Spring MVC的高级技术

1 Spring MVC配置的替代方案

1.1 自定义DIspatcherServlet配置

之前项目中用到的AbstractAnnotationConfigDispatcherServletInitializer

```
public class SpitterWebInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{RootConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{WebConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

}
```

以上三个方法是必须重载的，此外还有：

- **customizeRegistration** 在AbstractAnnotationConfigDispatcherServletInitializer将DispatcherServlet注册到Servlet容器之后，就会调用该方法。并将Servlet注册后得到的ServletRegistration.Dynamic传递进来，可以对DispatcherServlet进行额外配置

```
@Override
protected void customizeRegistration(ServletRegistration.Dynamic registration) {
    registration.setMultipartConfig(
            new MultipartConfigElement("/tmp/spittr/uploads", 2097152, 4194304,
0));
}
```

通过setMultipartConfig配置Servlet对multipart的支持，将上传文件的临时存储目录设置在”/tmp/spittr/uploads“中。

1.2 添加其他Servlet和Filter

重载AbstractAnnotationConfigDispatcherServletInitializer的方法

```
public class WebInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
```

```
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{RootConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[]{WebConfig.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    protected void customizeRegistration(ServletRegistration.Dynamic
registration) {
        registration.setMultipartConfig(
                new MultipartConfigElement("/tmp/spittr/uploads", 2097152,
4194304, 0));
    }

    @Override
    protected void registerDispatcherServlet(ServletContext servletContext) {
        super.registerDispatcherServlet(servletContext);
    }

    @Override
    protected Filter[] getServletFilters() {
        return super.getServletFilters();
    }
}
```

2 处理multipart形式额数据

用于文件上传

2.1 配置multipart解析器

Spring内置的MUltipartResolver：

- **CommonsMultipartResolver:** 是哟共Jakarta COmmons FileUpload解析multipart请求
- **StandardServletMultipartResolver**：依赖于Servlet对multipart请求的支持

```
@Bean
public MultipartResolver multipartResolver() throws IOException {
    return new StandardServletMultipartResolver();
}
```

在web.xml或Servlet初始化挂类中，将multipart的具体细节作为DIspatcherServlet配置的一部分

如果采用Servlet初始化类的方式来配置DispatcherServlet，这个初始化类会实现WebApplicationInitializer，可以在Servlet registration上调用setMultipartConfig()方法，传入一个MultipartConfigElement实例

```
@Override
protected void customizeRegistration(ServletRegistration.Dynamic registration) {
    registration.setMultipartConfig(
            new MultipartConfigElement("/tmp/spittr/uploads", 2097152, 4194304,
0));
}
```

2.2 处理multipart请求

控制器方法接受上传文件，需要在方法参数上添加@RequestPart

```
@RequestMapping(value = "/register", method = POST)
public String processRegistration(
    @RequestPart(value = "profilePictures", required = false) Part fileBytes,
    RedirectAttributes redirectAttributes,
    @Valid Spitter spitter,
    Errors errors) throws IOException {
    if (errors.hasErrors()) {
        return "registerForm";
    }

    spitterRepository.save(spitter);
    redirectAttributes.addAttribute("username", spitter.getUsername());
    redirectAttributes.addFlashAttribute(spitter);

    fileBytes.write("/tmp/spittr/" + fileBytes.getSubmittedFileName());

    return "redirect:/spitter/" + spitter.getUsername();
}
```

Part包含了上传文件的信息。

3 处理异常

处理请求的时候，不管发生什么事情，Servlet请求的输出都是一个Servlet响应。如果处理发生了异常，它的输出依然回事Servlet响应。异常必须要以某种方式转换为响应。

- **特定的Spring异常将会自动映射为指定的HTTP状态码**
- **异常上可以添加@ResponseStatus注解，从而将其映射为某一个HTTP状态码**
- **在方法上可以添加@ExceptionHandler注解，使其用来处理异常**

3.1 将异常映射为HTTP状态码

Spring异常	HTTP状态码
BindException	400 - Bad Request
ConversionNotSupportedException	500 - Internal Server Error
HttpMediaTypeNotAcceptableException	406 - Not Acceptable
HttpMediaTypeNotSupportedException	415 - Unsupported Media Type
HttpMessageNotReadableException	400 - Bad Request
HttpMessageNotWritableException	500 - Internal Server Error
HttpRequestMethodNotSupportedException	405 - Method Not Allowed
MethodArgumentNotValidException	400 - Bad Request
MissingServletRequestParameterException	400 - Bad Request
MissingServletRequestPartException	400 - Bad Request
NoSuchRequestHandlingMethodException	404 - Not Found
TypeMismatchException	400 - Bad Request

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Spittle Not Found")
public class SpittleNotFoundException extends RuntimeException {
}
```

```
@RequestMapping(value = "/{spittleId}", method = RequestMethod.GET)
public String spittle(@PathVariable("spittleId") long spittleId,Model model) {
    Spittle spittle = spittleRepository.findOne(spittleId);
    if (spittle == null) {
        throw new SpittleNotFoundException();
    }
    model.addAttribute(spittle);
    return "spittle";
}
```

3.2 编写异常处理的方法

如果想要在响应中不仅要包括状态码，还要包含所产生的错误，此时不能将一场视为HTTP错误，而是要按照处理请求的方式来处理异常。

```
@RequestMapping(method = RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
    try {
        spittleRepository.save(new Spittle(null, form.getMessage(), new Date(),
                form.getLongitude(), form.getLatitude()));
        return "redirect:/spittles";
    } catch (DuplicateSpittleException e) {
        return "error/duplicate";
    }
}
```

```
public class DuplicateSpittleException extends RuntimeException {
}
```

以上方法自己处理异常，

将异常处理方法剥离,saveSpittle()只关注正常路径

```
@RequestMapping(method = RequestMethod.POST)
public String saveSpittle(SpittleForm form, Model model) {
    spittleRepository.save(new Spittle(null, form.getMessage(), new Date(),
                                       form.getLongitude(),
form.getLatitude()));
    return "redirect:/spittles";
}

@ExceptionHandler(DuplicateSpittleException.class)
public String handleDuplicateSpittle() {
    return "error/duplicate";
}
```

handleDuplicateSpittle() 方法添加了@ExceptionHandler，当抛出DuplicateSpittleException异常时候，将会委托该方法来处理。他的返回值是String，制定了要渲染的逻辑视图名。

@ExceptionHandler 他能处理同一个空间中所有处理器方法抛出的该类型的异常。即：能够处理SpittleController中所有方法抛出的DuplicateSpittleException异常

4 为控制器添加通知

控制器类的特定切面能够运用大整个应员工程序的所有控制器中。。如果多个控制器类中都会抛出某个特定的异常，那么可能会发现要在所有控制器方法中重复相同的@ExceptionHandler。或者**为了避免重复，需要创建一个基础的控制类，所有控制类要扩展这个类，从而继承通用的@ExceptionHandler方法**

控制器通知 (constroller advice)：是任意带有@ControllerAdvice注解的类，这个类带有如下类型的方法：

- **@ExceptionHandler 标注的方法**
- **@InitBinder 标注的方法**
- **@ModelAttribute 标注的方法**

在带有

- @ControllerAdvice注解的类中，以上方法都会运用到真个应用程序所有可供之气中带有@RequestMapping注解的方法上
- @ControllerAdvice本身已经是一个你了@Component，因此@ControllerAdvice注解所标注的类将会自动被组件扫描获取到
- @ControllerAdvice最为实用的一个场景就是将所有@ExceptionHandler方法收集到一个类中，这样所有控制器的异常就能在一个地方进行一致的处理。

```
@ControllerAdvice
public class AppWideExceptionHandler {

    @ExceptionHandler(DuplicateSpittleException.class)
    public String duplicateSpittleHandler() {
        return "error/duplicate";
    }
}
```

任意的Controller控制器方法抛出了DuplicateSpittleException，都会调用这个duplicateSpittleHandler()方法来处理异常

5 跨重定向请求传递数据

借助”redirect：“前缀，当控制器方法返回的String以”redirect：“开头，那么这个string不是用来查找试图的，而是用来指导浏览器进行重定向的路径。

一般来讲，当一个处理器方法完成后，该方法所指定的模型数据及那个会发知道请求中，并作为请求中的属性，请求会转发到视图上进行渲染。因为控制器方法和视图所处理的是同一个请求，素以再转发的国政中，请求属性能够得以保存。

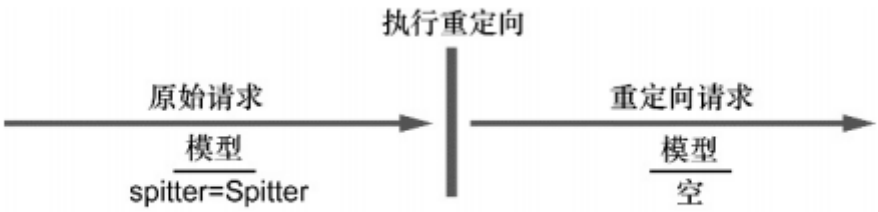

图7.1 模型的属性是以请求属性的形式存放在请求中的，在重定向后无法存活

显然，对于重定向来说，模型并蹦年用来传递数据。解决方案：

- **使用URL模板以路径变量和/或查询参数的形式传递数据**
- **通过flash属性发送数据**

5.1 通过URI模板进行重定向

```
@RequestMapping(value = "/register", method = POST)
public String ptoccessRegistration(Spitter spitter, Model model) {
    spitterRepository.save(spitter);
    model.addAttribute("username", spitter.getUsername());
    model.addAttribute("spittleId", spitter.getId());
    return "redirect:/spitter/{username}";
}
```

模型中的spitterId属性没有匹配重定向RUIL中的任何占位符，所以它会自动以查询参数的形式发家到重定向URL上。eg /spitter/wuu?spitterId=42

5.2 使用flash属性

将Spitter对下个放到一个位置，使其能够在重定向的过程中存活下来

Spring将数据发送为flash属性的共呢个，flash属性会一直携带这些数据直到下次请求，然后才会消失。

Spring通过RedirectAttributes设置flash属性的方法，该方法数Model的子接口。RedirectAttributes提供了Model的所有功能。

```
@RequestMapping(value = "/register", method = POST)
public String ptoccessRegistration(Spitter spitter, RedirectAttributes model) {
    spitterRepository.save(spitter);
    model.addAttribute("username", spitter.getUsername());
    model.addFlashAttribute("spitter", spitter);
    return "redirect:/spitter/{username}";
}
```

addFlashAttribute可以不设置key参数，让key根据值的类型自行推断得出

在重定向之前，所有的flash属性都会复制到会话过程中。在重定向后，存在会话中的flash属性会被去除，并从会话转移到模型之中，处理重定向的方法就能从模型中访问Spitter对象了，就像获取其他的模型对象一样

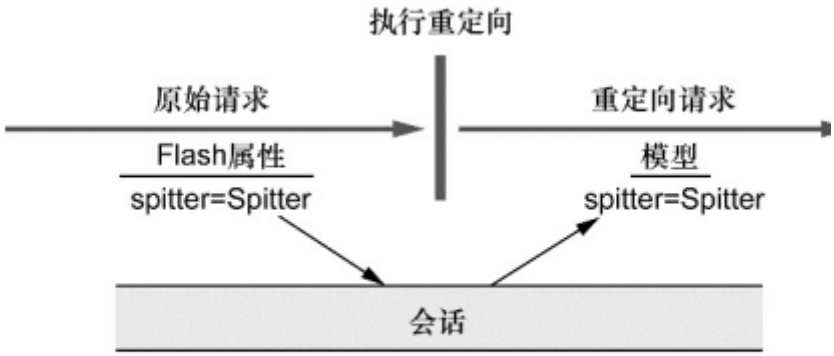

图7.2 flash属性保存在会话中，然后再放到模型中，因此能够在重定向的过程中存活

```
@RequestMapping(value = "/{username}", method = GET)
public String showSpitterProfile(
        @PathVariable String username, Model model) {
    if (!model.containsAttribute("spitter")) {
        model.addAttribute(spitterRepository.findByUsername(username));
    }
    return "profile";
}
```