# 1．云平台微服务架构

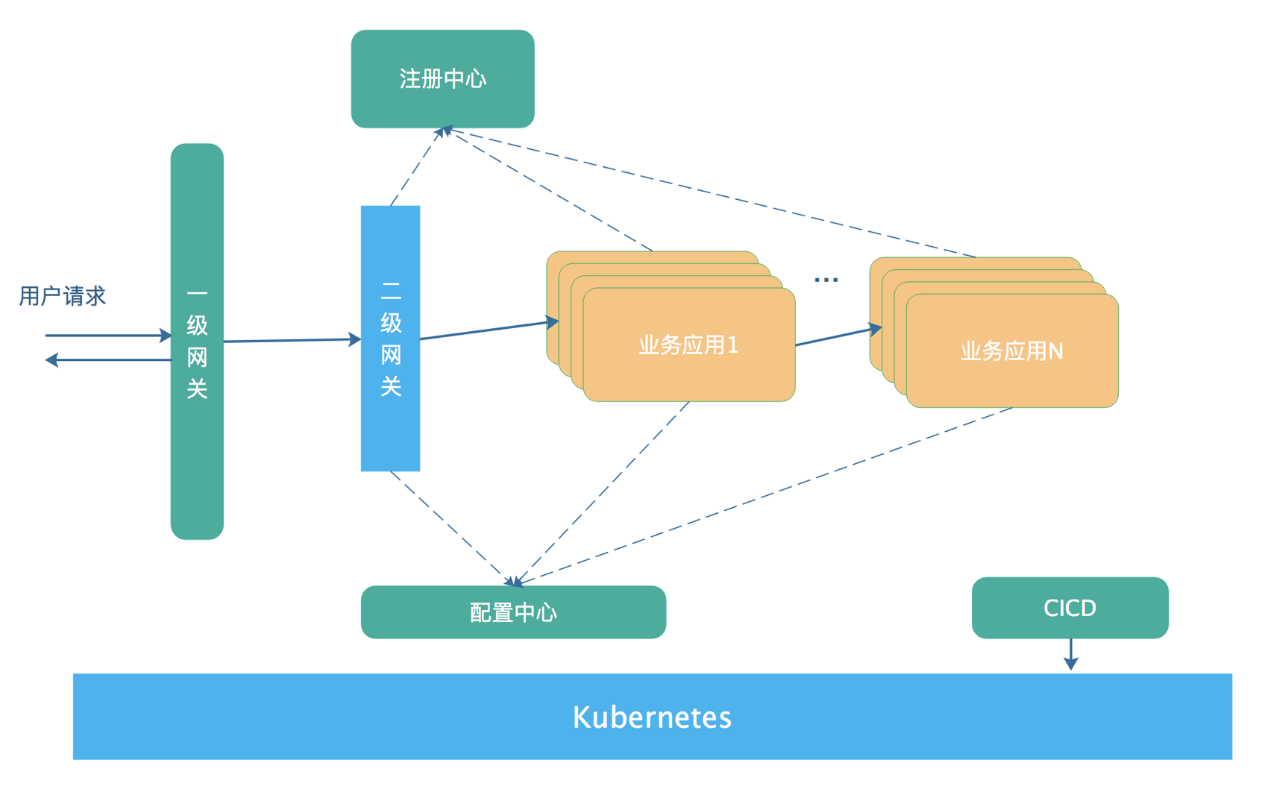


图1 云平台业务应用架构

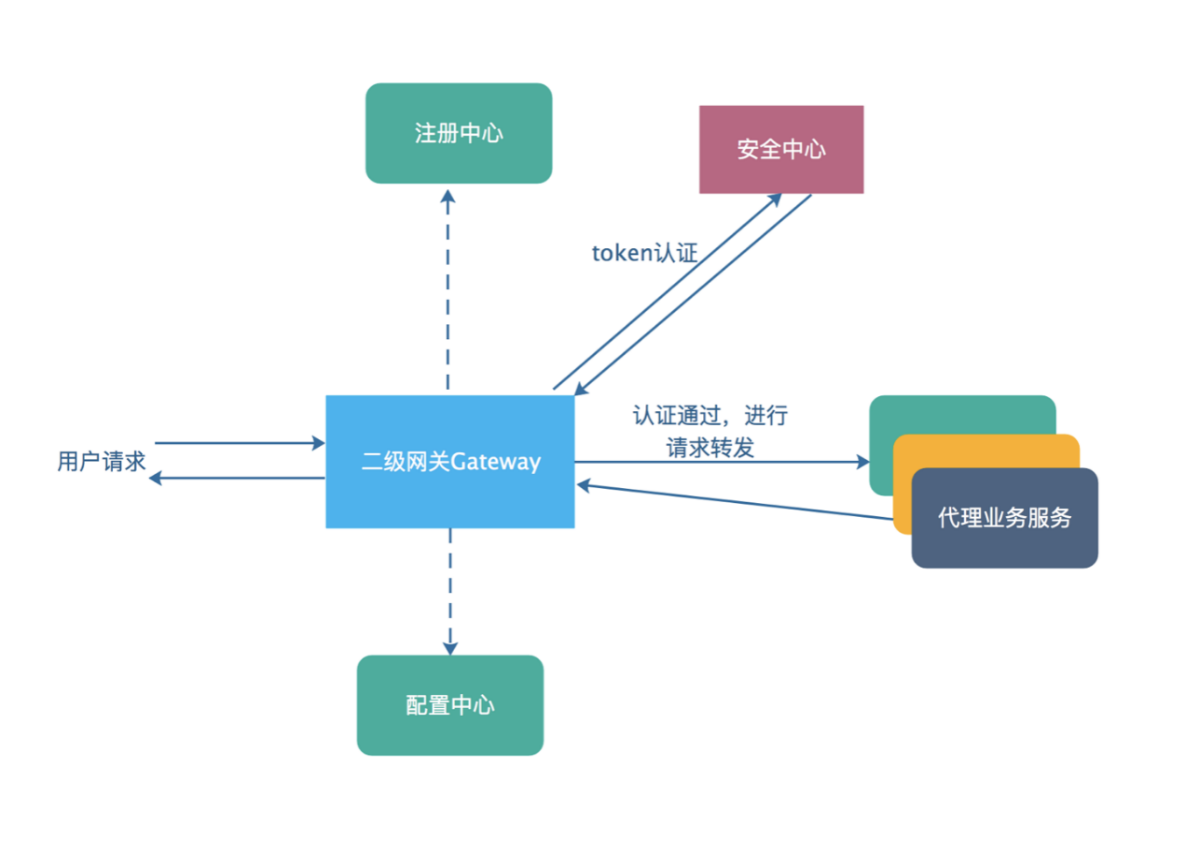


图2 平台二级网关架构

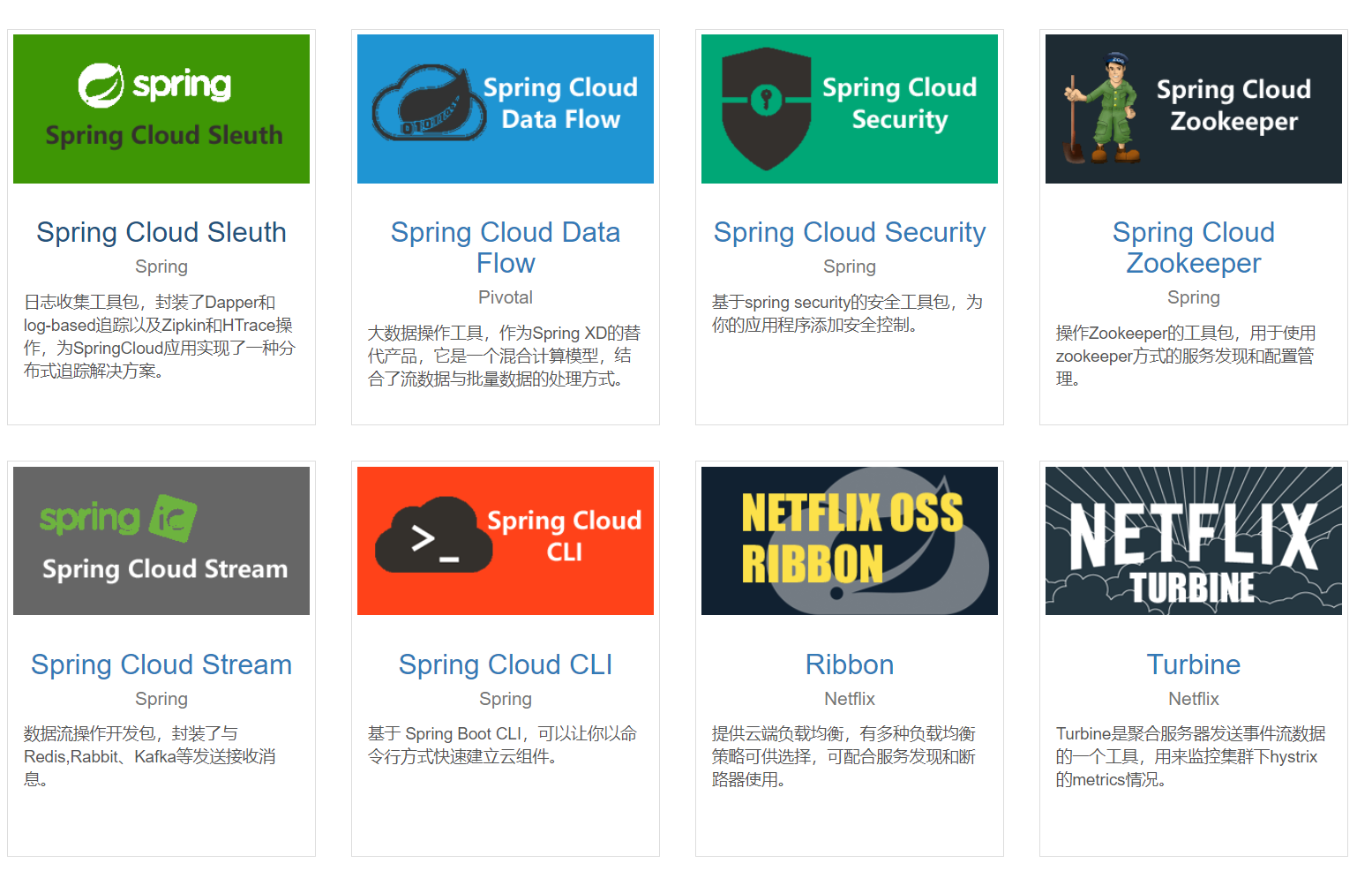
# 2．SpringCloud介绍

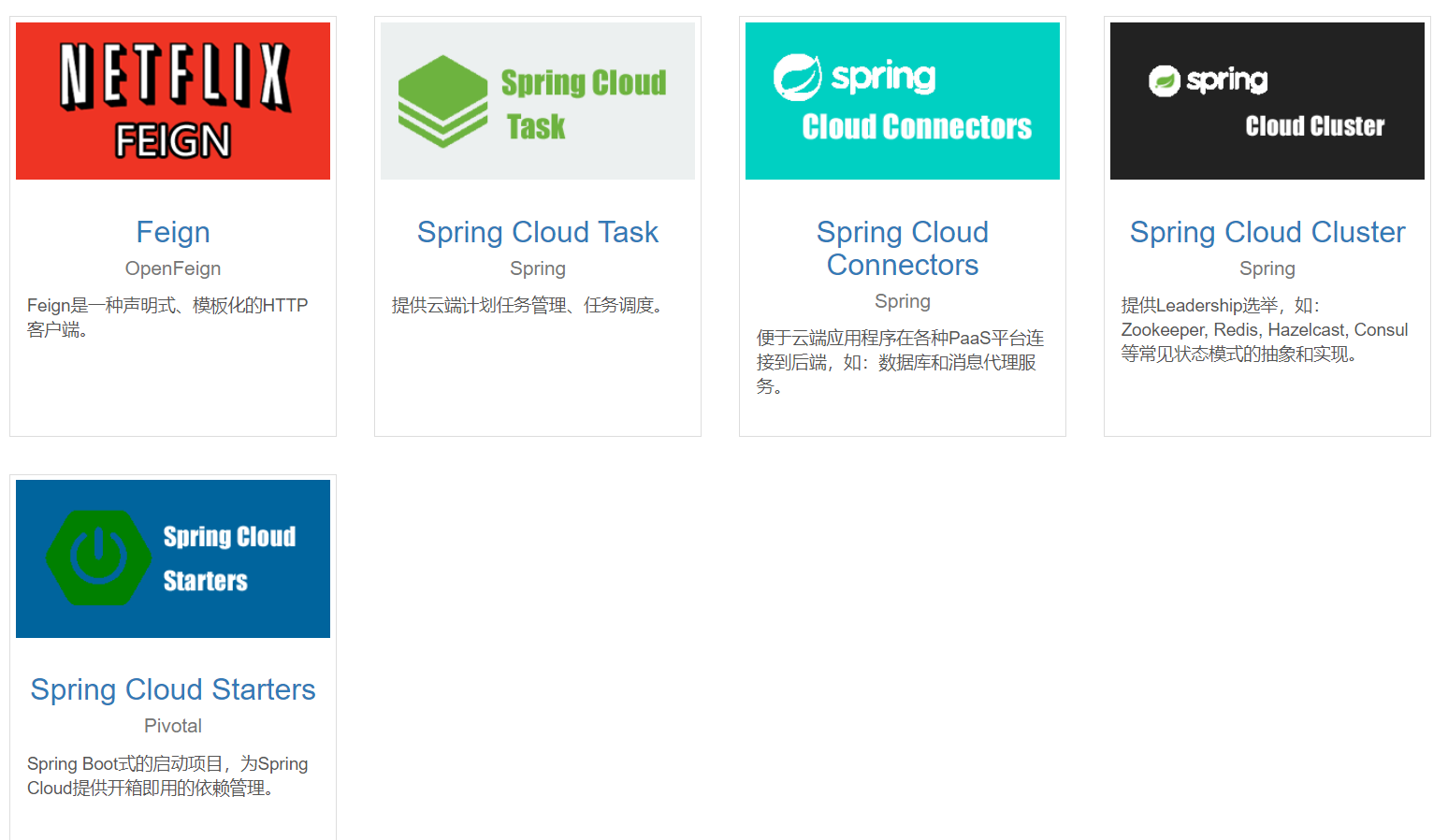
## 2.1 SpringCloud简介

Spring Cloud是一个基于Springboot实现的微服务架构开发工具，为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

SpringCloud包含了很多子项目，可去官方文档（<https://spring.io/projects/spring-cloud>）中查看，如下图所示；







## 2.2 SpringCloud与Springboot

### 2.2.1 SpringCloud与Springboot的关系

Spring boot 是 Spring 的一套快速配置脚手架，可以基于spring boot 快速开发单个微服务，Spring Boot，看名字就知道是Spring的引导，就是**用于启动Spring的**，**使得Spring的学习和使用变得快速简单**。不仅适合替换原有的工程结构，**更适合微服务开发**。

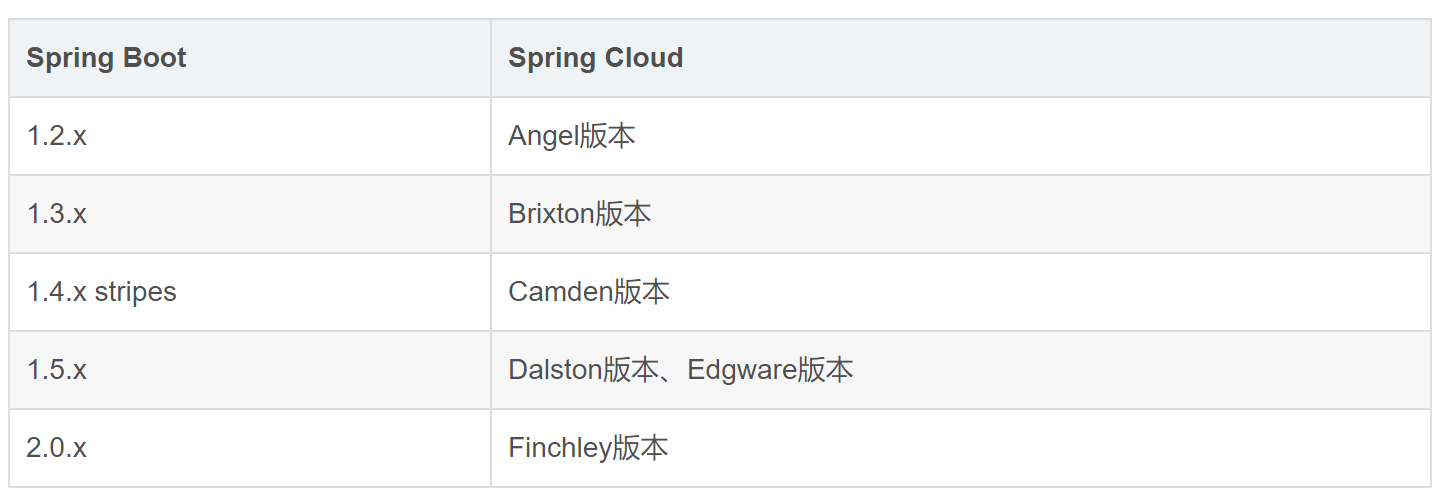
Spring Cloud是一个基于Spring Boot实现的云应用开发工具；**Spring boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；**spring boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring boot来实现。

Spring Boot的哲学就是约定大于配置。既然很多东西都是一样的，为什么还要去配置。

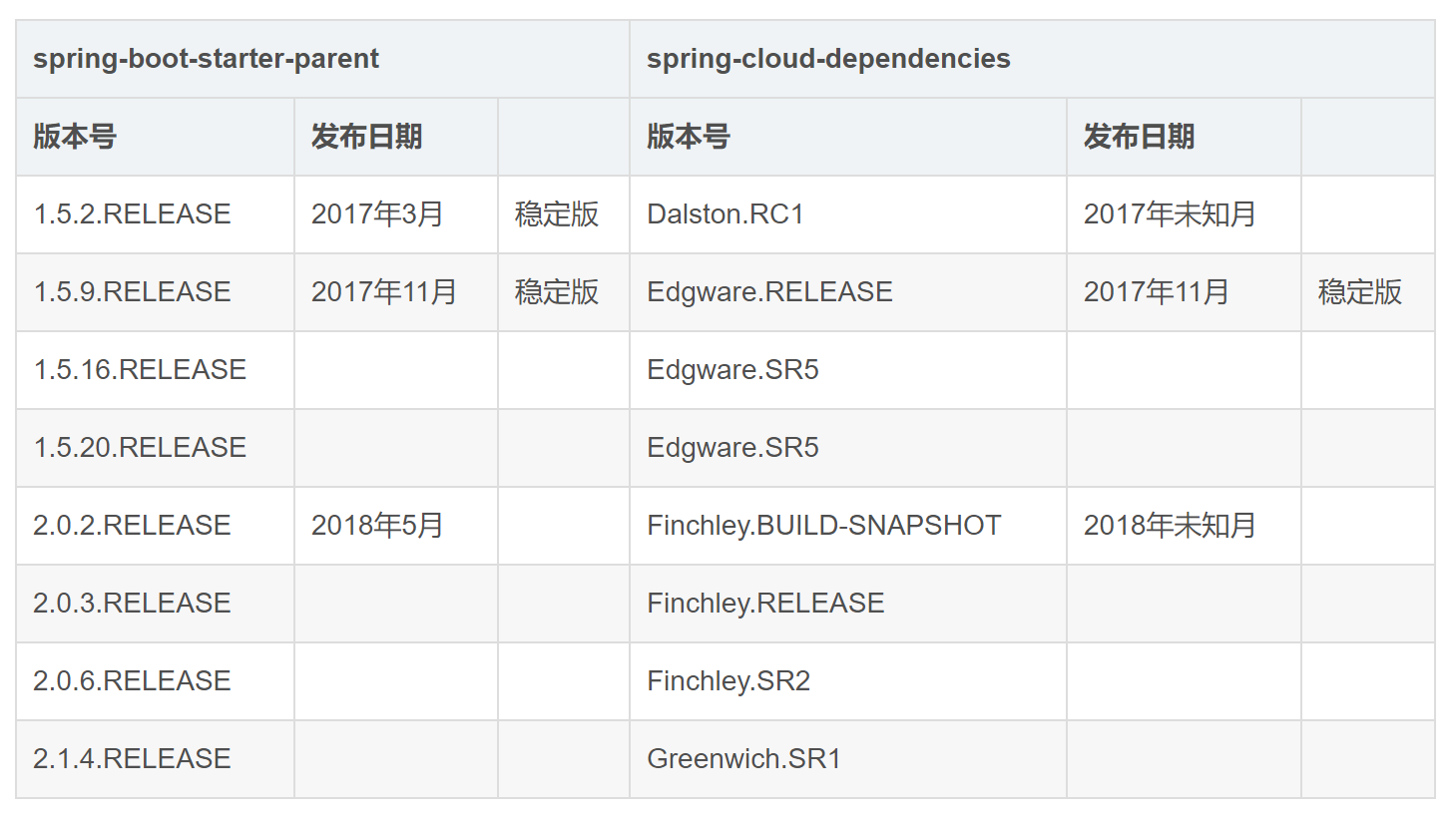
1. 通过starter和依赖管理解决依赖问题。  
2. 通过自动配置，解决配置复杂问题。  
3. 通过内嵌web容器，由应用启动tomcat，而不是tomcat启动应用，来解决部署运行问题。

### 2.2.2 SpringCloud与Springboot版本对应

大版本对应：



开发时候详细的版本对应关系：



**spring-cloud-dependencies**版本列表可查看：

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dependencies>

**spring-boot-starter-parent**版本列表可查看：

<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-parent>

### 2.2.3 Springboot介绍

#### 2.2.3.1 特点

Spring Boot的三大特点：

**1.自动配置**

Spring Boot自动配置，就是程序需要什么装配什么。比如当程序的pom文件引入了Feign的starter依赖，Spring Boot就会在程序中自动引入默认的Feign的配置Bean。

**2.起步依赖**

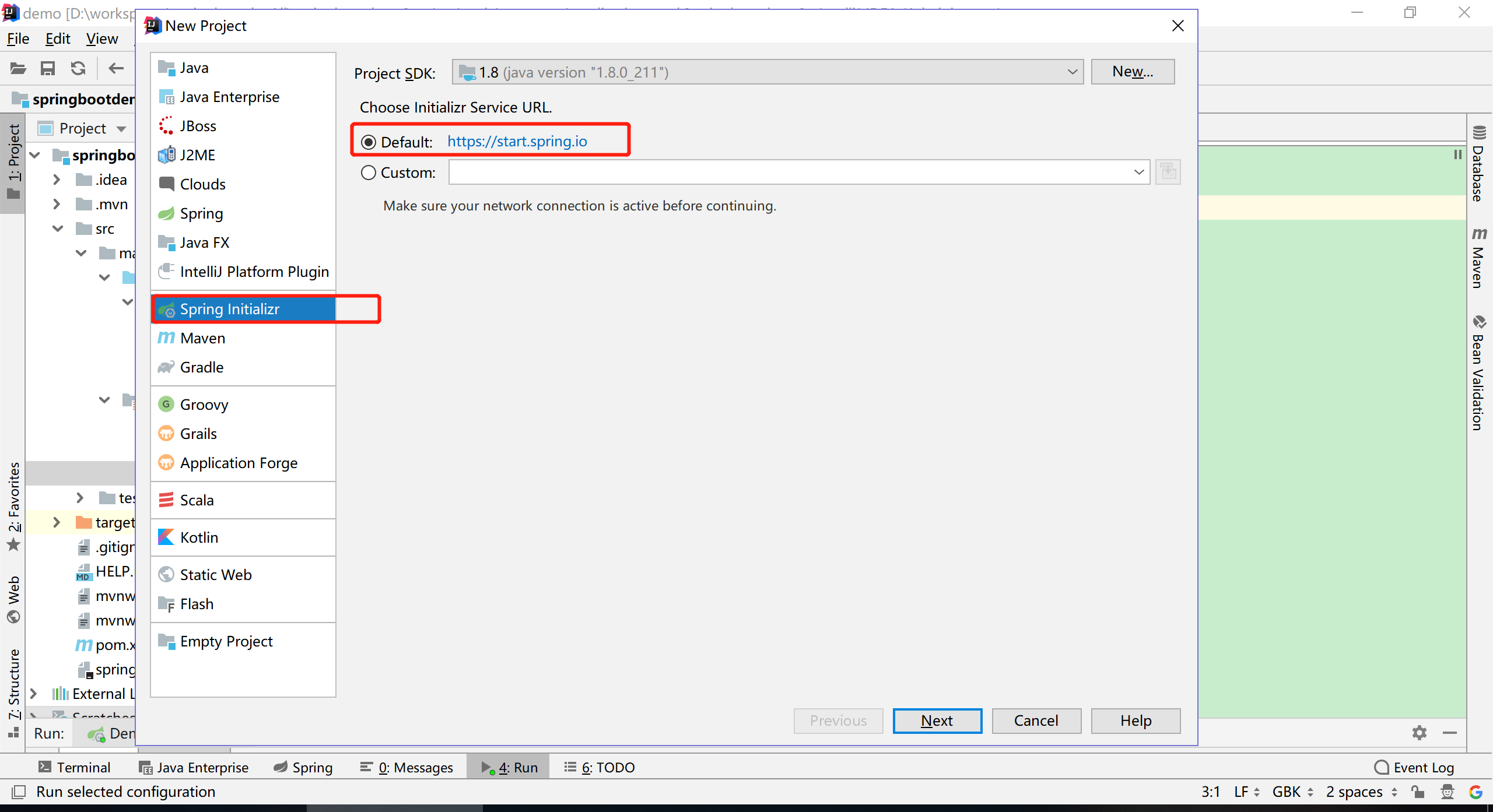
传统的开发过程中，向项目添加依赖常常会选择版本，解决版本冲突，十分耗费精力。而使用Spring Boot的起步依赖，只需要加入Spring-boot-starter-web的依赖，便会自动引入Spring MVC功能的相关依赖。

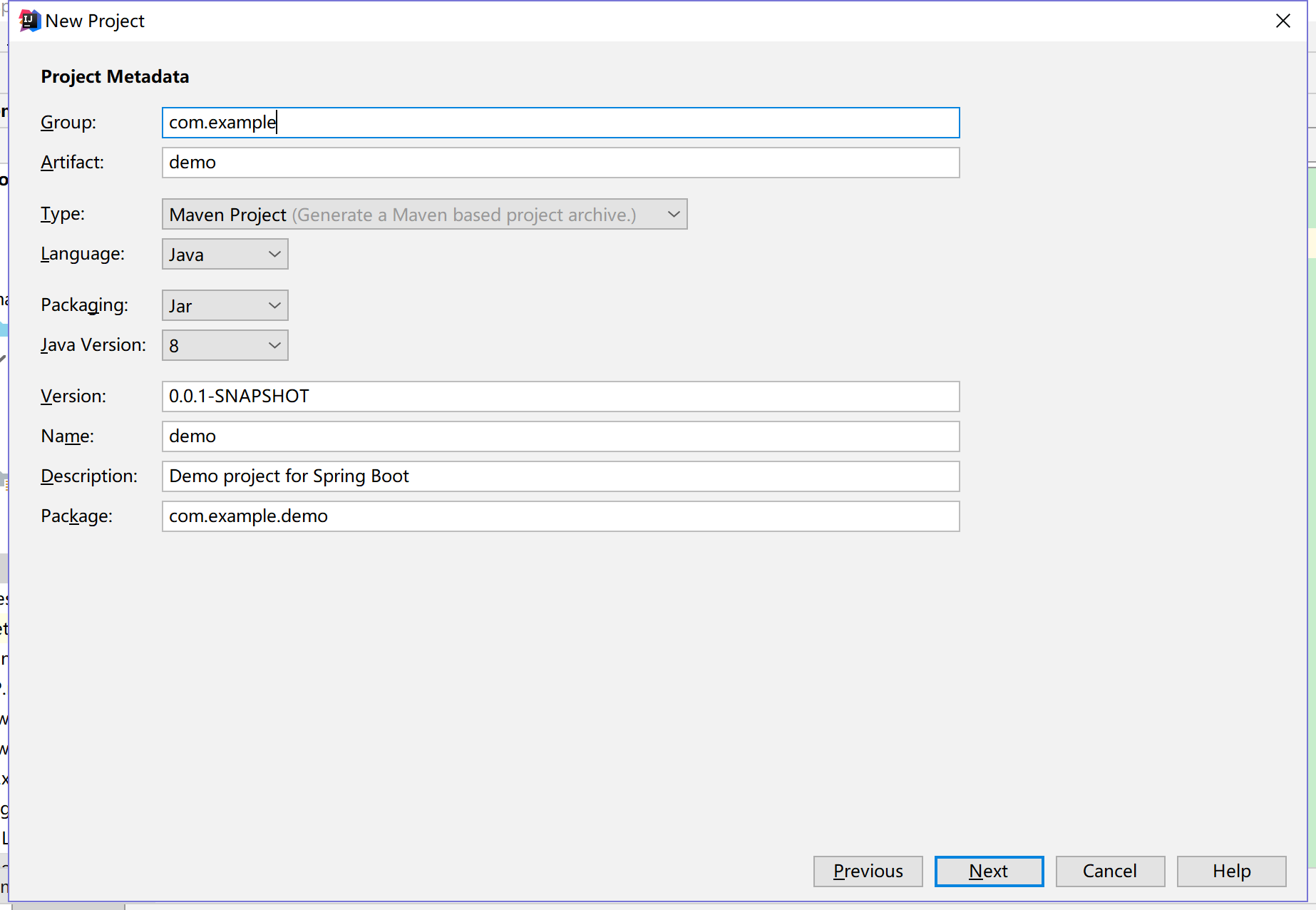
**3.Actuator对运行状态监控**

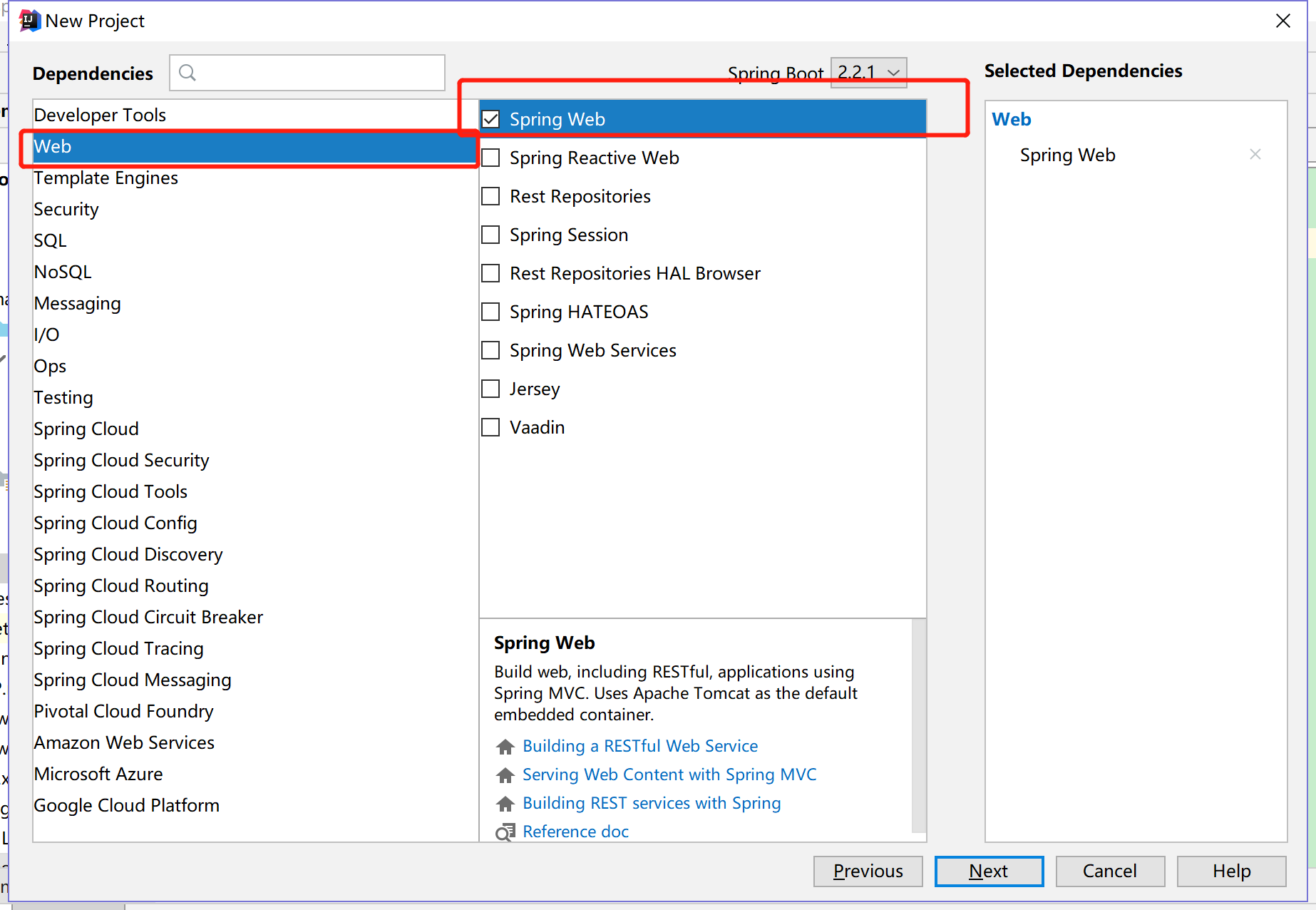
Spring Boot提供了Actuator组件，并提供了程序运行状态的监控功能，这种设计更贴心，让开发人员能够知道注入了哪些Bean，它们的运行状态如何。

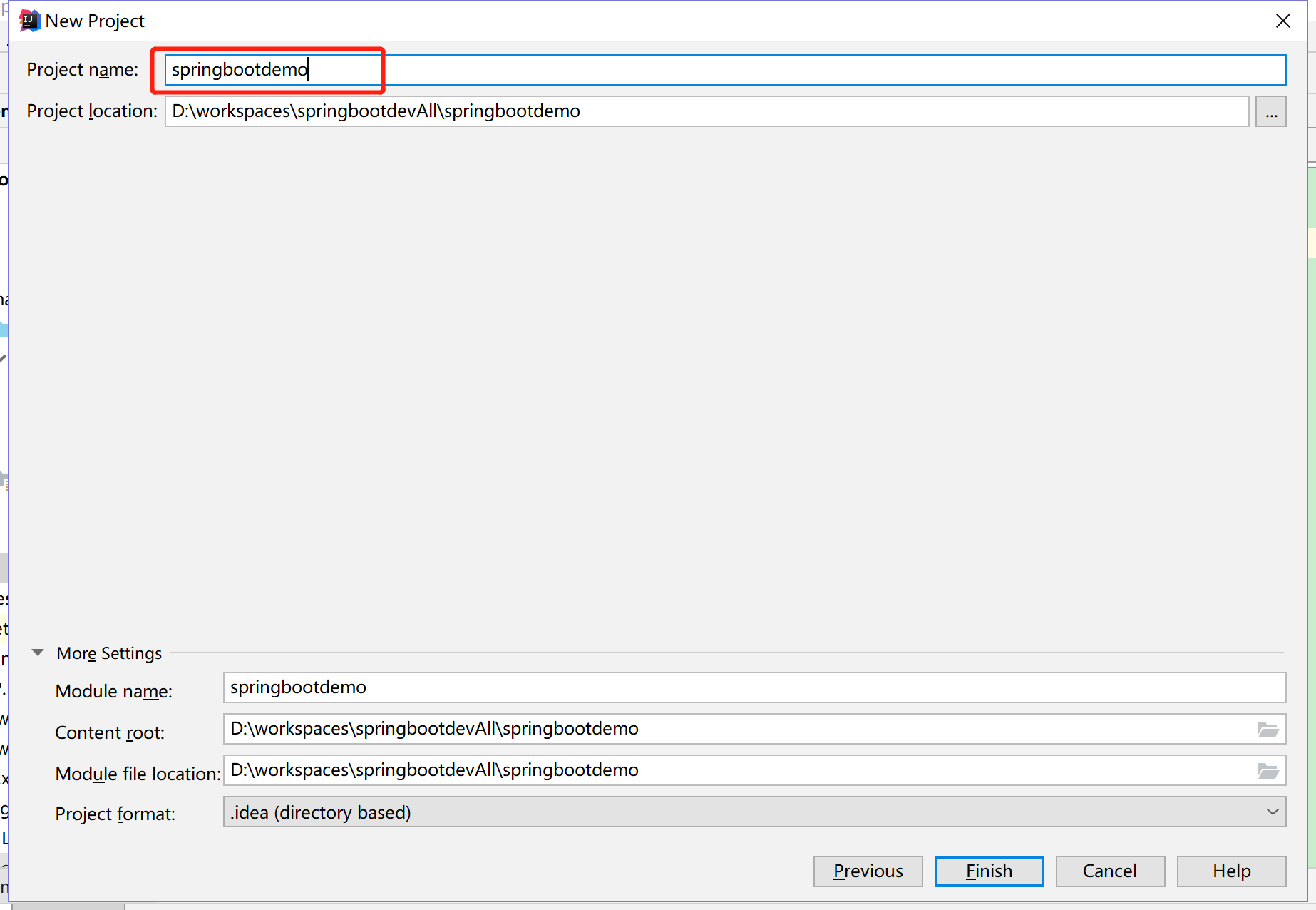
#### 2.2.3.2 工程示例

这里通过idea环境来创建springboot工程，通过以下几步就可简单创建；









经过以上步骤即可自动生成一个springboot的工程；

Springboot工程目录结构如下：

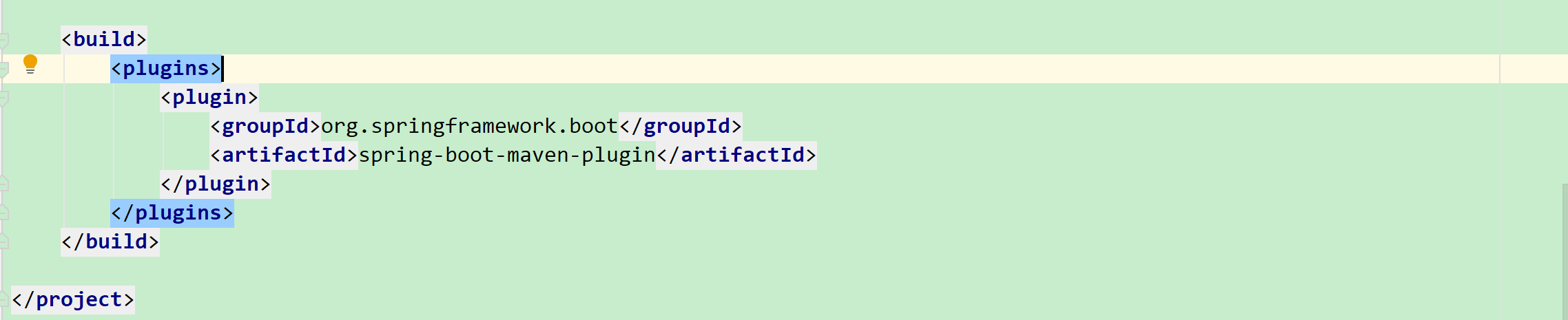


* src/main/java：目录下放置所有java文件（源代码文件）；
* src/main/resources：放置所有的配置文件、页面文件、静态资源文件；
* src/main/resources/static：是静态资源文件目录，在这个目录中的所有文件将可以被直接访问，如果没有这个文件夹可自行创建；
* src/main/resources/public：作用和src/main/resources/static目录一样。
* pom.xml:依赖管理文件

pom文件如下：



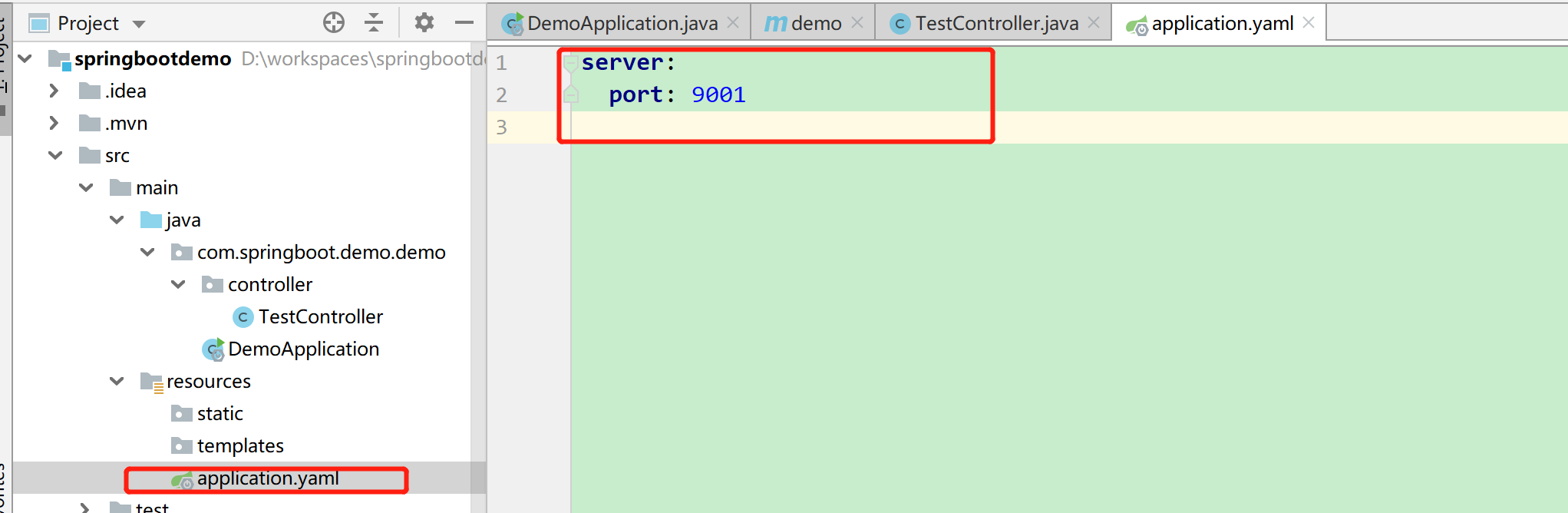




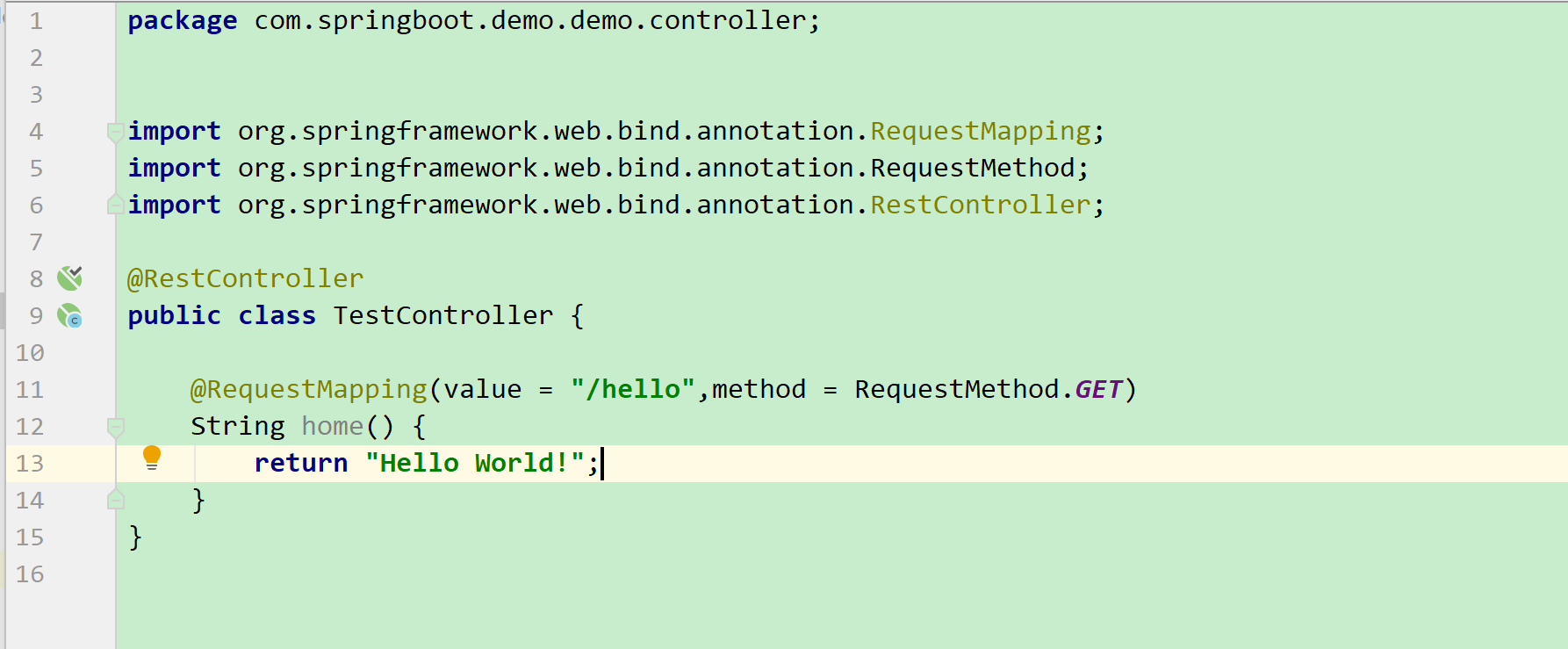
生成的springboot启动类如下：



在application.yaml文件中配置服务端口



然后编写一个controller，如下所示：



启动springboot工程，然后浏览器里访问<http://localhost:9001/hello>，可看到响应结果；



#### 2.2.3.3 Controller种类及接收参数的几种处理方式

Controller及Mapping其实不属于SpringBoot，SpringBoot只是个大杂烩的容器而已。Controller及Mapping分别在Spring的web和context包中存在着。

1.controller种类

Controller可以简单分为RestController和Controller。RestController位于Spring的web包中，Controller还是在Spring的context包中。

Controller：控制器Controller 负责处理由DispatcherServlet 分发的请求。@Controller注解的类，会作为访问的路径映射处理，不加特殊处理的返回值会被作为跳转路径。

RestController：就是@Controller + @ResponseBody 注解的综合，返回值如果是实体，一般作为json数据返回，也可以定制返回值。

**2.接收参数的几种常用方式**

**2.1请求路径参数**

@PathVariable

获取路径参数。即url/{id}这种形式。

@RequestParam

获取查询参数。即url?name=这种形式.

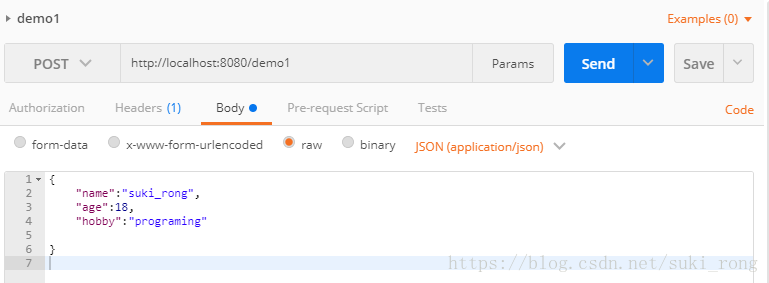
例如：

|  |
| --- |
| @GetMapping("/demo/{id}")  public void demo(@PathVariable(name = "id") String id, @RequestParam(name = "name") String name) {  System.out.println("id="+id);  System.out.println("name="+name);  } |

**2.2 Body参数**

2.2.1 @RequestBody

因为是post请求，所以这里使用postman来测试；



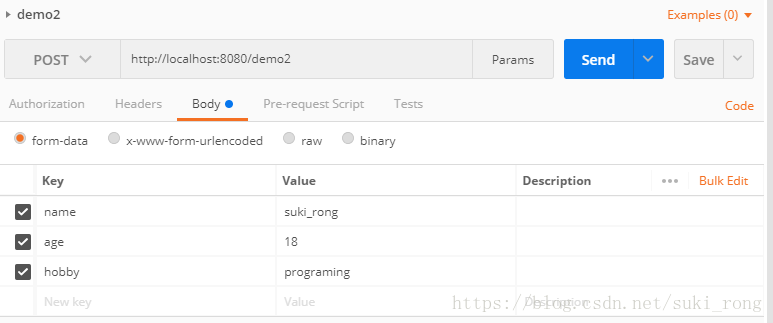
代码如下：

|  |
| --- |
| @PostMapping(path = "/demo1")  public void demo1(@RequestBody Person person) {  System.out.println(person.toString());  } |

也可以是如下代码：

|  |
| --- |
| @PostMapping(path = "/demo1")  public void demo1(@RequestBody Map<String, String> person) {  System.out.println(person.get("name"));  } |

2.2.2无注解



|  |
| --- |
| @PostMapping(path = "/demo2")  public void demo2(Person person) {  System.out.println(person.toString());  } |

**2.3 请求头参数以及Cookie**

@RequestHeader

@CookieValue

Java代码如下：

|  |
| --- |
| @GetMapping("/demo3")  public void demo3(@RequestHeader(name = "myHeader") String myHeader,  @CookieValue(name = "myCookie") String myCookie) {  System.out.println("myHeader=" + myHeader);  System.out.println("myCookie=" + myCookie);  } |

也可以是这样的：

|  |
| --- |
| @GetMapping("/demo3")  public void demo3(HttpServletRequest request) {  System.out.println(request.getHeader("myHeader"));  for (Cookie cookie : request.getCookies()) {  if ("myCookie".equals(cookie.getName())) {  System.out.println(cookie.getValue());  }  }  } |

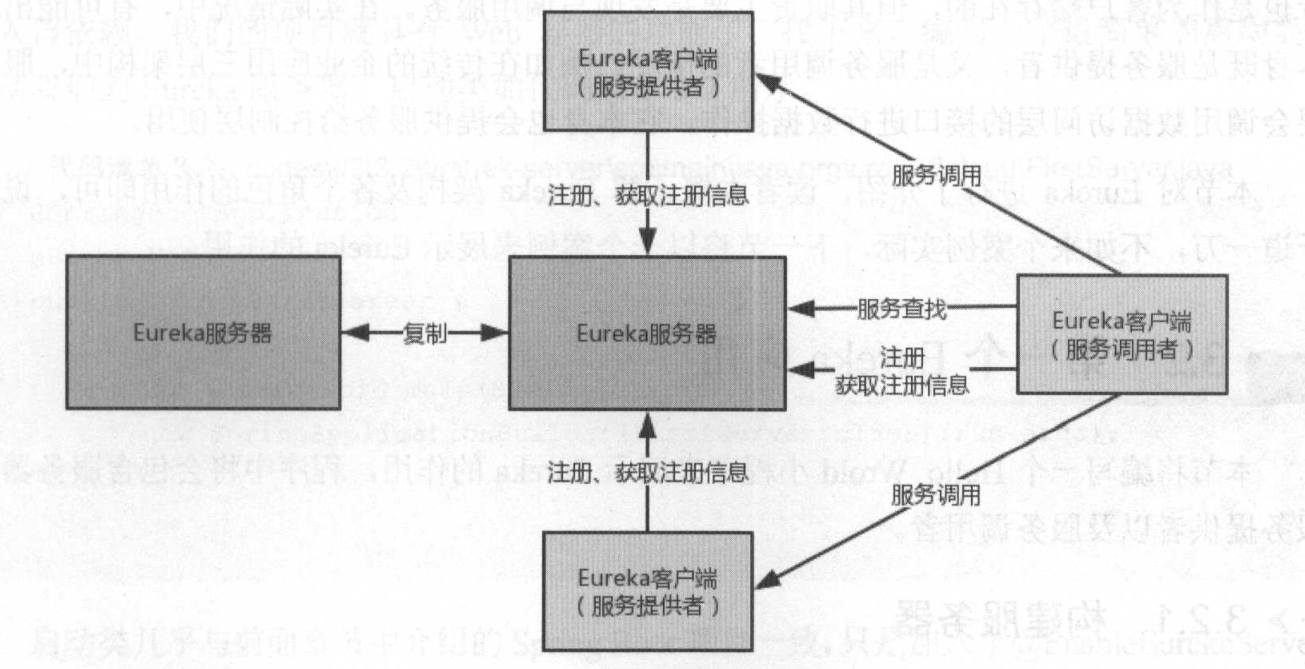
### 2.2.4 SpringCloud组件使用

这里仅介绍云平台中用到的几个组件的使用，其他组件可以参考官方文档以作了解。

#### 2.2.4.1 Eureka

Eureka提供基于REST的服务，在集群中主要用于服务管理。Eureka提供了基于java语言的客户端组件，客户端组件实现了负载均衡的功能，为业务组件的集群部署创造了条件。使用该框架，可以将业务组件注册到Eureka中，这些组件可进行集群部署，Eureka主要维护这些服务的列表并自动检查它们的状态。

一个简单的Eureka集群，需要一个Eureka服务器、若干个服务提供者。我们可以将业务组件注册到Eureka服务器中，其他客户端组件可以向服务器获取服务并且进行远程调用。架构图如下：

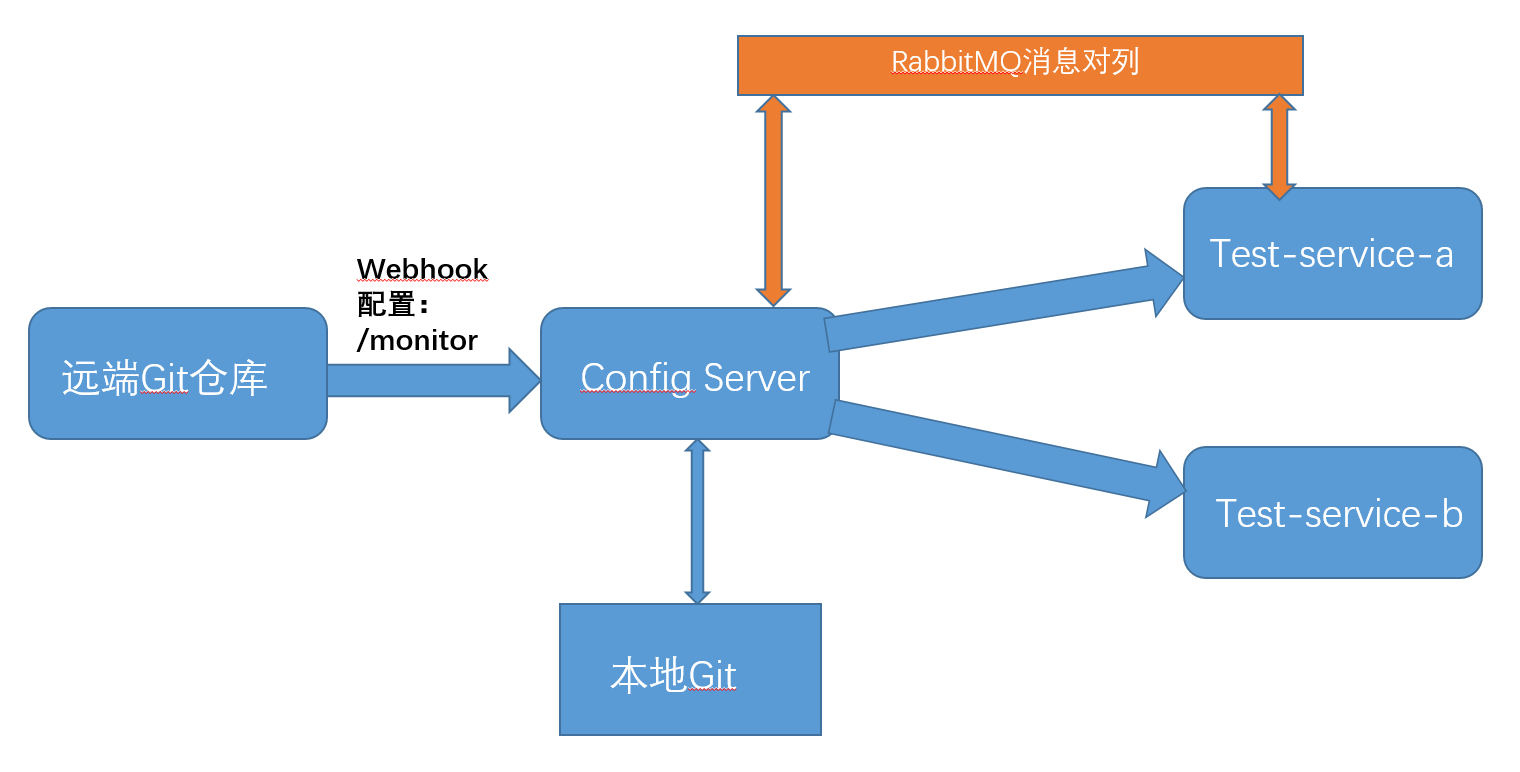


上图中有两个服务器，服务器支持集群部署，每个服务器也可以作为对方服务器的客户端进行相互注册与复制。图中的三个Eureka客户端，两个用于发布服务，另一个用于调用服务。不管是服务器还是客户端，都可以部署多个实例，这样一来，就很容易构建高可用的服务集群。

#### 2.2.4.2 Spring Cloud Config

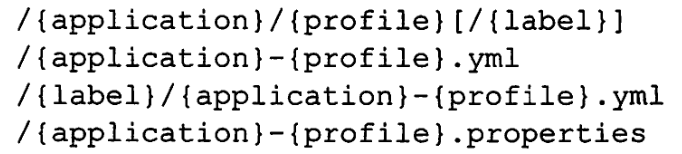
Spring Cloud Config是Spring Cloud团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，它分为服务端与客户端两个部分。**其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。**Spring Cloud Config实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于Spring构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于Spring Cloud Config实现的配置中心默认采用Git来存储配置信息，所以使用Spring Cloud Config构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过Git客户端工具来方便的管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如：SVN仓库、本地化文件系统。

动态刷新流程图如下，这里要基于Eureka注册中心，config server和config client服务都需要注册到Eureka中；



用户更新远端配置信息时，触发webhook配置地址的调用，ConfigServer接收到请求并发布消息，Bus将消息发送到config client，当config client接收到消息后会重新发送请求加载配置信息；这里使用RabbitMQ作为消息中间件；

官方给出的映射关系：

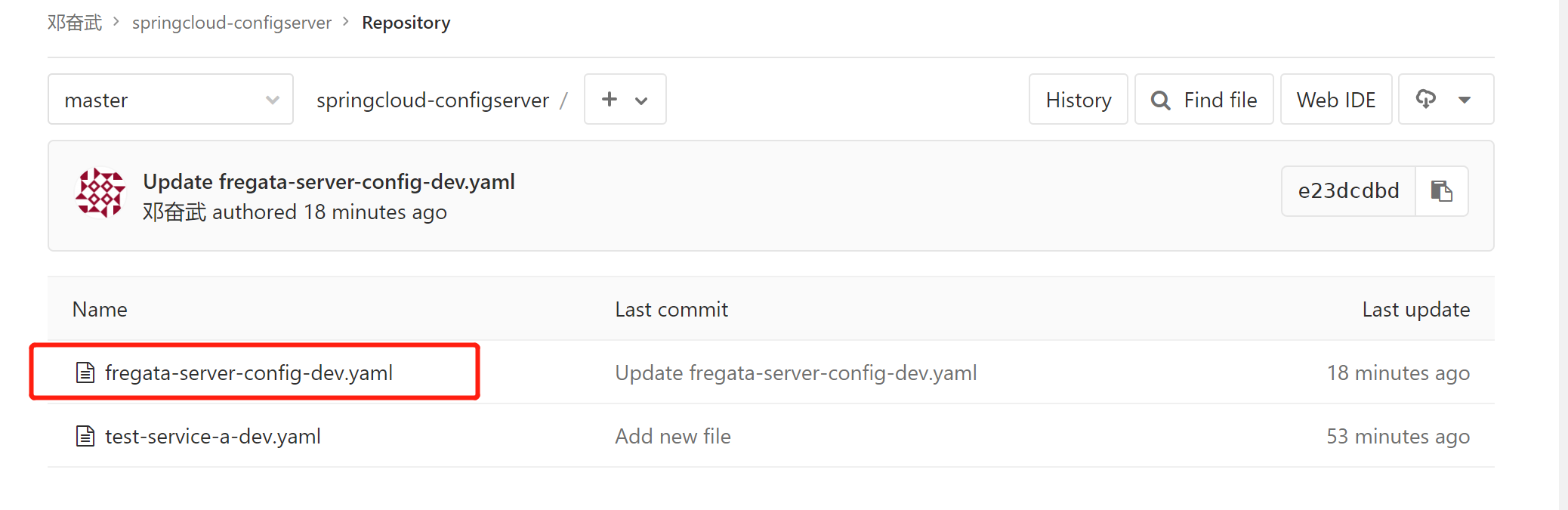




其中application是应用名，可以理解成git上面的文件名，profile是对应激活的环境名，例如dev，test，prod等；label指的是git的分支，如果不写，默认的分支为master；

我们在gitlab上自己的仓库里建立了对应的yaml配置文件，如下：

<http://192.168.102.73:8081/dengfenwu/springcloud-configserver.git>



在configserver启动来后，通过浏览器访问下面链接，可以拿到git上的配置文件的具体信息；

<http://127.0.0.1:7779/fregata-server-config/dev/master>



#### 2.2.4.3 Spring Cloud GateWay（二级网关）

Spring Cloud Gateway 是 Spring Cloud 的一个全新项目，该项目是基于 Spring 5.0，Spring Boot 2.0 和 Project Reactor 等技术开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

Spring Cloud Gateway 作为 Spring Cloud 生态系统中的网关，目标是替代 Netflix Zuul，其不仅提供统一的路由方式，并且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控/指标，和限流。

**相关概念:**

* Route（路由）：这是网关的基本构建块。它由一个 ID，一个目标 URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配。
* Predicate（断言）：这是一个 Java 8 的 Predicate。输入类型是一个 ServerWebExchange。我们可以使用它来匹配来自 HTTP 请求的任何内容，例如 headers 或参数。
* Filter（过滤器）：一个标准的Spring Filter。Spring Cloud Gateway中的Filter分为两种类型的Filter，分别是Gateway Filter和Global Filter。过滤器Filter将会对请求和响应进行修改处理。

简单的网关配置使用如下：

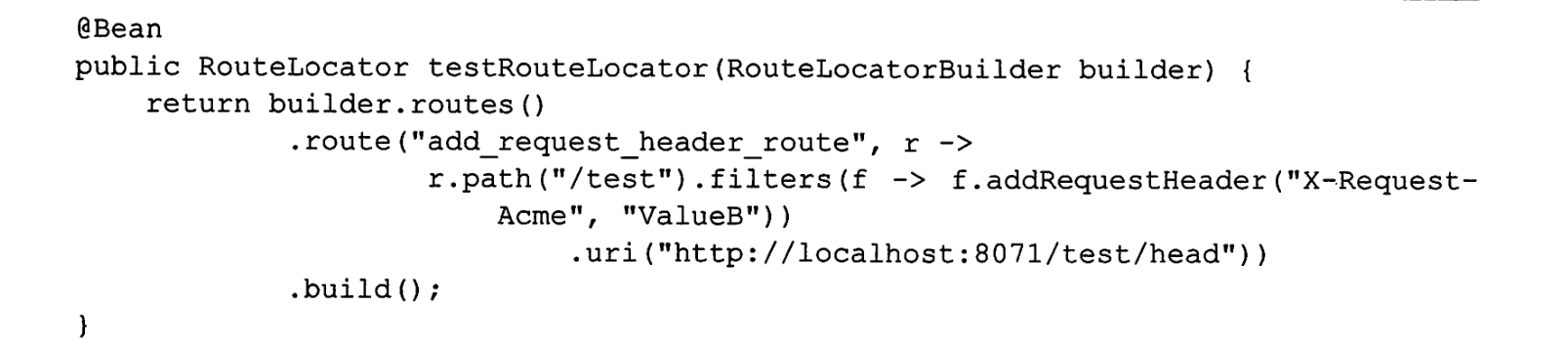


**SpringCloud Gateway内置的Filter：**

路由过滤器允许以某种方式修改请求进来的http请求或返回的http响应。路由过滤器主要作用于需要处理的特定路由。**SpringCloud Gateway提供了很多种类的过滤器工厂，过滤器的实现类将近二十多个。总的来说，可以分为七类：Header，Parameter，Path，Status，Redirect跳转，Hystrix熔断和RateLimiter。**下面简单介绍下：

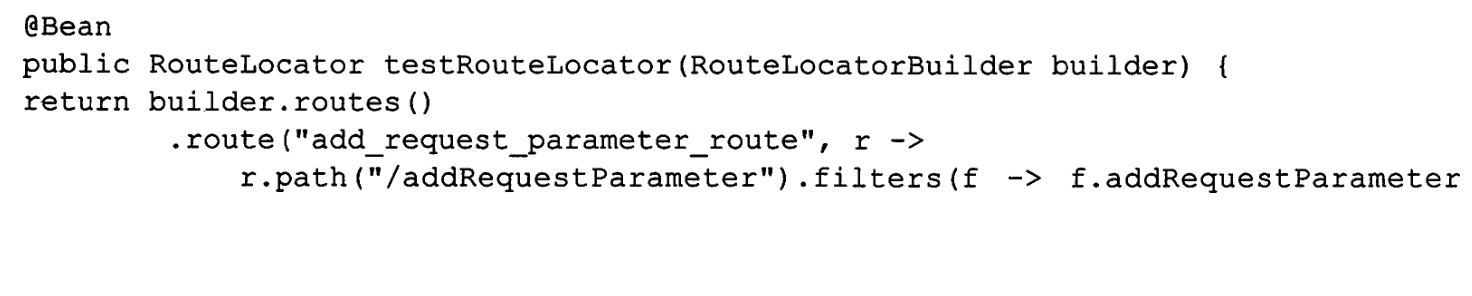
1. AddRequestHeader过滤器工厂

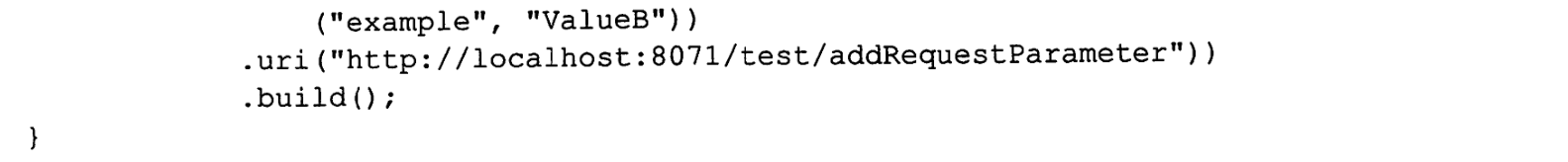
该过滤器工厂用于对匹配上的请求加上header，例如：



1. AddRequestParameter过滤器

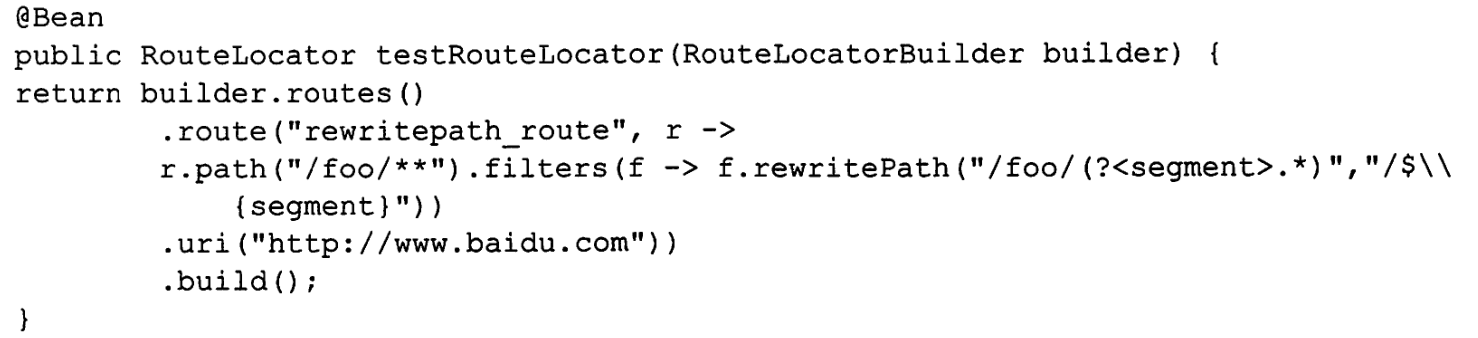
该过滤器作用是对匹配上的请求路由添加请求参数，例如：





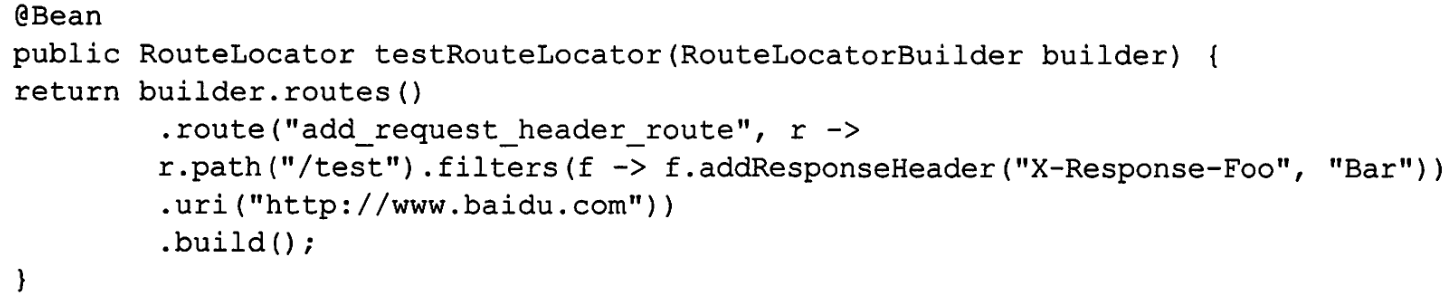
1. RewritePath过滤器

Springcloud gateway可以使用RewritePath替换Zuul的StripPrefix功能，而且功能更加强大；



1. AddResponseHeader过滤器

该过滤器工厂的作用是对从网关返回的响应添加header,例如：



1. StripPrefix过滤器

StripPrefixGatewayFilterFactory是一个针对请求url前缀进行处理的filter工厂，用于去除前缀。而PrefixPathGatewayFilterFactory是用于增加前缀。



更多详细的Spring Cloud Gateway的使用，请参考官方文档：

<https://cloud.spring.io/spring-cloud-static/Finchley.SR4/single/spring-cloud.html#_spring_cloud_gateway>

#### 2.2.4.4 Feign

Feign是Netflix开发的声明式、模板化的HTTP客户端，其灵感来自Retrofit、JAXRS-2.0以及WebScoket。可帮助我们更加快捷、优雅地调用HTTP API。

在Spring Cloud中，使用Feign非常简单：创建一个接口，并在接口上添加一些注解，就ok了。

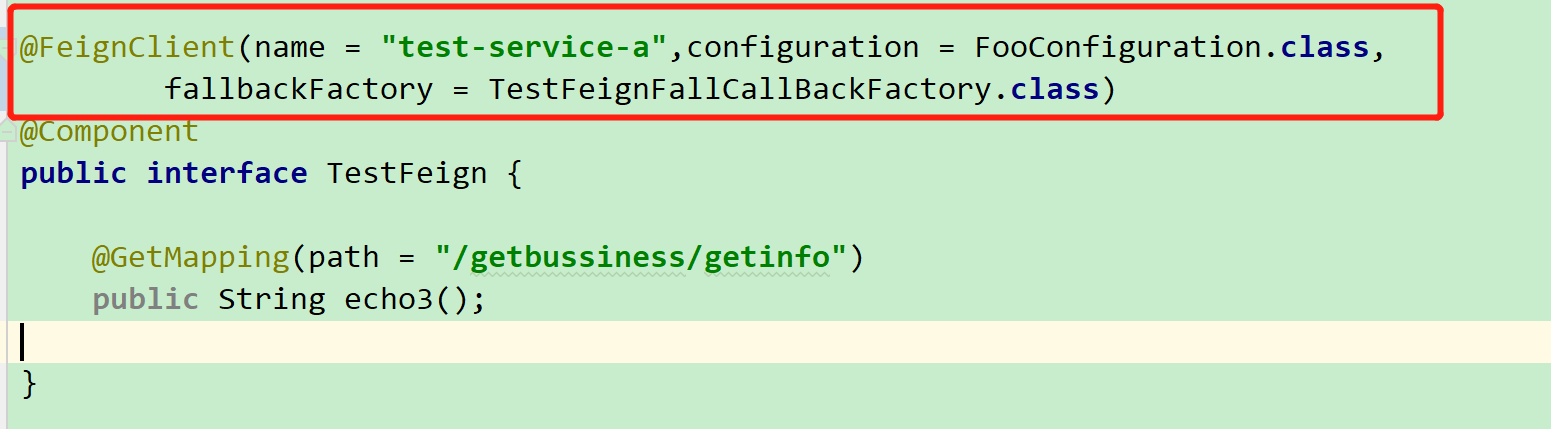
1.引入依赖：

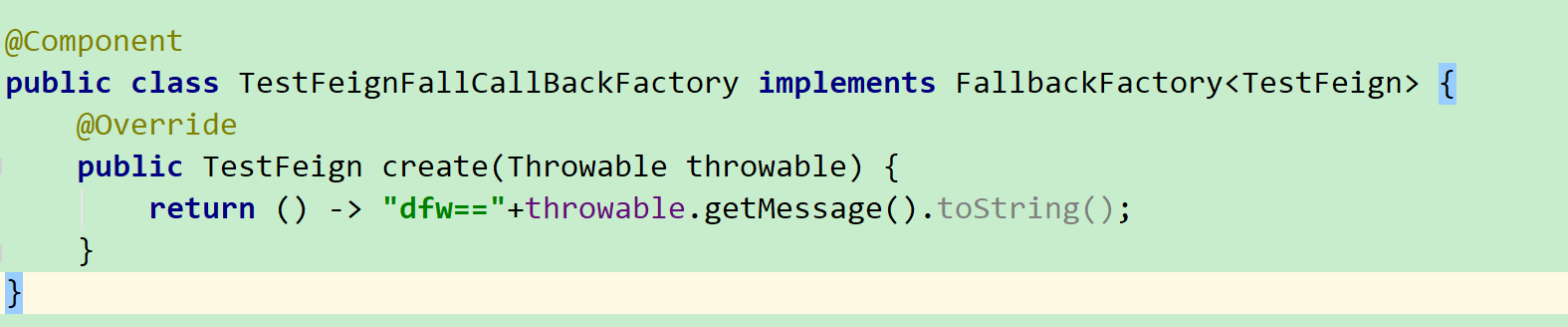
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-openfeign</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-actuator</**artifactId**> </**dependency**>  <**dependency**>  <**groupId**>com.netflix.hystrix</**groupId**>  <**artifactId**>hystrix-metrics-event-stream</**artifactId**>  <**version**>1.5.18</**version**>  <**scope**>compile</**scope**> </**dependency**> |

2.启动类中开启Feign的支持：



3.创建一个Feign接口，并添加@FeignClient注解；注解中的name是调用的服务的名称，用于创建Ribbon负载均衡器。在这里，使用了Eureka，所以Ribbon会把test-service-a解析成Eureka Server服务注册表中的服务。还可以使用url属性指定请求的URL（URL可以是完整的URL或者主机名），例如：@FeignClient(name = “test-service-a”,url = “http://localhost:8094/”)





4.创建controller，用Feign发起调用，到这里Feign的简单使用就完成了；

