# 微服务(马丁福勒)

## 1.1 微服务

提倡将单一应用程序拆分成一组小的服务，每个服务运行在其独立的进程中，服务之间互相协调、互相配合，为用户提供最终价值。服务之间采用轻量级的通信机制互相沟通(dubbo是rpc，springcloud是rest风格)。每个服务都围绕着具体业务进行构建，并且能够独立的部署到生产环境、类生产环境等，可以拥有自己独立的数据库。

注：标红的是微服务的特点(共三点)

## 1.2微服务和微服务架构

微服务强调的是服务的大小，关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用。即一个个的服务个体。

微服务架构是一种架构模式。

## 微服务的优缺点

优点：①每个服务足够内聚，足够小，代码容易理解，可以聚焦一个指定的业务功能或业务需求。

②开发简单、开发效率提高，一个服务基本上就是专一的一件事。

③微服务是松耦合的，无论是开发阶段或部署阶段都是独立的。

④微服务可以使用不同的语言开发。

⑤易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins，Hudson，bamboo。

⑥微服务易于被一个开发人员理解，修改和维护。

⑦微服务只是业务逻辑的代码实现，不会和HTML,CSS或其他界面组件混合。

⑧每个服务都有自己的存储能力，可以有自己的数据库，也可以有统一的数据库。

缺点：①开发人员要处理分布式系统的复杂性

②多服务运维难度增加，随着服务的增加，运维的压力也在增加

③系统部署依赖

④服务间通信成本增加

⑤数据一致性

⑥系统集成测试

⑦性能监控

## 微服务的技术栈

服务开发(Springboot、Spring、SpringMVC)、服务治理、服务注册、服务调用、服务负载均衡、服务监控、服务熔断器、消息队列、服务路由、服务接口调用等维度的技术栈。

# SpringCloud

## 2.1 概述

SpringCloud基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现、配置中心、全链路监控、服务网关、负载均衡、服务熔断等组件，除了基于NetFlix的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。

SpringCloud=分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体。

## 2.2 springboot和springcloud的关系

Springboot专注于快速方便的开发单个个体微服务，springcloud关注全局的微服务协调治理框架，它将springboot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务。

Springboot可以离开springcloud独立使用开发项目，但是springcloud离不开springboot，属于依赖的关系。

Springboot专注于快速、方便的开发单个微服务个体，springcloud关注全局的服务治理框架。

## dubbo和springcloud的对比

注：springcloud版本号不是数字形式的。

# 工程的搭建

## 3.1 父工程的创建

父工程的packing为pom，且可以只有一个pom.xml文件

## 3.2 api工程的创建

继承父工程，则父工程的配置该工程都有，且微服务的实体类必须要实现序列化接口，可结合lombok在实体类上加相应的注解完成相应的空参(@NoArgsConstructor)，全参(@AllArgsConstructor)，get/set(@Data)，链式访问数据(@Accessors(chain=true))的相应编码。

链式：如a.setName(“zhangsan”).setAge(10);

注：①约定>配置>编码(创建工程时可以遵循这个规则，先配置，再编码)

②SpringApplicationName就是SpringCloud微服务暴露的服务名

③mysql中database()函数返回当前所在的数据库名

## provider工程的创建

1. pom文件的配置
2. yml文件的配置
3. 数据库表的创建以及表数据的添加
4. Mapper接口的编写(接口一定要加@Mapper注解)
5. Mapper对应的表操作xml文件的编写
6. Service接口的编写
7. ServiceImpl实现类的编写
8. Controller的编写
9. 主启动类的编写

注：前后端分离工程，后端应返回数据json串给前端，故需要给controller类加上@RestController注解，则该类的方法返回的为json串。

## consumer工程的创建

1. pom文件的配置
2. yml文件的配置
3. ConfigBean配置注入Bean类的编写🡪即@Configuration标志的类中加上@Bean注解注入类。这里主要用来生成一个RestTemplate对象，用来发Rest请求。
4. Controller的编写🡪需要用③生成的RestTemplate对象完成相应的Rest风格的服务调用操作。
5. 主启动类的编写

注：RestTemplate提供了多种便捷访问远程Http服务的方法，是一种便捷的访问RestFul服务模板类，是Spring提供的用于访问Rest服务的客户端模板工具集。使用如：restTemplate.postForObject(“http://localhost:8081/dept/add”,dept,Boolean.class);这三个参数分别为url，request，responseType。

# Eureka(Netflix设计Eureka遵循AP原则)

## 4.1概述

Eureka是NetFlix的一个子模块，也是核心模块之一。Eureka是一个基于REST的服务，用于定位服务，以实现云端中间层服务发现和故障转移。服务注册与发现对于微服务架构来说是非常重要的，有了服务注册和发现，只需要使用服务的标识符，就可以访问到服务，而不需雅尼修改服务调用的配置文件了。功能类似于dubbo的注册中心，如Zookeeper。

## 4.2 基本架构

采用了C/S的设计架构。Eureka Server作为服务注册功能的服务器，是服务注册中心。

系统的其他微服务，使用Eureka的客户端连接到Eureka Server并维持心跳连接。这样系统维护人员就可以通过Eureka Server来监控系统中各个微服务是否正常运行。SpringCloud的一些其他模块(如Zuul)就可以通过Eureka Server来发现系统的其他微服务，并执行相关的逻辑。

Eureka包含两个组件：Eureka Server和Eureka Client。Eureka Server提供服务注册功能，各个节点启动后，会到Eureka Server进行注册，这样Eureka Server的服务注册表将会存储所有可用服务节点的信息，服务节点的信息可以再界面中直观的看到。

Eureka Client是一个java客户端，用于简化与Eureka Server的交互，同时也具备一个内置的、使用轮询负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认是30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中将这个服务节点移除(默认90秒)。

# Ribbon

## 5.1 ribbon的概述

Ribbon是一种基于客户端的软负载均衡组件

## 5.2 ribbon核心组件IRule(也是一个接口)

根据特定算法从服务列表中选取一个要访问的服务

算法：①RoundRobinRule ->轮询(默认)

②RandomRule –>随机

③AvailabilityFilteringRule –>先过滤由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阙值的服务，然后对剩余的服务列表按照轮询策略进行访问

④WeightedResponseTimeRule ->根据平均响应时间计算所有服务的权重，时间响应越快服务权重越大被选中的概率越高。刚启动时若统计信息不足，则使用轮询策略，等统计信息足够，会切换到该算法。

⑤RetryRule –>先按照轮询策略获取服务，若获取服务失败在指定时间内会进行重试，获取可用的服务。

⑥BestAvailableRule –>会先过滤掉由于多次访问而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务。

⑦ZoneAvoidanceRule ->默认规则，复合判断server所在区域的性能和server的可用性选择服务器。

自定义ribbon策略：

①ribbon策略在上述七个策略范围之内，则需在ConfigBean配置注入Bean类中加入返回IRule对象类型方法，返回实现IRule接口的实例。(可为上述的七种类型的实例对象)。

②ribbon策略在上述七个策略范围之外，则首先需要在客户端的主启动类上加@RibbonClient注解，如@RibbonClient(name=”微服务名”,configuration=自定义策略类.class)，则在启动该微服务的时候就能去加载我们自定义的Ribbon配置类，从而使配置生效。

注:自定义配置类不能放在@ComponentScan所扫描的当前包及其子类下，否则自定义的配置类就会被所有的Ribbon客户端所共享，就达不到特殊化定制的目的了。故配置类不能在主配置类路径及其子路径下，且配置类与ConfigBean配置ribbon策略一致。

# Feign(面向接口编程)

## 6.1 概述

Feign是一个声明式WebService客户端。使用Feign能让编写WebService客户端更加简单，它的使用方法是定义一个接口，然后在上面注解，同时也支持JAX-RS标准的注解。Feign也支持可热拔插的编码器和解码器。SpringCloud对Feign进行了封装，使其支持了SpringMVC标准注解和HttpMessageConverters。它可以与Eureka和Ribbon组合使用以支持负载均衡。

## 6.2 使用

①新建消费端工程，并且添加Feign的相关依赖

②在api工程里添加一个接口，和提供者里的服务接口内容一致，并加上@FeignClient(value=服务名)，各个方法上加上@RequestMapping的相关注解。

③在消费端的Controller类上注入②中定义的该接口实例，并在消费端的主启动类上添加@EnableFeignClients注解，[如@EnableFeignClients(basePackages={“com.atguigu.springcloud”})](mailto:如@EnableFeignClients(basePackages=%7b)

注：①Feign集成了Ribbon，利用Ribbon维护了服务提供者的服务列表信息，并且通过轮询实现了客户端的负载均衡。而与Ribbon不同的是，通过feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用。

②上述第二步相当于该接口通过接口上的服务名以及方法上的url，去向远程服务提供端发送访问数据请求

③Feign通过接口的方法调用Rest服务

# Hystrix断路器

## 7.1 扇出

多个微服务之间调用的时候，如微服务A调用微服务B和C，B和C又去调用其它的微服务，这就是所谓的扇出。

## 7.2 Hystrix概述

Hystrix是一个用于分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

## 服务熔断(服务提供端进行相应的配置操作)

### 7.3.1 概述

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回”错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阙值，缺省是5秒内20次调用失败就会启动熔断机制。

### 7.3.2 使用

1. 在服务提供方导入相应的Hystrix的相关依赖
2. 在Controller方法上@HystrixCommand注解，配置该方法调用出现异常后调用fallbackMethod指定的备选方法。如@HystrixCommand(fallbackMethod=”aaa”)。
3. 在主启动类上加上@EnableDisCircuitBreaker注解

## 服务降级(消费端进行相关的配置)

### 7.4.1 概述

整体资源快不够用了，将某些服务先关掉，待渡过难关后，再开启这些服务。

### 7.4.2 使用

①在完成Feign整合的api工程中根据已有的服务接口新建一个实现了FallbackFactory<该接口名>的实现类。

注：一定要在该类上添加spring注入注解，将其注入至IOC容器

②在该类里面的create方法中返回一个新创建的实现该接口的内部类，每一个实现的方法则为服务出问题后的回调方法。

③在该接口的@FeignClient注解里加上fallbackFactory=②编写的实现类.class即可。

④消费端的yml中配置启用Hystrix的相关配置。

## 服务监控(Hystrix Dashboard)

### 概述

微服务的准实时调用监控，Hystrix会持续的记录所有通过Hystrix发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求，多少成功，多少失败等。

### 7.5.2使用

①新建一个项目，并导入相关依赖

②编写配置文件，只需要配置一个服务端口即可

③主启动类上添加@EnableHystrixDashboard注解

④所有的微服务提供端都需要配置Actural的依赖

⑤启动该项目，并且访问该项目，并在页面上配置多长时间刷新页面，以及需要监控的服务(http://ip:port/hystrix.stream)等。则在访问监控的服务时，该项目会记录相关的监控信息。

监控信息主要包括：七色：和图上的标题字体颜色对上

一圈：是一个实心圆，通过颜色的变化反应实例的健康程度，健康色为绿色<黄色<橙色<红色。并且它的大小也会根据实例的流量请求发生变化，流量越大该实心圆就越大。故可以通过实心圆快速的发现故障实例和高压力实例。

一线：记录设置时间内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

注：若输入服务提供者的hystrix.stream显示无资源，需在配置类中注入该类。

@Bean  
**public** ServletRegistrationBean getServlet(){  
  
 HystrixMetricsStreamServlet streamServlet = **new** HystrixMetricsStreamServlet();  
  
 ServletRegistrationBean registrationBean = **new** ServletRegistrationBean(streamServlet);  
  
 registrationBean.setLoadOnStartup(1);  
  
 registrationBean.addUrlMappings(**"/hystrix.stream"**);  
  
 registrationBean.setName(**"HystrixMetricsStreamServlet"**);  
  
 **return** registrationBean;  
}

# 8. zuul(路由网关)

## 8.1 概述

Zuul包含了对请求的路由和过滤两个最主要的功能。其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部统一入口的基本。而过滤器的功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获取其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

## 8.2 路由基本配置

①新建一个zuul工程

②导入相关依赖(zuul和eureka)

③yml的配置

④主启动类加上@EnzbleZuulProxy

⑤访问<http://zuul>工程ip:port/服务提供者名/请求url->通过zuul访问服务

## 8.3 路由访问映射规则

①zuul工程中application.yml中增加zuul.routes.mydept.serviceId=原服务名、zuul.routes.mydept.serviceId=现在映射的服务名/\*\*。

②zuul工程中application.yml中增加zuul.ignore-services=原服务名，则可屏蔽原服务名访问服务，只能用现在暴露的服务名进行服务的访问。

注：若需要禁掉多个微服务的名字，则可用”\*”屏蔽。

③zuul工程中application.yml中增加zuul.prefix =前缀名。->设置访问服务的前缀。

# 9 config(分布式配置中心)

## 9.1 概述

为微服务架构中的微服务提供集中式的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

Config分为server服务端和client客户端两部分。

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，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过制定的配置中心来管理应用资源，以及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。配置服务器默认采用git来存储配置信息，这样有助于对环境配置进行版本管理，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。

## 9.2 服务端配置

①在github上新建一个项目

②获取上一步ssh协议的git地址

③本地上新建git本地库(不初始化)，并clone上一步的地址

⑤在当前git库新建一个application.yml配置文件(注：保存格式为UTF-8)

⑥将上一步的yml配置文件推送到github上

⑦新建config服务端配置工程，导入config-server的mvn坐标、config的git插件的mvn坐标(可导可不导)，并编写该工程的配置文件以及启动类上加上@EnableConfigServer注解。

⑧启动该配置工程，并通过url进行访问。方式如http:// config服务端配置工程ip:port/application-开发环境.yml即可获取github上的配置信息。

访问格式：/master/application-dev.yml，/applation/dev/master(master表示master分支)

注意：需在idea或eclipse配置github的用户名、密码以及application.yml中config的url为http协议的该请求。

## 9.3 客户端的配置

①新建config客户端工程，导入config的mvn坐标。

②在resources目录下新建一个bootstrap.yml，内容为spring.cloud.config.(name[要访问的配置文件名，不带.yml]、profile[环境名]、label[分支标签]、uri[config服务器的uri地址]。

注：application.yml是用户级的资源配置项，bootstrap.yml是系统级的，优先级更高。Springcloud会创建一个Bootstrap Context，作为Spring应用的Application Context的父上下文。初始化的时候，Bootstrap Context负责从外部源加载配置属性并解析配置。这两个上下文共享一个从外部获取的环境，Bootstrap属性有高优先级，默认情况下，他们不会被本地配置覆盖。Bootstrap Context和Application Context有着不同的约定，所以新增了一个bootstrap.yml文件，保证Bootstrap Context和Application Context配置的分离。

③可以通过spring注入对象的@Value注解和EL表达式注入相应的配置属性。[如@Value(“${server.port}”)](mailto:如@Value()。

注：客户端配置文件里面配置的profile是啥，就从要访问的配置文件中获取相应的profile的配置信息。

注：服务发现就是对注册进Eureka的微服务，可以通过服务发现来获得该服务的信息。也就是服务提供者需要向别人展示自己的简介一样的功能。需要借助DisCoveryClient实例，并在主启动类上加上@EnableDisCoveryClient注解来发现服务。并在消费端工程的Controller调用相应的服务提供者提供的服务发现url进而完成服务发现的功能。