朱晔和你聊 Spring 系列 S1E3：灵活但不算好用的 Spring MVC

本文会以一些例子来展现 Spring MVC 的常见功能和一些扩展点，然后我们来讨论一下 Spring MVC 好用不好用。

使用 SpringBoot 快速开始

基于之前的 parent 模块，我们来创建一个新的模块：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://maven.apache.org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>me.josephzhu</groupId>
    <artifactId>spring101-webmvc</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>spring101-webmvc</name>
    <description></description>

    <parent>
        <groupId>me.josephzhu</groupId>
        <artifactId>spring101</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </parent>


    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

使用 web 来启用 Spring MVC，使用 thymeleaf 来启用 thymeleaf 模板引擎。Thymeleaf 是一个强大的 Java 模板引擎，可以脱离于 Web 单独使用，本身就有非常多的可配置可扩展的点，这里不展开讨论，详见官网。

接下去我们创建主程序：

```
package me.josephzhu.spring101webmvc;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Spring101WebmvcApplication {

    public static void main(String[] args) {
        SpringApplication.run(Spring101WebmvcApplication.class, args);
    }
}
```

以及一个测试 Controller：

```
package me.josephzhu.spring101webmvc;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.stream.Collectors;
import java.util.stream.IntStream;

@Controller
public class MyController {
```

```
    @GetMapping("shop")
    public ModelAndView shop() {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("shop");
        modelAndView.addObject("items",
                IntStream.range(1, 5)
                        .mapToObj(i -> new MyItem("item" + i, i * 100))
                        .collect(Collectors.toList()));
        return modelAndView;
    }

}
```

这里使用到了一个自定义的类：

```
package me.josephzhu.spring101webmvc;

import lombok.AllArgsConstructor;
import lombok.Data;

@AllArgsConstructor
@Data
public class MyItem {
    private String name;
    private Integer price;
}
```

最后我们需要在 resources 目录下创建一个 templates 目录，在目录下再创建一个 shop.html 模板文件：

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello Shop</title>
</head>
<body>
Hello Shop
<table>
    <tr th:each="item : ${items}">
        <td th:text="${item.name}">...</td>
        <td th:text="${item.price}">...</td>
    </tr>
</table>
</body>
</html>
```

我们看到有了 SpringBoot，创建一个 Spring MVC 程序整个过程非常简单：

1 引入 starter
2 创建@Controller，设置@RequestMapping
3 创建模板文件

没有任何配置工作，一切都是 starter 自动配置。

快速配置 ViewController

几乎所有 Spring MVC 的扩展点都集成在了接口中，要进行扩展很简单，实现这个接口，加上@Configuration 和@EnableWebMvc 注解，实现需要的方法即可。

我们先用它来快速配置一些 ViewController：

```
package me.josephzhu.spring101webmvc;

import org.springframework.context.annotation.Configuration;
import org.springframework.format.FormatterRegistry;
import org.springframework.http.HttpStatus;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.servlet.config.annotation.*;
import org.springframework.web.servlet.resource.GzipResourceResolver;
import org.springframework.web.servlet.resource.VersionResourceResolver;

import java.util.List;

@EnableWebMvc
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/hello").setViewName("helloworld");
        registry.addRedirectViewController("/", "/hello");
        registry.addStatusController("/user", HttpStatus.BAD_REQUEST);
    }
}
```

代码中多贴了一些后面会用到的 import 在这里可以忽略。这里我们配置了三套策略：

1 访问/会跳转到/hello
2 访问/hello 会访问 helloworld 这个 view
3 访问/user 会给出 400 的错误代码

这里我们在 templats 目录再添加一个空白的 helloworld.html：

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
<body>
Hello World
</body>
</html>
```

这种配置方式可以省一些代码量，但是我个人认为在这里做配置可读性一般。

定制路径匹配

我们还可以实现路径匹配策略的定制：

```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    configurer.setUseTrailingSlashMatch(false);
}
```

比如这样就关闭了结尾为/的匹配（默认开启）。试着访问 http://localhost:8080/shop/得到如下错误：

```
2018-10-02 18:58:16.581  WARN 20264 --- [nio-8080-exec-1] o.s.web.servlet.PageNotFound             : No mapping found for HTTP request with URI [/shop/] in DispatcherServlet with name 'dispatcherServlet'
```

这个方法可以针对路径匹配进行相当多的配置，具体请参见文档，这里只列出了其中的一个功能。

配置静态资源

在配置类加上下面的代码：

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/static/**")
            .addResourceLocations("classpath:/static/")
            .resourceChain(true)
            .addResolver(new GzipResourceResolver())
            .addResolver(new VersionResourceResolver()
                    .addFixedVersionStrategy("1.0.0", "/**"));
}
```

这就实现了静态资源路由到 static 目录，并且为静态资源启用了 Gzip 压缩和基于版本号的缓存。配置后我们在 resources 目录下创建一个 static 目录，然后随便创建一个 a.html 文件，试试访问这个文件，测试可以发现：http://localhost:8080/static/1.0.0/a.html 和 http://localhost:8080/static/a.html 都可以访问到这个文件。

解析自定义的参数

HandlerMethodArgumentResolver 接口这是一个非常非常重要常用的扩展点。通过这个接口，我们可以实现通用方法来装配 HandlerMethod 上的自定义参数，我们现在来定义一个 MyDevice 类型，然后我们希望框架可以在所有出现 MyDevice 参数的时候自动为我们从 Header 里获取相应的设备信息构成 MyDevice 对象（如果我们 API 的使用者是客户端应用程序，这是不是一个挺常见的需求）。

```
package me.josephzhu.spring101webmvc;

import lombok.Data;

@Data
public class MyDevice {
    private String type;
    private String version;
    private String screen;
}
```

然后是自定义的 HandlerMethodArgumentResolver 实现：

```
package me.josephzhu.spring101webmvc;

import org.springframework.core.MethodParameter;
import org.springframework.web.bind.support.WebDataBinderFactory;
import org.springframework.web.context.request.NativeWebRequest;
import org.springframework.web.method.support.HandlerMethodArgumentResolver;
import org.springframework.web.method.support.ModelAndViewContainer;

public class DeviceHandlerMethodArgumentResolver implements
HandlerMethodArgumentResolver {
    @Override
    public boolean supportsParameter(MethodParameter methodParameter) {
        return methodParameter.getParameterType().equals(MyDevice.class);
    }

    @Override
    public Object resolveArgument(MethodParameter methodParameter,
ModelAndViewContainer modelAndViewContainer, NativeWebRequest nativeWebRequest,
```

```
WebDataBinderFactory webDataBinderFactory) throws Exception {
        MyDevice myDevice = new MyDevice();
        myDevice.setType(nativeWebRequest.getHeader("device.type"));
        myDevice.setVersion(nativeWebRequest.getHeader("device.version"));
        myDevice.setScreen(nativeWebRequest.getHeader("device.screen"));
        return myDevice;
    }
}
```

实现分两部分，第一部分告诉框架，我们这个 ArgumentResolver 支持解析怎么样的参数。这里我们的实现是根据参数类型，还有很多时候可以通过检查是否参数上有额外的自定义注解来实现（后面也会有例子）。第二部分就是真正的实现了，实现非常简单，从请求头里获取相应的信息构成我们的 MyDevice 对象。

要让这个 Resolver 被 MVC 框架识别到，我们需要继续扩展刚才的 WebConfig 类，加入下面的代码：

```
@Override
public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {
    resolvers.add(new DeviceHandlerMethodArgumentResolver());
}
```

然后，我们写一个例子来测试一下：

```
package me.josephzhu.spring101webmvc;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

@RestController
@Slf4j
@RequestMapping("api")
public class MyRestController {

    @RequestMapping(value = "items", method = RequestMethod.GET)
    public List<MyItem> getItems(MyDevice device) {
        log.debug("Device : " + device);
        List<MyItem> myItems = new ArrayList<>();
        myItems.add(new MyItem("aa", 10));
        myItems.add(new MyItem("bb", 20));
        return myItems;
    }
```

```
}
```

这里因为用了 debug，所以需要在配置文件中打开 debug 日志级别：

```
logging.level.me.josephzhu.spring101webmvc=DEBUG
```

测试一下：

```
curl -X GET \
  http://localhost:8080/api/items \
  -H 'device.screen: 1280*800' \
  -H 'device.type: android' \
  -H 'device.version: 1.1'
```

可以在控制台看到这样的日志：

```
2018-10-02 19:10:56.667 DEBUG 20325 --- [nio-8080-exec-9] m.j.spring101webmvc.MyRestController        : Device : MyDevice(type=android, version=1.1, screen=1280*800)
```

可以证明我们方法中定义的 MyDevice 的确是从请求中获取到了正确的结果。大家可以发挥一下想象，ArgumentResolver 不但可以做类似参数自动装配（从各个地方获取必要的数据）的工作，而且还可以做验证工作。大家可以仔细看一下 resolveArgument 方法的参数，是不是相当于要啥有啥了（当前参数定义、当前请求、Model 容器以及绑定工厂）。

自定义 ResponseBody 后处理

在刚才的实现中，我们直接返回了 List<MyItem>数据，对于 API 来说，我们一般会定义一套 API 的结果对象，包含 API 的数据、成功与否结果、错误消息、签名等等内容，这样客户端可以做签名验证，然后是根据成功与否来决定是要解析数据还是直接提示错误，比如：

```
package me.josephzhu.spring101webmvc;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class APIResponse<T> {
    T data;
```

```
    boolean success;
    String message;
    String sign;
}
```

如果我们在每个 API 方法中去返回这样的 APIResponse 当然可以实现这个效果，还有一种通用的实现方式是使用 ResponseBodyAdvice：

```
package me.josephzhu.spring101webmvc;

import org.springframework.core.MethodParameter;
import org.springframework.http.MediaType;
import org.springframework.http.converter.HttpMessageConverter;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;

@ControllerAdvice
public class APIResponseBodyAdvice implements ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter returnType, Class<? extends
HttpMessageConverter<?>> converterType) {
        return !returnType.getParameterType().equals(APIResponse.class);
    }

    @Override
    public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType
selectedContentType, Class<? extends HttpMessageConverter<?>> selectedConverterType,
ServerHttpRequest request, ServerHttpResponse response) {
        String sign = "";
        Sign signAnnotation = returnType.getMethod().getAnnotation(Sign.class);
        if (signAnnotation != null)
            sign = "abcd";
        return new APIResponse(body, true, "", sign);
    }
}
```

通过定义@ControllerAdvice 注解来启用这个 Advice。在实现上也是两部分，第一部分告诉框架我们这个 Advice 支持的是非 APIResponse 类型（如果返回的对象已经是 APIResponse 了，我们当然就不需要再包装一次了）。第二部分是实现，这里的实现很简单，我们先检查一下方法上是否有 Sign 这个注解，如果有的话进行签名（这里的逻辑是写死的签名），然后把得到的 body 塞入 APIResponse 后返回。

这里补上 Sign 注解的实现：

```java
package me.josephzhu.spring101webmvc;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Sign {

}
```

这是一个空注解，没啥可以说的，下面我们来测试一下这个 ResponseBodyAdvice：

```java
@RequestMapping(value = "item/{id}", method = RequestMethod.GET)
public MyItem getItem(@PathVariable("id") String id) {
    Integer i = null;
    try {
        i = Integer.parseInt(id);
    } catch (NumberFormatException ex) {
    }
    if (i == null || i < 1)
        throw new IllegalArgumentException("不合法的商品 ID");
    return new MyItem("item" + id, 10);
}
```

访问 http://localhost:8080/api/item/23 后得到如下图的结果：

是不是很方便呢？这个 API 包装的过程可以由框架进行，无需每次手动来做。

自定义异常处理

如果我们访问 http://localhost:8080/api/item/0 会看到错误白页，针对错误处理，我们希望：

1 可以使用统一的 APIResponse 方式进行错误返回
2 可以记录错误信息以便查看

实现这个功能非常简单，我们可以通过@ExceptionHandler 实现：

```
package me.josephzhu.spring101webmvc;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.method.HandlerMethod;

import javax.servlet.http.HttpServletRequest;

@ControllerAdvice(annotations = RestController.class)
@Slf4j
public class MyRestExceptionHandler {
    @ExceptionHandler
    @ResponseBody
    public APIResponse handle(HttpServletRequest req, HandlerMethod method, Exception
ex) {
        log.error(String.format("访问 %s -> %s 出错了！", req.getRequestURI(),
method.toString()), ex);
        return new APIResponse(null, false, ex.getMessage(), "");
    }
}
```

注意几点：

1 我们可以使用@ControllerAdvice 的 annotations 来关联我们需要拦截的 Controller 类型
2 handle 方法支持相当多的参数，可谓是要啥有啥，这里贴下官方文档说明的截图（在这里我们使用了 ServletRequest 来获取请求地址，使用了 HandlerMethod 来获取当前执行的方法）：

Method Arguments

`@ExceptionHandler` methods support the following arguments:

Method argument	Description
Exception type	For access to the raised exception.
`HandlerMethod`	For access to the controller method that raised the exception.
`WebRequest`, `NativeWebRequest`	Generic access to request parameters and request and session attributes without direct use of the Servlet API.
`javax.servlet.ServletRequest`, `javax.servlet.ServletResponse`	Choose any specific request or response type (for example, `ServletRequest` or `HttpServletRequest` or or Spring's `MultipartRequest` or `MultipartHttpServletRequest`).
`javax.servlet.http.HttpSession`	Enforces the presence of a session. As a consequence, such an argument is never `null`. Note that session access is not thread-safe. Consider setting the `RequestMappingHandlerAdapter` instance's `synchronizeOnSession` flag to `true` if multiple requests are allowed to access a session concurrently.
`java.security.Principal`	Currently authenticated user — possibly a specific `Principal` implementation class if known.
`HttpMethod`	The HTTP method of the request.
`java.util.Locale`	The current request locale, determined by the most specific `LocaleResolver` available — in effect, the configured `LocaleResolver` or `LocaleContextResolver`.

访问地址 http://localhost:8080/api/item/sd 可以看到如下输出：


```
{
    "data": null,
    "success": false,
    "message": "不合法的商品ID",
    "sign": ""
}
```


（注意，处理签名的 ResponseBodyAdvice 并不会针对这个返回进行处理，因为之前实现的时候我们就判断了返回内容不是 APIResponse 才去处理，在自己正式的实现中你可以实现的更合理，让签名的处理逻辑同时适用出现异常的情况）日志中也出现了错误信息：

2018-10-02 19:48:41.450 ERROR 20422 --- [nio-8080-exec-6] m.j.s.MyRestExceptionHandler : 访问 /api/item/sd -> public me.josephzhu.spring101webmvc.MyItem me.josephzhu.spring101webmvc.MyRestController.getItem(java.lang.String) 出错了！

java.lang.IllegalArgumentException: 不合法的商品 ID

at me.josephzhu.spring101webmvc.MyRestController.getItem(MyRestController.java:34) ~[classes/:na]

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_161]

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_161]

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_161]

at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_161]

at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:209) ~[spring-web-5.0.9.RELEASE.jar:5.0.9.RELEASE]

自动处理参数类型转换

比如有这么一个需求，我们希望可以接受自定义的枚举作为参数，而且枚举的名字不一定需要和请求的参数完全大小写匹配，这个时候我们需要实现自己的转换器：

```
package me.josephzhu.spring101webmvc;

import org.springframework.core.convert.converter.Converter;
import org.springframework.core.convert.converter.ConverterFactory;

import java.util.Arrays;

public class MyConverterFactory implements ConverterFactory<String, Enum> {

    @Override
    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {
        return new String2EnumConverter(targetType);
    }

    class String2EnumConverter<T extends Enum<T>> implements Converter<String, T> {

        private Class<T> enumType;

        private String2EnumConverter(Class<T> enumType) {
            this.enumType = enumType;
        }

        @Override
        public T convert(String source) {
            return Arrays.stream(enumType.getEnumConstants())
                    .filter(e -> e.name().equalsIgnoreCase(source))
                    .findAny().orElse(null);
```

```
        }
    }
}
```

这里实现了一个从字符串到自定义枚举的转换，在搜索枚举名字的时候我们忽略了大小写。

接下去我们通过 WebConfig 来注册这个转换器工厂：

```
@Override
public void addFormatters(FormatterRegistry registry) {
    registry.addConverterFactory(new MyConverterFactory());
}
```

来写一段代码测试一下：

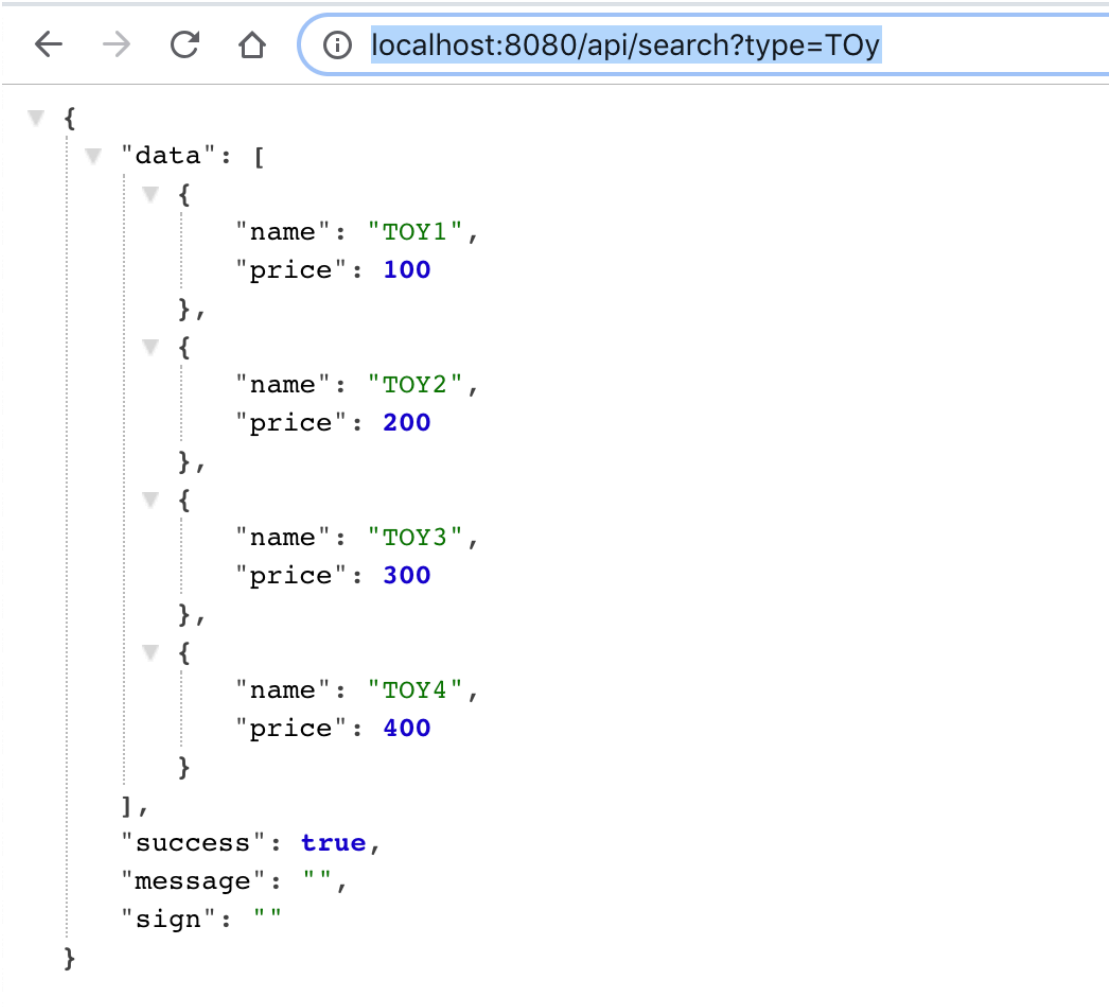
```
@GetMapping("search")
public List<MyItem> search(@RequestParam("type") ItemTypeEnum itemTypeEnum) {
    return IntStream.range(1, 5)
            .mapToObj(i -> new MyItem(itemTypeEnum.name() + i, i * 100))
            .collect(Collectors.toList());
}
```

这是一个 Get 请求的 API，接受一个 type 参数，参数是一个自定义枚举：

```
package me.josephzhu.spring101webmvc;

public enum ItemTypeEnum {
    BOOK, TOY, TOOL
}
```

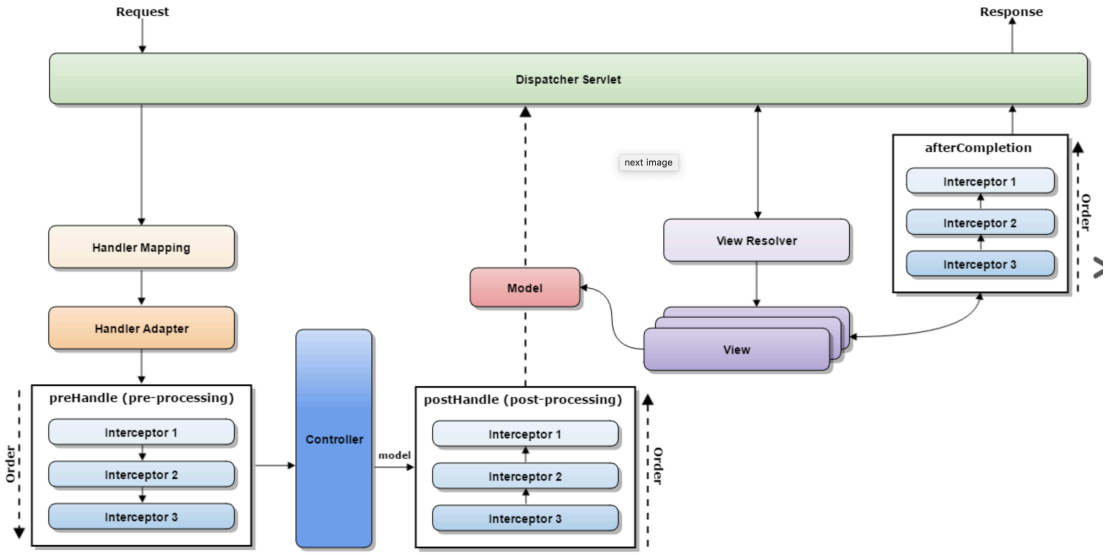
很明显枚举的名字都是大写的，我们来访问一下地址 http://localhost:8080/api/search?type=TOy 测试一下程序是否可以正确匹配：



TOy 的搜索参数匹配到了 TOY 枚举，结果符合我们的预期。

自定义拦截器

最后，我们来看看 Spring MVC 最通用的扩展点，也就是拦截器。



这个图清晰展现了拦截器几个重要方法事件节点。在这个例子中，我们利用 preHandle 和 postHandle 两个方法实现可以统计请求执行耗时的拦截器：

```
package me.josephzhu.spring101webmvc;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Slf4j
public class ExecutionTimeHandlerInterceptor extends HandlerInterceptorAdapter {

    private static final String START_TIME_ATTR_NAME = "startTime";
    private static final String EXECUTION_TIME_ATTR_NAME = "executionTime";

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute(START_TIME_ATTR_NAME, startTime);
        return true;
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
```

```
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        long startTime = (Long) request.getAttribute(START_TIME_ATTR_NAME);
        long endTime = System.currentTimeMillis();
        long executionTime = endTime - startTime;

        String time = "[" + handler + "] executeTime : " + executionTime + "ms";

        if (modelAndView != null) {
            modelAndView.addObject(EXECUTION_TIME_ATTR_NAME, time);
        }

        log.debug(time);
    }
}
```

在实现的时候，我们不仅仅把执行时间输出到了日志，而且还通过修改 ModelAndView 对象把这个信息加入到了视图模型内，这样页面也可以展现这个时间。要启用拦截器，我们还需要配置 WebConfig：

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new ExecutionTimeHandlerInterceptor());
}
```

接下去我们运行刚才那个例子，可以看到如下的日志输出：

```
2018-10-02 19:58:22.189 DEBUG 20422 --- [nio-8080-exec-9] m.j.s.ExecutionTimeHandlerInterceptor       : [public
java.util.List<me.josephzhu.spring101webmvc.MyItem>
me.josephzhu.spring101webmvc.MyRestController.search(me.josephzhu.spring101webmvc.ItemTypeEnum)]
executeTime : 22ms
```

页面上也可以引用到我们添加进去的对象：

```
<!DOCTYPE html SYSTEM "http://www.thymeleaf.org/dtd/xhtml1-strict-thymeleaf-4.dtd">
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>
<body>
Hello World
<div th:text="${executionTime}"></div>
```

```
</body>
</html>
```

拦截器是非常通用的一个扩展，可以全局实现权限控制、缓存、动态修改结果等等功能。

总结和讨论 Spring MVC

本文我们通过一个一个例子展现了 Spring MVC 的一些重要扩展点：

1. 使用拦截器做执行时间统计
2. 自定义 ResponseBodyAdvice 来处理 API 的包装
3. 自定义 ExceptionHandler 来统计错误处理
4. 自定义 ConverterFactory 来解析转换枚举
5. 自定义 ArgumentResolver 来组装设备信息参数
6. 快速实现静态资源、路径匹配以及 ViewController 的配置

其实 Spring MVC 还有很多扩展点，比如模型参数绑定和校验、允许我们实现动态的 RequestMapping 甚至是 DispatcherServlet 进行扩展，你可以继续自行研究。

最后，我想说说我对 Spring MVC 的看法，总体上我觉得 Spring MVC 实现很灵活，扩展点很多，几乎每一个组件都是松耦合，允许我们自己定义和替换。但是我觉得它的实现有点过于松散。ASP.NET MVC 的实现我就挺喜欢，相比 Spring MVC，ASP.NET MVC 的两个 ActionFilter 和 ActionResult 的实现是亮点：

1. ActionFilter 机制。Controller 里面的每一个方法称作 Action，我们可以在每一个 Action 上加上各种注解来启用 ActionFilter，ActionFilter 可以针对 Action 执行前、后、出异常等等情况做回调处理。ASP.NET MVC 的 ActionFilter 的 Filer 级别是方法，粒度上比拦截器精细很多，而且配置更直观。Spring MVC 虽然除了拦截器还有 ArgumentResolver 以及 ReturnValueHandler 可以分别进行参数处理和返回值处理，但是这两套扩展体系也是基于框架层面的，如果要和方法打通还需要自定义注解来实现。总觉得 Spring MVC 的这三套扩展点相互配合功能上虽然完整，但是有种支离破碎的感觉，如果我们真的要实现很多功能的，话可能会在这里有相当多的 if-else，没有 ActionFilter 来得直观。
2. 方法的返回值可以是 ModelAndView，可以是直接输出到@ResponseBody 的自定义类型，这两种输出类型的分法可以满足我们的需求，但是总感觉很别扭。在 ASP.NET MVC 中的方法返回抽象为了 ActionResult，可以是 ViewResult、JsonResult、FileContentResult、RedirectResult、FilePathResult、JavaScriptResult 等等，正如其名，看到返回值我们就可以看到方法实际的输出表现，非常直观容易理解。

ASP.NET MVC 并没有大量依赖 IOC 和 AOP 来实现，而是由框架的整体结构实现了插件机制，本质上这和 Spring 的风格就不同，加上 Spring MVC 从简化 Servlet 开始演化，两者理念上的区别也决定了设计上的区别，因此 Spring MVC 这样设计我也能理解。