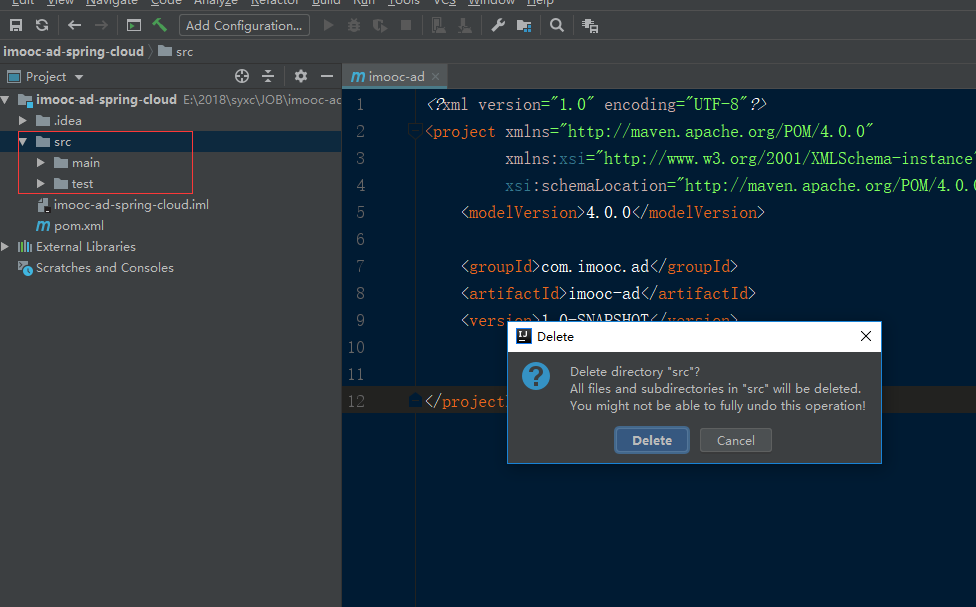
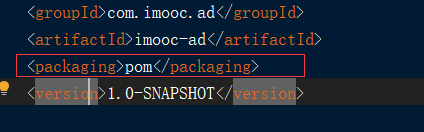
因为是一个maven项目

有一个源文件的目录和pom文件的目录 由于我们是一个主工程 pom文件取管理其他的子模块 所以直接将src直接删除掉就可以了





为了方便以后项目的管理我们还是需要加上这个配置

<name>imooc-ad-spring-cloud</name>  
<description>Project For imooc Ad SpringCloud</description>

MAVEN下载依赖的过程 首先会检验本地仓库是否有jar包 如果没有他就去中央仓库寻找

如果中央仓库也找不到 那么他会看你的仓库里面是否定义了远程仓库 如果没有定义远程仓库 那么就直接报错了 如果定义了远程仓库他会尝试从远程仓库中寻找 如果远程仓库中也找不到 那么就会抛出错误

我们的springcloud

总结：

主工程POM文件的依赖计划

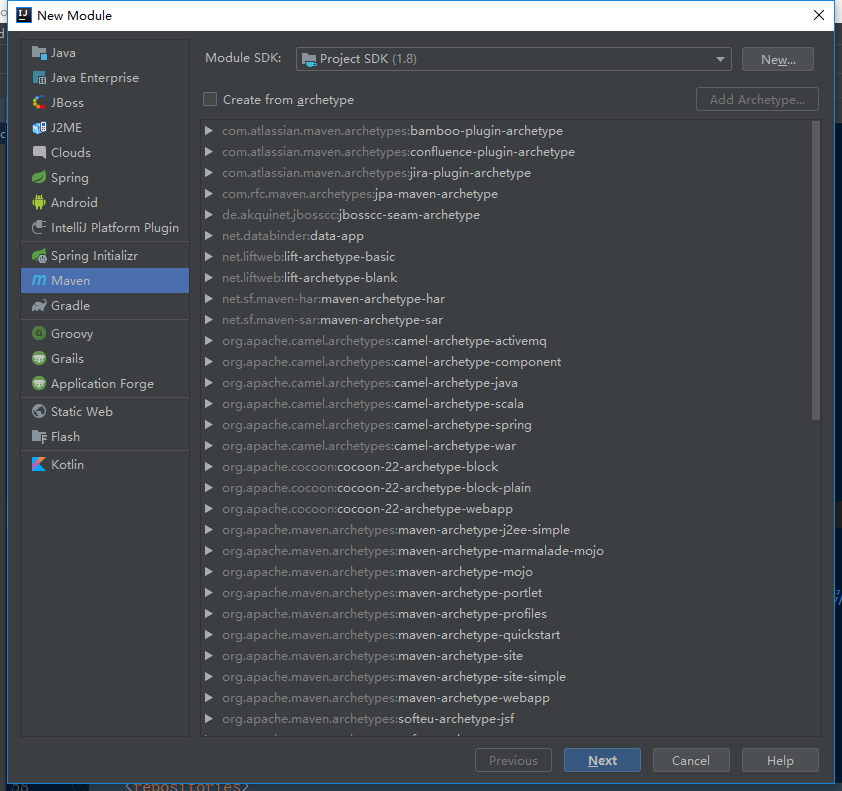
对于有些标签需要熟悉一下

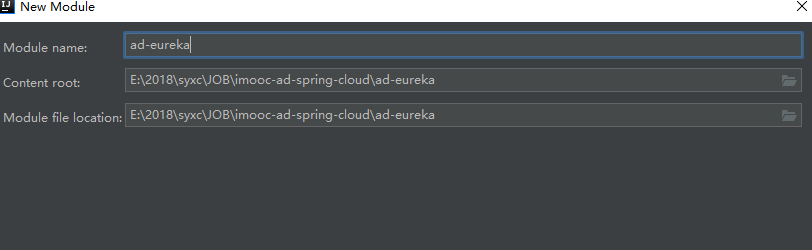
<!--父项目的坐标。如果项目中没有规定某个元素的值，那么父项目中的对应值即为项目的默认值。 坐标包括group ID，artifact ID和 version。-->

0.2 **Eureka是实现服务注册 服务发现的产品**

他是一个基于rest的服务 定位服务来进行中间层服务器的负载均衡和故障转移 eureka采用了cs的架构，他有一个服务器和多个客户端 EurekaServer他是服务注册中心 而系统中其他的为服务他会使用Eureka的客户端连接到eurekaserver上 维持一个心跳连接 告诉eurekaserver 我这个微服务还是存在的 系统的维护人员或者客户端来监控系统其它各个微服务是否正常运行 springcloud的一些其它模块 比如像zoo这样的网关

在主项目中创建一个子模块





创建一个启动类

Public class EurekaApplication{

Public static void main(String[] args){

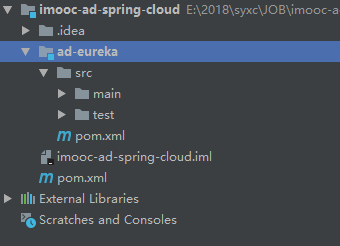
SpringApplication.run(EurekaApplication.class,args);

}

}

编写配置文件

项目目录结构



spring:

application:

name: ad-eureka

server:

port: 8000

eureka:

instance:

hostname: localhost

client:

#表示从eureka server 获取注册信息 默认是true 因为这里是一个单节点EurekaServer 不需要同步其它eureka结点的数据

#同时也没有eureka server 所以说这里需要设置为false

fetch-registry: false

#这个参数表示是否将自己注册到eurekaserver 默认是true

register-with-eureka: false

#查询服务和注册服务都要依赖于这个地址

service-url:

defaultZone: httt://${eureka.instance.hostname}:${server.port}/eureka/

Eureka Server的部署

1. 启动单节点的eureka server
2. 启动多节点的eureka server

访问

<http://127.0.0.1:8000/>

配置

---

spring:

application:

name: ad-eureka

#在启动程序的时候把它打成一个jar包之后 需要使用profiles来启动对应配置文件的服务

profiles: server1

server:

port: 8000

eureka:

instance:

hostname: server1

prefer-ip-address: false

client:

service-url:

defaultZone: http://server2:8001/eureka/,http://server3:8003/eureka/

---

spring:

application:

name: ad-eureka

#在启动程序的时候把它打成一个jar包之后 需要使用profiles来启动对应配置文件的服务

profiles: server2

server:

port: 8001

eureka:

instance:

hostname: server2

prefer-ip-address: false

client:

service-url:

defaultZone: http://server1:8000/eureka/,http://server3:8003/eureka/

---

spring:

application:

name: ad-eureka

#在启动程序的时候把它打成一个jar包之后 需要使用profiles来启动对应配置文件的服务

profiles: server3

server:

port: 8003

eureka:

instance:

hostname: server3

prefer-ip-address: false

client:

service-url:

defaultZone: [http://server1:8000/eureka/,http://server2:8002/eureka/](http://server1:8000/eureka/,http:/server2:8002/eureka/)

在host里面修改

Hosts

127.0.0.1 server1

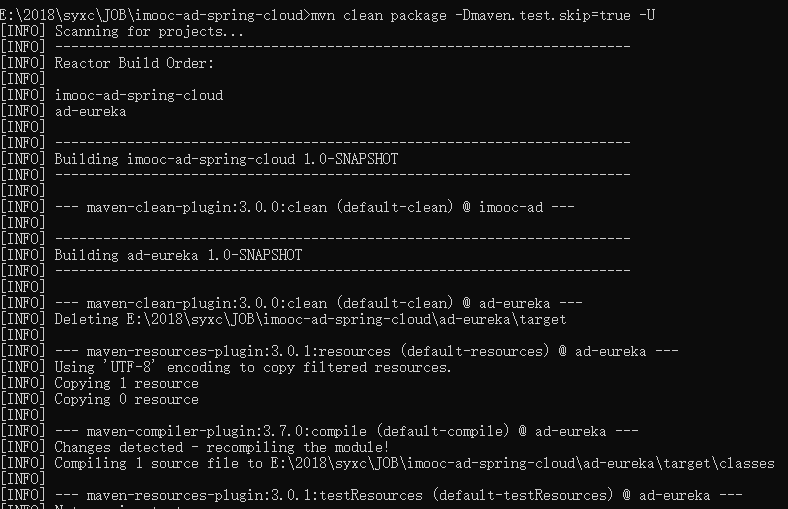
127.0.0.1 server2

127.0.0.1 server3

在父工程里面打包

E:\2018\syxc\JOB\imooc-ad-spring-cloud>mvn clean package -Dmaven.test.skip=true -U

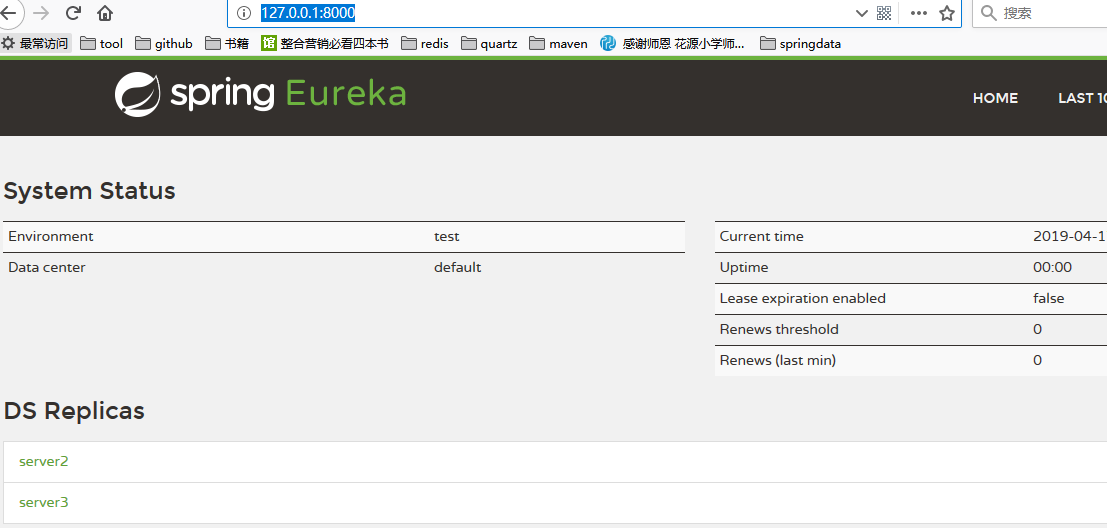
-D 跳过测试用例 打包过程会快一些

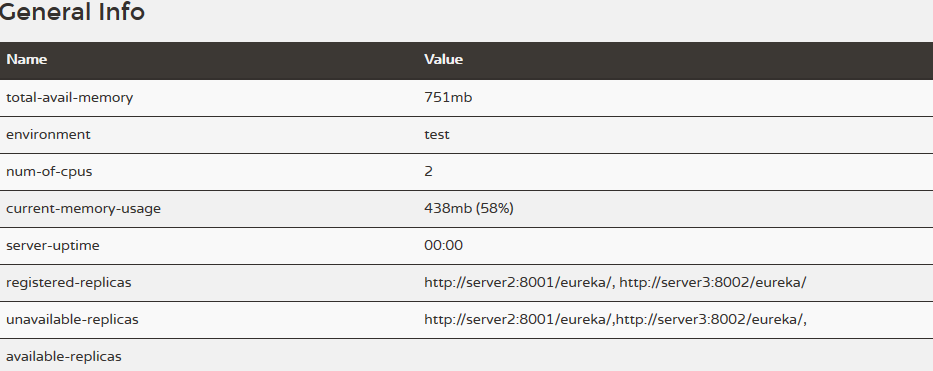


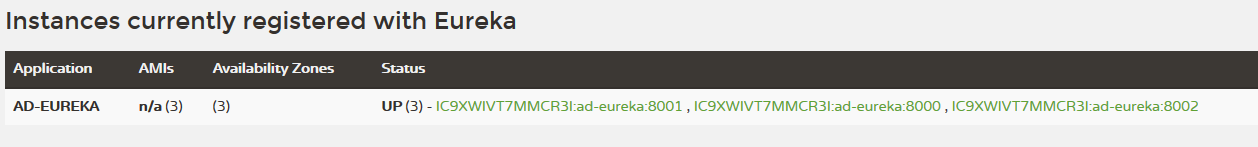
E:\2018\syxc\JOB\imooc-ad-spring-cloud\ad-eureka\target>java -jar ad-eureka-1.0-SNAPSHOT.jar --spring.profiles.active=server1

在启动的时候应该会报错 应为启动的时候 找不到server2 server3

访问



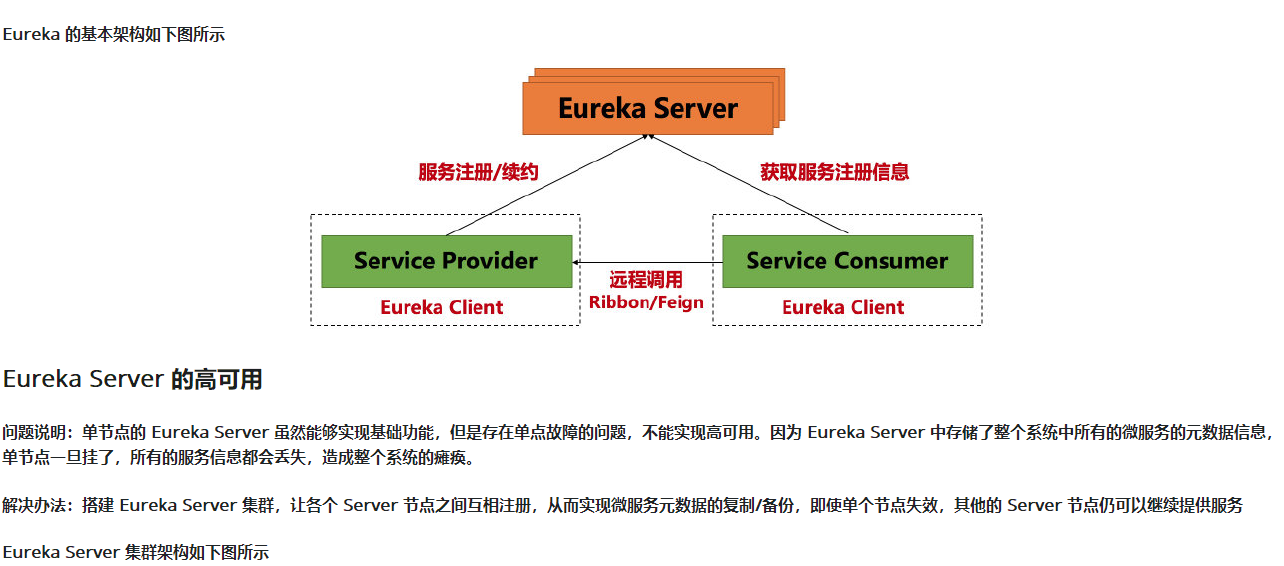
注册的副本是制定的8001和8002但是由于8001和8002没有启动

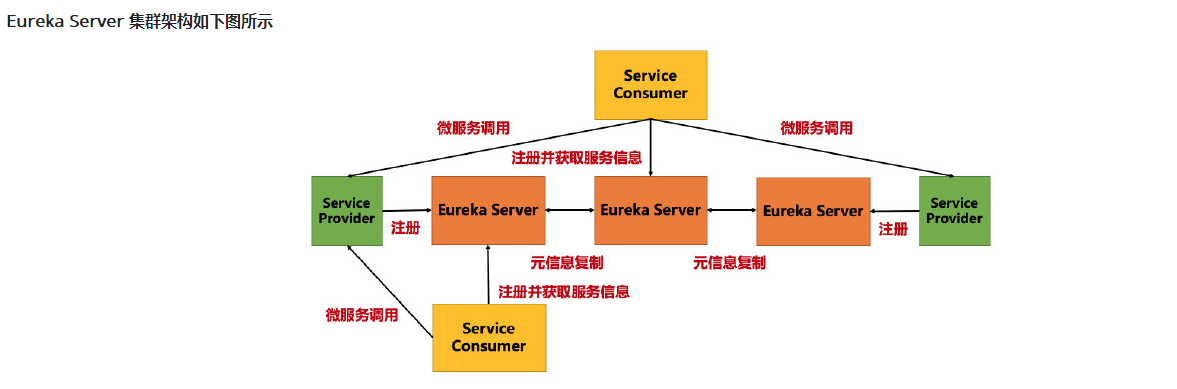


这样就完成eurekaserver 的高可用注册

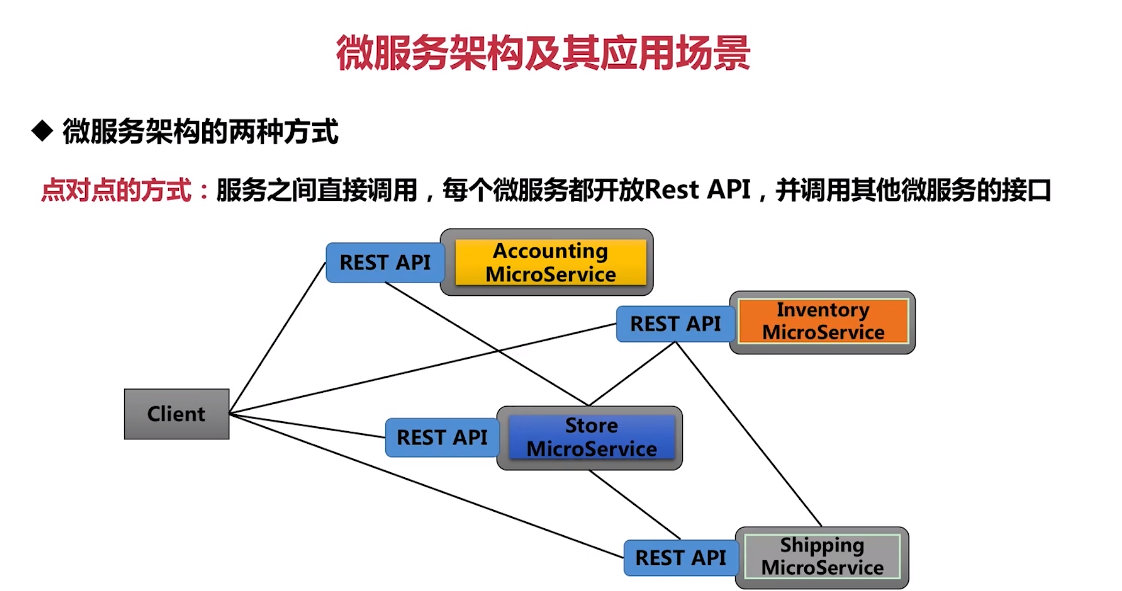
总结篇









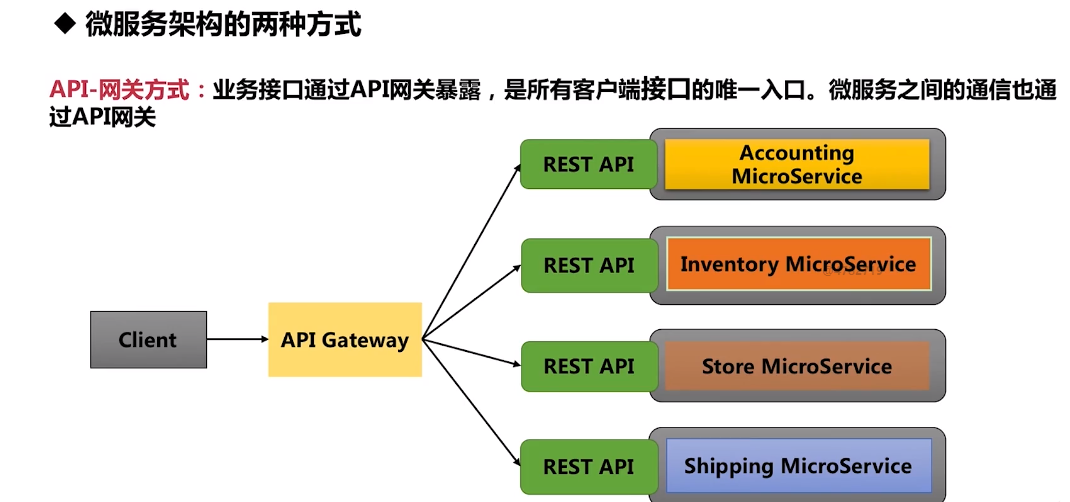


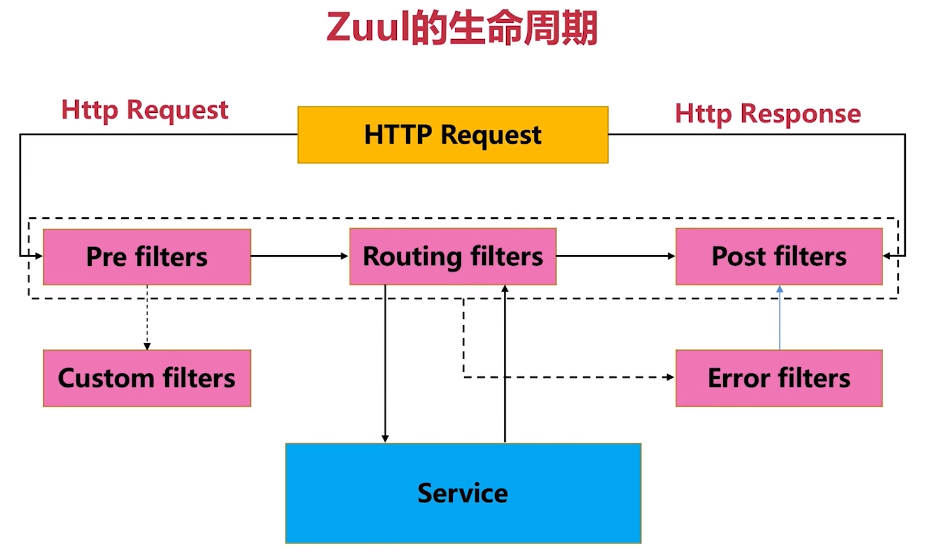
微服务架构的定义：

微服务是一种架构风格，一个大型的复杂的软件有一个或者多个为服务构成，系统中的各个微服务可以被独立部署各个微服务之间是松耦合的 每个为服务仅关注维护一件任务b并能够很好的完成这个任务，每个任务代表着一个小的业务能力关于微服务的架构目前有两种主要的方式

第一种：点对点 服务之间是直接调用的 每个为服务都开发Rest Api 提供http服务的调用并调用其它为服务接口很明显这在比较简单的为服务调用场景下是可行的并且非常简单 但是随着应用复杂度的低声变得越来越不可以维护

第二种：API网关的核心要点是所有的消费 提供端都可以通过网关接入微服务 在网关层处理非业务功能 所有API通过Gateway去暴露 微服务之间也通过微服务网关去通信





zuul网关是通过filter构成的 我们对这些过滤器实现简单的介绍 过滤器实现了哪些功能以及有什么样的作用(Pre filter是进来的时候执行的 Post filter是出去的时候执行的)

网关应用程序开发

创建另一个子模块

首先创建一个package

在创建两个filter类

PreRequestFilter AccessLogFilter

Public class PreRequestFilter extents ZullFilter{

//定义filter类型 总共有4种 pre post Routing filter

@Override

public String filterType() {

return FilterConstants.PRE\_TYPE;

}

//定义filter执行的顺序 数字越小表示顺序越高 越先被执行

//定义同种类型的过滤器有很多个 哪个过滤器先执行 哪个过滤器后执行 就是通过filterOrder去定义

@Override

public int filterOrder() {

return 0;

}

//表示是否需要执行这个过滤器 true表示执行 false表示不执行 之所以有这个是因为我们可以定义一个过滤器

//但是当某些条件发生的时候才会执行

@Override

public boolean shouldFilter() {

return true;// 需要永远执行

}

@Override

public Object run() throws ZuulException {

RequestContext ctx = RequestContext.getCurrentContext();///请求上下文

ctx.set("startTime",System.currentTimeMillis());

return null;

}

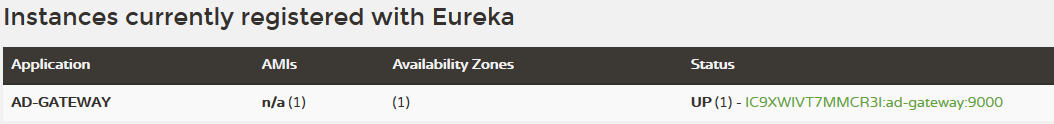
}

分别启动

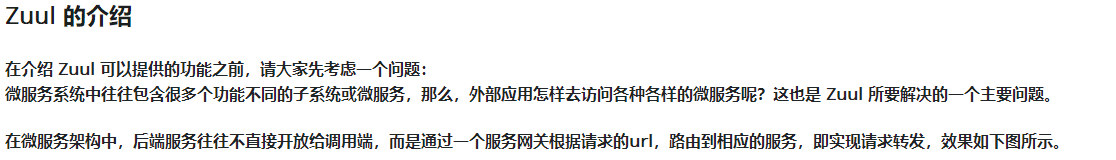
eurekaApplication

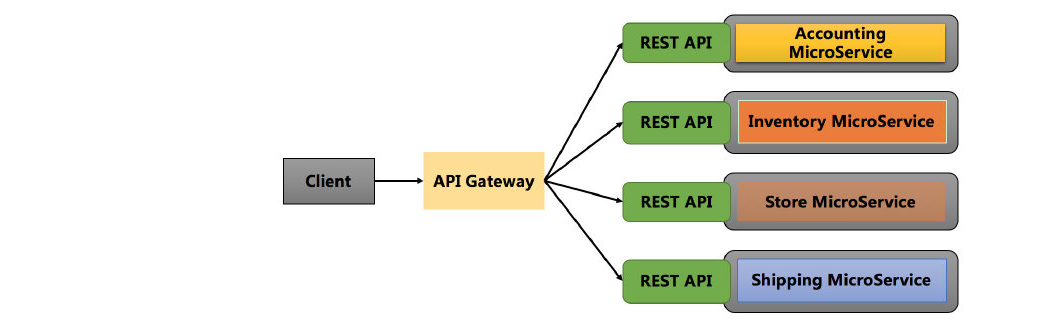
ZuulGatewayApplication

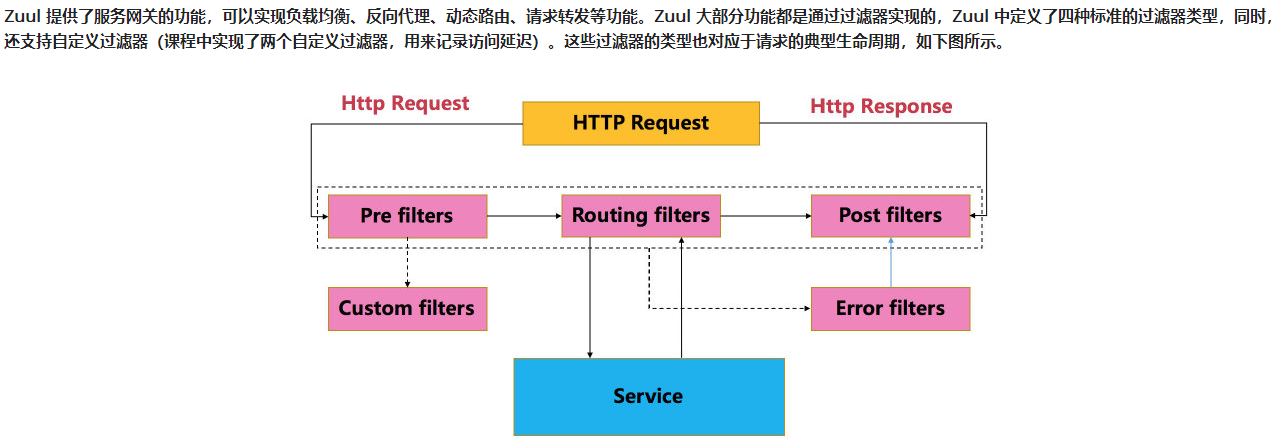
看到网关也就注册上去了

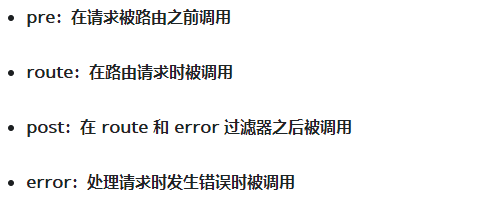


总结：



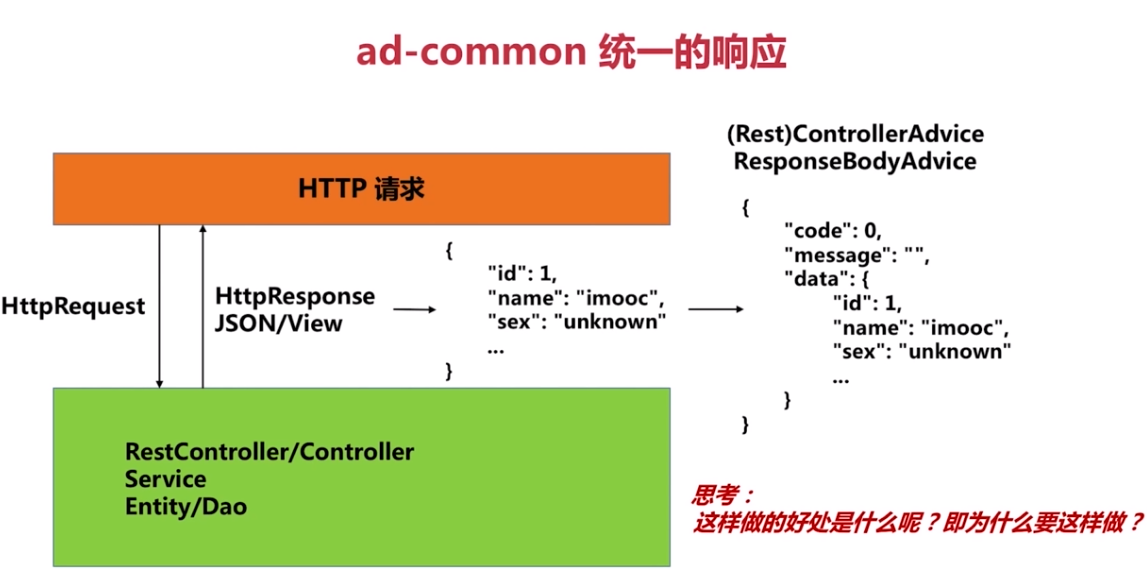








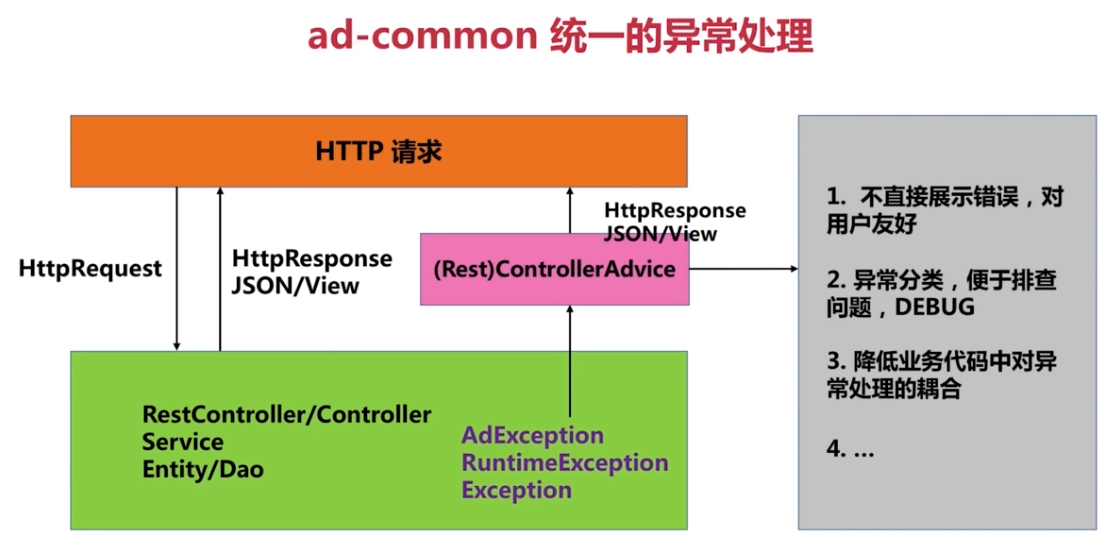
比如说在广告检索系统，广告投放系统的通用代码 那么就不适合在两处都定义 如果要更改的化两处都要更改 在一处定义后 其它拿过来用就好了



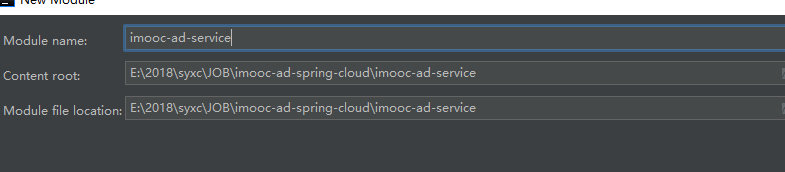
ad-common统一异常处理的概念

同样的http请求在处理的过程中 有可能会出现异常 比如在service里面抛出了异常 异常可能是adException 也是统一处理异常 如果我们直接返回一个错误给用户 用户看见这个错误是非常不友好的 所以有可能会对异常做统一的处理就比如对异常进行拦截拦截之后 往异常里面写入一些消息 用户你好我们的页面已经出错了请您稍后再试等等这样的包装 而不对用户显示404 502 第一对用户友好 第二对异常进行分类排查错误 第三点业务代码对异常处理的耦合 如果我们不做统一的异常处理 那么我就得在业务层里面 每一种可能的出现异常的地方进行try{}catch 然后在catch里面做一个异常的包装和处理，这样的化会使代码量大量增加 而且业务代码耦合比较紧密 如果我要把异常处理的过程处理一下 那么就需要修改service所有异常处理过程

我们在对异常处理的时候也会利用到restControllerAdvice或者是controllerAdvice 他是一个控制器增强 它是对响应的对象进行拦截 由于异常也是通过响应返回给用户的 所以对异常的拦截也就是对响应的拦截

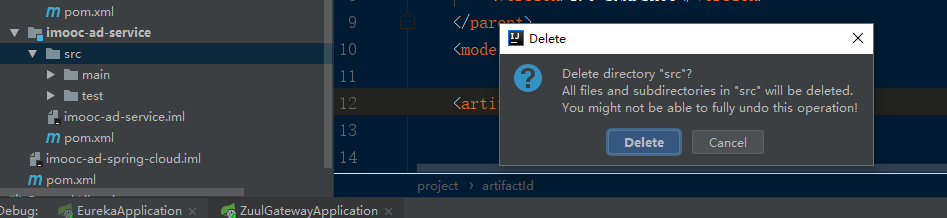


创建model;

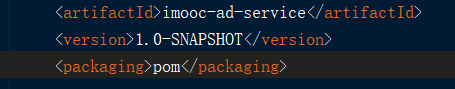


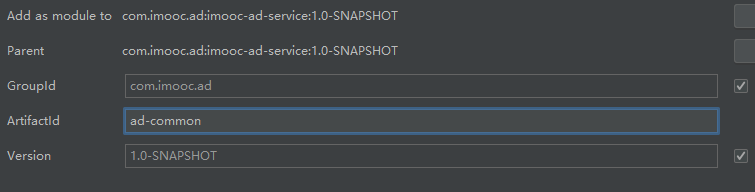
imooc-ad-service 将来回作为一个父模块出现

删除src (因为我们在这里并不去创建真正的代码在这里面)



修改完以后就可以管理广告层业务





创建ad-common的模块以后 跟着就是修改pom文件

创建CommonResponseDataAdvice implements ResponseBodyAdvice<Object>{

实现内部两个方法

1. 这个响应是否应该去拦截

（同时还要添加一个注解有些是不需要使用这个commonResponse）

1. 在写入响应之前可以做一些事情

}

统一异常处理开发

1. 定义一个异常 说明要处理的异常类 这里可以根据业务定义N种异常

@ExceptionHandler(value = AdException1.class)

public CommonResponse<String> handlerAdException1HttpServletRequest request, AdException1 ex){

//我们怎么告诉spring 我们要处理AdException这个异常呢

CommonResponse<String> response = new CommonResponse<>(-1,"business error");

response.setData(ex.getMessage());

return response;

}

@ExceptionHandler(value = AdException2.class)

public CommonResponse<String> handlerAdException1HttpServletRequest request, AdException2 ex){

//我们怎么告诉spring 我们要处理AdException这个异常呢

CommonResponse<String> response = new CommonResponse<>(-1,"business error");

response.setData(ex.getMessage());

return response;

}

统一配置开发

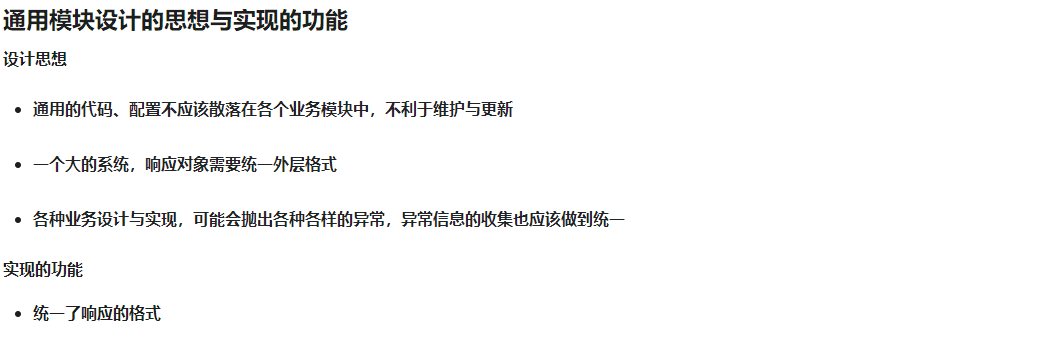
通用配置的定义 通用配置是非常非常重要的 平时我们在企业级开发中经常用到redis

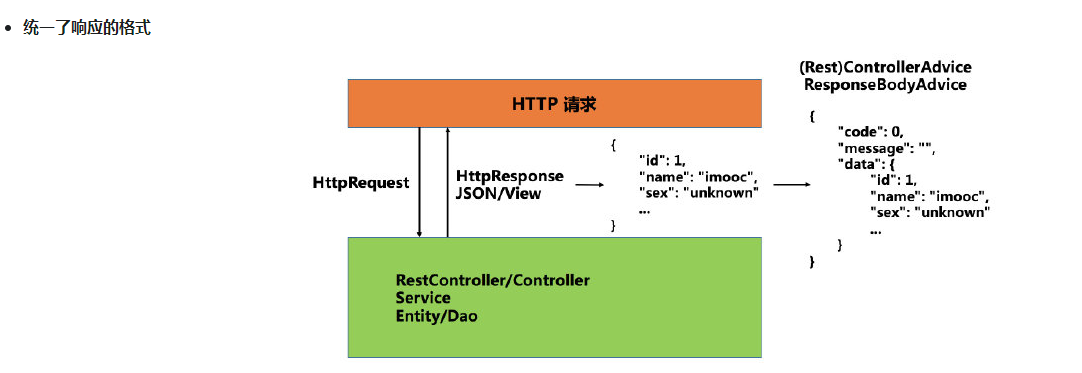
mysql的特殊配置

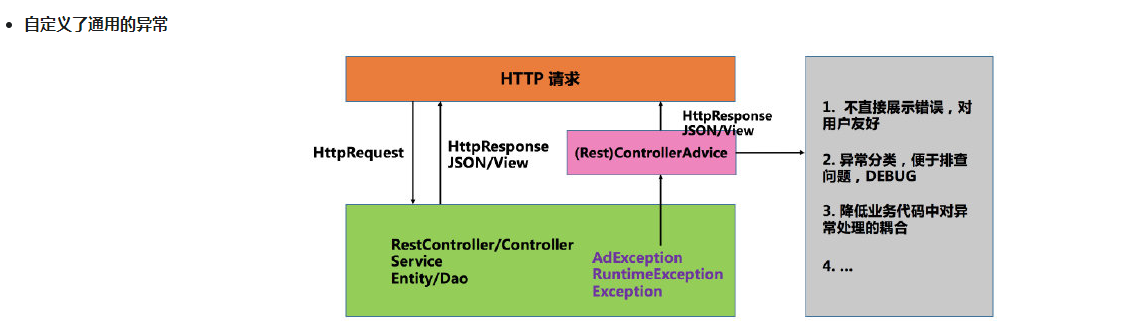
我们来实现一个http消息得转发器 将java的实体对象 转换为对象的实体输出流

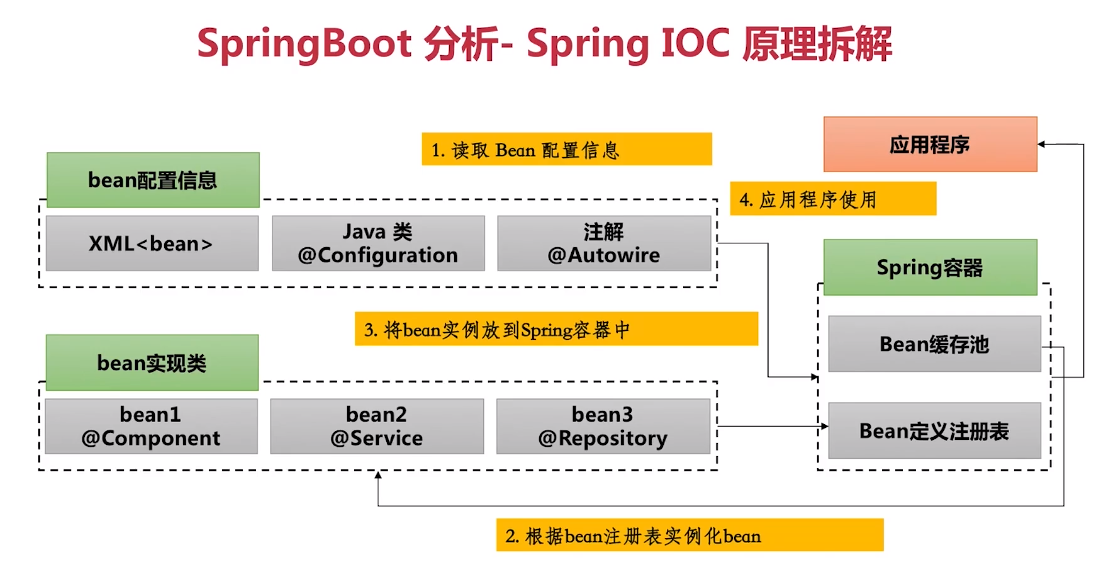
springboot的底层会通过httpmessageConverts会依靠这个json库将java的实体类输出为json格式当有多个转换器可用的时候 他会根据消息的对象类型和需要的内容类型选择最适合的转换器去使用 消息转换器的目标是将http输入请求格式向java对象转换 或者java对象向http输出请求的转换 有的消息转换器支持多个数据类型 有的只支持简单的输出格式 还有就是支持数据类型比较多 输出的格式也很多 去修改或者去定制这个http消息转换器最简单的方式是重写这个configmessageConverts方法添加自定义转换器

总结：







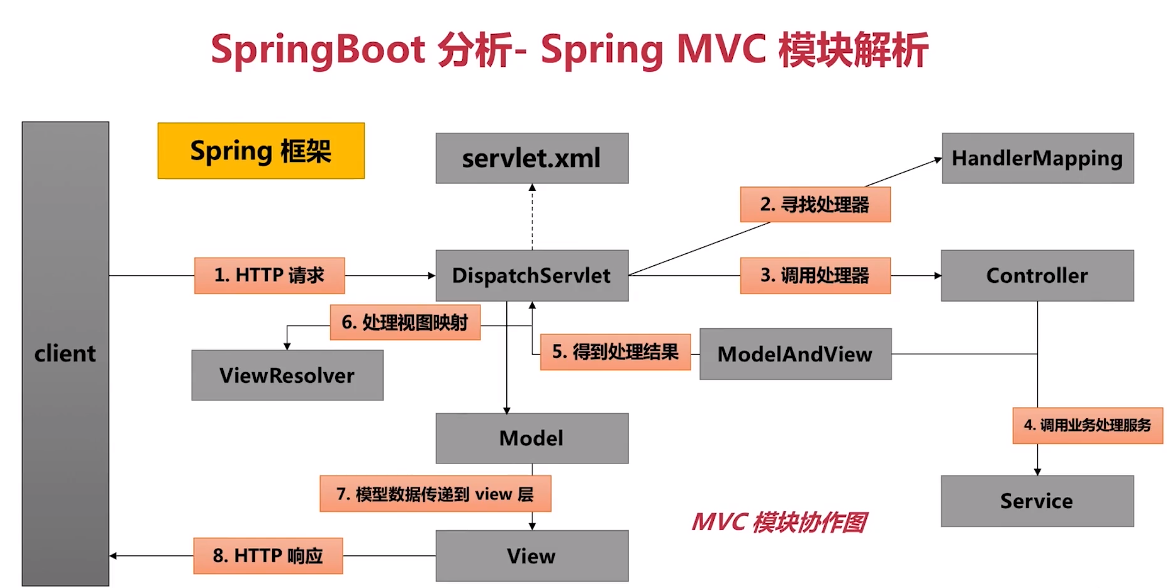


springboot其实就是基于springmvc和spring做了一些自动配置 形成了一个简易使用的自动框架 先对springboot做一个简单的分析

spring一般指的是框架他是一个开源的应用程序框架 提供了一个简易的开发方式通过这种开发方式他将避免那些可能致使代码变得繁杂混乱大量业务工具组合在一起包括创建与销毁等等 他去帮你管理 再被管理对象与业务逻辑之间呢spring会通过控制反转 就是IOC来架起使用的桥梁 IOC也可以看做是spring最重要最核心的思想 spring在启动的时候会读取应用程序bean配置的配置信息并在spring容器中生成一个bean配置表然后根据注册表实例化各个bean 装备好bean之间的依赖关系为上层应用准备就绪的应用环境

dispatcher得到了处理结果之后回去查询一个或者多个处理视图的映射叫做viewResolver,找到modelandview指定的视图然后对这个model数据进行渲染得到一个view 最后返回给客户端

1. 不仅仅是把所有bean实例化出来 我们bean需要做依赖注入 我们的一个bean需要依赖另一个bean 将spring创建和管理 然后放到spring容器中 最后提供给应用程序去使用



看一下springmvc的运行原理

首先有一个客户端client 对应用程序发起请求 对springmvc的map程序发起请求 http请求

最先是到了dispatchServlet (他是spring提供的前端控制器) 所有的请求都有他来统一分发 们可以认为他就类似于一个网关一样把我们的请求进行分发 depatcherServlet将请求分发给servlet之前需要借助spring提供的handlerMapping handler就是具体的处理方法 mapping就是找到对应的处理方法 3.depatcherServlet找到了controller就需要将请求提交给controller

dispatcherServlet主要三个功能

1. 捕获符合规则的请求
2. 初始化上下文执行各个逻辑
3. 初始化springmvc的各个组成部件 装配到dispatchServlet中以便完成操作



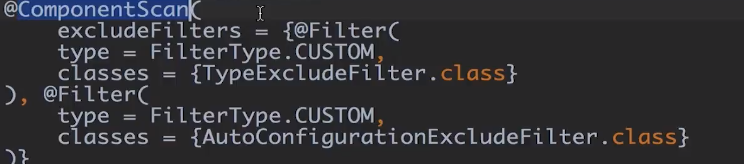
@SpringbootApplication这个注解的功能是一个组合注解

@Configuration 这个注解是一个类级别的注解 只是对象是一个bean定义的源

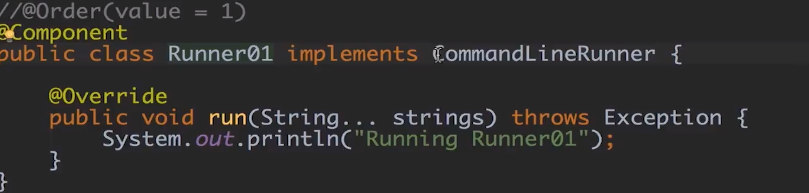
@Configuration 通过@bean用来表示一个方法实例化配置和初始化是由springioc管理的新的对象



@EnableAutoConfiguration 是启用spring上下文的自动配置 它会视图猜测和配置我们想要使用额bean 自动配置类通常采用基于你的类路径classpath 进行应用 自动配置springboot的核心功能点

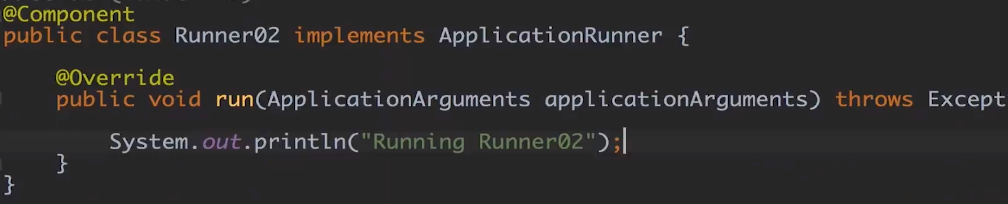


他是为@Configuration注解的类配置组件的扫描指令 同时提供了spring.xml元素并行的支持如果指定的包没有被定义 就会进行该注解所在的类声明的包进行扫描 这个注解就是@ComponentScan 他会自动扫描指令包下全部标有@Componnet的类并把他们注册为springioc中的一个类当然也包括@Component下的子注解 @Service @Controller



它里面这个run方法是在应用程序启动了自动运行

可以看到Runner02 优先于Runner01执行



push