# SpringCloud概述

## 为什么使用SpringCloud

Spring Cloud提供了一套完整的微服务整体解决方案。提供了分布式配置中心、分布式锁、分布式跟踪、分布式服务治理、分布式任务调度平台等模块解决对应的问题。

## 什么是SpringCloud

Spring Cloud是一个分布式的整体解决方案。Spring Cloud 为开发者提供了在分布式系统（配置管理，服务发现，熔断，路由，微代理，控制总线，一次性token，全局琐，leader选举，布式session，集群状态） 中快速构建的工具，使用Spring Cloud的开发者可以快速的启动服务或构建应用、同时能够快速和云平台资源进行对接。

**SpringCloud分布式常用组件**

服务发现一 Netflix Eureka

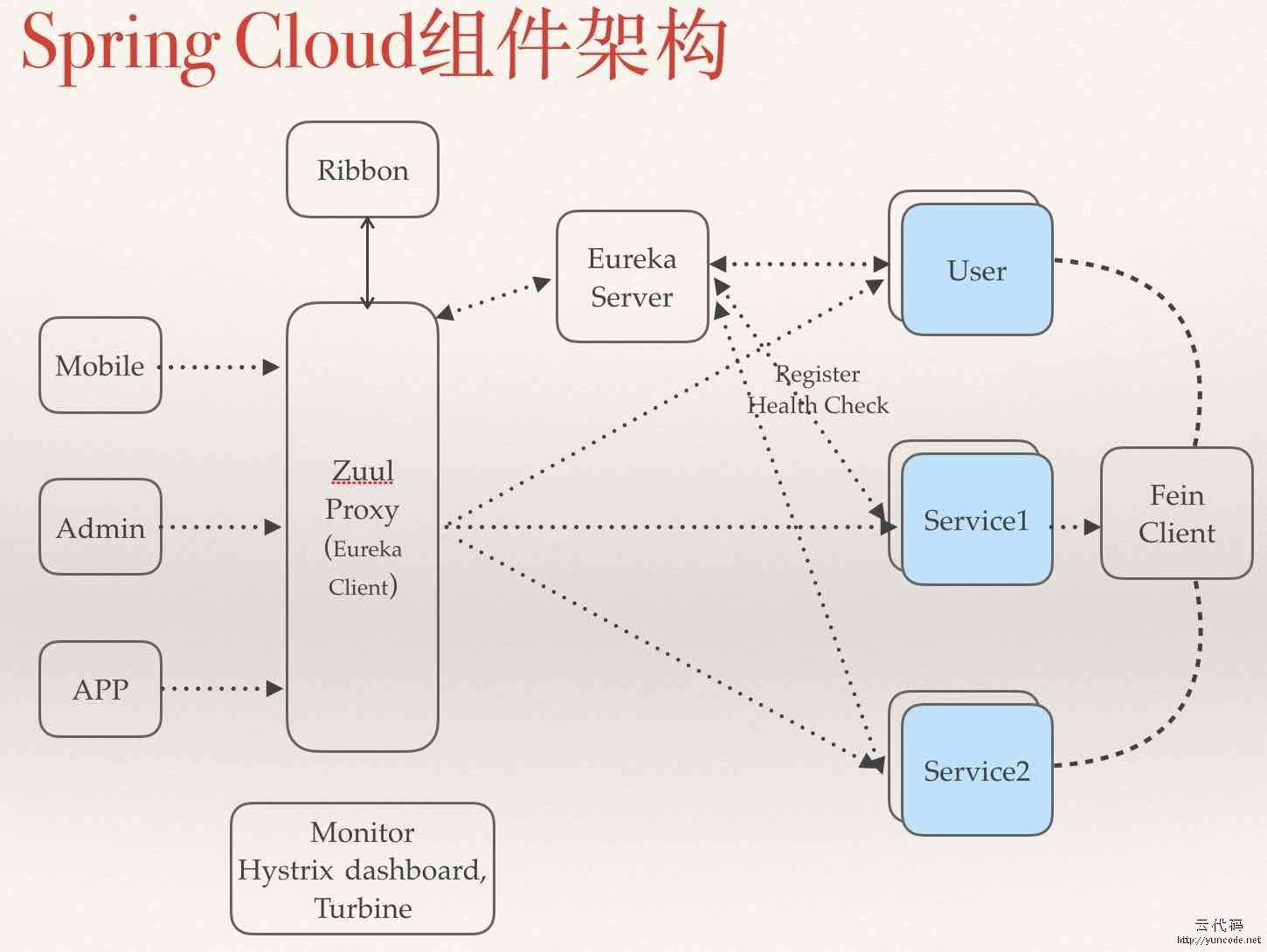
客服端负载均衡一Netflix Ribbon

断路器-Netflix Hystrix

服务网关-Netflix Zuul

分布式配置中心一Spring Cloud Config

服务调用组件—Feign



# 服务发现与注册

一个应用拆分为多个独立微服务。多个微服务可以使用Docker容器化进行多实例部署。

架构演化到这里遇到了一个难题，如果要查询用户所有的订单，用户服务可能会依赖订单服务，用户服务如何与订单服务交互呢？订单服务有多个实例该访问哪一个？服务注册和发现成为了一种很好的解决方案。

[**微服务**](https://so.csdn.net/so/search?q=%E5%BE%AE%E6%9C%8D%E5%8A%A1&spm=1001.2101.3001.7020)**架构中最核心的部分是服务治理**，服务治理最基础的组件是注册中心。随着微服务架构的发展，出现了很多微服务架构的解决方案，其中包括我们熟知的 Dubbo 和 Spring Cloud。

在SpringCloud 中提供了多种服务注册与发现组件：Eureka,Consul,Zookeeper。官方推荐使用Eureka。

在dubbo 支持了 Zookeeper、Redis、Multicast 和 Simple，官方推荐 Zookeeper。

## 相关概念

注册中心概念：注册中心是存放服务地址相关信息。

SpringCloud中支持Eureka、Consul、Zookeeper注册中心。

Dubbo支持常用的Redis和Zookeeper。

在传统架构中，服务与服务之间调用直接使用地址调用。

服务提供者：提供服务接口；

服务消费者：调用接口进行使用；

一个服务既可以做服务者，也可以作为消费者。

服务注册：将服务信息注册到注册中心上。

服务发现：从注册中心上获取服务信息。

## 基本步骤：

1. 首先启动注册中心
2. 启动服务提供者
3. 服务提供者启动的时候，会把当前服务的基本信息如服务地址和端口以别名方式注册到注册中心去。
4. 消费者在调用接口的时候使用别名去注册中心获取实际的RPC远程调用地址。
5. 如果消费者获取到了实际的RPC远程调用地址后，使用本地的HttpClient技术实现调用。默认情况下，eureka每隔30秒更新一次服务调用地址。

## 服务注册和发现搭建

## 搭建注册中心

**1.创建Springboot项目工程，并在父项目中添加springCloud依赖**

<dependencyManagement>

<dependencies>

<!-- springCloud核心包 -->

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>2021.0.4</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

**2.创建explore-eureka-server模块项目并添加对应依赖**

<!-- eureka server-->

<dependency>

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

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

</dependency>

**3.配置eureka注册中心**

server:

port: 9000 #服务注册中心端口号

spring:

application:

name: explore-eureka-server

eureka:

instance:

hostname: 127.0.0.1 #服务注册中心IP

client:

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka #注册地址

register-with-eureka: false #是否往服务注册中心注册自己，默认为true。单机版的可设置成false。集群搭建中，设置true。

fetch-registry: false #单机版的可设置成false，集群版的由于需要同步其他节点的服务注册数据，故设成true。

server:

enable-self-preservation: false #是否开启自我保护模式，默认值true

eviction-interval-timer-in-ms: 60000 #扫描失效服务的间隔时间，单位毫秒，默认值 60 \* 1000

**4.配置启动类**

@SpringBootApplication

@EnableEurekaServer

**public** **class** EurekaApp {

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

SpringApplication.*run*(EurekaApp.**class**, args);

}

}

## 搭建服务提供者

**1.创建服务模块(explore-member-service)并父项目中添加依赖**

<dependency>

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

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

</dependency>

**2.配置eurek将服务注册到eureka注册中心**

server:

port: 9010

spring:

application:

name: explore-member-service

profiles:

active: dev

eureka:

client:

service-url:

defaultZone: http://@eureka.instance.hostname@:@server.eureka.port@/eureka

instance:

prefer-ip-address: true

**3.配置启动类**

@EnableEurekaClient

@SpringBootApplication

**public** **class** MemberApp {

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

SpringApplication.*run*(MemberApp.**class**, args);

}

}

## 搭建Eureka集群环境

如果注册中心应为某一个原因出现故障了会导致整个微服务环境不可用，应该如何解决？解决办法：搭建注册中心集群

Eureka搭建集群原理使用相互注册原理，形成一组相互注册的注册中心，从而实现数据相互同步，达到高可用效果。

## Eureka自动保护机制

1. 服务提供者启动时，会把当前服务信息注册到eureka中；
2. 服务消费者调用服务提供者接口时，会去注册中心获取服务提供者实际RPC远程调用地址，获取地址后，在本地使用httpClient术进行远程调用。
3. Eureka自我保护机制：

默认情况下,EurekaClient定时向EurekServer端定时发送心跳包。

如果EurekaServer端在一定的时间内没有收到EurekaClient发送心跳包，便会直接从服务注册列表中剔除该服务。但是在短时间内丢失大量的服务实例心跳。这时候EurekaServer会开启自我保护机制，不会剔除该服务。

为了防止EurekaClient可正常运行，但是与EurekaSever网络不通的情况下，EurekaServer不会将EurekaClient服务进行剔除。

4.在什么环境下开启自我保护机制

建议在本地环境中禁用自我保护机制,在开发环境中开启自我保护机制。

如果服务真的宕机了，本地调用应该使用重试机制、保证接口网络延迟幂等性、服务降级功能。

## SpringCloud整合Zookeeper作为注册中心

Zookeeper:分布式调用工具，可以实现注册中心；采用零时节点方式注册服务，零时节点和生命周期进行关联，如果服务断开连接后，给节点会自动被删除。

安装zookeeper工具

创建服务消费者

## Zookeerp整合Ribbon负载均衡

使用RestTemplate做远程调用；

SpringCloud整合Consul注册中心

安装consul注册中心

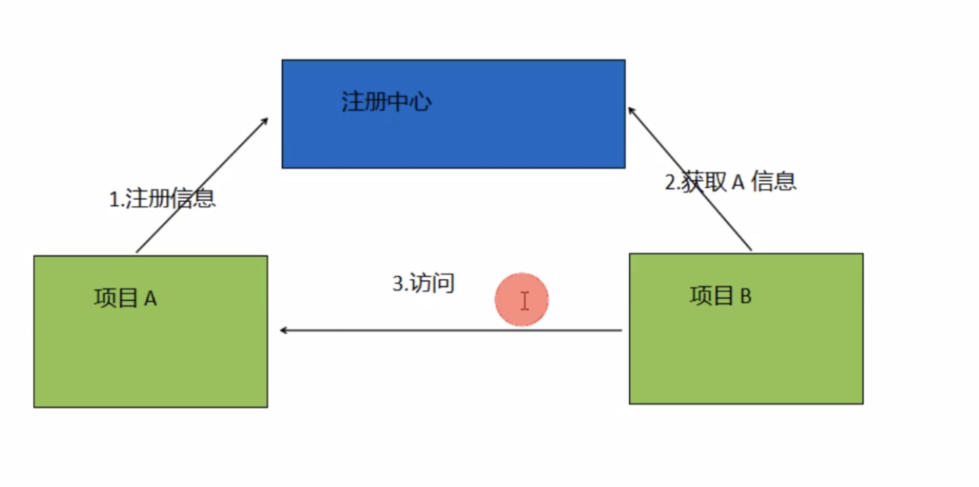
创建服务消费者

# Eureka

## 什么是Eureka

Eureka是Netflix公司退出的服务注册和发现工具（注册中心/服务治理），已被springcloud集成，提供了开箱即用的的支持。

## 包含注册中心的软件简易架构图



注册中心在项目中的意义：

实现应用程序解耦。项目之间不是直接进行项目通信。项目A把自己服务器的IP、端口等信息注册到注册中心中，项目B从注册中心中获取项目 A 的信息后再进行访问项目 A。

项目发布所在服务器地址可以随意变化。如果在项目 B 中进行编码写项目A的IP和端

口，当项目 A 的服务器地址发生变化或项目 A 重新部署到其他服务器上时，重新编码项目 B中的IP和端口。

借助注册中心可以实现负载均衡等效果。

## Eureka中的角色

Eureka 中分为两个角色：Eureka Server （Eureka 服务）和Eureka Client (Eureka

客户端）。

无论是服务端还是客户端其本质都是一个Java 项目，在Spring Cloud 中主要通过启

动类上添加@EnableEurekaServer 和@EnableEurekaClient(可以省略）来区分当前应用

程序是服务端还是客户端。

Eureka Server 可以理解成zookeeper 注册中心，只是现在使用的是Java 项目实现的（spring cloud内嵌Eureka）。

Eureka Client 可以理解成所有需要注朋到 Eureka Server 中的项目。

## CAP 理论（分布式一致性定理）

著名的CAP理论指出，一个分布式系統不可能同时满足 C(一致性）、A(可用性)和P(分

区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在 A 和C之间进行权衡。而Eureka 则是 AP.

C(一致性 Consistency)：在分布式系统中，是否立即达到数据同步的效果 （平时多说的强一致性）。在分布式系統一般都是最終会一致的。如果请求时，整个分布式系統同步后才返回

结果，叫做强一致性（满足一致性）。如果先返回结果，在一定时间后才实现一致性就叫做

弱一致性。就不满足C的要求。

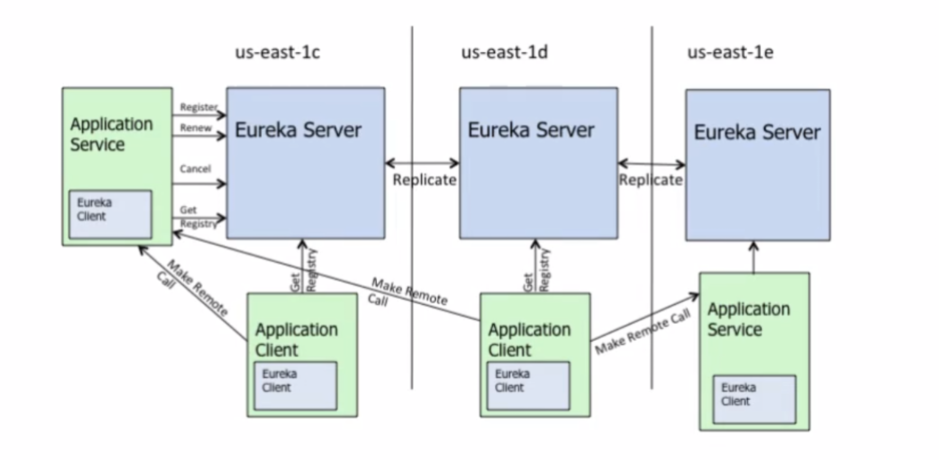
A（可用性 Availabilty)：在分布式系统中，其中一些节点出现问题，整个整体是否还

可用。

P（分区容错性）：在分布式系统中，是否可以在有限的时间内达到数据一致的效果，如果因为网络等问题最终没有达到一致性，这时称为出现分区错误。

## 分布式和集群的区别

## Eureka集群实现原理



所有 Eureka Server 通过 Replicate 进行数据同步。无论Eureka Client 向哪个Eureka

Server 中汪册信息，最終所有Eureka Server 中都会存储注册的信息，这些信息都缓存到

Eureka Server 的本地。每个Eureka Server 中同步后的数据都是完全相同的。

Eureka Client 向 Eureka Server 注册信息的时候我们称它为 Application Service, 当

获取汪册的信息时称为 Application Client， 由于可能出现某个 Eureka Client 即需要注册

服务，又需要获取其他服务，所以很多 Eureka Client 既是 Application Service也是 Application client。

Eureka Client 启动后，每隔30 秒向 Eureka Server 发送一次心跳，证明自己的可用

通过eureka.instance.lease-renewal-interval-in-seconds=30 修改心跳的间隔时间。当Eureka Server 超过 90秒没有收到提供者的心跳后，会认为这个提供者已经宕机，会销毁实例。可以通过eureka.instance.1ease-expiration-duration-in-seconds=90修改时间。

如果重新启动 Eureka Server ，会销毁所有实例。

## Eureka自我保护机制

Eureka 中有一种自我保护机制。当15分钟内超过 85%的Eureka Client 都没有正常

的心跳包时，Eureka 认为 Server 和 Client 之间出现了网络问题。这个时候将不在因为没

有收到心跳而销毁实例。Eureka Client 依然可以访问 Server，但是Server 不会把内容同

步到其他 Server 中。当网络稳定后：Server 会把注册的信息同步到其他 Server中。

在Eureka 中自我保护机制默认为开启的：eureka.server.enable-self-preservation=true

## 关闭自我保护机制

eureka:

server:

enable-self-preservation: false #是否开启自我保护模式，默认值true

eviction-interval-timer-in-ms: 10000 #扫描失效服务的间隔时间，单位毫秒，默认值 10 \* 1000

## Eureka 保证 AP

在Eureka 集群中所有的节点都是保存完整的信息的，当 Eureka Client 向 Eureka 中

注册信息时，如果发现节点不可用，会自动切换到另一台 Eureka Sever， 也就是说整个集群中即使只有一个 Eureka 可用，那么整个集群也是可用的。

同时 Eureka 的自我保护机制也是实现可用性非常重要的体现。

## 搭建Eureka-Server

1.导入依赖

添加Springboot依赖合Spring cloud依赖集成Eureka server依赖。

<dependencyManagement>

<dependencies>

<!-- springCloud核心包 -->

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>2021.0.4</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- eureka server-->

<dependency>

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

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

</dependency>

2.编写配置文件

server:

port: 9000 #服务注册中心端口号

spring:

application:

name: explore-eureka-server

eureka:

instance:

hostname: 127.0.0.1 #服务注册中心IP

client:

service-url:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka #注册地址

register-with-eureka: false #是否往服务注册中心注册自己，默认为true。单机版的可设置成false。集群搭建中，设置true。

fetch-registry: false #单机版的可设置成false，集群版的由于需要同步其他节点的服务注册数据，故设成true。

server:

enable-self-preservation: false #是否开启自我保护模式，默认值true

eviction-interval-timer-in-ms: 60000 #扫描失效服务的间隔时间，单位毫秒，默认值 60 \* 1000

**3.配置启动类**

@SpringBootApplication

@EnableEurekaServer

**public** **class** EurekaApp {

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

SpringApplication.*run*(EurekaApp.**class**, args);

}

}

## 搭建Eureka-Client

## Eureka集群搭建

1.导入依赖

添加Springboot依赖合Spring cloud依赖集成Eureka server依赖。

2.编写配置文件

server:

port: 9000 #服务注册中心端口号

spring:

application:

name: explore-eureka-server

eureka:

instance:

hostname: eureka-server1 #服务注册中心主机名

client:

service-url:

defaultZone: #eureka集群中的其他集群的主机名和端口号的

http:// eureka-server2:9001 /eureka,

http:// eureka-server3:9002/eureka,

http:// eureka-server4:9003/eureka #注册地址

register-with-eureka: false #是否往服务注册中心注册自己，默认为true。单机版的可设置成false。集群搭建中，设置true。

fetch-registry: false #单机版的可设置成false，集群版的由于需要同步其他节点的服务注册数据，故设成true。

server:

enable-self-preservation: false #是否开启自我保护模式，默认值true

eviction-interval-timer-in-ms: 60000 #扫描失效服务的间隔时间，单位毫秒，默认值 60 \* 1000

# RestTemplate

# OpenFegin

## 什么是OpenFegin

平常我们说的feign是指的Netflix下的feign。OpenFegin前是springcloud的二级子项目，是Spring提供的。

OpenFegin是一种声明式、模板化的客户端。声明式调用是指，就像调用本地方法一样调用远程方法，无需感知操作远程RPC请求。

Spring cloud的声明式调用，可以做到使用http请求远程服务时能像调用本地方法一样的体验，开发者完全感知不到这是远程方法，更感知不到这是一个http请求。Fegin的应用中，Application client直接通过接口方法调用Application service.而不需要通过常规的RestTemplate构造请求解析返回数据。他解决了让开发者调用远程接口就像调用本地方法一样，无效关注远程的交互细节，更无需关注分布式环境开发。

使用OpenFegin是就好像在写控制器方法，**OpenFegin都是写在接口中，在声明的方法上添加Springmvc注解或声明的参数上添加springmvc注解就可以完成调用远程控制器方法。**

## 使用OpenFeign时程序执行流程

OpenFeign 代替RestTemplate 代码。也是写在 Application Client 中。把 OpenFeign接口单独放在 feign 包中，表示服务调用层。当需要调用其他服务时，直接注入 penFeig接口对象就可以像调用本地方法一样调用远程服务。

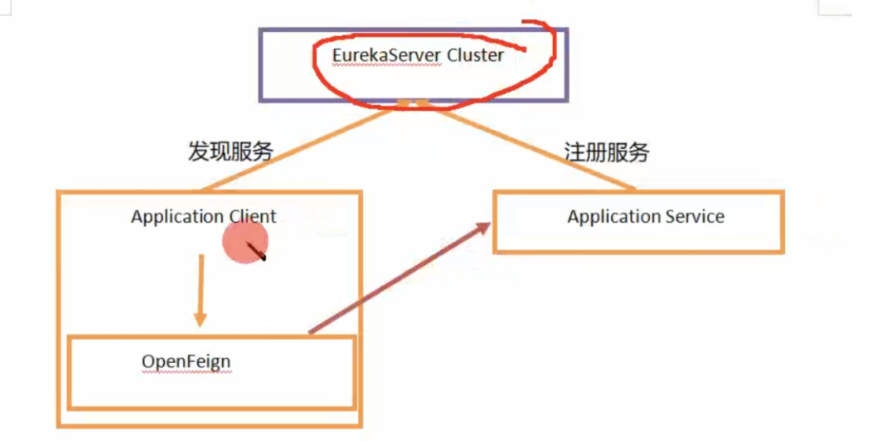
整体流程说明：

1. Applicationservice 向 Eureka Server 注册服务.

2. Application Client 从 Eureka Server 中发现服务信息。

3. 在 Application Client 中调用 OpenFeign 接口中方法

4. Application Client 中 OpenFeign 通过应用程序名调用 Application Service



## 实例代码

1.添加依赖

<dependency>

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

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

</dependency>

1. 接口上编写@FeignClient

@FeignClient(name = "explore-member-service")

**public** **interface** TestApi {

@RequestMapping("test/feign")

String openFeign();

}

3.调用方启动类上添加@EnableFeignClients

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

**public** **class** ShiroApp {

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

SpringApplication.*run*(ShiroApp.**class**, args);

}

## 遇到问题

**1.当接口上配了 FeignClient 和 RequestMapping 两个注解，结果错误提示 重复mapping处理方法**

@FeignClient(name = "explore-member-service")

@RequestMapping("test")

**public** **interface** TestApi {

@RequestMapping("/feign")

String openFeign();

}

@Controller

@RequestMapping("test")

**public** **class** TestController {

@Autowired

TestApi testApi;

@ResponseBody

@RequestMapping("/feign")

**public** String testFeign() {

String openFeign = testApi.openFeign();

**return** openFeign;

}

}

**解决方法：**

**将类上的RequestMapping注解去掉，然后把路径放在方法上的RequestMapping**

@FeignClient(name = "explore-member-service")

**public** **interface** TestApi {

@RequestMapping("test/feign")

String openFeign();

}

## OpenFeign通讯优化

### 1 GZIP 简介

gzip 介绍：gzip 是一种数据格式，采用用 delate 算法压缩数据;gzip 是一种流行的数据压缩算法，应用十分广泛，尤其是在Linux平台。

gzip 能力：当 Gzip 压缩到一个纯文本数据时，效果是非常明显的，大约可以减少 70%以上的数据大小。

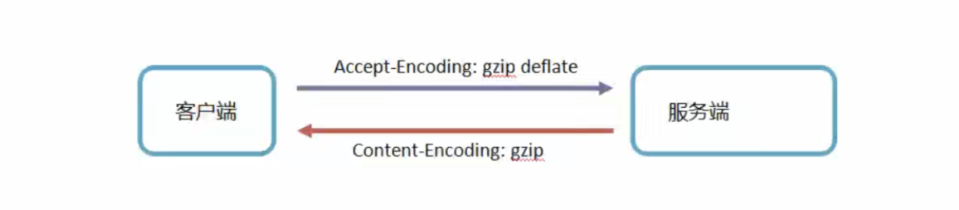
Bzip作用：网络数据经过压缩后实际上降低了网络传输的字节数，最明显的好处就是

可以加快网页加载的速度。网页加载速度加快的好处不言而喻，除了节省流量，改善用户

的浏览体验外，另一个潜在的好处是 Gzip 与搜素引擎的抓取工具有着更好的关系。例如

Google 就可以通过直接读取 Bzip 文件来比普通手工抓取更快地检索网页。

### Http协议中关于压缩传输的规定(原理)



第一：客户端向服务器请求头中带有：Accept-Encoding:gzip, deflate 字段，向服务器

表示，客户端支持的压缩格式（gzip 或者 deflate)，如果不发送该消息头，服务器是不会压缩的。

第二：服务端在收到请求之后，如果发现请求头中含有 Accept-Encoding 宇段，并目支

持该类型的压缩，就对响应报文压缩之后返回给客户端，并目携带 Content-Encoding:8zip

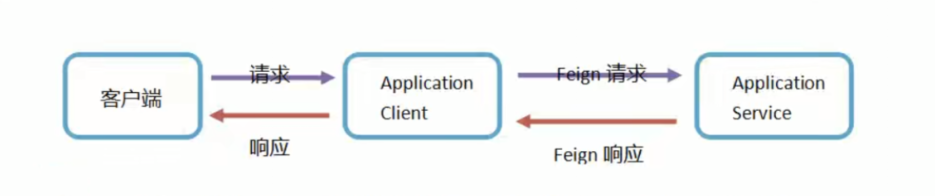
消息头，表示响应报文是根据该格式压缩过的。

第三：客户端接收到响应之后，先判断是否有 Content -Encoding 消息头，如果有，按

该格式解压报文。否则按正常报文处理。

### 在openFeign技术中应用GZIP压缩

在SprngCloud微服务体系中，一起请求的完整流程如下



在整体流程中，如果使用 GZ1P 压缩来传输数据，涉及到两次请求-应答。而这两次请

求-应答的连接点是 Application Client， 那么我们需要在 Application Client 中配置开启 GZIP压缩，来实现压缩数据传输。

### 配置OpenFeign请求-应答过程中的GZIP压缩

在交互数据量级不够的时候，看不到压缩内容。这里只开启 Feign 请求-应答过程中的 GzIP，也就是浏览器-Application Client 之间的请求应答不开启 GZIP 压缩。

在全局配置文件中，使用下述配置来实现 Open Feign 请求-应答的 GZIP 压缩

# feign gzip

＃ 开启请求 GzIP

feign.compression.request.enabled=true

＃ 开启响应 GZIP

feign.compression.response.enabled=true

＃ 设置支持 GZIP 压缩的 MIME 类型，即请求/响应类型。

feign.compression.request.mime-types=text/xml,application/xml,application/ison

＃配置启动压缩数据的最小阈值，单位字节。默认为 2048

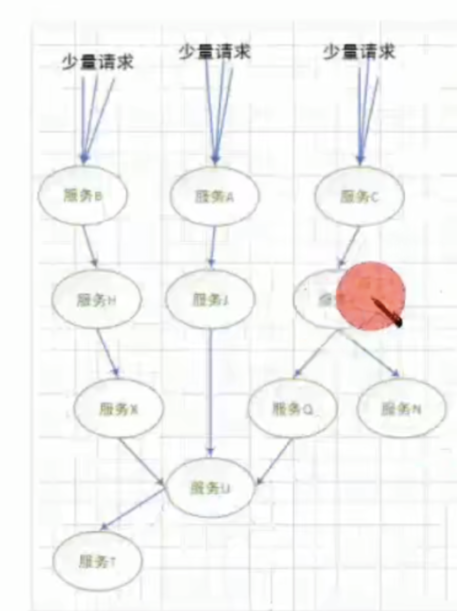
feign.compression.request.min-request-size=512

# Hystrix

## 什么是灾难性雪崩效应

在微服务架构项目中，尤其是中大型项目，肯定会出现一个服务调用其他服务，其他服务又调用别的服务，服务与服务之间形成了一中链式调用关系。

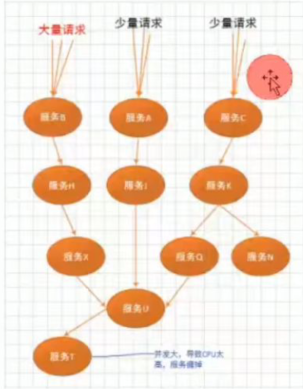
当少量请求时，对于整个服务链条是没有过多的影响的。



虽然每个服务的请求都是少量的，但是最终都会访问服务T,所有对服务T来说请求量都是比较大的，所在服务器的CPU压力比较高。



当其中某一个服务突然遇到大量请求时，整个链条上所有服务负载骤增，导致服务U和T的负载过高，运行性能下降，会导致其他调用服务U和T的链条出现问题,从而导致所有的项目都出现问题，这种情况称为灾难性的雪崩效应。



## 造成灾难性雪崩效应的原因

造成灾难性雪崩效应的原因，可以简单归结为下述三种

1.服务提供者 (Application Service）不可用。如：硬件故障、程序BUG，缓存击穿、并

发请求量过大等.

2.重试加大流量。如：用户重试：代码重试逻辑等。

服务调用者 （Application Client)不可用。如：同步请求阻塞造成的资源耗尽等

雪崩效应最终的结果就是：服务链条中的某一个服务不可用，导致一系列的服务不可用，

最终造成服务 逻辑崩溃。这种问题造成的后果，往往是无法预料的。

## 如何防止灾难性雪崩效应

**降级(每次请求都会访问访问另外一个服务)**

超时降级、资源不足时（线程或信号量）降级，降级后可以配合降级接口返回托底数据。

实现一个fallback 方法，当请求后端服务出现异常的时候，可以使用 fallback 方法返回的值.

保证：服务出现问题整个项目还可以.

**熔断**（达到一定条件后，在指定时间内不会访问另外一个服务）

当失败率(如因网络故障/超时造成的失败率高)达到阀值自动触发降级，熔断器触发的快

速失败会进行快速恢复。

通俗理解：熔断就是具有特定条件的降级，当出现熔断时在设定的时间内就不在请求

Application Service 了。**所以在代码上熔断和降级都是一个注解。**

保证：服务出现问题整个项目还可以继续运行。

**请求缓存（请求走缓存）**

提供了请求缓存。服务 A调用服务 B，如果在 A 中添加请求缓存，第一次请求后走缓存

了，就不在访问服务 B了，即使出现大量请求时，也不会对B产生高负载。请求缓存可以使用 Spring Cache 实现。

保证：减少对Application Service的调用。

**请求合井（一段时间内，多个请求合并为一个请求）**

提供请求合并。当服务 A 调用服务 B时，设定在5 毫秒内所有请求合并到一起，对于

服务B的负载就会大大减少，解决了对于服务B负载激增的问题。

保证：減少对 Application Service 的调用

**隔离**

隔离分为线程池隔离和信号量隔离。通过判断线程池或信号量是否已满，超出容量的请

求直接降级，从而达到**限流**的作用。

## 降级

降级是指，当请求超时、资源不足等情况发生时进行服务降级处理，不调用真实服务

逻辑，而是使用快速失败(fallback）方式直接返回一个托底数据，保证服务链条的完整，

避免服务雪崩。

解决服务雪崩效应，都是避免 application client 请求 application service 时，出现服务

调用错误或网络问题。处理手法都是在application client 中实现。我们需要在 application

client 相关工程中导入hystrix 依赖信息。并在对应的启动类上增加新的注解

@EnableCircuitBreaker，这个注解是用于开启 hystrix 熔断器的，简言之，就是让代码中的

hystrix 相关注解生效。

### 代码实现

远程服务访问错误：远程服务不可用或远程服务超时。服务可用：有返回结果或有异常传递。

默认情况下SpringCloud远程访问，不管远程访问多长时间，SpringCloud都会等待。想要连接超时可以在启动类上使用@EnableHystrix注解，开启Hystrix容错处理能力。Hystrix默认设置一个远程访问超时时间配置，默认为1秒。

在服务方法上添加@HystrixCommand注解。代表该方法是一个需要做容错处理的方法。默认设置远程超时1秒。fallbackMethod属性代表如果远程调用发生错误(远程服务不可以或超时)，则调用对应的方法并返回。

为需要服务降级的方法编写降级方法，设置远程服务调用错误是返回的托底数据。降级方法要求：

1. 返回值和对应服务方法的返回值类型必须一致；
2. 参数列表和对应服务方法必须一致；

注意：

**1.降级每次远程请求都会调用服务方法；**

@ResponseBody

@RequestMapping("/hystrixDegradation")

@HystrixCommand(fallbackMethod = "hystrixDegradationBack")

**public** String hystrixDegradation() {

***logger***.info("本地降级方法执行-----------------------");

String openFeign = testHystrixtApi.hystrixTime();

**return** openFeign;

}

/\*\*

\* 降级方法要求:

\* 1.返回值和对应的服务方法返回类型必须一致；

\* 2.方法的参数列表和对应服务方法一致；

\* **@return**

\*/

**public** String hystrixDegradationBack() {

**return** "服务繁忙，请稍后再试！";

}

## 熔断

熔断

当一定时间内，异常请求比例（请求超时、网络故障、服务异常等） 达到阀值时，启

动熔断器，熔断器一旦启动，则会停止调用具体服务逻辑，通过 faillback 快速返回托底数

据，保证服务链的完整。

熔断有自动恢复机制，如：当熔断器启动后，每隔5秒，尝试将新的请求发送给

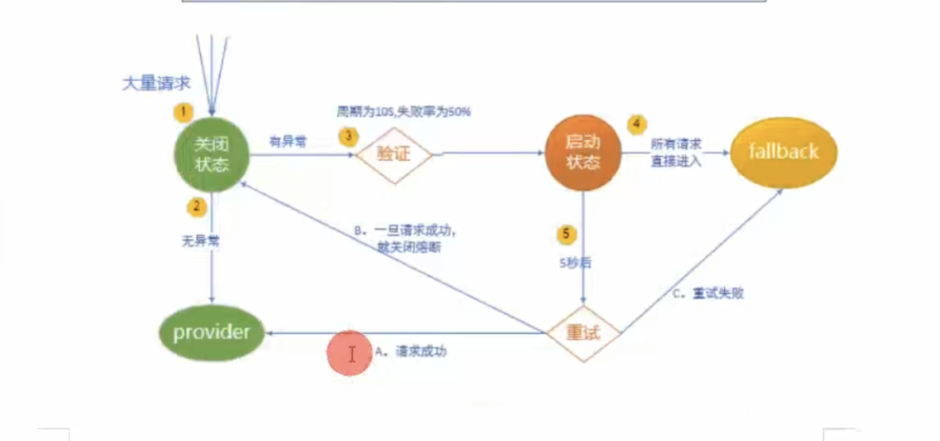
Application Service，如果服务可正常执行并返回结果，则关闭熔断器，服务恢复。如果仍

旧调用失败，则继续返回托底数据，熔断器持续开启状态。

降级是出错了返回托底数据，而熔断是出错后如果开启了熔断格会一定时间不在访问

application service

熔断机制相当于电路的跳闸功能。例如：我们可以配置熔断策略为当请求错误比例在10s内>50%时，该服务将进入熔断状志，后续请求都会进入fallback.



### 属性

1属性

熔断的实现是在调用远程服务的方法 上增加@HystrixCommand 注解。当注解配置满足

则开启或关闭熔断器。

@HystrixProperty 的name 属性取值可以使用HystrixPropertiesManger常量，也可以直接使用字符串进行操作。

注解属性描述：

CIRCUIT BREAKER ENABLED

"circuitBreaker.enabled";

是否开启熔断策略。默认值为 true。

CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD

"circuitBreaker.requestVolumeThreshold";

单单位时间内（默认 10s 内） ，请求超时数超出则触发熔断策略。默认值为 20次请求

数。通俗说明：单位时间内容要判断多少次请求。

EXECUTION\_ISOLATION\_THREAD\_TIMEOUT\_ IN\_MILLISECONDS

execution.isolation.thread.timeoutInMilliseconds

单位时间内，判断execution.isolation.thread.timeoutInMilliseconds的时间单位，默认10秒，单位毫秒。

CIRCUIT\_ BREAKER\_ SLEEP\_ WINDOW\_ IN\_ MILLISECONDS

"circuitBreaker.sleep WindowlnMilliseconds":

当熔断策略开启后，延迟多久尝试再次请求远程服务。默认为5秒。单位毫秒。这5

秒直接执行 fallback 方法，不在请求远程 application service.

CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE

"circuitBreaker.error ThresholdPercentage":

单位时间内，出现错误的请求百分比达到限制，则触发熔断策略。默认为 50%

CIRCUIT BREAKER FORCE OPEN

"circuitBreaker.forceOpen"

是否强制开启熔断策略。即所有请求都返回 fallback托底数据。默认为 false。

CIRCUIT BREAKER FORCE CLOSED

"circuitBreaker.forceClosed".

是否强制关闭熔断策略。即所有请求一定调用远程服务。默认为 false。

### 实例代码

**熔断是强化版的降级。**

**在降级的基础上配置属性commandProperties**

**@HystrixProperty(name=“”，value=””)**

**HystrixProperty提供了一个服务容错处理配置，name是要配置的属性；value是具体配置的数组，name属性，可以参考一个工具类XystrixPropertiesManger类。**

### 1.导入hystrix

<!-- hystrix -->

<dependency>

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

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

<version>${spring-cloud-hystrix.version}</version>

</dependency>

### 2.在application client请求方法上添加@HystrixCommand注解

@ResponseBody

@RequestMapping("/hystrixFusing")

//如果10秒内，失败率达到请求次数(2)的50%，也就是1次就会就会开启熔断。

@HystrixCommand(fallbackMethod = "reqTimeout",commandProperties = {

//@HystrixProperty提供了一个服务容错处理配置，name为要配置的属性，value是具体配置数据， name属性可以参考工具类HystrixPropertiesManager中的常量或直接使用字符串;

//CIRCUIT\_BREAKER\_ENABLED：是否开启熔断策略。默认值为 true。

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_ENABLED***,value ="true"),

//CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD:在单位时间内(10秒),发送远程访问错误的次数阈值，达到开启熔断。默认值为 20次请求

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_REQUEST\_VOLUME\_THRESHOLD***,value="2"),

//CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE:在单位时间内(10秒),发送远程访问错误的百分比，达到阈值开启熔断。默认50%

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_ERROR\_THRESHOLD\_PERCENTAGE***,value="50"),

//CIRCUIT\_BREAKER\_SLEEP\_WINDOW\_IN\_MILLISECONDS:开启熔断后，多少毫秒内不发起远程访问.默认5秒

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_SLEEP\_WINDOW\_IN\_MILLISECONDS***,value="5000"),

//CIRCUIT\_BREAKER\_FORCE\_OPEN:是否强制开启熔断，如果强制开启，永远不远程访问，默认false;

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_FORCE\_OPEN***,value="false"),

//CIRCUIT\_BREAKER\_FORCE\_CLOSED:是否强制关闭熔断，如果强制关闭，前置配置无效.即一定调用远程服务。默认为false

@HystrixProperty(name=HystrixPropertiesManager.***CIRCUIT\_BREAKER\_FORCE\_CLOSED***,value="false"),

//单位时间设置，默认10秒，单位毫秒；

@HystrixProperty(name=HystrixPropertiesManager.***EXECUTION\_ISOLATION\_THREAD\_TIMEOUT\_IN\_MILLISECONDS***,value="10000"),

})

**public** String hystrixFusing() {

***logger***.info("本地熔断方法执行。。。。。。。。。。。。。。。。。。。。。。。。。。。。");

String openFeign = testHystrixtApi.hystrixFusing();

//String openFeign = testHystrixtApi.hystrixTime();

**return** openFeign;

}

/\*\*

\* 熔断方法要求:

\* 1.返回值和对应的服务方法返回类型必须一致；

\* 2.方法的参数列表和对应服务方法一致；

\* **@return**

\*/

**public** String reqTimeout() {

**return** Thread.*currentThread*().getName()+"服务不可用--开启熔断";

}

### 3.在对应的启动类上增加新的注解@EnableCircuitBreaker

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

@EnableHystrix//开启 hystrix 熔断器，让代码中的hystrix 相关注解生效。

**public** **class** ShiroApp {

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

SpringApplication.*run*(ShiroApp.**class**, args);

}

## 请求缓存

Hystrix为了降低服务访问的频率，支持将一个请求与返回结果做缓存处理，如果再次请求的URL没变化，那么Hystrix不会请求服务，而是直接从缓存中将结果返回。这样可以大大降低访问服务的压力。是基于Spring cache实现的。在需要使用缓存服务方法删添加@cacheabel注解，在启动类使用@EnableCach主键开启缓存。需要借助redis支持。

### 代码实现

/\*\*

\* 请求缓存

\* 如果相同的请求，且缓存中有数据，当前方法不执行，直接返回缓存结果

\*如果需要加Hystrix容错逻辑，直接编写相关注解即可。如果增加了Hystrix容错逻辑，必须给请求缓存逻辑，增加例外，避免缓存拖地数据。

\*需要在配置类上使用@EnableCaching开启缓存，需要redis支持

\* **@param** id

\* **@return**

\*/

@ResponseBody

@RequestMapping("/reqCache")

@Cacheable(cacheNames = "hystrix:cache",key = "'reqCache('+#id+')'")

@HystrixCommand(fallbackMethod = "reqCacheBack")

**public** String reqCache(Long id) {

***logger***.info("本地方法执行。。。。。。。。。。。。。。。。。。。。。。。。。。。。");

String reqCache = **this**.testHystrixtApi.reqCache(id);

**return** reqCache;

}

**public** String reqCacheBack(Long id) {

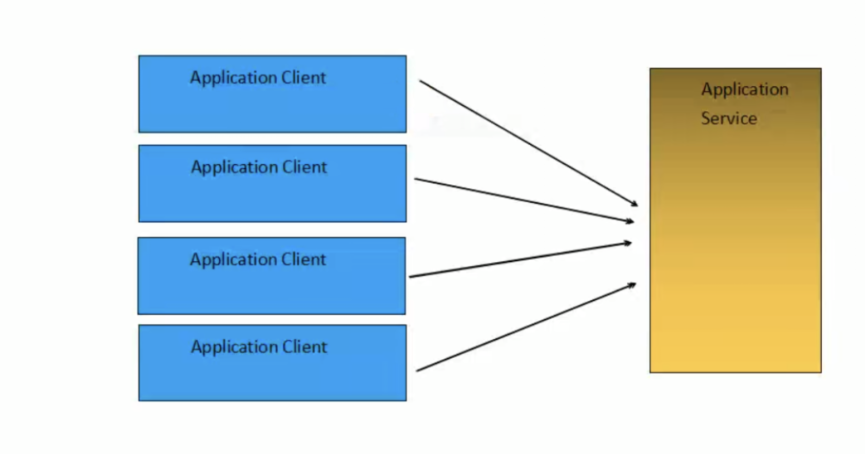
**return** Thread.*currentThread*().getName()+"缓存方法-降级";

}

## 请求合并

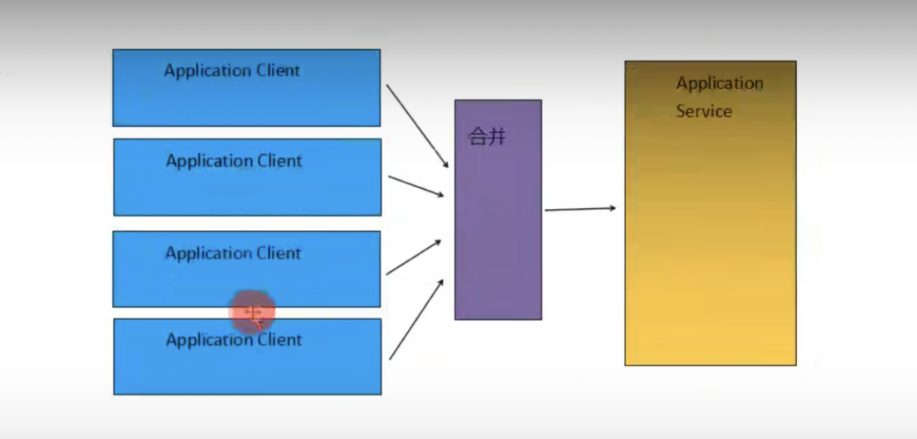
**没有请求合并**

Application Service 负载是Application Client发送请求的总数量。有多少次请求，它的负载就有多少个。



**请求合并**

请求合并是包一段时间范围内的所有请求合并为一个请求，大大降低了Application Service负载。



**什么情况下使用请求合井**

在微服务架构中，我们将一个项目拆分成很多个独立的项目，这些独立的项目通过远程调用来互相配合工作，但是，在高并发情况下，通信次数的增加会导致总的通信时间增加，同时，线程池的资源也是有限的，高并发环境会导致有大量的线程处于等待状态，进而导致响应延迟，为了解决这些问题，我们需要来了解 Hystrix 的请求合并。1个Tomacat并发量大约在600左右。

**请求合并的缺点**

设置请求合并之后，本来一个请求可能 5ms 就搞定了，但是现在心须再等 10ms 看看还

有没有其他的请求一起的，这样一个请求的耗时就从 5ms 增加到 15ms了，不过，如果我们要发起的命令本身就是一个高延迟的命令，那么这个时候就可以使用请求合并了，因为这个时候时间窗的时间消耗就显得微不足道了，另外高并发也是请求合并的一个非常重要的场景。

**请求合并参数介绍**

请求合并理论学习明白后，代码实现起来还是很容易的。关键就是一个@HystrixCollapser  
注解

|  |  |  |  |
| --- | --- | --- | --- |
| **请求合并参数** | | | |
| **参数** | **作用** | **默认值** | **备注** |
| @HystrixCollapser |  |  | 被@HystrixCollapser标注的方法，返回类型必须为Future，使用异步方法，否则无法进行请求合并. |
| batchmethod | 合并请求的方法 |  | 方法只能接受一个参数，如果需要传递多个参数，那么请将他们封装成一个类参数 |
| scop | 请求方式 | REQUEST | 请求方式分为:REQUEST, GLOBAL,  REQUEST范围只对一个request请求内的多次服务请求进行合并。  GLOBAL是将多单个应用中的所有线程的请求中的多次服务请求进行台并。 |
| timerDelayinMiliseconds | 请求时间间隔在10ms之内的请求会被合并为一个请求。 | 10ms |  |
| maxRequestsInBatch | 设置触发批量执行之前，在批处理中允许的最大请求数 | Interger.MAX\_VALUE | 建议尽量设置的小一点，如果并发量不大的话，其实也没有必要使用HystrixCollapser来处理 |

### 代码实现

高并发场景下，Application client接受到客户端(浏览器)发送的请求时，每次请求发送个Application server上，Application server的压力比较大，解决方案就是将请求合并为一个，Application server提供批处理逻辑，在Application client中提供拆分批处理结果。一旦请求合并，请求的参数就发生变化了，Application client参数是多个的，返回的结果也是多个的。Application server 批处理。

@ResponseBody

@RequestMapping("/selectOne")

**public** OrderDto selectOne(Long id) {

OrderDto orderDto = **null**;

**try** {

//问题将处理的方法放一个类合并失败

orderDto = hystrixReqMergeService.selectOrderById(id).get();

} **catch** (InterruptedException | ExecutionException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** orderDto;

}

/\*\*

\* 请求合并，远程服务的参数是集合，远程服务的返回值是集合.

\* HystrixCollapser注解代表当前方法是一个需要做请求合并处理的方法，有Hystrix创建代理实现。

\* 实现逻辑是：合并请求参数成位一个集合，并处理返回结果集，拆分后，封装成Future对象并返回指定的客户端。

\*

\* **@param** id

\* **@return** 多线程并发结果类型，代表预计结果可能是什么，泛型是具体返回类型

\*/

@HystrixCollapser(batchMethod = "selectOrderByIds",collapserProperties = {

//TIMER\_DELAY\_IN\_MILLISECONDS:合并时间窗，最多等待多久，一定请求远程服务。

//避免无限制等待合并，影响客户端性能。

@HystrixProperty(name = HystrixPropertiesManager.***TIMER\_DELAY\_IN\_MILLISECONDS***,value = "200"),

//MAX\_REQUESTS\_IN\_BATCH:一次合并，最多多少个请求，避免请求参数过多，影响处理效率。

@HystrixProperty(name = HystrixPropertiesManager.***MAX\_REQUESTS\_IN\_BATCH***,value="3")

},

//scop:合并请求的范围是什么，默认REQUEST。可选择：GLOBAL-全局范围；REQUEST-一次请求中

scope = com.netflix.hystrix.HystrixCollapser.Scope.***GLOBAL***

)

**public** Future<OrderDto> selectOrderById(Long id) {

**return** **null**;

}

//合并的批处理远程服务调用逻辑

@HystrixCommand//代表当前方法由Hystrix管理

**private** List<OrderDto> selectOrderByIds(List<Long> ids){

***logger***.info("本次请求参数位："+ids);

List<OrderDto> list = **this**.testHystrixtApi.selectOrderByIds(ids);

***logger***.info("远程访问结果："+list);

**return** list;

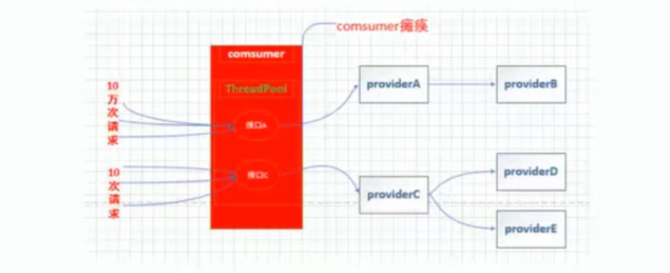
}

## 隔离

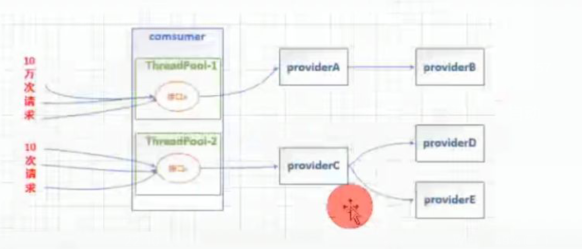
### 线程池隔离

**为什么使用线程池隔离**

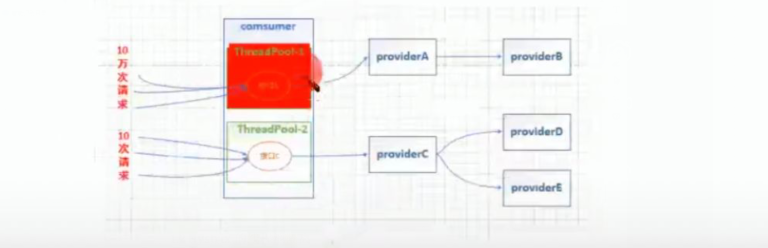
没有线程池隔离的时候，可能因为某个接口的高并发导致其他接口也出现问题。



当使用线程池隔离的时候，不同接口有自己独立的线程池。



即是某一个线程池被占用，也不影响其他线程。



**Hystrix的线程池隔离**

Hystrix采用bulkhead Partiton舱壁隔离技术。

舱壁隔离指的是将船体内部分为多个隔舱，一旦其中某几个隔舱发生破损进水，水流不会在其他的舱壁中流动，从而保证船舱依然具有足够的浮力和稳定性，降低沉船风险。

**线程池隔离的优缺点：**

1.任何一个服务都会被隔离在自己的线程池内，即使自己的线程池资源填满也不会影响其他服务。

2. 当依赖的服务重新恢复时，可通过清理线程池，瞬间恢复服务的调用。但是如

果是 tomcat线程池被填满，再恢复就会很麻烦。

3. 每个都是独立线程池。一定程度上解决了高并发问题。

4. 由于线程池中线程个数是有限制，所以也解决了限流（限制最多访问量）问题。

缺点：

1.增加了 CPU 开销。因为不仅仅有Tomcat 的线程池，还需要有Hystrix线程池。

2.每个操作都是独立的线程，就有排队、调度和上下文切换等问题。

#### 代码示例

/\*\*

\* 线程池隔离

\* groupKey:分组命名，默认和Threadpool的key相同

\* commandKey:命令,一般等于方法

\* threadPoolKey:独立线程池命名前缀

\* threadPoolProperties:线程池配置信息

\* **@return**

\*/

@ResponseBody

@RequestMapping("/isolateThreadPool")

@HystrixCommand(fallbackMethod = "isolateThreadPoolBack",groupKey = "test", commandKey = "isolateThreadPool",

threadPoolKey = "hystrix-test-pool",threadPoolProperties = {

//CORE\_SIZE：线程池容量

@HystrixProperty(name = HystrixPropertiesManager.***CORE\_SIZE***,value ="4"),

//MAX\_QUEUE\_SIZE:线程池中的线程消耗完毕后，提供的阻塞队列容量

@HystrixProperty(name = HystrixPropertiesManager.***MAX\_QUEUE\_SIZE***,value ="10"),

})

**public** String isolateThreadPool() {

String hystrixShow = **this**.testHystrixtApi.hystrixShow();

**return** Thread.*currentThread*().getName();

}

**public** String isolateThreadPoolBack() {

**return** "线程池隔离";

}

## 信号量隔离

java.util.concurrent.semaphore 用来控制可同时并发的线程数。通过构造方法指定内部虚

拟许可的数量。每次线程执行操作时先通过 acauire 方法获得许可 ，执行完毕再通过 release方法释放许可。如果无可用许可，那么acquire 方法将一直阻塞，直到其它线程释放许可。

如果采用信号量隔离技术，每接收一个请求，都是服务自身线程去直接调用依赖服务，

信号量就相当于一道关卡，每个线程通过关卡后，信号量数量减1，当为0时不再允许线程通过，而是直接执行 fallback 逻辑并返回，说白了仅仅做了一个限流。

#### 代码实现

//信号量隔离

@ResponseBody

@RequestMapping("/isolateMount")

@HystrixCommand(fallbackMethod = "isolateMountBack",groupKey = "test", commandKey = "isolateThreadPool",

threadPoolKey = "hystrix-test-pool",threadPoolProperties = {

//EXECUTION\_ISOLATION\_STRATEGY：线程隔离机制，默认是THREAD线程池隔离，可选SEMAPHORE信号量隔离

@HystrixProperty(name = HystrixPropertiesManager.***EXECUTION\_ISOLATION\_STRATEGY***,value ="SEMAPHORE"),

//EXECUTION\_ISOLATION\_SEMAPHORE\_MAX\_CONCURRENT\_REQUESTS:设置信号量阈值

@HystrixProperty(name = HystrixPropertiesManager.***EXECUTION\_ISOLATION\_SEMAPHORE\_MAX\_CONCURRENT\_REQUESTS***,value ="10"),

})

**public** String isolateMount() {

String hystrixShow = **this**.testHystrixtApi.hystrixShow();

**return** Thread.*currentThread*().getName();

}

**public** String isolateMountBack() {

**return** Thread.*currentThread*().getName();

}

# Springboot Actuator监控中心

# 客户端本地负载均衡

本地负载均衡本地从Eureka上获取对应的注册信息列表，获取到本地列表之后，让本地实现负载均衡，是在客户端实现的。

# Ribbon和Nginx实现负载均衡的区别

1. ribbon是本地负载均衡，在调用接口的时候，会在Eureka注册中心上获取注册信息列表，获取成功后，缓存在JVM本地。在本地使用rpc远程调用技术进行调用。即是在客户端实现负载均衡的。
2. Nginx是服务器端负载均衡，客户端发送的所有请求都会发个Nginx，让后让Ngnix实现转发请求。即是在服务端实行负载均衡的。
3. 本地负载均衡器适合在微服务rpc远程调用，如dubbo、springCloud。
4. Nginx服务器负载均衡适合于针对服务器端比如tomacat。

# 服务网关

为什么使用网关服务

<https://blog.51cto.com/u_15486212/5286296>

# SpringCloud Feign声明式客户端调用工具

SpringCloud中支持两种客户端调用工具：feign和rest；

Feign是一个声明式的Http客户端调用工具，才有接口+注解的方式实现，可读性强。

Feign客户端超时配置

Fegin客户对Ribbon的支持

1. 添加依赖

<dependency>

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

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

</dependency>

2.在服务消费者启动类中添加@EnableFeignClients注解

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

**public** **class** ShiroApp {

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

SpringApplication.*run*(ShiroApp.**class**, args);

}

}

1. 在接口上添加对应注解@FeignClient

@FeignClient(name="explore-member-service")

**public** **interface** MemberTestApi {

@RequestMapping("/hello")

String hello();

}

# RestTemplate调用服务接口

在SpringCloud中有两种调用方式：rest和fegin

RestTemplate是springboot web组件提供的，底层使用了httpClinent技术。默认整合了ribbon负载均衡器。

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients

**public** **class** ShiroApp {

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

SpringApplication.*run*(ShiroApp.**class**, args);

}

//1.将RestTemplate注册到Spring容器中

//2.使用rest方式别名方式调用依赖负载均衡器,@LoadBalanced能让RestTemplate在请求时拥有客户端负载均衡的作用

@Bean

@LoadBalanced

RestTemplate restTemplate() {

**return** **new** RestTemplate();

}

}

使用

**public** String testRest() {

//使用别名方式,使用别名方式取从注册中心上获取对应的服务调用地址。使用rest方式别名方式调用依赖ribbon负载均衡器

String forObject = restTemplate.getForObject("http://explore-member-service/member/hello", String.**class**);

**return** forObject;

}

# SpringCloud服务保护框架Hystrix

服务雪崩产生原因

Tomcat有个线程池，每个线程去处理客户端发送的请求。默认情况下tomacat只有一个线程池处理客户端发送的所有服务请求，在高并发的情况下,如果客户端的所有请求堆积到同一个服务接口上，就会产生tomcat的所有线程去处该理服务接口，可能导致服务接口无法访问。就会导致其他服务接口访问的时候，产生延迟和等待。可能导致整个微服务接口不能访问，所有的服务都会瘫痪。

如果tomcat最大请求数20，客户端同一时间发送了100个请求，会有80个请求产生延迟等待。如何证明tomacat默认情况下只有一个线程去处理所有客户端的请求，看线程的名称。Thread.currentTheread.getName()

Hystrix是服务保护框架，在微服务中Hystrix能帮我们解决断路器、服务器降级、服务器雪崩效应、服务熔断、服务隔离机制。

基于Hysrix解决服务器雪崩效应原理：服务降级、服务隔离、服务熔断。

服务降级：在高并发情况下（在tomcat中没有线程进行处理客户端请求的时候，不应该让用户一直等待。），防止用户一直等待，使用服务降级方式，返回一个友好的提示给客户端，不会去处理请求，调用fallback方法。目的是为了用户体验。

服务熔断：服务熔断是为了保护服务，在高并发的情况下，如果请求达到极限（阈值），如果流量超出阈值，自动开启服务保护功能，使用服务降级方式返回一个友好提示。服务熔断机制和服务降级一起使用。

服务隔离：

线程池隔离:每个服务都有自己独立的线程池，每个线程池互不影响。缺点是cpu占用率非常高。一般用于核心的接口。

信号量隔离：