**Part III. Spring Cloud Netflix**

**11.服务发现：Eureka Clients**

服务发现是微服务框架中的一项重要原则。而如何对每一个client或他们之间的约束进行处理却是非常困难和不稳定的。Eureka作为Netflix服务发现的server和client。它的每个节点都可以复制自身的状态，并且相互注册服务，这样就保证了配置和部署的高可用性。

**11.1如何引入Eureka Client**

引入Eureka Client需要org.springframework.cloud组下的spring-cloud-starter-netflix-eureka-client 启动包。有关使用当前的Spring Cloud发布列表设置构建系统的详细信息，请参阅Spring Cloud项目页面。

**11.2 在Eureka上注册**

当一个服务节点在Eureka上注册时，它会向Eureka发送自己的元数据，其中包括主机地址,端口号，健康检查URL，主页等等。Eureka接收每一个服务注册实例的心跳信息。如果心跳信息在可配置的一定时间内发送失败，那么注册中心通常会将该实例移除。

Eureka client 示例：

*@Configuration*

*@ComponentScan*

*@EnableAutoConfiguration*

*@RestController*

**public** **class** Application {

*@RequestMapping("/")*

**public** String home() {

**return** "Hello world";

}

**public** **static** **void** main(String[] args) {

**new** SpringApplicationBuilder(Application.**class**).web(true).run(args);

}

}

（该示例是完全普通的一个Spring Boot工程）。通过引入spring-cloud-starter-netflix-eureka-client,你的工程会自动在Eureka服务器上注册，你需要做的只是在配置文件中定位Eureka服务器的位置，示例如下：

**application.yml.**

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

其中“defaultZone”表示服务节点提供服务的URL。

默认的应用工程名称（服务ID）,虚拟host和非安全端口分别由Environment中的${spring.application.name}, ${spring.application.name}和${server.port}属性决定。

引入spring-cloud-starter-netflix-eureka-client使得应用既是一个Eureka服务实例（即注册自己）也是一个客户端（即它能通过注册信息定位其他的服务）。The instance behaviour is driven by eureka.instance.\* configuration keys, but the defaults will be fine if you ensure that your application has a spring.application.name (this is the default for the Eureka service ID, or VIP).

有关可配置选项的更多详细信息，请参阅EurekaInstanceConfigBean和EurekaClientConfigBean。

禁止Eureka的服务发现可以将eureka.client.enabled设置为false;

**11.3 使用Eureka服务器进行身份验证**

如果eureka.client.serviceUrl.defaultZone中配置的一系列URL有凭证的嵌入（curl 风格，如<http://user:password@localhost:8761/eureka>），对于更为复杂的需求，你可以创建一个DiscoveryClientOptionalArgs的@Bean并且注入ClientFilter实例，所有这些都将应用于从客户端到服务器的调用。

注意： eureka client不支持每个eureka server身份验证方法不同, only the first set that are found will be used。

**11.4 状态页和健康指示器**

一个Eureka实例的状态页和健康指示器地址默认为“/info”和”/health”，同时也是Spring Boot Actuator 应用中一些有用端点的默认位置。如果你使用了非默认的上下文路径或者servlet路径(例如server.servletPath=/foo)或管理端点路径(例如 management.contextPath=/admin)，那么你需要更改配置信息，示例如下：

**application.yml.**

eureka:

instance:

statusPageUrlPath: ${management.context-path}/info

healthCheckUrlPath: ${management.context-path}/health

这些链接显示在客户端使用的元数据中，并在某些情况下用于决定是否将请求发送到应用程序，因此如果它们是准确配置的。

**11.5 注册一个安全的应用**

如果你的应用想通过https协议进行通信，你可以在EurekaInstanceConfig中设置两个标志位，即eureka.instance.[nonSecurePortEnabled,securePortEnabled]=[false,true]。这将使Eureka发布实例信息表明安全通信的优先级为最高。Spring Cloud 的DiscoveryClient将始终给服务返回以https开头的URI，并且Eureka（本机）实例信息将对URL进行安全健康检查。

由于Eureka内部的工作方式，它仍然会发布状态页和主页的非安全URL，除非你也明确地覆盖这些配置。你可以使用占位符来配置eureka实例URL，例如：

**application.yml.**

eureka:

instance:

statusPageUrl: https://${eureka.hostname}/info

healthCheckUrl: https://${eureka.hostname}/health

homePageUrl: https://${eureka.hostname}/

（注意，${eureka.hostname}是仅在Eureka较后版本中可用的本地占位符，你也可以使用Spring占位符实现同样的功能，例如使用${eureka.instance.hostName）。

如果您的应用程序通过代理服务器运行，并且SSL终端在代理中（例如，如果您运行在Cloud Foundry或其他平台的程序作为服务），则需要确保代理“转发”头部被应用程序拦截并处理。Spring Boot应用程序中，如果“X-Forwarded - \ \*”头被显示配置，嵌入式Tomcat容器会自动进行拦截处理。这个错误的一个标志是指向你自身应用的链接会是错误的。（错误的host，端口或协议）。

**11.6 Eureka的健康状态检查**

默认情况下，Eureka使用客户端心跳来判断客户端是否启动。除非有特别设定，否则Discovery Client将不会根据Spring Boot Actuator发送应用当前的健康检查状态。这意味着成功注册后Eureka将永远声明该应用处于“UP”状态。通过启用Eureka运行健康检查可以改变这种状况，从而将应用程序状态发送给Eureka。因此，每个其他应用将不会在“UP”之外的状态下将流量发送到当前应用。  
**application.yml.**

eureka:

client:

healthcheck:

enabled: true

eureka.client.healthcheck.enabled=true只能在application.yml中设置。在bootstrap.yml中对该值进行设置将导致意想不到的错误，譬如在Eureka上注册了未知的状态。

如果你需要对应用的健康检查有更多的控制，你可以自己实现com.netflix.appinfo.HealthCheckHandler

**11.7 实例和客户端的Eureka元数据**

花点时间了解Eureka元数据的工作原理是很有用的，so you can use it in a way that makes sense in your platform。元数据包括有主机名，IP地址，端口号，状态页和运行状况检查等标准元数据。这些信息被放在在服务注册表中，客户端通过服务注册表直接的访问服务，除此之外，额外的元数据也可以添加到注册实例的eureka.instance.metadataMap中，并且远程客户端也可以访问这些数据，但一般不会影响客户端的运行，除非客户端使用元数据作为有意义的数据进行处理。下面描述了几个特殊情况，其中Spring Cloud已经为metadata map指定了含义。

**11.7.3 更改Eureka实例ID**

一个vanilla Netflix Eureka实例注册后，ID与其主机名一一对应（即每个host只有一个服务）。Spring Cloud Eureka 提供了一个如下默认配置：

${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance\_id:${server.port}}}.

例如myhost:myappname:8080.

使用Spring Cloud你可以通过设置eureka.instance.instanceId为一个unique identifier覆盖这种默认配置。例如：

**application.yml.**

eureka:

instance:

instanceId: ${spring.application.name}:${vcap.application.instance\_id:${spring.application.instance\_id:${random.value}}}

通过这个元数据和在localhost上部署的多个服务实例，随机值会分配给各个实例使得每个实例都是唯一的。在Cloudfoundry中，vcap.application.instance\_id将在Spring Boot应用程序中自动填充，因此不需要随机值。

**11.8 使用EurekaClient**

你的应用一旦具有了服务发现功能，你就可以利用它通过Eureka Server找到其他的服务实例。其中一种方法是使用本地的com.netflix.discovery.EurekaClient（而不是Spring Cloud 的DiscoveryClient），例如：

@Autowired

private EurekaClient discoveryClient;

public String serviceUrl() {

InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);

return instance.getHomePageUrl();

}

不要在@PostConstruct方法或者@Scheduled方法中（或者ApplicationContext还未运行的任何地方）使用EurekaClient。它会在一个SmartLifecycle (with phase=0)被初始化，所以，当其他SmartLifecycle 的phase值比0大时，你才能使用这个EurekaClient。

**11.8.1 使用没有Jersey的EurekaClient**

EurekaClient默认使用Jersey作为HTTP通信，如果你想排除对Jersey的依赖，你可以在你的dependencies中排除。Spring Cloud会自动基于Spring的RestTemplate构建一个transport client。

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

<exclusions>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-client</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-core</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jersey.contribs</groupId>

<artifactId>jersey-apache-client4</artifactId>

</exclusion>

</exclusions>

</dependency>

**11.9 本机Netflix EurekaClient的替代方案**

您不必使用原生的Netflix EurekaClient，而且通常封装后使用它更为方便。Spring Cloud支持Feign（REST客户端构建器），还支持Spring RestTemplate使用logical Eureka service identifiers (VIPs)而不是物理URL。要使用固定的物理服务器列表配置Ribbon，您可以将<client>.ribbon.listOfServers设置为逗号分隔的物理地址（或主机名）列表，其中<client>是客户端的ID。

你还可以使用org.springframework.cloud.client.discovery.DiscoveryClient，它发现客户端提供了一个简单的接口而并不是只针对Netflix，例如：

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

}

**11.10 为什么注册一个服务这么慢？**

作为一个实例，需要定期发送心跳信息到注册中心（通过客户端的serviceUrl），默认持续时间为30秒。在服务实例，注册服务器和客户端在其本地缓存中都具有相同的元数据（因此可能需要3个心跳）之前，客户端才能发现服务。你可以通过更改eureka.instance.leaseRenewalIntervalInSeconds更改这段时长，这会加快客户端连接到其他服务的过程。在生产环境中，最好坚持使用默认值，因为服务器内部有一些计算可以make assumptions about the lease renewal period。

**11.11 区**

如果您已将Eureka客户端部署到多个区域，您可能希望这些客户端在使用另一个区域中的服务之前，首先调用同一区域的服务。为此，您需要正确配置您的Eureka客户端。

首先，您需要确保将Eureka服务器部署到每个区域，并且它们是彼此的对等体。有关详细信息，请参阅区域和区域部分 。

接下来，您需要告知Eureka您的服务所在的区域。您可以使用metadataMap属性来执行此操作。例如，如果service 1部署到zone 1和zone 2，则需要在service 1中设置以下Eureka属性

**Service 1 in Zone 1**

eureka.instance.metadataMap.zone = zone1

eureka.client.preferSameZoneEureka = true

**Service 1 in Zone 2**

eureka.instance.metadataMap.zone = zone2

eureka.client.preferSameZoneEureka = true

**12 服务发现：Eureka服务器**

**12.1 如何include Eureka服务器**

要在项目中include Eureka服务器，请使用组org.springframework.cloud下组件id为 spring-cloud-starter-eureka-server的启动器。

**12.2 如何运行 Eureka服务器**

Eureka服务器示例：

@SpringBootApplication

@EnableEurekaServer

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

服务器有一个带UI的主页， /eureka/\*下的各HTTP API端点就是Eureka的各个功能。

**12.3 高可用，区域和地区**

Eureka服务器没有后端存储，但是注册表中的服务实例都必须向Eureka服务器发送心跳来更新注册信息（可以在服务器内存中完成）。同时，客户端还具有eureka注册的内存缓存（因此，在请求一个服务时，没有必要每次都去eureka服务器上查找注册表信息）。

默认情况下，每个Eureka服务器也是一个Eureka客户端，并且至少需要一个Eureka服务器注册地址完成注册。虽然不配置这个注册地址服务依旧能够运行，但是日志中会有大量的没有注册的提示信息。

**12.4 独立模式**

一个eureka向自己注册时，便成了单点部署，这种模式叫做独立模式，在独立模式下，你可能更加关注他作为客户端的作用。配置示例如下：  
**application.yml (Standalone Eureka Server).**

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

serviceUrl:

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

注意到 serviceUrl指向了本地的实例地址。

**12.5 同伴意识**

通过eureka实例之间的相互注册，eureka可以变得更加可靠。事实上，这种配置是默认的，你所需要做的就是向其他eureka实例提供有效的serviceUrl。示例如下：

**application.yml (Two Peer Aware Eureka Servers).**

---

spring:

profiles: peer1

eureka:

instance:

hostname: peer1

client:

serviceUrl:

defaultZone: http://peer2/eureka/

---

spring:

profiles: peer2

eureka:

instance:

hostname: peer2

client:

serviceUrl:

defaultZone: http://peer1/eureka/

在这个例子中，通过使用不同的Spring profiles，我们可以让相同的server在这两个不同host上运行。在这种配置下，通过设置host，将peer1和peer2绑定相同的域名，可以测试eureka的这种“同伴意识”。事实上，如果运行的服务器知道自己的hostname（默认从java.net.InetAddress中取），eureka.instance.hostname是不需要配置的。

你可以在一个系统中添加多个peers，只要它们能通过至少一条注册关系联系起来，它们就能互相同步自身保存的注册表信息。如果某些peers被物理隔离了，整个系统也能在脑裂下继续正常运行。

**12.6 优先使用IP地址**

Eureka推荐使用IP地址而不是域名作为注册地址。当应用向eureka注册时,将eureka.instance.preferIpAddress设置为true，就能使用IP地址而不是域名来注册。

注意：如果java代码中没有指定hostname，那么Eureka就会使用其ip地址。你也可以在程序运行时设定hostname的环境变量，如eureka.instance.hostname=${HOST\_NAME}

**13 熔断器:Hystrix Clients**

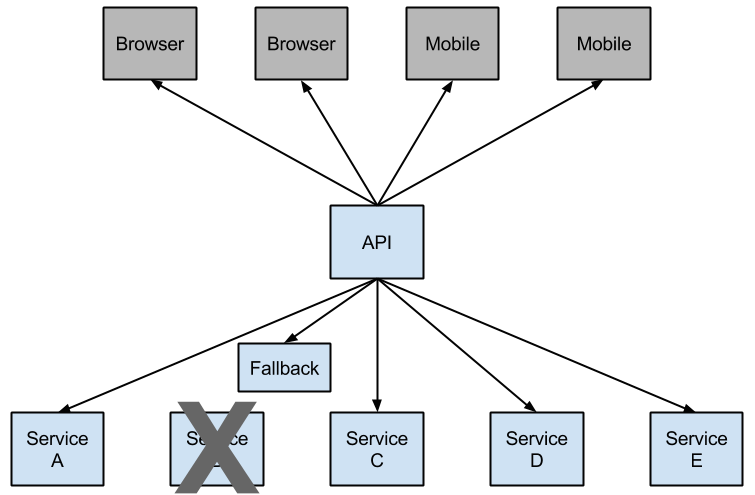
Netflix公司开发了Hystrix，它实现了熔断器模式。在微服务框架中，多层的服务调用十分常见。

**Figure 13.1. Microservice Graph**



底层服务的失效会引起连锁的调用失败，从而直接影响到用户层。当调用某个服务时，如果在 metrics.rollingStats.timeInMilliseconds中定义的时间内（默认为10秒）调用超过circuitBreaker.requestVolumeThreshold中设定的次数（默认为20次），并且调用失败率大于circuitBreaker.errorThresholdPercentage中设定的值（默认为50%），熔断器就会开启。在哪些情况下开启断路器可以有开发者设定。

**Figure 13.2. Hystrix fallback prevents cascading failures**



熔断器机制有效的防止调用失败的连锁反应，让一些过载或者失效的服务能够有时间恢复。服务降级（fallback）可以是一个Hystrix的一个保护性调用，一个静态数据或者一个合理的空值。Fallback也会产生级联，所以第一个fallback会做一些业务调用，而这些调用也会通过fallback返回静态数据。

**13.1 怎么引入Hystrix**

引入hystrix需要org.springframework.cloud组下的组件id为spring-cloud-starter-netflix-hystrix的启动包。

Example boot app:

@SpringBootApplication

@EnableCircuitBreaker

public class Application {

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

@Component

public class StoreIntegration {

@HystrixCommand(fallbackMethod = "defaultStores")

public Object getStores(Map<String, Object> parameters) {

//do stuff that might fail

}

public Object defaultStores(Map<String, Object> parameters) {

return /\* something useful \*/;

}

}

@HystrixCommand是Netflix发布的"javanica"库中的一个注解。Spring Cloud通过该注解，自动将其封装到了Spring beans中，从而作为Hystrix熔断器的代理，熔断器通过判断选择是否开启或关闭调用线路，并且设定调用失败后的动作。

配置@HystrixCommand，你可以在commandProperties中使用一系列的注解@HystrixProperty来进行配置，例如：  
@HystrixCommand(commandProperties = {

@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "500")

})

public User getUserById(String id) {

return userResource.getUserById(id);

}

**13.2 安全上下文的传播机制或者Spring作用域配置**

如果你想将一些thread local上下文传递到@HystrixCommand中，默认情况下是无法实现的。这是因为command是在线程池中进行的（在超时情况下）。你可以让Hystrix使用和调用者相同的线程，或者直接更改隔离策略，比如：

*@HystrixCommand(fallbackMethod = "stubMyService",*

*commandProperties = {*

*@HystrixProperty(name="execution.isolation.strategy", value="SEMAPHORE")*

*}*

*)*

...

同样的，如果你使用过@SessionScope(该针对每一次HTTP请求都会产生一个新的bean，同时该bean仅在当前HTTP session内有效)或者@RequestScope(每次请求都会new一个bean，但是跟prototype不同的地方在于，request在这个请求内有效，而prototype返回给客户端后就不问不管了)。你就会知道你这样做是因为，运行时异常会找不到作用域的上下文。

你还可以配置hystrix.shareSecurityContext为true。这样做，将会让Hystris自动配置一个并发策略插件，当使用Hystris命令从主线程迁移SecurityContext时自动同步。Hystris不允许多个并发策略同时存在，所以需要自己扩展一个HystrixConcurrencyStrategy注册到Spring中。Spring Cloud会自动在Spring上下文中发现你的实现类，并自动包装成一个自定义Hystrix插件。

**13.3 健康指示器**

断路器的状态也可以通过/health接口被应用查看到。

**{**

"hystrix": **{**

"openCircuitBreakers": **[**

"StoreIntegration::getStoresByLocationLink"

]**,**

"status": "CIRCUIT\_OPEN"

**},**

"status": "UP"

**}**

**13.4 Hystrix Metrics Stream**

使用Hystrix metrics stream需要引入依赖：spring-boot-starter-actuator。将会提供一个/hystrix.stream管理接口。

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

**14.** **Hystrix仪表板**

Hystrix中一个主要的特性，就是对每一条HystrixCommand操作的各个方面都可以度量检测。 Hystrix通过一个仪表板页面来有效的显示每一个断路器的健康状态。

**Figure 14.1. Hystrix Dashboard**



**15.** **Hystrix超时和Ribbon Clients**

当使用封装了Ribbon clients的Hystrix commands时，要保证Hystrix超时时间要设置成比Ribbon超时时间长一点，时间要包括有可能的重试调用。举例来说，假如Ribbon连接的超时时间为1秒，而且ribbon clients有可能重试三次请求，那么Hystrix的超时时间就要比3秒长一点。

**15.1** **如何引入Hystrix仪表板**

引入Hystrix需要在工程中引入相关starter： group：org.springframework.cloud artifact id :spring-cloud-starter-hystrix-dashboard。

可以通过@EnableHystrixDashboard注解来开启仪表板。然后，就可以通过/hystrix访问仪表板页面；而且仪表板相关数据，也会通过/hystrix.stream接口提供给客户端应用。

**15.2 Turbine**

其实对于一个实例个体来说，Hystrix的数据对于系统整体健康程度来说并不十分重要。 Turbine提供了一个仪表板页面来聚合所有/hystrix.stream相关数据，同时把所有数据合成到一个/turbine.stream接口中, 运行Turbine非常简单，只需要在主类中加上一个@EnableTurbine注解。（当然，前提是已经引入spring-cloud-starter-turbine），Turbine完整配置属性请参见 WIKI。 有一点不同的是，turbine.instanceUrlSuffix不再需要添加端口配置（端口会自动处理）。如果需要手动配置，需要设置turbine.instanceInsertPort=false。

注意：默认情况下，Turbine会根据Eureka中实例的hostname和port来发现他们的/hystrix.stream接口，然后上报给/hystrix.stream。如果一个实例的元数据中包含了management.port。port值就会替代/hystrix.stream作为接口url。设置management.port的配置如下：

eureka:

instance:

metadata-map:

management.port: ${management.port:8081}

turbine.appConfig配置成Eureka的serviceIds集合，Turbine就会使用这个属性来发现实例。 Turbine会在Hystrix仪表板页面使用一个链接：http://my.turbine.sever:8080/turbine.stream?cluster=<CLUSTERNAME>，如果集群名字是default，那可以省略参数cluster。参数cluster必须对应着turbine.aggregator.clusterConfig配置的一个实体。参数cluster值为大写字符，因此配置的时候需要注意。

turbine:

aggregator:

clusterConfig: CUSTOMERS

appConfig: customers

如果你不想再turbine.aggregator.clusterConfig中写死这个值，那么你可以在Turbine提供的Turbine Clusters Provider这个bean中进行配置。

可以在根实例的InstanceInfo中设置turbine.clusterNameExpression,这样clusterName可以使用SPEL表达式来配置。 默认值是appName，意味着Eureka的serviceId是以集群key作为命名的。（例如:customers的InstanceInfo有一个CUSTOMERS的appName）。例子：

turbine:

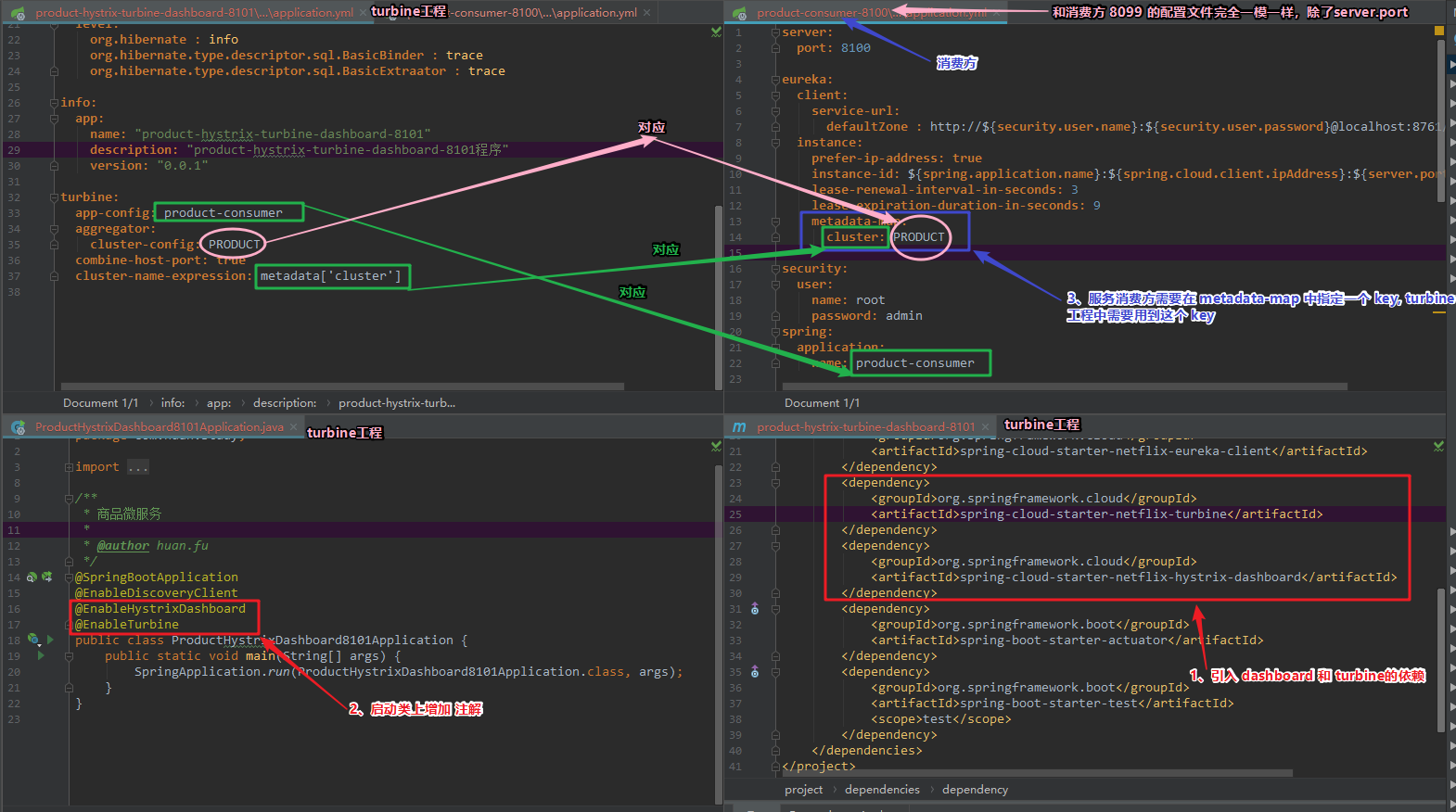
aggregator:

clusterConfig: SYSTEM,USER

appConfig: customers,stores,ui,admin

clusterNameExpression: metadata['cluster']

在这个例子中，cluster name是从四个服务的metadatamap中得到的，期望获得的值为“SYSTEM”和“USER”。



如果需要使用"default"作为所有应用的集群名，需要一个字符串直译表达式（在YAML中是双引号加上一层单引号）：

turbine:

appConfig: customers,stores

clusterNameExpression: "'default'"

Spring Cloud提供了一个spring-cloud-starter-turbine，其中包含了Turbine服务端运行时的所有依赖。然后，只需要在Spring Boot应用中加上@EnableTurbine就行。

注意： 默认情况下，Spring Cloud 允许Turbine使用地址 + 端口来逐个处理主机、集群的。 如果想要在Turbine中使用Netflix的策略（不允许处理多主机多集群）（实例ID就是主机名），那可以设置：turbine.combineHostPort=false。

**15.3 Turbine Stream**

标准Turbine的策略，是从各个分布式节点中拉取Hystrix命令来得到相关数据。但是在某些场景中（比如：PaaS），这个策略就不行了。如果想让Hystrix命令执行时主动推送数据到Turbine，并且让Spring Cloud能够处理这些消息。那可以在客户端增加一个spring-cloud-netflix-hystrix-stream依赖，并在提供的诸多spring-cloud-starter-stream-\*中，选择一个。

在服务端(被监控端)，只需要在Spring Boot 中通过注解@EnableTurbineStream来开启就行，默认情况下会启用8989端口。如果需要的话，可以通过server.port或者turbine.stream.port来指定端口。如果已经使用了spring-boot-starter-web和spring-boot-starter-actuator,那可以通过management.port来指定端口。

可以设置Hystrix仪表板使用Turbine Stream Server来替代Hystrix Stream。 假设Turbine Stream运行在主机名：myhost 的8989端口，那么可以通过http://myhost:8989作为Hystrix仪表板的数据来源。链路中各节点将使用<serviceId>.<name>作为节点名。

Spring Cloud 提供了一个spring-cloud-starter-turbine-stream，其包括了Turbine Stream服务运行时所有的依赖。使用时，只需要选择一个Stream实现即可。例如：spring-cloud-starter-stream-rabbit(这个是基于最新Netty实现的，所以需要java 8)

参考：<https://windmt.com/2018/04/17/spring-cloud-6-turbine/>

**16 客户端负载均衡：Ribbon**

Ribbon是一个HTTP/TCP客户端的负载均衡实现。Feign已经集成了Ribbon，因此，如果你已经使用@FeignClient那么，你可以直接使用Ribbon。

Ribbon中有个核心概念：客户端命名。每一个负载均衡器都是指集群中处理某个请求的一组组件，开发者需要给每一个这样的小组进行命名。（比如使用@FeignClient注解） Spring Cloud使用了一种新的方式来分组，在业务处理端在ApplicationContext中注册一个RibbonClientConfiguration,通过这个RibbonClientConfiguration来进行分组配置。这其中就会包含一个ILoadBalancer,一个RestClient,以及一个ServerListFilter。

**16.1 如何引入Ribbon**

引入Ribbon，需要在工程中增加如下依赖：group：org.springframework.cloud artifact id:spring-cloud-starter-ribbon。如何构建Spring Cloud 工程，参见Spring Cloud文档。

**16.2 配置Ribbon客户端**

可以通过扩展配置属性：<client>.ribbon.\*，来配置Ribbon客户端相关属性，这和直接使用Netflix API没什么区别。除此之外，还可以通过Spring Boot配置文件来配置Ribbon。 配置属性可以参见CommonClientConfigKey（在ribbon-core中）。

Spring Cloud也允许通过@RibbonClient对Ribbon进行完全的控制（基于RibbonClientConfiguration）。 例如：

*@Configuration*

*@RibbonClient(name = "foo", configuration = FooConfiguration.class)*

**public** **class** TestConfiguration {

}

在这个例子中，FooConfiguration中的配置会整合到RibbonClientConfiguration中，并会覆盖之前的参数值。

注意：需要确保FooConfiguration不会被@Configuration或者@ComponentScan扫描到，否则，就会被所有@RibbonClients共享了（自动注入）。如果使用@ComponentScan或者@SpringBootApplication时，也需要避免FooConfiguration被自动扫描。

Spring Cloud Netflix默认情况下为Ribbon自动生成了一下bean:  
bean类型 bean名称 类名

IClientConfig ribbonClientConfig: DefaultClientConfigImpl

IRule ribbonRule: ZoneAvoidanceRule

IPing ribbonPing: DummyPing

ServerList<Server> ribbonServerList: ConfigurationBasedServerList

ServerListFilter<Server> ribbonServerListFilter: ZonePreferenceServerListFilter

ILoadBalancer ribbonLoadBalancer: ZoneAwareLoadBalancer

ServerListUpdater ribbonServerListUpdater: PollingServerListUpdater

在@RibbonClient配置的配置类（如上例中的 FooConfiguration），允许对上面表格中的Bean进行自定义。例如：

*@Configuration*

**protected** **static** **class** FooConfiguration {

*@Bean*

**public** ZonePreferenceServerListFilter serverListFilter() {

ZonePreferenceServerListFilter filter = **new** ZonePreferenceServerListFilter();

filter.setZone("myTestZone");

**return** filter;

}

*@Bean*

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

}

这个例子使用PingUrl替换NoOpPing并提供了一个特定的serverListFilter

**16.3 为所有的Ribbon客户端设置默认配置**

使用@RibbonClients注解可以提供客户端的默认配置。默认配置的例子如下：

*@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)*

**public** **class** RibbonClientDefaultConfigurationTestsConfig {

**public** **static** **class** BazServiceList **extends** ConfigurationBasedServerList {

**public** BazServiceList(IClientConfig config) {

**super**.initWithNiwsConfig(config);

}

}

}

*@Configuration*

**class** DefaultRibbonConfig {

*@Bean*

**public** IRule ribbonRule() {

**return** **new** BestAvailableRule();

}

*@Bean*

**public** IPing ribbonPing() {

**return** **new** PingUrl();

}

*@Bean*

**public** ServerList<Server> ribbonServerList(IClientConfig config) {

**return** **new** RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);

}

*@Bean*

**public** ServerListSubsetFilter serverListFilter() {

ServerListSubsetFilter filter = **new** ServerListSubsetFilter();

**return** filter;

}

}

**16.4 使用配置属性定制化Ribbon客户端**

从1.2.0版本开始，Spring Cloud Netflix支持使用属性对于Ribbon客户端进行配置。属性名同Ribbon文档中一样。

这样应用启动时可以在不同的环境中使用不同的Ribbon配置信息。  
配置项只需要在下列配置前加上<clientName>.ribbon.前缀就行：

NFLoadBalancerClassName: should implement ILoadBalancer

NFLoadBalancerRuleClassName: should implement IRule

NFLoadBalancerPingClassName: should implement IPing

NIWSServerListClassName: should implement ServerList

NIWSServerListFilterClassName should implement ServerListFilter

注意：这些属性指定的类，会覆盖使用@RibbonClient(configuration=MyRibbonConfig.class)注解中的定义，同时也会覆盖Spring Cloud Netflix的默认策略。

例如，在users服务中设置IRule，可以按照如下方式配置：

**application.yml.**

users:

ribbon:

NIWSServerListClassName: com.netflix.loadbalancer.ConfigurationBasedServerList

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.WeightedResponseTimeRule

**16.5 在Eureka中使用Ribbon**

当需要在Eureka中整合Ribbon，可以在Eureka中配置DiscoveryEnabledNIWSServerList来覆盖Ribbon的服务列表ribbonServerList。也可以配置NIWSDiscoveryPing来替代Ribbon的IPing（Eureka用其判断服务是否可用）。 默认情况下Ribbon的服务列表会被替换成DomainExtractingServerList,这样就可以达到负载均衡使用配置的元数据而不是去使用AWS AMI元数据。默认情况下，在实例的元数据中，服务列表会被封装成一个"区域"信息。（因此，在远程客户端中可以设置eureka.instance.metadataMap.zone） 当发生错误时，可以从服务端获取一个域名作为代理区域（需要打开approximateZoneFromHostname配置项）。默认的，区域中的客户端也会以同样的方式来作为远程实例。如：eureka.instance.metadataMap.zone。

注意：按"archaius"规范，使用一个"@zone"来设置客户端区域,Spring Cloud将会优先使用这个策略（需要在YAML中配置）

注意：如果不指定区域，那就会按照客户端配置信息中去推测。通过eureka.client.availabilityZones对区域名进行映射，使用第一个列表中的第一个作为区域信息。（例如：eureka.client.region默认就是"us-east-1"）

**16.6如何不在Eureka下使用Ribbon**

如果不想在客户端手动指定服务端，Eureka是一个很方便的服务发现方案。但是如果你就是不想用它，Ribbon和Feign也是可以使用的。假设，你已经为"stores"服务定义好一个@RibbonClient（不使用Eureka），在Ribbon客户端可以额外的配置一个服务列表，如下：

**application.yml.**

stores:

ribbon:

listOfServers: example.com,google.com

**16.7在Eureka中关闭Ribbon**

可以配置ribbon.eureka.enabled = false可以在Eureka中关闭Ribbon。例如：

**application.yml.**

ribbon:

eureka:

enabled: false

**16.8 直接操作Ribbon API**

你也可以直接使用LoadBalancerClient：

**public** **class** MyClass {

*@Autowired*

**private** LoadBalancerClient loadBalancer;

**public** **void** doStuff() {

ServiceInstance instance = loadBalancer.choose("stores");

URI storesUri = URI.create(String.format("http://%s:%s", instance.getHost(), instance.getPort()));

*// ... do something with the URI*

}

}

**16.9 Ribbon配置的缓存**

在spring Cloud中，每一个命名了的Ribbon客户端都有其对应的子应用上下文，这个上下文只有在第一次请求该ribbon客户端时才会被加载。这种懒加载机制也可以通过配置改为提前加载（指定Ribbon客户端名字时进行加载）。

**application.yml.**

ribbon:

eager-load:

enabled: true

clients: client1, client2, client3

**16.10 如何配置Hystrix thread pools**

如果你将zuul.ribbonIsolationStrategy 改为THREAD，Hystrix的线程隔离策略将会应用于所有路由。在这种情况下，HystrixThreadPoolKey 默认为“RibbonCommand”。这意味着，所有路由的HystrixCommand都会在相同的Hystrix线程池中执行，可使用以下配置，让每个路由使用独立的线程池：

**application.yml.**

zuul:

threadPool:

useSeparateThreadPools: true

使用如上配置后，默认的HystrixThreadPoolkey 将与每个路由的服务标识相同。如果你想HystrixThreadPoolKey 添加前缀，可使用类似如下的配置：

**application.yml.**

zuul:

threadPool:

useSeparateThreadPools: true

threadPoolKeyPrefix: zuulgw

**16.11 How to Provide a Key to Ribbon’s IRule**

如果你想自己实现IRule（用于选择怎样的负载均衡算法）来实现特殊的路由需求，你可以在IRule的choose方法中传一些信息：

**com.netflix.loadbalancer.IRule.java.**

public interface IRule{

public Server choose(Object key);

:

你可以通过实现IRule提供一些信息用于选择目标服务器：

RequestContext.getCurrentContext()

.set(FilterConstants.LOAD\_BALANCER\_KEY, "canary-test");

如果你将key为FilterConstants.LOAD\_BALANCER\_KEY的对象传入RequestContext，这个对象会被传递到IRule实现类的choose方法中。上述代码必须在RibbonRoutingFilter之前运行。放在Zuul的pre filter中是最好的。在pre filter中你可以通过RequestContext轻松获取到http头和请求参数。所以，它可以用来判定是否将LOAD\_BALANCER\_KEY传递给Ribbon。如果在RequestContext中没有给LOAD\_BALANCER\_KEY赋值，那么null会作为参数传递到choose方法中。