Spring Cloud 原理

邋遢的流浪剑客 Java后端 2019-12-21

作者 | 邋遢的流浪剑客

链接 | blog.csdn.net/qq_40378034/article/details/86552677

之前一直在看《Spring Cloud微服务实战》，最近又看了公众号石衫的架构笔记的《拜托！面试请不要再问我Spring Cloud底层原理》，对Spring Cloud的主要组件的原理有了更深的理解，特地做一下总结

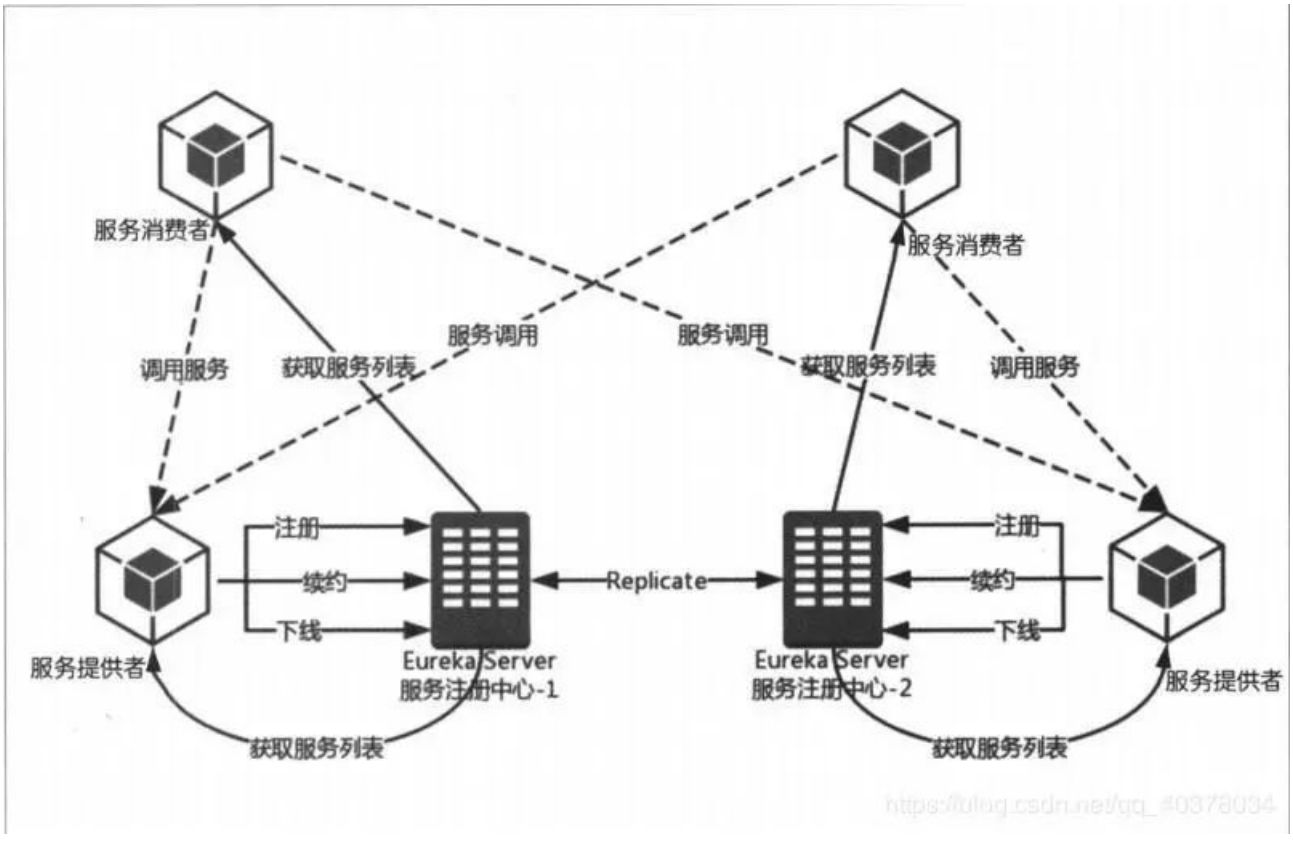
(1)Netflix Eureka

1）、Eureka服务端：也称服务注册中心，同其他服务注册中心一样，支持高可用配置。如果Eureka以集群模式部署，当集群中有分片出现故障时，那么Eureka就转入自我保护模式。它允许在分片故障期间继续提供服务的发现和注册，当故障分片恢复运行时，集群中其他分片会把它们的状态再次同步回来

2）、Eureka客户端：主要处理服务的注册与发现。客户端服务通过注解和参数配置的方式，嵌入在客户端应用程序的代码中，在应用程序运行时，Eureka客户端想注册中心注册自身提供的服务并周期性地发送心跳来更新它的服务租约。同时，它也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性地刷新服务状态

3)、Eureka Server的高可用实际上就是将自己作为服务向其他注册中心注册自己，这样就可以形成一组互相注册的服务注册中心，以实现服务清单的互相同步，达到高可用效果

(2)Eureka详解



1）、服务提供者

A. 服务注册

服务提供者在启动的时候会通过发送REST请求的方式将自己注册到Eureka Server上，同时带上了自己服务的一些元数据信息。Eureka Server接收到这个REST请求之后，将元数据信息存储在一个双层结构Map中，其中第一层的key是服务名，第二层的key是具体服务的实例名

B.服务同步

两个服务提供者分别注册到了两个不同的服务注册中心上，也就是说，它们的信息分别被两个服务注册中心所维护。此时，由于服务注册中心之间因互相注册为服务，当服务提供者发送注册请求到一个服务注册中心时，它会将该请求转发给集群中相连的其他注册中心，从而实现注册中心之间的服务同步。通过服务同步，两个服务提供者的服务信息就可以通过这两台服务注册中心中的任意一台获取到

C.服务续约

在注册完服务之后，服务提供者会维护一个心跳用来持续告诉Eureka Server：“我还活着”，以防止Eureka Server的剔除任务将该服务实例从服务列表中排除出去，我们称该操作为服务续约

```
# 定义服务续约任务的调用间隔时间，默认30秒
eureka.instance.lease-renewal-interval-in-seconds=30
# 定义服务失效的时间，默认90秒
eureka.instance.lease-expiration-duration-in-seconds=90
```

2）、服务消费者

A.获取服务

当我们启动服务消费者的时候，它会发送一个REST请求给服务注册中心，来获取上面注册的服务清单。为了性能考虑，Eureka Server会维护一份只读的服务清单来返回给客户端，同时该缓存清单会每隔30秒更新一次

```
# 缓存清单的更新时间，默认30秒
eureka.client.registry-fetch-interval-seconds=30
```

B.服务调用

服务消费者在获取服务清单后，通过服务名可以获得具体提供服务的实例名和该实例的元数据信息。在Ribbon中会默认采用轮询的方式进行调用，从而实现客户端的负载均衡

对于访问实例的选择，Eureka中有Region和Zone的概念，一个Region中可以包含多个Zone，每个服务客户端需要被注册到一个Zone中，所以每个客户端对应一个Region和一个Zone。在进行服务调用的时候，优先访问同处一个一个Zone中的服务提供方，若访问不到，就访问其他的Zone

C.服务下线

当服务实例进行正常的关闭操作时，它会触发一个服务下线的REST请求给Eureka Server，告诉服务注册中心：“我要下线了”。服务端在接收到请求之后，将该服务状态置为下线（DOWN），并把该下线事件传播出去

3）、服务注册中心

A.失效剔除

Eureka Server在启动的时候会创建一个定时任务，默认每隔一段时间（默认为60秒）将当前清单中超时（默认为90秒）没有续约的服务剔除出去

B.自我保护

在服务注册中心的信息面板中出现红色警告信息：

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

该警告就是触发了Eureka Server的自我保护机制。Eureka Server在运行期间，会统计心跳失败的比例在15分钟之内是否低于85%，如果出现低于的情况，Eureka Server会将当前的实例注册信息保护起来，让这些实例不会过期，尽可能保护这些注册信息。但是，在这段保护期间内实例若出现问题，那么客户端很容易拿到实际已经不存在的服务实例，会出现调用失败的情况，所以客户端必须要有容错机制，比如可以使用请求重试、断路器等机制

```
# 关闭保护机制，以确保注册中心可以将不用的实例正确剔除（本地调试可以使用，线上不推荐）
eureka.server.enable-self-preservation=false
```

Ribbon是一个基于HTTP和TCP的客户端负载均衡器，它可以在通过客户端中配置的ribbonServerList服务端列表去轮询访问以达到服务均衡的作用。当Ribbon和Eureka联合使用时，Ribbon的服务实例清单RibbonServerList会被DiscoveryEnabledNIWSServerList重写，扩展成从Eureka注册中心中获取服务端列表。同时它也会用NIWSDiscoveryPing来取代IPing，它将职责委托给Eureka来去定服务端是否已经启动

在客户端负载均衡中，所有客户端节点都维护着自己要访问的服务端清单，而这些服务端的清单来自于服务注册中心（比如Eureka）。在客户端负载均衡中也需要心跳去维护服务端清单的健康性，只是这个步骤需要与服务注册中心配合完成

通过Spring Cloud Ribbon的封装，我们在微服务架构中使用客户端负载均衡调用只需要如下两步：

- 服务提供者只需要启动多个服务实例并且注册到一个注册中心或是多个相关联的服务注册中心
- 服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate来实现面向服务的接口调用

Feign的关键机制是使用了动态代理

1）、首先，对某个接口定义了@FeignClient注解，Feign就会针对这个接口创建一个动态代理

2）、接着调用接口的时候，本质就是调用Feign创建的动态代理

3）、Feign的动态代理会根据在接口上的@RequestMapping等注解，来动态构造要请求的服务的地址

4）、针对这个地址，发起请求、解析响应

Feign是和Ribbon以及Eureka紧密协作的

1)、首先Ribbon会从Eureka Client里获取到对应的服务注册表，也就知道了所有的服务都部署在了哪些机器上，在监听哪些端口

2）、然后Ribbon就可以使用默认的Round Robin算法，从中选择一台机器

3）、Feign就会针对这台机器，构造并发起请求

在微服务架构中，存在着那么多的服务单元，若一个单元出现故障，就很容易因依赖关系而引发故障的蔓延，最终导致整个系统的瘫痪，这样的架构相较传统架构更加不稳定。为了解决这样的问题，产生了断路器等一系列的服务保护机制

在分布式架构中，当某个服务单元发生故障之后，通过断路器的故障监控，向调用方返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延

Hystrix具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能

Hystrix使用舱壁模式实现线程池的隔离，它会为每一个依赖服务创建一个独立的线程池，这样就算某个依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而不会拖慢其他的依赖服务

Spring Cloud Zuul通过与Spring Cloud Eureka进行整合，将自身注册为Eureka服务治理下的应用，同时从Eureka中获得了所有其他微服务的实例信息

对于路由规则的维护，Zuul默认会将通过以服务名作为ContextPath的方式来创建路由映射

Zuul提供了一套过滤器机制，可以支持在API网关无附上进行统一调用来对微服务接口做前置过滤，已实现对微服务接口的拦截和校验

- **Eureka**：

  各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里
- **Ribbon**：

  服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台
- **Feign**：

  基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求
- **Hystrix**：

  发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题
- **Zuul**：

  如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

-END-

推荐阅读

1. 955 不加班的公司名单：955.WLB

2. 8 岁上海小学生B站教编程惊动苹果

3. 我在华为做外包的真实经历！

4. 什么是一致性 Hash 算法？

5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

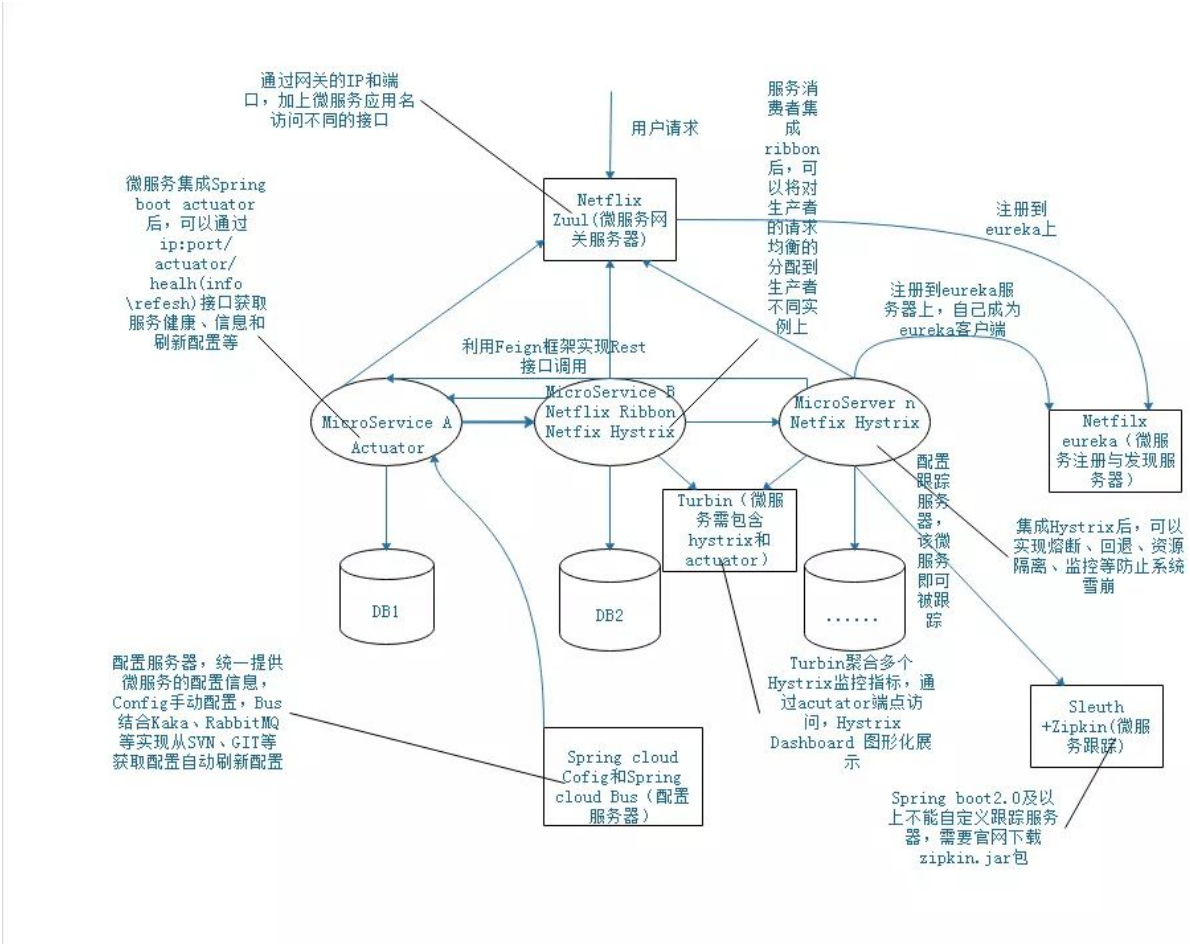
一张图了解 Spring Cloud 微服务架构

SimpleEasy Java后端 2019-10-20

作者 | SimpleEasy

链接 | www.jianshu.com/p/84d2824980fe

Spring cloud作为当下主流的微服务框架，让我们实现微服务架构简单快捷，Spring cloud中各个组件在微服务架构中扮演的角色如下图所示，黑线表示注释说明，蓝线由A指向B，表示B从A处获取服务。



由上图所示微服务架构大致由上图的逻辑结构组成，其包括各种微服务、注册发现、服务网关、熔断器、统一配置、跟踪服务等。下面说说Spring cloud中的组件分别充当其中的什么角色。

Fegin(接口调用)：微服务之间通过Rest接口通讯，Spring Cloud提供Feign框架来支持Rest的调用，Feign使得不同进程的Rest接口调用得以用优雅的方式进行，这种优雅表现得就像同一个进程调用一样。

Netflix eureka(注册发现)：微服务模式下，一个大的Web应用通常都被拆分为很多比较小的web应用(服务)，这个时候就需要有

一个地方保存这些服务的相关信息，才能让各个小的应用彼此知道对方，这个时候就需要在注册中心进行注册。每个应用启动时向配置的注册中心注册自己的信息（ip地址，端口号，服务名称等信息），注册中心将他们保存起来，服务间相互调用的时候，通过服务名称就可以到注册中心找到对应的服务信息，从而进行通讯。注册与发现服务为微服务之间的调用带来了方便，解决了硬编码的问题。服务间只通过对方的服务id，而无需知道其ip和端口即可以获取对方方服务。

Ribbon(负载均衡)：Ribbon是Netflix发布的负载均衡器，它有助于控制HTTP和TCP客户端的行为。为Ribbon，配置服务提供者的地址列表后，Ribbon就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为Ribbon实现自定义的负载均衡算法。在SpringCloud中，当Ribbon与Eureka配合使用时，Ribbon可自动从EurekaServer获取服务提供者的地址列表，并基于负载均衡算法，请求其中一个服务提供者的实例（为了服务的可靠性，一个微服务可能部署多个实例）。

tips：大家可以关注微信公众号：Java后端，获取更多优秀博文推送。

Hystrix(熔断器)：当服务提供者响应非常缓慢，那么消费者对提供者的请求就会被强制等待，直到提供者响应或超时。在高负载场景下，如果不做任何处理，此类问题可能会导致服务消费者的资源耗竭甚至整个系统的崩溃（雪崩效应）。Hystrix正是为了防止此类问题发生。Hystrix是由Netflix开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。Hystrix主要通过以下几点实现延迟和容错。

包裹请求：使用HystrixCommand（或HystrixObservableCommand）包裹对依赖的调用逻辑，每个命令在独立线程中执行。这使用了设计模式中的“命令模式”。

跳闸机制：当某服务的错误率超过一定阈值时，Hystrix可以自动或者手动跳闸，停止请求该服务一段时间。

资源隔离：Hystrix为每个依赖都维护了一个小型的线程池（或者信号量）。如果该线程池已满，发往该依赖的请求就被立即拒绝，而不是排队等候，从而加速失败判定。

监控：Hystrix可以近乎实时地监控运行指标和配置的变化，例如成功、失败、超时和被拒绝的请求等。

回退机制：当请求失败、超时、被拒绝，或当断路器打开时，执行回退逻辑。回退逻辑可由开发人员指定。

Zuul(微服务网关)　：不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求。例如一个电影购票的手机APP，可能调用多个微服务的接口才能完成一次购票的业务流程，如果让客户端直接与各个微服务通信，会有以下的问题：

客户端会多次请求不同的微服务，增加了客户端的复杂性。

存在跨域请求，在一定场景下处理相对复杂。

认证复杂，每个服务都需要独立认证。

难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将很难实施。

某些微服务可能使用了对防火墙/浏览器不友好的协议，直接访问时会有一定的困难。

以上问题可借助微服务网关解决。微服务网关是介于客户端和服务器端之间的中间层，所有的外部请求都会先经过微服务网关。使用微服务网关后，微服务网关将封装应用程序的内部结构，客户端只用跟网关交互，而无须直接调用特定微服务的接口。这样，开发就可以得到简化。不仅如此，使用微服务网关还有以下优点：

易于监控。可在微服务网关收集监控数据并将其推送到外部系统进行分析。

易于认证。可在微服务网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。

减少了客户端与各个微服务之间的交互次数。

Spring cloud bus（统一配置服务）： 对于传统的单体应用，常使用配置文件管理所有配置。例如一个SpringBoot开发的单体应用，可将配置内容放在application.yml文件中。如果需要切换环境，可设置多个Profile，并在启动应用时指定spring.profiles.active={profile}。然而，在微服务架构中，微服务的配置管理一般有以下需求：

集中管理配置。一个使用微服务架构的应用系统可能会包含成百上千个微服务，因此集中管理配置是非常有必要的。

不同环境，不同配置。例如，数据源配置在不同的环境（开发、测试、预发布、生产等）中是不同的。

运行期间可动态调整。例如，可根据各个微服务的负载情况，动态调整数据源连接池大小或熔断阈值，并且在调整配置时不停止微服务。

配置修改后可自动更新。如配置内容发生变化，微服务能够自动更新配置。综上所述，对于微服务架构而言，一个通用的配置管理机制是必不可少的，常见做法是使用配置服务器管理配置。Spring cloud bus利用Git或SVN等管理配置、采用Kafka或者RabbitMQ等消息总线通知所有应用，从而实现配置的自动更新并且刷新所有微服务实例的配置。

Sleuth+ZipKin(跟踪服务)： Sleuth和Zipkin结合使用可以通过图形化的界面查看微服务请求的延迟情况以及各个微服务的依赖情况。需要注意的是Spring boot2及以上不在支持Zipkin的自定义，需要到官方网站下载ZipKin相关的jar包。

另外需要提一点的是**Spring boot actuator,** 提供了很多监控端点如/actuator/info、/actuator/health、/acutator/refresh等，可以查看微服务的信息、健康状况、刷新配置等。

-END-

推荐阅读

1. 感受 Lambda 之美，推荐收藏，需要时查阅
2. 如何优雅的导出 Excel
3. 文艺互联网公司 vs 二逼互联网公司
4. Java 开发中常用的 4 种加密方法
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

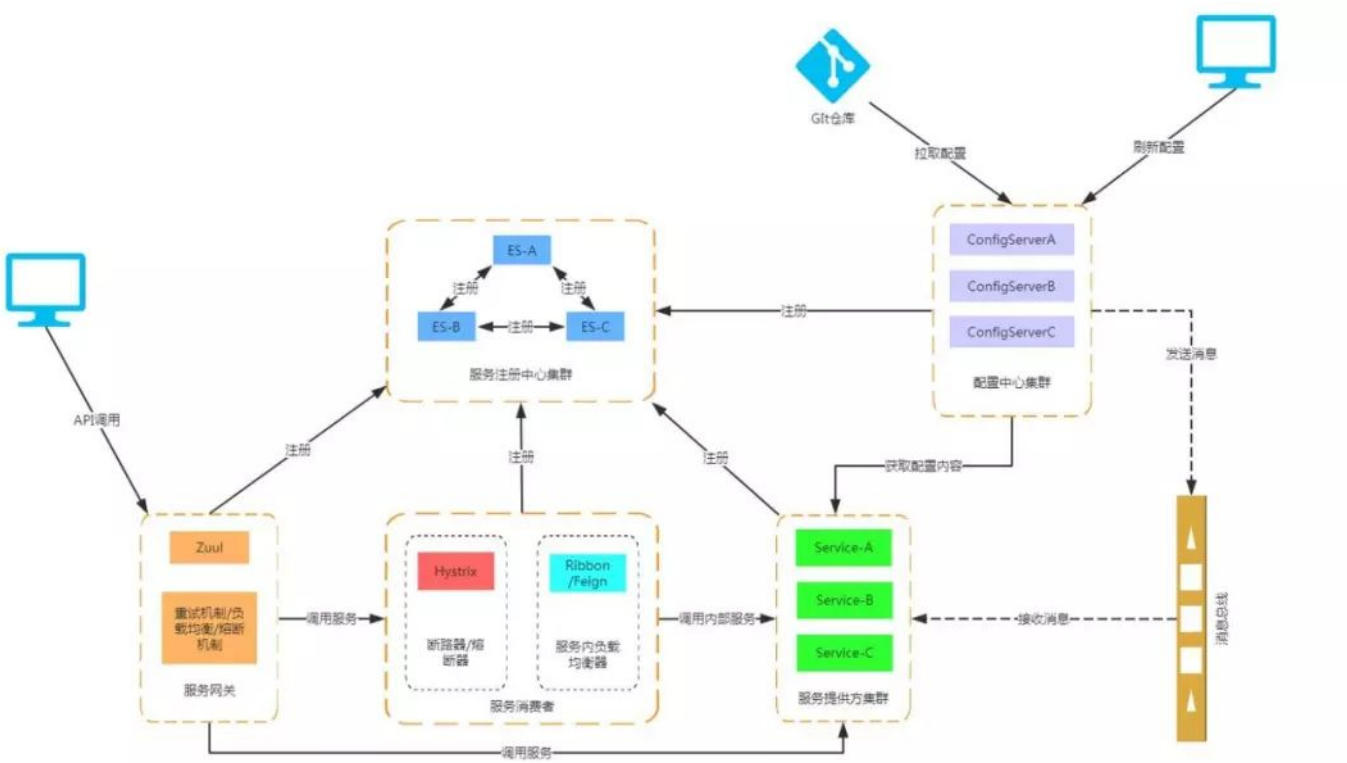
厉害！ Spring Cloud 20000 字总结！

FrancisQ Java后端 2月25日

作者 | FrancisQ

链接 | juejin.im/post/5de2553e5188256e885f4fa3

首先我给大家看一张图，如果大家对这张图有些地方不太理解的话，我希望你们看完我这篇文章会恍然大悟。



什么是 Spring cloud

> 构建分布式系统不需要复杂和容易出错。Spring Cloud 为最常见的分布式系统模式提供了一种简单且易于接受的编程模型，帮助开发人员构建有弹性的、可靠的、协调的应用程序。Spring Cloud 构建于 Spring Boot 之上，使得开发者很容易入手并快速应用于生产中。

官方果然官方，介绍都这么有板有眼的。

我所理解的Spring Cloud就是微服务系统架构的一站式解决方案，在平时我们构建微服务的过程中需要做如**服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控**等操作，而 Spring Cloud 为我们提供了一套简易的编程模型，使我们能在 Spring Boot 的基础上轻松地实现微服务项目的构建。

Spring Cloud 的版本

当然这个只是个题外话。

Spring Cloud 的版本号并不是我们通常见的数字版本号，而是一些很奇怪的单词。这些单词均为英国伦敦地铁站的站名。同时根

据字母表的顺序来对应版本时间顺序，比如：最早 的 Release 版本 Angel，第二个 Release 版本 Brixton（英国地名），然后是 Camden、 Dalston、Edgware、Finchley、Greenwich、Hoxton。

Spring Cloud 的服务发现框架——Eureka

> Eureka是基于REST（代表性状态转移）的服务，主要在AWS云中用于定位服务，以实现负载均衡和中间层服务器的故障转移。我们称此服务为Eureka服务器。Eureka还带有一个基于Java的客户端组件Eureka Client，它使与服务的交互变得更加容易。客户端还具有一个内置的负载平衡器，可以执行基本的循环负载平衡。在Netflix，更复杂的负载均衡器将Eureka包装起来，以基于流量，资源使用，错误条件等多种因素提供加权负载均衡，以提供出色的弹性。

总的来说，Eureka 就是一个服务发现框架。何为服务，何又为发现呢？

举一个生活中的例子，就比如我们平时租房子找中介的事情。

在没有中介的时候我们需要一个一个去寻找是否有房屋要出租的房东，这显然会非常的费力，一你找凭一个人的能力是找不到很多房源供你选择，再者你也懒得这么找下去(找了这么久，没有合适的只能将就)。**这里的我们就相当于微服务中的Consumer，而那些房东就相当于微服务中的Provider。消费者Consumer需要调用提供者Provider提供的一些服务，就像我们现在需要租他们的房子一样。**

但是如果只是租客和房东之间进行寻找的话，他们的效率是很低的，房东找不到租客赚不到钱，租客找不到房东住不了房。所以，后来房东肯定就想到了广播自己的房源信息(比如在街边贴贴小广告)，这样对于房东来说已经完成他的任务(将房源公布出去)，但是有两个问题就出现了。第一、其他不是租客的都能收到这种租房消息，这在现实世界没什么，但是在计算机的世界中就会出现**资源消耗**的问题了。第二、租客这样还是很难找到你，试想一下我需要租房，我还需要东一个西一个地去找街边小广告，麻不麻烦？

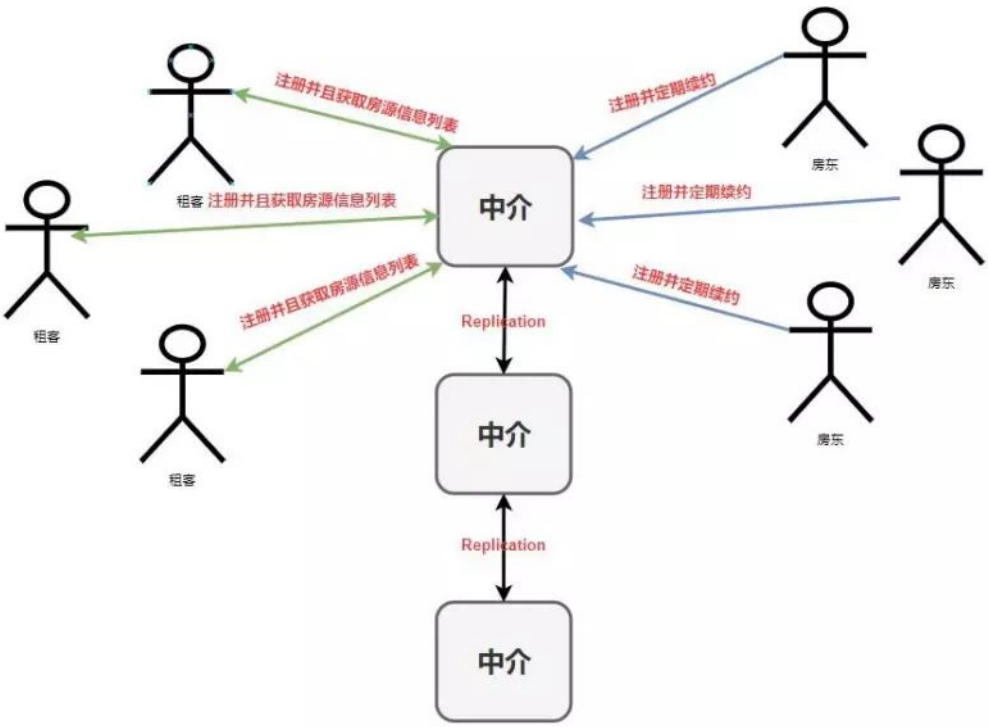
那怎么办呢？我们当然不会那么傻乎乎的，第一时间就是去找**中介**呀，它为我们提供了统一房源的地方，我们消费者只需要跑到它那里去找就行了。而对于房东来说，他们也只需要把房源在中介那里发布就行了。

那么现在，我们的模式就是这样的了。

但是，这个时候还会出现一些问题。

1. 房东注册之后如果不想卖房子了怎么办？我们是不是需要让房东**定期续约**？如果房东不进行续约是不是要将他们从中介那里的注册列表中**移除**。
2. 租客是不是也要进行**注册**呢？不然合同乙方怎么来呢？
3. 中介可不可以做**连锁店**呢？如果这一个店因为某些不可抗力因素而无法使用，那么我们是否可以换一个连锁店呢？

针对上面的问题我们来重新构建一下上面的模式图



好了，举完这个🌰我们就可以来看关于 Eureka 的一些基础概念了，你会发现这东西理解起来怎么这么简单。👇👇👇

服务发现：其实就是一个“中介”，整个过程中有三个角色：**服务提供者(出租房子的)、服务消费者(租客)、服务中介(房屋中介)**。

服务提供者：就是提供一些自己能够执行的一些服务给外界。

服务消费者：就是需要使用一些服务的“用户”。

服务中介：其实就是服务提供者和服务消费者之间的“桥梁”，服务提供者可以把自己注册到服务中介那里，而服务消费者如需要消费一些服务(使用一些功能)就可以在服务中介中寻找注册在服务中介的服务提供者。

服务注册 Register:

官方解释:当 Eureka 客户端向 [Eureka] Server注册时，它提供自身的**元数据**，比如IP地址、端口，运行状况指示符URL，主页等。

结合中介理解:房东 (提供者 [Eureka] Client Provider)在中介 (服务器[Eureka] Server) 那里登记房屋的信息,比如面积,价格，地段等等(元数据metaData)。

服务续约 Renew:

官方解释：**Eureka 客户会每隔30秒(默认情况下)发送一次心跳来续约**。通过续约来告知[Eureka] Server该 Eureka 客户仍然存在，没有出现问题。正常情况下，如果[Eureka] Server在90秒没有收到 Eureka 客户的续约，它会将实例从其注册表中删除。

结合中介理解:房东 (提供者 [Eureka] Client Provider) 定期告诉中介 (服务器[Eureka] Server) 我的房子还租(续约) ,中介 (服务器[Eureka] Server) 收到之后继续保留房屋的信息。

获取注册列表信息 Fetch Registries:

官方解释:Eureka 客户端从服务器获取注册表信息,并将其缓存在本地。客户端会使用该信息查找其他服务,从而进行远程调用。该注册列表信息定期(每30秒钟)更新一次。每次返回注册列表信息可能与 Eureka 客户端的缓存信息不同, Eureka 客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配,Eureka 客户端则会重新获取整个注册表信息。Eureka 服务器缓存注册列表信息,整个注册表以及每个应用程序的信息进行了压缩,压缩内容和没有压缩的内容完全相同。Eureka 客户端和 Eureka 服务器可以使用JSON / XML格式进行通讯。在默认的情况下 Eureka 客户端使用压缩JSON格式来获取注册列表的信息。

结合中介理解：租客(消费者[Eureka] Client Consumer) 去中介 (服务器[Eureka] Server) 那里获取所有的房屋信息列表 (客户端列表[Eureka] Client List) ，而且租客为了获取最新的信息会定期向中介 (服务器[Eureka] Server) 那里获取并更新本地列表。

服务下线 Cancel：

官方解释：Eureka客户端在程序关闭时向Eureka服务器发送取消请求。发送请求后，该客户端实例信息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：DiscoveryManager.getInstance().shutdownComponent();

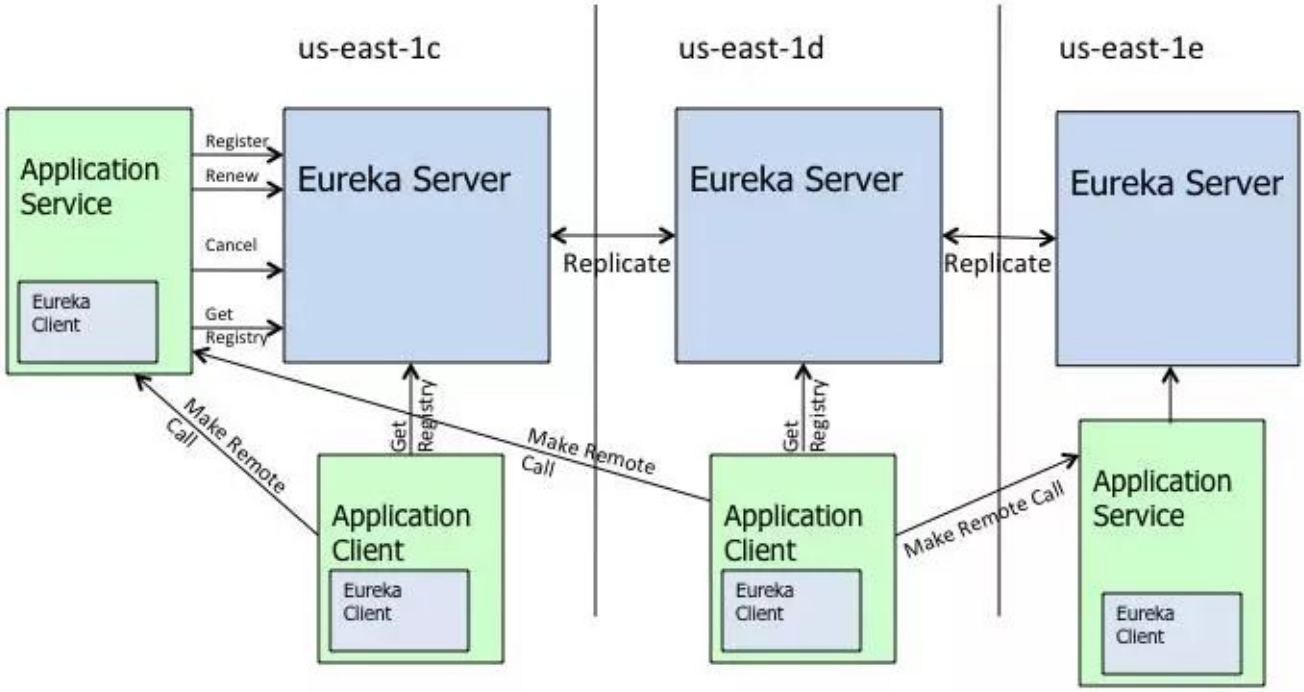
结合中介理解：房东 (提供者[Eureka] Client Provider) 告诉中介 (服务器[Eureka] Server) 我的房子不租了，中介之后就将注册的房屋信息从列表中剔除。

服务剔除 Eviction：

官方解释：在默认的情况下，**当Eureka客户端连续90秒(3个续约周期)没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除**，即服务剔除。

结合中介理解：房东(提供者[Eureka] Client Provider) 会定期联系 中介 (服务器[Eureka] Server) 告诉他我的房子还租(续约)，如果中介 (服务器[Eureka] Server) 长时间没收到提供者的信息，那么中介会将他的房屋信息给下架(服务剔除)。

下面就是Netflix官方给出的 Eureka 架构图，你会发现和我们前面画的中介图别无二致。



当然，可以充当服务发现的组件有很多：Zookeeper，Consul， Eureka 等。

更多关于 Eureka 的知识(自我保护，初始注册策略等等)可以自己去官网查看，或者查看我的另一篇文章 [深入理解 Eureka](https://juejin.im/post/5dd497e3f265da0ba7718018)。

负载均衡之 Ribbon

什么是 RestTemplate?

不是讲Ribbon么？怎么扯到了RestTemplate了？你先别急，听我慢慢道来。

我不听我不听我不听🙈🙈🙈。

我就说一句！**RestTemplate是Spring提供的一个访问Http服务的客户端类**，怎么说呢？就是微服务之间的调用是使用的RestTemplate。比如这个时候我们 消费者B 需要调用 提供者A 所提供的服务我们就需要这么写。如我下面的伪代码

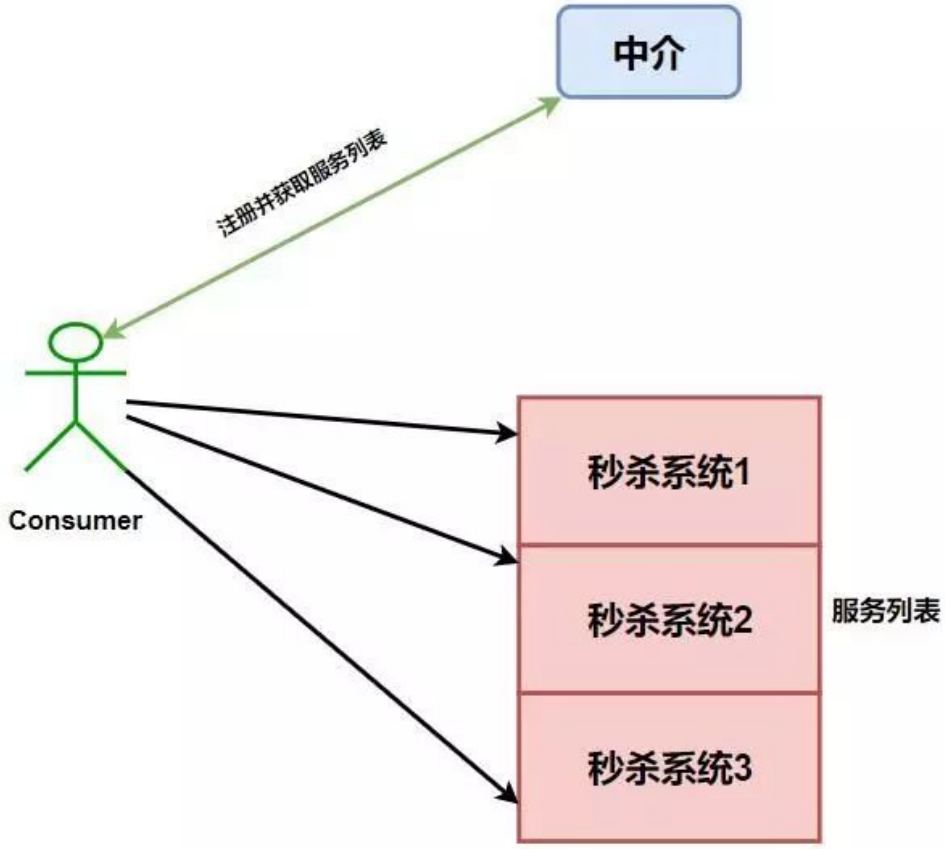
```
@Autowired
private RestTemplate restTemplate;
// 这里是提供者A的ip地址，但是如果使用了 Eureka 那么就应该是提供者A的名称
private static final String SERVICE_PROVIDER_A = "http://localhost:8081";

@PostMapping("/judge")
public boolean judge(@RequestBody Request request) {
  String url = SERVICE_PROVIDER_A + "/service1";
  return restTemplate.postForObject(url, request, Boolean.class);
}
```

如果你对源码感兴趣的话，你会发现上面我们所讲的 Eureka 框架中的**注册**、**续约**等，底层都是使用的RestTemplate。

为什么需要 Ribbon?

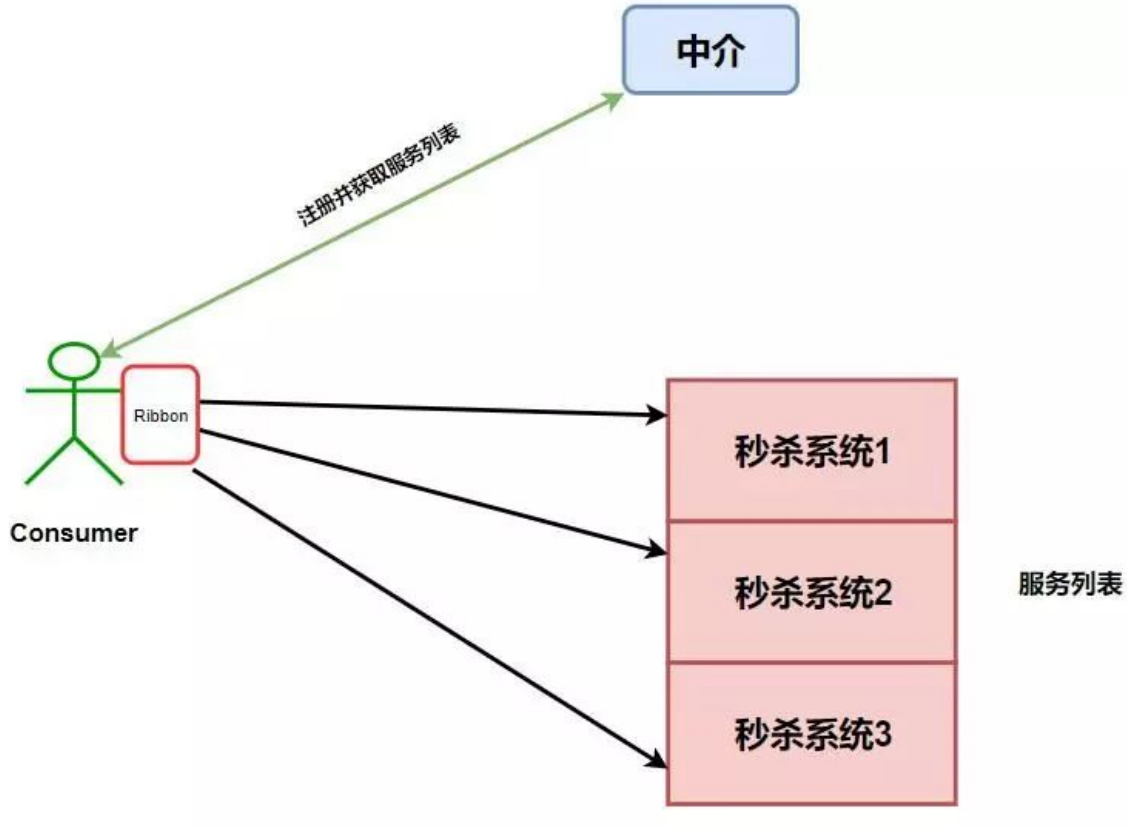
Ribbon 是Netflix公司的一个开源的负载均衡 项目，是一个客户端/进程内负载均衡器，**运行在消费者端**。

我们再举个🌰，比如我们设计了一个秒杀系统，但是为了整个系统的 **高可用**，我们需要将这个系统做一个集群，而这个时候我们消费者就可以拥有多个秒杀系统的调用途径了，如下图。



如果这个时候我们没有进行一些**均衡操作**，如果我们对秒杀系统1进行大量的调用，而另外两个基本不请求，就会导致秒杀系统1崩溃，而另外两个就变成了傀儡，那么我们为什么还要做集群，我们高可用体现的意义又在哪呢？

所以Ribbon出现了，注意我们上面加粗的几个字——**运行在消费者端**。指的是，Ribbon是运行在消费者端的负载均衡器，如下图。



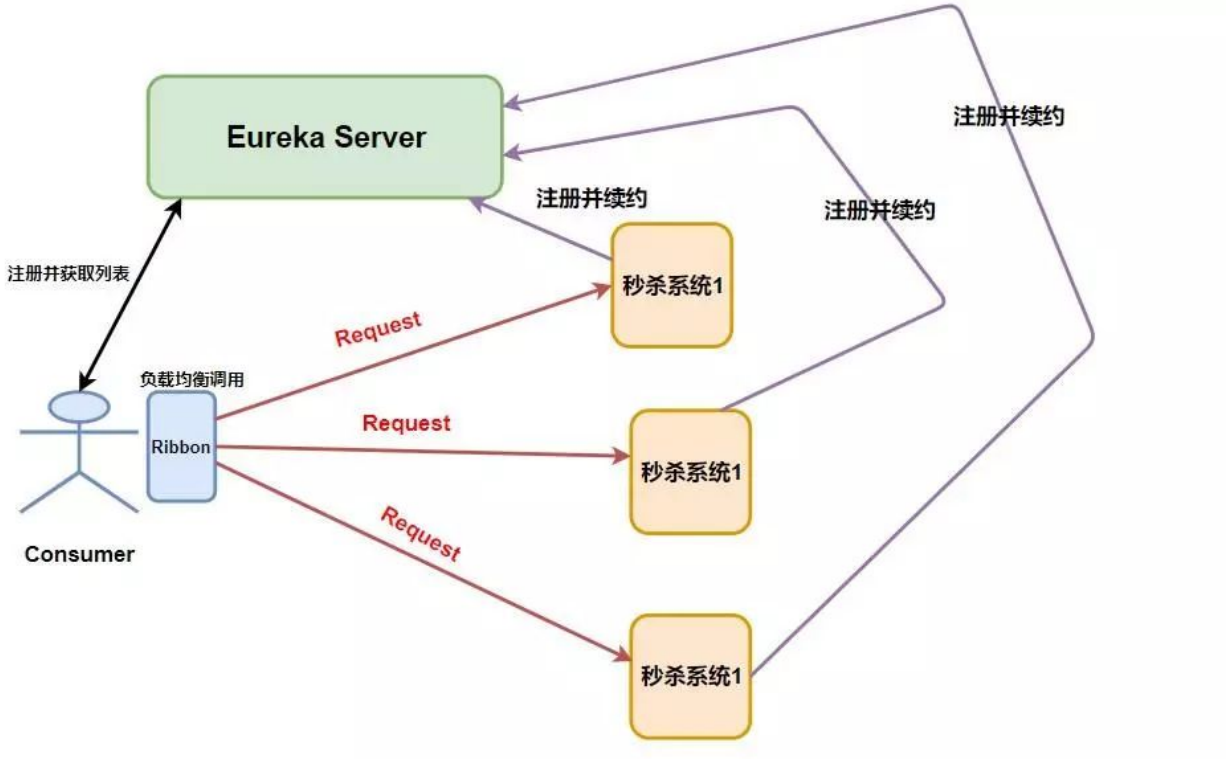
其工作原理就是Consumer端获取到了所有的服务列表之后，在其**内部**使用**负载均衡算法**，进行对多个系统的调用。

Nginx 和 Ribbon 的对比

提到**负载均衡**就不得不提到大名鼎鼎的Nignx了，而和Ribbon不同的是，它是一种**集中式**的负载均衡器。

何为集中式呢？简单理解就是**将所有请求都集中起来，然后再进行负载均衡**。如下图。

我们可以看到Nginx是接收了所有的请求进行负载均衡的，而对于Ribbon来说它是在消费者端进行的负载均衡。如下图。



> 请注意Request的位置，在Nginx中请求是先进入负载均衡器，而在Ribbon中是先在客户端进行负载均衡才进行请求的。

Ribbon 的几种负载均衡算法

负载均衡，不管Nginx还是Ribbon都需要其算法的支持，如果我没记错的话Nginx使用的是 轮询和加权轮询算法。而在Ribbon中有更多的负载均衡调度算法，其默认是使用的RoundRobinRule轮询策略。

- **RoundRobinRule**：轮询策略。Ribbon默认采用的策略。若经过一轮轮询没有找到可用的provider，其最多轮询 10 轮。若最终还没有找到，则返回 null。
- **RandomRule**: 随机策略，从所有可用的 provider 中随机选择一个。
- **RetryRule**: 重试策略。先按照 RoundRobinRule 策略获取 provider，若获取失败，则在指定的时限内重试。默认的时限为 500 毫秒。

🐂🐂🐂 还有很多，这里不一一举🌰了，你最需要知道的是默认轮询算法，并且可以更换默认的负载均衡算法，只需要在配置文件中做出修改就行。

```
providerName:
 ribbon:
  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

当然，在Ribbon中你还可以**自定义负载均衡算法**，你只需要实现IRule接口，然后修改配置文件或者自定义Java Config类。

什么是 Open Feign

有了 Eureka，RestTemplate，Ribbon我们就可以😄愉快地进行服务间的调用了，但是使用 RestTemplate还是不方便，我们每次都要进行这样的调用。

```
@Autowired
private RestTemplate restTemplate;
// 这里是提供者A的ip地址，但是如果使用了 Eureka 那么就应该是提供者A的名称
private static final String SERVICE_PROVIDER_A = "http://localhost:8081";

@PostMapping("/judge")
public boolean judge(@RequestBody Request request) {
  String url = SERVICE_PROVIDER_A + "/service1";
  // 是不是太麻烦了？？？ 每次都要 url、请求、返回类型的
  return restTemplate.postForObject(url, request, Boolean.class);
}
```

这样每次都调用RestRemplate的API是否太麻烦，我能不能像**调用原来代码一样进行各个服务间的调用呢?**

聪明的小朋友肯定想到了，那就用**映射**呀，就像域名和IP地址的映射。我们可以将被调用的服务代码映射到消费者端，这样我们就可以**“无缝开发”**啦。

> OpenFeign 也是运行在消费者端的，使用 Ribbon 进行负载均衡，所以 OpenFeign 直接内置了 Ribbon。

在导入了Open Feign之后我们就可以进行愉快编写 Consumer端代码了。

```
// 使用 @FeignClient 注解来指定提供者的名字
@FeignClient(value = "eureka-client-provider")
public interface TestClient {
  // 这里一定要注意需要使用的是提供者那端的请求相对路径，这里就相当于映射了
  @RequestMapping(value = "/provider/xxx",
  method = RequestMethod.POST)
  CommonResponse<List<Plan>> getPlans(@RequestBody planGetRequest request);
}
```

然后我们在Controller就可以像原来调用Service层代码一样调用它了。

```
@RestController
public class TestController {
  // 这里就相当于原来自动注入的 Service
  @Autowired
  private TestClient testClient;
  // controller 调用 service 层代码
  @RequestMapping(value = "/test", method = RequestMethod.POST)
  public CommonResponse<List<Plan>> get(@RequestBody planGetRequest request) {
    return testClient.getPlans(request);
  }
}
```

必不可少的 Hystrix

什么是 Hystrix之熔断和降级

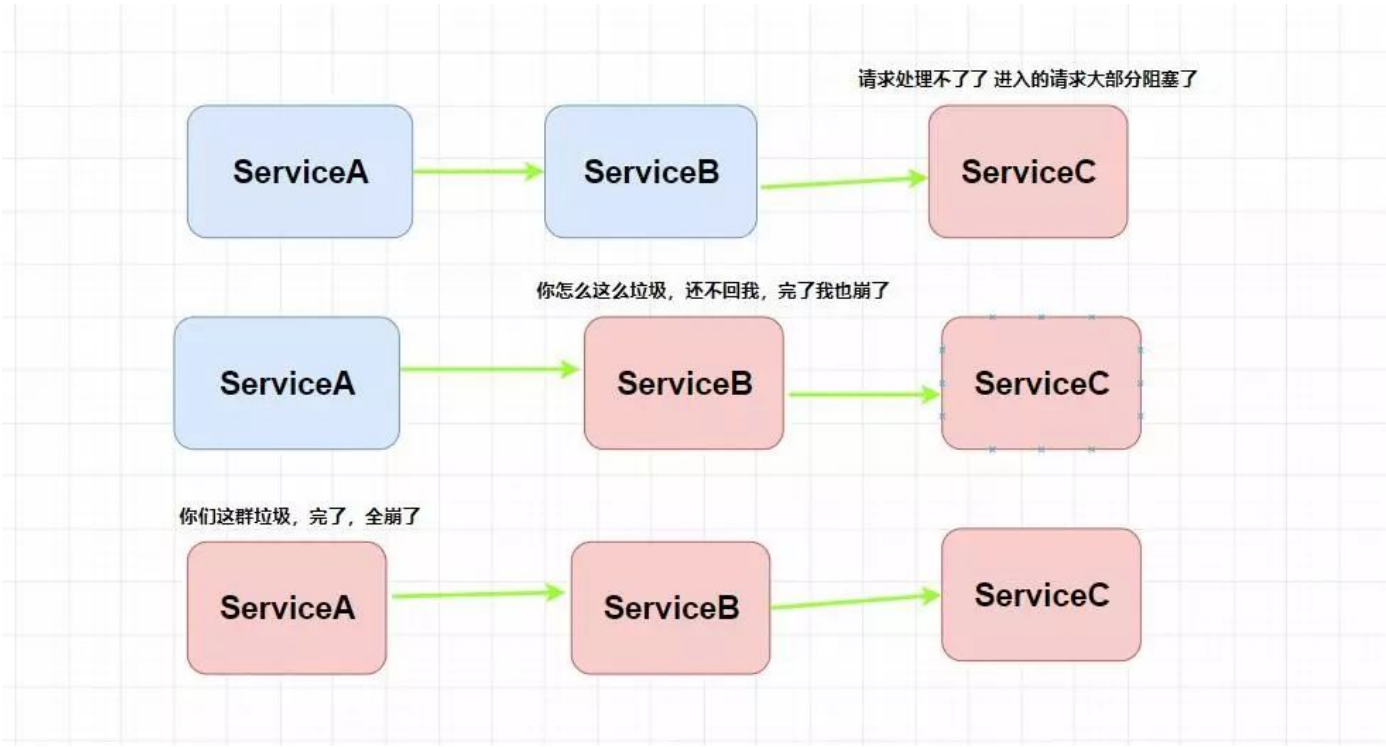
> 在分布式环境中，不可避免地会有许多服务依赖项中的某些失败。Hystrix是一个库，可通过添加等待时间容限和容错逻辑来帮助您控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点，停止服务之间的级联故障并提供后备选项来实现此目的，所有这些都可以提高系统的整体弹性。

总体来说[Hystrix]就是一个能进行**熔断**和**降级**的库，通过使用它能提高整个系统的弹性。

那么什么是　熔断和降级　呢?再举个⌓,此时我们整个微服务系统是这样的。服务A调用了服务B,服务B再调用了服务C,但是因为某些原因，服务C顶不住了，这个时候大量请求会在服务C阻塞。

服务C阻塞了还好，毕竟只是一个系统崩溃了。但是请注意这个时候因为服务C不能返回响应，那么服务B调用服务C的的请求就会阻塞，同理服务B阻塞了，那么服务A也会阻塞崩溃。

> 请注意，为什么阻塞会崩溃。因为这些请求会消耗占用系统的线程、IO 等资源，消耗完你这个系统服务器不就崩了么。



这就叫**服务雪崩**。妈耶，上面两个**熔断**和**降级**你都没给我解释清楚，你现在又给我扯什么**服务雪崩**？😨😨😨

别急，听我慢慢道来。

不听我也得讲下去！

所谓**熔断**就是服务雪崩的一种有效解决方案。当指定时间窗内的请求失败率达到设定阈值时，系统将通过**断路器**直接将此请求链路断开。

也就是我们上面服务B调用服务C在指定时间窗内，调用的失败率到达了一定的值，那么[Hystrix]则会自动将 服务B与C 之间的请求都断了，以免导致服务雪崩现象。

其实这里所讲的**熔断**就是指的[Hystrix]中的**断路器模式**，你可以使用简单的@[Hystrix]Command注解来标注某个方法，这样[Hystrix]就会使用**断路器**来“包装”这个方法，每当调用时间超过指定时间时(默认为1000ms)，断路器将会中断对这个方法的调用。

当然你可以对这个注解的很多属性进行设置，比如设置超时时间，像这样。

```
@HystrixCommand(
  commandProperties = {@HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",value = "1200")}
)
public List<Xxx> getXxxx() {
  //...省略代码逻辑
}
```

但是，我查阅了一些博客，发现他们都将**熔断**和**降级**的概念混淆了，以我的理解，**降级是为了更好的用户体验，当一个方法调用异常时，通过执行另一种代码逻辑来给用户友好的回复**。这也就对应着[Hystrix]的**后备处理**模式。你可以通过设置fallbackMethod来给一个方法设置备用的代码逻辑。比如这个时候有一个热点新闻出现了，我们会推荐给用户查看详情，然后用户会通过id去查询新闻的详情，但是因为这条新闻太火了(比如最近什么*易对吧)，大量用户同时访问可能会导致系统崩溃，那么我们就进行**服务降级**，一些请求会做一些降级处理比如当前人数太多请稍后查看等等。

```
//指定了后备方法调用
@HystrixCommand(fallbackMethod = "getHystrixNews")
@GetMapping("/get/news")
public News getNews(@PathVariable("id") int id) {
  //调用新闻系统的获取新闻api 代码逻辑省略
}
//
public News getHystrixNews(@PathVariable("id") int id) {
  //做服务降级
  //返回当前人数太多，请稍后查看
}
```

什么是Hystrix之其他

我在阅读 《Spring微服务实战》这本书的时候还接触到了一个 **舱壁模式**的概念。在不使用舱壁模式的情况下，服务A调用服务B，这种调用默认的是**使用同一批线程来执行**的，而在一个服务出现性能问题的时候，就会出现所有线程被刷爆并等待处理工作，同时阻塞新请求，最终导致程序崩溃。而舱壁模式会将远程资源调用隔离在他们自己的线程池中，以便可以控制单个表现不佳的服务，而不会使该程序崩溃。

具体其原理我推荐大家自己去了解一下，本篇文章中对**舱壁模式**不做过多解释。当然还有**[Hystrix]仪表盘**，它是**用来实时监控[Hystrix]的各项指标信息的**，这里我将这个问题也抛出去，希望有不了解的可以自己去搜索一下。

微服务网关——Zuul

> ZUUL 是从设备和 web 站点到 Netflix 流应用后端的所有请求的前门。作为边界服务应用，ZUUL 是为了实现动态路由、监视、弹性和安全性而构建的。它还具有根据情况将请求路由到多个 Amazon Auto Scaling Groups（亚马逊自动缩放组，亚马逊的一种云计算方式） 的能力

在上面我们学习了 Eureka 之后我们知道了*服务提供者*是*消费者*通过[Eureka] Server进行访问的，即[Eureka] Server是*服务提供者*的统一入口。那么整个应用中存在那么多*消费者*需要用户进行调用，这个时候用户该怎样访问这些*消费者工程*呢？当然可以像之前那样直接访问这些工程。但这种方式没有统一的消费者工程调用入口，不便于访问与管理，而 Zuul 就是这样的一个对于*消费者*的统一入口。

> 如果学过前端的肯定都知道 Router 吧，比如 Flutter 中的路由，Vue，React中的路由，用了 Zuul 你会发现在路由功能方面和前端配置路由基本是一个理。😁 我偶尔撸撸 Flutter。

大家对网关应该很熟吧，简单来讲网关是系统唯一对外的入口，介于客户端与服务器端之间，用于对请求进行**鉴权、限流、路由、监控**等功能。

没错，网关有的功能，Zuul基本都有。而Zuul中最关键的就是**路由和过滤器**了，在官方文档中Zuul的标题就是

> Router and Filter : Zuul

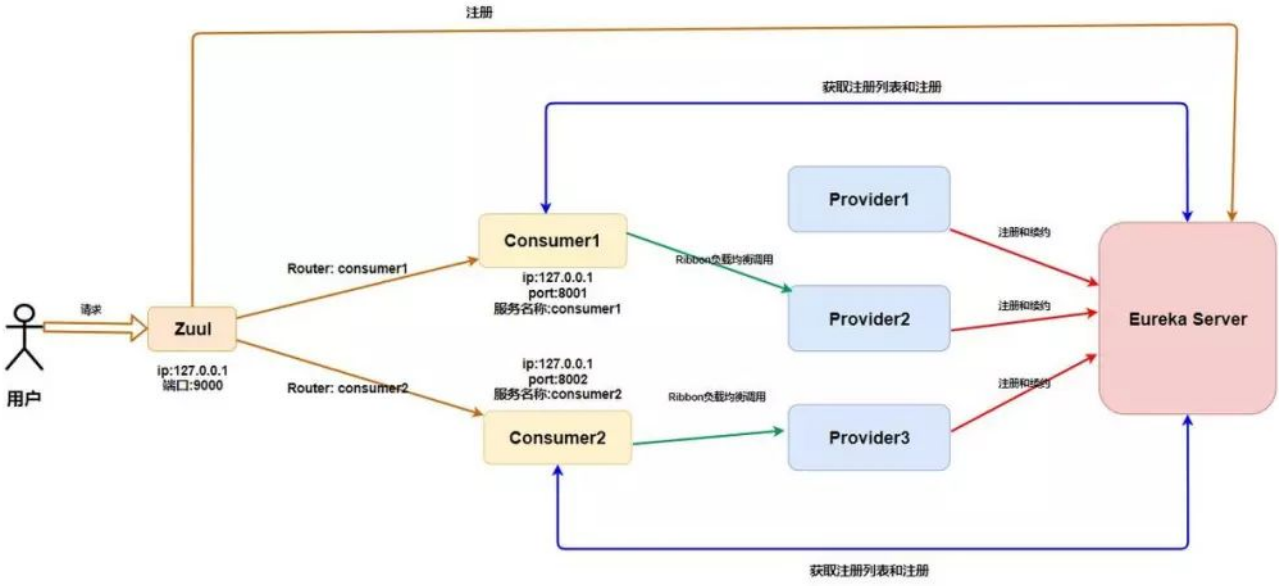
Zuul 的路由功能

简单配置

本来想给你们复制一些代码，但是想了想，因为各个代码配置比较零散，看起来也比较零散，我决定还是给你们画个图来解释吧。

> 请不要因为我这么好就给我点赞👍。疯狂暗示。

比如这个时候我们已经向[Eureka] Server注册了两个Consumer、三个Provicer，这个时候我们再加个Zuul网关应该变成这样子了。



emmm，信息量有点大，我来解释一下。关于前面的知识我就不解释了😐。

首先，Zuul需要向 Eureka 进行注册，注册有啥好处呢？

你傻呀，Consumer都向[Eureka] Server进行注册了，我网关是不是只要注册就能拿到所有Consumer的信息了？

拿到信息有什么好处呢？

我拿到信息我是不是可以获取所有的Consumer的元数据(名称，ip，端口)?

拿到这些元数据有什么好处呢？拿到了我们是不是直接可以做**路由映射**？比如原来用户调用Consumer1的接口localhost:8001/studentInfo/update这个请求，我们是不是可以这样进行调用了呢？localhost:9000/consumer1/studentInfo/update呢？你这样是不是恍然大悟了？

> 这里的url为了让更多人看懂所以没有使用 restful 风格。

上面的你理解了，那么就能理解关于Zuul最基本的配置了，看下面。

```
server:
 port: 9000
eureka:
 client:
  service-url:
   # 这里只要注册 Eureka 就行了
   defaultZone: http://localhost:9997/eureka
```

然后在启动类上加入@EnableZuulProxy注解就行了。没错，就是那么简单😉。

统一前缀

这个很简单，就是我们可以在前面加一个统一的前缀，比如我们刚刚调用的是localhost:9000/consumer1/studentInfo/update，这个时候我们在yaml配置文件中添加如下。

```
zuul:
 prefix: /zuul
```

这样我们就需要通过localhost:9000/zuul/consumer1/studentInfo/update来进行访问了。

路由策略配置

你会发现前面的访问方式(直接使用服务名)，需要将微服务名称暴露给用户，会存在安全性问题。所以，可以自定义路径来替代微服务名称，即自定义路由策略。

```
zuul:
 routes:
  consumer1: /FrancisQ1/**
  consumer2: /FrancisQ2/**
```

这个时候你就可以使用localhost:9000/zuul/FrancisQ1/studentInfo/update进行访问了。

服务名屏蔽

这个时候你别以为你好了，你可以试试，在你配置完路由策略之后使用微服务名称还是可以访问的，这个时候你需要将服务名屏蔽。

```
zuul:
 ignore-services: "*"
```

路径屏蔽

Zuul还可以指定屏蔽掉的路径 URI，即只要用户请求中包含指定的 URI 路径，那么该请求将无法访问到指定的服务。通过该方式可以限制用户的权限。

```
zuul:
 ignore-patterns: **/auto/**
```

这样关于 auto 的请求我们就可以过滤掉了。

> ** 代表匹配多级任意路径
> *代表匹配一级任意路径

敏感请求头屏蔽

默认情况下，像 Cookie、Set-Cookie 等敏感请求头信息会被 zuul 屏蔽掉，我们可以将这些默认屏蔽去掉，当然，也可以添加要屏蔽的请求头。

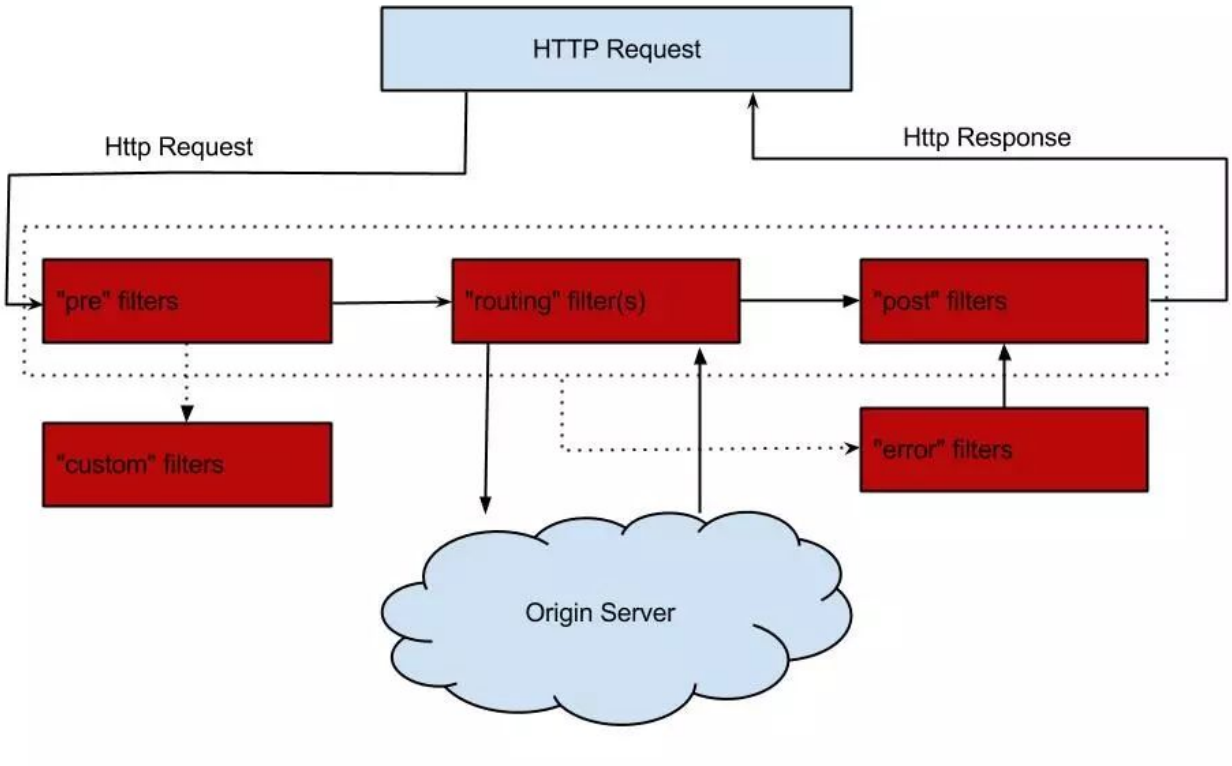
Zuul 的过滤功能

如果说，路由功能是Zuul的基操的话，那么**过滤器**就是Zuul的利器了。毕竟所有请求都经过网关(Zuul)，那么我们可以进行各种过滤，这样我们就能实现**限流**，**灰度发布**，**权限控制**等等。

简单实现一个请求时间日志打印

要实现自己定义的Filter我们只需要继承ZuulFilter然后将这个过滤器类以@Component注解加入 Spring 容器中就行了。

在给你们看代码之前我先给你们解释一下关于过滤器的一些注意点。



过滤器类型：Pre、Routing、Post。前置Pre就是在请求之前进行过滤，Routing路由过滤器就是我们上面所讲的路由策略，而Post后置过滤器就是在Response之前进行过滤的过滤器。你可以观察上图结合着理解，并且下面我会给出相应的注释。

```
// 加入Spring容器
@Component
public class PreRequestFilter extends ZuulFilter {
  // 返回过滤器类型 这里是前置过滤器
  @Override
  public String filterType() {
    return FilterConstants.PRE_TYPE;
  }
  // 指定过滤顺序 越小越先执行，这里第一个执行
  // 当然不是只真正第一个 在Zuul内置中有其他过滤器会先执行
  // 那是写死的 比如 SERVLET_DETECTION_FILTER_ORDER = -3
  @Override
```

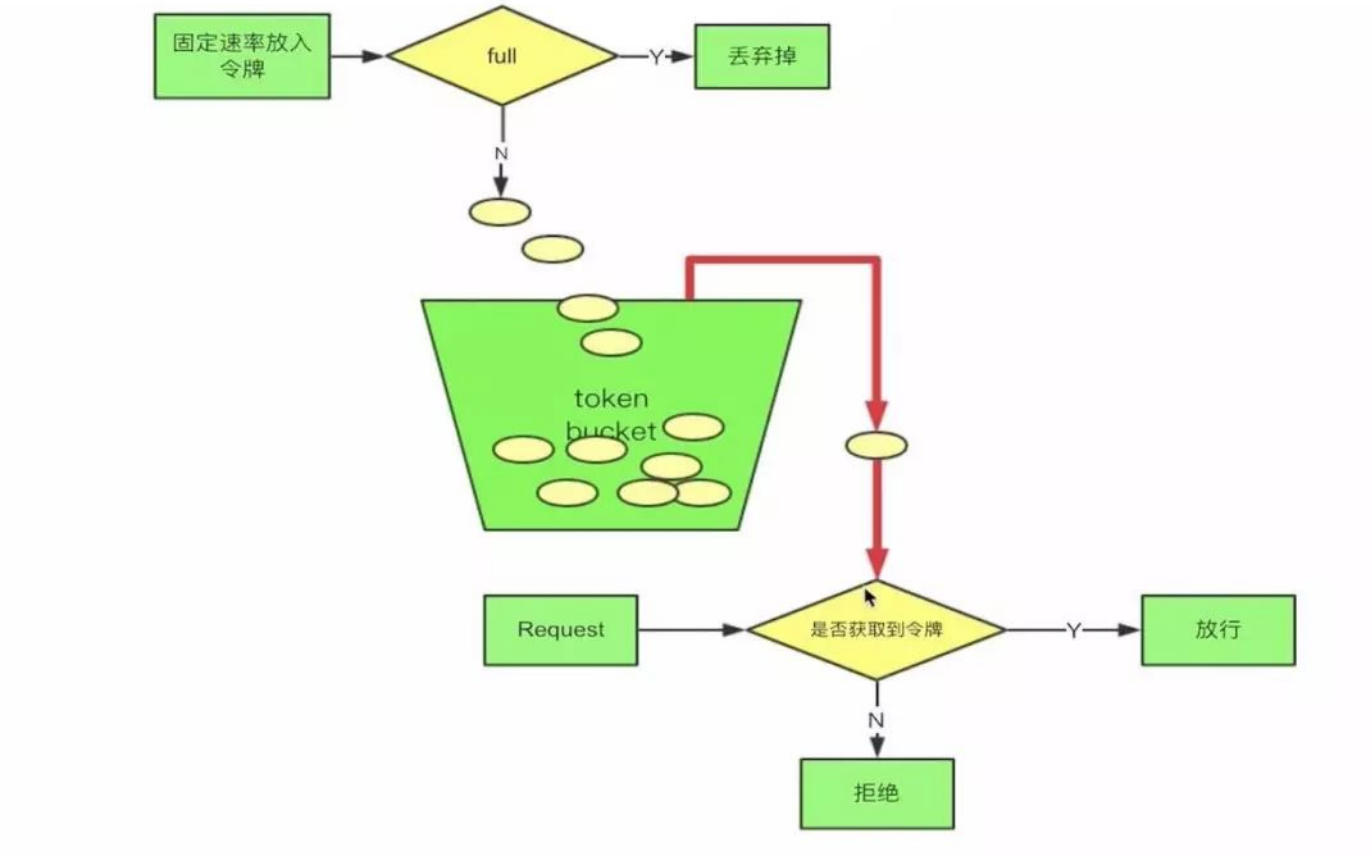
```
  public int filterOrder() {
    return 0;
  }
  // 什么时候该进行过滤
  // 这里我们可以进行一些判断，这样我们就可以过滤掉一些不符合规定的请求等等
  @Override
  public boolean shouldFilter() {
    return true;
  }
  // 如果过滤器允许通过则怎么进行处理
  @Override
  public Object run() throws ZuulException {
    // 这里我设置了全局的RequestContext并记录了请求开始时间
    RequestContext ctx = RequestContext.getCurrentContext();
    ctx.set("startTime", System.currentTimeMillis());
    return null;
  }
}

// lombok的日志
@Slf4j
// 加入 Spring 容器
@Component
public class AccessLogFilter extends ZuulFilter {
  // 指定该过滤器的过滤类型
  // 此时是后置过滤器
  @Override
  public String filterType() {
    return FilterConstants.POST_TYPE;
  }
  // SEND_RESPONSE_FILTER_ORDER 是最后一个过滤器
  // 我们此过滤器在它之前执行
  @Override
  public int filterOrder() {
    return FilterConstants.SEND_RESPONSE_FILTER_ORDER - 1;
  }
  @Override
  public boolean shouldFilter() {
    return true;
  }
  // 过滤时执行的策略
  @Override
  public Object run() throws ZuulException {
    RequestContext context = RequestContext.getCurrentContext();
    HttpServletRequest request = context.getRequest();
    // 从RequestContext获取原先的开始时间 并通过它计算整个时间间隔
    Long startTime = (Long) context.get("startTime");
    // 这里我可以获取HttpServletRequest来获取URI并且打印出来
    String uri = request.getRequestURI();
    long duration = System.currentTimeMillis() - startTime;
    log.info("uri: " + uri + ", duration: " + duration / 100 + "ms");
    return null;
  }
}
```

上面就简单实现了请求时间日志打印功能，你有没有感受到Zuul过滤功能的强大了呢？

没有？好的、那我们再来。

令牌桶限流

当然不仅仅是令牌桶限流方式，Zuul只要是限流的活它都能干，这里我只是简单举个🌰。



我先来解释一下什么是**令牌桶限流**吧。

首先我们会有个桶，如果里面没有满那么就会以一定**固定的速率**会往里面放令牌，一个请求过来首先要从桶中获取令牌，如果没有获取到，那么这个请求就拒绝，如果获取到那么就放行。很简单吧，啊哈哈、

下面我们就通过Zuul的前置过滤器来实现一下令牌桶限流。

```java
@Component
@Slf4j
public class RouteFilter extends ZuulFilter {
  // 定义一个令牌桶，每秒产生2个令牌，即每秒最多处理2个请求
  private static final RateLimiter RATE_LIMITER = RateLimiter.create(2);
  @Override
  public String filterType() {
    return FilterConstants.PRE_TYPE;
  }

  @Override
  public int filterOrder() {
    return -5;
  }

  @Override
  public Object run() throws ZuulException {
    log.info("放行");
    return null;
  }

  @Override
  public boolean shouldFilter() {
    RequestContext context = RequestContext.getCurrentContext();
    if(!RATE_LIMITER.tryAcquire()) {
      log.warn("访问量超载");
      // 指定当前请求未通过过滤
      context.setSendZuulResponse(false);
      // 向客户端返回响应码429，请求数量过多
      context.setResponseStatusCode(429);
      return false;
    }
    return true;
  }
}
```

这样我们就能将请求数量控制在一秒两个，有没有觉得很酷?

关于 Zuul 的其他

Zuul的过滤器的功能肯定不止上面我所实现的两种，它还可以实现**权限校验**，包括我上面提到的**灰度发布**等等。

当然，Zuul作为网关肯定也存在**单点问题**，如果我们要保证Zuul的高可用，我们就需要进行Zuul的集群配置，这个时候可以借助额外的一些负载均衡器比如Nginx。

Spring Cloud配置管理——Config

为什么要使用进行配置管理?

当我们的微服务系统开始慢慢地庞大起来，那么多Consumer、Provider、[Eureka] Server、Zuul系统都会持有自己的配置，这个时候我们在项目运行的时候可能需要更改某些应用的配置，如果我们不进行配置的统一管理，我们只能**去每个应用下一个一个寻找配置文件然后修改配置文件再重启应用**。

首先对于分布式系统而言我们就不应该去每个应用下去分别修改配置文件，再者对于重启应用来说，服务无法访问所以直接抛弃了可用性，这是我们更不愿见到的。

那么有没有一种方法**既能对配置文件统一地进行管理，又能在项目运行时动态修改配置文件呢?**

那就是我今天所要介绍的Spring Cloud Config。

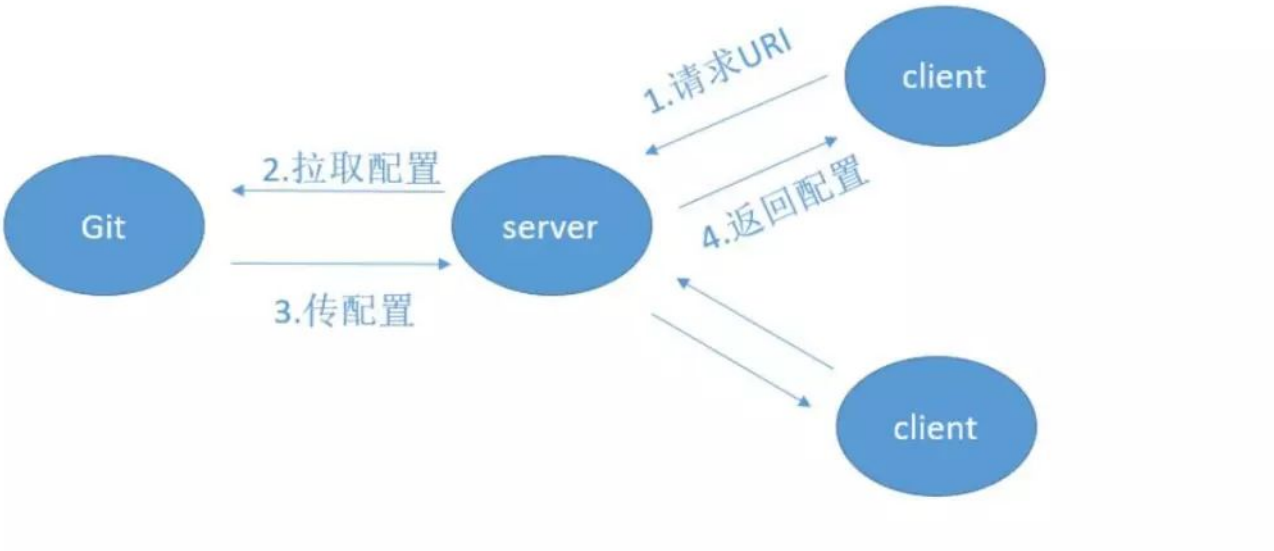
> 能进行配置管理的框架不止Spring Cloud Config一种，大家可以根据需求自己选择（disconf，阿波罗等等）。而且对于Config来说有些地方实现的不是那么尽人意。

Config 是什么

> Spring Cloud Config为分布式系统中的外部化配置提供服务器和客户端支持。使用Config服务器，可以在中心位置管理所有环境中应用程序的外部属性。

简单来说，Spring Cloud Config就是能将各个 应用/系统/模块 的配置文件存放到**统一的地方然后进行管理**(Git 或者 SVN)。

你想一下，我们的应用是不是只有启动的时候才会进行配置文件的加载，那么我们的Spring Cloud Config就暴露出一个接口给启动应用来获取它所想要的配置文件，应用获取到配置文件然后再进行它的初始化工作。就如下图。



当然这里你肯定还会有一个疑问，如果我在应用运行时去更改远程配置仓库(Git)中的对应配置文件，那么依赖于这个配置文件的已启动的应用会不会进行其相应配置的更改呢？

答案是不会的。

什么？那怎么进行动态修改配置文件呢？这不是出现了**配置漂移**吗？你个渣男😩，你又骗我！

别急嘛，你可以使用Webhooks， 这 是 github提供的功能，它能确保远程库的配置文件更新后客户端中的配置信息也得到更新。

噢噢，这还差不多。我去查查怎么用。

慢着，听我说完，Webhooks虽然能解决，但是你了解一下会发现它根本不适合用于生产环境，所以基本不会使用它的。

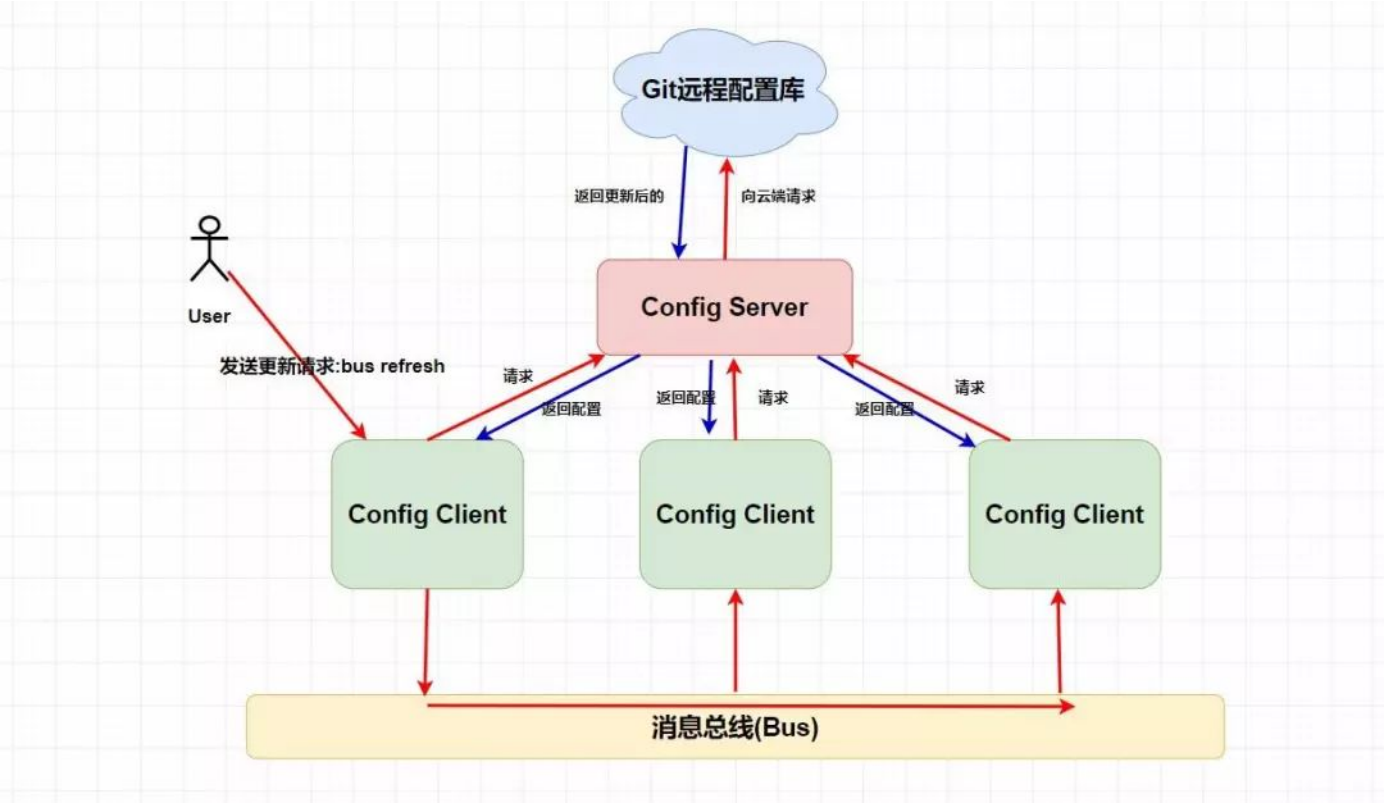
而一般我们会使用Bus消息总线 +Spring Cloud Config进行配置的动态刷新。

引出 Spring Cloud Bus

> 用于将服务和服务实例与分布式消息系统链接在一起的事件总线。在集群中传播状态更改很有用（例如配置更改事件）。

你可以简单理解为Spring Cloud Bus的作用就是**管理和广播分布式系统中的消息**，也就是消息引擎系统中的广播模式。当然作为**消息总线**的Spring Cloud Bus可以做很多事而不仅仅是客户端的配置刷新功能。

而拥有了Spring Cloud Bus之后，我们只需要创建一个简单的请求，并且加上@ResfreshScope注解就能进行配置的动态修改了，下面我画了张图供你理解。



总结

这篇文章中我带大家初步了解了Spring Cloud的各个组件，他们有

- Eureka 服务发现框架
- Ribbon 进程内负载均衡器
- Open Feign 服务调用映射
- Hystrix 服务降级熔断器
- Zuul 微服务网关
- Config 微服务统一配置中心
- Bus 消息总线

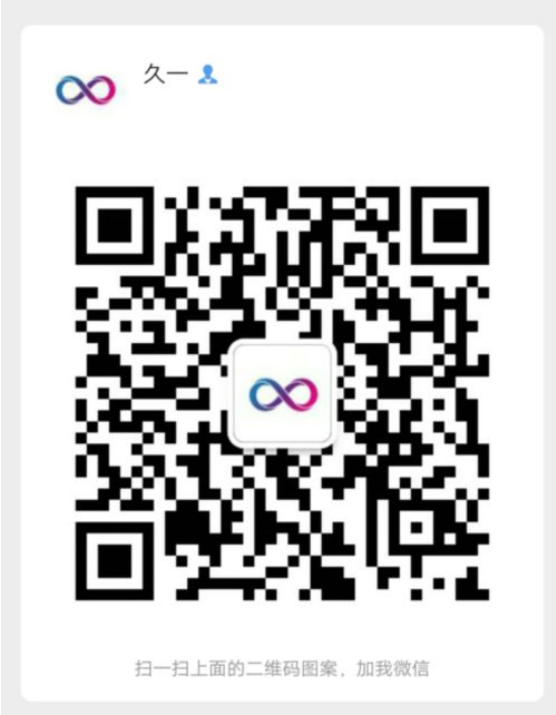
如果你能这个时候能看懂下面那张图，也就说明了你已经对Spring Cloud微服务有了一定的架构认识。

有偿投稿： 欢迎投稿原创技术博文，一旦采用将给予 50元 - 200元 不等稿费，要求个人原创，图文并茂。可以是职场经验、面试经历，也可是技术教程，学习笔记。投稿请联系微信「web527zsd」，备注投稿。

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓扫描二维码添加小编↓



推荐阅读

1. 一张图帮你记忆：Spring Boot

2. 10 分钟实现 Java 发送邮件功能

3. Spring Boot 项目并发提升几倍

4. Spring Boot 如何跨域请求

面试官最常问的 Spring Cloud 面试题

Java后端 2月2日

作者 | moakun

blog.csdn.net/moakun/article/details/82817757

今天跟大家分享下SpringCloud常见面试题的知识。

1 什么是Spring Cloud?

Spring cloud流应用程序启动器是基于Spring Boot的Spring集成应用程序，提供与外部系统的集成。Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。

2 使用Spring Cloud有什么优势?

使用Spring Boot开发分布式微服务时，我们面临以下问题:

与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。

服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。

冗余-分布式系统中的冗余问题。

负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。

性能-问题 由于各种运营开销导致的性能问题。部署复杂性-Devops技能的要求。

3 服务注册和发现是什么意思?Spring Cloud如何实现?

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka服务注册和发现可以在这种情况下提供帮助。由于所有服务都在Eureka服务器上注册并通过调用Eureka服务器完成查找，因此无需处理服务地点的任何更改和处理。

4 负载平衡的意义什么?

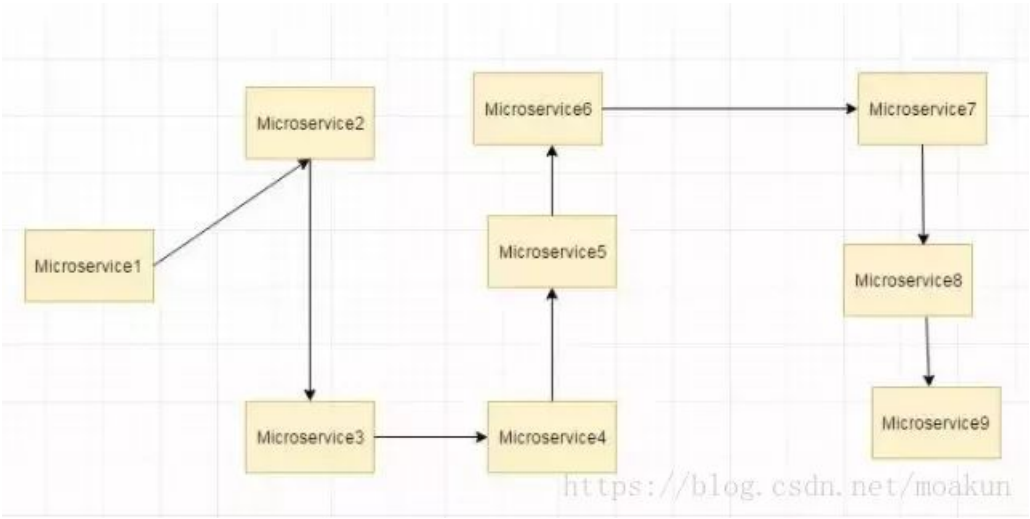
在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。

5 什么是Hystrix?它如何实现容错?

Hystrix是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

思考以下微服务



假设如果上图中的微服务9失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

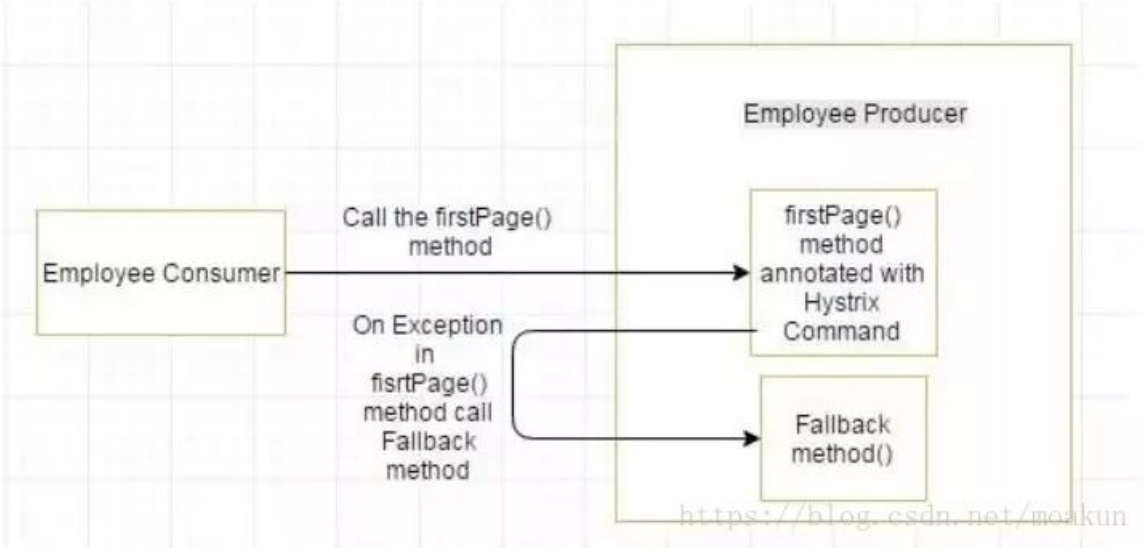
随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达1000.这是hystrix出现的地方　我们将使用Hystrix在这种情况下的Fallback方法功能。我们有两个服务employee-consumer使用由employee-consumer公开的服务。

简化图如下所示

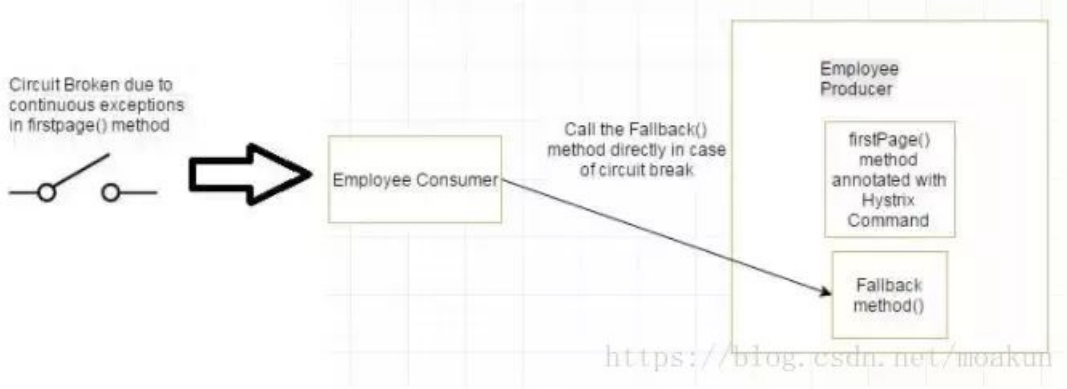
现在假设由于某种原因，employee-producer公开的服务会抛出异常。我们在这种情况下使用Hystrix定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

6 什么是Hystrix断路器?我们需要它吗?

由于某些原因，employee-consumer公开服务会引发异常。在这种情况下使用Hystrix我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。



如果firstPage method() 中的异常继续发生，则Hystrix电路将中断，并且员工使用者将一起跳过firtsPage方法，并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。



7 什么是Netflix Feign?它的优点是什么?

Feign是受到Retrofit，JAXRS-2.0和WebSocket启发的java客户端联编程序。Feign的第一个目标是将约束分母的复杂性统一到http apis，而不考虑其稳定性。在employee-consumer的例子中，我们使用了employee-producer使用REST模板公开的REST服务。

但是我们必须编写大量代码才能执行以下步骤：

使用功能区进行负载平衡。

获取服务实例，然后获取基本URL。

利用REST模板来使用服务。前面的代码如下：

```
@Controller
public class ConsumerControllerClient {
@Autowired
private LoadBalancerClient loadBalancer;
public void getEmployee() throws RestClientException, IOException
{
    ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");
    System.out.println(serviceInstance.getUri());
    String baseUrl=serviceInstance.getUri().toString();
    baseUrl=baseUrl+"/employee"
```

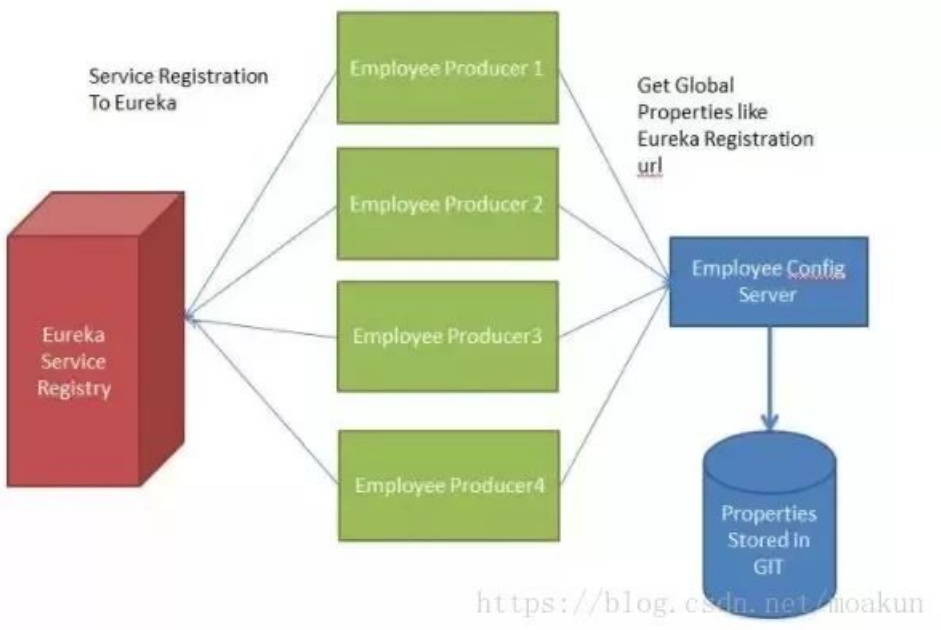
```
;
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> response=null
;
try{
    response=restTemplate.exchange(baseUrl,
            HttpMethod.GET, getHeaders(),String.class);
    }catch (Exception ex)
    {
        System.out.println(ex);
    }
    System.out.println(response.getBody());
}
```

之前的代码，有像NullPointer这样的例外的机会，并不是最优的。我们将看到如何使用Netflix Feign使呼叫变得更加轻松和清洁。如果Netflix Ribbon依赖关系也在类路径中，那么Feign默认也会负责负载平衡。

8 什么是Spring Cloud Bus?我们需要它吗?

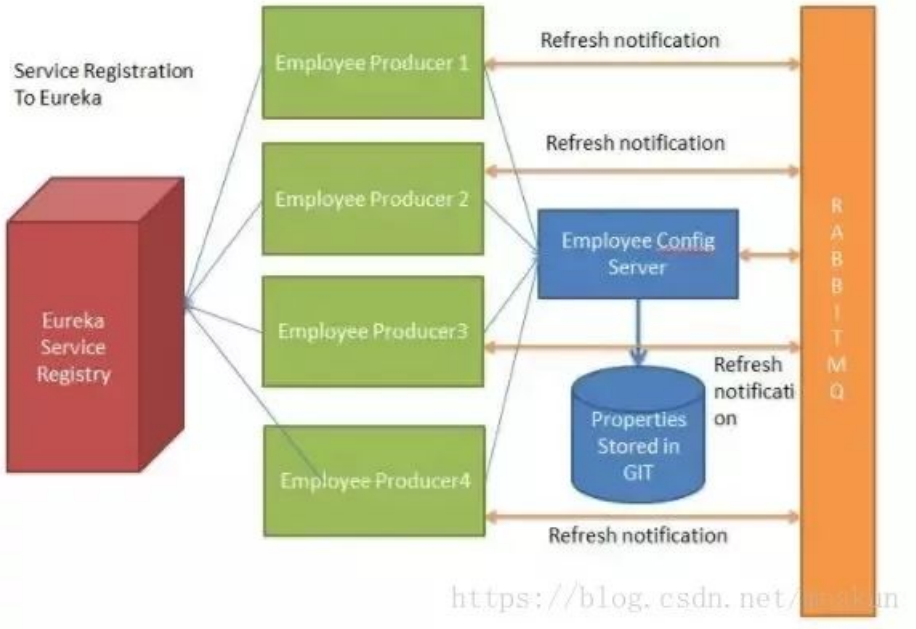
考虑以下情况：我们有多个应用程序使用Spring Cloud Config读取属性，而Spring Cloud Config从GIT读取这些属性。

下面的例子中多个员工生产者模块从Employee Config Module获取Eureka注册的财产。



如果假设GIT中的Eureka注册属性更改为指向另一台Eureka服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个url。例如，如果Employee Producer1部署在端口8080上，则调用 http:// localhost:8080 / refresh。同样对于Employee Producer2 http:// localhost:8081 / refresh等等。这又很麻烦。这就是Spring Cloud Bus发挥作用的地方。



Spring Cloud Bus提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

9 SpringCloud和Dubbo

SpringCloud和Dubbo都是现在主流的微服务架构

SpringCloud是Apache旗下的Spring体系下的微服务解决方案

Dubbo是阿里系的分布式服务治理框架

从技术维度上,其实SpringCloud远远的超过Dubbo,Dubbo本身只是实现了服务治理,而SpringCloud现在以及有21个子项目以后还会更多

所以其实很多人都会说Dubbo和SpringCloud是不公平的

但是由于RPC以及注册中心元数据等原因,在技术选型的时候我们只能二者选其一,所以我们常常为用他俩来对比

服务的调用方式Dubbo使用的是RPC远程调用,而SpringCloud使用的是 Rest API,其实更符合微服务官方的定义

服务的注册中心来看,Dubbo使用了第三方的ZooKeeper作为其底层的注册中心,实现服务的注册和发现,SpringCloud使用Spring Cloud Netflix Eureka实现注册中心,当然SpringCloud也可以使用ZooKeeper实现,但一般我们不会这样做

服务网关,Dubbo并没有本身的实现,只能通过其他第三方技术的整合,而SpringCloud有Zuul路由网关,作为路由服务器,进行消费者的请求分发,SpringCloud还支持断路器,与git完美集成分布式配置文件支持版本控制,事务总线实现配置文件的更新与服务自动装配等等一系列的微服务架构要素

10 技术选型

目前国内的分布式系统选型主要还是Dubbo毕竟国产,而且国内工程师的技术熟练程度高,并且Dubbo在其他维度上的缺陷可以由其他第三方框架进行集成进行弥补

而SpringCloud目前是国外比较流行,当然我觉得国内的市场也会慢慢的偏向SpringCloud,就连刘军作为Dubbo重启的负责人也发表过观点,Dubbo的发展方向是积极适应SpringCloud生态,并不是起冲突

11 Rest和RPC对比

其实如果仔细阅读过微服务提出者马丁福勒的论文的话可以发现其定义的服务间通信机制就是Http Rest

RPC最主要的缺陷就是服务提供方和调用方式之间依赖太强,我们需要为每一个微服务进行接口的定义,并通过持续继承发布,需要严格的版本控制才不会出现服务提供和调用之间因为版本不同而产生的冲突

而REST是轻量级的接口,服务的提供和调用不存在代码之间的耦合,只是通过一个约定进行规范,但也有可能出现文档和接口不一致而导致的服务集成问题,但可以通过swagger工具整合,是代码和文档一体化解决,所以REST在分布式环境下比RPC更加灵活

这也是为什么当当网的DubboX在对Dubbo的增强中增加了对REST的支持的原因

12 文档质量和社区活跃度

SpringCloud社区活跃度远高于Dubbo,毕竟由于梁飞团队的原因导致Dubbo停止更新迭代五年,而中小型公司无法承担技术开发的成本导致Dubbo社区严重低落,而SpringCloud异军突起,迅速占领了微服务的市场,背靠Spring混的风生水起

Dubbo经过多年的积累文档相当成熟,对于微服务的架构体系各个公司也有稳定的现状

13 SpringBoot和SpringCloud

SpringBoot是Spring推出用于解决传统框架配置文件冗余,装配组件繁杂的基于Maven的解决方案,旨在快速搭建单个微服务

而SpringCloud专注于解决各个微服务之间的协调与配置,服务之间的通信,熔断,负载均衡等,技术维度并相同,并且SpringCloud是依赖于SpringBoot的,而SpringBoot并不是依赖与SpringCloud,甚至还可以和Dubbo进行优秀的整合开发

总结:

SpringBoot专注于快速方便的开发单个个体的微服务

SpringCloud是关注全局的微服务协调整理治理框架,整合并管理各个微服务,为各个微服务之间提供,配置管理,服务发现,断路器,路由,事件总线等集成服务

SpringBoot不依赖于SpringCloud,SpringCloud依赖于SpringBoot,属于依赖关系

SpringBoot专注于快速,方便的开发单个的微服务个体,SpringCloud关注全局的服务治理框架

14 Eureka和ZooKeeper都可以提供服务注册与发现的功能,请说说两个的区别

1.ZooKeeper保证的是CP,Eureka保证的是AP

ZooKeeper在选举期间注册服务瘫痪,虽然服务最终会恢复,但是选举期间不可用的

Eureka各个节点是平等关系,只要有一台Eureka就可以保证服务可用,而查询到的数据并不是最新的

自我保护机制会导致：

Eureka不再从注册列表移除因长时间没收到心跳而应该过期的服务

Eureka仍然能够接受新服务的注册和查询请求,但是不会被同步到其他节点(高可用)

当网络稳定时,当前实例新的注册信息会被同步到其他节点中(最终一致性)

Eureka可以很好的应对因网络故障导致部分节点失去联系的情况,而不会像ZooKeeper一样使得整个注册系统瘫痪

2.ZooKeeper有Leader和Follower角色,Eureka各个节点平等

3.ZooKeeper采用过半数存活原则,Eureka采用自我保护机制解决分区问题

4.Eureka本质上是一个工程,而ZooKeeper只是一个进程

15 微服务之间是如何独立通讯的

微服务通信机制

系统中的各个微服务可被独立部署，各个微服务之间是松耦合的。每个微服务仅关注于完成一件任务并很好地完成该任务。

围绕业务能力组织服务、自动化部署、智能端点、对语言及数据的去集中化控制。

将组件定义为可被独立替换和升级的软件单元。

以业务能力为出发点组织服务的策略。

倡导谁开发，谁运营的开发运维一体化方法。

RESTful HTTP协议是微服务架构中最常用的通讯机制。

每个微服务可以考虑选用最佳工具完成(如不同的编程语言)。

允许不同微服务采用不同的数据持久化技术。

微服务非常重视建立架构及业务相关指标的实时监控和日志机制，必须考虑每个服务的失败容错机制。

注重快速更新，因此系统会随时间不断变化及演进。可替代性模块化设计。

微服务通信方式：

同步：RPC，REST等

异步：消息队列。要考虑消息可靠传输、高性能，以及编程模型的变化等。

消息队列中间件如何选型

1.协议：AMQP、STOMP、MQTT、私有协议等。

2.消息是否需要持久化。

3.吞吐量。

4.高可用支持,是否单点。

5.分布式扩展能力。

6.消息堆积能力和重放能力。

7.开发便捷,易于维护。

8.社区成熟度。

RabbitMQ是一个实现了AMQP(高级消息队列协议)协议的消息队列中间件。RabbitMQ支持其中的最多一次和最少一次两种。网易蜂巢平台的服务架构,服务间通过RabbitMQ实现通信。

16 什么是服务熔断?什么是服务降级

在复杂的分布式系统中,微服务之间的相互调用,有可能出现各种各样的原因导致服务的阻塞,在高并发场景下,服务的阻塞意味着线程的阻塞,导致当前线程不可用,服务器的线程全部阻塞,导致服务器崩溃,由于服务之间的调用关系是同步的,会对整个微服务系统造成服务雪崩,为了解决某个微服务的调用响应时间过长或者不可用进而占用越来越多的系统资源引起雪崩效应就需要进行服务熔断和服务降级处理。

所谓的服务熔断指的是某个服务故障或异常一起类似显示世界中的“保险丝"当某个异常条件被触发就直接熔断整个服务,而不是一直等到此服务超时。

服务熔断就是相当于我们电闸的保险丝,一旦发生服务雪崩的,就会熔断整个服务,通过维护一个自己的线程池,当线程达到阈值的时候就启动服务降级,如果其他请求继续访问就直接返回fallback的默认值。

17 微服务的优缺点分别是什么?说下你在项目开发中碰到的坑

优点:

每一个服务足够内聚,代码容易理解

开发效率提高,一个服务只做一件事

微服务能够被小团队单独开发

微服务是松耦合的,是有功能意义的服务

可以用不同的语言开发,面向接口编程

易于与第三方集成

微服务只是业务逻辑的代码,不会和HTML,CSS或者其他界面组合

开发中,两种开发模式

前后端分离

全栈工程师

可以灵活搭配,连接公共库/连接独立库

缺点:

分布式系统的负责性;多服务运维难度,随着服务的增加,运维的压力也在增大;系统部署依赖;服务间通信成本;数据一致性;系统集成测试;性能监控.

18 你所知道的微服务技术栈有哪些?请列举一二

多种技术的集合体

我们在讨论一个分布式的微服务架构的话,需要哪些维度

维度(SpringCloud)

```
服务开发
            SpringBoo
t
            Sprin
g
            SpringMV
C
        服务配置与管理

Netfilx公司的Archaiusm,阿里的Diamond
        服务注册与发
现
            Eureka,ZooKeeper
        服务调
用
            Rest,RPC,gRPC
        服务熔断
器
            Hystri
x
        服务负载均
衡
            Ribbon,Nginx
        服务接口调
用
            Feig
n
        消息队
列
            Kafka,RabbitMq,ActiveMq
        服务配置中心管
理
            SpringCloudConfing
        服务路由(API网关
)
            Zuu
```

```
l
        事件消息总
线
            SpringCloud Bu
s
```

推荐阅读

1. 村干部们的智慧

2. IntelliJ 发布 2020 RoadMap

3. 互联网公司的抗疫情行动！

4. 如何获取靠谱的新型冠状病毒疫情