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

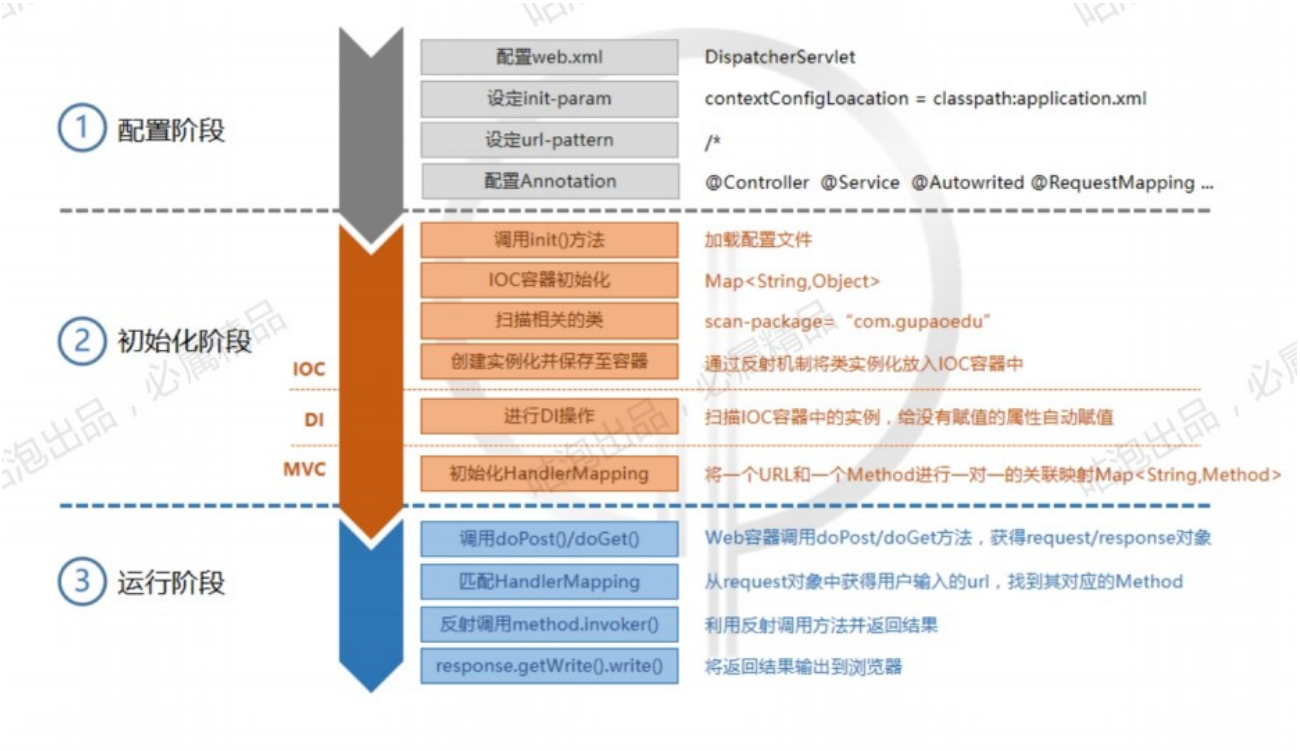
1、了解看源码最有效的方式，先猜测后验证，不要一开始就去调试代码。
2、浓缩就是精华，用300行最简洁的代码提炼Spring的基本设计思想。
3、掌握Spring 框架的基本脉络。

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：

```
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。

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

```
        <servlet-class>cn.sitedev.mvcframework.v2.servlet.MyDispatchServlet</servlet-cl
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath*:application.properties</param-value>
        </init-param>

        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>mvc</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</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：

```
package cn.sitedev.mvcframework.annotation;

import java.lang.annotation.*;

@Target(ElementType.TYPE) // 类/接口/枚举
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyService {
    String value() default "";
}
```

@MyAutowired 注解：

```
package cn.sitedev.mvcframework.annotation;

import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

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

@MyController 注解：

```
package cn.sitedev.mvcframework.annotation;

import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyController {
    String value() default "";
}
```

@MyRequestMapping 注解：

```
package cn.sitedev.mvcframework.annotation;

import java.lang.annotation.*;

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface MyRequestMapping {
    String value() default "";
}
```

@MyRequestParam 注解：

```
package cn.sitedev.mvcframework.annotation;

import java.lang.annotation.*;

@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
```

```
@Documented
public @interface MyRequestParam {
    String value() default "";
}
```

2.4. 配置Annotation

配置业务实现类DemoService：

```
package cn.sitedev.demo.service.impl;

import cn.sitedev.demo.service.IDemoService;
import cn.sitedev.mvcframework.annotation.MyService;

/**
 * 核心业务逻辑
 */
@MyService
public class DemoService implements IDemoService {
    @Override
    public String get(String name) {
        return "My name is " + name;
    }
}
```

配置请求入口类DemoAction：

```
package cn.sitedev.demo.mvc.action;

import cn.sitedev.demo.service.IDemoService;
import cn.sitedev.mvcframework.annotation.MyAutowired;
import cn.sitedev.mvcframework.annotation.MyController;
import cn.sitedev.mvcframework.annotation.MyRequestMapping;
import cn.sitedev.mvcframework.annotation.MyRequestParam;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

```

```java
@MyController
@MyRequestMapping("/demo")
public class DemoAction {
    @MyAutowired
    private IDemoService demoService;
    private IDemoService demoService2;

    @MyRequestMapping("/query")
    public void query(HttpServletRequest request, HttpServletResponse response, @MyRequ
            "name") String name, @MyRequestParam("id") String id) {
        String result = demoService.get(name);
        try {
            response.getWriter().write(result);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @MyRequestMapping("/add")
    public void add(HttpServletRequest request, HttpServletResponse response,
                    @MyRequestParam("a") Integer a, @MyRequestParam("b") Integer b) {
        try {
            response.getWriter().write(a + "+" + b + "=" + (a + b));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @MyRequestMapping("/remove")
    public void remove(HttpServletRequest request, HttpServletResponse response, @MyReq
            "id") Integer id) {

    }

}
```

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。

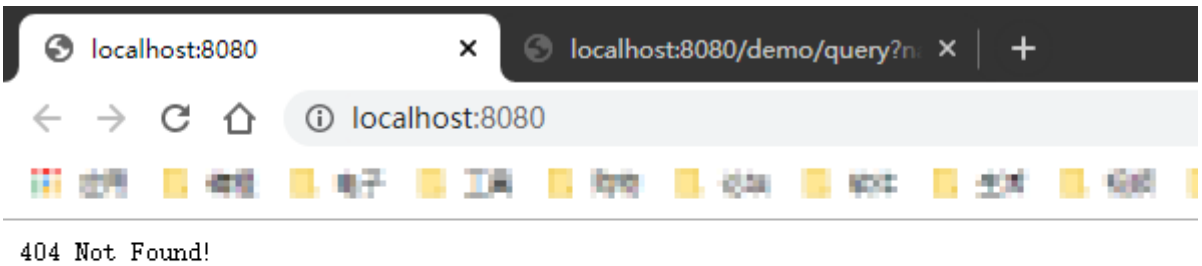
```
package cn.sitedev.mvcframework.servlet.v1;

import cn.sitedev.mvcframework.annotation.MyAutowired;
import cn.sitedev.mvcframework.annotation.MyController;
import cn.sitedev.mvcframework.annotation.MyRequestMapping;
import cn.sitedev.mvcframework.annotation.MyService;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class MyDispatcherServlet extends HttpServlet {
    private Map<String, Object> mapping = new HashMap<>();

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws Servl
        this.doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOEx
        try {
            doDispatch(req, resp);
        } catch (Exception e) {
            resp.getWriter().write("500 Exception " + e.getMessage());
        }
    }

    private void doDispatch(HttpServletRequest request, HttpServletResponse response) t
```

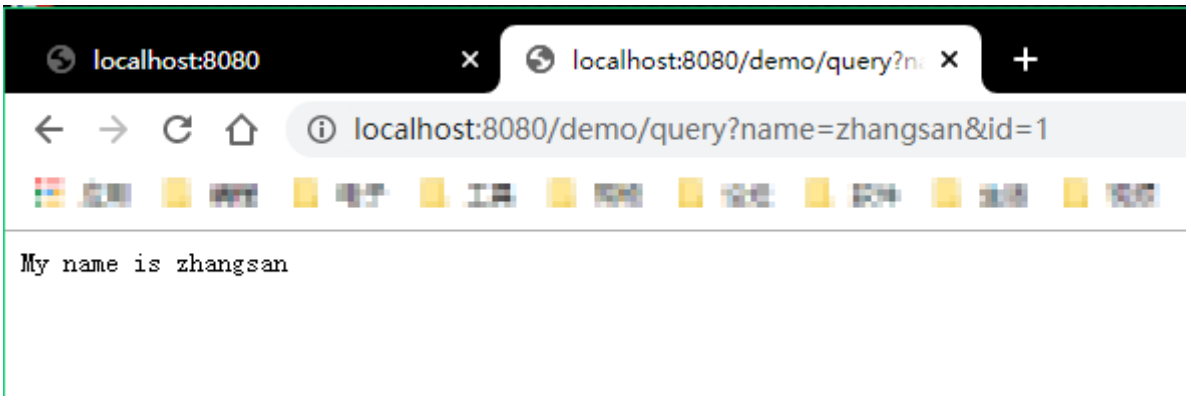
```
        // 获取请求url
        String url = request.getRequestURI();
        // 获取context path
        String contextPath = request.getContextPath();
        // 从请求url中截去context path
        url = url.replace(contextPath, "").replaceAll("/+", "/");
        // 判断是否有对应的方法来处理该url对应的请求
        if (!this.mapping.containsKey(url)) {
            response.getWriter().write("404 Not Found!");
            return;
        }
        // 获取该请求url对应的方法对象
        Method method = (Method) this.mapping.get(url);
        //获取请求参数map, 该map形如: "name"={key="name",value=["zhangsan"]}, "id"={key
        Map<String, String[]> params = request.getParameterMap();
        // 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
        method.invoke(this.mapping.get(method.getDeclaringClass().getName()),
                new Object[]{request, response, params.get("name")[0], params.get("id")
    }

    @Override
    public void init(ServletConfig config) throws ServletException {
        InputStream inputStream = null;
        try {
            // 读取配置文件application.properties
            Properties configContext = new Properties();
            inputStream =
                    this.getClass().getClassLoader().getResourceAsStream(config.getInit
                            "contextConfigLocation"));
            configContext.load(inputStream);
            // 获取scanPackage配置
            String scanPackage = configContext.getProperty("scanPackage");
            // 递归扫描指定包路径下的子包以及类
            doScanner(scanPackage);

            Map<String, Object> extMapping = new HashMap<>();
            // 遍历指定包路径下的类
            for (String className : mapping.keySet()) {
                if (!className.contains(".")) {
                    continue;
                }
                // 加载指定类
                Class<?> clazz = Class.forName(className);
```

```
                // 判断该类上是否有MyController注解
                if (clazz.isAnnotationPresent(MyController.class)) {
                    // key: 类的完全限定名 value:该类对应实例
                    extMapping.put(className, clazz.newInstance());
                    String baseUrl = "";
                    // 判断该类上是否有MyRequestMapping注解
                    if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
                        MyRequestMapping requestMapping =
                                clazz.getAnnotation(MyRequestMapping.class);
                        // 获取MyRequestMapping注解的value值
                        baseUrl = requestMapping.value();
                    }
                    // 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
                    Method[] methods = clazz.getMethods();
                    // 遍历所有方法对象
                    for (Method method : methods) {
                        // 判断方法上是否有MyRequestMapping注解
                        if (!method.isAnnotationPresent(MyRequestMapping.class)) {
                            continue;
                        }
                        // 获取MyRequestMapping注解的value值
                        MyRequestMapping requestMapping =
                                method.getAnnotation(MyRequestMapping.class);
                        // 拼接请求的url(如果存在多个/, 则替换成一个/)
                        String url = (baseUrl + "/" + requestMapping.value()).replaceAl
                        // key: 请求url, value: 处理对应url的方法对象
                        extMapping.put(url, method);
                        System.out.println("Mapped : " + url + ", " + method);

                    }
                    // 判断类上是否有MyService注解
                } else if (clazz.isAnnotationPresent(MyService.class)) {
                    // 获取MyService注解的value值
                    MyService service = clazz.getAnnotation(MyService.class);
                    String beanName = service.value();
                    if ("".equals(beanName)) {
                        beanName = clazz.getName();
                    }
                    Object instance = clazz.newInstance();
                    // key:类的完全限定名, value:类的实例
                    extMapping.put(beanName, instance);
                    for (Class<?> i : clazz.getInterfaces()) {
                        // key: 类实现的接口的完全限定名, value:类的实例
```

```
                        extMapping.put(i.getName(), instance);
                    }
                } else {
                    continue;
                }
            }

            mapping.putAll(extMapping);
            for (Object object : mapping.values()) {
                if (object == null) {
                    continue;
                }
                Class<?> clazz = object.getClass();
                // 判断类上是否有MyController注解
                if (clazz.isAnnotationPresent(MyController.class)) {
                    // 获取类所有的字段
                    Field[] fields = clazz.getDeclaredFields();
                    // 遍历类的所有字段
                    for (Field field : fields) {
                        // 判断当前字段上是否有MyAutowired注解
                        if (!field.isAnnotationPresent(MyAutowired.class)) {
                            continue;
                        }
                        // 获取MyAutowired注解的value值
                        MyAutowired autowired = field.getAnnotation(MyAutowired.class);
                        String beanName = autowired.value();
                        if ("".equals(beanName)) {
                            beanName = field.getType().getName();
                        }
                        // 使该字段可被访问
                        field.setAccessible(true);
                        // 给该字段设置值
                        field.set(mapping.get(clazz.getName()), mapping.get(beanName));
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            // 释放资源
            if (inputStream != null) {
                try {
                    inputStream.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        System.out.println("My MVC framework is init !");

    }

    private void doScanner(String scanPackage) {
        URL url = this.getClass().getClassLoader().getResource("/" + scanPackage.replac
                , "/"));
        File classDir = new File(url.getFile());
        for (File file : classDir.listFiles()) {
            if (file.isDirectory()) {
                // 递归扫描
                doScanner(scanPackage + "." + file.getName());
            } else {
                // 处理.class文件
                if (!file.getName().endsWith(".class")) {
                    continue;
                }
                // 存入mapping => key:类的完全限定名, value:null
                String clazzName = scanPackage + "." + file.getName().replace(".class",
                mapping.put(clazzName, null);
            }
        }
    }
}
```





3.2. 实现V2版本

在V1版本上进了优化，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：

```
package cn.sitedev.mvcframework.servlet.v2;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

public class MyDispatcherServlet extends HttpServlet {

    @Override
    public void init(ServletConfig config) throws ServletException {

        // 1. 加载配置文件
        doLoadConfig(config.getInitParameter("contextConfigLocation"));

        // 2. 扫描相关的类
        doScanner(contextConfig.getProperty("scanPackage"));

        // 3. 初始化扫描到的类，并且放入到IOC容器之中
        doInstance();

        // 4. 完成自动化的依赖注入
        doAutowired();

        // 5. 初始化HandlerMapping
        doInitHandlerMapping();

        System.out.println("My Spring Framework is init...");
    }
```

```
}
```

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：

```
    // 保存application.properties配置文件中的内容
    private Properties contextConfig = new Properties();
    // 保存扫描到的所有的类名
    private List<String> classNames = new ArrayList<>();
    // IOC容器
    // 为了简化程序，我们暂不考虑ConcurrentHashMap
    // 主要还是关注设计思想和原理
    private Map<String, Object> ioc = new HashMap<>();

    // 保存url和Method的对应关系
    private Map<String, Method> handlerMapping = new HashMap<>();
```

实现doLoadConfig()方法：

```
    // 加载配置文件
    private void doLoadConfig(String contextConfigLocation) {
        // 直接从类路径下找到Spring主配置文件所在的路径
        // 并且将其读取出来放到Properties对象中
        // 相对于scanPackage=cn.sitedev.demo从文件中保存到了内存中
        InputStream inputStream =
                this.getClass().getClassLoader().getResourceAsStream(contextConfigLocat
        try {
            contextConfig.load(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 释放资源
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

实现doScanner()方法：

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

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

```
    private void doInstance() {
        if (classNames.isEmpty()) {
            return;
        }
        try {
            for (String className : classNames) {
                Class<?> clazz = Class.forName(className);

                // 什么样的类才需要初始化呢?
                // 加了注解的类, 才会初始化, 怎么判断?
                // 为了简化代码逻辑, 主要体会设计思想, 只举例@Controller和Service,
                //@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：

```
    // 如果类名本身是小写字母，确实会出问题
    // 但是我要说明的是：这个方法是我自己用的，private类型的
    // 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
    // 为了简化程序逻辑，就不做其他判断了
    private String toLowerFirstCase(String simpleName) {
        char[] chars = simpleName.toCharArray();
        // 之所以采用 += ，是因为大小写字母的ASCII码值相差32
        // 而且大写字母的ASCII码要小于小写字母的
        // 在Java中，对char做数学运算，实际上就是对ASCII码做数学运算
        chars[0] += 32;
        return String.valueOf(chars);
    }
```

实现doAutowired()方法：

```
    private void doAutowired() {
        if (ioc.isEmpty()) {
            return;
        }

        for (Map.Entry<String, Object> entry : ioc.entrySet()) {
            // 拿到实例的所有字段
            // Declared 所有的，特定的 字段，包括private/protected/default
            // 正常来说，普通的OOP编程只能拿到public的属性
            Field[] fields = entry.getValue().getClass().getDeclaredFields();
            // 遍历类的所有字段，并给指定注解的字段赋值
            for (Field field : fields) {
                // 判断当前字段上是否有MyAutowired注解
                if (!field.isAnnotationPresent(MyAutowired.class)) {
                    continue;
                }
                // 获取MyAutowired注解
                MyAutowired autowired = field.getAnnotation(MyAutowired.class);
                // 如果用户没有自定义beanName，默认就根据类型注入
                // 这个地方省去了对类名首字母小写的情况的判断
                String beanName = autowired.value().trim();
                if ("".equals(beanName)) {
                    // 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
                    beanName = field.getType().getName();
                }
```

```
                // 如果是public以外的修饰符，只要加了@MyAutowired注解，都要强制赋值
                // 反射中叫做暴力访问
                field.setAccessible(true);

                // 反射调用的方式
                // 给entry.getValue()这个对象的field字段，赋ioc.get(beanName)这个值
                try {
                    field.set(entry.getValue(), ioc.get(beanName));
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                    continue;
                }

            }

        }
    }
```

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

```
    // 初始化url和Method的一对一对应关系
    private void doInitHandlerMapping() {
        if (ioc.isEmpty()) {
            return;
        }

        // 遍历ioc容器中的bean
        for (Map.Entry<String, Object> entry : ioc.entrySet()) {
            Class<?> clazz = entry.getValue().getClass();
            // 判断当前类上是否有MyController注解
            if (!clazz.isAnnotationPresent(MyController.class)) {
                continue;
            }

            // 保存写在类上面的@MyRequestMapping("/demo")
            String baseUrl = "";
            // 判断当前类上是否有MyRequestMapping注解
            if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
                // 获取MyRequestMapping注解
                MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.
```

```
                baseUrl = requestMapping.value();
            }

            // 默认获取所有的public方法
            for (Method method : clazz.getMethods()) {
                // 判断方法实例上是否有MyRequestMapping注解
                if (!method.isAnnotationPresent(MyRequestMapping.class)) {
                    continue;
                }
                // 获取MyRequestMapping注解
                MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping

                // 将类上的MyRequestMapping注解的value值与类方法上的MyRequestMapping注解
                // "demo" + "query"  =>  "/demo/query""
                // "/demo" + "/query" => "//demo//query" => "/demo/query""
                String url = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll(
                // key:请求url, value: 处理对应url的method
                handlerMapping.put(url, method);

                System.out.println("Mapped : " + url + ", " + method);
            }
        }
    }
```

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：

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws Servl
        this.doPost(req, resp);
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws Serv
        // 6. 根据url调用method
        try {
            doDispatch(req, resp);
        } catch (Exception e) {
            e.printStackTrace();
            resp.getWriter().write("500 Exception, Detail: " + Arrays.toString(e.getSta
        }
    }
```

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

```
    private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
        String url = req.getRequestURI();
        String contextPath = req.getContextPath();
        // 从请求url中截去context path
        url = url.replaceAll(contextPath, "").replaceAll("/+", "/");

        if (!this.handlerMapping.containsKey(url)) {
            resp.getWriter().write("404 Not Found !");
            return;
        }

        Method method = this.handlerMapping.get(url);

        // 获取请求参数map
        Map<String, String[]> paramsMap = req.getParameterMap();

        String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
        // 调用处理该请求url对应的方法
        method.invoke(ioc.get(beanName), new Object[]{req, resp, paramsMap.get("name")[
                paramsMap.get("id")[0]});
    }
```

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：

```
    private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
        String url = req.getRequestURI();
        String contextPath = req.getContextPath();
        // 从请求url中截去context path
        url = url.replaceAll(contextPath, "").replaceAll("/+", "/");

        if (!this.handlerMapping.containsKey(url)) {
            resp.getWriter().write("404 Not Found !");
            return;
        }

```

```
        Method method = this.handlerMapping.get(url);

        // 获取请求参数map
        // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", value=["
        Map<String, String[]> paramsMap = req.getParameterMap();

//        String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName()
//        // 调用处理该请求url对应的方法
//        method.invoke(ioc.get(beanName), new Object[]{req, resp, paramsMap.get("name"
//                paramsMap.get("id")[0]});

        // 实参列表
        // 实参列表要根据形参列表才能决定, 首先得拿到形参列表
        Class<?>[] paramterTypes = method.getParameterTypes();

        Object[] paramValues = new Object[paramterTypes.length];
        for (int i = 0, j = paramterTypes.length; i < j; i++) {
            Class<?> parameterType = paramterTypes[i];
            if (parameterType == HttpServletRequest.class) {
                paramValues[i] = req;
                continue;
            } else if (parameterType == HttpServletResponse.class) {
                paramValues[i] = resp;
                continue;
            } else if (parameterType == String.class) {
                // 获取方法参数上的所有注解
                Annotation[][] pa = method.getParameterAnnotations();
                for (int m = 0, n = pa.length; m < n; m++) {
                    // 遍历一个参数上的所有注解
                    for (Annotation a : pa[i]) {
                        // 该注解是否是MyRequestParam
                        if (a instanceof MyRequestParam) {
                            String paramName = ((MyRequestParam) a).value();
                            if (!"".equals(paramName.trim())) {
                                // 该value值形如: ["张三"]
                                // "\\[|\\]" => 匹配 [ 和 ]
                                // "\\s"  => 匹配任何空白字符, 包括空格、制表符、换页符
                                String value =
                                        Arrays.toString(paramsMap.get(paramName)).repla
                                                "\\[|\\]", "").replaceAll("\\s", ",");
                                paramValues[i] = value;
                            }
                        }
```

```
                    }
                }
            }
        }

        // 调用目标方法
        String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
        method.invoke(ioc.get(beanName), paramValues);
    }
```

完整代码如下所示:

```
package cn.sitedev.mvcframework.servlet.v2;

import cn.sitedev.mvcframework.annotation.*;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.net.URL;
import java.util.*;

public class MyDispatcherServlet extends HttpServlet {
    // 保存application.properties配置文件中的内容
    private Properties contextConfig = new Properties();
    // 保存扫描到的所有的类名
    private List<String> classNames = new ArrayList<>();
    // IOC容器
    // 为了简化程序，我们暂不考虑ConcurrentHashMap
    // 主要还是关注设计思想和原理
    private Map<String, Object> ioc = new HashMap<>();

    // 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 Servl
55          this.doPost(req, resp);
56      }
57
58      @Override
59      protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws Serv
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.getSta
66          }
67      }
68
69      private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
70          String url = req.getRequestURI();
71          String contextPath = req.getContextPath();
72          // 从请求url中截去context path
```

```
        url = url.replaceAll(contextPath, "").replaceAll("/+", "/");

        if (!this.handlerMapping.containsKey(url)) {
            resp.getWriter().write("404 Not Found !");
            return;
        }

        Method method = this.handlerMapping.get(url);

        // 获取请求参数map
        // 该map形如: "name"={key="name", value=["zhangsan"]}, "id"={key="id", value=["
        Map<String, String[]> paramsMap = req.getParameterMap();

//        String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName()
//        // 调用处理该请求url对应的方法
//        method.invoke(ioc.get(beanName), new Object[]{req, resp, paramsMap.get("name"
//                paramsMap.get("id")[0]});

        // 实参列表
        // 实参列表要根据形参列表才能决定, 首先得拿到形参列表
        Class<?>[] paramterTypes = method.getParameterTypes();

        Object[] paramValues = new Object[paramterTypes.length];
        for (int i = 0, j = paramterTypes.length; i < j; i++) {
            Class<?> parameterType = paramterTypes[i];
            if (parameterType == HttpServletRequest.class) {
                paramValues[i] = req;
                continue;
            } else if (parameterType == HttpServletResponse.class) {
                paramValues[i] = resp;
                continue;
            } else if (parameterType == String.class) {
                // 获取方法参数上的所有注解
                Annotation[][] pa = method.getParameterAnnotations();
                for (int m = 0, n = pa.length; m < n; m++) {
                    // 遍历一个参数上的所有注解
                    for (Annotation a : pa[i]) {
                        // 该注解是否是MyRequestParam
                        if (a instanceof MyRequestParam) {
                            String paramName = ((MyRequestParam) a).value();
                            if (!"".equals(paramName.trim())) {
                                // 该value值形如: ["张三"]
                                // "\\[|\\]" => 匹配 [ 和 ]
```

```
                                // "\\s"  => 匹配任何空白字符，包括空格、制表符、换页符
                                String value =
                                        Arrays.toString(paramsMap.get(paramName)).repla
                                                "\\[|\\]", "").replaceAll("\\s", ",");
                                paramValues[i] = value;
                            }
                        }
                    }
                }
            }

            // 调用目标方法
            String beanName = toLowerFirstCase(method.getDeclaringClass().getSimpleName());
            method.invoke(ioc.get(beanName), paramValues);
        }

        // 初始化url和Method的一对一对应关系
        private void doInitHandlerMapping() {
            if (ioc.isEmpty()) {
                return;
            }

            // 遍历ioc容器中的bean
            for (Map.Entry<String, Object> entry : ioc.entrySet()) {
                Class<?> clazz = entry.getValue().getClass();
                // 判断当前类上是否有MyController注解
                if (!clazz.isAnnotationPresent(MyController.class)) {
                    continue;
                }

                // 保存写在类上面的@MyRequestMapping("/demo")
                String baseUrl = "";
                // 判断当前类上是否有MyRequestMapping注解
                if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
                    // 获取MyRequestMapping注解
                    MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.
                    baseUrl = requestMapping.value();
                }

                // 默认获取所有的public方法
                for (Method method : clazz.getMethods()) {
                    // 判断方法实例上是否有MyRequestMapping注解
```

```
                if (!method.isAnnotationPresent(MyRequestMapping.class)) {
                    continue;
                }
                // 获取MyRequestMapping注解
                MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping

                // 将类上的MyRequestMapping注解的value值与类方法上的MyRequestMapping注解
                // "demo" + "query"  =>  "/demo/query""
                // "/demo" + "/query" => "//demo//query" => "/demo/query""
                String url = ("/" + baseUrl + "/" + requestMapping.value()).replaceAll(
                // key:请求url, value: 处理对应url的method
                handlerMapping.put(url, method);

                System.out.println("Mapped : " + url + ", " + method);
            }
        }
    }

    private void doAutowired() {
        if (ioc.isEmpty()) {
            return;
        }

        for (Map.Entry<String, Object> entry : ioc.entrySet()) {
            // 拿到实例的所有字段
            // Declared 所有的, 特定的 字段, 包括private/protected/default
            // 正常来说, 普通的OOP编程只能拿到public的属性
            Field[] fields = entry.getValue().getClass().getDeclaredFields();
            // 遍历类的所有字段, 并给指定注解的字段赋值
            for (Field field : fields) {
                // 判断当前字段上是否有MyAutowired注解
                if (!field.isAnnotationPresent(MyAutowired.class)) {
                    continue;
                }
                // 获取MyAutowired注解
                MyAutowired autowired = field.getAnnotation(MyAutowired.class);
                // 如果用户没有自定义beanName, 默认就根据类型注入
                // 这个地方省去了对类名首字母小写的情况的判断
                String beanName = autowired.value().trim();
                if ("".equals(beanName)) {
                    // 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
                    beanName = field.getType().getName();
                }
```

```
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Service，
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         }
```

```
    }
}
```

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：

```
    /**
     * Handler记录Controller中的RequestMapping和Method的关系
     */
    @Data
    private class Handler {
        // 保存方法对应的实例
        protected Object controller;
        // 保存映射的方法
        protected Method method;
        // ${} url占位符解析
        protected Pattern pattern;
        // 参数顺序
        protected Map<String, Integer> paramIndexMapping;

        /**
         * 构造一个Handler基本的参数
         *
         * @param pattern
         * @param controller
         * @param method
         */
        public Handler(Pattern pattern, Object controller, Method method) {
            this.pattern = pattern;
            this.controller = controller;
            this.method = method;
            this.paramIndexMapping = new HashMap<>();
            putParamIndexMapping(method);
        }
```

```
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.cla
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的一对一对应关系
2       private void doInitHandlerMapping() {
3           if (ioc.isEmpty()) {
4               return;
5           }
6
```

```
        // 遍历ioc容器中的bean
        for (Map.Entry<String, Object> entry : ioc.entrySet()) {
            Class<?> clazz = entry.getValue().getClass();
            // 判断当前类上是否有MyController注解
            if (!clazz.isAnnotationPresent(MyController.class)) {
                continue;
            }

            // 保存写在类上面的@MyRequestMapping("/demo")
            String baseUrl = "";
            // 判断当前Controller类上是否有MyRequestMapping注解
            if (clazz.isAnnotationPresent(MyRequestMapping.class)) {
                // 获取MyRequestMapping注解
                MyRequestMapping requestMapping = clazz.getAnnotation(MyRequestMapping.
                baseUrl = requestMapping.value();
            }

            // 默认获取所有的public方法
            for (Method method : clazz.getMethods()) {
                // 判断Method方法实例上是否有MyRequestMapping注解
                // 没有的直接忽略
                if (!method.isAnnotationPresent(MyRequestMapping.class)) {
                    continue;
                }
                // 获取MyRequestMapping注解
                MyRequestMapping requestMapping = method.getAnnotation(MyRequestMapping

                // 映射URL
                String regex = ("/" + baseUrl + "/" + requestMapping.value()).replaceAl
                Pattern pattern = Pattern.compile(regex);
                handlerMapping.add(new Handler(pattern, entry.getValue(), method));
                System.out.println("mapping : " + regex + ", " + method);
            }
        }
    }
```

修改doDispatch()方法：

```
    /**
     * 匹配URL
     *
```

```
     * @param req
     * @param resp
     * @throws Exception
     */
    private void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Ex
        try {
            Handler handler = getHandler(req);

            if (handler == null) {
                // 如果没有匹配上，就返回404错误
                resp.getWriter().write("404 Not Found");
                return;
            }

            // 获取方法的参数列表
            Class<?>[] paramTypes = handler.method.getParameterTypes();

            // 保存所有需要自动赋值的参数值
            Object[] paramValues = new Object[paramTypes.length];

            // 获取请求参数map
            // 该map形如：  "name"={key="name", value=["zhangsan"]}, "id"={key="id", valu
            Map<String, String[]> params = req.getParameterMap();

            for (Map.Entry<String, String[]> param : params.entrySet()) {
                String value =
                        Arrays.toString(param.getValue()).replaceAll("\\[|\\]", "").rep
                                "\\s", ",");

                // 如果找到匹配的对象，则开始填充参数值
                if (!handler.paramIndexMapping.containsKey(param.getKey())) {
                    continue;
                }
                int index = handler.paramIndexMapping.get(param.getKey());
                paramValues[index] = convert(paramTypes[index], value);
            }

            // 设置方法中的request和response对象
            int reqIndex = handler.paramIndexMapping.get(HttpServletRequest.class.getNa
            paramValues[reqIndex] = req;
            int respIndex = handler.paramIndexMapping.get(HttpServletResponse.class.get
            paramValues[respIndex] = resp;

```

```
            handler.method.invoke(handler.controller, paramValues);

        } catch (Exception e) {
            throw e;
        }
    }

    // url传过来的参数都是String类型的，http是基于字符串协议的
    // 只需要把String转换为任意类型就好
    private Object convert(Class<?> paramType, String value) {
        if (paramType == Integer.class) {
            return Integer.valueOf(value);
        }
        // 如果还有double或者其他类型，继续加if条件
        // 这个时候，我们应该想到策略模式
        // 这里我们暂时不实现
        return value;
    }

    private Handler getHandler(HttpServletRequest req) {
        if (handlerMapping.isEmpty()) {
            return null;
        }

        String url = req.getRequestURI();
        String contextPath = req.getContextPath();
        // 从url中截去context path
        url = url.replace(contextPath, "").replaceAll("/+", "/");

        // 从handlerMapping中找到一个合适的handler来处理当前url
        for (Handler handler : handlerMapping) {
            try {
                Matcher matcher = handler.pattern.matcher(url);
                // 如果没有匹配上就继续下一个匹配
                if (!matcher.matches()) {
                    continue;
                }
                return handler;
            } catch (Exception e) {
                throw e;
            }
        }
        return null;
```

```
    }
```

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:

```
package cn.sitedev.mvcframework.servlet.v3;

import cn.sitedev.mvcframework.annotation.*;
import lombok.Data;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.net.URL;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MyDispatcherServlet extends HttpServlet {
    // 保存application.properties配置文件中的内容
    private Properties contextConfig = new Properties();
    // 保存扫描到的所有的类名
    private List<String> classNames = new ArrayList<>();
    // IOC容器
    // 为了简化程序，我们暂不考虑ConcurrentHashMap
    // 主要还是关注设计思想和原理
    private Map<String, Object> ioc = new HashMap<>();

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

```

```
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 Servl
58          this.doPost(req, resp);
59      }
60
61      @Override
62      protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws Serv
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.getSta
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.getNa
114             paramValues[reqIndex] = req;
115             int respIndex = handler.paramIndexMapping.get(HttpServletResponse.class.get
116             paramValues[respIndex] = resp;
117 
118             handler.method.invoke(handler.controller, paramValues);
119 
120         } catch (Exception e) {
```

```
            throw e;
        }
    }

    // url传过来的参数都是String类型的，http是基于字符串协议的
    // 只需要把String转换为任意类型就好
    private Object convert(Class<?> paramType, String value) {
        if (paramType == Integer.class) {
            return Integer.valueOf(value);
        }
        // 如果还有double或者其他类型，继续加if条件
        // 这个时候，我们应该想到策略模式
        // 这里我们暂时不实现
        return value;
    }

    private Handler getHandler(HttpServletRequest req) {
        if (handlerMapping.isEmpty()) {
            return null;
        }

        String url = req.getRequestURI();
        String contextPath = req.getContextPath();
        // 从url中截去context path
        url = url.replace(contextPath, "").replaceAll("/+", "/");

        // 从handlerMapping中找到一个合适的handler来处理当前url
        for (Handler handler : handlerMapping) {
            try {
                Matcher matcher = handler.pattern.matcher(url);
                // 如果没有匹配上就继续下一个匹配
                if (!matcher.matches()) {
                    continue;
                }
                return handler;
            } catch (Exception e) {
                throw e;
            }
        }
        return null;
    }

    // 初始化url和Method的一对一对应关系
```

```
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.
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
195 
196                 // 映射URL
197                 String regex = ("/" + baseUrl + "/" + requestMapping.value()).replaceAl
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()) {
```

```
            return;
        }

        for (Map.Entry<String, Object> entry : ioc.entrySet()) {
            // 拿到实例的所有字段
            // Declared 所有的，特定的 字段，包括private/protected/default
            // 正常来说，普通的OOP编程只能拿到public的属性
            Field[] fields = entry.getValue().getClass().getDeclaredFields();
            // 遍历类的所有字段，并给指定注解的字段赋值
            for (Field field : fields) {
                // 判断当前字段上是否有MyAutowired注解
                if (!field.isAnnotationPresent(MyAutowired.class)) {
                    continue;
                }
                // 获取MyAutowired注解
                MyAutowired autowired = field.getAnnotation(MyAutowired.class);
                // 如果用户没有自定义beanName，默认就根据类型注入
                // 这个地方省去了对类名首字母小写的情况的判断
                String beanName = autowired.value().trim();
                if ("".equals(beanName)) {
                    // 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
                    beanName = field.getType().getName();
                }

                // 如果是public以外的修饰符，只要加了@MyAutowired注解，都要强制赋值
                // 反射中叫做暴力访问
                field.setAccessible(true);

                // 反射调用的方式
                // 给entry.getValue()这个对象的field字段，赋ioc.get(beanName)这个值
                try {
                    field.set(entry.getValue(), ioc.get(beanName));
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                    continue;
                }

            }

        }
    }

    private void doInstance() {
```

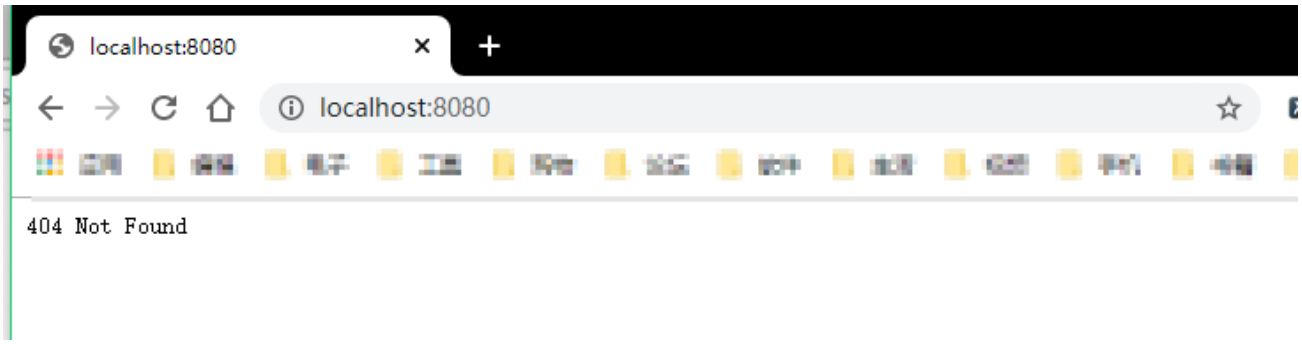
```
250         if (classNames.isEmpty()) {
251             return;
252         }
253         try {
254             for (String className : classNames) {
255                 Class<?> clazz = Class.forName(className);
256 
257                 // 什么样的类才需要初始化呢?
258                 // 加了注解的类，才会初始化，怎么判断?
259                 // 为了简化代码逻辑，主要体会设计思想，只举例@Controller和Service,
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                     }
```

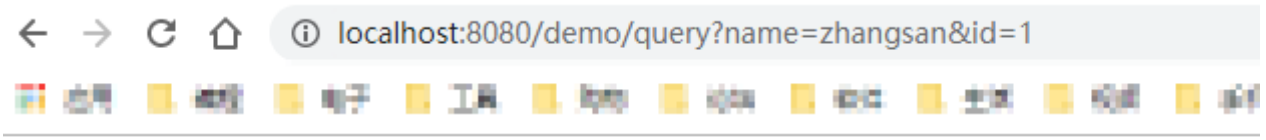
```
                } else {
                    continue;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 如果类名本身是小写字母，确实会出问题
    // 但是我要说明的是：这个方法是我自己用的，private类型的
    // 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
    // 为了简化程序逻辑，就不做其他判断了
    private String toLowerFirstCase(String simpleName) {
        char[] chars = simpleName.toCharArray();
        // 之所以采用 += ，是因为大小写字母的ASCII码值相差32
        // 而且大写字母的ASCII码要小于小写字母的
        // 在Java中，对char做数学运算，实际上就是对ASCII码做数学运算
        chars[0] += 32;
        return String.valueOf(chars);
    }

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

```
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(contextConfigLocat
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
```

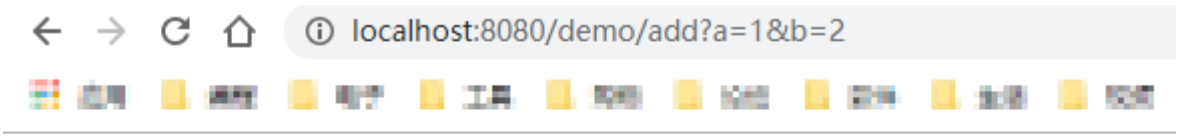
```
379          * @param controller
380          * @param method
381          */
382         public Handler(Pattern pattern, Object controller, Method method) {
383             this.pattern = pattern;
384             this.controller = controller;
385             this.method = method;
386             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.cla
409                     paramIndexMapping.put(type.getName(), i);
410                 }
411             }
412 
413         }
414     }
415 }
```

运行效果演示





My name is zhangsan



1+2=3

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。