Spring Boot与Web开发

SpringMVC快速使用

1.基于restful http接口 的CURD

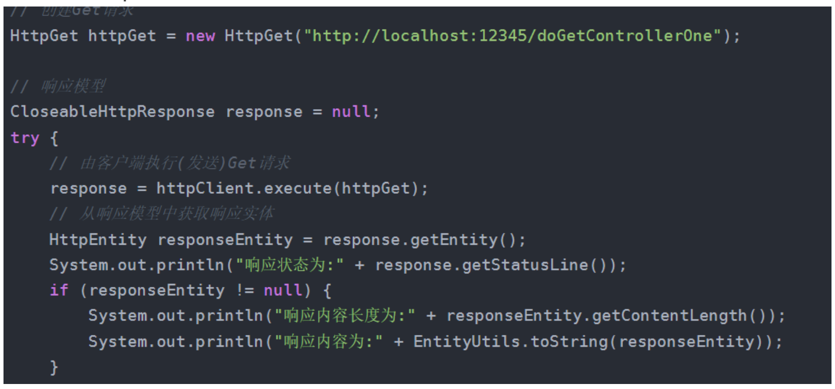
```java

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@RestController
@RequestMapping("/user")
public class UserController {

  @Autowired
  UserService userService;

  //Rest /user/1
  @GetMapping("/{id}")
  public Result getUser(@PathVariable Integer id) {
  User user=userService.getUserById(id);
  return new Result<>(200,"查询成功",user);
  }

// 新增 /user/add
@PostMapping("/add")
public Result addUser(User user) {
userService.add(user);
return new Result<>(200,"添加成功");
}

// 修改 /user1
@PutMapping("/{id}")
public Result editUser(User user) {
userService.update(user);
return new Result<>(200,"修改成功");
}

// 修改 /user1
@DeleteMapping("/{id}")
public Result editUser(@PathVariable Integer id) {
userService.delete(id);
return new Result<>(200,"删除成功");
}

}
```

2.调用rest http接口

- 通过**RestTemplate**调用

> RestTemplate是Spring提供的用于访问Rest服务的，RestTemplate提供了多种便捷访问远程Http服务的方法，传统情况下在java代码里访问restful服务，一般使用Apache的HttpClient。不过此种方法使用起来太过繁琐。spring提供了一种简单便捷的模板类来进行操作，这就是RestTemplate。
> 适用于微服务架构下 服务之间的远程调用 ps: 以后使用微服务架构， spring cloud feign
>
> **WebClient** 都可以调用远程服务， 区别：webclient 依赖webflux , webclient 请求远程服务是无阻塞的，响应的。 **RestTemplate 它是阻塞的，需要等待请求响应后才能执行下一句代码**

以前通过HttpClient



```java
// 创建Get请求
HttpGet httpGet = new HttpGet("http://localhost:12345/doGetControllerOne");

// 响应模型
CloseableHttpResponse response = null;
try {
    // 由客户端执行(发送)Get请求
    response = httpClient.execute(httpGet);
    // 从响应模型中获取响应实体
    HttpEntity responseEntity = response.getEntity();
    System.out.println("响应状态为:" + response.getStatusLine());
    if (responseEntity != null) {
        System.out.println("响应内容长度为:" + responseEntity.getContentLength());
        System.out.println("响应内容为:" + EntityUtils.toString(responseEntity));
    }
```

DELETE	delete
GET	getForObject 按照指定Class返回对象
	getForEntity 返回对象为ResponseEntity对象，包含了响应中的一些重要信息，比如响应头、响应状态码、响应体等
HEAD	headForHeaders

OPTIONS	optionsForAllow
POST	postForLocation
	postForObject
PUT	put
any 支持任何请求方法类型	exchange
	execute

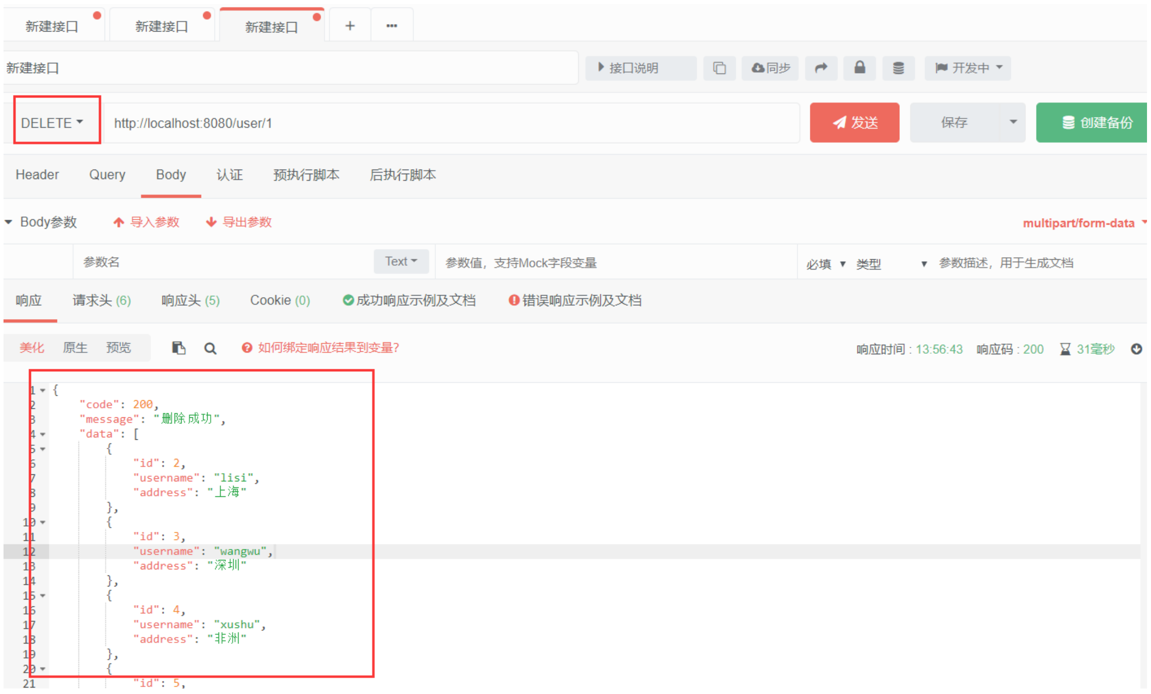
```java
@RestController
public class OrderController {

 // 声明了RestTemplate
 private final RestTemplate restTemplate;

 // 当bean 没有无参构造函数的时候，spring将自动拿到有参的构造函数，参数进行自动注入
 public OrderController(RestTemplateBuilder restTemplateBuilder) {
 this.restTemplate = restTemplateBuilder.build();
  }

  @RequestMapping("/order")
  public String order(){
  // 下单 需要远程访问rest服务

  // 基于restTemplate 调用查询
  /*Result forObject = restTemplate.getForObject("http://localhost:8080/user/{id}", Result.class, 1);
  return forObject.toString();*/


  // 基于restTemplate 调用新增
  /*

  User user = new User("徐庶", "随便");

  // url: 请求的远程rest url
  // object : post请求的参数
  // Class<T>:返回的类型
  // ...Object: 是@PathVariable 占位符的参数
  ResponseEntity<Result> resultResponseEntity = restTemplate.postForEntity("http://localhost:8080/user/add", user, Result.class);
  System.out.println(resultResponseEntity.toString());
  return resultResponseEntity.getBody().toString();*/


  // 基于restTemplate 调用修改
  /* User user = new User(1,"徐庶", "随便");
  //restTemplate.put("http://localhost:8080/user/add", user, Result.class);
  HttpEntity<User> httpEntity = new HttpEntity<>(user);

  ResponseEntity<Result> resultResponseEntity = restTemplate.exchange("http://localhost:8080/user/{id}", HttpMethod.PUT, httpEntity, Result.class, 1);
  System.out.println(resultResponseEntity.toString());
  return resultResponseEntity.getBody().toString();*/
```

```java



    // 基于restTemplate 调用删除
    ResponseEntity<Result> resultResponseEntity = restTemplate.exchange("http://localhost:8080/user/{id}", HttpMethod.DELETE, null, Result.class, 1);
    System.out.println(resultResponseEntity.toString());
    return resultResponseEntity.getBody().toString();
    }
  }
```

也可以在单元测试下使用：

```java
@SpringBootTest()
class ApplicationTests {

  @Test
  void contextLoads() {
  TestRestTemplate restTemplate=new TestRestTemplate();
  // 基于restTemplate 调用删除
  ResponseEntity<Result> resultResponseEntity = restTemplate.exchange("http://localhost:8080/user/{id}", HttpMethod.DELETE, null, Result.class, 1);
  System.out.println(resultResponseEntity.toString());

   }

}
```

3.通过postman调用



DELETE http://localhost:8080/user/1

响应时间 : 13:56:43 响应码 : 200 31毫秒

```
{
    "code": 200,
    "message": "删除成功",
    "data": [
        {
            "id": 2,
            "username": "lisi",
            "address": "上海"
        },
        {
            "id": 3,
            "username": "wangwu",
            "address": "深圳"
        },
        {
            "id": 4,
            "username": "xushu",
            "address": "非洲"
        },
        {
            "id": 5,
```

- ○ 通过MockMvc测试

> MockMvc是由spring-test包提供，实现了对Http请求的模拟，能够直接使用网络的形式，转换到Controller的调用，使得测试速度快、不依赖网络环境。同时提供了一套验证的工具，结果的验证十分方便。

SpringBoot中使用

编写测试类。实例化MockMvc有两种形式，一种是使用StandaloneMockMvcBuilder，另外一种是使用DefaultMockMvcBuilder。测试类及初始化MockMvc初始化：

```java

@SpringBootTest
@AutoConfigureMockMvc
class MockMvcExampleTests {

 @Test
 void exampleTest(@Autowired MockMvc mvc) throws Exception {
 mvc.perform(get("/")).andExpect(status().isOk()).andExpect(content().string("Hello World"));
 }

}
```

单元测试方法：

```java
@Test
public void testHello() throws Exception {

 /*
 * 1、mockMvc.perform执行一个请求。
 * 2、MockMvcRequestBuilders.get("XXX")构造一个请求。
 * 3、ResultActions.param添加请求传值
 * 4、ResultActions.accept(MediaType.TEXT_HTML_VALUE))设置返回类型
 * 5、ResultActions.andExpect添加执行完成后的断言。
  * 6、ResultActions.andDo添加一个结果处理器，表示要对结果做点什么事情
  * 比如此处使用MockMvcResultHandlers.print()输出整个响应结果信息。
  * 7、ResultActions.andReturn表示执行完成后返回相应的结果。
  */
  mockMvc.perform(MockMvcRequestBuilders
  .get("/hello")
  // 设置返回值类型为utf-8，否则默认为ISO-8859-1
  .accept(MediaType.APPLICATION_JSON_UTF8_VALUE)
  .param("name", "Tom"))
  .andExpect(MockMvcResultMatchers.status().isOk())
  .andExpect(MockMvcResultMatchers.content().string("Hello Tom!"))
  .andDo(MockMvcResultHandlers.print());
}
```

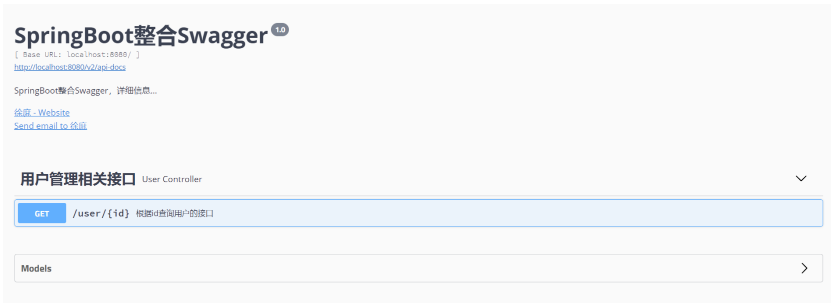
测试结果打印：

```
FlashMap:
 Attributes = null

MockHttpServletResponse:
 Status = 200
 Error message = null
 Headers = [Content-Type:"application/json;charset=UTF-8", Content-Length:"10"]
 Content type = application/json;charset=UTF-8
 Body = Hello Tom!
  Forwarded URL = null
  Redirected URL = null
  Cookies = []
 2019-04-02 21:34:27.954 INFO 6937 --- [ Thread-2] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting
down ExecutorService 'applicationTaskExecutor'
```

4.通过swagger调用

相信无论是前端还是后端开发，都或多或少地被接口文档折磨过。前端经常抱怨后端给的接口文档与实际情况不一致。后端又觉得编写及维护接口文档会耗费不少精力，经常来不及更新。其实无论是前端调用后端，还是后端调用后端，都期望有一个好的接口文档。但是这个接口文档对于程序员来说，就跟注释一样，经常会抱怨别人写的代码没有写注释，然而自己写起代码起来，最讨厌的，也是写注释。所以仅仅只通过强制来规范大家是不够的，随着时间推移，版本迭代，接口文档往往很容易就跟不上代码了。

- SpringBoot 整合swagger2.x


SpringBoot整合Swagger 1.0
[Base URL: localhost:8080/]
http://localhost:8080/v2/api-docs

SpringBoot整合Swagger，详细信息...

徐庶 - Website
Send email to 徐庶

用户管理相关接口 User Controller

GET /user/{id} 根据id查询用户的接口

Models

1. 添加依赖

```
<dependency>
 <groupId>io.springfox</groupId>
 <artifactId>springfox-swagger2</artifactId>
 <version>2.9.2</version>
</dependency>
<dependency>
 <groupId>io.springfox</groupId>
 <artifactId>springfox-swagger-ui</artifactId>
 <version>2.9.2</version>
 </dependency>
```

2. 添加swagger配置类

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Configuration
@EnableSwagger2
public class SwaggerConfig {
 @Bean
  public Docket createRestApi() {
  return new Docket(DocumentationType.SWAGGER_2)
  .pathMapping("/")
  .select()
  .apis(RequestHandlerSelectors.basePackage("com.mine.controller"))
  .paths(PathSelectors.any())
  .build().apiInfo(new ApiInfoBuilder()
  .title("SpringBoot整合Swagger")
  .description("SpringBoot整合Swagger，详细信息......")
  .version("1.0")
  .contact(new Contact("徐庶","www.tulingxueyuan.cn","123@qq.com"))
  .build());
  }
 }
```

访问：http://localhost:8080/swagger-ui.html

3. 配置htpp接口

```java
/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@RestController
@Api(tags = "用户管理相关接口")
@RequestMapping("/user")
public class UserController {

 @GetMapping("/{id}")
 @ApiOperation("根据id查询用户的接口")
 @ApiImplicitParam(name = "id", value = "用户id", defaultValue = "99", required = true)
 public User getUserById(@PathVariable Integer id) {
 User user = new User();
 user.setId(id);
 return user;
 }
}
```

4. 配置pojo类

```java

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@ApiModel
public class User {
 @ApiModelProperty(value = "用户id")
 private Integer id;
 @ApiModelProperty(value = "用户名")
 private String username;
 @ApiModelProperty(value = "用户地址")
 private String address;

 public Integer getId() {
 return id;
 }

 public void setId(Integer id) {
 this.id = id;
 }

 public String getUsername() {
 return username;
 }

 public void setUsername(String username) {
 this.username = username;
 }

 public String getAddress() {
 return address;
 }

 public void setAddress(String address) {
 this.address = address;
 }
}
```
访问：http://localhost:8080/swagger-ui.html

扩展： 实现通过swagger记录请求日志

```


@Aspect
@Component
public class Testaop {

 @Pointcut("execution( public * com.tulingxueyuan.controller.*(..))" +
 "&&@annotation(io.swagger.annotations.ApiOperation)" )
 public void log() {}

  @Around("log()&&@annotation(apiOperation)")
  public Object around(ProceedingJoinPoint pjp, ApiOperation apiOperation) {
  Object result = null;
  try {
  //获取类对象
  Class<?> controller = pjp.getThis().getClass();
  //获取swagger接口对象
  Api annotation = controller.getAnnotation(Api.class);
  //获取类接口的内容
  String description = annotation.description();
  System.out.println("调用接口为: "+description +"-"+ apiOperation.value());
  //执行方法
  result = pjp.proceed();
  } catch (Throwable throwable) {
  throwable.printStackTrace();
  }

  return result;
  }
```

2.SpringMVC自动配置原理分析

Spring Boot为Spring MVC提供了自动配置，可与大多数应用程序完美配合。
自动配置在Spring的默认值之上添加了以下功能：

- 包含`ContentNegotiatingViewResolver`和`BeanNameViewResolver`。
  - `ViewResolver` 都是SpringMVC内置的视图解析器
    - `ContentNegotiatingViewResolver`
      - 他并不会解析视图、而是委派给其他视图解析器进行解析
        - 所有视图解析器，都会根据返回回的视图名称进行解析视图 resolveViewName

```
public View resolveViewName(String viewName, Locale locale) throws Exception {
 RequestAttributes attrs = RequestContextHolder.getRequestAttributes();
```

```java
 Assert.state(attrs instanceof ServletRequestAttributes, "No current ServletRequestAttributes");
 List<MediaType> requestedMediaTypes = getMediaTypes(((ServletRequestAttributes) attrs).getRequest())
 if (requestedMediaTypes != null) {
 // 获得所有匹配的视图
 List<View> candidateViews = getCandidateViews(viewName, locale, requestedMediaTypes);
 // 获取最终的这个
 View bestView = getBestView(candidateViews, requestedMediaTypes, attrs);
  if (bestView != null) {
  return bestView;
  }
  }
```

委派给其他视图解析器进行解析：

```java
@Override
protected void initServletContext(ServletContext servletContext) {
 Collection<ViewResolver> matchingBeans =
 BeanFactoryUtils.beansOfTypeIncludingAncestors(obtainApplicationContext(), ViewResolver.class).values();
 if (this.viewResolvers == null) {
 this.viewResolvers = new ArrayList<>(matchingBeans.size());
 for (ViewResolver viewResolver : matchingBeans) {
 if (this != viewResolver) {
 this.viewResolvers.add(viewResolver);
  }
  }
  }
```

- `BeanNameViewResolver`
    - 会根据handler方法返回的视图名称 (xushu), 去ioc容器中到到名字叫xushu的一个Bean，并且这个bean要实现了VIew接口
    - 示例：

```java
 //Rest /user/1
 @GetMapping("/test")
public String getUser() {
 return "xushu";
 }
```

可以配置一个名字叫xushu的视图(VIew)

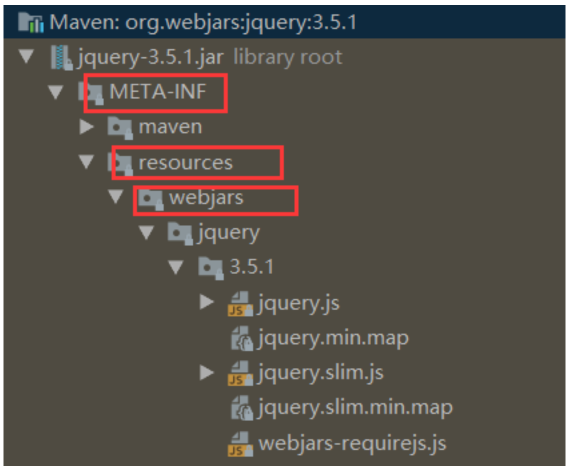
```java

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Component
public class Xushu implements View {
 @Override
 public String getContentType() {
  return "text/html";
  }

  @Override
  public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {
  response.getWriter().print("Welcome to XushuView");
  }
}
```
由以上代码可以得出结论，它是从Spring IOC容器获得ViewResolver类型Bean，那么我们可以自己定制一个ViewResolver,ContentNegotiatingViewResolver也会帮我们委派解析

```
@Bean
public ViewResolver xushuViewResolver(){
 InternalResourceViewResolver resolver = new InternalResourceViewResolver();
 resolver.setPrefix("/");
 resolver.setSuffix(".html");
 return resolver;
}
```

- 支持提供静态资源。包括对WebJars的支持（在本文档的后面部分中有介绍）。
  - 以前要访问jpg\css、js 等 这些静态资源文件， 需要在web.xml配置 ,在springboot不需要配置，只需要放在约定文件夹中就可以（约定大于配置）
  - 原理：
  - WebJars: 就是将静态资源放在jar包中进行访问
    - webjars官网：https://www.webjars.org/

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
 if (!this.resourceProperties.isAddMappings()) {
 logger.debug("Default resource handling disabled");
 return;
 }
 Duration cachePeriod = this.resourceProperties.getCache().getPeriod();
 CacheControl cacheControl =
this.resourceProperties.getCache().getCachecontrol().toHttpCacheControl();
 if (!registry.hasMappingForPattern("/webjars/**")) {
  customizeResourceHandlerRegistration(registry.addResourceHandler("/webjars/**")
  .addResourceLocations("classpath:/META-INF/resources/webjars/")
  .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
  }
```

- 当访问/webjars/** 时 就会去classpath:/META-INF/resources/webjars/ 对应进行映射
  - 当访问http://localhost:8080/webjars/juqery/3.5.1/jquery.js 对应映射到 /META-INF/resources/webjars/juqery/3.5.1/jquery.js



- 在static文件中访问的静态资源： 又是什么原理呢？：

```
String staticPathPattern = this.mvcProperties.getStaticPathPattern();
 if (!registry.hasMappingForPattern(staticPathPattern)) {
 customizeResourceHandlerRegistration(registry.addResourceHandler(staticPathPattern)
 .addResourceLocations(getResourceLocations(this.resourceProperties.getStaticLocations()))
 .setCachePeriod(getSeconds(cachePeriod)).setCacheControl(cacheControl));
 }
}
```

getStaticLocations

```
public String[] getStaticLocations() {
 return this.staticLocations;
}
```

对应的映射地址：

```
{ "classpath:/META-INF/resources/",
 "classpath:/resources/",
 "classpath:/static/",
 "classpath:/public/" }
```

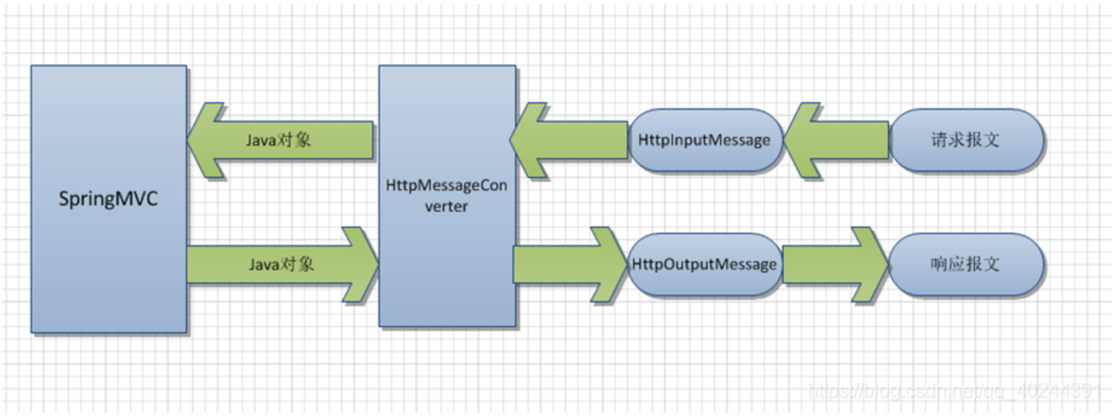
配置欢迎页：

```
private Optional<Resource> getWelcomePage() {
 // 拿到上面静态资源地址
 String[] locations = getResourceLocations(this.resourceProperties.getStaticLocations());
 // 去里面找一个index.html的首页
 return Arrays.stream(locations).map(this::getIndexHtml).filter(this::isReadable).findFirst();
}
```

- 也可以通过配置文件指定具体的静态资源地址：

```
spring.resources.static-locations=classpath:/static/
```

- 自动注册`Converter`，`GenericConverter`和`Formatter` Bean类。
  - 使用方式大家可以参考SpringMvc课程中讲得
- 支持`HttpMessageConverters`（在本文档后面介绍）。
  - `HttpMessageConverters` 负责`http`请求和响应的报文处理



- 自动注册`MessageCodesResolver`（在本文档后面介绍）。
  - 修改4xx 错误下 格式换转换出错 类型转换出错的 错误代码：
  - 以前的格式
  - errorCode + "." + object name + "." + field
    - typeMismatch.user.birthday
  - 可以通过

```
spring.mvc.message-codes-resolver-format=postfix_error_code
```

    - 将格式修改为：object name + "." + field + "." + errorCode

- 静态`index.html`支持。
    - 在springboot中可以直接返回html的视图
    - 因为在自动`WebMvcAutoConfiguration`配置类配置

```java
@Bean
@ConditionalOnMissingBean
public InternalResourceViewResolver defaultViewResolver() {
 InternalResourceViewResolver resolver = new InternalResourceViewResolver();
 resolver.setPrefix(this.mvcProperties.getView().getPrefix());
 resolver.setSuffix(this.mvcProperties.getView().getSuffix());
 return resolver;
}
```

    - 所以就可以通过在配置文件中完成

```
spring.mvc.view.prefix=/pages/
spring.mvc.view.suffix=.html
```

- 自动使用`ConfigurableWebBindingInitializer` bean（在本文档后面部分中介绍）。

3.定制SpringMvc的自动配置

SpringMVC的自动配置类：`WebMvcAutoConfiguration`

1. 在大多数情况，SpringBoot在自动配置中标记了很多`@ConditionalOnMissingBean(xxxxxxxxx.class);`(意思就是如果容器中没有，当前的@bean才会生效）。 只需要在自己的配置类中配置对应的一个@Bean就可以覆盖默认自动配置。还得结合源码的实际功能进行定制。 （经验之谈）

2.通过`WebMvcConfigurer`进行扩展

a. 扩展视图控制器
b. 扩展拦截器
c. 扩展全局CORS

```java

@Configuration
public class MyWebMvcConfigurer implements WebMvcConfigurer {
 /**
 * 添加视图控制器
 * 立即访问
 * <mvc:view-controller path="/" view-name="index" />
 * @param registry
 */
  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
  registry.addViewController("/tuling").setViewName("hello");
  }

  /**
  * 添加拦截器
  * @param registry
  */
@Override
public void addInterceptors(InterceptorRegistry registry) {
registry.addInterceptor(new TimeInterceptor()) //添加拦截器
.addPathPatterns("/**") // 拦截映射规则
.excludePathPatterns("/pages/**"); // 设置排除的映射规则
}

/**
* 全局CORS配置
* @param registry

@Override
public void addCorsMappings(CorsRegistry registry) {
registry.addMapping("/user/*") // 映射服务器中那些http接口运行跨域访问
.allowedOrigins("http://localhost:8081") // 配置哪些来源有权限跨域
.allowedMethods("GET","POST","DELETE","PUT"); // 配置运行跨域访问的请求方法
}*/

}
```

WebMvcConfigurer 原理

实现WebMvcConfigurer接口可以扩展Mvc实现， 又既保留SpringBoot的自动配置

1.在WebMvcAutoConfiguration 也有一个实现了WebMvcConfigurer的配置类

2.WebMvcAutoConfigurationAdapter 它也是利用这种方式去进行扩展， 所以我们通过查看这个类我们发现它帮我们实现了其他不常用的方法，帮助我们进行自动配置，我们只需定制（拦截器、视图控制器、CORS 在开发中需要额外定制的定制的功能）

```
@Configuration(proxyBeanMethods = false)
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class, ResourceProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter implements WebMvcConfigurer {
```

2. 导入了EnableWebMvcConfiguration

```
@Import(EnableWebMvcConfiguration.class)
```

3.EnableWebMvcConfiguration它的父类上 setConfigurers 使用@Autowired

- 它会去容器中将所有实现了WebMvcConfigurer 接口的Bean都自动注入进来， 添加到configurers 变量中

```
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {

 private final WebMvcConfigurerComposite configurers = new WebMvcConfigurerComposite();


 @Autowired(required = false)
 public void setConfigurers(List<WebMvcConfigurer> configurers) {
 if (!CollectionUtils.isEmpty(configurers)) {
 this.configurers.addWebMvcConfigurers(configurers);
  }
  }
```

- 添加到delegates委派器中

```java
public void addWebMvcConfigurers(List<WebMvcConfigurer> configurers) {
 if (!CollectionUtils.isEmpty(configurers)) {
 this.delegates.addAll(configurers);
 }
}
```

- 底层调用WebMvcConfigurer对应的方法时 就是去拿到之前注入到delegates的WebMvcConfigurer ,依次调用

```java
@Override
public void addInterceptors(InterceptorRegistry registry) {
 for (WebMvcConfigurer delegate : this.delegates) {
 delegate.addInterceptors(registry);
 }
}
```

- 当添加了@EnableWebMvc就不会使用SpringMVC自动配置类的默认配置就失效了
    - 为什么呢？原理：
        - 在@EnableWebMvc 中

@Import(DelegatingWebMvcConfiguration.class)

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(DelegatingWebMvcConfiguration.class)
public @interface EnableWebMvc {
}
```

- 在DelegatingWebMvcConfiguration中继承了

```java

@Configuration(proxyBeanMethods = false)
public class DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport {
```

- 在WebMvcAutoConfiguration 中

@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)

- 当容器中不存在WebMvcConfigurationSupport 这个Bean的时候当前自动配置类才会生效
- 正因为通过@EnableWebMvc 导入了DelegatingWebMvcConfiguration extends WebMvcConfigurationSupport 从而才使自动配置类失效

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class, TaskExecutionAutoConfiguration.class,
 ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {
```

2. Json 开发

Spring Boot提供了与三个JSON映射库的集成：

- Gson
- Jackson 性能最好
- JSON-B

Jackson 是我们使用的默认json库

jsckson的使用

- @JsonIgnore
  - 进行排除json序列化，将它标注在属性上将不会进行json序列化
- @JsonFormat(pattern = "yyyy-MM-dd hh:mm:ss",locale = "zh")
  - 进行日期格式化
- @JsonInclude(JsonInclude.Include.NON_NULL)
  - 当属性值为null时则不进行json序列化
- @JsonProperty("uname")
  - 来设置别名

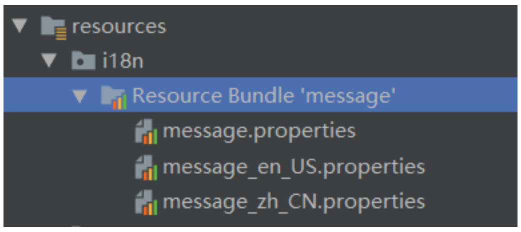
SpringBoot 还提供了`@JsonComponent` 来根据自己的业务需求进行json的序列化和反序列化

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@JsonComponent
public class UserJsonCustom {
 public static class Serializer extends JsonObjectSerializer<User> {

  @Override
  protected void serializeObject(User user, JsonGenerator jgen, SerializerProvider provider) throws IO
Exception {
  jgen.writeObjectField("id",user.getId()); //{"id","xxx"}
  jgen.writeObjectField("uname","xxxxx");
  /*jgen.writeFieldName(""); 单独写key
  jgen.writeObject(); 单独写value
  */
  // 1. 一次查不出完整的数据返回给客户端， 需要根据需求去做一些个性化调整
  // 2. 根据不同的权限给他返回不同的序列化数据
  }
  }

  public static class Deserializer extends JsonObjectDeserializer<User> {

  @Override
  protected User deserializeObject(JsonParser jsonParser, DeserializationContext context, ObjectCodec
codec, JsonNode tree) throws IOException {
  User user=new User();
  user.setId(tree.findValue("id").asInt());

  return user;
  }
  }
 }
```

3.国际化

SpringBoot

1. 添加国际化资源文件 resource



2. 配置messageResource　设置国际化资源文件

    a. 在Springboot中提供了MessageSourceAutoConfiguration 所以，我们不需要去配置messageResource

    b. 但是它并生效:开启debug=true

```
MessageSourceAutoConfiguration:
 Did not match:
 - ResourceBundle did not find bundle with basename messages (MessageSourceAutoConfiguration.ResourceB
undleCondition)
```

自动配置类没有生效的原因:

```
@Configuration(proxyBeanMethods = false)
// 如果自己配置了@Bean 名字叫messageSource的bean 就会用自定义的
@ConditionalOnMissingBean(name = AbstractApplicationContext.MESSAGE_SOURCE_BEAN_NAME, search = SearchS
trategy.CURRENT)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
// @Conditional 自定义条件匹配 会传入一个实现了Condition接口的类——ResourceBundleCondition
// ResourceBundleCondition 会重写matches， 自定义匹配规则 ， 如果该方法返回true 就匹配成功
@Conditional(ResourceBundleCondition.class)
@EnableConfigurationProperties
public class MessageSourceAutoConfiguration {
```

matches:

```
@Override
public final boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
 String classOrMethodName = getClassOrMethodName(metadata);
 try {
 // getMatchOutcome 具体的匹配规则就在这个里面
 ConditionOutcome outcome = getMatchOutcome(context, metadata);
 logOutcome(classOrMethodName, outcome);
 recordEvaluation(context, classOrMethodName, outcome);
 return outcome.isMatch();
  }
```

- getMatchOutcome
    - 只要在这个方法中将返回的ConditionOutcome .match =true 那匹配成功

```
@Override
public ConditionOutcome getMatchOutcome(ConditionContext context, AnnotatedTypeMetadata metadata) {
 // 获取配置文件中spring.messages.basename ， 由于我们没有配置 默认值是: messages
 String basename = context.getEnvironment().getProperty("spring.messages.basename", "messages");
 ConditionOutcome outcome = cache.get(basename);
 if (outcome == null) {
 outcome = getMatchOutcomeForBasename(context, basename);
 cache.put(basename, outcome);
 }
  return outcome;
}
```

- getMatchOutcomeForBasename

```
private ConditionOutcome getMatchOutcomeForBasename(ConditionContext context, String basename) {
```

```java
ConditionMessage.Builder message = ConditionMessage.forCondition("ResourceBundle");
for (String name :
StringUtils.commaDelimitedListToStringArray(StringUtils.trimAllWhitespace(basename))) {
// 根据messages获取 该类路径下的所有propeties的资源文件
for (Resource resource : getResources(context.getClassLoader(), name)) {
// 这个条件非常关键:  只要basename的类路径下又资源文件就会匹配成功
if (resource.exists()) {
return ConditionOutcome.match(message.found("bundle").items(resource));
}
 }
 }
 return ConditionOutcome.noMatch(message.didNotFind("bundle with basename " + basename).atAll());
}
```

- 如果要让它生效
    - 必须保证 在类路径下的messages文件夹中有国际化的资源文件
    - 或者自己配置spring.messages.basename 告诉它资源文件在哪

```
spring.messages.basename=i18n.message
```

- 只要找到了国际化的属性资源文件那就会设置ConditionOutcome.match=true
- 当ConditionOutcome.match=true 那么@Conditional(ResourceBundleCondition.class) 就匹配成功
- 一旦匹配成功那自动配置类就会生效，就会帮我们配置一个meesageSource

```java
@Bean
public MessageSource messageSource(MessageSourceProperties properties) {
```

3. 需要去解析请求头中的accept-language   或者 解析url参数中?local=

- 其实WebMvcAutoConfiguration 类也帮我配置了一个解析请求头中的accept-language 的localResolver

```java
@Bean
@ConditionalOnMissingBean
@ConditionalOnProperty(prefix = "spring.mvc", name = "locale")
public LocaleResolver localeResolver() {
 // 当配置spring.mvc.locale-resolver=fiexd
 if (this.mvcProperties.getLocaleResolver() == WebMvcProperties.LocaleResolver.FIXED) {
 // 就会使用配置文件中的本地化语言：spring.mvc.locale=en_US 就可以设死本地化语言
 return new FixedLocaleResolver(this.mvcProperties.getLocale());
 }
  // 默认就是使用AcceptHeaderLocaleResolver 作为本地化解析器
  AcceptHeaderLocaleResolver localeResolver = new AcceptHeaderLocaleResolver();
  //spring.mvc.locale=en_US 作为默认的本地化语言
  localeResolver.setDefaultLocale(this.mvcProperties.getLocale());
  return localeResolver;
}
```

```java
@Override
public Locale resolveLocale(HttpServletRequest request) {
 // 当Accept-Languag为null 才会使用使用配置文件中设置的locale:spring.mvc.locale
 Locale defaultLocale = getDefaultLocale();
 if (defaultLocale != null && request.getHeader("Accept-Language") == null) {
 return defaultLocale;
 }
 // 就是使用request.getLocale();
Locale requestLocale = request.getLocale();
 List<Locale> supportedLocales = getSupportedLocales();
 if (supportedLocales.isEmpty() || supportedLocales.contains(requestLocale)) {
 return requestLocale;
 }
 Locale supportedLocale = findSupportedLocale(request, supportedLocales);
 if (supportedLocale != null) {
 return supportedLocale;
 }
 return (defaultLocale != null ? defaultLocale : requestLocale);
}
```

4. 随意切换本地语言，进行缓存
    a. 覆盖原有localeResolver 因为自动配置类中的localeResovler它只会从accept-language中解析

5. 通过messageResource 获取国际化信息

```
1.第一种方法 在Handler方法参数中加入Locale参数，注入ResourceBundleMessageSource 对象
 messageSource.getMessage(code, args,locale);

2.第二种方法 或者使用定义工具类：
package com.ns.utils;

import javax.servlet.http.HttpServletRequest;

import org.springframework.beans.factory.annotation.Autowired;
 import org.springframework.context.support.ResourceBundleMessageSource;

 /**
  * 国际化帮助类
  * @author Administrator
  *
  */
 public class I18nMessageUtil {

    private static ResourceBundleMessageSource messageSource;

    private static HttpServletRequest request;

    /**
     * 获取国际化资源属性
     * @param code
     * @param args
     * @return
     */
    public static String getMessage(String code,String... args){
       return messageSource.getMessage(code, args,request.getLocale());
    }
    @Autowired
    public void setMessageSource(ResourceBundleMessageSource messageSource) {
       this.messageSource = messageSource;
    }
    @Autowired
    public void setRequest(HttpServletRequest request) {
        this.request = request;
    }

}



```

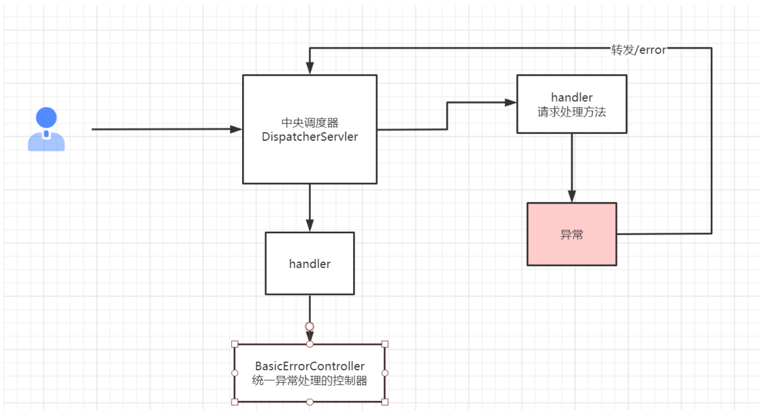
4.统一异常处理

1.SpringBoot 有统一异常处理自动配置类

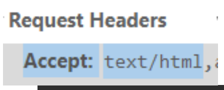
- **ErrorMvcAutoConfiguration**
    - 它统一异常处理自动配置类
- 重要组件
    - `DefaultErrorAttributes`
    - `BasicErrorController`
    - `DefaultErrorViewResolver` 用来解析错误视图页面

    - **BasicErrorController**
        - 它其实就是一个处理/error请求的一个控制器

```
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorController {
```


转发/error
中央调度器 DispatcherServler
handler 请求处理方法
异常
handler
BasicErrorController 统一异常处理的控制器

- **怎么处理的:**
    - 当使用浏览器发送请求时 请求头是
    - 所以如果是浏览器请求会交给**errorHtml**方法处理
    - 那除了text/html的其他请求都会交给error方法处理


Request Headers

Accept: text/html,

```
@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {
 HttpStatus status = getStatus(request);
 Map<String, Object> model = Collections
 .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions(request,
MediaType.TEXT_HTML)));
 response.setStatus(status.value());
 ModelAndView modelAndView = resolveErrorView(request, response, status, model);
 return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
}

@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
 HttpStatus status = getStatus(request);
 if (status == HttpStatus.NO_CONTENT) {
 return new ResponseEntity<>(status);
 }
 Map<String, Object> body = getErrorAttributes(request, getErrorAttributeOptions(request, MediaType.A
LL));
 return new ResponseEntity<>(body, status);
}
```

- **errorHtml 怎么去定制它的返回页面**
  - getErrorAttributes
    - 用来获取所需要的异常信息
- resolveErrorView **解析视图**
  - **会调用**DefaultErrorViewResolver的resolveErrorView方法

```
@Override
public ModelAndView resolveErrorView(HttpServletRequest request, HttpStatus status, Map<String,
Object> model) {
 ModelAndView modelAndView = resolve(String.valueOf(status.value()), model);
 if (modelAndView == null && SERIES_VIEWS.containsKey(status.series())) {
 modelAndView = resolve(SERIES_VIEWS.get(status.series()), model);
 }
 return modelAndView;
}
```

- 先从模板视图去解析（由于没有配置模板视图所以并不解析出来）

<li>{@code '/<templates>/error/404.<ext>'}</li>

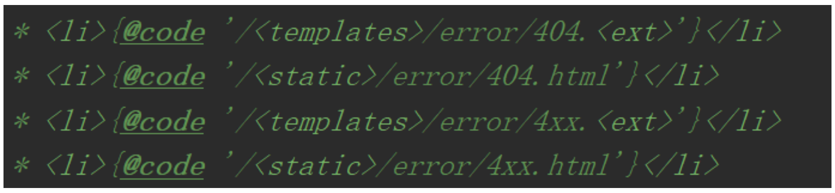
```
private ModelAndView resolve(String viewName, Map<String, Object> model) {
 String errorViewName = "error/" + viewName;
 TemplateAvailabilityProvider provider = this.templateAvailabilityProviders.getProvider(errorViewName
 this.applicationContext);
 if (provider != null) {
 return new ModelAndView(errorViewName, model);
 }
 return resolveResource(errorViewName, model);
}
```

- resolveResource
  - 就是去classpath:/static/error/xxxhtml

<li>{@code '/<static>/error/404.html'}</li>

```
private ModelAndView resolveResource(String viewName, Map<String, Object> model) {
 for (String location : this.resourceProperties.getStaticLocations()) {
 try {
```

```
Resource resource = this.applicationContext.getResource(location);
resource = resource.createRelative(viewName + ".html");
if (resource.exists()) {
return new ModelAndView(new HtmlResourceView(resource), model);
}
}
 catch (Exception ex) {
 }
 }
 return null;
}
```



```
* <li>{@code '/<templates>/error/404.<ext>'}</li>
* <li>{@code '/<static>/error/404.html'}</li>
* <li>{@code '/<templates>/error/4xx.<ext>'}</li>
* <li>{@code '/<static>/error/4xx.html'}</li>
```

总结： 从errorHtml方法可以得出结论： 我们需要使用自定义的页面响应错误只需要在对应的路径上创建对应错误代码的页面就行了，， **但是如果想记录日志就需要自己定制了。**

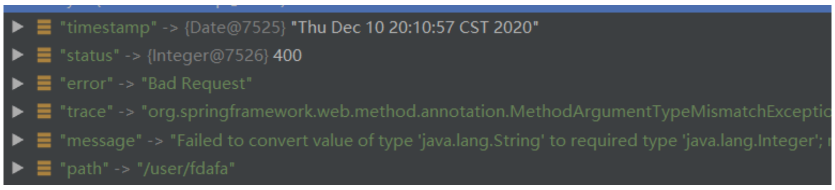
- error
    - 看它是怎么返回json数据的， 从而要定制自己的json数据

```
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
 HttpStatus status = getStatus(request);
 if (status == HttpStatus.NO_CONTENT) {
 return new ResponseEntity<>(status);
 }
 // 就是调用getErrorAttributes 获取了异常信息
 Map<String, Object> body = getErrorAttributes(request, getErrorAttributeOptions(request, MediaType.A
L));
 return new ResponseEntity<>(body, status);
}
```

- getErrorAttributeOptions 可以动态控制返回的异常熟悉
    - 根据配置文件 中server.error.xxx

```
protected ErrorAttributeOptions getErrorAttributeOptions(HttpServletRequest request, MediaType mediaTy
pe) {
 ErrorAttributeOptions options = ErrorAttributeOptions.defaults();
 if (this.errorProperties.isIncludeException()) {
 options = options.including(Include.EXCEPTION);
 }
 if (isIncludeStackTrace(request, mediaType)) {
 options = options.including(Include.STACK_TRACE);
 }
 if (isIncludeMessage(request, mediaType)) {
  options = options.including(Include.MESSAGE);
  }
  if (isIncludeBindingErrors(request, mediaType)) {
  options = options.including(Include.BINDING_ERRORS);
  }
  return options;
}
```

返回的异常信息：



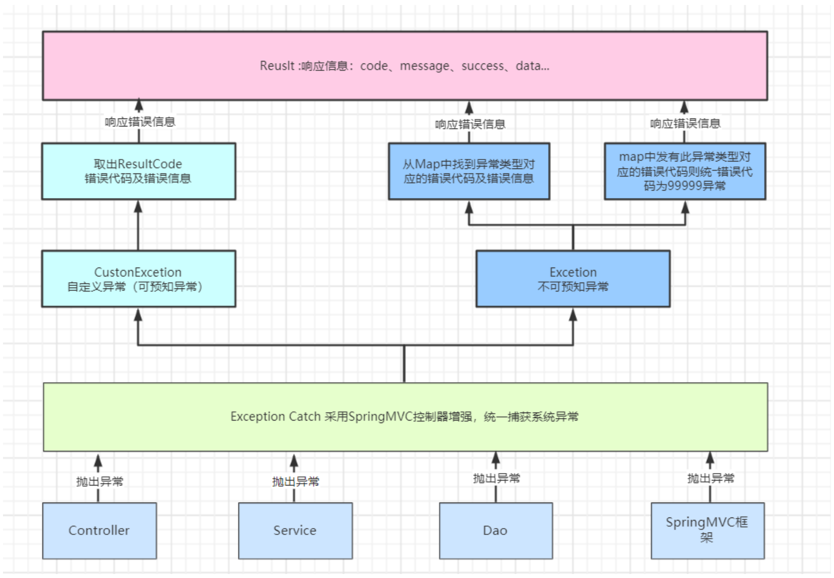
```
▶ "timestamp" -> {Date@7525} "Thu Dec 10 20:10:57 CST 2020"
▶ "status" -> {Integer@7526} 400
▶ "error" -> "Bad Request"
▶ "trace" -> "org.springframework.web.method.annotation.MethodArgumentTypeMismatchExceptio
▶ "message" -> "Failed to convert value of type 'java.lang.String' to required type 'java.lang.Integer'; r
▶ "path" -> "/user/fdafa"
```

通过研究上面自动配置类定制自己的统一异常处理

```java

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Controller
@RequestMapping("/error")
public class CustomErrorController extends AbstractErrorController {
 public CustomErrorController(ErrorAttributes errorAttributes, List<ErrorViewResolver> errorViewResol
ers) {
  super(errorAttributes, errorViewResolvers);
  }

  Logger logger= LoggerFactory.getLogger(CustomErrorController.class);

  /**
  * 处理浏览器请求的
  * 加上异常日志记录
  * @param request
  * @param response
  * @return
  */
  @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
  public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse response) {
  HttpStatus status = getStatus(request);
  Map<String, Object> model = Collections
  .unmodifiableMap(getErrorAttributes(request, getErrorAttributeOptions()));
  response.setStatus(status.value());
  ModelAndView modelAndView = resolveErrorView(request, response, status, model);
  logger.error(model.get("trace").toString());
  return (modelAndView != null) ? modelAndView : new ModelAndView("error", model);
  }

  /**
  * 处理ajax
  * 修改返回类型：Result 完成
  * 加上异常日志记录
  * @param request
  * @return
  */
  @RequestMapping
  @ResponseBody
  public Result error(HttpServletRequest request) {
  HttpStatus status = getStatus(request);
  if (status == HttpStatus.NO_CONTENT) {
  return new Result(204,"No Content");
  }

  Map<String, Object> body = getErrorAttributes(request, getErrorAttributeOptions());
  String code = body.get("status").toString();
  String message = body.get("message").toString();
  logger.error(body.get("trace").toString());
  return new Result(Integer.parseInt(code),message);
  }


  /**
  * 异常信息的选项
  * @return
  */
  protected ErrorAttributeOptions getErrorAttributeOptions() {
  ErrorAttributeOptions of = ErrorAttributeOptions.of(ErrorAttributeOptions.Include.MESSAGE,
  ErrorAttributeOptions.Include.STACK_TRACE,
  ErrorAttributeOptions.Include.EXCEPTION);
  return of;
  }

  @Override
  public String getErrorPath() {
  return null;
  }
 }
```

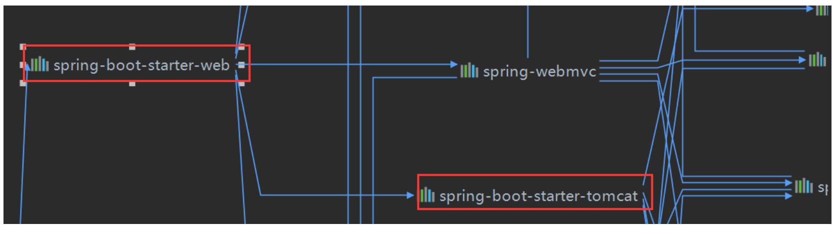
2.@ControllerAdivce

```java

@ControllerAdvice
public class GeneralExceptionHandler {

 @ExceptionHandler(Exception.class)
 public ModelAndView handleException(HttpServletRequest request,
 HttpServletResponse reponse, Exception ex,
 HandlerMethod handle){
 System.out.println("全局异常处理");
  // 如果当前请求是ajax就返回json

  // 1.根据用户请求的处理方法，是否是一个返回json的处理方法
  //RestController restAnnotation = handle.getClass().getAnnotation(RestController.class); // 获得类上面的某个注解
  //ResponseBody responseBody = handle.getMethod().getAnnotation(ResponseBody.class);//获得方法上面的某个注解
  // if(restAnnotation!=null || responseBody!=null){ }

  // 2.可以根据请求头中的类型Content-Type包含application/json


  if(request.getHeader("Accept").indexOf("application/json")>-1){
  // 可以直接输出json reponse.getWriter().write(); 或者集成jackson

 // 集成jackson的方式:
 //ModelAndView 同时支持视图返回和json返回
 // 这种方式就是返回json
 ModelAndView modelAndView = new ModelAndView(new MappingJackson2JsonView());
 // 通常会根据不同的异常返回不同的编码
 modelAndView.addObject("code", HttpStatus.INTERNAL_SERVER_ERROR.value());
 modelAndView.addObject("message",ex.getMessage());
 return modelAndView;
 }
 else{
 ModelAndView modelAndView = new ModelAndView();
 modelAndView.setViewName("error");
 modelAndView.addObject("ex", ex);
 StringWriter sw = new StringWriter();
 PrintWriter pw = new PrintWriter(sw);
 ex.printStackTrace(pw);
 System.out.println(sw.toString()); // 日志记录
 return modelAndView;
 }
 }
}
```



4.SpringBoot的嵌入式Servlet容器

Spring 默认的Servlet容器是：Tomcat， 当前SpringBoot 2.3.6 的版本是对应 tomcat9



1.嵌入式Servlet容器配置修改

- **1.通过全局配置文件修改**
  - **可以通过server.xxx 来进行web服务配置， 没有带服务器名称的则是通用配置**
  - **通过带了具体的服务器名称则是单独对该服务器进行设置，比如server.tomcat.xxx 就是专门针对tomcat的配置**

```
server.port=8080
server.servlet.context-path=/tomcat
```

- **2.通过**WebServerFactoryCustomizer的Bean修改
  - 修改server.xxx 配置的相关内容
  - 会跟配置文件形成互补

```

@Component
public class CustomizationBean implements WebServerFactoryCustomizer<ConfigurableServletWebServerFacto
ry> {

 @Override
 public void customize(ConfigurableServletWebServerFactory server) {
 server.setPort(8088);
 server.setContextPath("/customTomcat");
 }

}
```

2.注册servlet三大组件

- servlet listener filter

- servlet3.0规范提供的注解方式注册

```
@WebServlet
@WebListener
@WebFilter
```

1.声明servlet 及映射

```
@WebServlet(name="HelloServlet",urlPatterns = "/HelloServlet")
/*@WebListener
@WebFilter*/
public class HelloServlet extends HttpServlet {
 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOEx
eption {
 PrintWriter writer = resp.getWriter();
 writer.println("hello servlet!");

  }
}
```

2.加上@ServletComponentScan才会扫描那3个注解

```
@SpringBootApplication
@ServletComponentScan
public class Application {

```

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

}
```

- SpringBoot提供的注册
  - 使用ServletRegistrationBean，FilterRegistrationBean以及ServletListenerRegistrationBean

```

@Configuration
public class MyWebMvcConfigurer {

 @Bean
 public ServletRegistrationBean myServlet(){
 // 声明一个servlet注册器Bean
 ServletRegistrationBean servletRegistrationBean = new ServletRegistrationBean();
 // 设置相应的servlet
  servletRegistrationBean.setServlet(new BeanServlet());
  // 设置名字
  servletRegistrationBean.setName("BeanServlet");
  // 添加映射规则
  servletRegistrationBean.addUrlMappings("/BeanServlet");
  return servletRegistrationBean;

  }
 }
```

3.切换其他嵌入式Servlet容器

  - Spring Boot包含对嵌入式Tomcat，Jetty和Undertow服务器的支持
  - tomcat（默认）
  - Jetty(socket)
  - Undertow(响应式)

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-web</artifactId>
 <!--1.排除tomcat-->
 <exclusions>
 <exclusion>
 <artifactId>spring-boot-starter-tomcat</artifactId>
 <groupId>org.springframework.boot</groupId>
 </exclusion>
  </exclusions>
</dependency>

<!--2.依赖jetty
<dependency>
 <artifactId>spring-boot-starter-jetty</artifactId>
 <groupId>org.springframework.boot</groupId>
</dependency>-->
```

```

<!--2.依赖undertow
<dependency>
 <artifactId>spring-boot-starter-undertow</artifactId>
 <groupId>org.springframework.boot</groupId>
</dependency>-->
```

4.嵌入式Servlet容器自动配置原理 SpringBoot 2.3.6

- ServletWebServerFactoryAutoConfiguration servlet容器自动配置类

```
@Configuration(proxyBeanMethods = false)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
// 只要依赖任意一个servlet容器都会存在该来ServletRequest
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
// 启用servet.xxx的所有的配置信息绑定到ServerProperties
@EnableConfigurationProperties(ServerProperties.class)
@Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
 ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
  ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
  ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
```

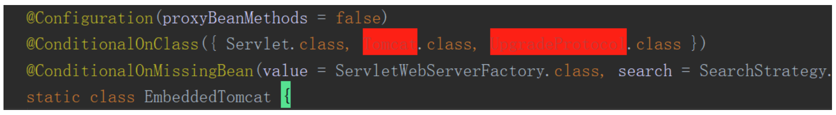
  - **1.为什么可以根据配置的依赖自动使用对应的servlet容器?**
    - 通过@Import 导入Embeddedxxxx

```
@Import({
 ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
 ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
 ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {
```

    - 每个Embeddedxxxx 中都配置了相应的@ConditionalOnClass,会根据当前servlet容器start依赖判断classpath是否存在对应的类， 如果存在就使用对应的servlet容器。 比如EmbeddedTomcat:

```
@Configuration(proxyBeanMethods = false)
// 只要添加了tomcat的场景启动器 则该注解才会生效
@ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.class })
@ConditionalOnMissingBean(value = ServletWebServerFactory.class, search = SearchStrategy.CURRENT)
static class EmbeddedTomcat {
```

      - 如果没有对应的tomcat的场景启动器 该注解就不会生效

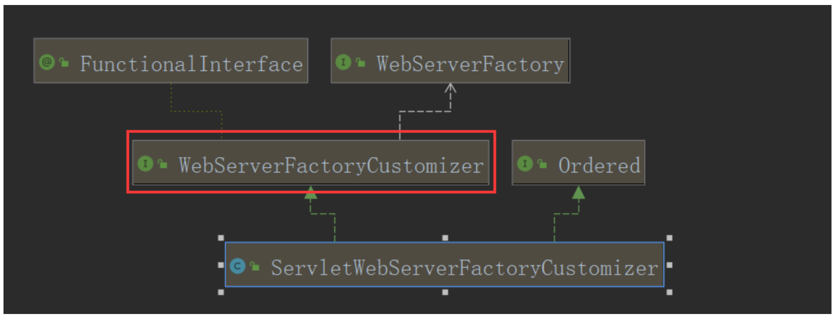


```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ Servlet.class, [redacted].class, [redacted].class })
@ConditionalOnMissingBean(value = ServletWebServerFactory.class, search = SearchStrategy.
static class EmbeddedTomcat {
```

  - **2.怎么根据配置文件中server.xxx 已经 WebServerFactoryCustomizer 去设置servlet容器?**

- ServletWebServerFactoryCustomizer 也实现了WebServerFactoryCustomizer ,说明它也是定制servlet容器的
- Servlet容器配置文件通用定制器

```
@Bean
public ServletWebServerFactoryCustomizer servletWebServerFactoryCustomizer(ServerProperties serverProperties) {
 return new ServletWebServerFactoryCustomizer(serverProperties);
}
```



- 根据配置文件中server.xxx 来进行定制servlet容器

```java
@Override
public void customize(ConfigurableServletWebServerFactory factory) {
 PropertyMapper map = PropertyMapper.get().alwaysApplyingWhenNonNull();
 // if(serverProperties.getPort()!=null){
 // factory.setPort(serverProperties.getPort())
 //}
 map.from(this.serverProperties::getPort).to(factory::setPort);
 map.from(this.serverProperties::getAddress).to(factory::setAddress);
 map.from(this.serverProperties.getServlet()::getContextPath).to(factory::setContextPath);
  map.from(this.serverProperties.getServlet()::getApplicationDisplayName).to(factory::setDisplayName);
  map.from(this.serverProperties.getServlet()::isRegisterDefaultServlet).to(factory::setRegisterDefaultServlet);
  map.from(this.serverProperties.getServlet()::getSession).to(factory::setSession);
  map.from(this.serverProperties::getSsl).to(factory::setSsl);
  map.from(this.serverProperties.getServlet()::getJsp).to(factory::setJsp);
  map.from(this.serverProperties::getCompression).to(factory::setCompression);
  map.from(this.serverProperties::getHttp2).to(factory::setHttp2);
  map.from(this.serverProperties::getServerHeader).to(factory::setServerHeader);
  map.from(this.serverProperties.getServlet()::getContextParameters).to(factory::setInitParameters);
  map.from(this.serverProperties.getShutdown()).to(factory::setShutdown);
 }
```

- TomcatServletWebServerFactoryCustomizer Tomcat配置文件定制器
  - 根据配置文件中servet.tomcat.xxxx 定制servlet容器

```java
@Bean
@ConditionalOnClass(name = "org.apache.catalina.startup.Tomcat")
public TomcatServletWebServerFactoryCustomizer tomcatServletWebServerFactoryCustomizer(
 ServerProperties serverProperties) {
 return new TomcatServletWebServerFactoryCustomizer(serverProperties);
}
```

- **怎么让所有的WebServerFactoryCustomizer Bean一一调用的**
  - BeanPostProcessorsRegistrar
    - 实现ImportBeanDefinitionRegistrar 会提供一个方法，并且提供BeanDefinitionRegistar 让我们去注册bean

```java
@Override
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
 BeanDefinitionRegistry registry) {
 if (this.beanFactory == null) {
```

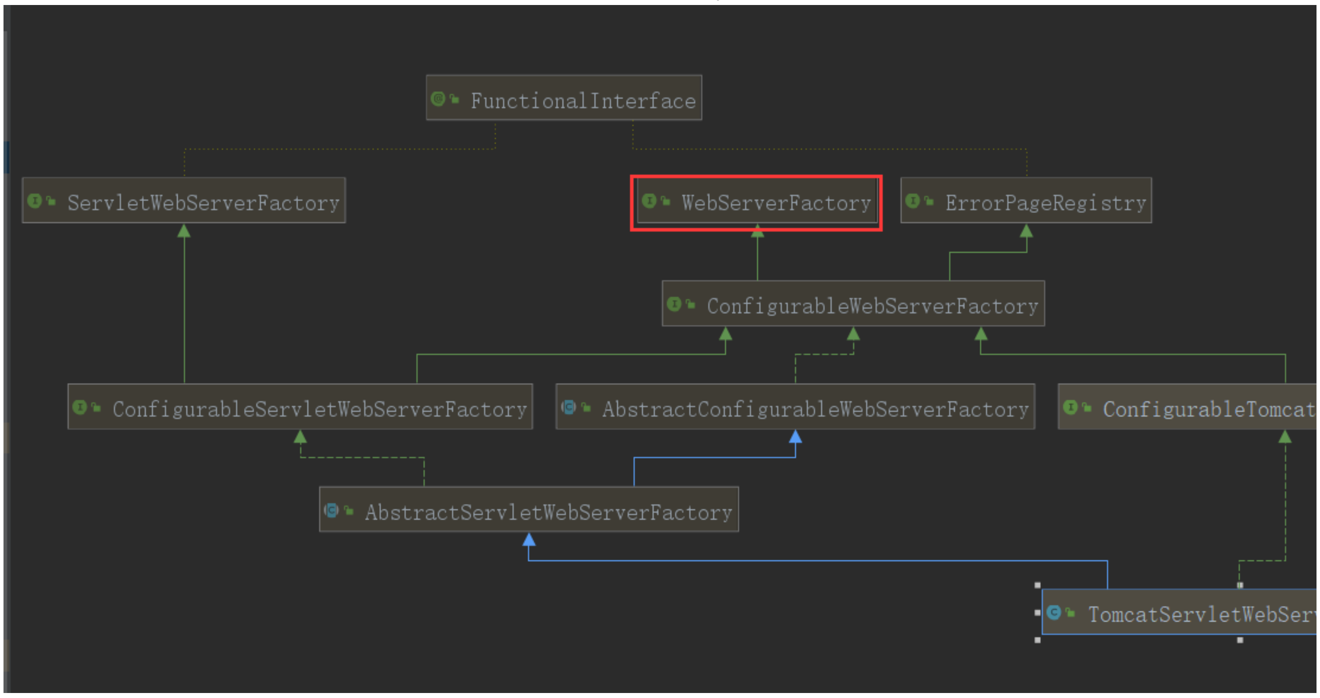
```java
return;
}
registerSyntheticBeanIfMissing(registry, "webServerFactoryCustomizerBeanPostProcessor",
WebServerFactoryCustomizerBeanPostProcessor.class);
registerSyntheticBeanIfMissing(registry, "errorPageRegistrarBeanPostProcessor",
 ErrorPageRegistrarBeanPostProcessor.class);
}
```

- 注册了：WebServerFactoryCustomizerBeanPostProcessor

```java
// BeanPostProcessor 在bean创建时 在初始化时就会调用
public class WebServerFactoryCustomizerBeanPostProcessor implements BeanPostProcessor, BeanFactoryAwar
e {
```

```java
@Override
public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
 // 判断当前创建的bean是不是WebServerFactory
 if (bean instanceof WebServerFactory) {
 postProcessBeforeInitialization((WebServerFactory) bean);
 }
 return bean;
}
```

// 当对应Embeddedxxxx 启用时， 就会在里面配置一个WebServerFactory 类型的一个Bean, 负责创建对应的容器和启动


FunctionalInterface
ServletWebServerFactory
WebServerFactory
ErrorPageRegistry
ConfigurableWebServerFactory
ConfigurableServletWebServerFactory
AbstractConfigurableWebServerFactory
ConfigurableTomcat
AbstractServletWebServerFactory
TomcatServletWebSer

- 1.调用getCustomizers()
- 2.getWebServerFactoryCustomizerBeans() 就获取了所有实现了WebServerFactoryCustomizer接口的Bean
  - 获取 自定义的，和ServletWebServerFactoryCustomizer、TomcatServletWebServerFactoryCustomizer
- 3.在invoke方法中循环调用所有实现了WebServerFactoryCustomizer接口的Bean的customize方法进行一一定制

```java
private void postProcessBeforeInitialization(WebServerFactory webServerFactory) {
 // 调用getCustomizers()
 LambdaSafe.callbacks(WebServerFactoryCustomizer.class, getCustomizers(), webServerFactory)
 .withLogger(WebServerFactoryCustomizerBeanPostProcessor.class)

 .invoke((customizer) -> customizer.customize(webServerFactory));
}
```

```java
private Collection<WebServerFactoryCustomizer<?>> getCustomizers() {
 if (this.customizers == null) {
 // Look up does not include the parent context
 this.customizers = new ArrayList<>(getWebServerFactoryCustomizerBeans());
 this.customizers.sort(AnnotationAwareOrderComparator.INSTANCE);
 this.customizers = Collections.unmodifiableList(this.customizers);
 }
 return this.customizers;
}

 @SuppressWarnings({ "unchecked", "rawtypes" })
 private Collection<WebServerFactoryCustomizer<?>> getWebServerFactoryCustomizerBeans() {
  // beanFactory代表spring容器
  return (Collection) this.beanFactory.getBeansOfType(WebServerFactoryCustomizer.class, false,
false).values();
 }
```

- **3.嵌入式servlet容器是怎么启动的**
- TomcatServletWebServerFactory
  - 自动配置中根据不同的依赖， 启动对应一个Embeddedxxxx， 然后配置一个对应的servlet容器工厂类， 比如tomcat:TomcatServletWebServerFactory
  - 在springboot应用启动的时候， 就会调用容器refresh方法， onRefresh， 调用getWebServer， 创建servlet及启动

```java
@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
 if (this.disableMBeanRegistry) {
 Registry.disableRegistry();
 }
 Tomcat tomcat = new Tomcat();
 File baseDir = (this.baseDirectory != null) ? this.baseDirectory : createTempDir("tomcat");
 tomcat.setBaseDir(baseDir.getAbsolutePath());
 Connector connector = new Connector(this.protocol);
  connector.setThrowOnFailure(true);
  tomcat.getService().addConnector(connector);
  customizeConnector(connector);
  tomcat.setConnector(connector);
  tomcat.getHost().setAutoDeploy(false);
  configureEngine(tomcat.getEngine());
  for (Connector additionalConnector : this.additionalTomcatConnectors) {
  tomcat.getService().addConnector(additionalConnector);
  }
  prepareContext(tomcat.getHost(), initializers);
  return getTomcatWebServer(tomcat);
 }
```

```java
private void initialize() throws WebServerException {
```

```
logger.info("Tomcat initialized with port(s): " + getPortsDescription(false));
synchronized (this.monitor) {
try {
addInstanceIdToEngineName();

Context context = findContext();
context.addLifecycleListener((event) -> {
if (context.equals(event.getSource()) && Lifecycle.START_EVENT.equals(event.getType())) {
 // Remove service connectors so that protocol binding doesn't
 // happen when the service is started.
 removeServiceConnectors();
 }
 });

 // 它就会启动tomcat
 this.tomcat.start();
```

5.使用外部Servlet容器

- 外部servlet容器
    - 服务器、本机 安装tomcat 环境变量...
    - 部署： war---运维--->tomcat webapp startup.sh   启动
    - 开发： 将开发绑定本地tomcat
    - 开发 、 运维   服务器配置 war
- 内嵌servlet容器：
    - 部署： jar---> 运维---java -jar 启动

使用：

1. 下载tomcat服务

2.设置当前maven项目的打包方式

```

<!--打包方式 默认是jar-->
<packaging>war</packaging>
```

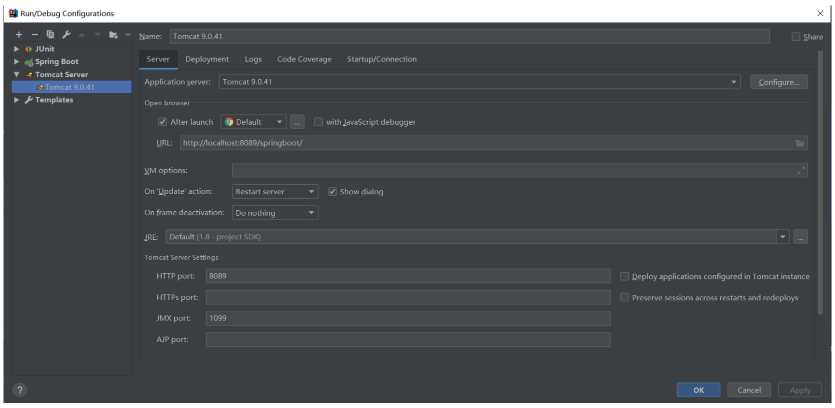
3.让tomcat相关的依赖不参与打包部署 ，因为外置tomcat服务器已经有这些jar包

```
<!--让它不参与打包部署-->
<dependency>
 <artifactId>spring-boot-starter-tomcat</artifactId>
 <groupId>org.springframework.boot</groupId>
 <scope>provided</scope>
</dependency>
```

4. 为了让它支持springboot需要加上： 才能启动springboot应用

```java
public class TomcatStartSpringBoot extends SpringBootServletInitializer {
 @Override
 protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
 return builder.sources(Application.class);
 }
}
```

5. 在idea中运行



Run/Debug Configurations

Name: Tomcat 9.0.41

Server | Deployment | Logs | Code Coverage | Startup/Connection

Application server: Tomcat 9.0.41

Open browser

☑ After launch: Default ☐ with JavaScript debugger

URL: http://localhost:8089/springboot/

VM options:

On 'Update' action: Restart server ☑ Show dialog

On frame deactivation: Do nothing

JRE: Default (1.8 - project SDK)

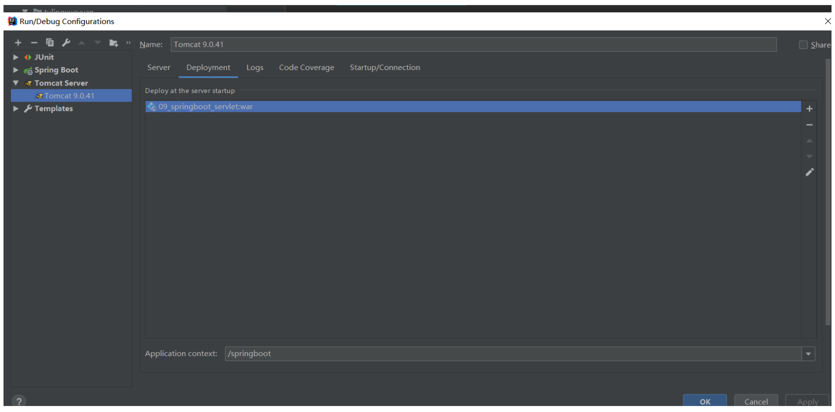
Tomcat Server Settings

HTTP port: 8089 ☐ Deploy applications configured in Tomcat instance

HTTPs port: ☐ Preserve sessions across restarts and redeploys

JMX port: 1099

AJP port:



Run/Debug Configurations

Name: Tomcat 9.0.41

Server | Deployment | Logs | Code Coverage | Startup/Connection

Deploy at the server startup

09_springboot_servlet:war

Application context: /springboot

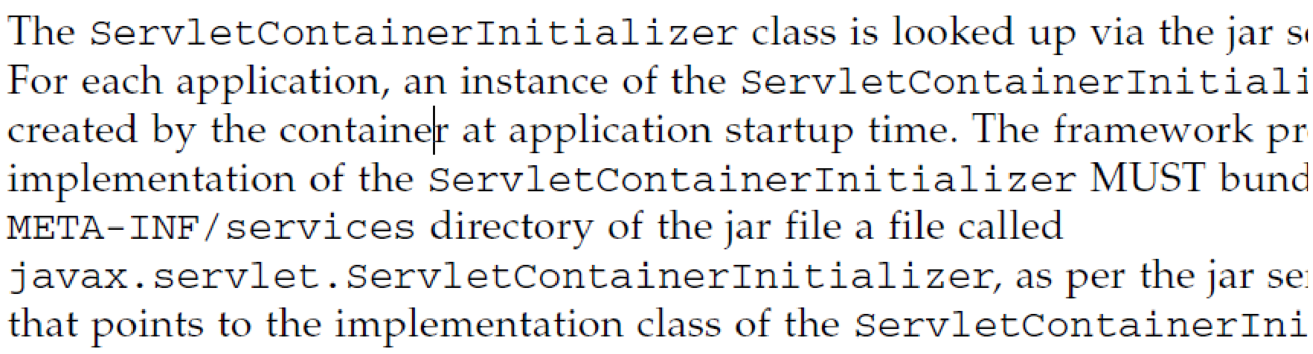
6.外部Servlet容器启动SpringBoot应用原理

tomcat---> web.xml--filter servlet listener 3.0+

tomcat不会主动去启动springboot应用，， 所以tomcat启动的时候肯定调用了SpringBootServletInitializer的SpringApplicationBuilder， 就会启动springboot

```java
public class TomcatStartSpringBoot extends SpringBootServletInitializer {
 @Override
 protected SpringApplicationBuilder (SpringApplicationBuilder builder) {
 return builder.sources(Application.class);
 }
}
```

servlet3.0 规范官方文档： 8.2.4

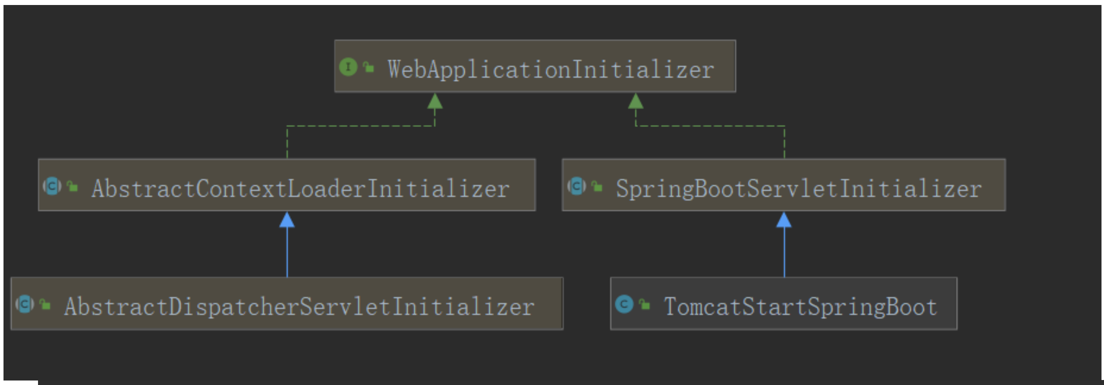

The ServletContainerInitializer class is looked up via the jar s
For each application, an instance of the ServletContainerInitiali
created by the container at application startup time. The framework pr
implementation of the ServletContainerInitializer MUST bund
META-INF/services directory of the jar file a file called
javax.servlet.ServletContainerInitializer, as per the jar se
that points to the implementation class of the ServletContainerIni

大概： 当servlet容器启动时候 就会去META-INF/services 文件夹中找到javax.servlet.ServletContainerInitializer, 这个文件里面肯定绑定一个ServletContainerInitializer. 当servlet容器启动时候就会去该文件中找到ServletContainerInitializer的实现类，从而创建它的实例调用onstartUp

- @HandlesTypes(WebApplicationInitializer.class).
    - @HandlesTypes传入的类为ServletContainerInitializer感兴趣的
    - 容器会自动在classpath中找到 WebApplicationInitializer 会传入到onStartup方法的webAppInitializerClasses中
    - Set<Class<?>> webAppInitializerClasses 这里面也包括之前定义的TomcatStartSpringBoot

```java

@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {
```

```java
@Override
public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses, ServletContext servletContext
 throws ServletException {

 List<WebApplicationInitializer> initializers = new LinkedList<>();

 if (webAppInitializerClasses != null) {
 for (Class<?> waiClass : webAppInitializerClasses) {
 // 如果不是接口 不是抽象 跟WebApplicationInitializer有关系 就会实例化
  if (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&
  WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
  try {
  initializers.add((WebApplicationInitializer)
  ReflectionUtils.accessibleConstructor(waiClass).newInstance());
  }
  catch (Throwable ex) {
  throw new ServletException("Failed to instantiate WebApplicationInitializer class", ex);
  }
  }
  }
  }

  if (initializers.isEmpty()) {
  servletContext.log("No Spring WebApplicationInitializer types detected on classpath");
  return;
  }

  servletContext.log(initializers.size() + " Spring WebApplicationInitializers detected on
classpath");
  // 排序
  AnnotationAwareOrderComparator.sort(initializers);
  for (WebApplicationInitializer initializer : initializers) {
  initializer.onStartup(servletContext);
  }
 }
```


WebApplicationInitializer
AbstractContextLoaderInitializer
SpringBootServletInitializer
AbstractDispatcherServletInitializer
TomcatStartSpringBoot

```java
@Override
public void onStartup(ServletContext servletContext) throws ServletException {
 // Logger initialization is deferred in case an ordered
 // LogServletContextInitializer is being used
 this.logger = LogFactory.getLog(getClass());
 WebApplicationContext rootApplicationContext = createRootApplicationContext(servletContext);
 if (rootApplicationContext != null) {
 servletContext.addListener(new SpringBootContextLoaderListener(rootApplicationContext,
servletContext));
 }
  else {
  this.logger.debug("No ContextLoaderListener registered, as createRootApplicationContext() did not "
  + "return an application context");
  }
 }
```

- SpringBootServletInitializer
    - 之前定义的TomcatStartSpringBoot 就是继承它

```java
protected WebApplicationContext createRootApplicationContext(ServletContext servletContext) {
 SpringApplicationBuilder builder = createSpringApplicationBuilder();
 builder.main(getClass());
 ApplicationContext parent = getExistingRootWebApplicationContext(servletContext);
 if (parent != null) {
 this.logger.info("Root context already created (using as parent).");
 servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, null);
 builder.initializers(new ParentContextApplicationContextInitializer(parent));
 }
  builder.initializers(new ServletContextApplicationContextInitializer(servletContext));
  builder.contextClass(AnnotationConfigServletWebServerApplicationContext.class);
  // 调用configure
  builder = configure(builder);
  builder.listeners(new WebEnvironmentPropertySourceInitializer(servletContext));
  SpringApplication application = builder.build();
  if (application.getAllSources().isEmpty()
  && MergedAnnotations.from(getClass(), SearchStrategy.TYPE_HIERARCHY).isPresent(Configuration.class))
{
```

```
application.addPrimarySources(Collections.singleton(getClass()));
}
Assert.state(!application.getAllSources().isEmpty(),
"No SpringApplication sources have been defined. Either override the "
+ "configure method or add an @Configuration annotation");
// Ensure error pages are registered
if (this.registerErrorPageFilter) {
application.addPrimarySources(Collections.singleton(ErrorPageFilterConfiguration.class));
}
application.setRegisterShutdownHook(false);
return run(application);
}
```

- 当调用configure就会来到TomcatStartSpringBoot .configure
    - 将Springboot启动类传入到builder.source

```
@Override
protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
 return builder.sources(Application.class);
}
```

// 调用SpringApplication application = builder.build(); 就会根据传入的Springboot启动类来构建一个SpringApplication

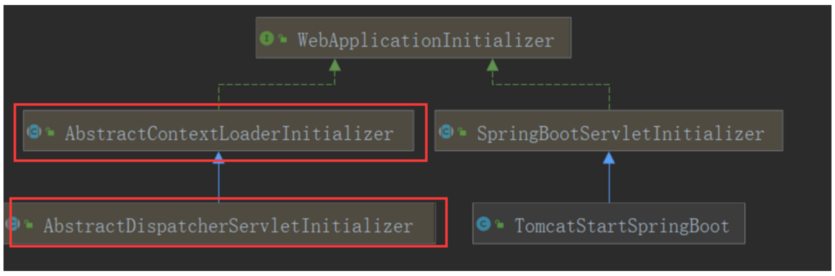
```
public SpringApplication build(String... args) {
 configureAsChildIfNecessary(args);
 this.application.addPrimarySources(this.sources);
 return this.application;
}
```

// 调用 return run(application); 就会帮我启动springboot应用

```
protected WebApplicationContext run(SpringApplication application) {
 return (WebApplicationContext) application.run();
}
```

它就相当于我们的

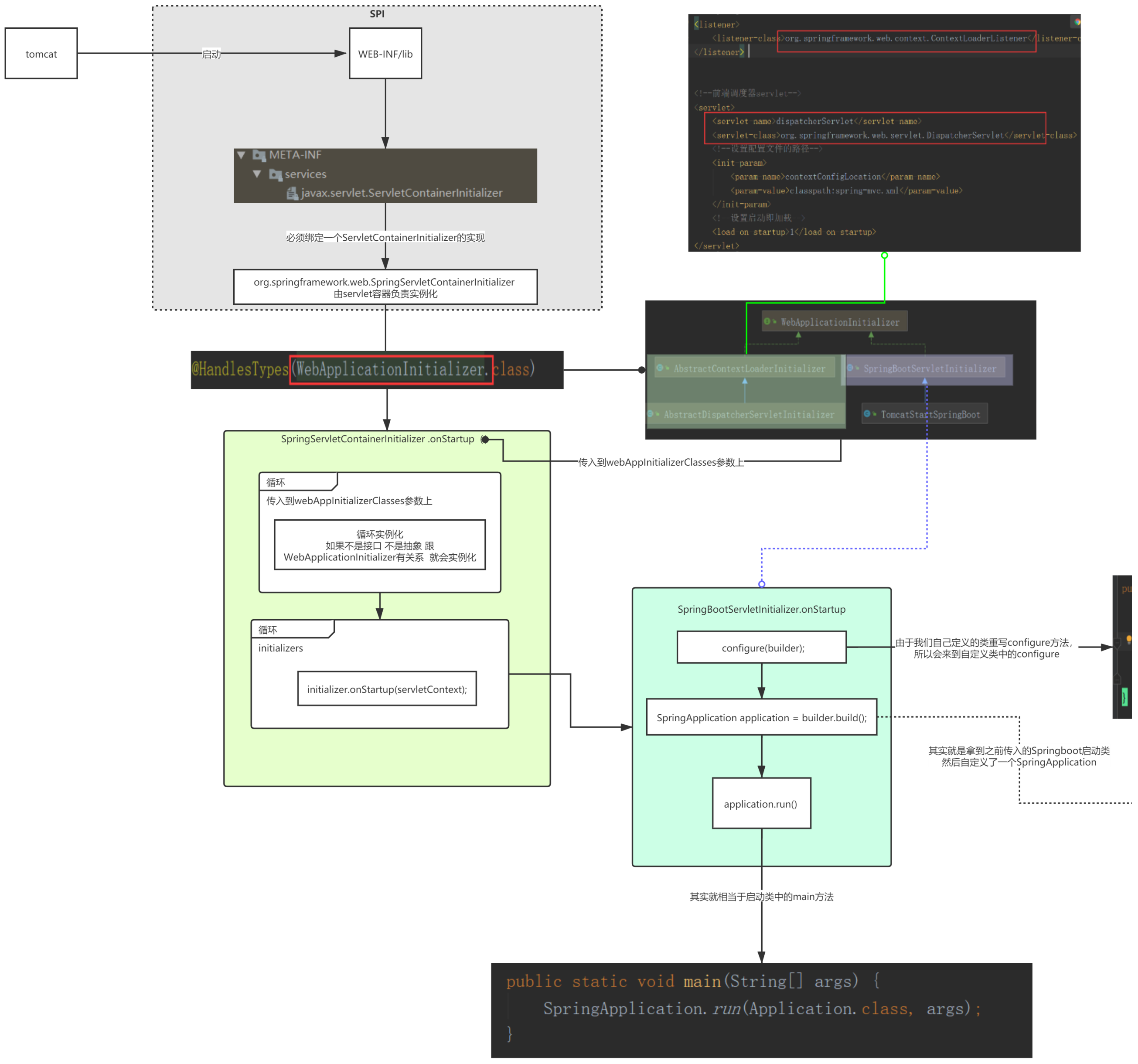
```
public static void main(String[] args) {
 SpringApplication.run(Application.class, args);
}
```


WebApplicationInitializer
AbstractContextLoaderInitializer
SpringBootServletInitializer
AbstractDispatcherServletInitializer
TomcatStartSpringBoot

其实这2个实现类就是帮我创建ContextLoaderListener 和DispatcherServlet

```
<listener>
 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<!--全局参数：spring配置文件-->
<context-param>
 <param-name>contextConfigLocation</param-name>
```

```xml
 <param-value>classpath:spring-core.xml</param-value>
</context-param>


<!--前端调度器servlet-->
<servlet>
 <servlet-name>dispatcherServlet</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <!--设置配置文件的路径-->
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>classpath:spring-mvc.xml</param-value>
 </init-param>
 <!--设置启动即加载-->
 <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
 <servlet-name>dispatcherServlet</servlet-name>
 <url-pattern>/</url-pattern>
</servlet-mapping>
```


SPI
tomcat
启动
WEB-INF/lib
META-INF
services
javax.servlet.ServletContainerInitializer
必须绑定一个ServletContainerInitializer的实现
org.springframework.web.SpringServletContainerInitializer
由servlet容器负责实例化
@HandlesTypes(WebApplicationInitializer.class)
WebApplicationInitializer
AbstractContextLoaderInitializer
SpringBootServletInitializer
AbstractDispatcherServletInitializer
TomcatStartSpringBoot
传入到webAppInitializerClasses参数上
SpringServletContainerInitializer .onStartup
循环
传入到webAppInitializerClasses参数上
循环实例化
如果不是接口 不是抽象 跟
WebApplicationInitializer有关系 就会实例化
循环
initializers
initializer.onStartup(servletContext);
SpringBootServletInitializer.onStartup
configure(builder);
SpringApplication application = builder.build();
application.run()
由于我们自己定义的类重写configure方法，
所以会来到自定义类中的configure
其实就是拿到之前传入的Springboot启动类
然后自定义了一个SpringApplication
其实就相当于启动类中的main方法

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-c
</listener>


<!--前端调度器servlet-->
<servlet>
    <servlet-name>dispatcherServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <!--设置配置文件的路径-->
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
    </init-param>
    <!--设置启动即加载-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

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

SpringBoot作为单体Web应用的使用

如果需要动态展示Springmvc的数据到页面上需要使用模板引擎技术：
SpringBoot提供以下模板引擎技术的支持：

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

以Freemarker为例

1. 添加freemarker的依赖

```
<dependency>
 <groupId>org.springframework.boot</groupId>
```

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

2. 设置freemakrer的全局配置

```
spring.freemarker.cache=false

spring.freemarker.charset=UTF-8

spring.freemarker.suffix=.html
```

3. 添加freemarker的页面

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<#list usernames as username>
<h1>${username}</h1>
 </#list>
 </body>
 </html>
```

4. 对应的控制器

```

/***
 * @Author 徐庶 QQ:1092002729
 * @Slogan 致敬大师，致敬未来的你
 */
@Controller
@RequestMapping("user")
public class UserController {

  @RequestMapping("index")
  public String index(Model model){
  model.addAttribute("username","xushu");
  return "index";
  }

  @RequestMapping("list")
  public String list(Model model){
  List<String> list= Arrays.asList("xushu","zhangsan","lisi");
  model.addAttribute("usernames",list);
  return "list";
  }
 }
```