.factory.config.BeanFactoryPostProcessor;
13 import com.minis.beans.factory.config.ConfigurableListableBeanFactory;
14 import com.minis.beans.factory.support.DefaultListableBeanFactory;
15 import com.minis.context.AbstractApplicationContext;
16 import com.minis.context.ApplicationEvent;
17 import com.minis.context.ApplicationEventPublisher;
18 import com.minis.context.ApplicationListener;
19 import com.minis.context.SimpleApplicationEventPublisher;
20
21 public class AnnotationConfigWebApplicationContext
22     extends AbstractApplicationContext implements WebApplicationContext{
23     private WebApplicationContext parentApplicationContext;
24     private ServletContext servletContext;
25     DefaultListableBeanFactory beanFactory;
26     private final List<BeanFactoryPostProcessor> beanFactoryPostProcessors =
27         new ArrayList<BeanFactoryPostProcessor>();
28
29     public AnnotationConfigWebApplicationContext(String fileName) {
```

```

30     this(fileName, null);
31 }
32 public AnnotationConfigWebApplicationContext(String fileName, WebApplicationCon
33     this.parentApplicationContext = parentApplicationContext;
34     this.servletContext = this.parentApplicationContext.getServletContext();
35     URL xmlPath = null;
36     try {
37         xmlPath = this.getServletContext().getResource(fileName);
38     } catch (MalformedURLException e) {
39         e.printStackTrace();
40     }
41
42     List<String> packageNames = XmlScanComponentHelper.getNodeValue(xmlPath);
43     List<String> controllerNames = scanPackages(packageNames);
44     DefaultListableBeanFactory bf = new DefaultListableBeanFactory();
45     this.beanFactory = bf;
46     this.beanFactory.setParent(this.parentApplicationContext.getBeanFactory())
47     loadBeanDefinitions(controllerNames);
48
49     if (true) {
50         try {
51             refresh();
52         } catch (Exception e) {
53             e.printStackTrace();
54         }
55     }
56 }
57 public void loadBeanDefinitions(List<String> controllerNames) {
58     for (String controller : controllerNames) {
59         String beanID=controller;
60         String beanClassName=controller;
61         BeanDefinition beanDefinition=new BeanDefinition(beanID,beanClassName
62         this.beanFactory.registerBeanDefinition(beanID,beanDefinition);
63     }
64 }
65 private List<String> scanPackages(List<String> packages) {
66     List<String> tempControllerNames = new ArrayList<>();
67     for (String packageName : packages) {
68         tempControllerNames.addAll(scanPackage(packageName));
69     }
70     return tempControllerNames;
71 }
72 private List<String> scanPackage(String packageName) {
73     List<String> tempControllerNames = new ArrayList<>();
74     URL url =this.getClass().getClassLoader().getResource("/"+packageName.re
75     File dir = new File(url.getFile());
76     for (File file : dir.listFiles()) {
77         if(file.isDirectory()){
78             scanPackage(packageName+"."+file.getName());


```

```

79         }else{
80             String controllerName = packageName + "." +file.getName().replace(
81                 tempControllerNames.add(controllerName);
82         }
83     }
84     return tempControllerNames;
85 }
86 public void setParent(WebApplicationContext parentApplicationContext) {
87     this.parentApplicationContext = parentApplicationContext;
88     this.beanFactory.setParent(this.parentApplicationContext.getBeanFactory());
89 }
90 public ServletContext getServletContext() {
91     return this.servletContext;
92 }
93 public void setServletContext(ServletContext servletContext) {
94     this.servletContext = servletContext;
95 }
96 public void publishEvent(ApplicationEvent event) {
97     this.getApplicationEventPublisher().publishEvent(event);
98 }
99 public void addApplicationListener(ApplicationListener listener) {
100     this.getApplicationEventPublisher().addApplicationListener(listener);
101 }
102 public void registerListeners() {
103     ApplicationListener listener = new ApplicationListener();
104     this.getApplicationEventPublisher().addApplicationListener(listener);
105 }
106 public void initApplicationEventPublisher() {
107     ApplicationEventPublisher aep = new SimpleApplicationEventPublisher();
108     this.setApplicationEventPublisher(aep);
109 }
110 public void postProcessBeanFactory(ConfigurableListableBeanFactory bf) {
111 }
112 public void registerBeanPostProcessors(ConfigurableListableBeanFactory bf) {
113     this.beanFactory.addBeanPostProcessor(new AutowiredAnnotationBeanPostProcesso
114 }
115 public void onRefresh() {
116     this.beanFactory.refresh();
117 }
118 public void finishRefresh() {
119 }
120 public ConfigurableListableBeanFactory getBeanFactory() throws IllegalStateException
121     return this.beanFactory;
122 }
123 }

```


这段代码的核心是扩充原有的构造方法。通过下面两行代码得到 `parentApplicationContext` 和 `servletContext` 的引用。

 复制代码

```
1  this.parentApplicationContext = parentApplicationContext;
2  this.servletContext = this.parentApplicationContext.getServletContext();
```


为了兼容原有构造方法，在只有 1 个参数的时候，给 `WebApplicationContext` 传入了一个 `null`。可以看到，修改后的 `AnnotationConfigWebApplicationContext` 继承自抽象类 `AbstractApplicationContext`，所以也具备了上下文的通用功能，例如注册监听器、发布事件等。

其次是改造 `DefaultListableBeanFactory`，因为 `AnnotationConfigWebApplicationContext` 里调用了 `DefaultListableBeanFactory` 的 `setParent` 方法，所以我们需要提供相应的实现方法，你可以看一下相关代码。

 复制代码

```
1  ConfigurableListableBeanFactory parentBeanFactory;
2
3  public void setParent(ConfigurableListableBeanFactory beanFactory) {
4      this.parentBeanFactory = beanFactory;
5  }
```

接下来我们还要改造 `XmlWebApplicationContext`，在继承 `ClassPathXmlApplicationContext` 的基础上实现 `WebApplicationContext` 接口，基本上我们可以参考 `AnnotationConfigWebApplicationContext` 来实现。

 复制代码


```
1  package com.minis.web;
2
3  import javax.servlet.ServletContext;
4  import com.minis.context.ClassPathXmlApplicationContext;
5
6  public class XmlWebApplicationContext
7      extends ClassPathXmlApplicationContext implements WebApplicationContext
```

```

8     private ServletContext servletContext;
9
10    public XmlWebApplicationContext(String fileName) {
11        super(fileName);
12    }
13
14    public ServletContext getServletContext() {
15        return this.servletContext;
16    }
17    public void setServletContext(ServletContext servletContext) {
18        this.servletContext = servletContext;
19    }
20 }

```

到这里，我们就进一步拆解了 DispatcherServlet，拆分出两级 ApplicationContext，当然启动过程还是由 Listener 来负责。所以最后 ContextLoaderListener 初始化时是创建 XmlWebApplicationContext 对象。


 复制代码

```

1  WebApplicationContext wac = new XmlWebApplicationContext(sContextLocation);

```

到这里，Web 环境下的两个 ApplicationContext 都构建完毕了，WebApplicationContext 持有对 parentApplicationContext 的单向引用。当调用 getBean() 获取 Bean 时，先从 WebApplicationContext 中获取，若为空则通过 parentApplicationContext 获取，你可以看一下代码。

 复制代码


```

1     public Object getBean(String beanName) throws BeansException {
2         Object result = super.getBean(beanName);
3         if (result == null) {
4             result = this.parentBeanFactory.getBean(beanName);
5         }
6         return result;
7     }

```

抽取调用方法

拆解的工作还要继续进行，基本的思路是将专业事情交给不同的专业部件来做，我们来看看还有哪些工作是可以分出来的。从代码可以看到现在 doGet() 方法是这样实现的。

 复制代码

```
1  Method method = this.mappingMethods.get(sPath);
2  obj = this.mappingObjs.get(sPath);
3  objResult = method.invoke(obj);
4  response.getWriter().append(objResult.toString());
```


这个程序就是简单地根据 URL 找到对应的方法和对象，然后通过反射调用方法，最后把方法执行的返回值写到 response 里。我们考虑把通过 URL 映射到某个实例方法的过程抽取出来，还要考虑把对方法的调用也单独抽取出来。仿照 Spring 框架，我们新增 RequestMappingHandlerMapping 与 RequestMappingHandlerAdapter，分别对应这两个独立的部件。

首先将 HandlerMapping 与 HandlerAdapter 抽象出来，定义接口，然后基于接口来编程。

 复制代码

```
1  package com.minis.web.servlet;
2
3  import javax.servlet.http.HttpServletRequest;
4
5  public interface HandlerMapping {
6      HandlerMethod getHandler(HttpServletRequest request) throws Exception;
7  }
8
9
10
11 package com.minis.web.servlet;
12
13 import javax.servlet.http.HttpServletRequest;
14 import javax.servlet.http.HttpServletResponse;
15
16 public interface HandlerAdapter {
17     void handle(HttpServletRequest request, HttpServletResponse response, Object ha
18 }
19
```


其中可以看到，HandlerMapping 中定义的 getHandler 方法参数是 http request，返回一个 HandlerMethod 对象，这个地方就是封装的这种映射关系。你可以看一下 HandlerMethod 对象的定义。

 复制代码

```
1 package com.minis.web.servlet;
2
3 import java.lang.reflect.Method;
4
5 public class HandlerMethod {
6     private Object bean;
7     private Class<?> beanType;
8     private Method method;
9     private MethodParameter[] parameters;
10    private Class<?> returnType;
11    private String description;
12    private String className;
13    private String methodName;
14
15    public HandlerMethod(Method method, Object obj) {
16        this.setMethod(method);
17        this.setBean(obj);
18    }
19    public Method getMethod() {
20        return method;
21    }
22    public void setMethod(Method method) {
23        this.method = method;
24    }
25    public Object getBean() {
26        return bean;
27    }
28    public void setBean(Object bean) {
29        this.bean = bean;
30    }
31 }
```


接下来增加一个 MappingRegistry 类，这个类有三个属性：urlMappingNames、mappingObjs 和 mappingMethods，用来存储访问的 URL 名称与对应调用方法及 Bean 实例的关系。你可以看一下相关定义。

```
1 package com.minis.web.servlet;
2
3 import java.lang.reflect.Method;
4 import java.util.ArrayList;
5 import java.util.HashMap;
6 import java.util.List;
7 import java.util.Map;
8
9 public class MappingRegistry {
10     private List<String> urlMappingNames = new ArrayList<>();
11     private Map<String,Object> mappingObjs = new HashMap<>();
12     private Map<String,Method> mappingMethods = new HashMap<>();
13
14     public List<String> getUrlMappingNames() {
15         return urlMappingNames;
16     }
17     public void setUrlMappingNames(List<String> urlMappingNames) {
18         this.urlMappingNames = urlMappingNames;
19     }
20     public Map<String,Object> getMappingObjs() {
21         return mappingObjs;
22     }
23     public void setMappingObjs(Map<String,Object> mappingObjs) {
24         this.mappingObjs = mappingObjs;
25     }
26     public Map<String,Method> getMappingMethods() {
27         return mappingMethods;
28     }
29     public void setMappingMethods(Map<String,Method> mappingMethods) {
30         this.mappingMethods = mappingMethods;
31     }
32 }
```

通过上面的代码可以看出，这三个属性以前其实都已经存在了，是定义在 DispatcherServlet 里的，现在换一个位置，通过 MappingRegistry 这个单独的部件来存放和管理这个映射关系。

好了，有了这些准备之后，我们来看 RequestMappingHandlerMapping 的实现，它要实现 HandlerMapping 接口，初始化过程就是遍历 WAC 中已经注册的所有的 Bean，并处理带有 @RequestMapping 注解的类，使用 mappingRegistry 存储 URL 地址与方法和实例的映射关系。对外它要实现 getHandler() 方法，通过 URL 拿到 method 的调用。

相关源代码如下:

 复制代码

```
1 package com.minis.web.servlet;
2
3 import java.lang.reflect.Method;
4 import java.util.ArrayList;
5 import java.util.Arrays;
6 import javax.servlet.http.HttpServletRequest;
7 import com.minis.beans.BeansException;
8 import com.minis.web.RequestMapping;
9 import com.minis.web.WebApplicationContext;
10
11 public class RequestMappingHandlerMapping implements HandlerMapping{
12     WebApplicationContext wac;
13     private final MappingRegistry mappingRegistry = new MappingRegistry();
14     public RequestMappingHandlerMapping(WebApplicationContext wac) {
15         this.wac = wac;
16         initMapping();
17     }
18     //建立URL与调用方法和实例的映射关系, 存储在mappingRegistry中
19     protected void initMapping() {
20         Class<?> clz = null;
21         Object obj = null;
22         String[] controllerNames = this.wac.getBeanDefinitionNames();
23         //扫描WAC中存放的所有bean
24         for (String controllerName : controllerNames) {
25             try {
26                 clz = Class.forName(controllerName);
27                 obj = this.wac.getBean(controllerName);
28             } catch (Exception e) {
29                 e.printStackTrace();
30             }
31             Method[] methods = clz.getDeclaredMethods();
32             if (methods != null) {
33                 //检查每一个方法声明
34                 for (Method method : methods) {
35                     boolean isRequestMapping =
36 method.isAnnotationPresent(RequestMapping.class);
37                     //如果该方法带有@RequestMapping注解, 则建立映射关系
38                     if (isRequestMapping) {
39                         String methodName = method.getName();
40                         String urlmapping =
41 method.getAnnotation(RequestMapping.class).value();
42
43                         this.mappingRegistry.getUrlMappingNames().add(urlmapping)
44                         this.mappingRegistry.getMappingObjs().put(urlmapping,
```

```

45 obj);
46             this.mappingRegistry.getMappingMethods().put(urlmapping,
47 method);
48         }
49     }
50 }
51 }
52 }
53
54 //根据访问URL查找对应的调用方法
55 public HandlerMethod getHandler(HttpServletRequest request) throws Exception
56 {
57     String sPath = request.getServletPath();
58     if (!this.mappingRegistry.getUrlMappingNames().contains(sPath)) {
59         return null;
60     }
61     Method method = this.mappingRegistry.getMappingMethods().get(sPath);
62     Object obj = this.mappingRegistry.getMappingObjs().get(sPath);
63     HandlerMethod handlerMethod = new HandlerMethod(method, obj);
64     return handlerMethod;
65 }
66 }

```

这样我们就得到了独立的 RequestMappingHandlerMapping 部件，把以前写在 DispatcherServlet 里的代码移到这里来了。

接下来就轮到 RequestMappingHandlerAdapter 的实现了，它要实现 HandlerAdapter 接口，主要就是实现 handle() 方法，基本过程是接受前端传 request、response 与 handler，通过反射中的 invoke 调用方法并处理返回数据。

相关源代码如下：

 复制代码

```

1 package com.minis.web.servlet;
2
3 import java.io.IOException;
4 import java.lang.reflect.InvocationTargetException;
5 import java.lang.reflect.Method;
6 import javax.servlet.http.HttpServletRequest;
7 import javax.servlet.http.HttpServletResponse;
8 import com.minis.web.WebApplicationContext;
9

```

```

10 public class RequestMappingHandlerAdapter implements HandlerAdapter {
11     WebApplicationContext wac;
12
13     public RequestMappingHandlerAdapter(WebApplicationContext wac) {
14         this.wac = wac;
15     }
16
17     public void handle(HttpServletRequest request, HttpServletResponse response, Ob
18         throws Exception {
19         handleInternal(request, response, (HandlerMethod) handler);
20     }
21     private void handleInternal(HttpServletRequest request, HttpServletResponse res
22         HandlerMethod handler) {
23         Method method = handler.getMethod();
24         Object obj = handler.getBean();
25         Object objResult = null;
26         try {
27             objResult = method.invoke(obj);
28         } catch (Exception e) {
29             e.printStackTrace();
30         }
31         try {
32             response.getWriter().append(objResult.toString());
33         } catch (IOException e) {
34             e.printStackTrace();
35         }
36     }
37 }

```

重点看一下 `handleInternal()` 方法就知道了，这里就是简单地通过反射调用某个方法，然后把返回值写到 `response` 里。这些程序代码以前就有，只不过现在移到单独的这个部件中了。

最后需要修改 `DispatcherServlet` 中的实现，相关代码移走，放到了上面的两个部件中。所以在 `DispatcherServlet` 类中需要增加对 `HandlerMapping` 与 `HandlerAdapter` 的引用，在初始化方法 `refresh()` 中增加 `initHandlerMapping` 与 `initHandlerAdapter` 两个方法，为引用的 `HandlerMapping` 与 `HandlerAdapter` 赋值。

你可以看下 `DispatcherServlet` 的 `refresh()` 的改造结果。

 复制代码


```

1 refresh() {
2     initController();

```


```
3
4     initHandlerMappings(this.webApplicationContext);
5     initHandlerAdapters(this.webApplicationContext);
6 }
```

初始化这两个部件的代码如下：

 复制代码

```
1     protected void initHandlerMappings(WebApplicationContext wac) {
2         this.handlerMapping = new RequestMappingHandlerMapping(wac);
3     }
4     protected void initHandlerAdapters(WebApplicationContext wac) {
5         this.handlerAdapter = new RequestMappingHandlerAdapter(wac);
6     }
```

DispatcherServlet 的分发过程也要改造一下，不再通过 doGet() 方法了，而是通过重写的 service 方法来实现的，而 service 方法则调用了 doDispatch 方法，这个方法内部通过 handlerMapping 获取到对应 handlerMethod，随后通过 HandlerAdapter 进行处理，你可以看一下这个类修改后的源代码。

 复制代码

```
1     protected void service(HttpServletRequest request, HttpServletResponse
2 response) {
3         request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
4 this.webApplicationContext);
5         try {
6             doDispatch(request, response);
7         } catch (Exception e) {
8             e.printStackTrace();
9         }
10        finally {
11        }
12    }
13    protected void doDispatch(HttpServletRequest request, HttpServletResponse
14 response) throws Exception{
15        HttpServletRequest processedRequest = request;
16        HandlerMethod handlerMethod = null;
17        handlerMethod = this.handlerMapping.getHandler(processedRequest);
18        if (handlerMethod == null) {
19            return;
20        }
```

```
21     HandlerAdapter ha = this.handlerAdapter;
22     ha.handle(processedRequest, response, handlerMethod);
23 }
```

可以看到，经过这么一改造，相比之前 DispatcherServlet 的代码简化了很多，并且当前业务程序不用再固定写死在 doGet() 方法里面，可以按照自身的业务需求随意使用任何方法名，也为今后提供多种请求方式，例如 POST、PUT、DELETE 等提供了便利。

以前，用原始的 Servlet 规范，我们的业务逻辑全部写在 doGet()、doPost() 等方法中，每一个业务逻辑程序都是一个独立的 Servlet。现在经过我们这几节课的操作，整个系统用一个唯一的 DispatcherServlet 来拦截请求，并根据注解，定位需要调用的方法，我们就能够更加专注于本身业务代码的实现。这种我们称之为 Dispatcher 的设计模式也是要用心学习的。

小结

这节课我们的主要工作就是拆解 Dispatcher。首先拆解的是 ApplicationContext，现在有了两级上下文，一级用于 IoC 容器，我们叫 parent 上下文，一级用于 Web 上下文，WebApplicationContext 持有对 parent 上下文的引用。方便起见，我们还增加了 @RequestMapping 注解来声明 URL 映射，然后新增 RequestMappingHandlerMapping 与 RequestMappingHandlerAdapter，分别包装 URL 映射关系和映射后的处理过程。

通过这些拆解工作，我们就把 DispatcherServlet 的功能进行了分治，把专门的事情交给专门的部件去完成，有利于今后的扩展。

完整源代码参见 [🔗 https://github.com/YaleGuo/minis](https://github.com/YaleGuo/minis)

课后题

学完这节课，我也给你留一道思考题。目前，我们只支持了 GET 方法，你能不能尝试自己增加 POST 方法。想一想，需要改变现有的程序结构吗？欢迎你在留言区和我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (7)



马儿

2023-04-01 来自四川

总结一下：

1. Listener初始化的时候将交给Ioc管理的Bean初始化

2.Servlet初始化的时候将controller相关的bean初始化

这两步初始化将bean的管理从DispatcherServlet剥离交给了第一章创建的Ioc容器

3.将具体的url和对象+方法的管理从Servlet交给HandlerMapping来处理

4.将具体的方法执行剥离到HandlerAdapter

这两步将DispatcherServlet变得更抽象了，利用service方法可以同时处理不同类型的请求

一点建议：

1. DispatcherServlet中的controller相关bean的初始化已经交给AnnotationConfigWebApplication
nContext管理了，它的init方法不用在调用initController了

作者回复：赞你一个



1



Imnsds

2023-05-15 来自北京

在github代码的geek_mvc3分支找了半天 这节课的 DispatcherServlet，原来不是在web包下改的原有类，而是在web.servlet包下新增了个DispatcherServlet！浪费了好多时间！给后来人提个醒吧。

作者回复：多谢提醒



风轻扬

2023-04-08 来自北京

思考题:我的想法是模仿SpringMVC,在RequestMapping注解上增加一个HttpMethod的属性(当前方法允许的请求方式)。在解析RequestMapping注解的时候改动一下,拿到RequestMapping注解上的HttpMethod,将其放到HandlerMethod中,然后将HandlerMethod对象放进MappingRegistry的一个map中,key:path,value:HandlerMethod。用户发起请求时,doDispatch方法中,获取到HttpServletRequest对象中的请求方式和MappingRegistry中存储的HandlerMethod上的请求方

式进行比较,如果符合就可以访问,否则就报出方法类型不匹配

另外, 有两个问题请教一下老师。

1、问题一:

```
DefaultListableBeanFactory beanFactory;
```

```
DefaultListableBeanFactory bf = new DefaultListableBeanFactory();
```

```
this.beanFactory = bf;
```

我看老师在很多地方都是这样写的, 为啥不直接给成员变量赋值呢?
`this.beanFactory = new DefaultListableBeanFactory();`

2、问题2

为什么Spring要搞出两个容器来呢?

我从StackOverFlow上搜了一下相关解释:<https://stackoverflow.com/questions/18578143/about-multiple-containers-in-spring-framework>

看上面的解释是:

这样分开更清晰,Dispatcher驱动的子容器专门用来处理controller组件,ContextLoaderListener驱动的父亲容器专门用来处理业务逻辑组件以及持久化组件。

除了这个原因,Spring搞2个容器还有其他原因吗?

作者回复: 思考题后面有再回首章节里面给出了参考, 你可以到时候再思考一下。

Q1, 习惯问题吧, Spring框架本身也是经常这样写。有一个潜在原因, 但是我不是很确定, 就是方法中尽量使用local变量, 只在尽量少的场合使用实例变量和方法参数, 这样提高性能减少开销。编译器优化是这么说的, 但是我也不确定一定就是这样。另外, 对源码, 我还是建议最后去读Spring框架的源代码, 那是世界顶级程序员的力作(推广到别的也是一样的, Apache Tomcat, JDK, Apache Dubbo)。我自己也是在跟着他们的时候一旁偷学了几招, MiniSpring的目的是剖析Spring框架内部结构, 作为一个简明地图引导大家理解Spring。

Q2, 就是这样解释的。Spring体系中IoC是核心层, MVC只是外周的部分, 理论上是可以不启用的可选部件。软件是一层层堆积的, 层次感要慢慢培养。

共 2 条评论 >



梦某人

2023-04-05 来自河北

首先以个人理解回答课后题, 目前的请求并不分 get 或者 post, 主要是以请求的路径进行区分, 如果想要处理 post 请求,

需要构建新的 HandlerAdapter, 对 Request 中的 post body 内容进行额外的解析和处理, 然后操作方法。当然可能还需要构建 HandlerMapping 来处理请求路径, 但是个人没想到什么 get 和 post 区别很大的地方。

第二和第三点是个人跟写的时候遇到的一些问题, 给其他同学一点参考。

第二个,

```

```
// getBeanDefinitionNames 方法中
return (String[]) this.beanNames.toArray();
//替换成
return this.beanNames.toArray(new String[0]);
```
```

可以减少一些类型转换异常，特别是当 ArrayList 里面只有一个 String 元素的时候。

第三个，BeanDefinition 中的部分 get 方法应该增加运算符 防止返回 Null 而不是 empty，导致空指针异常。例如：

```

```
public ArgumentValues getArgumentValues() {
 return argumentValues == null? new ArgumentValues(): argumentValues;
}
```
```

第四点是课程个人理解了：两级的 WebApplicationContext 第一级在 Listener 的时候加载，加载了 beans.xml (或者 Application.xml) 中的 bean，然后作为 第二级 AnnotationConfigWebApplicationContext 的父级，第二级别通过 mvc.xml 提供的扫描路径进行扫描加载 bean，同时注册带有注解的方法。当路由请求来的时候，先从第二级的 WebApplicationContext 获取 bean 和其方法进行处理，所以这个两级在最后的时候以 Controller 和 Service 来进行讲解，不是真的 Controller 和 Service，而是说 第二级处理事物的触发逻辑比第一级更早，加载的逻辑则比他更晚，就好像 请求先到 Controller 后到 Service 一样。

最后的最后，，，看着老师文稿给的代码来吵，已经是和 GitHub 中的代码差别越来越大了，Debug 起来更加费时，但是好处是理解加深了。

作者回复: 写得很好很好。我觉得，不管通过什么方式，最后能达到理解了Spring框架的目的就好了，我的建议还是读文稿，听我讲，手工练习，自己动手过一遍就是不一样的，线下班学员还要做扩展练习，加上更多功能和鲁棒性。源代码方面，MiniSpring主要是便于教学时理解Spring框架的结构，代码最后要学习Spring框架本身的源代码。



C.

2023-04-03 来自江苏

出去玩了两天，今天把这章也结束掉了。代码运行一切正常。

作者回复: 赞



Geek_320730

2023-04-02 来自北京

1. 课后题: 重写service后不是Get 和Post都能处理吗?
2. 扫描的包里有接口, 接口应该是不能实例化的, 我过滤了下接口类型才能启动起来, 对比了下老师的代码, 好像并没有处理。
3. 尝试了下在HelloWorldBean里注入parentApplicationContext中创建的Bean, 发现了个小问题, AbstractBeanFactory#getBean方法中如果获取不到BeanDefinition 应该返回个null, 而不是抛出异常, 否则不会去父类查找。对构造器注入参数和set注入参数增加null校验

作者回复: 你的思考和练习很好有深度, 用心了。MiniSpring是为了学习构建的, 不是工业级的。线下听课的几波学生, 要对照代码进行扩展练习, 自己增加功能特性增加鲁棒性。

共 2 条评论 >



peter

2023-04-01 来自北京

请教两个问题:

DispatcherServlet 这个类里, 有两个WebApplicationContext对象: private WebApplicationContext webApplicationContext;

private WebApplicationContext parentApplicationContext;

请问, 这两个对象是同一个对象吗??

Q2: 文中的controller和service是业务层的吗?

文中有这样的描述: “按照通行的 Web 分层体系, 一个程序它在结构上会有 Controller 和 Service 两层。在我们的程序中, Controller 由 DispatcherServlet 负责启动, Service 由 Listener 负责启动。”

程序员写业务代码的时候, 会按照controller、service、dao来写。

请问, 文中的controller和service是业务层的controller、service吗? (即程序员写的controller、service)

作者回复: 不是同一个, 是两级context.

controller和service是两层, 并不特别规定是谁。我这里主要想描述的是dispatcherServlet中的控制逻辑与ioc容器中管理的bean的业务逻辑是前后两层的关系。

共 2 条评论>

