



下载APP



17 | 答疑现场：Spring Web 篇思考题合集

2021-05-31 傅健

《Spring编程常见错误50例》

[课程介绍 >](#)**讲述：傅健**

时长 00:42 大小 665.03K



你好，我是傅健。


欢迎来到第二次答疑现场，恭喜你，已经完成了三分之二的课程。到今天为止，我们已经解决了 38 个线上问题，不知道你在工作中有所应用了吗？老话说得好，“纸上得来终觉浅，绝知此事要躬行”。希望你能用行动把知识从“我的”变成“你的”。

闲话少叙，接下来我就开始逐一解答第二章的课后思考题了，有任何想法欢迎到留言区补充。



🔗 第 9 课


关于 URL 解析，其实还有许多让我们惊讶的地方，例如案例 2 的部分代码：

 复制代码

```
1 @RequestMapping(path = "/hi2", method = RequestMethod.GET)
2 public String hi2(@RequestParam("name") String name){
3     return name;
4 };
```


在上述代码的应用中，我们可以使用 <http://localhost:8080/hi2?name=xiaoming&name=hanmeimei> 来测试下，结果会返回什么呢？你猜会是 [xiaoming&name=hanmeimei](http://localhost:8080/hi2?name=xiaoming&name=hanmeimei) 么？

针对这个测试，返回的结果其实是"xiaoming,hanmeimei"。这里我们可以追溯到请求参数的解析代码，参考 org.apache.tomcat.util.http.Parameters#addParameter：

 复制代码


```
1 public void addParameter( String key, String value )
2     throws IllegalStateException {
3     //省略其他非关键代码
4     ArrayList<String> values = paramHashValues.get(key);
5     if (values == null) {
6         values = new ArrayList<>(1);
7         paramHashValues.put(key, values);
8     }
9     values.add(value);
10 }
```

可以看出当使用 [name=xiaoming&name=hanmeimei](http://localhost:8080/hi2?name=xiaoming&name=hanmeimei) 这种形式访问时，name 解析出的参数值是一个 ArrayList 集合，它包含了所有的值（此处为 xiaoming 和 hanmeimei）。但是这个数组在最终是需要转化给我们的 String 类型的。转化执行可参考其对应转化器 `ArrayToStringConverter` 所做的转化，关键代码如下：

 复制代码


```
1 public Object convert(@Nullable Object source, TypeDescriptor sourceType, Type
2     return this.helperConverter.convert(Arrays.asList(ObjectUtils.toObjectArray
3 }
```

其中 `helperConverter` 为 `CollectionToStringConverter`，它使用了 "," 作为分隔将集合转化为 String 类型，分隔符定义如下：

 复制代码

```
1 private static final String DELIMITER = ",";
```

通过上述分析可知，对于参数解析，解析出的结果其实是一个数组，只是在最终转化时，可能因不同需求转化为不同的类型，从而呈现出不同的值，有时候反倒让我们很惊讶。分析了这么多，我们可以改下代码，测试下刚才的源码解析出的一些结论，代码修改如下：

 复制代码


```
1 @RequestMapping(path = "/hi2", method = RequestMethod.GET)
2 public String hi2(@RequestParam("name") String[] name){
3     return Arrays.toString(name);
4 };
```

这里我们将接收类型改为 String 数组，然后我们重新测试，会发现结果为 [xiaoming, hanmeimei]，这就更好理解和接受了。

🔗 第 10 课

在案例 3 中，我们以 Content-Type 为例，提到在 Controller 层中随意自定义常用头有时候会失效。那么这个结论是不是普适呢？即在使用其他内置容器或者在其他开发框架下，是不是也会存在一样的问题？

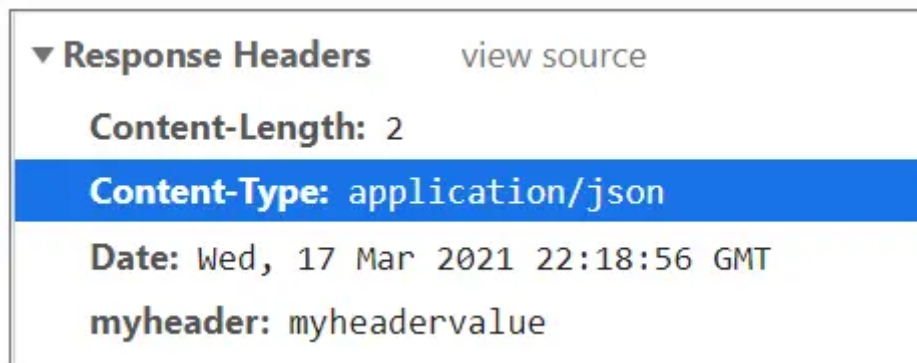
实际上，答案是否定的。这里我们不妨修改下案例 3 的 pom.xml。修改的目标是让其不要使用默认的内嵌 Tomcat 容器，而是 Jetty 容器。具体修改示例如下：

 复制代码

```
1     <dependency>
2         <groupId>org.springframework.boot</groupId>
3         <artifactId>spring-boot-starter-web</artifactId>
4         <exclusions>
5             <exclusion>
6                 <groupId>org.springframework.boot</groupId>
7                 <artifactId>spring-boot-starter-tomcat</artifactId>
8             </exclusion>
9         </exclusions>
10    </dependency>
11    <!-- 使用 Jetty -->
12    <dependency>
13        <groupId>org.springframework.boot</groupId>
```

```
14         <artifactId>spring-boot-starter-jetty</artifactId>
15     </dependency>
```

经过上面的修改后，我们再次运行测试程序，我们会发现 Content-Type 确实可以设置成我们想要的样子，具体如下：



同样是执行 addHeader()，但是因为置换了容器，所以调用的方法实际是 Jetty 的方法，具体参考 org.eclipse.jetty.server.Response#addHeader：

[复制代码](#)

```
1 public void addHeader(String name, String value)
2 {
3     //省略其他非关键代码
4     if (HttpHeader.CONTENT_TYPE.is(name))
5     {
6         setContentType(value);
7         return;
8     }
9     //省略其他非关键代码
10    _fields.add(name, value);
11 }
```

在上述代码中，setContentType() 最终是完成了 Header 的添加。这点和 Tomcat 完全不同。具体可参考其实现：

[复制代码](#)

```
1 public void setContentType(String contentType)
2 {
3     //省略其他非关键代码
4     if (HttpGenerator.__STRICT || _mimeType == null)
5         //添加CONTENT_TYPE
6         _fields.put(HttpHeader.CONTENT_TYPE, _contentType);
```

```
7         else
8         {
9             _contentType = _mimeType.asString();
10            _fields.put(_mimeType.getContentTypeField());
11        }
12    }
13 }
```

再次对照案例 3 给出的部分代码，在这里，直接贴出关键一段（具体参考 `AbstractMessageConverterMethodProcessor#writeWithMessageConverters`）：

[复制代码](#)

```
1 MediaType selectedMediaType = null;
2 MediaType contentType = outputMessage.getHeaders().getContentType();
3 boolean isContentTypePreset = contentType != null && contentType.isConcrete();
4 if (isContentTypePreset) {
5     selectedMediaType = contentType;
6 } else {
7     //根据请求 Accept 头和注解指定的返回类型 (RequestMapping#produces) 协商用何种 MediaType
8 }
9 //省略其他代码：else
```

从上述代码可以看出，最终选择的 `MediaType` 已经不需要协商了，这是因为在 Jetty 容器中，Header 里面添加进了 `contentType`，所以可以拿出来直接使用。而之前介绍的 Tomcat 容器没有把 `contentType` 添加进 Header 里，所以在上述代码中，它不能走入 `isContentTypePreset` 为 `true` 的分支。此时，它只能根据请求 Accept 头和注解指定的返回类型等信息协商用何种 `MediaType`。

追根溯源，主要在于不同的容器对于 `addHeader()` 的实现不同。这里我们不妨再深入探讨下。首先，回顾我们案例 3 代码中的方法定义：

[复制代码](#)

```
1 import javax.servlet.http.HttpServletResponse;
2 public String hi3(HttpServletResponse httpServletResponse)
```

虽然都是接口 `HttpServletResponse`，但是在 Jetty 容器下，会被装配成 `org.eclipse.jetty.server.Response`，而在 Tomcat 容器下，会被装配成 `org.apache.catalina.connector.Response`。所以调用的方法才会发生不同。

如何理解这个现象？容器是通信层，而 Spring Boot 在这其中只是中转，所以在 Spring Boot 中，HTTP Servlet Response 来源于最原始的通信层提供的对象，这样也就合理了。

通过这个思考题，我们可以看出：对于很多技术的使用，一些结论并不是一成不变的。可能只是换下容器，结论就会失效。所以，只有洞悉其原理，才能从根本上避免各种各样的麻烦，而不仅仅是凭借一些结论去“刻舟求剑”。

🔗 第 11 课

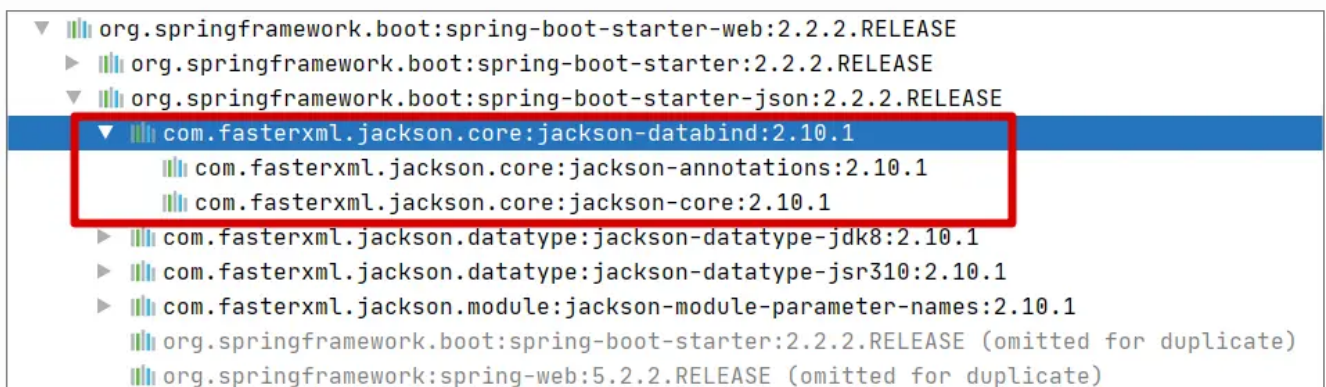
通过案例 1 的学习，我们知道直接基于 Spring MVC 而非 Spring Boot 时，是需要我们手工添加 JSON 依赖，才能解析出 JSON 的请求或者编码 JSON 响应，那么为什么基于 Spring Boot 就不需要这样做了呢？

实际上，当我们使用 Spring Boot 时，我们都会添加相关依赖项：

📄 复制代码

```
1 <dependencies>
2     <dependency>
3         <groupId>org.springframework.boot</groupId>
4         <artifactId>spring-boot-starter-web</artifactId>
5     </dependency>
6 </dependencies>
```

而这个依赖项会间接把 Jackson 添加进去，依赖关系参考下图：




后续 Jackson 编解码器的添加，和普通 Spring MVC 关键逻辑相同：都是判断相关类是否存在。不过这里可以稍微总结下，判断相关类是否存在有两种风格：

1. 直接使用反射来判断

例如前文介绍的关键语句：

```
ClassUtils.isPresent("com.fasterxml.jackson.databind.ObjectMapper", null)
```

2. 使用 @ConditionalOnClass 参考 JacksonHttpMessageConvertersConfiguration 的实现：


 复制代码

```
1 package org.springframework.boot.autoconfigure.http;
2
3 @Configuration(proxyBeanMethods = false)
4 class JacksonHttpMessageConvertersConfiguration {
5     @Configuration(proxyBeanMethods = false)
6     @ConditionalOnClass(ObjectMapper.class)
7     @ConditionalOnBean(ObjectMapper.class)
8     @ConditionalOnProperty(name = HttpMessageConvertersAutoConfiguration.PREFER
9         havingValue = "jackson", matchIfMissing = true)
10    static class MappingJackson2HttpMessageConverterConfiguration {
11        @Bean
12        @ConditionalOnMissingBean(value = MappingJackson2HttpMessageConverter.cl
13        //省略部分非关键代码
14        MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter(
15            return new MappingJackson2HttpMessageConverter(objectMapper);
16    }
17 }
```

以上即为判断某个类是否存在的两种方法。

🔗 第 12 课

在上面的学籍管理系统中，我们还存在一个接口，负责根据学生的学号删除他的信息，代码如下：

 复制代码

```
1 @RequestMapping(path = "students/{id}", method = RequestMethod.DELETE)
2 public void deleteStudent(@PathVariable("id") @Range(min = 1,max = 10000) Stri
3     log.info("delete student: {}",id);
4     //省略业务代码
5 };
```

这个学生的编号是从请求的 Path 中获取的，而且它做了范围约束，必须在 1 到 10000 之间。那么你能找出负责解出 ID 的解析器（HandlerMethodArgumentResolver）是哪一种吗？校验又是如何触发的？

按照案例 1 的案例解析思路，我们可以轻松地找到负责解析 ID 值的解析器是 PathVariableMethodArgumentResolver，它的匹配要求参考如下代码：

[复制代码](#)

```
1 @Override
2 public boolean supportsParameter(MethodParameter parameter) {
3     if (!parameter.hasParameterAnnotation(PathVariable.class)) {
4         return false;
5     }
6     if (Map.class.isAssignableFrom(parameter.nestedIfOptional().getNestedParameterType())) {
7         PathVariable pathVariable = parameter.getParameterAnnotation(PathVariable.class);
8         return (pathVariable != null && StringUtils.hasText(pathVariable.value()));
9     }
10    //要返回true，必须标记@PathVariable注解
11    return true;
12 }
```


查看上述代码，当 String 类型的方法参数 ID 标记 @PathVariable 时，它就能符合上 PathVariableMethodArgumentResolver 的匹配条件。

翻阅这个解析类的实现，我们很快就可以定位到具体的解析方法，但是当我们顺藤摸瓜去找 Validation 时，却无蛛丝马迹，这点完全不同于案例 1 中的解析器 RequestResponseBodyMethodProcessor。那么它的校验到底是怎么触发的？你可以把这个问题当做课后作业去思考下，这里仅仅给出一个提示，实际上，对于这种直接标记在方法参数上的校验是通过 AOP 拦截来做校验的。

🔗 第 13 课

在案例 2 中，我们提到一定要避免在过滤器中调用多次 FilterChain#doFilter()。那么假设一个过滤器因为疏忽，在某种情况下，这个方法一次也没有调用，会出现什么情况呢？

这样的过滤器可参考改造后的 DemoFilter：

 复制代码

```
1 @Component
2 public class DemoFilter implements Filter {
3     public void doFilter(ServletRequest request, ServletResponse response, Fil
4         System.out.println("do some logic");
5     }
6 }
```

对于这样的情况，如果不了解 Filter 的实现逻辑，我们可能觉得，它最终会执行到 Controller 层的业务逻辑，最多是忽略掉排序在这个过滤器之后的一些过滤器而已。但是实际上，结果要严重得多。

以我们的改造案例为例，我们执行 HTTP 请求添加用户返回是成功的：

POST <http://localhost:8080/regStudent/fujian>

HTTP/1.1 200


Content-Length: 0

Date: Tue, 13 Apr 2021 11:37:43 GMT

Keep-Alive: timeout=60

Connection: keep-alive

但是实际上，我们的 Controller 层压根没有执行。这里给你解释下原因，还是贴出之前解析过的过滤器执行关键代码（ApplicationFilterChain#internalDoFilter）：

 复制代码

```
1 private void internalDoFilter(ServletRequest request,
2                               ServletResponse response){
3     if (pos < n) {
4         // pos会递增
5         ApplicationFilterConfig filterConfig = filters[pos++];
6         try {
7             Filter filter = filterConfig.getFilter();
8             // 省略非关键代码
9             // 执行filter
10            filter.doFilter(request, response, this);
11            // 省略非关键代码
12        }
13        // 省略非关键代码
14        return;
15    }
```

```
16         // 执行真正实际业务
17         servlet.service(request, response);
18     }
19     // 省略非关键代码
20 }
```

当我们的过滤器 DemoFilter 被执行，而它没有在其内部调用 FilterChain#doFilter 时，我们会执行到上述代码中的 return 语句。这不仅导致后续过滤器执行不到，也会导致能执行业务的 servlet.service(request, response) 执行不了。此时，我们的 Controller 层逻辑并未执行就不稀奇了。

相反，正是因为每个过滤器都显式调用了 FilterChain#doFilter，才有机会让最后一个过滤器在调用 FilterChain#doFilter 时，能看到 pos = n 这种情况。而这种情况下，return 就走不到了，能走到的是业务逻辑（servlet.service(request, response)）。

🔗 第 14 课

这节课的两个案例，它们都是在 Tomcat 容器启动时发生的，但你了解 Spring 是如何整合 Tomcat，使其在启动时注册这些过滤器吗？

当我们调用下述关键代码行启动 Spring 时：

```
1 SpringApplication.run(Application.class, args);
```

[📄 复制代码](#)


会创建一个具体的 ApplicationContext 实现，以 ServletWebServerApplicationContext 为例，它会调用 onRefresh() 来与 Tomcat 或 Jetty 等容器集成：

```
1 @Override
2 protected void onRefresh() {
3     super.onRefresh();
4     try {
5         createWebServer();
6     }
7     catch (Throwable ex) {
8         throw new ApplicationContextException("Unable to start web server", ex);
9     }
}
```

[📄 复制代码](#)

10 }

查看上述代码中的 createWebServer() 实现：

 复制代码

```
1 private void createWebServer() {
2     WebServer webServer = this.webServer;
3     ServletContext servletContext = getServletContext();
4     if (webServer == null && servletContext == null) {
5         ServletWebServerFactory factory = getWebServerFactory();
6         this.webServer = factory.getWebServer(getSelfInitializer());
7     }
8     // 省略非关键代码
9 }
```

第 6 行，执行 factory.getWebServer() 会启动 Tomcat，其中这个方法调用传递了参数 getSelfInitializer()，它返回的是一个特殊格式回调方法 this::selfInitialize 用来添加 Filter 等，它是当 Tomcat 启动后才调用的。

 复制代码

```
1 private void selfInitialize(ServletContext servletContext) throws ServletException {
2     prepareWebApplicationContext(servletContext);
3     registerApplicationScope(servletContext);
4     WebApplicationContextUtils.registerEnvironmentBeans(getBeanFactory(), servletContext);
5     for (ServletContextInitializerBeans beans : getServletContextInitializerBeans())
6         beans.onStartup(servletContext);
7 }
8 }
```

那说了这么多，你可能对这个过程还不够清楚，这里我额外贴出了两段调用栈帮助你理解。

1. 启动 Spring Boot 时，启动 Tomcat：

```
start:459, Tomcat (org.apache.catalina.startup)
initialize:107, TomcatWebServer (org.springframework.boot.web.embedded.tomcat)
<init>:88, TomcatWebServer (org.springframework.boot.web.embedded.tomcat)
getTomcatWebServer:438, TomcatServletWebServerFactory (org.springframework.boot.web.embedded.tomcat)
getWebServer:191, TomcatServletWebServerFactory (org.springframework.boot.web.embedded.tomcat)
createWebServer:180, ServletWebServerApplicationContext (org.springframework.boot.web.servlet.context)
onRefresh:153, ServletWebServerApplicationContext (org.springframework.boot.web.servlet.context)
refresh:544, AbstractApplicationContext (org.springframework.context.support)
refresh:141, ServletWebServerApplicationContext (org.springframework.boot.web.servlet.context)
refresh:747, SpringApplication (org.springframework.boot)
refreshContext:397, SpringApplication (org.springframework.boot)
run:315, SpringApplication (org.springframework.boot)
run:1226, SpringApplication (org.springframework.boot)
run:1215, SpringApplication (org.springframework.boot)
```

2. Tomcat 启动后回调 selfInitialize :

```
selfInitialize:224, ServletWebServerApplicationContext (org.springframework.boot.web.servlet.context)
onStartup:-1, 17815179 (org.springframework.boot.web.servlet.context.ServletWebServerApplicationContext$$Lambda$
onStartup:53, TomcatStarter (org.springframework.boot.web.embedded.tomcat)
startInternal:5135, StandardContext (org.apache.catalina.core)
start:183, LifecycleBase (org.apache.catalina.util)
call:1384, ContainerBase$StartChild (org.apache.catalina.core)
call:1374, ContainerBase$StartChild (org.apache.catalina.core)
run$$$capture:266, FutureTask (java.util.concurrent)
run:-1, FutureTask (java.util.concurrent)
```

相信通过上述调用栈，你能更清晰地理解 Tomcat 启动和 Filter 添加的时机了。

🔗 第 15 课

通过案例 1 的学习，我们知道在 Spring Boot 开启 Spring Security 时，访问需要授权的 API 会自动跳转到如下登录页面，你知道这个页面是如何产生的么？

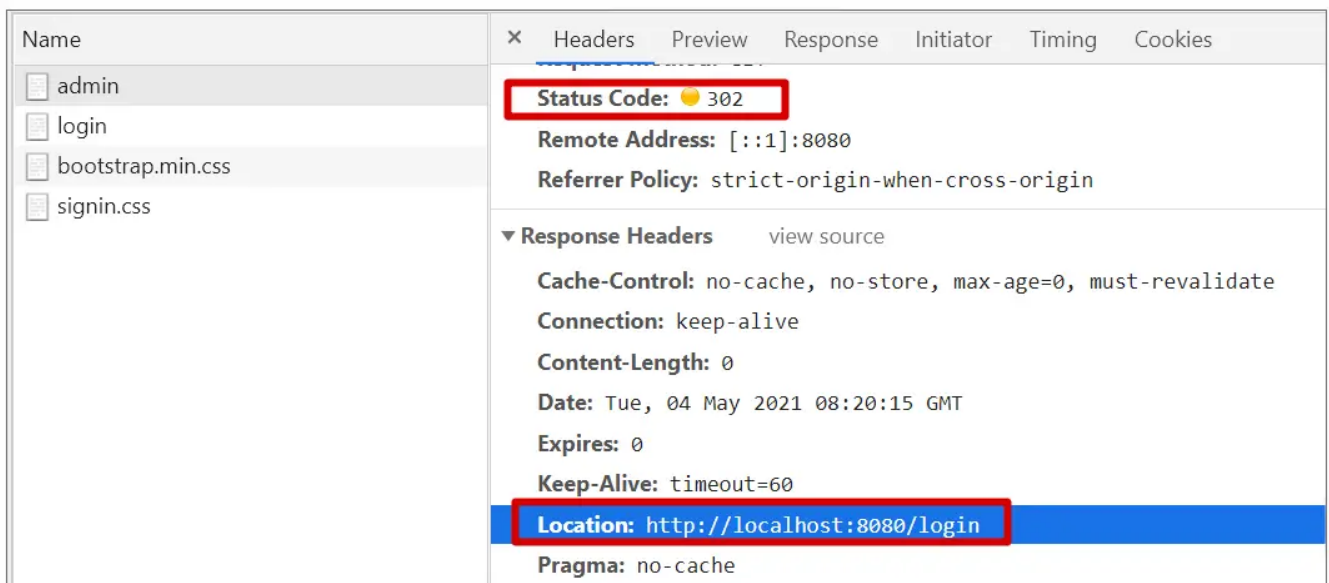


实际上，在 Spring Boot 启用 Spring Security 后，匿名访问一个需要授权的 API 接口时，我们会发现这个接口授权会失败，从而进行 302 跳转，跳转的关键代码可参考 `ExceptionTranslationFilter` 调用的 `LoginUrlAuthenticationEntryPoint#commence` 方法：

[复制代码](#)

```
1 public void commence(HttpServletRequest request, HttpServletResponse response,  
2     AuthenticationException authException) throws IOException, ServletException  
3     //省略非关键代码  
4     redirectUrl = buildRedirectUrlToLoginPage(request, response, authException)  
5     //省略非关键代码  
6     redirectStrategy.sendRedirect(request, response, redirectUrl);  
7 }
```

具体的跳转情况可参考 Chrome 的开发工具：



在跳转后，新的请求最终看到的效果图是由下面的代码生产的 HTML 页面，参考 `DefaultLoginPageGeneratingFilter#generateLoginPageHtml`：

[复制代码](#)

```
1 private String generateLoginPageHtml(HttpServletRequest request, boolean login  
2     boolean logoutSuccess) {  
3     String errorMsg = "Invalid credentials";  
4     //省略部分非关键代码  
5  
6     StringBuilder sb = new StringBuilder();  
7     sb.append("<!DOCTYPE html>\n")
```


```
8         + "<html lang=\"en\">\n"
9         + "   <head>\n"
10        + "     <meta charset=\"utf-8\">\n"
11        + "     <meta name=\"viewport\" content=\"width=device-width, initial-"
12        + "     <meta name=\"description\" content=\"\">\n"
13        + "     <meta name=\"author\" content=\"\">\n"
14        + "     <title>Please sign in</title>\n"
15        + "     <link href=\"https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-b"
16        + "     <link href=\"https://getbootstrap.com/docs/4.0/examples/signin"
17        + "   </head>\n"
18        + "   <body>\n"
19        + "     <div class=\"container\">\n");
20    //省略部分非关键代码
21    sb.append("</div>\n");
22    sb.append("</body></html>");
23
24    return sb.toString();
25 }
```

上即为登录页面的呈现过程，可以看出基本都是由各种 Filter 来完成的。

第 16 课

这节课的两个案例，在第一次发送请求的时候，会遍历对应的资源处理器和异常处理器，并注册到 DispatcherServlet 对应的类成员变量中，你知道它是如何被触发的吗？

实现了 FrameworkServlet 的 onRefresh() 接口，这个接口会在 WebApplicationContext 初始化时被回调：

 复制代码

```
1 public class DispatcherServlet extends FrameworkServlet {
2     @Override
3     protected void onRefresh(ApplicationContext context) {
4         initStrategies(context);
5     }
6
7     /**
8      * Initialize the strategy objects that this servlet uses.
9      * <p>May be overridden in subclasses in order to initialize further strategy
10     */
11     protected void initStrategies(ApplicationContext context) {
12         initMultipartResolver(context);
13         initLocaleResolver(context);
14         initThemeResolver(context);
15         initHandlerMappings(context);
16         initHandlerAdapters(context);
```



```
17     initHandlerExceptionResolvers(context);
18     initRequestToViewNameTranslator(context);
19     initViewResolvers(context);
20     initFlashMapManager(context);
21 }
22 }
```

以上就是这次答疑的全部内容，我们下一章节再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

👍 赞 3 💡 提建议

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

上一篇 16 | Spring Exception 常见错误

下一篇 18 | Spring Data 常见错误

更多学习推荐

Java 面试必考 300 题
最新汇总

限时免费领取 📌

精选留言

💬 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。