# Spring Boot初体验

## 基本概念

***应用分为两个方面：功能性、非功能性***

功能性：系统所设计的业务范畴

非功能性：安全、性能、监控、数据指标（CPU利用率、网卡使用率）

***Spring Boot 规约大于配置***，大多数组件，不需要自行配置，而是自动组装！简化开发，大多数情况，使用默认即可！

production-ready 就是非功能性范畴！

独立Spring 应用，不需要外部依赖，依赖容器（Tomcat）

嵌入式 Tomcat Jetty

外部配置：启动参数、配置文件、环境变量

外部应用：Servlet 应用、Spring Web MVC、Spring Web Flux、WebSocket、WebService

SQL：JDBC、JPA、ORM

NoSQL（Not Only SQL）：Redis、ElasticSearch、Hbase

Mono : 0 - 1 元素，Optional

Flux：0 - N 个元素，类似于 Iterable 或者 Collection

Req -> WebFlux -> 1 - N 线程执行任务执行函数式任务

它是推的方式！

Java 9 里面API 称之为 Flow（流）

Publisher -> publish(1)

Subscription（1）：订阅消息

Subs(A)#onNext() -> Subs(B)#onNext() -> Subs(C)#onNext()

Reactive 是推模式（Push）

Iterator 是拉模式（Pull）

## 构建多模块应用

***创建工程命令：***

***mvn archetype:generate -DgroupId=com.rqb -DartifactId=first-spring-boot-app -Dversion=1.0.0-SNAPSHOT -DinteractiveMode=false***

1. 修改主工程类型jar -> pom
2. 新建 web 工程，讲遗留代码移动到 web java 目录下
3. 再从 web 工程，独立出 model 工程
4. 将 web 工程依赖 model 工程
5. 重复步骤 3，独立出 persistence
6. 再从 persistence 添加 model 的依赖
7. 最终依赖关系 web -> persistence -> model

## 构建可执行 jar 或者 war

web-1.0.0-SNAPSHOT.jar中没有主清单属性？

需要一个 Spring Boot 的插件：

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

jar 规范里面，有一个 MANIFEST.MF，里面有一个 Main-Class 的属性。API：java.util.jar.Manifest#getAttributes

***从 jar 切换成 war 打包方式***

修改成 war，绕过war插件的限制

(1)笨办法：创建webapp/WEB-INF 目录（相对于src/main）新建一个空的web.xml

或者

(2)使用

<plugin>

<artifactId>maven-war-plugin</artifactId>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

注意事项

BOOT-INF 是 Spring Boot 1.4 开始才有

当使用依赖或者插件时，如果版本是 Milestone的时候，需要增加：

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

<url>https://repo.spring.io/libs-milestone</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<url>http://repo.spring.io/snapshot</url>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<url>http://repo.spring.io/milestone</url>

</pluginRepository>

</pluginRepositories>

META-INF/MANIFEST.MF 里面有指定两个属性

Main-Class

Start-Class

例子：

Main-Class: org.springframework.boot.loader.JarLauncher

Start-Class: com.gupao.App

除了 jar 或者 war 启动的方式，还有目录启动方式

目录启动方式可以帮助解决老旧的jar 不支持 Spring Boot 新方式，比如老版本的 MyBatis

如果是 jar 包，解压后，跳转解压目录，并且执行java命令启动，启动类是 org.springframework.boot.loader.JarLauncher

如果是 war包，解压后，跳转解压目录，并且执行java命令启动类是org.springframework.boot.loader.WarLauncher

查看端口

注意控制台日志：Netty started on port(s): 8080

参数[--server.port=0]使用随机可用端口

自动装配的模式

XXX-AutoConfiguration

## 问答互动

问：webFluxConfiguration里面的映射路径和controller 里面的路径有什么区别吗

答：基本上是没有区别的，注意，不重复定义，或者URL 语义有重复！

问：webFlux不是跟mvc不能一起吗，怎么一起启动了

答：spring-boot-starter-webmvc 和 spring-boot-starter-webflux 可以放在同一个应用，可是 webflux 不会工作，默认使用webmvc，webflux 不会被采用。其实 webflux 是兼容 Annotation 驱动，比如@RequestMapping

问：webFlux可以定义restFull吗

答：可以的，支持的！

问：spring的老项目迁移到springboot，怎么弄

答：老的XML 方式采用 @ImportResource 导入！

问：嵌入式tomcat如何调优

答：第一种通过 application.properties文件调整配置参数

第二种通过接口回调：

TomcatConnectorCustomizer

TomcatContextCustomizer

# Spring MVC

## Spring MVC视图技术

### Thymeleaf 视图技术

### 模板寻址

### 模板缓存

### 模板渲染逻辑

Spring MVC核心总控制器DispatcherServlet

C: DispatcherServlet

M:

Spring MVC （部分）

Model/ModelMap/ModelAndView

@ModelAttribute

模板引擎（通常）

## Spring MVC源码分析

# Spring REST

## Spring REST理论基础

Wiki

<https://en.wikipedia.org/wiki/Representational_state_transfer>

RPC ( Remote Procedure Call )

语言相关的远程调用：

* Java - RMI（Remote Method Invocation）
* .NET - COM+、

语言无关的远程调用

* SOA
* Web Services

SOAP（传输介质协议）

HTTP、SMTP（通讯协议）

* 微服务（MSA）
* REST

HTML、JSON、XML 等等

HTTP（通讯协议）

HTTP 1.1

短连接

Keep-Alive

连接池

Long Polling

HTTP/2

长连接

技术

Spring 客户端 ： RestTemplate

Spring WebMVC ： @RestController = @Controller + @ResponseBody + @RequestBody

Spring Cloud : RestTemplate 扩展 + @LoadBalanced

REST（英文）

## 服务端核心接口

--定义相关

@Controller

@RestController

--映射相关

@RequestMapping

@PathVariable

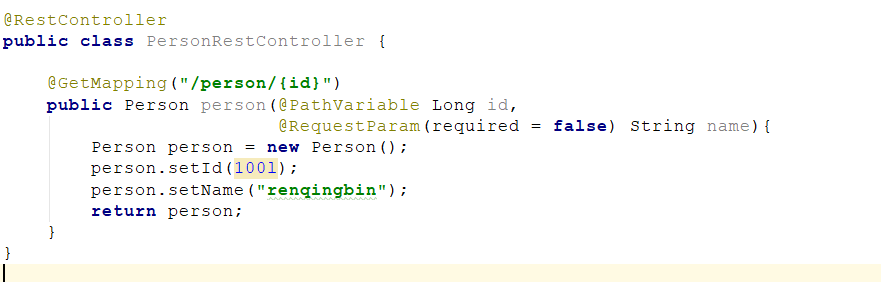
@RequestMapping(method = RequestMethod.***GET***)  
**public** @**interface** GetMapping {

}

--方法相关

RequestMethod

示例：



幂等操作

* PUT

初始状态：0

修改状态：1 \* N

最终状态：1

* DELETE

初始状态：1

修改状态：0 \* N

最终状态：0

* 非幂等

POST

初始状态：1

修改状态：1 + 1 =2

N次修改： 1+ N = N+1

最终状态：N+1

幂等/非幂等 依赖于服务端实现，这种方式是一种契约

## 自描述消息

第一优先顺序：text/html -> application/xhtml+xml -> application/xml

第二优先顺序：image/webp -> image/apng

学习源码的路径：

@EnableWebMvc

​ DelegatingWebMvcConfiguration

​ WebMvcConfigurationSupport#addDefaultHttpMessageConverters

所有的 HTTP 自描述消息处理器均在 messageConverters（类型：HttpMessageConverter)，这个集合会传递到 RequestMappingHandlerAdapter，最终控制写出。

messageConverters，其中包含很多自描述消息类型的处理，比如 JSON、XML、TEXT等等

以 application/json 为例，Spring Boot 中默认使用 Jackson2 序列化方式，其中媒体类型：application/json，它的处理类 MappingJackson2HttpMessageConverter，提供两类方法：

读read\* ：通过 HTTP 请求内容转化成对应的 Bean

写write\*： 通过 Bean 序列化成对应文本内容作为响应内容

问题：为什么第一次是JSON，后来怎加了 XML 依赖，又变成了 XML 内用输出

回答：Spring Boot 应用默认没有增加XML 处理器（HttpMessageConverter）实现，所以最后采用轮训的方式去逐一尝试是否可以 canWrite(POJO) ,如果返回 true，说明可以序列化该 POJO 对象，那么 Jackson 2 恰好能处理，那么Jackson 输出了。

问题：当 Accept 请求头未被制定时，为什么还是 JSON 来处理

回答：这个依赖于 messageConverters 的插入顺序。

问题：优先级是默认的是吧 可以修改吗

回答：是可以调整的，通过extendMessageConverters 方法调整

**扩展自描述消息**

Person

JSON 格式（application/json)

{

"id":1,

"name":"小马哥"

}

XML 格式（application/xml）

<Person>

<id>1</id>

<name>小马哥</name>

</Person>

Properties 格式（application/properties+person)

（需要扩展）

person.id = 1

person.name = 小马哥

实现 AbstractHttpMessageConverter 抽象类

supports 方法：是否支持当前POJO类型

readInternal 方法：读取 HTTP 请求中的内容，并且转化成相应的POJO对象（通过 Properties 内容转化成 JSON）

writeInternal 方法：将 POJO 的内容序列化成文本内容（Properties格式），最终输出到 HTTP 响应中（通过 JSON 内容转化成 Properties ）

@RequestMappng 中的 consumes 对应 请求头 “Content-Type”

@RequestMappng 中的 produces 对应 请求头 “Accept”

HttpMessageConverter 执行逻辑：

读操作：尝试是否能读取，canRead 方法去尝试，如果返回 true 下一步执行 read

写操作：尝试是否能写入，canWrite 方法去尝试，如果返回 true 下一步执行 write

注解驱动

@RequestBody

JSON -> MappingJackson2HttpMessageConverter

TEXT -> StringHttpMessageConverter

@ResponseBody

JSON -> MappingJackson2HttpMessageConverter

TEXT -> StringHttpMessageConverter

返回值处理类：RequestResponseBodyMethodProcessor

接口编程

ResponseEntity extends HttpEntity

RequestEntity extends HttpEntity

返回值处理类：HttpEntityMethodProcessor

媒体类型（MediaType）

org.springframework.http.MediaType#APPLICATION\_JSON\_UTF8\_VALUE

"application/json;charset=UTF-8"

HTTP 消息转换器（HttpMessageConverter）

application/json

MappingJackson2HttpMessageConverter

text/html

StringHttpMessageConverter

代码导读

@EnableWebMvc

导入 DelegatingWebMvcConfiguration（配置 Class）

注册 WebMvcConfigurer

装配各种 Spring MVC 需要的Bean

注解驱动扩展点

HandlerMethodArgumentResolver

HandlerMethodReturnValueHandler

@RequestBody 和 @ResponseBody 实现类

RequestResponseBodyMethodProcessor

HttpEntityMethodProcessor

实现 WebMvcConfigurer？

WebMvcConfigurerAdapter 实现

URI 与 URL

U : Uniform

R : Resource

I：鉴别

L : 定位

山东和河南都有一个张三，张三就是URI，具体的河南的张三或者山东的张三就是URL

URI

URI = scheme:[//authority]path[?query][#fragment]

scheme : HTTP、wechat

URL

protocol 协议

## Cacheability（可缓存性）

@ResponseBody -> 响应体（Response Body）

响应（Response）

响应头（Headers）

请求方法

HEAD

元信息（Meta-Data）

Accept-Language -> Locale

Connection -> Keep-Alive

实现

多值 Map MultiValueMap

Key : Value = 1 : N

Name : Value = 1 : N

public class HttpHeaders implements MultiValueMap<String, String>, Serializable {

...

}

响应体

业务信息（Business Data）

Body：HTTP 实体、REST

@ResponseBody

HttpEntity.body 属性（泛型结构）

Payload ： 消息 JMS、事件、SOAP

public class HttpEntity<T> {

...

private final HttpHeaders headers;

@Nullable

private final T body;

}

HTTP 状态码（org.springframework.http.HttpStatus）

200

org.springframework.http.HttpStatus#OK

304

org.springframework.http.HttpStatus#NOT\_MODIFIED

第一次完整请求，获取响应头（200），直接获取

第二次请求，只读取头信息，响应头（304），客户端（流量器）取上次 Body 结果

400

org.springframework.http.HttpStatus#BAD\_REQUEST

404

500

## Uniform interface（统一接口）

资源定位 - URI

资源操作 - HTTP 动词

GET

@GetMapping

注解属性别名和覆盖（https://github.com/spring-projects/spring-framework/wiki/Spring-Annotation-Programming-Model#attribute-aliases-and-overrides）

Spring Framework 4.2 引入

Spring Boot 1.3 才可以使用

Spring Boot 加以发展

@RequestMapping(method = RequestMethod.POST) // 注解“派生性”

public @interface PostMapping {

...

@AliasFor(annotation = RequestMapping.class) // 注解别名

String name() default "";

...

}

@PostMapping 是注解，@RequestMapping 是 @PostMapping 的注解：

@RequestMapping 是 @PostMapping 的元注解

@RequestMapping 元标注了 @PostMapping

@AliasFor 只能标注在目标注解的属性，所annotation()的注解必须是元注解，该注解 attribute() 必须元注解的属性

PUT

@PutMapping

POST

@PostMapping

PATCH

@PatchMapping

限制

Servlet API 没有规定 PATCH

Spring Web 对其做了扩展

public abstract class FrameworkServlet extends HttpServletBean implements ApplicationContextAware {

...

protected void service(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

HttpMethod httpMethod = HttpMethod.resolve(request.getMethod());

if (httpMethod == HttpMethod.PATCH || httpMethod == null) {

processRequest(request, response);

}

else {

super.service(request, response);

}

}

...

}

DELETE

@DeleteMapping