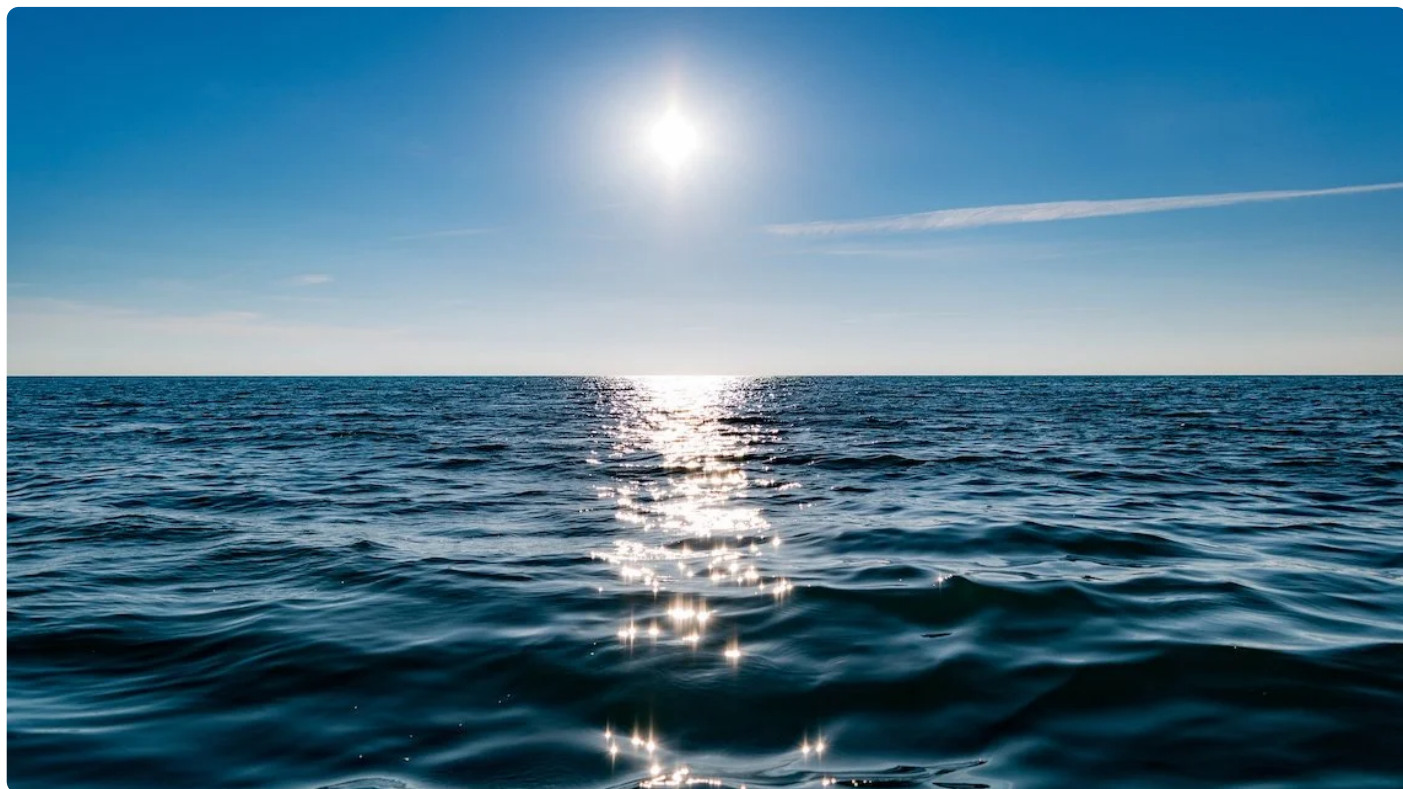


11 | ModelAndView：如何将处理结果返回给前端？

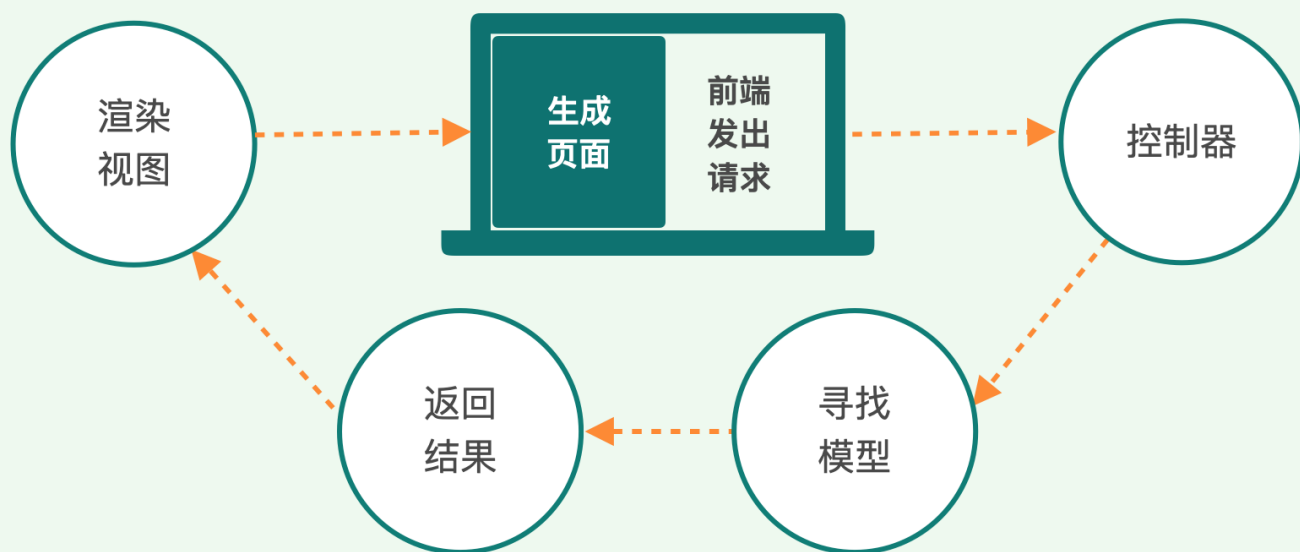
2023-04-05 郭屹 来自北京

《手把手带你写一个MiniSpring》



你好，我是郭屹。今天我们继续手写 MiniSpring。这也是 MVC 内容的最后一节。

上节课，我们对 HTTP 请求传入的参数进行了自动绑定，并调用了目标方法。我们再看一下整个 MVC 的流程，现在就到最后一步了，也就是把返回数据回传给前端进行渲染。



调用目标方法得到返回值之后，我们有两条路可以返回给前端。第一，返回的是简单的纯数据，第二，返回的是一个页面。

最近几年，第一种情况渐渐成为主流，也就是我们常说的“前后端分离”，后端处理完成后，只是把数据返回给前端，由前端自行渲染界面效果。比如前端用 React 或者 Vue.js 自行组织界面表达，这些前端脚本只需要从后端 service 拿到返回的数据就可以了。


第二种情况，由后端 controller 根据某种规则拿到一个页面，把数据整合进去，然后整个回传给前端浏览器，典型的技术就是 JSP。这条路前些年是主流，最近几年渐渐不流行了。

我们手写 MiniSpring 的目的是深入理解 Spring 框架，剖析它的程序结构，所以作为学习的对象，这两种情况我们都会分析到。

处理返回数据


和绑定传入的参数相对，处理返回数据是反向的，也就是说，要从后端把方法得到的返回值（一个 Java 对象）按照某种字符串格式回传给前端。我们以这个 `@ResponseBody` 注解为例，来分析一下。

先定义一个接口，增加一个功能，让 controller 返回给前端的字符流数据可以进行格式转换。

 复制代码

```
1 package com.minis.web;
2
3 import java.io.IOException;
4 import javax.servlet.http.HttpServletResponse;
5
6 public interface HttpMessageConverter {
7     void write(Object obj, HttpServletResponse response) throws IOException;
8 }
```

我们这里给一个默认的实现——DefaultHttpMessageConverter，把 Object 转成 JSON 串。

 复制代码

```
1 package com.minis.web;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.http.HttpServletResponse;
6
7 public class DefaultHttpMessageConverter implements HttpMessageConverter {
8     String defaultContentType = "text/json;charset=UTF-8";
9     String defaultCharacterEncoding = "UTF-8";
10    ObjectMapper objectMapper;
11
12    public ObjectMapper getObjectMapper() {
13        return objectMapper;
14    }
15    public void setObjectMapper(ObjectMapper objectMapper) {
16        this.objectMapper = objectMapper;
17    }
18    public void write(Object obj, HttpServletResponse response) throws IOException {
19        response.setContentType(defaultContentType);
20        response.setCharacterEncoding(defaultCharacterEncoding);
21        writeInternal(obj, response);
22        response.flushBuffer();
23    }
24    private void writeInternal(Object obj, HttpServletResponse response) throws IOException {
25        String sJsonStr = this.objectMapper.writeValueAsString(obj);
26        PrintWriter pw = response.getWriter();
```

```
27     pw.write(sJsonStr);
28 }
29 }
```

这个 message converter 很简单，就是给 response 写字符串，用到的工具是 ObjectMapper。我们就重点看看这个 mapper 是怎么做的。


定义一个接口 ObjectMapper。

 复制代码

```
1 package com.minis.web;
2 public interface ObjectMapper {
3     void setDateFormat(String dateFormat);
4     void setDecimalFormat(String decimalFormat);
5     String writeValuesAsString(Object obj);
6 }
```

最重要的接口方法就是 writeValuesAsString()，将对象转成字符串。

我们给一个默认的实现——DefaultObjectMapper，在 writeValuesAsString 中拼 JSON 串。

 复制代码

```
1 package com.minis.web;
2
3 import java.lang.reflect.Field;
4 import java.math.BigDecimal;
5 import java.text.DecimalFormat;
6 import java.time.LocalDate;
7 import java.time.ZoneId;
8 import java.time.format.DateTimeFormatter;
9 import java.util.Date;
10
11 public class DefaultObjectMapper implements ObjectMapper{
12     String dateFormat = "yyyy-MM-dd";
13     DateTimeFormatter datetimeFormatter = DateTimeFormatter.ofPattern(dateFormat);
14
15     String decimalFormat = "#,##0.00";
16     DecimalFormat decimalFormatter = new DecimalFormat(decimalFormat);
```

```
17
18 public DefaultObjectMapper() {
19 }
20
21 @Override
22 public void setDateFormat(String dateFormat) {
23     this.dateFormat = dateFormat;
24     this.datetimeFormatter = DateTimeFormatter.ofPattern(dateFormat);
25 }
26
27 @Override
28 public void setDecimalFormat(String decimalFormat) {
29     this.decimalFormat = decimalFormat;
30     this.decimalFormatter = new DecimalFormat(decimalFormat);
31 }
32 public String writeValuesAsString(Object obj) {
33     String sJsonStr = "{";
34     Class<?> clz = obj.getClass();
35
36     Field[] fields = clz.getDeclaredFields();
37     //对返回对象中的每一个属性进行格式转换
38     for (Field field : fields) {
39         String sField = "";
40         Object value = null;
41         Class<?> type = null;
42         String name = field.getName();
43         String strValue = "";
44         field.setAccessible(true);
45         value = field.get(obj);
46         type = field.getType();
47
48         //针对不同的数据类型进行格式转换
49         if (value instanceof Date) {
50             LocalDate localDate = ((Date)value).toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
51             strValue = localDate.format(this.datetimeFormatter);
52         }
53         else if (value instanceof BigDecimal || value instanceof Double || value instanceof Float) {
54             strValue = this.decimalFormatter.format(value);
55         }
56         else {
57             strValue = value.toString();
58         }
59
60         //拼接Json串
61         if (sJsonStr.equals("{")) {
62             sField = "\"" + name + "\":\"" + strValue + "\"";
63         }
64         else {
65             sField = "\",\"" + name + "\":\"" + strValue + "\"";
```

```


66     }
67
68     sJsonStr += sField;
69 }
70 sJsonStr += "}";
71 return sJsonStr;
72 }
73 }

```

实际转换过程用到了 `LocalDate` 和 `DecimalFormatter`。从上述代码中也可以看出，目前为止，我们也只支持 `Date`、`Number` 和 `String` 三种类型。你自己可以考虑扩展到更多的数据类型。

那么我们在哪个地方用这个工具来处理返回的数据呢？其实跟绑定参数一样，数据返回之前，也是要经过方法调用。所以我们还是要回到 `RequestMappingHandlerAdapter` 这个类，增加一个属性 `messageConverter`，通过它来转换数据。

程序变成了这个样子。

 复制代码


```

1  public class RequestMappingHandlerAdapter implements HandlerAdapter {
2      private WebBindingInitializer webBindingInitializer = null;
3      private HttpMessageConverter messageConverter = null;

```

现在既有传入的 `webBindingInitializer`，也有传出的 `messageConverter`。

在关键方法 `invokeHandlerMethod()` 里增加对 `@ResponseBody` 的处理，也就是调用 `messageConverter.write()` 把方法返回值转换成字符串。

 复制代码


```

1  protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
2      HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
3      ... ...
4      if (invocableMethod.isAnnotationPresent(ResponseBody.class)){ //ResponseBody
5          this.messageConverter.write(returnObj, response);
6      }

```


```
7     ... ..
8 }
```

同样的 `webBindingInitializer` 和 `messageConverter` 都可以通过配置注入。

 复制代码

```
1 <bean id="handlerAdapter" class="com.minis.web.servlet.RequestMappingHandlerAda
2 <property type="com.minis.web.HttpMessageConverter" name="messageConverter" re
3 <property type="com.minis.web.WebBindingInitializer" name="webBindingInitializ
4 </bean>
5
6 <bean id="webBindingInitializer" class="com.test.DateInitializer" />
7
8 <bean id="messageConverter" class="com.minis.web.DefaultHttpMessageConverter">
9 <property type="com.minis.web.ObjectMapper" name="objectMapper" ref="objectMap
10 </bean>
11 <bean id="objectMapper" class="com.minis.web.DefaultObjectMapper" >
12 <property type="String" name="dateFormat" value="yyyy/MM/dd"/>
13 <property type="String" name="decimalFormat" value="###.##"/>
14 </bean>
```

最后在 `DispatcherServlet` 里，通过 `getBean` 获取 `handlerAdapter`，当然这里需要约定一个名字，整个过程就连起来了。

 复制代码

```
1 protected void initHandlerAdapters(WebApplicationContext wac) {
2     this.handlerAdapter = (HandlerAdapter) wac.getBean(HANDLER_ADAPTER_BEAN_NAME
3 }
```

测试的客户程序 `HelloWorldBean` 修改如下：

```
1  @RequestMapping("/test7")
2  @ResponseBody
3  public User doTest7(User user) {
4      user.setName(user.getName() + "----");
5      user.setBirthday(new Date());
6      return user;
7  }
```

程序里面声明了一个注解 `@ResponseBody`，程序中返回的是对象 `User`，框架处理的时候用 `message converter` 将其转换成 JSON 字符串返回。

到这里，我们就知道 MVC 是如何把方法返回对象自动转换成 response 字符串的了。我们在调用目标方法后，通过 `messageConverter` 进行转换，它要分别转换每一种数据类型的格式，同时格式可以由用户自己指定。

ModelAndView

调用完目标方法，得到返回值，把数据按照指定格式转换好之后，就该处理它们，并把它们送到前端去了。我们用一个统一的结构，包装调用方法之后返回的数据，以及需要启动的前端页面，这个结构就是 `ModelAndView`，我们看下它的定义。

```
1  package com.minis.web.servlet;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public class ModelAndView {
7      private Object view;
8      private Map<String, Object> model = new HashMap<>();
9
10     public ModelAndView() {
11     }
12     public ModelAndView(String viewName) {
13         this.view = viewName;
14     }
15     public ModelAndView(View view) {
16         this.view = view;
17     }
```



```
18 public ModelAndView(String viewName, Map<String, ?> modelData) {
19     this.view = viewName;
20     if (modelData != null) {
21         addAllAttributes(modelData);
22     }
23 }
24 public ModelAndView(View view, Map<String, ?> model) {
25     this.view = view;
26     if (model != null) {
27         addAllAttributes(model);
28     }
29 }
30 public ModelAndView(String viewName, String modelName, Object modelObject) {
31     this.view = viewName;
32     addObject(modelName, modelObject);
33 }
34 public ModelAndView(View view, String modelName, Object modelObject) {
35     this.view = view;
36     addObject(modelName, modelObject);
37 }
38 public void setViewName(String viewName) {
39     this.view = viewName;
40 }
41 public String getViewName() {
42     return (this.view instanceof String ? (String) this.view : null);
43 }
44 public void setView(View view) {
45     this.view = view;
46 }
47 public View getView() {
48     return (this.view instanceof View ? (View) this.view : null);
49 }
50 public boolean hasView() {
51     return (this.view != null);
52 }
53 public boolean isReference() {
54     return (this.view instanceof String);
55 }
56 public Map<String, Object> getModel() {
57     return this.model;
58 }
59 private void addAllAttributes(Map<String, ?> modelData) {
60     if (modelData != null) {
61         model.putAll(modelData);
62     }
63 }
64 public void addAttribute(String attributeName, Object attributeValue) {
65     model.put(attributeName, attributeValue);
66 }
```


```

67     public ModelAndView addObject(String attributeName, Object attributeValue) {
68         addAttribute(attributeName, attributeValue);
69         return this;
70     }
71 }

```

这个类里面定义了 Model 和 View，分别代表返回的数据以及前端表示，我们这里就是指 JSP。

有了这个结构，我们回头看调用目标方法之后返回的那段代码，把类 RequestMappingHandlerAdapter 的方法 invokeHandlerMethod() 返回值改为 ModelAndView。

 复制代码

```

1  protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
2      HttpServletResponse response, HandlerMethod handlerMethod) throws Ex
3      ModelAndView mav = null;
4      //如果是ResponseBody注解，仅仅返回值，则转换数据格式后直接写到response
5      if (invocableMethod.isAnnotationPresent(ResponseBody.class)){ //ResponseBody
6          this.messageConverter.write(returnObj, response);
7      }
8      else { //返回的是前端页面
9          if (returnObj instanceof ModelAndView) {
10             mav = (ModelAndView)returnObj;
11         }
12         else if(returnObj instanceof String) { //字符串也认为是前端页面
13             String sTarget = (String)returnObj;
14             mav = new ModelAndView();
15             mav.setViewName(sTarget);
16         }
17     }
18
19     return mav;
20 }

```


通过上面这段代码我们可以知道，调用方法返回的时候，我们处理了三种情况。

1. 如果声明返回的是 ResponseBody，那就用 MessageConvert 把结果转换一下，之后直接写回 response。

2. 如果声明返回的是 ModelAndView，那就把结果包装成一个 ModelAndView 对象返回。
3. 如果声明返回的是字符串，就以这个字符串为目标，最后还是包装成 ModelAndView 返回。

View

到这里，调用方法就返回了。不过事情还没完，之后我们就把注意力转移到 MVC 环节的最后一部分：View 层。View，顾名思义，就是负责前端界面展示的部件，当然它最主要的功能就是，把数据按照一定格式显示并输出到前端界面上，因此可以抽象出它的核心方法 render()，我们可以看下 View 接口的定义。


 复制代码

```
1 package com.minis.web.servlet;
2
3 import java.util.Map;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6
7 public interface View {
8     void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)
9         throws Exception;
10    default String getContentType() {
11        return null;
12    }
13    void setContentType(String contentType);
14    void setUrl(String url);
15    String getUrl();
16    void setRequestContextAttribute(String requestContextAttribute);
17    String getRequestContextAttribute();
18 }
```

这个 render() 方法的思路很简单，就是获取 HTTP 请求的 request 和 response，以及中间产生的业务数据 Model，最后写到 response 里面。request 和 response 是 HTTP 访问时由服务器创建的，ModelAndView 是由我们的 MiniSpring 创建的。


准备好数据之后，我们以 JSP 为例，来看看怎么把结果显示在前端界面上。其实，这跟我们自己手工写 JSP 是一样的，先设置属性值，然后把请求转发 (forward) 出去，就像下面我给出

的这几行代码。

 复制代码


```
1 request.setAttribute(key1, value1);
2 request.setAttribute(key2, value2);
3 request.getRequestDispatcher(url).forward(request, response);
```

照此办理，DispatcherServlet 的 doDispatch() 方法调用目标方法后，可以通过一个 render() 来渲染这个 JSP，你可以看一下 doDispatch() 相关代码。

 复制代码

```
1 HandlerAdapter ha = this.handlerAdapter;
2 mv = ha.handle(processedRequest, response, handlerMethod);
3 render(processedRequest, response, mv);
```

这个 render() 方法可以考虑这样实现。


 复制代码

```
1 //用jsp 进行render
2 protected void render( HttpServletRequest request, HttpServletResponse response
3     //获取model, 写到request的Attribute中:
4     Map<String, Object> modelMap = mv.getModel();
5     for (Map.Entry<String, Object> e : modelMap.entrySet()) {
6         request.setAttribute(e.getKey(), e.getValue());
7     }
8     //输出到目标JSP
9     String sTarget = mv.getViewName();
10    String sPath = "/" + sTarget + ".jsp";
11    request.getRequestDispatcher(sPath).forward(request, response);
12 }
```

我们看到了，程序从 Model 里获取数据，并将其作为属性值写到 request 的 attribute 里，然后获取页面路径，再显示出来，跟手工写 JSP 过程一样，简明有效。


但是上面的程序有两个问题，一是这个程序是怎么找到显示目标 View 的呢？上面的例子，我们是写了一个固定的路径 /xxxx.jsp，但实际上这些应该是可以让用户自己来配置的，不应该写死在代码中。二是拿到 View 后，直接用的是 request 的 forward() 方法，这只对 JSP 有效，没办法扩展到别的页面，比如说 Excel、PDF。所以上面的 render() 是需要改造的。

先解决第一个问题，怎么找到需要显示的目标 View？这里又得引出了一个新的部件 ViewResolver，由它来根据某个规则或者是用户配置来确定 View 在哪里，下面是它的定义。

 复制代码


```
1 package com.minis.web.servlet;
2
3 public interface ViewResolver {
4     View resolveViewName(String viewName) throws Exception;
5 }
```

这个 ViewResolver 就是根据 View 的名字找到实际的 View，有了这个 ViewResolver，就不用写死 JSP 路径，而是可以通过 resolveViewName() 方法来获取一个 View。拿到目标 View 之后，我们把实际渲染的功能交给 View 自己完成。我们把程序改成下面这个样子。

 复制代码

```
1 protected void render( HttpServletRequest request, HttpServletResponse response
2     String sTarget = mv.getViewName();
3     Map<String, Object> modelMap = mv.getModel();
4     View view = resolveViewName(sTarget, modelMap, request);
5     view.render(modelMap, request, response);
6 }
```

在 MiniSpring 里，我们提供一个 InternalResourceViewResolver，作为启动 JSP 的默认实现，它是这样定位到显示目标 View 的。

 复制代码

```
1 package com.minis.web.servlet.view;
2
3 import com.minis.web.servlet.View;
```

```
4 import com.minis.web.servlet.ViewResolver;
5
6 public class InternalResourceViewResolver implements ViewResolver{
7     private Class<?> viewClass = null;
8     private String viewClassName = "";
9     private String prefix = "";
10    private String suffix = "";
11    private String contentType;
12
13    public InternalResourceViewResolver() {
14        if (getViewClass() == null) {
15            setViewClass(JstlView.class);
16        }
17    }
18
19    public void setViewClassName(String viewClassName) {
20        this.viewClassName = viewClassName;
21        Class<?> clz = null;
22        try {
23            clz = Class.forName(viewClassName);
24        } catch (ClassNotFoundException e) {
25            e.printStackTrace();
26        }
27        setViewClass(clz);
28    }
29
30    protected String getViewClassName() {
31        return this.viewClassName;
32    }
33    public void setViewClass(Class<?> viewClass) {
34        this.viewClass = viewClass;
35    }
36    protected Class<?> getViewClass() {
37        return this.viewClass;
38    }
39    public void setPrefix(String prefix) {
40        this.prefix = (prefix != null ? prefix : "");
41    }
42    protected String getPrefix() {
43        return this.prefix;
44    }
45    public void setSuffix(String suffix) {
46        this.suffix = (suffix != null ? suffix : "");
47    }
48    protected String getSuffix() {
49        return this.suffix;
50    }
51    public void setContentType(String contentType) {
52        this.contentType = contentType;
```

```

53     }
54     protected String getContentType() {
55         return this.contentType;
56     }
57
58     @Override
59     public View resolveViewName(String viewName) throws Exception {
60         return buildView(viewName);
61     }
62
63     protected View buildView(String viewName) throws Exception {
64         Class<?> viewClass = getViewClass();
65
66         View view = (View) viewClass.newInstance();
67         view.setUrl(getPrefix() + viewName + getSuffix());
68
69         String contentType = getContentType();
70         view.setContentType(contentType);
71
72         return view;
73     }
74 }

```

从代码里可以知道，它先创建 View 实例，通过配置生成 URL 定位到显示目标，然后设置 ContentType。这个过程也跟我们手工写 JSP 是一样的。通过这个 resolver，就解决了第一个问题，框架会根据配置从 /jsp/ 路径下拿到 xxxx.jsp 页面。

对于第二个问题，DispatcherServlet 是不应该负责实际的渲染工作的，它只负责控制流程，并不知道如何渲染前端，这些工作由具体的 View 实现类来完成。所以我们不再把 request forward() 这样的代码写到 DispatcherServlet 里，而是写到 View 的 render() 方法中。

MiniSpring 也提供了一个默认的实现：JstlView。

 复制代码

```

1 package com.minis.web.servlet.view;
2
3 import java.util.Map;
4 import java.util.Map.Entry;
5 import javax.servlet.http.HttpServletRequest;
6 import javax.servlet.http.HttpServletResponse;
7 import com.minis.web.servlet.View;
8

```


```

9 public class JstlView implements View{
10     public static final String DEFAULT_CONTENT_TYPE = "text/html;charset=ISO-8859-1";
11     private String contentType = DEFAULT_CONTENT_TYPE;
12     private String requestContextAttribute;
13     private String beanName;
14     private String url;
15
16     public void setContentType(String contentType) {
17         this.contentType = contentType;
18     }
19     public String getContentType() {
20         return this.contentType;
21     }
22     public void setRequestContextAttribute(String requestContextAttribute) {
23         this.requestContextAttribute = requestContextAttribute;
24     }
25     public String getRequestContextAttribute() {
26         return this.requestContextAttribute;
27     }
28     public void setBeanName(String beanName) {
29         this.beanName = beanName;
30     }
31     public String getBeanName() {
32         return this.beanName;
33     }
34     public void setUrl(String url) {
35         this.url = url;
36     }
37     public String getUrl() {
38         return this.url;
39     }
40     public void render(Map<String, ?> model, HttpServletRequest request, HttpServlet
41         throws Exception {
42         for (Entry<String, ?> e : model.entrySet()) {
43             request.setAttribute(e.getKey(), e.getValue());
44         }
45         request.getRequestDispatcher(getUrl()).forward(request, response);
46     }
47 }

```

从代码里可以看到，程序其实还是一样的，因为要完成的任务是一样的，只不过现在这个代码移到了 View 这个位置。但是这个位置的移动，就让前端的渲染工作解耦了，DispatcherServlet 不负责渲染了，我们可以由此扩展到多种前端，如 Excel、PDF 等等。

然后，对于 `InternalResourceViewResolver` 和 `JstlView`，我们可以再次利用 IoC 容器机制通过配置进行注入。

 复制代码

```
1    <bean id="viewResolver" class="com.minis.web.servlet.view.InternalResourceView" />
2    <property type="String" name="viewClassName" value="com.minis.web.servlet.view.JstlView" />
3    <property type="String" name="prefix" value="/jsp/" />
4    <property type="String" name="suffix" value=".jsp" />
5    </bean>
```

当 `DispatcherServlet` 初始化的时候，根据配置获取实际的 `ViewResolver` 和 `View`。

整个过程就完美结束了。

小结

这节课，我们重点探讨了 MVC 调用目标方法之后的处理过程，如何自动转换数据、如何找到指定的 View、如何去渲染页面。我们可以看到，作为一个框架，我们没有规定数据要如何转换格式，而是交给了 `MessageConverter` 去做；我们也没有规定如何找到这些目标页面，而是交给了 `ViewResolver` 去做；我们同样没有规定如何去渲染前端界面，而是通过 `View` 这个接口去做。我们可以自由地实现具体的场景。

这里，我们的重点并不是去看具体代码如何实现，而是要学习 Spring 框架如何分解这些工作，把专门的事情交给专门的部件去完成。虽然现在已经不流行 JSP，我们不用特地去学习它，但是把这些部件解耦的框架思想，却是值得我们好好琢磨的。

完整源代码参见：<https://github.com/YaleGuo/minis>

课后题

学完这节课，我也给你留一道思考题。现在我们返回的数据只支持 `Date`、`Number` 和 `String` 三种类型，如何扩展到更多的数据类型？现在也只支持 JSP，如何扩展到别的前端？欢迎你在留言区和我交流讨论，也欢迎你把这节课分享给需要的朋友。我们下节课见！

精选留言 (4)



梦某人

2023-04-18 来自河北

打卡成功，从理解上这节课并不难，虽然很多代码（主要是User类和一些辅助）需要参考GitHub不然无法进行。但是调整环境浪费了接近2个小时，因为访问jsp一直报404的错误，后来意识到是没在idea的Project Structure 中的 Module设置资源文件夹。。。另外目前的返回来讲，string包装成了 ModelAndView，但是这样做在render的时候无法辨别，导致最基础的 /test反而无法访问。思考题来说，View的两个类，一个负责分析内容，一个负责渲染内容，将 ViewResolver 进行扩展就可以解决相关问题了。



马儿

2023-04-08 来自四川

- 1.目前如果加了ResponseBody注解返回String的话返回的是String在内存中的信息，而需要的字符串值字段在这个json中也是内存中的地址值，这将导致结果不符合预期，这里应该还需要对writeValuesAsString这个函数优化一下，或者是拓展一些其他的实现。
2. 目前在InternalResourceViewResolver中写死了处理Jsp的View，可以在加一个有参构造函数，传入参数为资源类型，InternalResourceViewResolver内部维护一个资源类型和View的Map

希望老师可以抽时间加一些答疑课，对之前一些同学问到的问题在课上统一解答一下。或者是一些mini-spring中的一些拓展点提供一个思路，比如上节课遇到的传参数不支持基本类型和自定义类型中WebDataBinder不可用的问题。谢谢老师。

作者回复: 1 你说的是，线下课确实是作为扩展练习的。

2 你可以这么考虑，对别的view，要用不同的view resolver，即写一个与InternalResourceViewResolver对等的实现来支持别的view。



peter

2023-04-06 来自北京

请教老师几个问题：

Q1：本文所讲的内容，就是模仿SpringMVC，对吗？

Q2：很多信息都存在request中，那这个request对象会占用很大内存吗？对于一个用户，一般地讲，会占用多大内存？比如10M？

Q3：View这个类，是生成一个页面文件吗？还是把数据填充到已经存在的页面上？

作者回复：Peter你好。MiniSpring是模仿Spring框架的一个简化版本，目的是作为一个简要地图便于大家理解Spring的结构和源代码，MiniSpring的包结构类名和主要流程方法都是跟Spring框架本身一样的，所以学习了MiniSpring，会比较容易继续深入了解Spring的源代码。

request是对http request的包装，大小主要依赖于客户端传上来的数据包大小，不考虑附件，一般不会太大，应该在200K以内。你要再学一下我的MiniTomcat后就会更加清楚。

View这个类的定位是前端展示，如果是JSP，就是把数据填充到JSP中，然后展示出来。最近这些年都是前后端分离了，这一部分简单了解一下就可以。



C.

2023-04-06 来自江苏

结束结束！

