**[学习微服务-第2天] ServiceComb + SpringCloud Zuul源码解读**

[**上一篇文章**](https://mp.weixin.qq.com/s?__biz=MzUxNTEwNTg5Mg==&mid=2247485412&idx=1&sn=eb37201cde786da0e51627d47a61d600&chksm=f9baf2efcecd7bf9d6228db97cffa76279b0b626b4372e902761db319034ae8df3a41ba149cb&mpshare=1&scene=21&srcid=0123jPnb8HqnUwcTt1ODNdeN&key=38f60bf83a35f895b74c563b1d6f5817c61529175bffb1cde1ce7eacc94e212a15d11ce564b59f4911971618f40f60c9861b463f924a735df85872c2ad1c1a43d0063da1a581780796653727515dec15&ascene=1&uin=MTU2MzQyMjM1&devicetype=Windows%20Server%202008%20R2&version=62060719&lang=zh_CN&pass_ticket=/jguedqyXORdMXwtrGGWtyQr5vi4HHQZKuGhdsstuCI=&token=138424242#wechat_redirect)我们介绍了ServiceComb与SpringCloud的Zuul网关组件协同工作，以构建微服务应用。

为了给ServiceComb做贡献的伙伴提供指引，本篇将介绍ServiceComb与SpringCloud Zuul的集成源码。

**ServiceComb 对接 Spring Cloud Zuul 思路**

ServiceComb没有修改SpringCloud Zuul的源代码，而是利用了 SpringCloud 提供的可扩展的接口。

**Spring Cloud Zuul官网有如下两段描述**

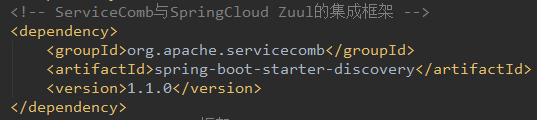
•Zuul starter不包括服务发现客户端, 所以为了实现基于service ID的路由转发，你必须同时在类路径下提供一个服务发现客户端 （ 可以使用 Eureka ）

•DiscoveryClientRouteLocator 过滤器从一个DiscoveryClient（例如Eureka）和属性文件中加载了路由定义信息。

详情参考：https://cloud.spring.io/spring-cloud-netflix/multi/multi\_\_router\_and\_filter\_zuul.html

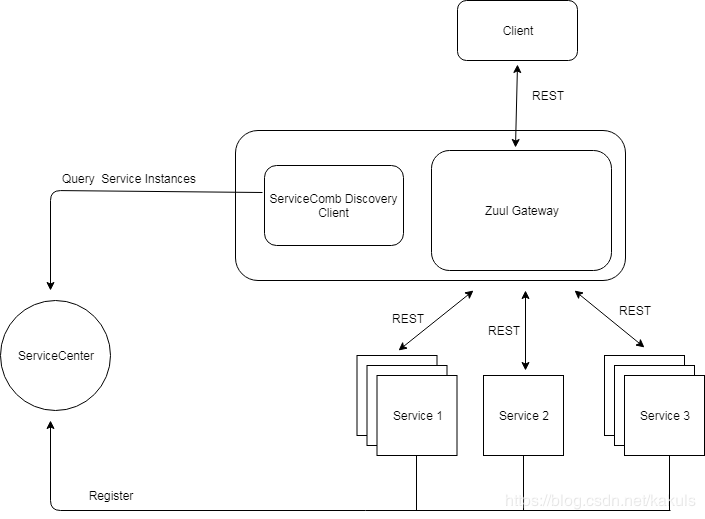
从以上的描述看，SpringCloud Zuul允许我们自定义服务发现客户端来实现自己的服务发现逻辑。其中DiscoveryClient接口是Spring Cloud Commons提供的与服务治理相关的抽象接口，Spring Cloud Commons做了一层抽象，很好的解耦了服务治理体系，使得我们可以轻易的替换不同的服务治理设施。

在我们的上篇文章实现的zuulserver项目中，pom文件中有如下依赖



以上就是一个自定义discovery client。这个discovery client是专门与ServiceComb的服务与发现注册中心ServiceCenter进行交互的。

**如下图所示**，客户端和各个微服务都与Zuul网关直接通信，而Zuul网关通过ServiceComb Discovery与ServiceCenter获取服务的实例信息（真实IP地址和端口等）

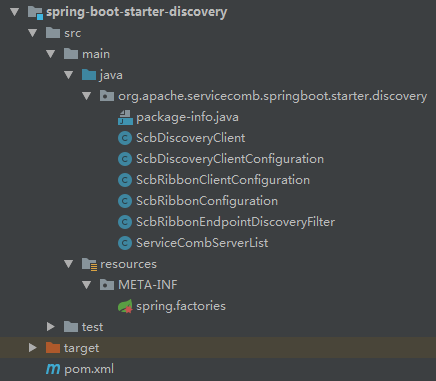


**ServiceComb Discovery源码分析**

目录结构

**源码从github上下载**

https://github.com/apache/servicecomb-java-chassis/tree/master/java-chassis-spring-boot/spring-boot-starter/spring-boot-starter-discovery



很明显这是一个自定义的SpringBoot的starter项目。  
关于SpringBoot自定义starter可以参考： Creating your own starter（https://docs.spring.io/spring-boot/docs/1.5.12.RELEASE/reference/htmlsingle/#boot-features-custom-starter）

springboot的starter项目一般是做自动配置（auto-configure）。那么以下按照starter项目的思路来分析。

**1. 自动配置入口**

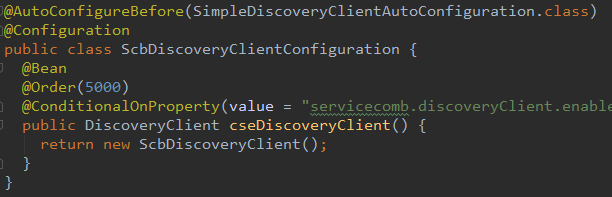
org.apache.servicecomb.springboot.starter.discovery包下面的ScbDiscoveryClientConfiguration类中和ScbRibbonConfiguration类中找到了AutoConfigureBefore和AutoConfigureAfter注解

* AutoConfigureBefore：类级别的注解，在指定类初始化配置之前自动执行当前配置类
* AutoConfigureAfter：类级别的注解，在指定类初始化配置之后自动执行当前配置类

**tips:**

查看spring.factories文件是否有org.springframework.boot.autoconfigure.EnableAutoConfiguration的配置或者包下面的类是否有@AutoConfigureBefore，@AutoConfigureAfter注解

**2. ScbDiscoveryClientConfiguration类**

这是一个Spring自动配置类，在这个类里只实例化了DiscoveryClient对象

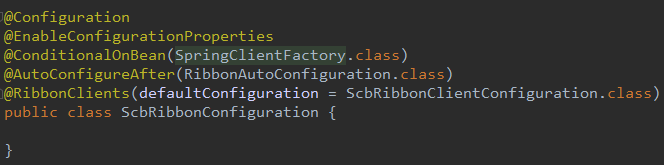
这个配置类只实例化了一个DiscoveryClient对象。

**重点来了，这个DiscoveryClient对象就是给Zuul使用的！！！**

最终会被Zuul的 DiscoveryClientRouteLocator 过滤器 用来加载路由定义信息。

**3. ScbRibbonConfiguration类**

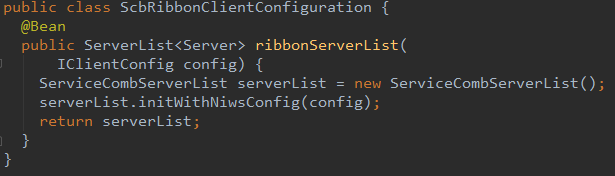
这是个Spring自动配置类，这个类是为了导入配置类RibbonAutoConfiguration.class



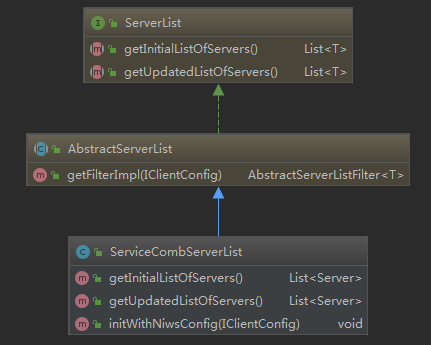
代码如上，没有任何实现，注意看注解@RibbonClients ，指定了ScbRibbonClientConfiguration类来配置RibbonClient。下面看这个类。

**4. ScbRibbonClientConfiguration类**

这个类是Spring配置类，实例化了Ribbon相关的Bean



ServerList是Ribbon框架的东西（Ribbon是客户端负载均衡框架）。ServerList是获取服务列表的接口  
ServiceCombServerList类继承于抽象类AbstractServerList，而AbstractServerList实现了ServerList接口

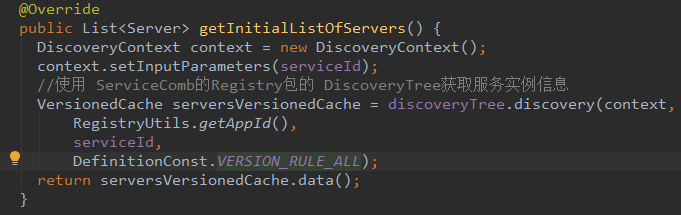


接着看下ServiceCombServerList做了什么事情。

**5. ServiceCombServerList 类**

这个类是给Ribbon使用，负责获取服务实例信息（真实IP地址和端口等）。

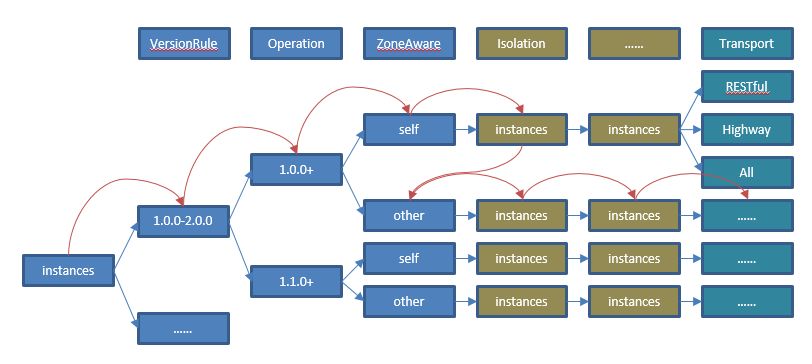
重点在下面这个方法，这个方法使用DiscoveryTree对象真正获取服务实例。最终返回的是可用微服务实例的真实ip地址和端口。  
DiscoveryTree是ServiceComb的service-registry包的，这个包是负责服务注册的。



DiscoveryTree的逻辑比较复杂，可以通过下面的处理流程了解其处理过程。

参考官方文档

https://docs.servicecomb.io/java-chassis/zh\_CN/references-handlers/loadbalance.html



感兴趣的同学可以研究下ServiceComb的service-registry包的实现，这里不展开分析。

**DiscoveryTree最终会调用到ServiceRegistryClientImpl.findServiceInstances方法**

**ServiceRegistryClientImpl.findServiceInstances**

在这个方法内直接调用ServiceCenter的rest接口Const.REGISTRY\_API.MICROSERVICE\_INSTANCES来获取相应的微服务实例信息（http://127.0.0.1:30100/v4/default/registry/instances）

