

课程目标

内容定位

1. 实现思路

2. 自定义配置

2.1. 配置application.properties文件

2.2. 配置web.xml文件

2.3. 自定义Annotation

2.4. 配置Annotation

3. 容器初始化

3.1. 实现V1版本

3.2. 实现V2版本

3.3. 实现V3版本

课程目标

- 1、了解看源码最有效的方式，先猜测后验证，不要一开始就去调试代码。
- 2、浓缩就是精华，用300行最简洁的代码提炼Spring的基本设计思想。
- 3、掌握Spring 框架的基本脉络。

内容定位

- 1、具有1年以上的SpringMVC使用经验。
- 2、希望深入了解Spring 源码的人群，对Spring有一个整体的宏观感受。
- 3、全程手写实现SpringMVC的核心功能，从最简单的V1版本一步一步优化为V2版本，最后到V3版本。

1. 实现思路

先来介绍一下Mini版本的Spring基本实现思路，如下图所示：



2. 自定义配置

2.1. 配置application.properties文件

为了解析方便，我们用application.properties 来代替application.xml文件，具体配置内容如下：

```
1 scanPackage=cn.sitedev.demo
```

2.2. 配置web.xml文件

大家都知道，所有依赖于web容器的项目，都是从读取 web.xml文件开始的。我们先配置好web.xml中的内容。

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:javaee="http://java.sun.com/xml/
4     xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns
6     version="2.4">
7     <display-name>My Web Application</display-name>
8     <servlet>
9         <servlet-name>mvc</servlet-name>
```

```

10     <servlet-class>cn.sitedev.mvcframework.v2.servlet.MyDispatchServlet</servlet-cl
11     <init-param>
12         <param-name>contextConfigLocation</param-name>
13         <param-value>classpath*:application.properties</param-value>
14     </init-param>
15
16     <load-on-startup>1</load-on-startup>
17 </servlet>
18 <servlet-mapping>
19     <servlet-name>mvc</servlet-name>
20     <url-pattern>/*</url-pattern>
21 </servlet-mapping>
22 </web-app>

```

其中MyDispatcherServlet是有自己模拟Spring 实现的核心功能类。

2.3. 自定义Annotation

@MyService 注解：

```

1 package cn.sitedev.mvcframework.annotation;
2
3 import java.lang.annotation.*;
4
5 @Target(ElementType.TYPE) // 类/接口/枚举
6 @Retention(RetentionPolicy.RUNTIME)
7 @Documented
8 public @interface MyService {
9     String value() default "";
10 }

```

@MyAutowired 注解：

```

1 package cn.sitedev.mvcframework.annotation;
2
3 import java.lang.annotation.*;
4
5 @Target(ElementType.FIELD)
6 @Retention(RetentionPolicy.RUNTIME)
7 @Documented

```

```
8 public @interface MyAutowired {
9     String value() default "";
10 }
```

@MyController 注解：

```
1 package cn.sitedev.mvcframework.annotation;
2
3 import java.lang.annotation.*;
4
5 @Target(ElementType.TYPE)
6 @Retention(RetentionPolicy.RUNTIME)
7 @Documented
8 public @interface MyController {
9     String value() default "";
10 }
```

@MyRequestMapping 注解：

```
1 package cn.sitedev.mvcframework.annotation;
2
3 import java.lang.annotation.*;
4
5 @Target({ElementType.TYPE, ElementType.METHOD})
6 @Retention(RetentionPolicy.RUNTIME)
7 @Documented
8 public @interface MyRequestMapping {
9     String value() default "";
10 }
```

@MyRequestParam 注解：

```
1 package cn.sitedev.mvcframework.annotation;
2
3 import java.lang.annotation.*;
4
5 @Target(ElementType.PARAMETER)
6 @Retention(RetentionPolicy.RUNTIME)
```

```
7 @Documented
8 public @interface MyRequestParam {
9     String value() default "";
10 }
```

2.4. 配置Annotation

配置业务实现类DemoService：

...

```
1 package cn.sitedev.demo.service.impl;
2
3 import cn.sitedev.demo.service.IDemoService;
4 import cn.sitedev.mvcframework.annotation.MyService;
5
6 /**
7  * 核心业务逻辑
8  */
9 @MyService
10 public class DemoService implements IDemoService {
11     @Override
12     public String get(String name) {
13         return "My name is " + name;
14     }
15 }
```

配置请求入口类DemoAction：

```
1 package cn.sitedev.demo.mvc.action;
2
3 import cn.sitedev.demo.service.IDemoService;
4 import cn.sitedev.mvcframework.annotation.MyAutowired;
5 import cn.sitedev.mvcframework.annotation.MyController;
6 import cn.sitedev.mvcframework.annotation.MyRequestMapping;
7 import cn.sitedev.mvcframework.annotation.MyRequestParam;
8
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11 import java.io.IOException;
12
```

```

13 @MyController
14 @MyRequestMapping("/demo")
15 public class DemoAction {
16     @MyAutowired
17     private IDemoService demoService;
18     private IDemoService demoService2;
19
20     @MyRequestMapping("/query")
21     public void query(HttpServletRequest request, HttpServletResponse response, @MyRequest
22         "name") String name, @MyRequestParam("id") String id) {
23         String result = demoService.get(name);
24         try {
25             response.getWriter().write(result);
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29     }
30
31     @MyRequestMapping("/add")
32     public void add(HttpServletRequest request, HttpServletResponse response,
33         @MyRequestParam("a") Integer a, @MyRequestParam("b") Integer b) {
34         try {
35             response.getWriter().write(a + "+" + b + "=" + (a + b));
36         } catch (IOException e) {
37             e.printStackTrace();
38         }
39     }
40
41     @MyRequestMapping("/remove")
42     public void remove(HttpServletRequest request, HttpServletResponse response, @MyReq
43         "id") Integer id) {
44
45     }
46
47 }

```

至此，配置阶段就已经完成。

3. 容器初始化

3.1. 实现V1版本

所有的核心逻辑全部写在一个init()方法中。

```
1 package cn.sitedev.mvcframework.servlet.v1;
2
3 import cn.sitedev.mvcframework.annotation.MyAutowired;
4 import cn.sitedev.mvcframework.annotation.MyController;
5 import cn.sitedev.mvcframework.annotation.MyRequestMapping;
6 import cn.sitedev.mvcframework.annotation.MyService;
7
8 import javax.servlet.ServletConfig;
9 import javax.servlet.ServletException;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13 import java.io.File;
14 import java.io.IOException;
15 import java.io.InputStream;
16 import java.lang.reflect.Field;
17 import java.lang.reflect.Method;
18 import java.net.URL;
19 import java.util.HashMap;
20 import java.util.Map;
21 import java.util.Properties;
22
23 public class MyDispatcherServlet extends HttpServlet {
24     private Map<String, Object> mapping = new HashMap<>();
25
26     @Override
27     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
28         this.doPost(req, resp);
29     }
30
31     @Override
32     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException {
33         try {
34             doDispatch(req, resp);
35         } catch (Exception e) {
36             resp.getWriter().write("500 Exception " + e.getMessage());
37         }
38     }
39
40     private void doDispatch(HttpServletRequest request, HttpServletResponse response) {
```

```

41 // 获取请求url
42 String url = request.getRequestURI();
43 // 获取context path
44 String contextPath = request.getContextPath();
45 // 从请求url中截去context path
46 url = url.replace(contextPath, "").replaceAll("/+", "/");
47 // 判断是否有对应的方法来处理该url对应的请求
48 if (!this.mapping.containsKey(url)) {
49     response.getWriter().write("404 Not Found!");
50     return;
51 }
52 // 获取该请求url对应的方法对象
53 Method method = (Method) this.mapping.get(url);
54 // 获取请求参数map, 该map形如: "name"={key="name",value=["zhangsan"]}, "id"={key
55 Map<String, String[]> params = request.getParameterMap();
56 // 通过反射调用处理该url对应的方法, 这里只处理了/demo/query对应的请求
57 method.invoke(this.mapping.get(method.getDeclaringClass().getName()),
58     new Object[]{request, response, params.get("name")[0], params.get("id")
59 }
60
61 @Override
62 public void init(ServletConfig config) throws ServletException {
63     InputStream inputStream = null;
64     try {
65         // 读取配置文件application.properties
66         Properties configContext = new Properties();
67         inputStream =
68             this.getClass().getClassLoader().getResourceAsStream(config.getInit
69                 "contextConfigLocation"));
70         configContext.load(inputStream);
71         // 获取scanPackage配置
72         String scanPackage = configContext.getProperty("scanPackage");
73         // 递归扫描指定包路径下的子包以及类
74         doScanner(scanPackage);
75
76         Map<String, Object> extMapping = new HashMap<>();
77         // 遍历指定包路径下的类
78         for (String className : mapping.keySet()) {
79             if (!className.contains(".")) {
80                 continue;
81             }
82             // 加载指定类
83             Class<?> clazz = Class.forName(className);

```



```

84 // 判断该类上是否有MyController注解
85 if (clazz.isAnnotationPresent(MyController.class)) {
86     // key: 类的完全限定名 value: 该类对应实例
87     extMapping.put(className, clazz.newInstance());
88     String baseUrl = "";
89     // 判断该类上是否有MyRequestMapping注解
90     if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
91         MyRequestMapping requestMapping =
92             clazz.getAnnotation(MyRequestMapping.class);
93         // 获取MyRequestMapping注解的value值
94         baseUrl = requestMapping.value();
95     }
96     // 获取类中的所有方法对象(包含自定义的方法,以及继承自Object类的方法
97     Method[] methods = clazz.getMethods();
98     // 遍历所有方法对象
99     for (Method method : methods) {
100         // 判断方法上是否有MyRequestMapping注解
101         if (!method.isAnnotationPresent(MyRequestMapping.class)) {
102             continue;
103         }
104         // 获取MyRequestMapping注解的value值
105         MyRequestMapping requestMapping =
106             method.getAnnotation(MyRequestMapping.class);
107         // 拼接请求的url(如果存在多个/, 则替换成一个/)
108         String url = (baseUrl + "/" + requestMapping.value()).replaceAll("/+", "/");
109         // key: 请求url, value: 处理对应url的方法对象
110         extMapping.put(url, method);
111         System.out.println("Mapped : " + url + ", " + method);
112     }
113     // 判断类上是否有MyService注解
114 } else if (clazz.isAnnotationPresent(MyService.class)) {
115     // 获取MyService注解的value值
116     MyService service = clazz.getAnnotation(MyService.class);
117     String beanName = service.value();
118     if ("".equals(beanName)) {
119         beanName = clazz.getName();
120     }
121     Object instance = clazz.newInstance();
122     // key: 类的完全限定名, value: 类的实例
123     extMapping.put(beanName, instance);
124     for (Class<?> i : clazz.getInterfaces()) {
125         // key: 类实现的接口的完全限定名, value: 类的实例

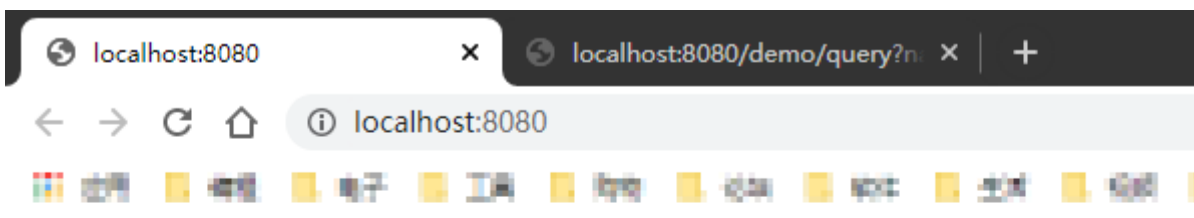
```

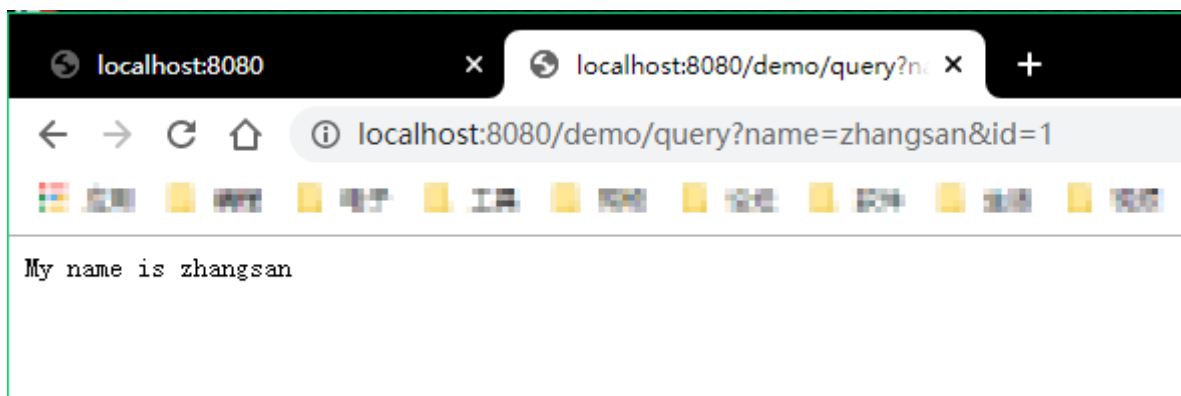
```
127         extMapping.put(i.getName(), instance);
128     }
129     } else {
130         continue;
131     }
132 }
133
134 mapping.putAll(extMapping);
135 for (Object object : mapping.values()) {
136     if (object == null) {
137         continue;
138     }
139     Class<?> clazz = object.getClass();
140     // 判断类上是否有MyController注解
141     if (clazz.isAnnotationPresent(MyController.class)) {
142         // 获取类所有的字段
143         Field[] fields = clazz.getDeclaredFields();
144         // 遍历类的所有字段
145         for (Field field : fields) {
146             // 判断当前字段上是否有MyAutowired注解
147             if (!field.isAnnotationPresent(MyAutowired.class)) {
148                 continue;
149             }
150             // 获取MyAutowired注解的value值
151             MyAutowired autowired = field.getAnnotation(MyAutowired.class);
152             String beanName = autowired.value();
153             if ("".equals(beanName)) {
154                 beanName = field.getType().getName();
155             }
156             // 使该字段可被访问
157             field.setAccessible(true);
158             // 给该字段设置值
159             field.set(mapping.get(clazz.getName()), mapping.get(beanName));
160         }
161     }
162 }
163 } catch (Exception e) {
164     e.printStackTrace();
165 } finally {
166     // 释放资源
167     if (inputStream != null) {
168         try {
169             inputStream.close();
```

```

170         } catch (IOException e) {
171             e.printStackTrace();
172         }
173     }
174 }
175 System.out.println("My MVC framework is init !");
176
177 }
178
179 private void doScanner(String scanPackage) {
180     URL url = this.getClass().getClassLoader().getResource("/" + scanPackage.replace(
181         , "/"));
182     File classDir = new File(url.getFile());
183     for (File file : classDir.listFiles()) {
184         if (file.isDirectory()) {
185             // 递归扫描
186             doScanner(scanPackage + "." + file.getName());
187         } else {
188             // 处理.class文件
189             if (!file.getName().endsWith(".class")) {
190                 continue;
191             }
192             // 存入mapping => key:类的完全限定名, value:null
193             String clazzName = scanPackage + "." + file.getName().replace(".class",
194                 mapping.put(clazzName, null));
195         }
196     }
197 }
198 }

```





3.2. 实现V2版本

在V1版本上进行了优化，采用了常用的设计模式（工厂模式、单例模式、委派模式、策略模式），将init()方法中的代码进行封装。按照之前的实现思路，先搭基础框架，再填肉注血，具体代码如下：

```
1 package cn.sitedev.mvcframework.servlet.v2;
2
3 import javax.servlet.ServletConfig;
4 import javax.servlet.ServletException;
5 import javax.servlet.http.HttpServlet;
6
7 public class MyDispatcherServlet extends HttpServlet {
8
9     @Override
10     public void init(ServletConfig config) throws ServletException {
11
12         // 1. 加载配置文件
13         doLoadConfig(config.getInitParameter("contextConfigLocation"));
14
15         // 2. 扫描相关的类
16         doScanner(contextConfig.getProperty("scanPackage"));
17
18         // 3. 初始化扫描到的类，并且放入到IOC容器之中
19         doInstance();
20
21         // 4. 完成自动化的依赖注入
22         doAutowired();
23
24         // 5. 初始化HandlerMapping
25         doInitHandlerMapping();
26
27         System.out.println("My Spring Framework is init...");
28     }
```

```
29 }
```

声明全局的成员变量，其中IOC容器就是注册时单例的具体案例：

```
1 // 保存application.properties配置文件中的内容
2 private Properties contextConfig = new Properties();
3 // 保存扫描到的所有的类名
4 private List<String> classNames = new ArrayList<>();
5 // IOC容器
6 // 为了简化程序，我们暂不考虑ConcurrentHashMap
7 // 主要还是关注设计思想和原理
8 private Map<String, Object> ioc = new HashMap<>();
9
10 // 保存url和方法的对应关系
11 private Map<String, Method> handlerMapping = new HashMap<>();
```

实现doLoadConfig()方法：

```
1 // 加载配置文件
2 private void doLoadConfig(String contextConfigLocation) {
3     // 直接从类路径下找到Spring主配置文件所在的路径
4     // 并且将其读取出来放到Properties对象中
5     // 相对于scanPackage=cn.sitedev.demo从文件中保存到了内存中
6     InputStream inputStream =
7         this.getClass().getClassLoader().getResourceAsStream(contextConfigLocation);
8     try {
9         contextConfig.load(inputStream);
10    } catch (IOException e) {
11        e.printStackTrace();
12    } finally {
13        // 释放资源
14        if (inputStream != null) {
15            try {
16                inputStream.close();
17            } catch (IOException e) {
18                e.printStackTrace();
19            }
20        }
21    }
22 }
```

实现doScanner()方法：

```
1 // 扫描相关的类
2 private void doScanner(String scanPackage) {
3     URL url = this.getClass().getClassLoader().getResource("/" + scanPackage.replac
4         , "/"));
5     // scanPackage=cn.sitedev.demo, 存储的就是扫描的包路径
6     // 转换为文件路径, 实际上就是将.替换成/
7     // classpath下不仅有.class文件, .xml文件, .properties文件
8     File classPath = new File(url.getFile());
9     // 遍历该路径下的所有文件/文件夹
10    for (File file : classPath.listFiles()) {
11        if (file.isDirectory()) {
12            // 递归遍历
13            doScanner(scanPackage + "." + file.getName());
14        } else {
15            // 变成 包名.类名
16            if (!file.getName().endsWith(".class")) {
17                continue;
18            }
19            classNames.add(scanPackage + "." + file.getName().replace(".class", ""))
20        }
21    }
22 }
```

实现doInstance()方法, doInstance()方法就是工厂模式的具体实现：

```
1 private void doInstance() {
2     if (classNames.isEmpty()) {
3         return;
4     }
5     try {
6         for (String className : classNames) {
7             Class<?> clazz = Class.forName(className);
8
9             // 什么样的类才需要初始化呢?
10            // 加了注解的类, 才会初始化, 怎么判断?
11            // 为了简化代码逻辑, 主要体会设计思想, 只举例@Controller和服务,
12            // @Component... 就不一一举例了
```

```

13
14 // 如果该类上有MyController注解
15 if (clazz.isAnnotationPresent(MyController.class)) {
16     Object instance = clazz.newInstance();
17     String beanName = toLowerFirstCase(clazz.getSimpleName());
18     // key:类名的首字母小写, value:类的实例
19     ioc.put(beanName, instance);
20 }
21 // 如果该类上有MyService注解
22 else if (clazz.isAnnotationPresent(MyService.class)) {
23     // 1. 默认就根据beanName类名首字母小写
24     String beanName = toLowerFirstCase(clazz.getSimpleName());
25
26     // 2. 使用自定义的beanName
27     MyService service = clazz.getAnnotation(MyService.class);
28     if (!"".equals(service.value())) {
29         beanName = service.value();
30     }
31
32     Object instance = clazz.newInstance();
33     // key:类名的首字母小写或自定义的名称, value:类的实例
34     ioc.put(beanName, instance);
35
36     // 3. 根据包名.类名作为beanName
37     for (Class<?> i : clazz.getInterfaces()) {
38         if (ioc.containsKey(i.getName())) {
39             throw new Exception("The beanName is exists");
40         }
41         // 把接口的类型直接当成key了
42         // key:接口类型, value:类的实例
43         ioc.put(i.getName(), instance);
44     }
45 } else {
46     continue;
47 }
48 }
49 } catch (Exception e) {
50     e.printStackTrace();
51 }
52 }

```

为了处理方便，自己实现了toLowerFirstCase方法，来实现类名首字母小写，具体代码如下：

```

1 // 如果类名本身是小写字母，确实会出问题
2 // 但是我要说明的是：这个方法是我自己用的，private类型的
3 // 传值也是自己传，存在首字母小写的情况，也不可能出现非字母的情况
4 // 为了简化程序逻辑，就不做其他判断了
5 private String toLowerFirstCase(String simpleName) {
6     char[] chars = simpleName.toCharArray();
7     // 之所以采用 += ，是因为大小写字母的ASCII码值相差32
8     // 而且大写字母的ASCII码要小于小写字母的
9     // 在Java中，对char做数学运算，实际上就是对ASCII码做数学运算
10    chars[0] += 32;
11    return String.valueOf(chars);
12 }

```

实现doAutowired()方法：

```

1 private void doAutowired() {
2     if (ioc.isEmpty()) {
3         return;
4     }
5
6     for (Map.Entry<String, Object> entry : ioc.entrySet()) {
7         // 拿到实例的所有字段
8         // Declared 所有的，特定的 字段，包括private/protected/default
9         // 正常来说，普通的OOP编程只能拿到public的属性
10        Field[] fields = entry.getValue().getClass().getDeclaredFields();
11        // 遍历类的所有字段，并给指定注解的字段赋值
12        for (Field field : fields) {
13            // 判断当前字段上是否有MyAutowired注解
14            if (!field.isAnnotationPresent(MyAutowired.class)) {
15                continue;
16            }
17            // 获取MyAutowired注解
18            MyAutowired autowired = field.getAnnotation(MyAutowired.class);
19            // 如果用户没有自定义beanName，默认就根据类型注入
20            // 这个地方省去了对类名首字母小写的情况的判断
21            String beanName = autowired.value().trim();
22            if ("".equals(beanName)) {
23                // 获得接口的类型，作为key。待会拿到这个key到ioc容器中去取值
24                beanName = field.getType().getName();
25            }

```



```

26
27         // 如果是public以外的修饰符，只要加了@MyAutowired注解，都要强制赋值
28         // 反射中叫做暴力访问
29         field.setAccessible(true);
30
31         // 反射调用的方式
32         // 给entry.getValue()这个对象的field字段，赋ioc.get(beanName)这个值
33         try {
34             field.set(entry.getValue(), ioc.get(beanName));
35         } catch (IllegalAccessException e) {
36             e.printStackTrace();
37             continue;
38         }
39
40     }
41
42 }
43

```

实现doInitHandlerMapping()方法，handlerMapping就是策略模式的应用案例：

```

1         // 初始化url和Method的一一对应关系
2         private void doInitHandlerMapping() {
3             if (ioc.isEmpty()) {
4                 return;
5             }
6
7             // 遍历ioc容器中的bean
8             for (Map.Entry<String, Object> entry : ioc.entrySet()) {
9                 Class<?> clazz = entry.getValue().getClass();
10                // 判断当前类上是否有MyController注解
11                if (!clazz.isAnnotationPresent(MyController.class)) {
12                    continue;
13                }
14
15                // 保存写在类上面的@MyRequestMapping("/demo")
16                String baseUrl = "";
17                // 判断当前类上是否有MyRequestMapping注解
18                if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
19                    // 获取MyRequestMapping注解
20                    MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.class);

```

```

21         baseUrl = requestMapping.value();
22     }
23
24     // 默认获取所有的public方法
25     for (Method method : clazz.getMethods()) {
26         // 判断方法实例上是否有MyRequestMapping注解
27         if (!method.isAnnotationPresent(MyRequestMapping.class)) {
28             continue;
29         }
30         // 获取MyRequestMapping注解
31         MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping.class);
32
33         // 将类上的MyRequestMapping注解的value值与类方法上的MyRequestMapping注解
34         // "demo" + "query" => "/demo/query"
35         // "/demo" + "/query" => "//demo//query" => "/demo/query"
36         String url = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll("//", "/");
37         // key:请求url, value: 处理对应url的方法
38         handlerMapping.put(url, method);
39
40         System.out.println("Mapped : " + url + ", " + method);
41     }
42 }
43 }

```

到这里位置初始化阶段就已经完成，接下实现运行阶段的逻辑，来看 doPost/doGet的代码：

```

1     @Override
2     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
3         this.doPost(req, resp);
4     }
5
6     @Override
7     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
8         // 6. 根据url调用method
9         try {
10             doDispatch(req, resp);
11         } catch (Exception e) {
12             e.printStackTrace();
13             resp.getWriter().write("500 Exception, Detail: " + Arrays.toString(e.getStackTrace()));
14         }
15     }

```

doPost()方法中，用了委派模式，委派模式的具体逻辑在 doDispatch()方法中：

```
1     private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
2         String url = req.getRequestURI();
3         String contextPath = req.getContextPath();
4         // 从请求url中截去context path
5         url = url.replaceAll(contextPath, "").replaceAll("/+", "/");
6
7         if (!this.handlerMapping.containsKey(url)) {
8             resp.getWriter().write("404 Not Found !");
9             return;
10        }
11
12        Method method = this.handlerMapping.get(url);
13
14        // 获取请求参数map
15        Map<String, String[]> paramsMap = req.getParameterMap();
16
17        String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
18        // 调用处理该请求url对应的方法
19        method.invoke(ioc.get(beanName), new Object[] {req, resp, paramsMap.get("name")[
20            paramsMap.get("id")[0]});
21    }
```

在以上代码中，doDispatch()虽然完成了动态委派并反射调用，但对url参数处理还是静态代码。要实现url参数的动态获取，其实还稍微有些复杂。我们可以优化doDispatch()方法的实现逻辑，代码如下：

```
1     private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
2         String url = req.getRequestURI();
3         String contextPath = req.getContextPath();
4         // 从请求url中截去context path
5         url = url.replaceAll(contextPath, "").replaceAll("/+", "/");
6
7         if (!this.handlerMapping.containsKey(url)) {
8             resp.getWriter().write("404 Not Found !");
9             return;
10        }
11
```

```

12     Method method = this.handlerMapping.get(url);
13
14     // 获取请求参数map
15     // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", value=["
16     Map<String, String[]> paramsMap = req.getParameterMap();
17
18     //     String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName()
19     //     // 调用处理该请求url对应的方法
20     //     method.invoke(ioc.get(beanName), new Object[]{req, resp, paramsMap.get("name"
21     //     paramsMap.get("id")[0]});
22
23     // 实参列表
24     // 实参列表要根据形参列表才能决定, 首先得拿到形参列表
25     Class<?>[] paramterTypes = method.getParameterTypes();
26
27     Object[] paramValues = new Object[paramterTypes.length];
28     for (int i = 0, j = paramterTypes.length; i < j; i++) {
29         Class<?> parameterType = paramterTypes[i];
30         if (parameterType == HttpServletRequest.class) {
31             paramValues[i] = req;
32             continue;
33         } else if (parameterType == HttpServletResponse.class) {
34             paramValues[i] = resp;
35             continue;
36         } else if (parameterType == String.class) {
37             // 获取方法参数上的所有注解
38             Annotation[][] pa = method.getParameterAnnotations();
39             for (int m = 0, n = pa.length; m < n; m++) {
40                 // 遍历一个参数上的所有注解
41                 for (Annotation a : pa[i]) {
42                     // 该注解是否是MyRequestParam
43                     if (a instanceof MyRequestParam) {
44                         String paramName = ((MyRequestParam) a).value();
45                         if (!"".equals(paramName.trim())) {
46                             // 该value值形如: ["张三"]
47                             // "\\[|\\]" => 匹配 [ 和 ]
48                             // "\\s"  => 匹配任何空白字符, 包括空格、制表符、换页符
49                             String value =
50                                 Arrays.toString(paramsMap.get(paramName)).repla
51                                     "\\[|\\]", "").replaceAll("\\s", ",");
52                             paramValues[i] = value;
53                         }
54                     }
55                 }
56             }
57         }
58     }
59 }

```

```

55         }
56     }
57 }
58 }
59
60 // 调用目标方法
61 String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
62 method.invoke(ioc.get(beanName), paramValues);
63 }

```

完整代码如下所示:

```

1 package cn.sitedev.mvcframework.servlet.v2;
2
3 import cn.sitedev.mvcframework.annotation.*;
4
5 import javax.servlet.ServletConfig;
6 import javax.servlet.ServletException;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10 import java.io.File;
11 import java.io.IOException;
12 import java.io.InputStream;
13 import java.lang.annotation.Annotation;
14 import java.lang.reflect.Field;
15 import java.lang.reflect.Method;
16 import java.net.URL;
17 import java.util.*;
18
19 public class MyDispatcherServlet extends HttpServlet {
20     // 保存application.properties配置文件中的内容
21     private Properties contextConfig = new Properties();
22     // 保存扫描到的所有的类名
23     private List<String> classNames = new ArrayList<>();
24     // IOC容器
25     // 为了简化程序，我们暂不考虑ConcurrentHashMap
26     // 主要还是关注设计思想和原理
27     private Map<String, Object> ioc = new HashMap<>();
28
29     // 保存url和Method的对应关系

```

```

30     private Map<String, Method> handlerMapping = new HashMap<>();
31
32     @Override
33     public void init(ServletConfig config) throws ServletException {
34
35         // 1. 加载配置文件
36         doLoadConfig(config.getInitParameter("contextConfigLocation"));
37
38         // 2. 扫描相关的类
39         doScanner(contextConfig.getProperty("scanPackage"));
40
41         // 3. 初始化扫描到的类，并且放入到IOC容器之中
42         doInstance();
43
44         // 4. 完成自动化的依赖注入
45         doAutowired();
46
47         // 5. 初始化HandlerMapping
48         doInitHandlerMapping();
49
50         System.out.println("My Spring Framework is init...");
51     }
52
53     @Override
54     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
55         this.doPost(req, resp);
56     }
57
58     @Override
59     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
60         // 6. 根据url调用method
61         try {
62             doDispatch(req, resp);
63         } catch (Exception e) {
64             e.printStackTrace();
65             resp.getWriter().write("500 Exception, Detail: " + Arrays.toString(e.getStackTrace()));
66         }
67     }
68
69     private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Exception {
70         String url = req.getRequestURI();
71         String contextPath = req.getContextPath();
72         // 从请求url中截去context path

```

```

73 url = url.replaceAll(contextPath, "").replaceAll("/+", "/");
74
75 if (!this.handlerMapping.containsKey(url)) {
76     resp.getWriter().write("404 Not Found !");
77     return;
78 }
79
80 Method method = this.handlerMapping.get(url);
81
82 // 获取请求参数map
83 // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", value=["
84 Map<String, String[]> paramsMap = req.getParameterMap();
85
86 // String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName())
87 // 调用处理该请求url对应的方法
88 // method.invoke(ioc.get(beanName), new Object[]{req, resp, paramsMap.get("name"
89 //     paramsMap.get("id")[0]});
90
91 // 实参列表
92 // 实参列表要根据形参列表才能决定, 首先得拿到形参列表
93 Class<?>[] paramterTypes = method.getParameterTypes();
94
95 Object[] paramValues = new Object[paramterTypes.length];
96 for (int i = 0, j = paramterTypes.length; i < j; i++) {
97     Class<?> parameterType = paramterTypes[i];
98     if (parameterType == HttpServletRequest.class) {
99         paramValues[i] = req;
100         continue;
101     } else if (parameterType == HttpServletResponse.class) {
102         paramValues[i] = resp;
103         continue;
104     } else if (parameterType == String.class) {
105         // 获取方法参数上的所有注解
106         Annotation[][] pa = method.getParameterAnnotations();
107         for (int m = 0, n = pa.length; m < n; m++) {
108             // 遍历一个参数上的所有注解
109             for (Annotation a : pa[i]) {
110                 // 该注解是否是MyRequestParam
111                 if (a instanceof MyRequestParam) {
112                     String paramName = ((MyRequestParam) a).value();
113                     if (!"".equals(paramName.trim())) {
114                         // 该value值形如: ["张三"]
115                         // "\\[|\\]" => 匹配 [ 和 ]

```

```

116         // "\\s" => 匹配任何空白字符，包括空格、制表符、换页符
117         String value =
118             Arrays.toString(paramsMap.get(paramName)).replace(
119                 "\\[|\\]", "").replaceAll("\\s", ",");
120         paramValues[i] = value;
121     }
122 }
123 }
124 }
125 }
126 }
127
128 // 调用目标方法
129 String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
130 method.invoke(ioc.get(beanName), paramValues);
131 }
132
133 // 初始化url和方法的一对一对应关系
134 private void doInitHandlerMapping() {
135     if (ioc.isEmpty()) {
136         return;
137     }
138
139     // 遍历ioc容器中的bean
140     for (Map.Entry<String, Object> entry : ioc.entrySet()) {
141         Class<?> clazz = entry.getValue().getClass();
142         // 判断当前类上是否有MyController注解
143         if (!clazz.isAnnotationPresent(MyController.class)) {
144             continue;
145         }
146
147         // 保存写在类上面的@MyRequestMapping("/demo")
148         String baseUrl = "";
149         // 判断当前类上是否有MyRequestMapping注解
150         if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
151             // 获取MyRequestMapping注解
152             MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.class);
153             baseUrl = requestMapping.value();
154         }
155
156         // 默认获取所有的public方法
157         for (Method method : clazz.getMethods()) {
158             // 判断方法实例上是否有MyRequestMapping注解

```



```

159         if (!method.isAnnotationPresent(MyRequestMapping.class)) {
160             continue;
161         }
162         // 获取MyRequestMapping注解
163         MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping.class);
164
165         // 将类上的MyRequestMapping注解的value值与类方法上的MyRequestMapping注解的value值进行比较
166         // "demo" + "query" => "/demo/query"
167         // "/demo" + "/query" => "//demo//query" => "/demo/query"
168         String url = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll("//", "/");
169         // key:请求url, value: 处理对应url的method
170         handlerMapping.put(url, method);
171
172         System.out.println("Mapped : " + url + ", " + method);
173     }
174 }
175 }
176
177 private void doAutowired() {
178     if (ioc.isEmpty()) {
179         return;
180     }
181
182     for (Map.Entry<String, Object> entry : ioc.entrySet()) {
183         // 拿到实例的所有字段
184         // Declared 所有的, 特定的 字段, 包括private/protected/default
185         // 正常来说, 普通的OOP编程只能拿到public的属性
186         Field[] fields = entry.getValue().getClass().getDeclaredFields();
187         // 遍历类的所有字段, 并给指定注解的字段赋值
188         for (Field field : fields) {
189             // 判断当前字段上是否有MyAutowired注解
190             if (!field.isAnnotationPresent(MyAutowired.class)) {
191                 continue;
192             }
193             // 获取MyAutowired注解
194             MyAutowired autowired = field.getAnnotation(MyAutowired.class);
195             // 如果用户没有自定义beanName, 默认就根据类型注入
196             // 这个地方省去了对类名首字母小写的情况的判断
197             String beanName = autowired.value().trim();
198             if ("".equals(beanName)) {
199                 // 获得接口的类型, 作为key. 待会拿到这个key到ioc容器中去取值
200                 beanName = field.getType().getName();
201             }

```

```

202
203 // 如果是public以外的修饰符，只要加了@MyAutowired注解，都要强制赋值
204 // 反射中叫做暴力访问
205 field.setAccessible(true);
206
207 // 反射调用的方式
208 // 给entry.getValue()这个对象的field字段，赋ioc.get(beanName)这个值
209 try {
210     field.set(entry.getValue(), ioc.get(beanName));
211 } catch (IllegalAccessException e) {
212     e.printStackTrace();
213     continue;
214 }
215
216 }
217
218 }
219 }
220
221 private void doInstance() {
222     if (classNames.isEmpty()) {
223         return;
224     }
225     try {
226         for (String className : classNames) {
227             Class<?> clazz = Class.forName(className);
228
229             // 什么样的类才需要初始化呢?
230             // 加了注解的类，才会初始化，怎么判断?
231             // 为了简化代码逻辑，主要体会设计思想，只举例@Controller和服务,
232             // @Component... 就不一一举例了
233
234             // 如果该类上有MyController注解
235             if (clazz.isAnnotationPresent(MyController.class)) {
236                 Object instance = clazz.newInstance();
237                 String beanName = toLowerFirstCase(clazz.getSimpleName());
238                 // key:类名的首字母小写，value:类的实例
239                 ioc.put(beanName, instance);
240             }
241             // 如果该类上有MyService注解
242             else if (clazz.isAnnotationPresent(MyService.class)) {
243                 // 1. 默认就根据beanName类名首字母小写
244                 String beanName = toLowerFirstCase(clazz.getSimpleName());

```

```

245
246         // 2. 使用自定义的beanName
247         MyService service = clazz.getAnnotation(MyService.class);
248         if (!"".equals(service.value())) {
249             beanName = service.value();
250         }
251
252         Object instance = clazz.newInstance();
253         // key:类名的首字母小写或自定义的名称, value:类的实例
254         ioc.put(beanName, instance);
255
256         // 3. 根据包名.类名作为beanName
257         for (Class<?> i : clazz.getInterfaces()) {
258             if (ioc.containsKey(i.getName())) {
259                 throw new Exception("The beanName is exists");
260             }
261             // 把接口的类型直接当成key了
262             // key:接口类型, value:类的实例
263             ioc.put(i.getName(), instance);
264         }
265     } else {
266         continue;
267     }
268 }
269 } catch (Exception e) {
270     e.printStackTrace();
271 }
272 }
273
274 // 如果类名本身是小写字母, 确实会出问题
275 // 但是我要说明的是: 这个方法是我自己用的, private类型的
276 // 传值也是自己传, 存在首字母小写的情况, 也不可能出现非字母的情况
277 // 为了简化程序逻辑, 就不做其他判断了
278 private String toLowerFirstCase(String simpleName) {
279     char[] chars = simpleName.toCharArray();
280     // 之所以采用 += , 是因为大小写字母的ASCII码值相差32
281     // 而且大写字母的ASCII码要小于小写字母的
282     // 在Java中, 对char做数学运算, 实际上就是对ASCII码做数学运算
283     chars[0] += 32;
284     return String.valueOf(chars);
285 }
286
287 // 扫描相关的类

```

```

288     private void doScanner(String scanPackage) {
289         URL url = this.getClass().getClassLoader().getResource("/") + scanPackage.replac
290             , "/"));
291         // scanPackage=cn.sitedev.demo, 存储的就是扫描的包路径
292         // 转换为文件路径, 实际上就是将.替换成/
293         // classpath下不仅有.class文件, .xml文件, .properties文件
294         File classPath = new File(url.getFile());
295         // 遍历该路径下的所有文件/文件夹
296         for (File file : classPath.listFiles()) {
297             if (file.isDirectory()) {
298                 // 递归遍历
299                 doScanner(scanPackage + "." + file.getName());
300             } else {
301                 // 变成 包名.类名
302                 if (!file.getName().endsWith(".class")) {
303                     continue;
304                 }
305                 classNames.add(scanPackage + "." + file.getName().replace(".class", ""))
306             }
307         }
308     }
309
310     // 加载配置文件
311     private void doLoadConfig(String contextConfigLocation) {
312         // 直接从类路径下找到Spring主配置文件所在的路径
313         // 并且将其读取出来放到Properties对象中
314         // 相对于scanPackage=cn.sitedev.demo从文件中保存到了内存中
315         InputStream inputStream =
316             this.getClass().getClassLoader().getResourceAsStream(contextConfigLocat
317         try {
318             contextConfig.load(inputStream);
319         } catch (IOException e) {
320             e.printStackTrace();
321         } finally {
322             // 释放资源
323             if (inputStream != null) {
324                 try {
325                     inputStream.close();
326                 } catch (IOException e) {
327                     e.printStackTrace();
328                 }
329             }
330         }

```

```
331     }
332 }
```

3.3. 实现V3版本

在V2版本中，基本功能以及完全实现，但代码的优雅程度还不如人意。譬如HandlerMapping 还不能像 SpringMVC一样支持正则，url参数还不支持强制类型转换，在反射调用前还需要重新获取 beanName，在V3版本中，下面我们继续优化。

首先，改造HandlerMapping，在真实的Spring源码中，HandlerMapping 其实是一个List而非 Map。List中的元素是一个自定义的类型。现在我们来仿真写一段代码，先定义一个内部类Handler 类：

```
1      /**
2       * Handler记录Controller中的RequestMapping和Method的关系
3       */
4      @Data
5      private class Handler {
6          // 保存方法对应的实例
7          protected Object controller;
8          // 保存映射的方法
9          protected Method method;
10         // ${} url占位符解析
11         protected Pattern pattern;
12         // 参数顺序
13         protected Map<String, Integer> paramIndexMapping;
14
15         /**
16          * 构造一个Handler基本的参数
17          *
18          * @param pattern
19          * @param controller
20          * @param method
21          */
22         public Handler(Pattern pattern, Object controller, Method method) {
23             this.pattern = pattern;
24             this.controller = controller;
25             this.method = method;
26             this.paramIndexMapping = new HashMap<>();
27             putParamIndexMapping(method);
28         }
```

```

29
30     private void putParamIndexMapping(Method method) {
31         // 提取方法中加了注解的参数
32         Annotation[][] pa = method.getParameterAnnotations();
33         for (int i = 0; i < pa.length; i++) {
34             for (Annotation a : pa[i]) {
35                 if (a instanceof MyRequestParam) {
36                     String paramName = ((MyRequestParam) a).value();
37                     if (!"".equals(paramName.trim())) {
38                         paramIndexMapping.put(paramName, i);
39                     }
40                 }
41             }
42         }
43
44         // 提取方法中的request和response参数
45         Class<?>[] paramTypes = method.getParameterTypes();
46         for (int i = 0; i < paramTypes.length; i++) {
47             Class<?> type = paramTypes[i];
48             if (type == HttpServletRequest.class || type == HttpServletResponse.class) {
49                 paramIndexMapping.put(type.getName(), i);
50             }
51         }
52
53     }
54 }

```

然后，优化HandlerMapping的结构，代码如下：

```

1     // 保存url和Method的对应关系
2     private List<Handler> handlerMapping = new ArrayList<>();

```

修改doInitHandlerMapping()方法：

```

1     // 初始化url和Method的一一对应关系
2     private void doInitHandlerMapping() {
3         if (ioc.isEmpty()) {
4             return;
5         }
6

```

```

7      // 遍历ioc容器中的bean
8      for (Map.Entry<String, Object> entry : ioc.entrySet()) {
9          Class<?> clazz = entry.getValue().getClass();
10         // 判断当前类上是否有MyController注解
11         if (!clazz.isAnnotationPresent(MyController.class)) {
12             continue;
13         }
14
15         // 保存写在类上面的@MyRequestMapping("/demo")
16         String baseUrl = "";
17         // 判断当前Controller类上是否有MyRequestMapping注解
18         if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
19             // 获取MyRequestMapping注解
20             MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.class);
21             baseUrl = requestMapping.value();
22         }
23
24         // 默认获取所有的public方法
25         for (Method method : clazz.getMethods()) {
26             // 判断Method方法实例上是否有MyRequestMapping注解
27             // 没有的直接忽略
28             if (!method.isAnnotationPresent(MyRequestMapping.class)) {
29                 continue;
30             }
31             // 获取MyRequestMapping注解
32             MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping.class);
33
34             // 映射URL
35             String regex = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll("\\*", "");
36             Pattern pattern = Pattern.compile(regex);
37             handlerMapping.add(new Handler(pattern, entry.getValue(), method));
38             System.out.println("mapping : " + regex + ", " + method);
39         }
40     }
41 }

```

修改doDispatch()方法：

```

1      /**
2       * 匹配URL
3       *

```

```

4      * @param req
5      * @param resp
6      * @throws Exception
7      */
8      private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
9          try {
10              Handler handler = getHandler(req);
11
12              if (handler == null) {
13                  // 如果没有匹配上，就返回404错误
14                  resp.getWriter().write("404 Not Found");
15                  return;
16              }
17
18              // 获取方法的参数列表
19              Class<?>[] paramTypes = handler.method.getParameterTypes();
20
21              // 保存所有需要自动赋值的参数值
22              Object[] paramValues = new Object[paramTypes.length];
23
24              // 获取请求参数map
25              // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", valu
26              Map<String, String[]> params = req.getParameterMap();
27
28              for (Map.Entry<String, String[]> param : params.entrySet()) {
29                  String value =
30                      Arrays.toString(param.getValue()).replaceAll("\\[\\]", "").rep
31                      "\\s", ",");
32
33                  // 如果找到匹配的对象，则开始填充参数值
34                  if (!handler.paramIndexMapping.containsKey(param.getKey())) {
35                      continue;
36                  }
37                  int index = handler.paramIndexMapping.get(param.getKey());
38                  paramValues[index] = convert(paramTypes[index], value);
39              }
40
41              // 设置方法中的request和response对象
42              int reqIndex = handler.paramIndexMapping.get(HttpServletRequest.class.getNa
43              paramValues[reqIndex] = req;
44              int respIndex = handler.paramIndexMapping.get(HttpServletResponse.class.getNa
45              paramValues[respIndex] = resp;
46

```



```

47         handler.method.invoke(handler.controller, paramValues);
48
49     } catch (Exception e) {
50         throw e;
51     }
52 }
53
54 // url传过来的参数都是String类型的，http是基于字符串协议的
55 // 只需要把String转换为任意类型就好
56 private Object convert(Class<?> paramType, String value) {
57     if (paramType == Integer.class) {
58         return Integer.valueOf(value);
59     }
60     // 如果还有double或者其他类型，继续加if条件
61     // 这个时候，我们应该想到策略模式
62     // 这里我们暂时不实现
63     return value;
64 }
65
66 private Handler getHandler(HttpServletRequest req) {
67     if (handlerMapping.isEmpty()) {
68         return null;
69     }
70
71     String url = req.getRequestURI();
72     String contextPath = req.getContextPath();
73     // 从url中截去context path
74     url = url.replace(contextPath, "").replaceAll("/+", "/");
75
76     // 从handlerMapping中找到一个合适的handler来处理当前url
77     for (Handler handler : handlerMapping) {
78         try {
79             Matcher matcher = handler.pattern.matcher(url);
80             // 如果没有匹配上就继续下一个匹配
81             if (!matcher.matches()) {
82                 continue;
83             }
84             return handler;
85         } catch (Exception e) {
86             throw e;
87         }
88     }
89     return null;

```

```
90     }
```

在以上代码中，增加了两个方法，一个是getHandler()方法，主要负责处理url的正则匹配；一个是convert()方法，主要负责url参数的强制类型转换。至此，手写Mini版 SpringMVC框架就已全部完成。

完整代码如下：

```
1 package cn.sitedev.mvcframework.servlet.v3;
2
3 import cn.sitedev.mvcframework.annotation.*;
4 import lombok.Data;
5
6 import javax.servlet.ServletConfig;
7 import javax.servlet.ServletException;
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11 import java.io.File;
12 import java.io.IOException;
13 import java.io.InputStream;
14 import java.lang.annotation.Annotation;
15 import java.lang.reflect.Field;
16 import java.lang.reflect.Method;
17 import java.net.URL;
18 import java.util.*;
19 import java.util.regex.Matcher;
20 import java.util.regex.Pattern;
21
22 public class MyDispatcherServlet extends HttpServlet {
23     // 保存application.properties配置文件中的内容
24     private Properties contextConfig = new Properties();
25     // 保存扫描到的所有的类名
26     private List<String> classNames = new ArrayList<>();
27     // IOC容器
28     // 为了简化程序，我们暂不考虑ConcurrentHashMap
29     // 主要还是关注设计思想和原理
30     private Map<String, Object> ioc = new HashMap<>();
31
32     // 保存url和Method的对应关系
33     private List<Handler> handlerMapping = new ArrayList<>();
34 }
```

```

35     @Override
36     public void init(ServletConfig config) throws ServletException {
37
38         // 1. 加载配置文件
39         doLoadConfig(config.getInitParameter("contextConfigLocation"));
40
41         // 2. 扫描相关的类
42         doScanner(contextConfig.getProperty("scanPackage"));
43
44         // 3. 初始化扫描到的类，并且放入到IOC容器之中
45         doInstance();
46
47         // 4. 完成自动化的依赖注入
48         doAutowired();
49
50         // 5. 初始化HandlerMapping
51         doInitHandlerMapping();
52
53         System.out.println("My Spring Framework is init...");
54     }
55
56     @Override
57     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
58         this.doPost(req, resp);
59     }
60
61     @Override
62     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException {
63         // 6. 根据url调用method
64         try {
65             doDispatch(req, resp);
66         } catch (Exception e) {
67             e.printStackTrace();
68             resp.getWriter().write("500 Exception, Detail: " + Arrays.toString(e.getStackTrace()));
69         }
70     }
71
72     /**
73      * 匹配URL
74      *
75      * @param req
76      * @param resp
77      * @throws Exception

```

```

78      */
79      private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
80          try {
81              Handler handler = getHandler(req);
82
83              if (handler == null) {
84                  // 如果没有匹配上，就返回404错误
85                  resp.getWriter().write("404 Not Found");
86                  return;
87              }
88
89              // 获取方法的参数列表
90              Class<?>[] paramTypes = handler.method.getParameterTypes();
91
92              // 保存所有需要自动赋值的参数值
93              Object[] paramValues = new Object[paramTypes.length];
94
95              // 获取请求参数map
96              // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", valu
97              Map<String, String[]> params = req.getParameterMap();
98
99              for (Map.Entry<String, String[]> param : params.entrySet()) {
100                  String value =
101                      Arrays.toString(param.getValue()).replaceAll("\\[|\\]", "").rep
102                      "\\s", ",");
103
104                  // 如果找到匹配的对象，则开始填充参数值
105                  if (!handler.paramIndexMapping.containsKey(param.getKey())) {
106                      continue;
107                  }
108                  int index = handler.paramIndexMapping.get(param.getKey());
109                  paramValues[index] = convert(paramTypes[index], value);
110              }
111
112              // 设置方法中的request和response对象
113              int reqIndex = handler.paramIndexMapping.get(HttpServletRequest.class.getName());
114              paramValues[reqIndex] = req;
115              int respIndex = handler.paramIndexMapping.get(HttpServletResponse.class.getName());
116              paramValues[respIndex] = resp;
117
118              handler.method.invoke(handler.controller, paramValues);
119
120          } catch (Exception e) {

```

```
121         throw e;
122     }
123 }
124
125 // url传过来的参数都是String类型的，http是基于字符串协议的
126 // 只需要把String转换为任意类型就好
127 private Object convert(Class<?> paramType, String value) {
128     if (paramType == Integer.class) {
129         return Integer.valueOf(value);
130     }
131     // 如果还有double或者其他类型，继续加if条件
132     // 这个时候，我们应该想到策略模式
133     // 这里我们暂时不实现
134     return value;
135 }
136
137 private Handler getHandler(HttpServletRequest req) {
138     if (handlerMapping.isEmpty()) {
139         return null;
140     }
141
142     String url = req.getRequestURI();
143     String contextPath = req.getContextPath();
144     // 从url中截去context path
145     url = url.replace(contextPath, "").replaceAll("/+", "/");
146
147     // 从handlerMapping中找到一个合适的handler来处理当前url
148     for (Handler handler : handlerMapping) {
149         try {
150             Matcher matcher = handler.pattern.matcher(url);
151             // 如果没有匹配上就继续下一个匹配
152             if (!matcher.matches()) {
153                 continue;
154             }
155             return handler;
156         } catch (Exception e) {
157             throw e;
158         }
159     }
160     return null;
161 }
162
163 // 初始化url和方法的一一对应关系
```

```

164     private void doInitHandlerMapping() {
165         if (ioc.isEmpty()) {
166             return;
167         }
168
169         // 遍历ioc容器中的bean
170         for (Map.Entry<String, Object> entry : ioc.entrySet()) {
171             Class<?> clazz = entry.getValue().getClass();
172             // 判断当前类上是否有MyController注解
173             if (!clazz.isAnnotationPresent(MyController.class)) {
174                 continue;
175             }
176
177             // 保存写在类上面的@MyRequestMapping("/demo")
178             String baseUrl = "";
179             // 判断当前Controller类上是否有MyRequestMapping注解
180             if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
181                 // 获取MyRequestMapping注解
182                 MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.class);
183                 baseUrl = requestMapping.value();
184             }
185
186             // 默认获取所有的public方法
187             for (Method method : clazz.getMethods()) {
188                 // 判断Method方法实例上是否有MyRequestMapping注解
189                 // 没有的直接忽略
190                 if (!method.isAnnotationPresent(MyRequestMapping.class)) {
191                     continue;
192                 }
193                 // 获取MyRequestMapping注解
194                 MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping.class);
195
196                 // 映射URL
197                 String regex = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll("\\\\", "\\");
198                 Pattern pattern = Pattern.compile(regex);
199                 handlerMapping.add(new Handler(pattern, entry.getValue(), method));
200                 System.out.println("mapping : " + regex + ", " + method);
201             }
202         }
203     }
204
205     private void doAutowired() {
206         if (ioc.isEmpty()) {

```

```

207         return;
208     }
209
210     for (Map.Entry<String, Object> entry : ioc.entrySet()) {
211         // 拿到实例的所有字段
212         // Declared 所有的，特定的 字段，包括private/protected/default
213         // 正常来说，普通的OOP编程只能拿到public的属性
214         Field[] fields = entry.getValue().getClass().getDeclaredFields();
215         // 遍历类的所有字段，并给指定注解的字段赋值
216         for (Field field : fields) {
217             // 判断当前字段上是否有MyAutowired注解
218             if (!field.isAnnotationPresent(MyAutowired.class)) {
219                 continue;
220             }
221             // 获取MyAutowired注解
222             MyAutowired autowired = field.getAnnotation(MyAutowired.class);
223             // 如果用户没有自定义beanName，默认就根据类型注入
224             // 这个地方省去了对类名首字母小写的情况的判断
225             String beanName = autowired.value().trim();
226             if ("".equals(beanName)) {
227                 // 获得接口的类型，作为key。待会拿到这个key到ioc容器中去取值
228                 beanName = field.getType().getName();
229             }
230
231             // 如果是public以外的修饰符，只要加了@MyAutowired注解，都要强制赋值
232             // 反射中叫做暴力访问
233             field.setAccessible(true);
234
235             // 反射调用的方式
236             // 给entry.getValue()这个对象的field字段，赋ioc.get(beanName)这个值
237             try {
238                 field.set(entry.getValue(), ioc.get(beanName));
239             } catch (IllegalAccessException e) {
240                 e.printStackTrace();
241                 continue;
242             }
243         }
244     }
245
246 }
247
248
249 private void doInstance() {

```

```
250     if (classNames.isEmpty()) {
251         return;
252     }
253     try {
254         for (String className : classNames) {
255             Class<?> clazz = Class.forName(className);
256
257             // 什么样的类才需要初始化呢?
258             // 加了注解的类, 才会初始化, 怎么判断?
259             // 为了简化代码逻辑, 主要体会设计思想, 只举例@Controller和服务,
260             // @Component... 就不一一举例了
261
262             // 如果该类上有MyController注解
263             if (clazz.isAnnotationPresent(MyController.class)) {
264                 Object instance = clazz.newInstance();
265                 String beanName = toLowerFirstCase(clazz.getSimpleName());
266                 // key:类名的首字母小写, value:类的实例
267                 ioc.put(beanName, instance);
268             }
269             // 如果该类上有MyService注解
270             else if (clazz.isAnnotationPresent(MyService.class)) {
271                 // 1. 默认就根据beanName类名首字母小写
272                 String beanName = toLowerFirstCase(clazz.getSimpleName());
273
274                 // 2. 使用自定义的beanName
275                 MyService service = clazz.getAnnotation(MyService.class);
276                 if (!"".equals(service.value())) {
277                     beanName = service.value();
278                 }
279
280                 Object instance = clazz.newInstance();
281                 // key:类名的首字母小写或自定义的名称, value:类的实例
282                 ioc.put(beanName, instance);
283
284                 // 3. 根据包名.类名作为beanName
285                 for (Class<?> i : clazz.getInterfaces()) {
286                     if (ioc.containsKey(i.getName())) {
287                         throw new Exception("The beanName is exists");
288                     }
289                     // 把接口的类型直接当成key了
290                     // key:接口类型, value:类的实例
291                     ioc.put(i.getName(), instance);
292                 }

```



```

293         } else {
294             continue;
295         }
296     }
297     } catch (Exception e) {
298         e.printStackTrace();
299     }
300 }
301
302 // 如果类名本身是小写字母，确实会出问题
303 // 但是我要说明的是：这个方法是我自己用的，private类型的
304 // 传值也是自己传，存在首字母小写的情况，也不可能出现非字母的情况
305 // 为了简化程序逻辑，就不做其他判断了
306 private String toLowerFirstCase(String simpleName) {
307     char[] chars = simpleName.toCharArray();
308     // 之所以采用 += ，是因为大小写字母的ASCII码值相差32
309     // 而且大写字母的ASCII码要小于小写字母的
310     // 在Java中，对char做数学运算，实际上就是对ASCII码做数学运算
311     chars[0] += 32;
312     return String.valueOf(chars);
313 }
314
315 // 扫描相关的类
316 private void doScanner(String scanPackage) {
317     URL url = this.getClass().getClassLoader().getResource("/" + scanPackage.replace(
318         , "/"));
319     // scanPackage=cn.sitedev.demo，存储的就是扫描的包路径
320     // 转换为文件路径，实际上就是将.替换成/
321     // classpath下不仅有.class文件，.xml文件，.properties文件
322     File classPath = new File(url.getFile());
323     // 遍历该路径下的所有文件/文件夹
324     for (File file : classPath.listFiles()) {
325         if (file.isDirectory()) {
326             // 递归遍历
327             doScanner(scanPackage + "." + file.getName());
328         } else {
329             // 变成 包名.类名
330             if (!file.getName().endsWith(".class")) {
331                 continue;
332             }
333             classNames.add(scanPackage + "." + file.getName().replace(".class", ""))
334         }
335     }

```

```

336     }
337
338     // 加载配置文件
339     private void doLoadConfig(String contextConfigLocation) {
340         // 直接从类路径下找到Spring主配置文件所在的路径
341         // 并且将其读取出来放到Properties对象中
342         // 相对于scanPackage=cn.sitedev.demo从文件中保存到了内存中
343         InputStream inputStream =
344             this.getClass().getClassLoader().getResourceAsStream(contextConfigLocation);
345         try {
346             contextConfig.load(inputStream);
347         } catch (IOException e) {
348             e.printStackTrace();
349         } finally {
350             // 释放资源
351             if (inputStream != null) {
352                 try {
353                     inputStream.close();
354                 } catch (IOException e) {
355                     e.printStackTrace();
356                 }
357             }
358         }
359     }
360
361     /**
362     * Handler记录Controller中的RequestMapping和Method的关系
363     */
364     @Data
365     private class Handler {
366         // 保存方法对应的实例
367         protected Object controller;
368         // 保存映射的方法
369         protected Method method;
370         // ${} url占位符解析
371         protected Pattern pattern;
372         // 参数顺序
373         protected Map<String, Integer> paramIndexMapping;
374
375         /**
376         * 构造一个Handler基本的参数
377         *
378         * @param pattern

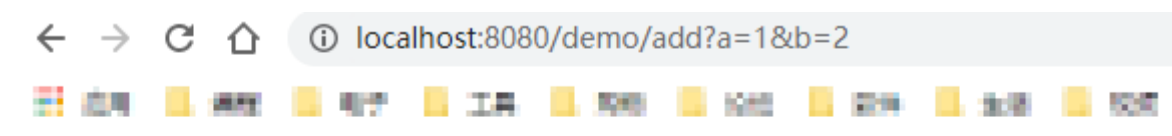
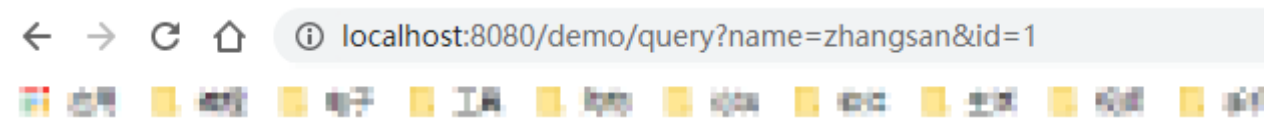
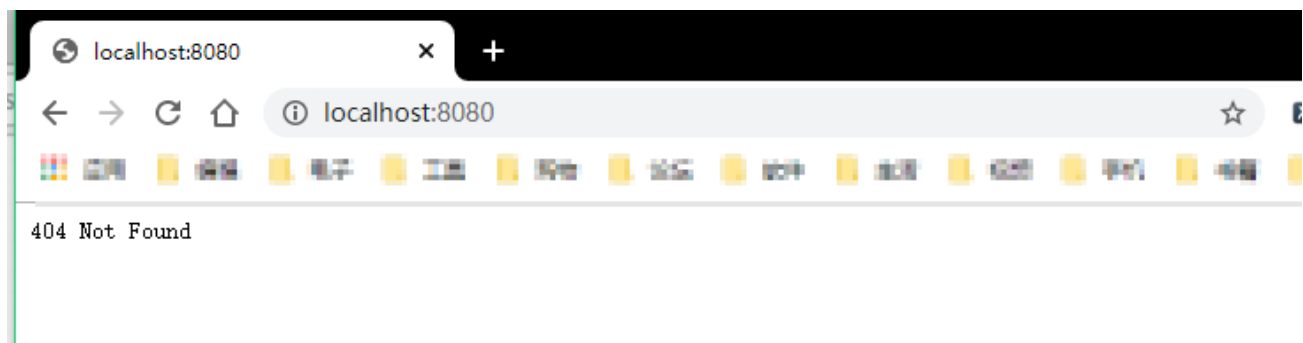
```

```

379     * @param controller
380     * @param method
381     */
382     public Handler(Pattern pattern, Object controller, Method method) {
383         this.pattern = pattern;
384         this.controller = controller;
385         this.method = method;
386         this.paramIndexMapping = new HashMap<>();
387         putParamIndexMapping(method);
388     }
389
390     private void putParamIndexMapping(Method method) {
391         // 提取方法中加了注解的参数
392         Annotation[][] pa = method.getParameterAnnotations();
393         for (int i = 0; i < pa.length; i++) {
394             for (Annotation a : pa[i]) {
395                 if (a instanceof MyRequestParam) {
396                     String paramName = ((MyRequestParam) a).value();
397                     if (!"".equals(paramName.trim())) {
398                         paramIndexMapping.put(paramName, i);
399                     }
400                 }
401             }
402         }
403
404         // 提取方法中的request和response参数
405         Class<?>[] paramTypes = method.getParameterTypes();
406         for (int i = 0; i < paramTypes.length; i++) {
407             Class<?> type = paramTypes[i];
408             if (type == HttpServletRequest.class || type == HttpServletResponse.class) {
409                 paramIndexMapping.put(type.getName(), i);
410             }
411         }
412     }
413 }
414 }
415 }

```

运行效果演示



当然，真正的Spring要复杂很多，本课中主要通过手写的形式，了解Spring的基本设计思路以及设计模式如何应用，在以后的课程中，我们还会继续手写更加高仿真版本的Spring2.0。