# SpringCloud

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，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，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

1:微服务:

微服务架构是一种将单应用程序作为一套小型服务开发的方法，每种应用程序都在其自己的进程中运行，并与轻量级机制（通常是HTTP资源的API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制进行独立部署。这些服务的集中化管理已经是最少的，它们可以用不同的编程语言编写，并使用不同的数据存储技术。

2:微服务架构

3:微服务优点：

每个服务都高度内聚，颗粒度很小，代码容易理解，能聚焦在一个指定的业务功能和业务需求。开发简单，效率高，一个服务注重一个业务模块的开发，微服务适合小团队开发，微服务是松耦合，具有功能意义的服务，无论在开发阶段还是部署阶段都是独立的。微服务。可以使用不同的语言开发，易于和第三方集成。微服务只是业务逻辑代码，不会和前端技术混合。每个微服务都有自己的存储能力，可以是自己的数据库，也可以统一数据库

4：微服务缺点：

开发人员需要处理分布式系统的复杂性

多服务运维难度，服务数量的增加，运维压力随之增大

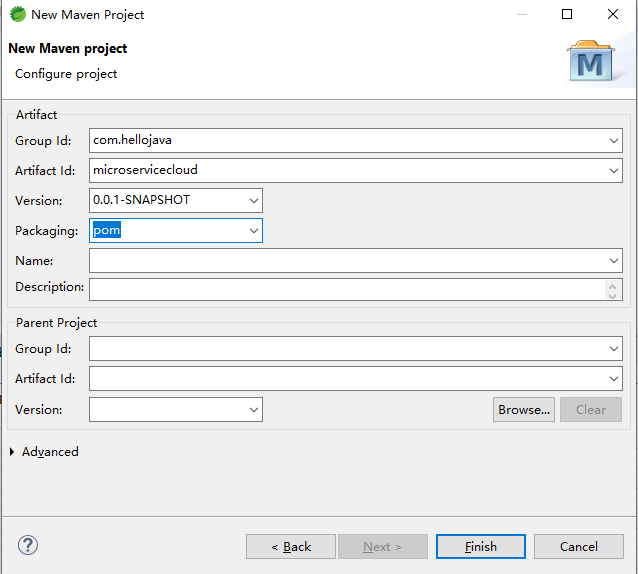
系统部署依赖

服务器通信成本

系统集成测试

微服务注册与发现

创建springcloud 父工程 (继承boot 以及 添加cloud和boot的依赖)



**修改父工程的pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<!-- 定义父工程的依赖管理 -->

<dependencyManagement>

<dependencies>

<!-- 加入spring cloud -->

<dependency>

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

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

<version>Finchley.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!-- 加入spring boot -->

<dependency>

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

<artifactId>spring-boot-dependencies</artifactId>

<version>2.1.2.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.40</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.mybatis.spring.boot/mybatis-spring-boot-starter -->

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

<version>1.3.2</version>

</dependency>

<!-- https://mvnrepository.com/artifact/junit/junit -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.4</version>

<scope>provided</scope>

</dependency>

</dependencies>

</dependencyManagement>

<modules>

<module>microservicecloud-api</module>

<module>microservicecloud-service</module>

</modules>

</project>

**创建maven Model microservicecloud-api**

**修改pom.xml**

<?xml version=*"1.0"*?>

<project xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"* xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-api</artifactId>

<name>microservicecloud-api</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

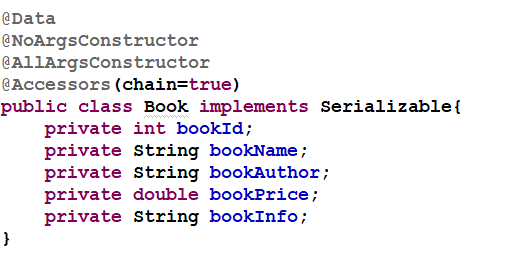
<artifactId>lombok</artifactId>

</dependency>

</dependencies>

</project>

**创建Book.java**



**创建Maven Model microservicecloud-service**

**修改pom.xml**

<?xml version=*"1.0"*?>

<project xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"* xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-service</artifactId>

<name>microservicecloud-service</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

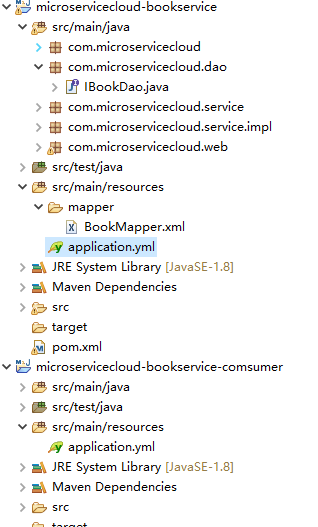
</dependencies>

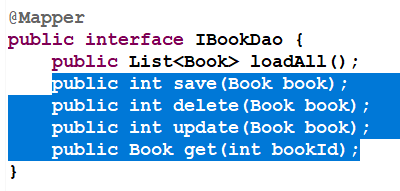
</project>

**在service工程中创建application.properties或者application.yml文件**

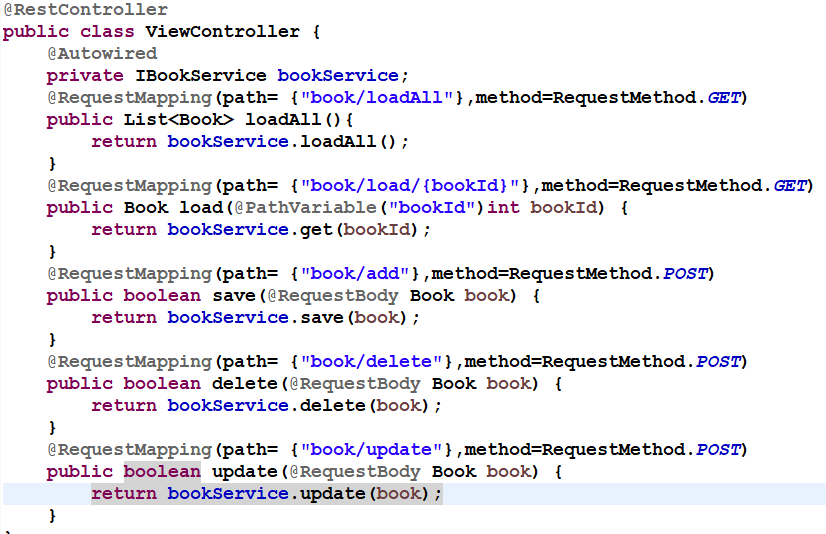


**工程结构图**

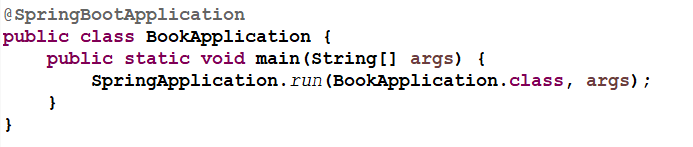




**添加服务提供者的controller**



添加服务器提供者的启动类



**新建工程 消费者(只添加spring web就可以 当前工程就是一个消费者调用 具体业务有bookservice工程完成)**

**Pom.xml**

<?xml version=*"1.0"*?>

<project xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"* xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-bookservice-comsumer</artifactId>

<name>microservicecloud-bookservice-comsumer</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>${project.groupId}</groupId>

<artifactId>microservicecloud-api</artifactId>

<version>${project.version}</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web -->

<dependency>

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

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

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools -->

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

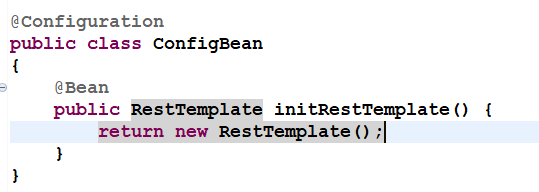
</project>

**创建application.yml**

server:

port: 80

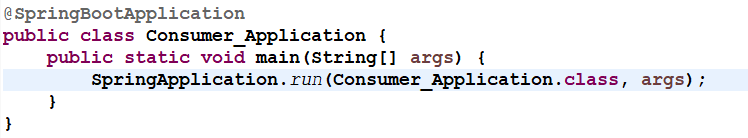
新建消费端的配置类:



**添加bookservice的访问类（consumer）**



**创建消费者的启动类**



**启动服务器提供者和服务消费者**

**测试访问**

**新建工程(物业公司（euraka server）当前工程只添加euraka依赖)**

**Pom.xml**

<?xml version=*"1.0"*?>

<project xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"* xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.hellojava</groupId>

<artifactId>microservicecloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<artifactId>microservicecloud-eurakaserver</artifactId>

<name>microservicecloud-eurakaserver</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Application.yml**

server:

port: 7001

eureka:

instance:

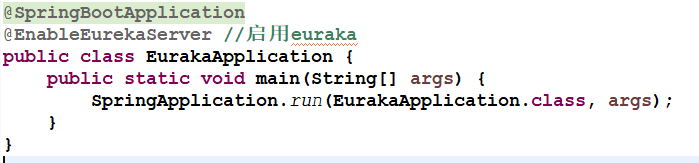
hostname: localhost #euraka服务的名字

client:

register-with-eureka: false #标识当前工程是一个euraka的服务器不需要注册

fetch-registry: false #标识当前工程不需要检测

创建euraka启动类



**启动测试(http://euraka.instace.hostname:server.port)**

[**http://localhost:7001**](http://localhost:7001) **出现euraka标识标识成功**

**将服务器提供者注册到euraka服务中**

**1:修改服务器提供者（bookservice工程） pom.xml中添加依赖包(euraka client)**

<dependency>

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

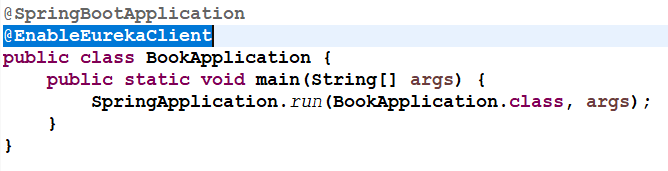
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

修改application.yml

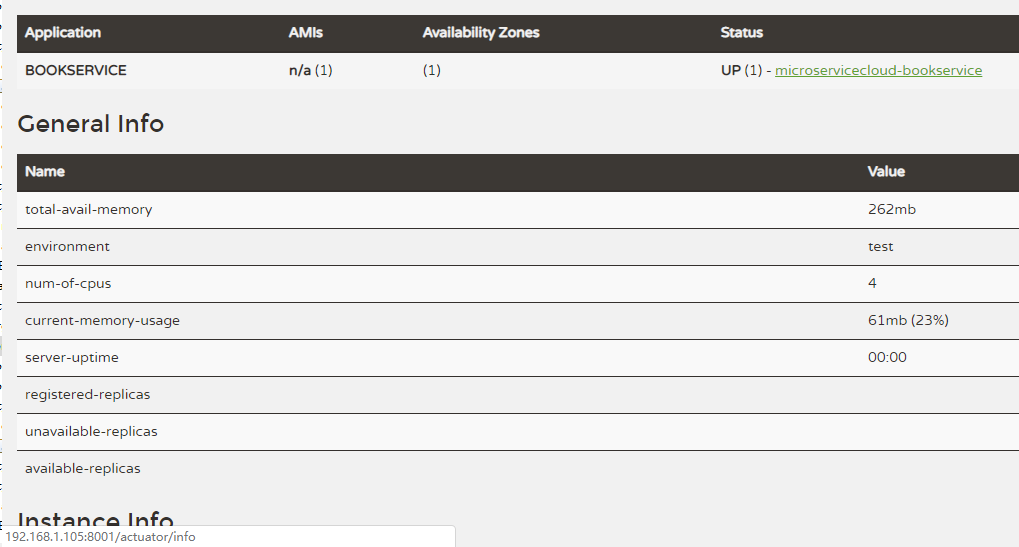


**在提供者的启动类上添加@EnableEurekaClient注解**



**启动服务器和服务提供者**

**Euraka info信息完善**



修改鼠标点击starus microservicecloud-bookservice 下面显示IP地址



Instance:instance-id: microservicecloud-bookservice 是status 的名称

Instance:prefer-ip-address:true 浏览器状态栏的IP显示

修改点击status 上的microservicecloud-bookservice 不出现错误页

再microservice-bookservice 提供者工程的pom.xml中添加spring boot actuator组件

<dependency>

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

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

</dependency>

然后再microservice-bookservice 提供者工程的application.yml文件中添加



Euraka 自我保护:

enable-self-preservation:false 关闭自我保护(不推荐使用)

euraka 集群

修改C:\Windows\System32\drivers\etc文件

添加

127.0.0.1 eureka7001.com

127.0.0.1 eureka7002.com

创建eureka1工程

Pom.xml 拷贝 eureka工程的pom.xml

Eureka工程

Application.yml文件



Eureka1工程

Application.yml文件



修改bookservice工程的application.yml



修改eureka.client.serviceUrl.defaultZone 把所有的eureka服务器全部添加上

**Ribbon(客户端 负载均衡):**

eureka：

服务提供者的主启动类(BookService)

@SpringBootApplication

@EnableEurekaClient(作用是向Eureka服务器注册自己)

服务器(Eureka)（主启动类）

@SpringBootApplication

@EnableEurekaServer

Ribbon:

在消费者工程的Configuration注解标识的类中返回RestTemplate的方法上添加@LoadBalanced注解

@Configuration

public class ConfigBean {

@Bean

@LoadBalanced //就会自动在发出请求的时候使用负载均衡算法

public RestTemplate initRestTemplate() {

return new RestTemplate();

}

}

在消费者工程的主启动类上添加@EnableEurekaClient(因为使用Ribbon 还是要使用Eureka 因为Ribbon先访问Eureka，从Eureka上获取能使用的服务地址，再去访问)

@SpringBootApplication

@EnableEurekaClient

public class Consumer\_Application {

public static void main(String[] args) {

SpringApplication.run(Consumer\_Application.class, args);

}

}

接着就可以使用Ribbon访问服务(以前在没有使用Ribbon的时候 消费端是使用RestTemplate方法访问但是使用的是http://localhost:8001/book/loadAll 访问 现在使用Ribbon了Ribbon可以从Eureka服务器上获取发布的服务，所以现在可以使用服务的名称访问，所以需要把原来的IP：端口换成服务名)

@RestController

public class ComsumerController {

@Autowired

private RestTemplate restTemplate;

@RequestMapping(value="/consumer/book/save")

public boolean save(Book book) {

return restTemplate.postForObject("http://microservicecloud-bookservice/book/add", book, Boolean.class);

}

@RequestMapping(value="/consumer/book/delete")

public boolean delete(Book book) {

return restTemplate.postForObject("http://microservicecloud-bookservice/book/delete", book, Boolean.class);

}

}

Ribbon支持7中负载均衡算法(默认是轮询)

@RibbonClient(name="服务名称",configuration=class) configuration指定产生负载均衡算法的类(这个类不能放置在当前工程主启动类的当前包以及子包下，必须新创建一个包) 这个类必须使用@Configuration注解，内部定义一个方法(方法必须使用@Bean注解，方法内部返回自定义的负载均衡算法的实例)

定义一个类继承AbstractLoadBalanceRule类 覆盖其中的choose方法(实现自定义的负载均衡算法(可以参照RandomRule算法))

feign(面向接口的服务调用 融合的Ribbon)

在服务器提供者的公共模块中(因为接口在调用者中也需要使用)添加接口(注意需要在公共模块的pom.xml添加feign的依赖)

接口和服务提供者的Controller中一样

@FeignClient(value="服务的名称(microservicecloud-bookservice)")

public interface BookServiceClient{

@RequestMapping(path= {"book/loadAll"},method=RequestMethod.GET)

public List<Book> loadAll();

@RequestMapping(path= {"book/load/{bookId}"},method=RequestMethod.GET)

public Book load(@PathVariable("bookId")int bookId) ;

@RequestMapping(path= {"book/add"},method=RequestMethod.POST)

public boolean save(@RequestBody Book book);

@RequestMapping(path= {"book/delete"},method=RequestMethod.POST)

public boolean delete(@RequestBody Book book) ;

@RequestMapping(path= {"book/update"},method=RequestMethod.POST)

public boolean update(@RequestBody Book book) ;

}

新建一个消费者工程(Feign)基本可以拷贝以前Ribbon的消费者工程(在POM.xml添加Feign依赖)

修改以前Ribbon工程的Controller类

@RestController

public class ComsumerController {

@Autowired

private BookServiceClient bookServlceClient;

@RequestMapping(value="/consumer/book/save")

public boolean save(Book book) {

return this.bookServiceClient.save(book);

}

@RequestMapping(value="/consumer/book/delete")

public boolean delete(Book book) {

return this.bookServiceClient.delete(book);

}

@RequestMapping(value="/consumer/book/update")

public boolean update(Book book) {

return this.bookServiceClient.update(book);

}

@RequestMapping(value="/consumer/book/load/{bookId}")

public Book load(@PathVariable("bookId")int bookId) {

return this.load(bookId);

}

@SuppressWarnings("unchecked")

@RequestMapping(value="/consumer/book/loadAll")

public List<Book> loadAll() {

return this.bookServiceClient.loadAll();

}

}

修改主启动类:

@SpringBootApplication

@EnableEurekaClient

@EnableFeignClients(basePackages={"当前工程的基础包名"})

public class Consumer\_Feign\_Application {

public static void main(String[] args) {

SpringApplication.run(Consumer\_Application.class, args);

}

}

hystrix(服务熔断:类似于家庭中的电路保险丝;在分布式系统中，当服务A调用服务B，服务B调用服务C，（这种叫扇出）如果服务C出现错误，那么hystrix就是保护服务之间的调用不会导致整个分布式系统的问题)

hystrix 是应对服务雪崩效应的一种微服务链路式保护，当删除链路的某个微服务不可用或者响应时间太长，会出现服务的降级，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回复调用链路。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态，当失败的调用到一定阈值，缺省是5秒内20次调用失败就会启动熔断机制，熔断机制的注解是@HystrixCommand

hystrix 服务出错(熔断)处理:

新建工程(继承springcloud)(参考provider(bookService))

添加pom.xml Hystrix maven坐标

修改eureka.instance.instance-id:在原来的microservicecloud-bookservice-hystrix(添加了hystrix)(可选操作)

在提供者的Controller方法中哪一个可能出错的方法上添加注解(@HystrixCommand)

@RequestMapping(path= {"book/load/{bookId}"},method=RequestMethod.GET)

@HystrixCommand(fallbackMethod="处理错误的方法名proccessHystrix\_Load")

public Book load(@PathVariable("bookId")int bookId) {

Book book=bookService.get(bookId);

if(book==null){

throw new RuntimeException("没有找到对应的记录")

}

return book;

}

public Book proccessHystrix\_Load(@PathVariable("bookId")int bookId){

return new Book().setBookId(bookId).setBookName("没有找到");

}

修改提供者的主启动类

@SpringBootApplication

@EnableEurekaClient

@EnableCircuitBreaker //启动熔断机制

public class BookApplication {

public static void main(String[] args) {

SpringApplication.run(BookApplication.class, args);

}

}

启动测试

hystrix 服务降级处理:

问题出现:

在提供者的Controller中所有RequestMapping标识的方法都需要去定义FallbackMethod 方法，那么Controller中的方法就会膨胀，所以需要使用AOP的思想把错误处理做解耦合(也就是说去掉提供者工程中Controller中@RequestMapping标识的方法上的@HystrixCommand注解，这样的话就不要给每个@RequestMapping注解的方法都添加熔断的方法)

1:我们在Controller类中RequestMapping标识的方法现在都使用Feign方式的接口调用,Feign 的接口定义在API工程中，所以在API工程中新增加一个类实现FallbackFactory接口

@Component //重要

public class BookClientServiceFallbackFactory implements FallbackFactory<BookServiceClient>{

@Override

public BookClientService create(Throwable throwable){

return new BookClientService(){

public Book load(long id){

return new Book().setBookId(bookId).setBookName("没有找到");

}

public List<Book> loadAll(){

}

public boolean save(@RequestBody Book book){}

public boolean delete(@RequestBody Book book) {}

public boolean update(@RequestBody Book book) {}

}

}

}

修改在API工程中定义的Feign 接口 BookClientService 上的@FeignClient注解

由

@FeignClient(value="服务的名称(microservicecloud-bookservice)")

public interface BookServiceClient{

@RequestMapping(path= {"book/loadAll"},method=RequestMethod.GET)

public List<Book> loadAll();

@RequestMapping(path= {"book/load/{bookId}"},method=RequestMethod.GET)

public Book load(@PathVariable("bookId")int bookId) ;

@RequestMapping(path= {"book/add"},method=RequestMethod.POST)

public boolean save(@RequestBody Book book);

@RequestMapping(path= {"book/delete"},method=RequestMethod.POST)

public boolean delete(@RequestBody Book book) ;

@RequestMapping(path= {"book/update"},method=RequestMethod.POST)

public boolean update(@RequestBody Book book) ;

}

改成

@FeignClient(value="MICROSERVICECLOUD-BOOKSERVICE",fallbackFactory=上面新增的实现FallbackFactory接口的类)

public interface BookServiceClient{

@RequestMapping(path= {"book/loadAll"},method=RequestMethod.GET)

public List<Book> loadAll();

@RequestMapping(path= {"book/load/{bookId}"},method=RequestMethod.GET)

public Book load(@PathVariable("bookId")int bookId) ;

@RequestMapping(path= {"book/add"},method=RequestMethod.POST)

public boolean save(@RequestBody Book book);

@RequestMapping(path= {"book/delete"},method=RequestMethod.POST)

public boolean delete(@RequestBody Book book) ;

@RequestMapping(path= {"book/update"},method=RequestMethod.POST)

public boolean update(@RequestBody Book book) ;

}

定义如果BookServiceClient接口中定义的方法出现问题，那么就去执行@FeignClient注解中fallbackFactory指定的类中相应的方法处理

最后再使用Feign组件的消费者工程中修改application.yml

添加:

feign:

hystrix:

enable: true

#启用hystrix

启动所有eureka server

启动原始的bookService 提供者工程（Hystrix 服务降级是再客户端工作了现在，所有不需要启动microservicecloud-bookservice-hystrix了）

启动feign消费者工程

测试正常的

然后关闭 原始的bookService 提供者工程

访问 这个时候熔断就起作用了

除了隔离依赖服务的调用，hystrix该提供了准实时的调用监控.

netflix 使用hystrix-metrics-event-stream 实现监控

新建工程microservicecloud-comsumer-hystrix-dashboard

pom.xml添加 spring-cloud-starter-hystrix 以及 spring-cloud-starter-hystrix-dashboard

application.yml

server:

port: 9001

主启动类

@PringBootApplication

@EnableHystrixDashboard

public class BookConsumer\_Dashboard\_Application{

public static void main(String[] args){

SpringApplication.run(BookConsumer\_Dashboard\_Application.class,args);

}

}

再被监控的微服务的提供者工程中添加监控依赖(也就是bookService工程)

<dependency>

<groupId>prg.springframework.boot</groupId>

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

</dependency>

先启动hystrix-dashboard工程

http://localhost:9001/hystrix 测试监控是否成功

测试

启动eureka server

启动provider-hystrix (带有hystrix的提供者)

启动consumer-hystrix-dahaboard(监控消费者)

http://localhost:8001/hystrix.stream 会不断的发出请求测试

Zuul：路由网关:

包含了堆请求的路由和过滤两个最主要的功能：

其中路由功能负责讲外部请求转发到具体的微服务实例上，是实现外部访问同一入口的基础而过滤器功能则负责堆请求的处理过程进行高于，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滋生注册为Eureka服务治理下的应用，同事Eureka中获得其他微服务的消息，也即以后的访问微服务都是通过Zuul跳转后获得。

注意：Zuul服务最终还是会注册进Eureka

Zuul提供：代理+路由+过滤三大功能

新建工程microservicecloud-zuul-gateway-9527

pom.xml

添加

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

application.yml

server:

port: 9527

spring:

application:

name: microservicecloud-zuul-gateway #注册到eureka服务中心后显示的application name

eureka:

client:

service-url:

defaultZone: http://eureka7001.com:7001/eureka ,http://eureka7002.com:7002/eureka ,http://eureka7003.com:7003/eureka

instance:

instance-id: gateway-9527.com

prefer-ip-address: true

info:

app.name: hellojava-microservice

company.name: www.hellojava.com

修改host文件

添加：127.0.0.1 myzuul.com

主启动类:

@SpringBootApplication

@EnableZuulProxy

public class Zuul\_9527\_App{

public static void main(String[] args){

SpringApplication.run(Zuul\_9527\_App.class,args);

}

}

启动eureka server

启动服务提供者 8001(原始的bookservice)

启动zuul 路由工程

测试：

http://前面host配置的服务器名称myzuul.com:9527/microservicecloud-bookservice(真是访问的服务提供者名称)/book/load/2

但是前面暴露了真是微服务的名称

zuul 路由访问映射

修改microservicecloud-zuul-gateway-9527 工程的application.yml文件

zuul:

routes:

mybook.serviceId: mycroservicecloud-bookservice

mybook.path: /mybook/\*\*

修改完以后：

http://myzuul.com:9527/mybook/book/load/2 访问

但是这个时候 真实的还是可以访问

再次修改application.yml

zuul:

ignored-services: mycroservicecloud-bookservice # 隐藏真是的地址访问

routes:

mybook.serviceId: mycroservicecloud-bookservice

mybook.path: /mybook/\*\*

如果隐藏所有当前Eureka Server上的所有的微服务的真实名称

zuul:

ignored-services: "\*"

routes:

mybook.serviceId: mycroservicecloud-bookservice

mybook.path: /mybook/\*\*

设置同意的公共的前缀

zuul:

prefix: /hellojava #设置所有的前缀

ignored-services: "\*"

routes:

mybook.serviceId: mycroservicecloud-bookservice

mybook.path: /mybook/\*\*

这个时候访问必须添加前缀

http://myzuul.com:9527/hellojava/mybook/book/load/2