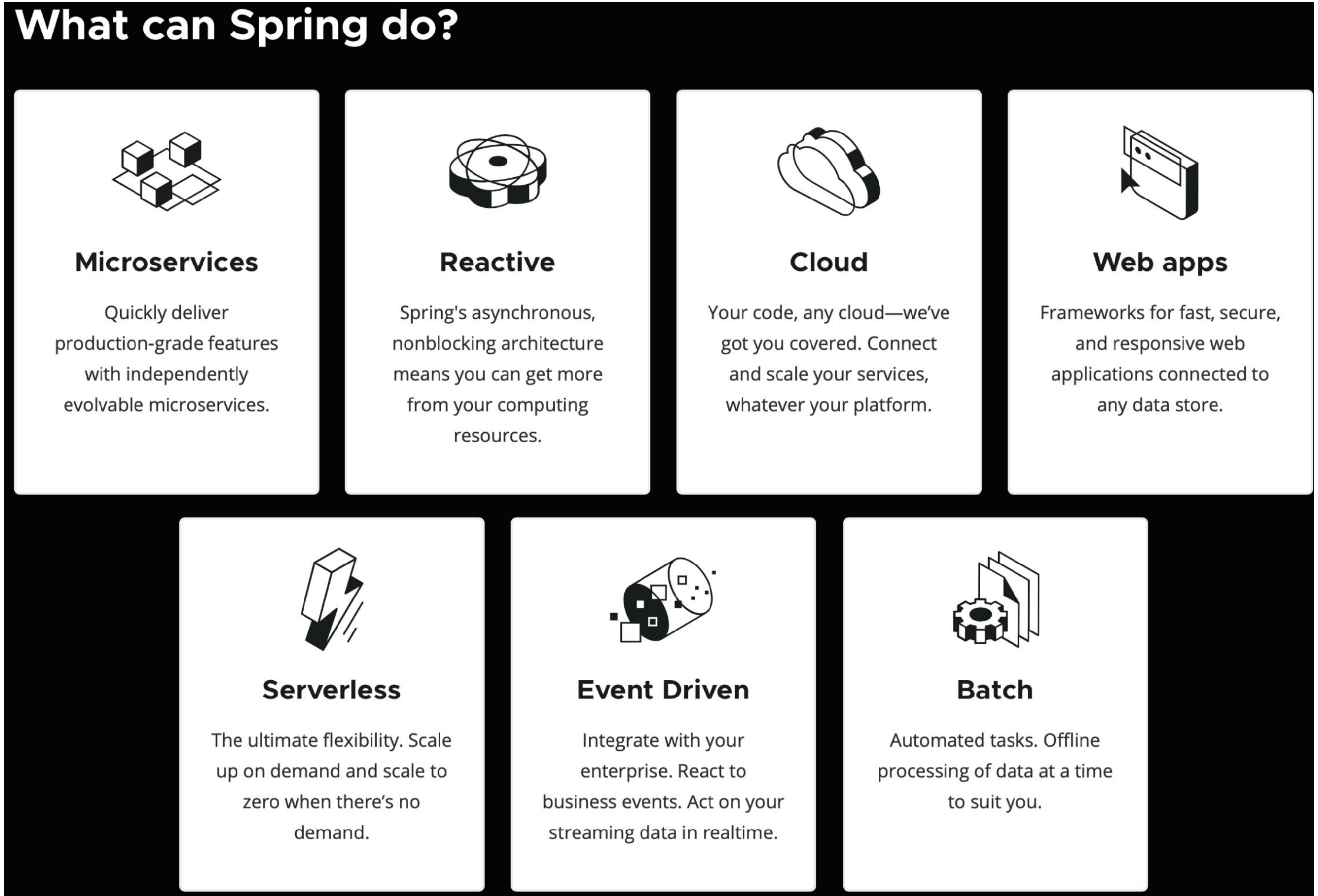
爪哇教育
Power Human With Education

SpringBoot

SpringBoot概述

Spring能够做什么？



What can Spring do?

Microservices

Quickly deliver production-grade features with independently evolvable microservices.

Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

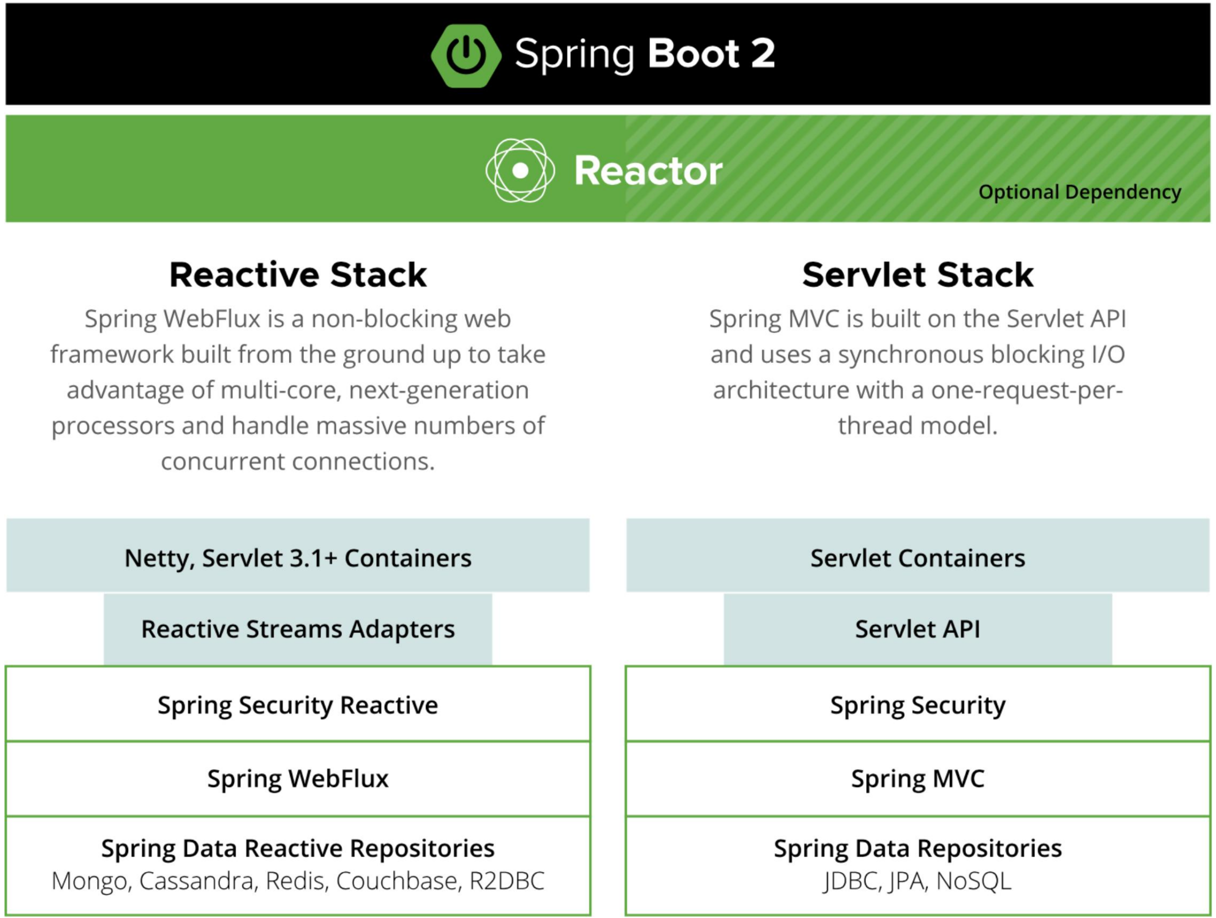
Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

Batch

Automated tasks. Offline processing of data at a time to suit you.

SpringBoot2架构



Reactive Stack

Spring WebFlux is a non-blocking web framework built from the ground up to take advantage of multi-core, next-generation processors and handle massive numbers of concurrent connections.

Servlet Stack

Spring MVC is built on the Servlet API and uses a synchronous blocking I/O architecture with a one-request-per-thread model.

SpringBoot2特征

Features

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Provide production-ready features such as metrics, health checks, and externalized configuration
- Absolutely no code generation and no requirement for XML configuration

创建SpringBoot项目

方式一：基于maven项目下创建

配置要求：java8及以上、maven3.5+

增加SpringBoot Maven父依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.4</version>
</parent>
```

增加Spring Web依赖：

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

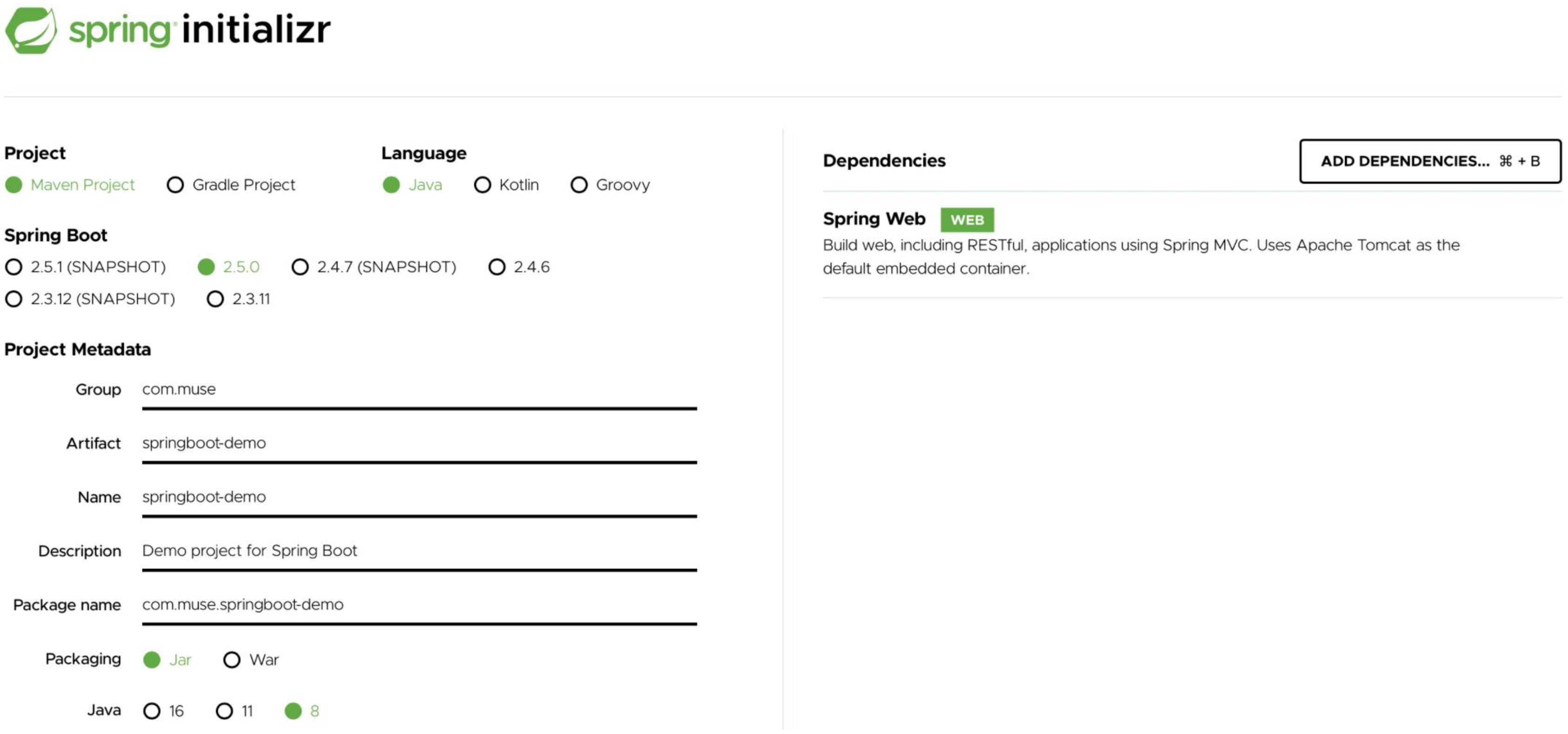
方式一：基于maven项目下创建

编写SpringBoot的主启动类：

```
@RestController
@EnableAutoConfiguration
public class MyApplication {
    @RequestMapping("/")
    String home() {
    return "Hello World!";
}
public static void main(String[] args) {
    SpringApplication.run(MyApplication.class, args);
    }
}
```

方式二：通过spring官网创建

创建springboot项目https://start.spring.io


spring initializr

Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
2.5.1 (SNAPSHOT) [✓ 2.5.0] 2.4.7 (SNAPSHOT) 2.4.6
2.3.12 (SNAPSHOT) 2.3.11

Project Metadata

Group	com.muse
Artifact	springboot-demo
Name	springboot-demo
Description	Demo project for Spring Boot
Package name	com.muse.springboot-demo
Packaging	[✓ Jar] War
Java	16 11 [✓ 8]

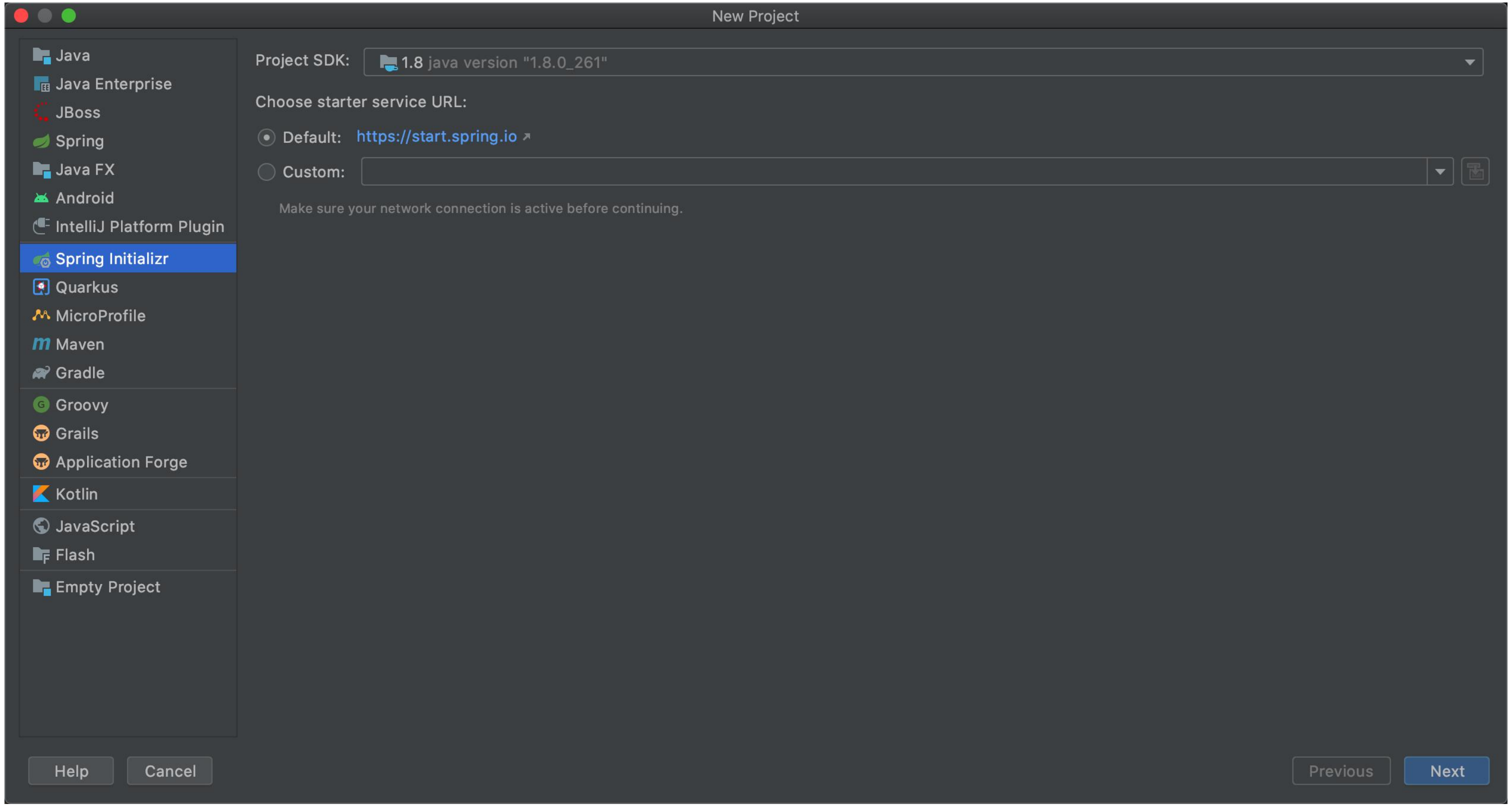
Dependencies

ADD DEPENDENCIES... ⌘ + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

方式三：通过Idea创建

File——>New——>Project...——>Spring Initializr



Project SDK: 1.8 java version "1.8.0_261"

Choose starter service URL:

(●) Default: https://start.spring.io ↗

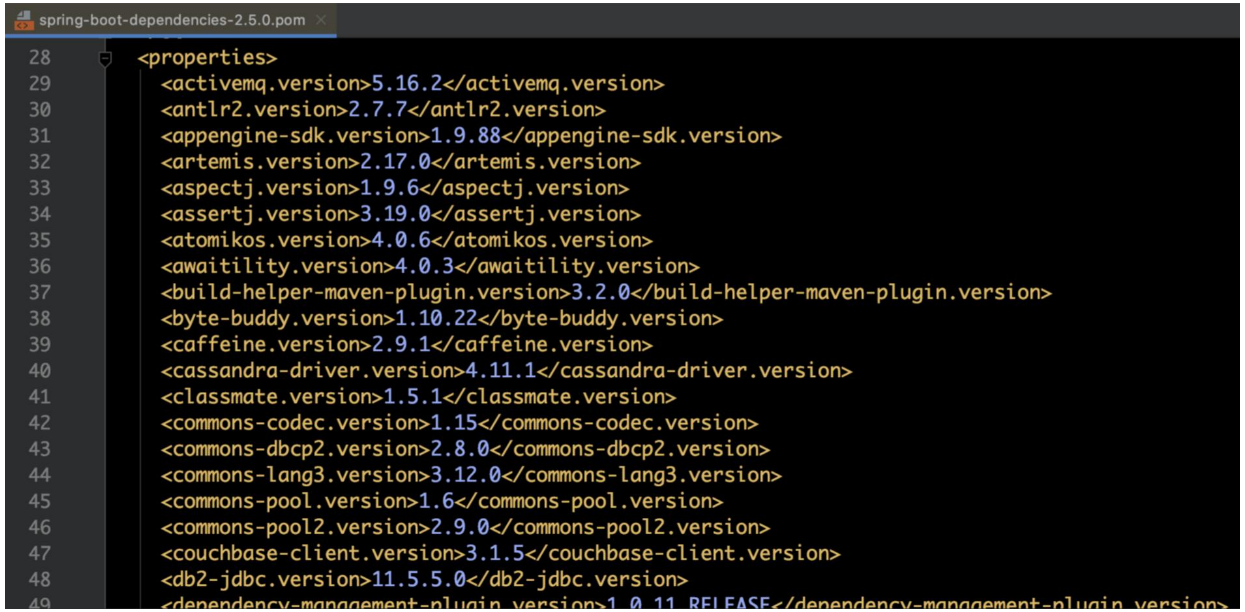
( ) Custom:

Make sure your network connection is active before continuing.

SpringBoot主要特性

SpringBoot的依赖管理

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-dependencies</artifactId>
<version>2.5.1</version>
```



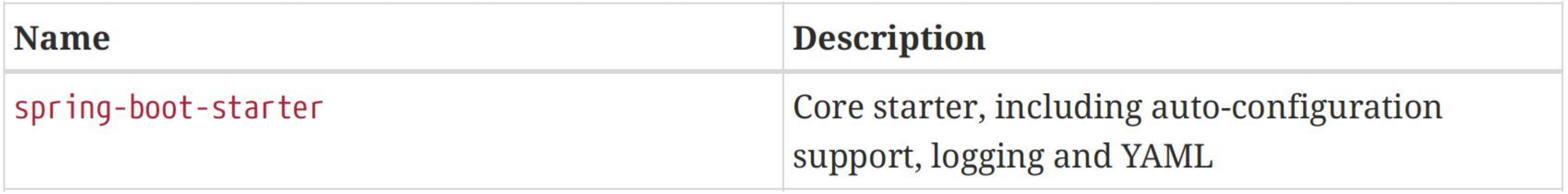
```
<properties>
  <activemq.version>5.16.2</activemq.version>
  <antlr2.version>2.7.7</antlr2.version>
  <appengine-sdk.version>1.9.88</appengine-sdk.version>
  <artemis.version>2.17.0</artemis.version>
  <aspectj.version>1.9.6</aspectj.version>
  <assertj.version>3.19.0</assertj.version>
  <atomikos.version>4.0.6</atomikos.version>
  <awaitility.version>4.0.3</awaitility.version>
  <build-helper-maven-plugin.version>3.2.0</build-helper-maven-plugin.version>
  <byte-buddy.version>1.10.22</byte-buddy.version>
  <caffeine.version>2.9.1</caffeine.version>
  <cassandra-driver.version>4.11.1</cassandra-driver.version>
  <classmate.version>1.5.1</classmate.version>
  <commons-codec.version>1.15</commons-codec.version>
  <commons-dbcp2.version>2.8.0</commons-dbcp2.version>
  <commons-lang3.version>3.12.0</commons-lang3.version>
  <commons-pool.version>1.6</commons-pool.version>
  <commons-pool2.version>2.9.0</commons-pool2.version>
  <couchbase-client.version>3.1.5</couchbase-client.version>
  <db2-jdbc.version>11.5.5.0</db2-jdbc.version>
  <dependency-management-plugin.version>1.0.11.RELEASE</dependency-management-plugin.version>
```

SpringBoot的Starters

- 我们引入什么场景的starter，那么就会将一整套场景的jar包都引入进来，我们也不需要关注多jar包直接的版本号是否兼容彼此，这块工作spring已经帮我们做好了。

- SpringBoot提供的Starter有哪些？
  https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.build-systems.starters

- 分为三类Starter，分别为：
  application starters
  production starters
  technical starters



Name	Description
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML

自动配置介绍

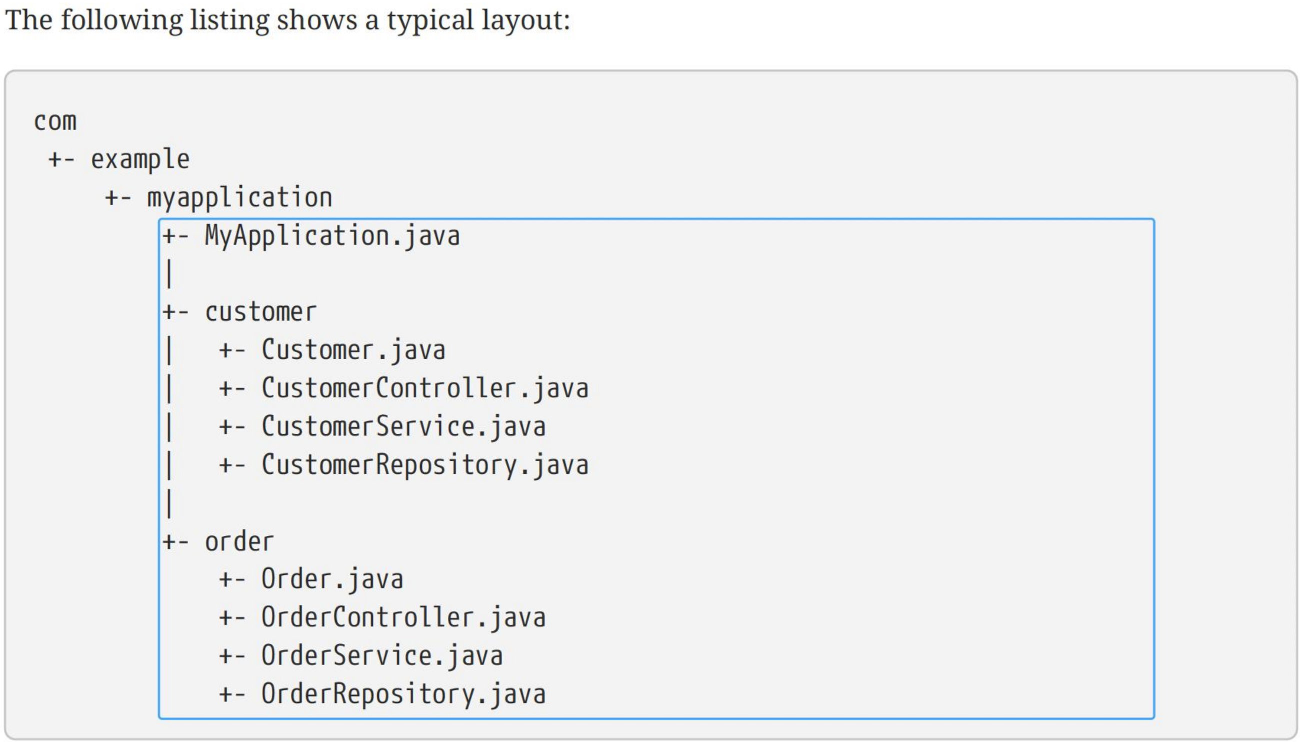
- SpringBoot所有的自动配置功能都在spring-boot-autoconfigure包里面。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
    <version>2.5.1</version>
    <scope>compile</scope>
</dependency>
```

- 以Spring Web场景，分析引入Starter情况

默认包扫描路径

➢ 主程序MyApplication.java所在的包及其下面的所有子包里面的组件都会被默认扫描。


The following listing shows a typical layout:

```
com
 +- example
     +- myapplication
         +- MyApplication.java
         |
         +- customer
         |   +- Customer.java
         |   +- CustomerController.java
         |   +- CustomerService.java
         |   +- CustomerRepository.java
         |
         +- order
             +- Order.java
             +- OrderController.java
             +- OrderService.java
             +- OrderRepository.java
```

自定义包扫描路径

- 如果想要被扫描到，可以指定如下注解：

  @SpringBootApplication(scanBasePackages = "com.muse")

- 或者指定如下注解：

  @ComponentScan("com.muse")

- 建议使用默认包扫描路径即可

SpringBoot配置相关介绍

- SpringBoot支持两种配置类型：
  - application.properties
  - application.yaml
- 简单介绍properties和yaml的配置项书写方式对比
- 默认配置值和配置对应处理类
- 自定义配置绑定方式
  - @ConfigurationProperties + @Component
  - @ConfigurationProperties + @EnableConfigurationProperties
- yaml配置的书写方式介绍
- 添加配置提醒

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

常用注解介绍

- @Configuration
- @ComponentScan
- @Bean
- @Import
- @Conditional
- @ImportResource

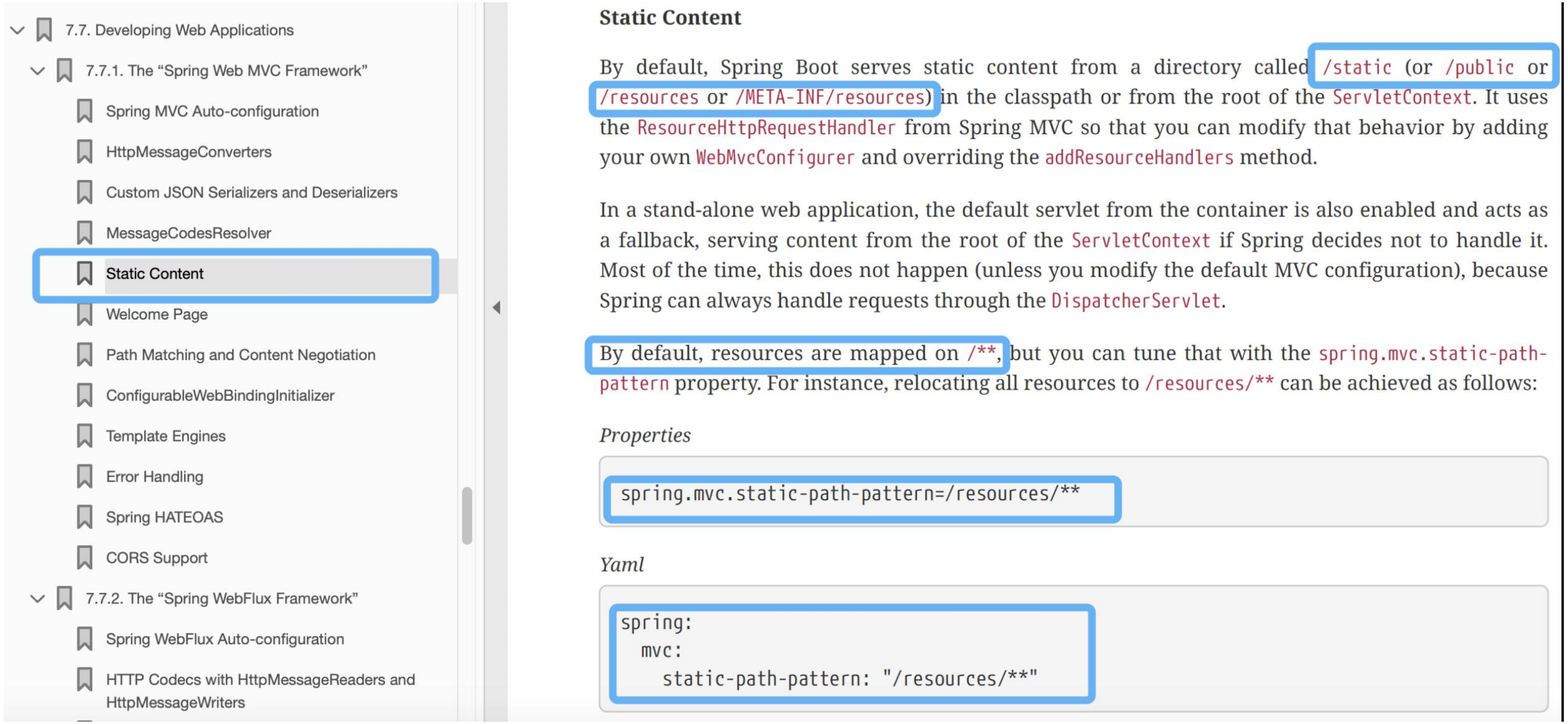
自动配置

- @SpringBootApplication 注解源码解析
- 自动配置特征介绍：
  - 自动配置按需加载原理
  - 容错兼容
  - 用户配置优先
  - 外部配置项修改组件行为
  - 查看自动配置情况

爪哇教育
Power Human With Education

SpringBoot WEB

静态资源访问



Static Content

By default, Spring Boot serves static content from a directory called /static (or /public or /resources or /META-INF/resources) in the classpath or from the root of the ServletContext. It uses the ResourceHttpRequestHandler from Spring MVC so that you can modify that behavior by adding your own WebMvcConfigurer and overriding the addResourceHandlers method.

In a stand-alone web application, the default servlet from the container is also enabled and acts as a fallback, serving content from the root of the ServletContext if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the DispatcherServlet.

By default, resources are mapped on /**, but you can tune that with the spring.mvc.static-path-pattern property. For instance, relocating all resources to /resources/** can be achieved as follows:

Properties

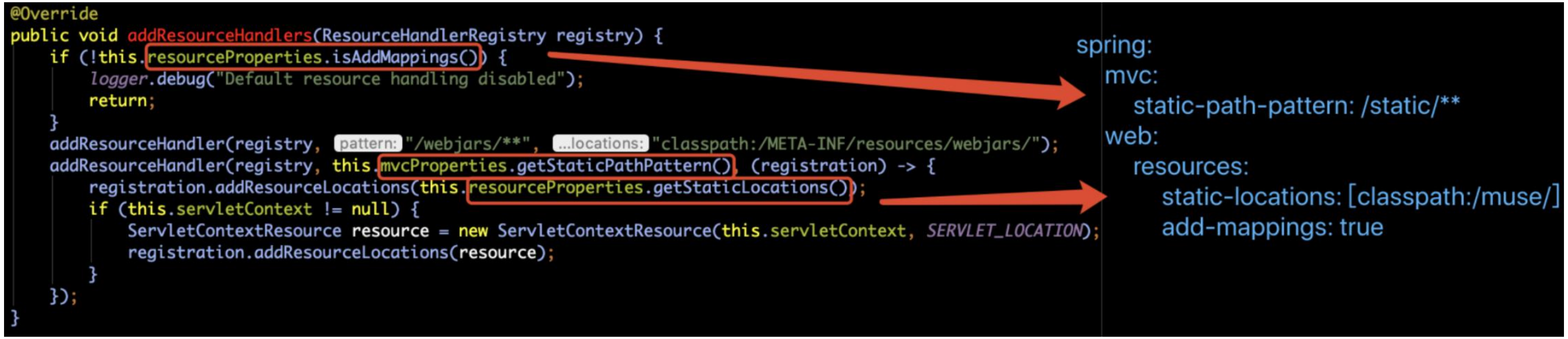
```
spring.mvc.static-path-pattern=/resources/**
```

Yaml

```
spring:
  mvc:
    static-path-pattern: "/resources/**"
```

静态资源配置原理解析

- ➢ 相关源码在WebMvcAutoConfiguration.addResourceHandlers(...)方法中



```java
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    if (!this.resourceProperties.isAddMappings()) {
        logger.debug("Default resource handling disabled");
        return;
    }
    addResourceHandler(registry, pattern: "/webjars/**", ...locations: "classpath:/META-INF/resources/webjars/");
    addResourceHandler(registry, this.mvcProperties.getStaticPathPattern(), (registration) -> {
        registration.addResourceLocations(this.resourceProperties.getStaticLocations());
        if (this.servletContext != null) {
            ServletContextResource resource = new ServletContextResource(this.servletContext, SERVLET_LOCATION);
            registration.addResourceLocations(resource);
        }
    });
}
```

```yaml
spring:
  mvc:
    static-path-pattern: /static/**
  web:
    resources:
      static-locations: [classpath:/muse/]
      add-mappings: true
```

- ➢ private String staticPathPattern = "/**";
- ➢ private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
- ➢ CLASSPATH_RESOURCE_LOCATIONS = {

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

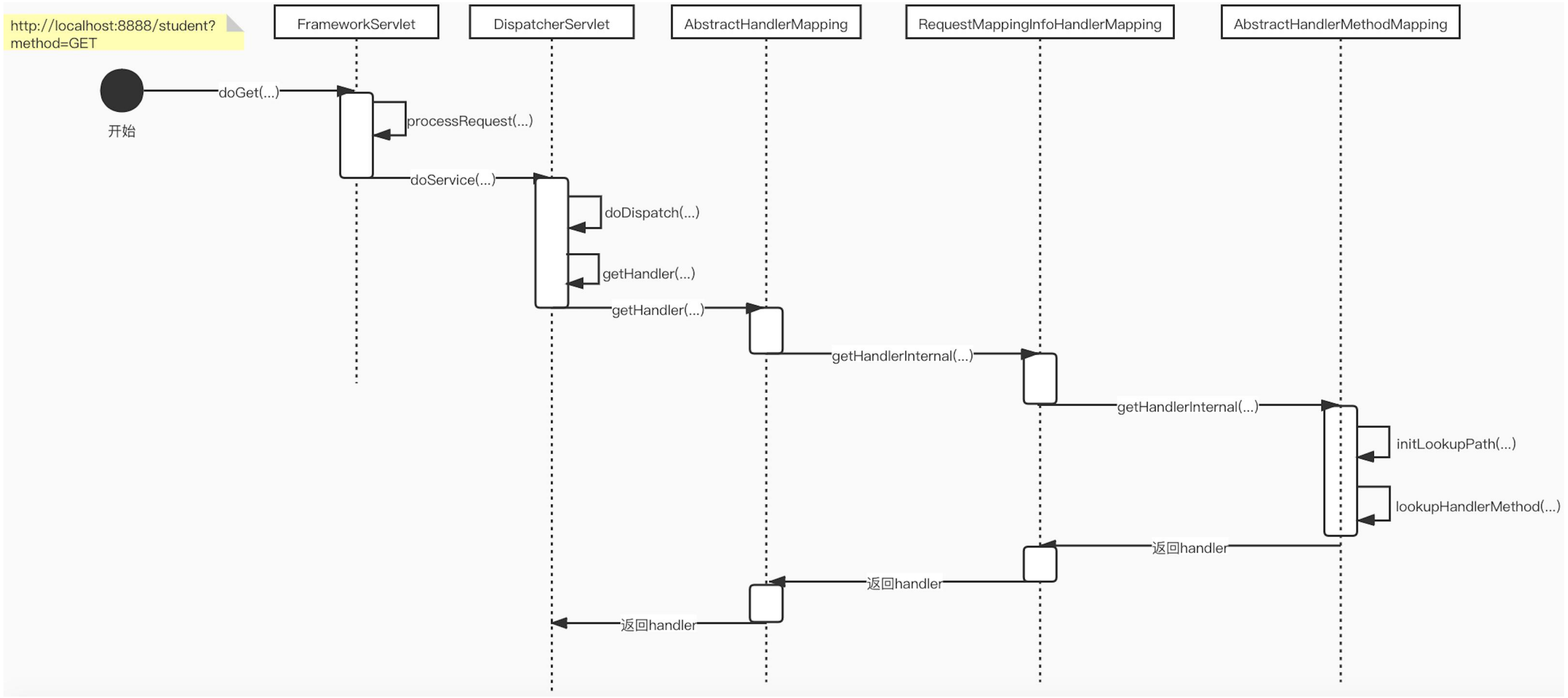
Rest风格请求映射

- 请求路径，采用@RequestMapping 或 @XxxMapping
- Rest风格支持（使用HTTP请求方式动词来表示对资源的操作）

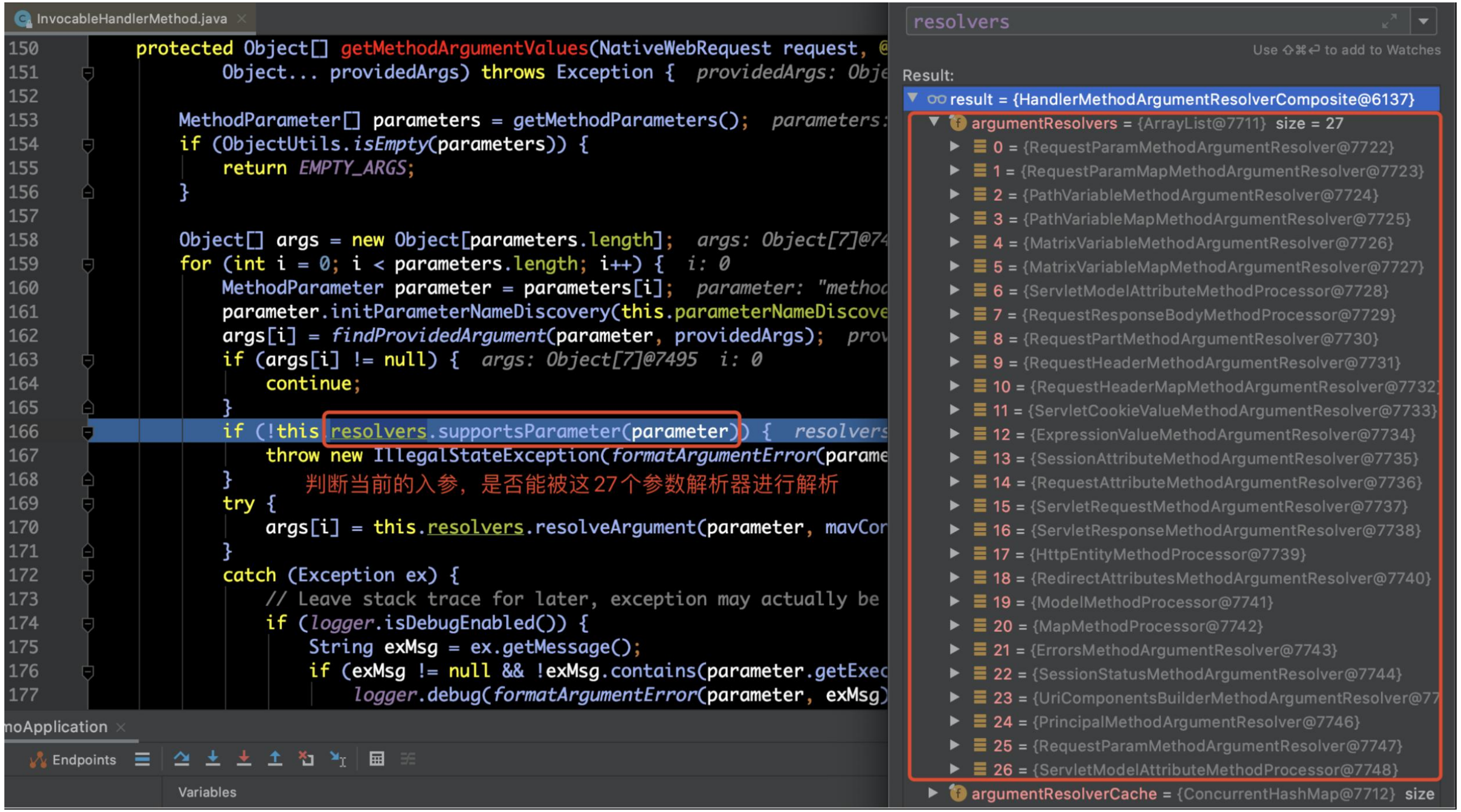
是否使用REST	获取用户信息	删除用户信息	更新用户信息	保存用户信息
否	/getUser	/deleteUser	/updateUser	/saveUser
是	/user method=**GET**	/user method=**DELETE**	/user method=**PUT**	/user method=**POST**

- 携带_method的表单提交
- 核心Filter——HiddenHttpMethodFilter源码解析

Spring MVC请求映射原理



注解作为参数的请求方式源码解析



InvocableHandlerMethod.java

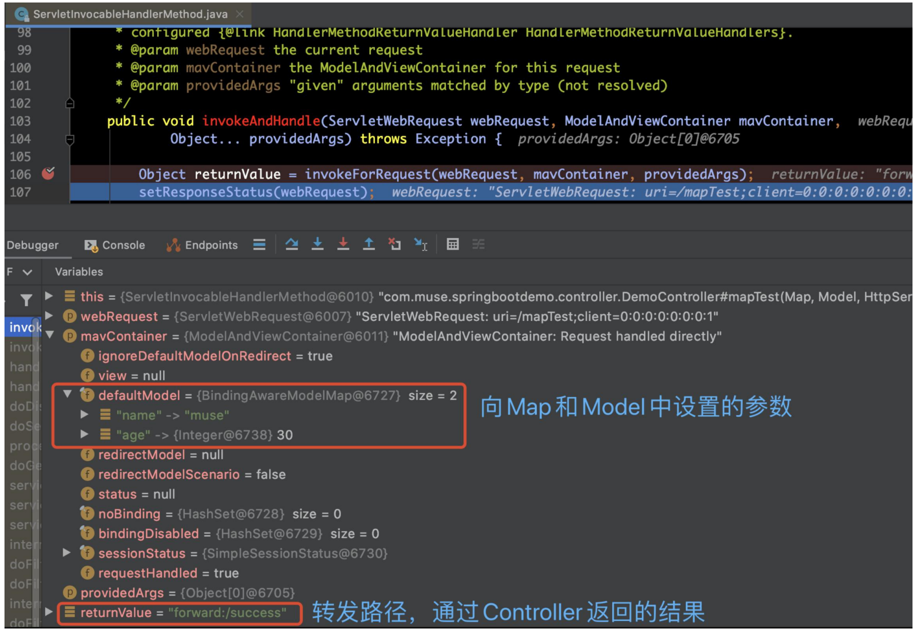
```
    protected Object[] getMethodArgumentValues(NativeWebRequest request, @
            Object... providedArgs) throws Exception {  providedArgs: Obje

        MethodParameter[] parameters = getMethodParameters();  parameters:
        if (ObjectUtils.isEmpty(parameters)) {
            return EMPTY_ARGS;
        }

        Object[] args = new Object[parameters.length];  args: Object[7]@74
        for (int i = 0; i < parameters.length; i++) {  i: 0
            MethodParameter parameter = parameters[i];  parameter: "methoc
            parameter.initParameterNameDiscovery(this.parameterNameDiscove
            args[i] = findProvidedArgument(parameter, providedArgs);  prov
            if (args[i] != null) {  args: Object[7]@7495  i: 0
                continue;
            }
            if (!this.resolvers.supportsParameter(parameter)) {  resolvers
                throw new IllegalStateException(formatArgumentError(parame
            }
            try {
                args[i] = this.resolvers.resolveArgument(parameter, mavCor
            }
            catch (Exception ex) {
                // Leave stack trace for later, exception may actually be
                if (logger.isDebugEnabled()) {
                    String exMsg = ex.getMessage();
                    if (exMsg != null && !exMsg.contains(parameter.getExec
                        logger.debug(formatArgumentError(parameter, exMsg)
```

判断当前的入参，是否能被这27个参数解析器进行解析

```
resolvers
Use ⇧⌘↩ to add to Watches
Result:
▼ oo result = {HandlerMethodArgumentResolverComposite@6137}
  ▼ f argumentResolvers = {ArrayList@7711}  size = 27
    ▶ 0 = {RequestParamMethodArgumentResolver@7722}
    ▶ 1 = {RequestParamMapMethodArgumentResolver@7723}
    ▶ 2 = {PathVariableMethodArgumentResolver@7724}
    ▶ 3 = {PathVariableMapMethodArgumentResolver@7725}
    ▶ 4 = {MatrixVariableMethodArgumentResolver@7726}
    ▶ 5 = {MatrixVariableMapMethodArgumentResolver@7727}
    ▶ 6 = {ServletModelAttributeMethodProcessor@7728}
    ▶ 7 = {RequestResponseBodyMethodProcessor@7729}
    ▶ 8 = {RequestPartMethodArgumentResolver@7730}
    ▶ 9 = {RequestHeaderMethodArgumentResolver@7731}
    ▶ 10 = {RequestHeaderMapMethodArgumentResolver@7732}
    ▶ 11 = {ServletCookieValueMethodArgumentResolver@7733}
    ▶ 12 = {ExpressionValueMethodArgumentResolver@7734}
    ▶ 13 = {SessionAttributeMethodArgumentResolver@7735}
    ▶ 14 = {RequestAttributeMethodArgumentResolver@7736}
    ▶ 15 = {ServletRequestMethodArgumentResolver@7737}
    ▶ 16 = {ServletResponseMethodArgumentResolver@7738}
    ▶ 17 = {HttpEntityMethodProcessor@7739}
    ▶ 18 = {RedirectAttributesMethodArgumentResolver@7740}
    ▶ 19 = {ModelMethodProcessor@7741}
    ▶ 20 = {MapMethodProcessor@7742}
    ▶ 21 = {ErrorsMethodArgumentResolver@7743}
    ▶ 22 = {SessionStatusMethodArgumentResolver@7744}
    ▶ 23 = {UriComponentsBuilderMethodArgumentResolver@77
    ▶ 24 = {PrincipalMethodArgumentResolver@7746}
    ▶ 25 = {RequestParamMethodArgumentResolver@7747}
    ▶ 26 = {ServletModelAttributeMethodProcessor@7748}
  ▶ f argumentResolverCache = {ConcurrentHashMap@7712}  size
```

Map或Model作为参数的请求方式源码解析

- Map入参由MapMethodProcessor进行解析，Model入参由ModelMethodProcessor进行解析。
- 目标方法执行完毕后，会将所有的数据都放在ModelAndViewContainer中，包含要跳转的页面地址view和model数据。



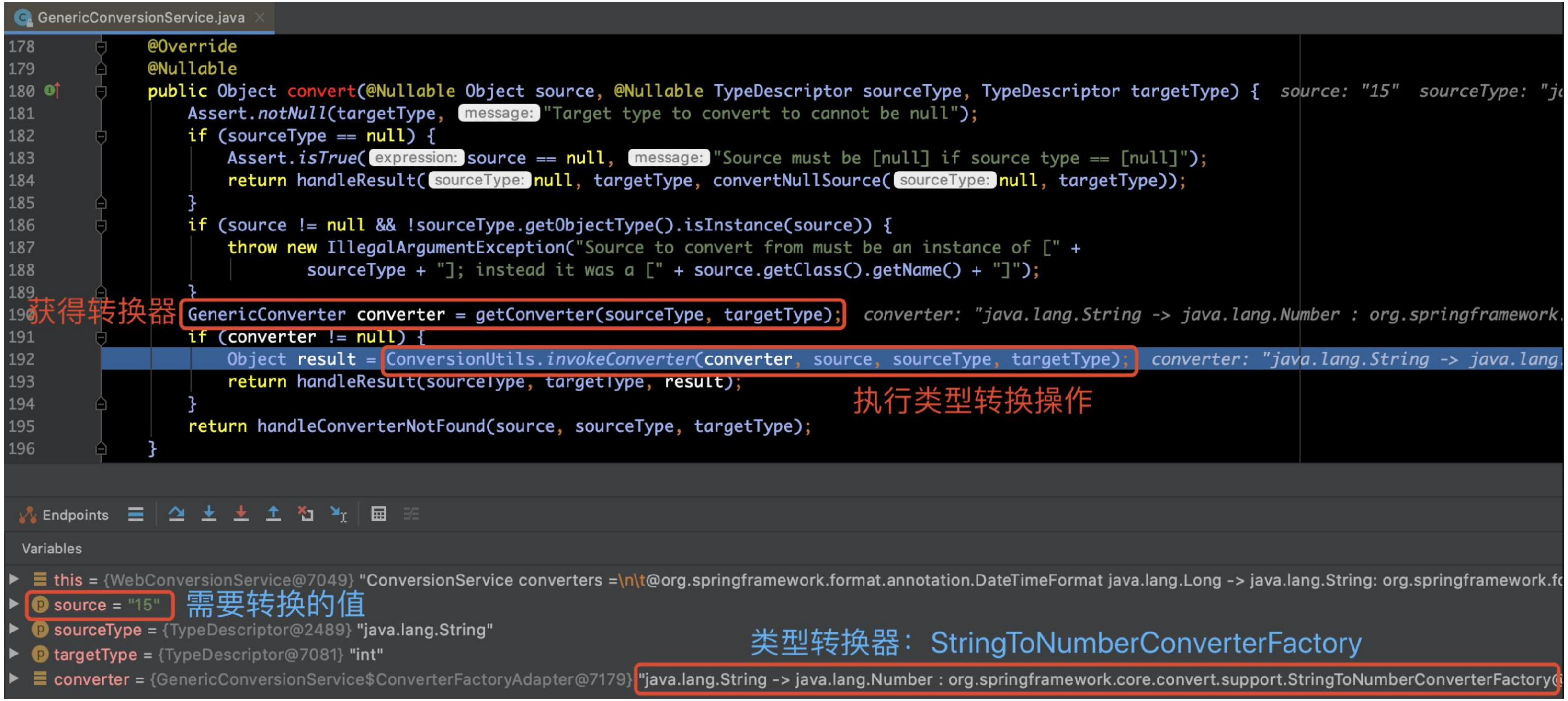
ServletInvocableHandlerMethod.java

```
	 * configured {@link HandlerMethodReturnValueHandler HandlerMethodReturnValueHandlers}.
	 * @param webRequest the current request
	 * @param mavContainer the ModelAndViewContainer for this request
	 * @param providedArgs "given" arguments matched by type (not resolved)
	 */
	public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,   webRequ
			Object... providedArgs) throws Exception {   providedArgs: Object[0]@6705

		Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);   returnValue: "forw
		setResponseStatus(webRequest);   webRequest: "ServletWebRequest: uri=/mapTest;client=0:0:0:0:0:0:
```

```
Variables
▶ this = {ServletInvocableHandlerMethod@6010} "com.muse.springbootdemo.controller.DemoController#mapTest(Map, Model, HttpSer
▶ webRequest = {ServletWebRequest@6007} "ServletWebRequest: uri=/mapTest;client=0:0:0:0:0:0:0:1"
▼ mavContainer = {ModelAndViewContainer@6011} "ModelAndViewContainer: Request handled directly"
    ignoreDefaultModelOnRedirect = true
    view = null
  ▼ defaultModel = {BindingAwareModelMap@6727}  size = 2
    ▶ "name" -> "muse"
    ▶ "age" -> {Integer@6738} 30
    redirectModel = null
    redirectModelScenario = false
    status = null
    noBinding = {HashSet@6728}  size = 0
    bindingDisabled = {HashSet@6729}  size = 0
  ▶ sessionStatus = {SimpleSessionStatus@6730}
    requestHandled = true
  providedArgs = {Object[0]@6705}
▶ returnValue = "forward:/success"
```

自定义Entity对象作为参数的请求方式源码解析

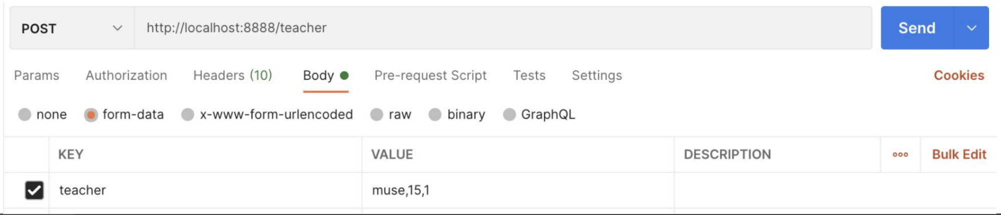
➢ 自定义类型参数是由ServletModelAttributeMethodProcessor进行解析的


GenericConversionService.java
获得转换器
执行类型转换操作
需要转换的值
类型转换器：StringToNumberConverterFactory
Endpoints
Variables

```
    @Override
    @Nullable
    public Object convert(@Nullable Object source, @Nullable TypeDescriptor sourceType, TypeDescriptor targetType) {   source: "15"   sourceType: "j
        Assert.notNull(targetType, message: "Target type to convert to cannot be null");
        if (sourceType == null) {
            Assert.isTrue( expression: source == null,  message: "Source must be [null] if source type == [null]");
            return handleResult( sourceType: null, targetType, convertNullSource( sourceType: null, targetType));
        }
        if (source != null && !sourceType.getObjectType().isInstance(source)) {
            throw new IllegalArgumentException("Source to convert from must be an instance of [" +
                    sourceType + "]; instead it was a [" + source.getClass().getName() + "]");
        }
        GenericConverter converter = getConverter(sourceType, targetType);   converter: "java.lang.String -> java.lang.Number : org.springframework.
        if (converter != null) {
            Object result = ConversionUtils.invokeConverter(converter, source, sourceType, targetType);   converter: "java.lang.String -> java.lang.
            return handleResult(sourceType, targetType, result);
        }
        return handleConverterNotFound(source, sourceType, targetType);
    }
```

```
▸ this = {WebConversionService@7049} "ConversionService converters =\n\t@org.springframework.format.annotation.DateTimeFormat java.lang.Long -> java.lang.String: org.springframework.fo
▸ source = "15"
▸ sourceType = {TypeDescriptor@2489} "java.lang.String"
▸ targetType = {TypeDescriptor@7081} "int"
▸ converter = {GenericConversionService$ConverterFactoryAdapter@7179} "java.lang.String -> java.lang.Number : org.springframework.core.convert.support.StringToNumberConverterFactory@
```

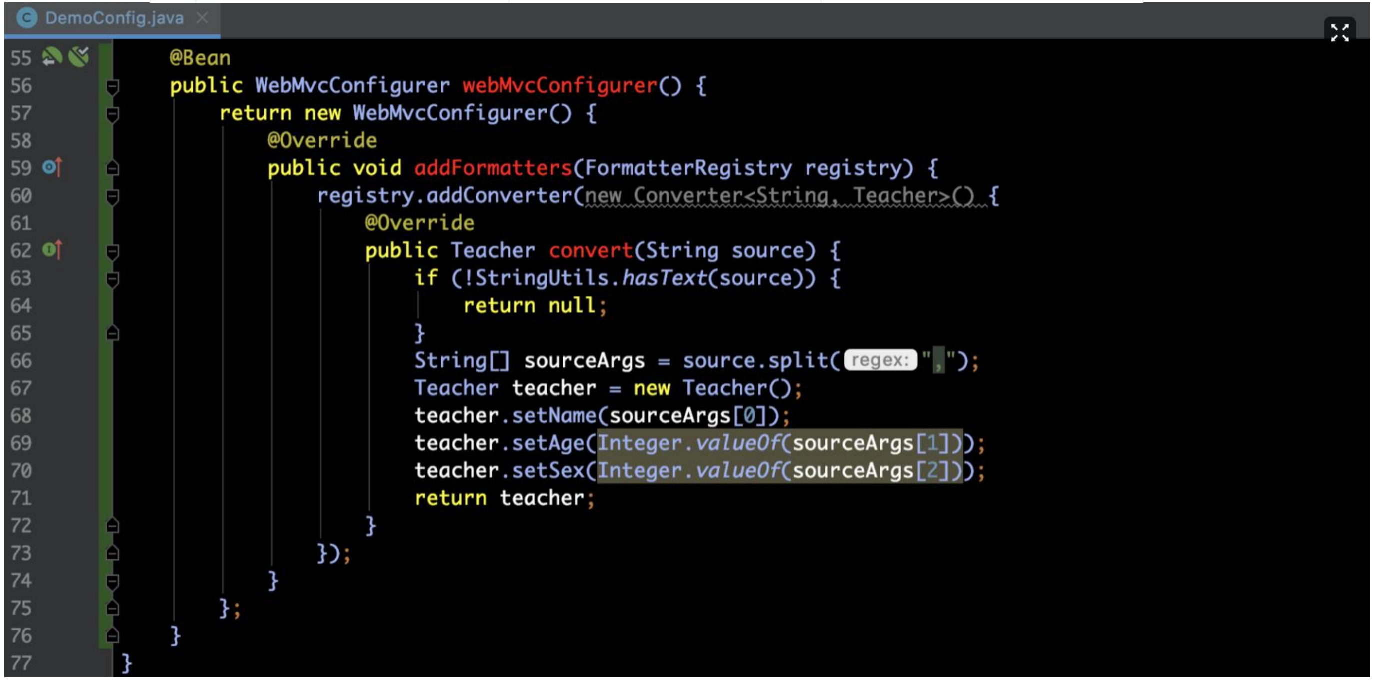
提供自定义入参的Converter实现



POST http://localhost:8888/teacher

Body: form-data

KEY	VALUE	DESCRIPTION
✓ teacher	muse,15,1	

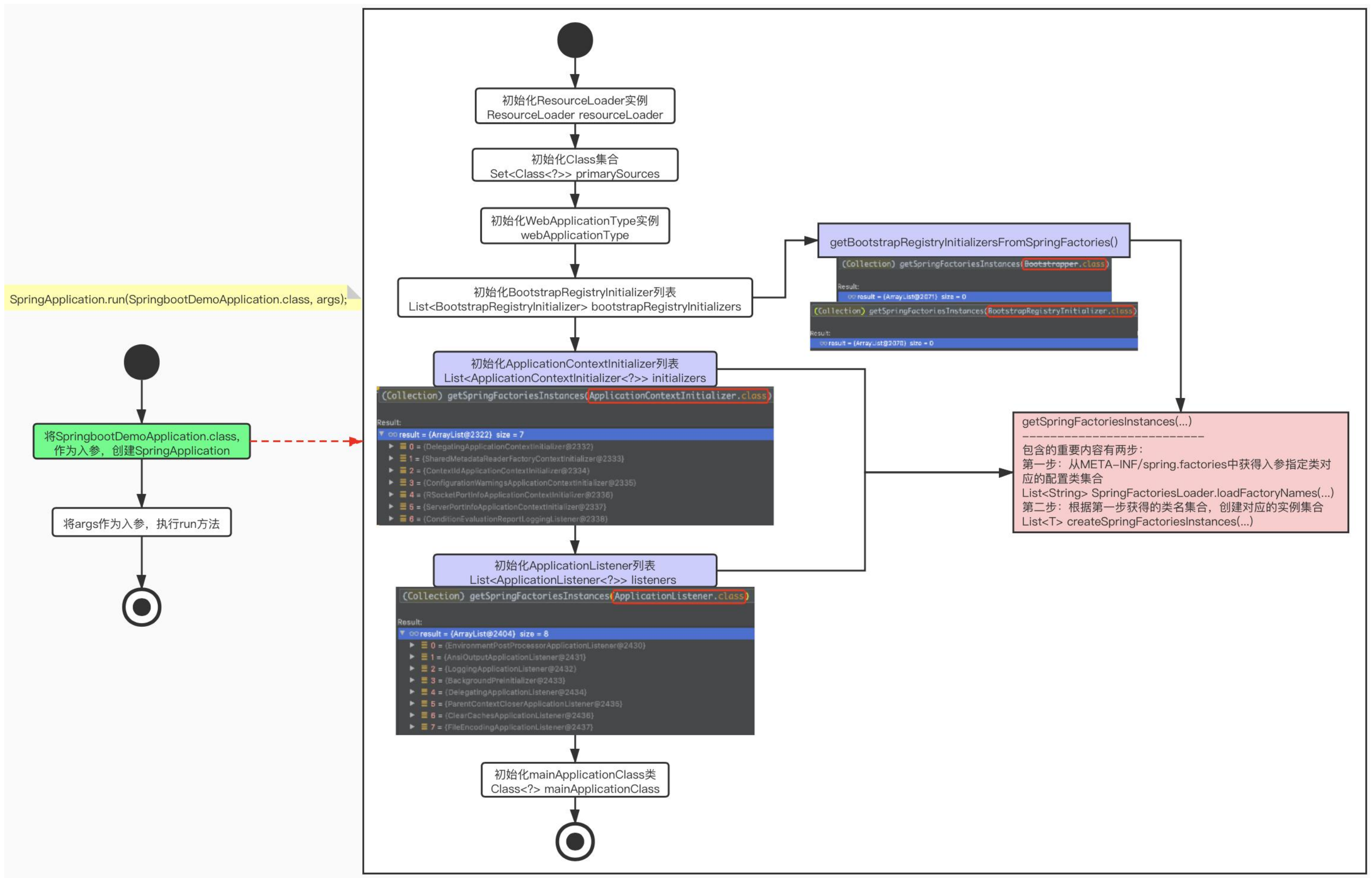


DemoConfig.java

```java
    @Bean
    public WebMvcConfigurer webMvcConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addFormatters(FormatterRegistry registry) {
                registry.addConverter(new Converter<String, Teacher>() {
                    @Override
                    public Teacher convert(String source) {
                        if (!StringUtils.hasText(source)) {
                            return null;
                        }
                        String[] sourceArgs = source.split( regex: ",");
                        Teacher teacher = new Teacher();
                        teacher.setName(sourceArgs[0]);
                        teacher.setAge(Integer.valueOf(sourceArgs[1]));
                        teacher.setSex(Integer.valueOf(sourceArgs[2]));
                        return teacher;
                    }
                });
            }
        };
    }
}
```

SpringBoot启动流程

构建SpringApplication流程图



SpringApplication.run(SpringbootDemoApplication.class, args);
将SpringbootDemoApplication.class，作为入参，创建SpringApplication
将args作为入参，执行run方法
初始化ResourceLoader实例
ResourceLoader resourceLoader
初始化Class集合
Set<Class<?>> primarySources
初始化WebApplicationType实例
webApplicationType
初始化BootstrapRegistryInitializer列表
List<BootstrapRegistryInitializer> bootstrapRegistryInitializers
getBootstrapRegistryInitializersFromSpringFactories()
初始化ApplicationContextInitializer列表
List<ApplicationContextInitializer<?>> initializers
初始化ApplicationListener列表
List<ApplicationListener<?>> listeners
初始化mainApplicationClass类
Class<?> mainApplicationClass

(Collection) getSpringFactoriesInstances(~~Bootstrapper~~.class)

```
Result:
  result = {ArrayList@2071}  size = 0
```

```
(Collection) getSpringFactoriesInstances(BootstrapRegistryInitializer.class)

Result:
  result = {ArrayList@2070}  size = 0
```

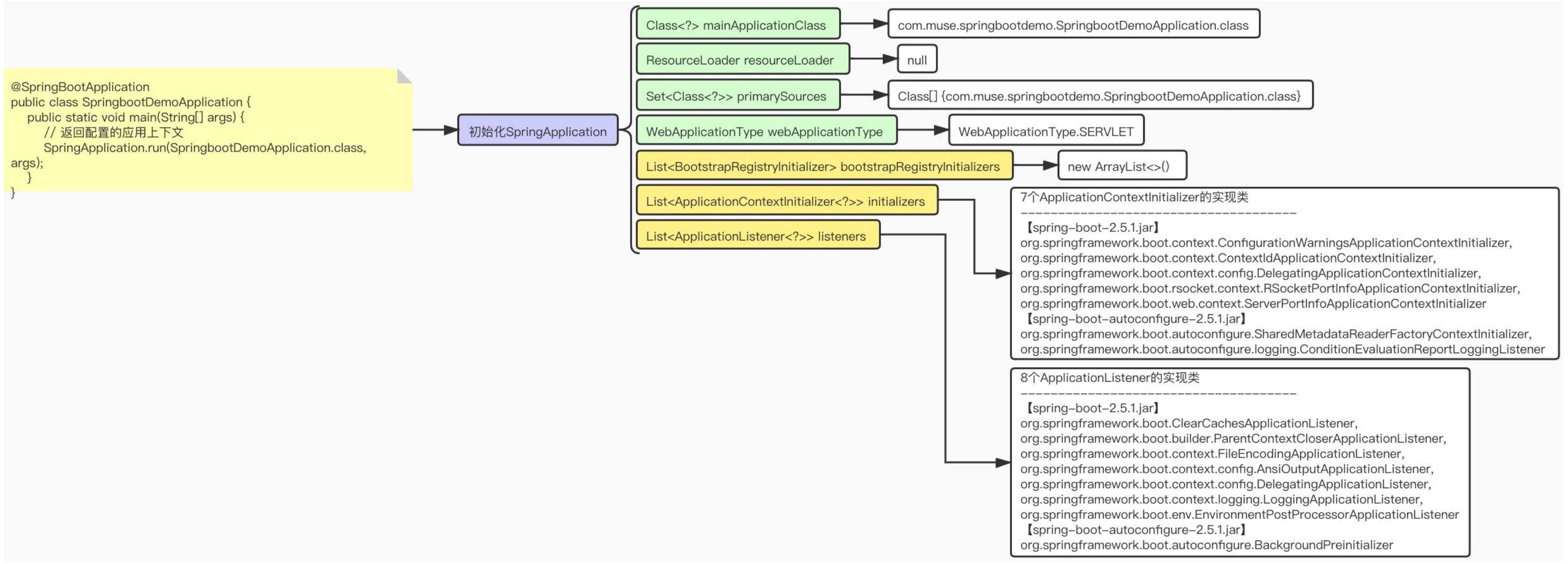
```
(Collection) getSpringFactoriesInstances(ApplicationContextInitializer.class)

Result:
▼ result = {ArrayList@2322}  size = 7
   ▶ 0 = {DelegatingApplicationContextInitializer@2332}
   ▶ 1 = {SharedMetadataReaderFactoryContextInitializer@2333}
   ▶ 2 = {ContextIdApplicationContextInitializer@2334}
   ▶ 3 = {ConfigurationWarningsApplicationContextInitializer@2335}
   ▶ 4 = {RSocketPortInfoApplicationContextInitializer@2336}
   ▶ 5 = {ServerPortInfoApplicationContextInitializer@2337}
   ▶ 6 = {ConditionEvaluationReportLoggingListener@2338}
```

```
(Collection) getSpringFactoriesInstances(ApplicationListener.class)

Result:
▼ result = {ArrayList@2404}  size = 8
   ▶ 0 = {EnvironmentPostProcessorApplicationListener@2430}
   ▶ 1 = {AnsiOutputApplicationListener@2431}
   ▶ 2 = {LoggingApplicationListener@2432}
   ▶ 3 = {BackgroundPreinitializer@2433}
   ▶ 4 = {DelegatingApplicationListener@2434}
   ▶ 5 = {ParentContextCloserApplicationListener@2435}
   ▶ 6 = {ClearCachesApplicationListener@2436}
   ▶ 7 = {FileEncodingApplicationListener@2437}
```

getSpringFactoriesInstances(...)

包含的重要内容有两步：
第一步：从META-INF/spring.factories中获得入参指定类对应的配置类集合
List<String> SpringFactoriesLoader.loadFactoryNames(...)
第二步：根据第一步获得的类名集合，创建对应的实例集合
List<T> createSpringFactoriesInstances(...)

关键属性赋值



```
@SpringBootApplication
public class SpringbootDemoApplication {
    public static void main(String[] args) {
        // 返回配置的应用上下文
        SpringApplication.run(SpringbootDemoApplication.class,
args);
    }
}
```

初始化SpringApplication
Class<?> mainApplicationClass
com.muse.springbootdemo.SpringbootDemoApplication.class
ResourceLoader resourceLoader
null
Set<Class<?>> primarySources
Class[] {com.muse.springbootdemo.SpringbootDemoApplication.class}
WebApplicationType webApplicationType
WebApplicationType.SERVLET
List<BootstrapRegistryInitializer> bootstrapRegistryInitializers
new ArrayList<>()
List<ApplicationContextInitializer<?>> initializers
List<ApplicationListener<?>> listeners

7个ApplicationContextInitializer的实现类

【spring-boot-2.5.1.jar】
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,
org.springframework.boot.context.ContextIdApplicationContextInitializer,
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer,
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer
【spring-boot-autoconfigure-2.5.1.jar】
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

8个ApplicationListener的实现类

【spring-boot-2.5.1.jar】
org.springframework.boot.ClearCachesApplicationListener,
org.springframework.boot.builder.ParentContextCloserApplicationListener,
org.springframework.boot.context.FileEncodingApplicationListener,
org.springframework.boot.context.config.AnsiOutputApplicationListener,
org.springframework.boot.context.config.DelegatingApplicationListener,
org.springframework.boot.context.logging.LoggingApplicationListener,
org.springframework.boot.env.EnvironmentPostProcessorApplicationListener
【spring-boot-autoconfigure-2.5.1.jar】
org.springframework.boot.autoconfigure.BackgroundPreinitializer

结束