Maven中dependencyManagement与dependencies

Maven使用dependencyManagement元素来提供了一种管理依赖版本号的方式,通常会在一个组织或者项目的最顶层的父pom中看到dependencyManagement元素

使用pom.xml中的dependencyManagement元素能够让所有在子项目中引用一个依赖而不用显式的列出版本号.Maven会沿着父子层次向上走,直到找到一个拥有dependencyManagement元素的项目,然后它就会使用这个**dependencyManagement**元素指定的版本号.

这样做的好处是:如果有多个子项目都引用同一样依赖,则可以避免在每个使用的子项目里都声明一个版本号,这样当想升级或切换到另一个版本时,只需要在**顶层父pom里更新**,而不需要一个一个子项目的修改;另外如果某个子项目需要另外的一个版本,只需要声明version就可.

dependencyManagement里只是**声明依赖**,**并不实际引入**,因此子项目需要显式的声明需要用的依赖

如果在子项目中指定了版本号,那么会使用子项目中指定的jar版本

热部署

```
1.添加依赖
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
2.在父工程中添加插件
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <fork>true</fork>
                <addResources>true</addResources>
            </configuration>
        </plugin>
    </plugins>
</build>
3.设置idea
```

lombok支持链式写法

```
pojo中lombok支持链式写法
@Accessors(chain = true) //链式写法
```

Spring Cloud 生态 一站式解决方案

```
1.Spring cloud NetFlix 一站式解决方案
    api网关, zuul组件
    Feign  --HttpClient --Http通信方式,同步,阻塞
    服务注册发现:Eureka
    熔断机制:Hystrix

2.Apache Dubbo Zookeeper 半自动,需要整合别人的
    Zookeeper

3.Spring Cloud Alibaba 一站式解决方案,更简单
```

springboot与springcloud关系

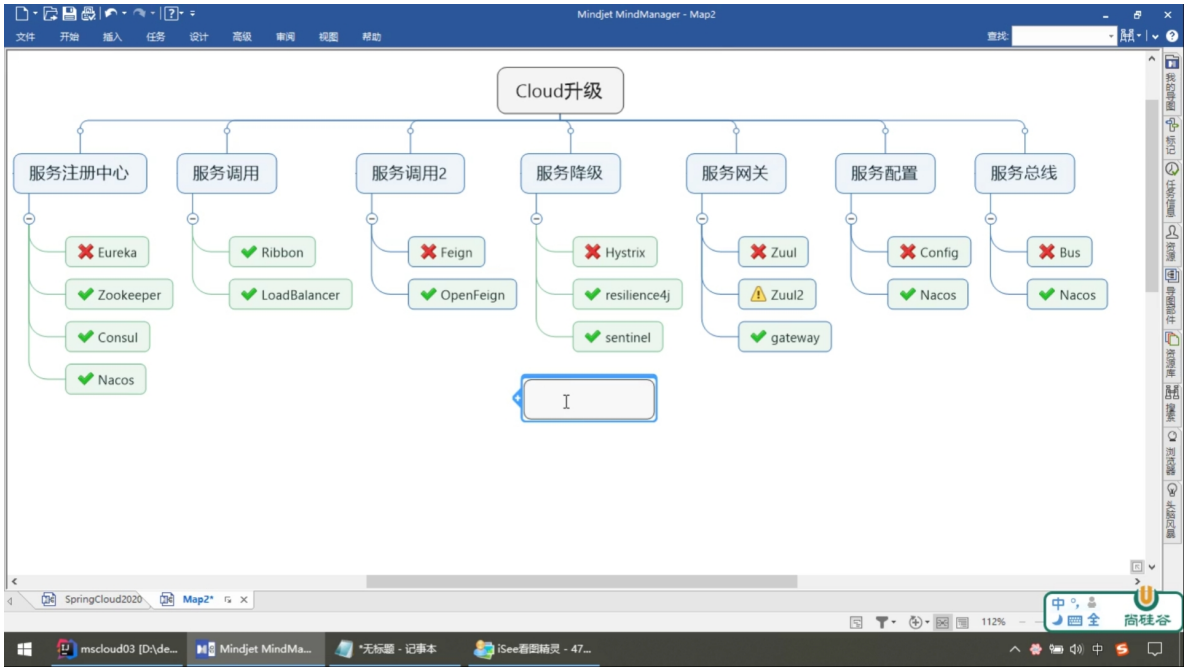
```
SpringBoot与Spring cloud关系

- Spring boot专注于快速方便的开发单个个体的微服务. -Jar
- Spring cloud是专注关注全局的微服务协调治理框架,它将Spring boot开发的一个一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,策略竞选,分布式会话等集成服务.
- Spring boot可以离开spring cloud独立使用,开发项目,但是Spring cloud不能离开spring boot,属于依赖关系
- spring boot专注于快速,方便的开发单个个体微服务,spring cloud专注于全局的服务治理框架.
```

restTemplate

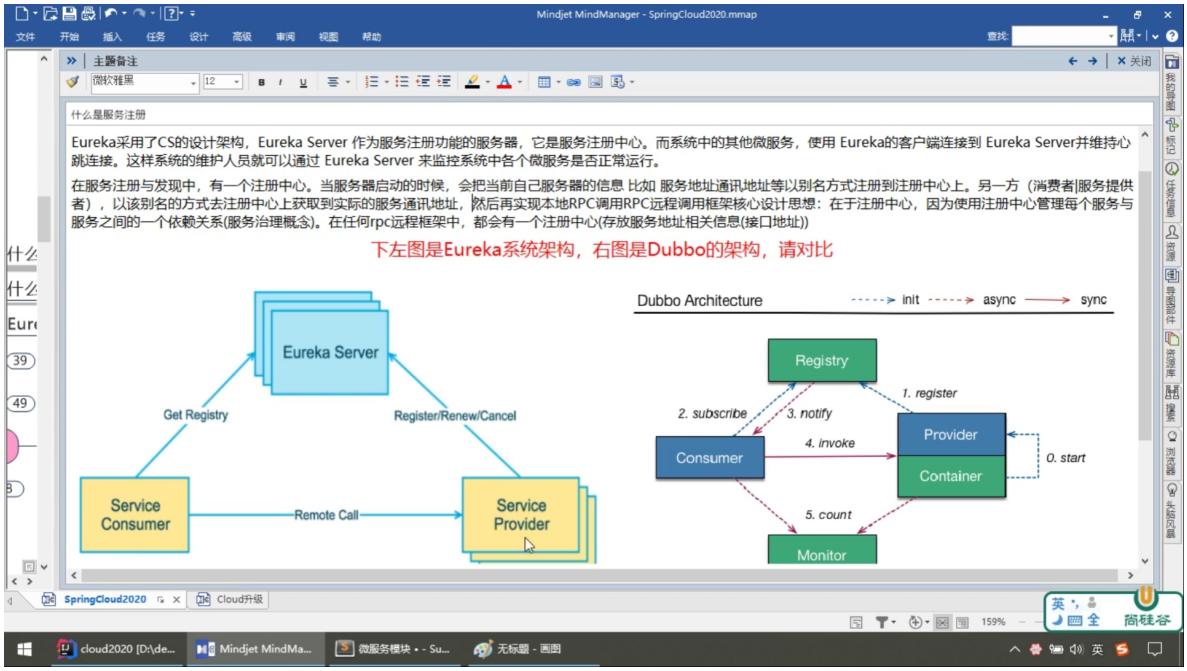
RestTemplate提供了多种便捷访问远程http服务的方法,是一种简单便捷的访问restful服务的模板类,是spring提供的用于访问Rest服务的**客户端模板工具类**.

```
restTemplate.getForObject //返回对象为响应体中数据转化成的对象,基本可以理解为Json
restTemplate.getForEntity //返回对象为ResponseEntity对象,包含了响应中的一些重要信息,比如响应头,响应状态码,响应体等.
```



Eureka服务注册与发现

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与故障转移,服务注册与发现对于微服务来说是非常重要的,有了服务注册与发现,只需要使用服务的**标识符**就可以访问到服务,而不需要改配置文件了,类似与Dubbo的注册中心,比如Zookeeper



Eureka的基本架构

Spring cloud封装了NetFlix公司开发的Eureka模块来实现服务注册和发现.

Eureka采用了C-S的架构设计,EurekaServer作为服务注册功能的服务器,它是注册中心.

Eureka Client是一个java客户端,用于简化Eureka Server的交互,客户端同时也具备一个内置的,使用轮循负载算法的负载均衡器.在应用启动后,将会向EurekaService发送心跳.

单机eureka实现

创建EurekaServer模块

```
1.pom依赖
<!--eureka-server-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>

2.启动类上添加注解  @EnableEurekaServer
3.编写配置
eureka:
  instance:
    hostname: localhost  #eureka服务端的实列名称
  client:
    register-with-eureka: false  #false表示不向注册中心注册自己
    fetch-registry: false  #表示自己就是注册中心
    service-url:
      defaultZone:
http://${eureka.instance.hostname}:${server.port}/eureka/
```

将服务注册进EurekaServer

```
1.pom依赖
<!--eureka client-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
2.启动类上添加注解  @EnableEurekaClient
3.编写配置
eureka:
  client:
    register-with-eureka: true  #表示是否将自己注册进EurekaServer 默认为true
    #是否从EurekaServer抓取已有的注册信息,默认为true. 单节点无所谓,集群必须设置为true
才能配合ribbon使用负载均衡
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

消费者从注册中心取服务

```
1.pom依赖
<!--eureka client-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
2.启动类上添加注解  @EnableEurekaClient
3.单机用restTemplate就可以得到服务,一般是集群模式,见下面集群模式的配置

```

eurekaServer集群

```
原理:相互守望,相互连接
在eurekaServer端口7001配置文件中,添加其他的eureka模块路径 用 , 分割开
service-url:
      defaultZone:
http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
同理在其他的eureka配置文件中也需配置.

```

```
将eurekaClient(即服务提供者provider)模块和消费者模块向eurekaServer集群中注册
service-url:
      defaultZone:
http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eu
reka7003.com:7003/eureka
```

eurekaClient集群(即服务提供者provider)

```
每一个eurekaClient都将eurekaServer集群的地址配置进去
service-url:
      defaultZone:
http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka,http://eu
reka7003.com:7003/eureka
```

服务消费者操作

```
1.配置类中
@Configuration
public class ApplicationContextConfig {
    @Bean
    @LoadBalanced //使用@LoadBalanced注解赋予RestTemplate负载均衡的能力
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}


@RestController
@Slf4j
@RequestMapping("/consumer")
public class OrderController {
    @Autowired
    private RestTemplate restTemplate;

    //CLOUD-PAYMENT-SERVICE 即为服务提供者在eureka中注册的名称
    public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-
SERVICE";

    @GetMapping("/payment/create")
    public CommonResult<Payment> create(Payment payment){
        return
restTemplate.postForObject(PAYMENT_URL+"/payment/create",payment,CommonResu
lt.class);
    }

    @GetMapping("/payment/get/{id}")
    public CommonResult<Payment> getPayment(@PathVariable("id")Long id){
```

```
        return restTemplate.getForObject(PAYMENT_URL+"/payment/get/"+id,CommonResult.class);
    }
}
```

服务发现

```
在eurekaClient主启动类上标注 @EnableDiscoveryClient
在controller中得到服务的信息
    注入客户端
    @Autowired
    private DiscoveryClient discoveryClient;
    得到信息
    @GetMapping("/discovery")
    public CommonResult discovery(){
        List<String> services = discoveryClient.getServices();
        return new CommonResult(200,"信息",services);
    }
```

eureka自我保护机制

某时刻一个微服务不可以用了,eureka不会立刻清理,依旧会对该微服务的信息进行保存.

比如当断电的时候,微服务实列就断了,但是这个微服务本身是健康的,此时不应该注销这个服务.这是eureka应对网络异常的安全保护机制.

禁止自我保护机制

```
#关闭自我保护机制
eureka:
  server:
    enable-self-preservation: false
```

CAP原则

```
一致性(Consistency)

可用性(Availability)

分区容错性(Partition tolerance)
```

指在一个分布式系统中,一致性(Consistency),可用性(Availability),分区容错性(Partition tolerance).这三个要素最多只能同时实现两点,不可能三者兼顾.分布式必须满足P.

一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。

可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端

分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

Eureka与zookeeper对比

zookeeper保证的是CP

当向注册中心查询服务列表时,我们可以容忍注册中心返回的是几分钟前的数据,但不能接收服务直接down掉不可用,也就是说,服务注册功能对可用性的要求要高于一致性.但是zk会出现一种情况,当master节点因为网络故障与其他节点失去联系时,剩余节点会重新进行leader选举,问题在于选举时间太长,30~120s,且选举期间整个zk集群都是不可用的,这就导致在选举期间注册服务瘫痪.

Eureka保证的是AP

Eureka各个节点都是平等的,几个节点挂掉不会影响正常节点的工作,当发现连接失败时,会自动切换到其他节点,只要有一台eureka还在,就能保证注册服务的可用性,只不过查到的信息可能不是最新的.除此之外,eureka还有一种自我保护机制,如果在15分钟内超过85%的节点都没有正常心跳,那么eureka就认为客户端与注册中心出现了网络故障,会出现一下几种情况:

1.eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务

2.eureka仍然能够接收新服务的注册和查询请求,但是不会同步到其他节点(即保证当前节点可用)

3.当网络稳定时,当前实列新的注册信息会被同步到其他节点中.

因此,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,而不是像zk那样使整个注册服务瘫痪.

Consul保证的是CP

Consul

Consul是一套开源的分布式服务发现与配置管理系统,由HashiCorp公司用**Go语言开发**

提供了微服务系统中的服务治理,配置中心,控制总线等功能.这些功能中的每一个都可以根据需要单独的使用,也可以一起使用以构建全方位的服务网络,总之Consul提供了一种完整的服务网格解决方案.

启动consul

cmd下 `consul agent -dev`

浏览器输入`localhost:8500`即可看到consul控制台

服务提供者注册进Consul

```
1.pom依赖
<!--SpringCloud consul-server -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
2.配置文件
spring:
  application:
    name: cloud-provider-consul-payment
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
这样服务就注册进consul了
```

消费者注册进consul

```
1.pom依赖 同上
2.配置文件  基本同上
3.注入TestTemplate
@Bean
@LoadBalanced //使用@LoadBalanced注解赋予RestTemplate负载均衡的能力
public RestTemplate restTemplate(){
return new RestTemplate();
}
4.调用 通过服务名称调用
@Autowired
private RestTemplate restTemplate;

public static final String INVOKE_URL = "http://cloud-provider-consul-payment";

@GetMapping(value = "/consumer/payment/consul")
public String paymentInfo()
{
String result = restTemplate.getForObject(INVOKE_URL+"/payment/consul",String.class);
return result;
}
```

Ribbon

SPring cloud Ribbon是基于Netflix Ribbon实现的一套**客户端负载均衡的工具**

简单的说,Ribbon是Netflix发布的开源项目,主要功能是提供客户端的软件负载均衡算法,将netflix的中间层服务连接在一起.Ribbon的客户端组件提供一系列完整的配置项如:连接超时,重试等..就是在配置文件中列出**Load Balance(简称LB:负载均衡)**后面的所有机器,Ribbon会自动的帮助你基于某种规则(如简单轮循,随机连接等)去连接这些机器,我们也很容易使用Ribbon实现**自定义的负载均衡算法.**

- 集中式LB

  即在服务的消费方和提供方之间使用独立的LB设施,如Nginx,由该设施负责把访问请求通过某种策略转发至服务的提供方.

- 进程式LB

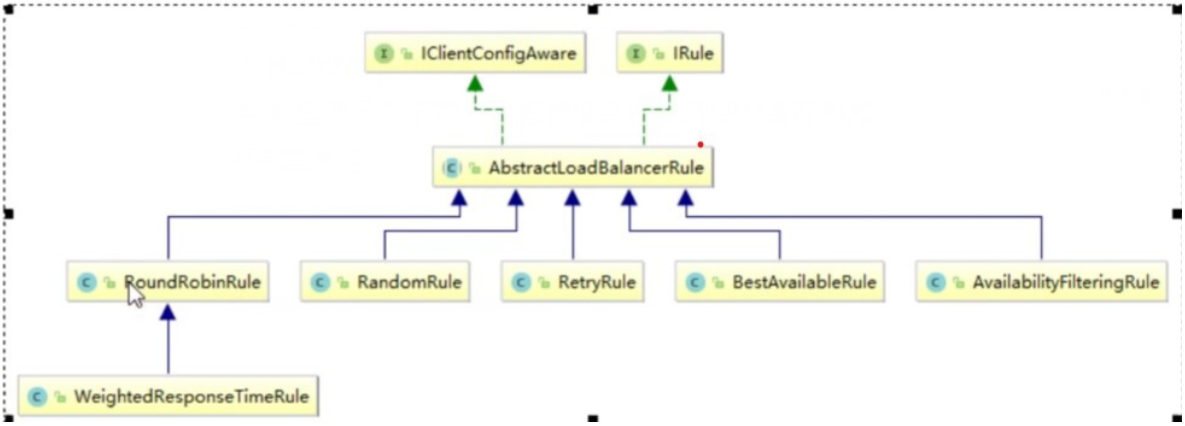
  将LB逻辑集成到**消费方**,消费方从服务注册中心获知有哪些地址可用,然后自己再从这些地址中选出一个合适的服务器

  Ribbon就属于进程式的LB,它只是一个类库,**集成于消费方进程**,消费方通过它来获取到服务方的地址.

```
1.pom  netflix-eureka-client中包含了ribbon,不需要额外添加依赖
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

IRule:根据特定算法从服务列表中选取一个要访问的服务

Spring cloud官方明确说明 存放IRule的类不能与主启动类同级或者在其下一级的包中



使用其他的轮循算法

```
1.重新规定轮循规则
@Configuration
public class MySelfRule {
    @Bean
    public IRule myRule(){
        return new RandomRule();
    }
}
2.在主启动类上说明
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE",configuration =
MySelfRule.class)
```

负载均衡算法: rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标 , 每次服务重启后rest接口计数从1开始.

OpenFeign

feign

feign是声明式的web service**客户端**,它让微服务之间的调用变得更简单了,类似controller调用service.Spring Cloud集成了ribbon和eureka可在使用feign时提供负载均衡的http客户端

只需要创建一个接口,然后添加注解即可.

feign,主要是社区,大家都习惯面向接口编程,这个是很多开发人员的规范.调用微服务访问两种方法

1. 微服务名字[ribbon]
2. 接口和注解[feign]

Feign旨在使编写java http客户端变得更容易前面在使用Ribbon+Rest Template时,利用RestTemplate对http请求的封装处理,形成了一套模板化的调用方法,但是在实际开发中,由于对服务依赖的调用可能不止一处,往往一个接口会被多处调用,所以通常都会针对每个微服务自行封装一些客户端来包装这些依赖服务的调用.所以,Feign在此基础上做了进一步的封装,由他来帮助我们定义和实现依赖服务接口的定义,在Feign的实现下,我们只需要创建一个接口并使用注解的方式来配置它(类似于以前Dao接口上标注Mapper注解,现在是一个微服务接口上标注一个Feign注解即可.)

Feign集成了Ribbon,天生具备负载均衡

OpenFeign

```
1.pom依赖
<!--openfeign-->
<dependency>
```

```
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
2.主启动类上注解  @EnableFeignClients
3.写接口
@Component
@FeignClient(value = "CLOUD-PAYMENT-SERVICE")  //提供的服务名
public interface PaymentFeignService {
    @GetMapping("/payment/get/{id}")  //提供服务的地址
    CommonResult getPaymentById(@PathVariable("id")Long id);
}
4.controller调用
@Autowired
private PaymentFeignService paymentFeignService;

@GetMapping("/consumer/payment/get/{id}")
public CommonResult<Payment> getPaymentById(@PathVariable("id")Long id){
    CommonResult payment = paymentFeignService.getPaymentById(id);
    return payment;
}

```

OpenFeign超时控制

默认客户端只等待1s,但是服务端处理需要超过1s,导致客户端不想等待了,直接返回报错.为了避免这样的情况,有时候我们需要设置Feign客户端的超时控制

```
#设置feign客户端超时时间(OpenFeign默认支持ribbon)
ribbon:
  #指的是建立连接所用的时间，适用于网络状况正常的情况下,两端连接所用的时间
  ConnectTimeout: 5000
  #指的是建立连接后从服务器读取到可用资源所用的时间
  ReadTimeout: 5000
```

OpenFeign日志打印

Feign提供了日志打印功能,我们可以通过配置来调整日志级别,从而了解Feign中http请求的细节,说白了就是**对Feign接口的调用情况进行监控和输出**

```
日志级别
NONE:默认的,不显示任何日志
BASIC:仅记录请求方法,URL,响应状态码及执行时间
HEADERS:除了BASIC中定义的信息外,还有请求和响应的头信息
FULL:除了HEADERS中的信息,还有请求和响应的正文及元数据
```

```
1.写一个配置类
@Configuration
public class FeignConfig {
    @Bean
    Logger.Level feignLoggerLevel(){
        return Logger.Level.FULL;
    }
}
2.配置文件中配置
logging:
  level:
    # feign日志监控哪个接口以什么方式输出
    com.atguigu.springcloud.service.PaymentFeignService: debug
```

Hystrix

服务雪崩

多个微服务之间调用的时候,假设微服务A调用微服务B和微服务C,微服务B和微服务C又调用其他的微服务,这就是所谓的"**扇出**",如果扇出的链路上某个微服务的响应时间过长或者不可用,对微服务A的调用就会占用越来越多的系统资源,进而引起系统崩溃,所谓的"雪崩效应"

对于高流量的应用来说,单一的后端依赖可能导致所以的服务器上的所有资源都在几秒内饱和.比失败更糟糕的是,这些应用程序还可能导致服务之间的延迟增加,备份队列,线程和其他系统资源紧张,导致整个系统发生更多的级联故障,这些都表示需要对故障和延迟进行隔离和管理,以便单个依赖关系的失败,不能取消整个应用程序或系统.

"断路器"本身是一种开关装置,当某个服务单元发生故障之后,通过断路器的故障监控(类似熔断保险丝),**向调用方返回一个服务预期的,可处理的备选响应(fallback),而不是长时间的等待或者抛出调用方法无法处理的异常,这样就可以保证了服务调用方的线程不会长时间不必要的占用,**从而避免了故障在分布式系统中的蔓延,乃至雪崩

Hystrix是一个用于处理分布式系统的**延迟**和**容错**的开源库,在分布式系统中,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.

功能

- 服务降级
- 服务熔断
- 服务限流
- 接近实时的监控
- ...

超时导致服务器变慢,不能一直等待

服务宕机了,调用者不能一直卡死等待

服务降级

向调用方返回一个服务预期的,可处理的备选响应(fallback),而不是长时间的等待或者抛出调用方法无法处理的异常,这样就可以保证了服务调用方的线程不会长时间不必要的占用,

哪些情况下出现服务降级: 程序异常运行,超时,服务熔断触发服务降级,线程池/信号量打满也会导致服务降级

服务端的服务降级

```
在主启动类上添加 @EnableCircuitBreaker
一旦调用方法失败并抛出错误信息后,会自动调用 @HystrixCommand 标注好的fallbackMethod指定的方法
@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler",commandProperties = {
   //表示这个线程运行运行3s 否则就调用兜底的方法
@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="3000")
    })
public String paymentInfo_timeout(Integer id){
    int timeNum = 5;
    //int a = 1/0;
    try {
        TimeUnit.SECONDS.sleep(timeNum);
    }catch (Exception e){
        e.printStackTrace();
    }
    return "线程池: "+Thread.currentThread().getName()+" paymentInfo_timeout "+id + "O(∩_∩)O"+" 耗时(s):"+timeNum;
}
//这是一个兜底方法
public String paymentInfo_TimeOutHandler(Integer id)
{
    return "线程池:  "+Thread.currentThread().getName()+"  8001系统繁忙或者运行报错，请稍后再试,id:  "+id+"\t"+"o(╥﹏╥)o";
}
```

客户端的服务降级

```
配置文件中
feign:
  hystrix:
    enabled: true
主启动类上开启 @EnableHystrix
```

```
@GetMapping("/consumer/payment/hystrix/timeout/{id}")
    @HystrixCommand(fallbackMethod = "paymentTimeOutFallbackMethod",commandProperties = {

 @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="1500")
    })
public String paymentInfo_TimeOut(@PathVariable("id") Integer id)
{
    int age = 10/0;
    String result = paymentHystrixService.paymentInfo_TimeOut(id);
    return result;
}
public String paymentTimeOutFallbackMethod(@PathVariable("id") Integer id)
{
    return "我是消费者80,对方支付系统繁忙请10秒钟后再试或者自己运行出错请检查自己,o(╥﹏╥)o";
}

// 下面是全局fallback方法  在类上标注 @DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
```

```
public String payment_Global_FallbackMethod() {
    return "Global异常处理信息，请稍后再试，/(ㄒoㄒ)/~~";
}
```

服务降级,客户端去调用服务端,碰上服务端宕机或关闭

本实列服务降级处理是在**客户端80**实现完成的,与服务端8001没有关系

只需要为Feign客户端定义的接口添加一个服务降级处理的实现类即可实现解耦

```
@Component
public class PaymentFallbackService implements PaymentHystrixService
{
    @Override
    public String paymentInfo_OK(Integer id)
    {
        return "-----PaymentFallbackService fall back-paymentInfo_OK ,o(╥﹏╥)o";
    }

    @Override
    public String paymentInfo_TimeOut(Integer id)
    {
        return "-----PaymentFallbackService fall back-paymentInfo_TimeOut ,o(╥﹏╥)o";
    }
}

//在接口上标注 指明异常时回调的处理类
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT",fallback = PaymentFallbackService.class)
```

服务熔断

break. 直接拒绝访问,然后调用服务降级的方法并返回友好提示.

当检测到该节点微服务调用响应正常后,恢复调用链路

```
//服务端service
//=======服务熔断===============
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback",commandProperties = {
            @HystrixProperty(name = "circuitBreaker.enabled",value = "true"),// 是否开启断路器
            @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value = "10"),// 请求次数
            @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value = "10000"), // 时间窗口期
            @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value = "60"),// 失败率达到多少后跳闸
    })
public String paymentCircuitBreaker(@PathVariable("id") Integer id)
{
    if(id < 0)
    {
        throw new RuntimeException("******id 不能负数");
```

```
    }
    String serialNumber = IdUtil.simpleUUID();

    return Thread.currentThread().getName()+"\t"+"调用成功，流水号: " +
serialNumber;
}
public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer
id)
{
    return "id 不能负数，请稍后再试，/(ㄒoㄒ)/~~   id: " +id;
}
```

服务限流

flowlimit. 秒杀等高并发情况 . 见ali sentinel

dashboard

创建一个dashboard模块端口9001作为监控模块

```
1.导入pom依赖
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
2.主启动类
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardMain9001 {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardMain9001.class,args);
    }
}

3.在要监控的模块中也需要导入上面的pom依赖,同时在主启动类中
    /**
     *此配置是为了服务监控而配置，与服务容错本身无关，springcloud升级后的坑
     *ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
     *只要在自己的项目里配置上下面的servlet就可以了
     */
    @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;
    }
```

```
浏览器输入 localhost:9001/hystrix/ 访问

```

Zuul服务网关

Zuul包含了对请求的路由和过滤两个主要的功能

其中路由功能负责将外部请求转发到具体的微服务实列上,是实现外部访问同一入口的基础,而过滤器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中获得其他微服务的消息,也即以后的访问微服务都是提供zuul跳转后获得

Gateway

Spring Cloud Gateway是Spring Cloud的一个全新项目,基于Spring5.0+SpringBoot2.0和Project Reactor等技术开发的网关,它旨在为微服务架构提供一种简单有效的统一的API路由管理方式.

Spring Cloud Gateway作为Spring Cloud生态系统中的网关,目标是代替Zuul,在Spring Cloud2.0以上版本中,没有对新版本的Zuul2.0以上最新高性能版本进行集成,任然是使用Zuul1.x 非Reactor模式的老版本.而为了提升网关的性能,Spring Cloud Gateway是基于WebFlux框架实现的,而WebFlux框架底层使用了高性能的Reactor模式通信框架**Netty**.

Spring Cloud Gateway的目标是提供统一的路由方式且基于Filter链的方式提供网关的基本的功能,列如:安全,监控/指标.和限流.使用的是异步非阻塞.

三大概念 路由(Route),断言(Predicate),过滤(Filter)

路由:路由是构建网关的基本模块,它由ID,目标URI,一系列的断言和过滤器组成,如果断言为true则匹配该路由.

Predicate(断言):参考的是java8的java.util.function.Predicate,开发人员可以匹配HTTP请求中的所有内容(列入请求头或参数),**如果请求与断言相匹配则进行路由**

Filter过滤:指的是Spring框架中GatewayFilter的实列,使用过滤器,可以在请求被路由前或者之后对请求进行修改.

web请求,通过一些匹配条件,定位到真正的服务节点.并在这个转发过程的前后,进行一些精细化控制.

predicate就是我们的匹配条件;而filter就可以理解为一个无所不能的拦截器.有了这两个元素,在加上目标的uri,就可以实现一个具体的路由了.

核心逻辑:路由转发+执行过滤器链

```
新建一个gateway9527模块
导入pom,注意gateway的pom依赖不需要web和actuator
<!--gateway-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

配置
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
```

```
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true #开启从注册中心动态创建路由的功能，利用微服务名进行路由
      routes:
        - id: payment_routh #payment_route    #路由的ID，没有固定规则但要求唯一，建议配合服务名
          uri: http://localhost:8001          #匹配后提供服务的路由地址
          #uri: lb://cloud-payment-service #匹配后提供服务的路由地址
          predicates:
            - Path=/payment/get/**          # 断言，路径相匹配的进行路由

        - id: payment_routh2 #payment_route    #路由的ID，没有固定规则但要求唯一，建议配合服务名
          #uri: http://localhost:8001          #匹配后提供服务的路由地址
          uri: lb://cloud-payment-service #匹配后提供服务的路由地址
          predicates:
            - Path=/payment/lb/**          # 断言，路径相匹配的进行路由
            #- After=2020-02-21T15:51:37.485+08:00[Asia/Shanghai]
            #- Cookie=username,zzyy
            #- Header=X-Request-Id, \d+  # 请求头要有X-Request-Id属性并且值为整数的正则表达式
```

predicate

```
https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.2.2.RELEASE/reference/html/#gateway-request-predicates-factories
```

Spring Cloud Config

Spring Cloud Config为微服务架构中的微服务提供**集中化的外部配置支持**,配置服务器为各个不同的微服务应用的所有环境提供了一个**中心化的外部配置**

Spring Cloud Config分为**服务端**和**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**,用来连接配置服务端并为客户端提供获取配置信息,加密/解密信息等访问接口.

客户端则是通过指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.

用途:

集中管理配置文件,

不同环境不同配置,动态化的配置更新,分环境部署比如 dev/test/prod/beta/release,

运行期间动态调整配置,不再需要在每个服务部署的机器上编写配置文件,服务会向配置中心统一拉取配置自己的信息,

当配置发生变动时,服务不需要重启即可感知到配置的变化并应用新的配置,

将配置以rest接口的形式暴露.

```
git上新建一个仓库存放配置文件
新建一个config-center3344模块服务端,注册到eureka中
1.pom依赖
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>

2.配置文件
spring:
  application:
    name: cloud-config-center
  cloud:
    config:
      server:
        git:
          uri: https://github.com/springcloud-zym/springcloud-config.git
#git仓库名
          search-paths:
            - springcloud-config  #搜索目录
      label: master  #读取分支


```

bootstrap.yml

application.yml是用户级的资源配置文件

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**加载配置文件属性并解析配置.这两个上下文共享一个从外部获取的'Environment'.

"Bootstrap Context"属性有高优先级,默认情况下,它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.

要将Client模块下的application.yml文件改为bootstrap.yml这是很关键的,

因为bootstrap.yml是比application.yml先加载.bootstrap.yml优先级高于application.yml.

```
客户端配置
创建config-client3355客户端模块
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
创建bootstrap.yml
spring:
  application:
    name: config-client
  cloud:
    config:
      label: master #分支名称
      name: config #配置文件名称
      profile: dev #读取后缀名称   环境名称  拼接后即   master分支下的config-dev.yml
      uri: http://localhost:3344

创建controller来访问
public class ConfigClientController
```

```
{
    @Value("${config.info}")
    private String configInfo;

    @GetMapping("/configInfo")
    public String getConfigInfo()
    {
        return configInfo;
    }
}
```

手动刷新客户端配置文件

```
要有监控
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
控制器类上添加 @RefreshScope 注解
当github上的配置文件发生修改时,使用cmd命令行发送
curl -X POST "http://localhost:3355/actuator/refresh" 通知客户端进行刷新,这样就不用客户端重启服务来重新获取配置文件信息.
```

Spring Cloud Bus

Bus支持两种消息代理:RabbitMQ和Kafka

什么是总线:

在微服务架构的系统中,通常会使用**轻量级的消息代理**来构建一个**共用的消息主题**,并让系统中所有微服务实列都连接上来.由于该主题中产生的消息会被所有实列监听和消费,所有称他为消息总线.在总线上的各个实例,都可以方便的广播一些需要让其他连接在该主题上的实例都知道的消息.

基本原理:

ConfigClient实例都监听MQ中同一个topic(默认是SpringCloudBus).当一个服务刷新数据的时候,它会把这个消息放入Topic中,这样其他监听同一个Topic的服务就能得到通知,然后去更新自身的配置.

Spring Cloud Stream

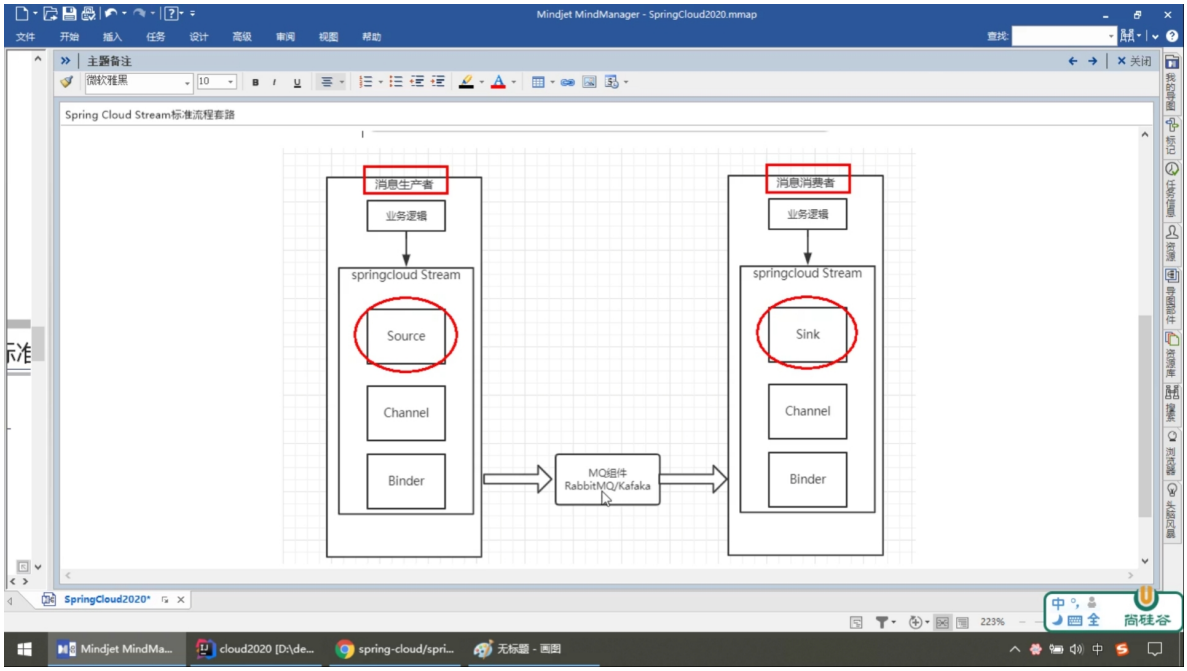
屏蔽底层消息中间件的差异,降低切换成本,统一消息的编程模型.

这些中间件的差异性导致我们实际开发给我们造成了一定的困扰,我们如果用了两个消息队列的其中一种,后面的业务需求,我们想往另一种消息队列进行迁移,这时候无疑是一个灾难性的,**一大堆东西都要重新推到重新做**,因为它和我们的系统耦合了,这时候spring cloud stream给我们提供了一种解耦合的方式.

通过定义绑定器作为中间件,完美的实现了**应用程序与消息中间件细节之间的隔离.**

通过向应用程序暴露统一的Channel通道,使得应用程序不需要再考虑各种不同的消息中间件实现.

- binder:很方便的连接中间件,屏蔽差异
- Channel:通道,是队列Queue的一种抽象,在消息通讯系统中就是实现存储和转发的媒介,通过channel对队列进行配置
- Source和Sink:简单的可理解为参照对象是SpringCloudStream自身,从stream发布消息就是输出,接受消息就是输入.



Spring cloud sleuth

Spring cloud Alibaba

服务限流降级:默认支持Servlet,Feign,RestTemplate,Dubbo和RocketMQ限流降级功能的接入,可以在运行时通过控制台实时修改限流降级规则,还支持查看限流降级Metrics.

服务注册与发现:适配SpringCloud服务注册与发现标准,默认集成了Ribbon的支持.

分布式配置管理:支持分布式系统中的外部化配置,配置更改时自动刷新.

消息驱动能力:基于Spring Cloud Stream为微服务应用构建消息驱动能力.

阿里云对象存储:阿里云提供的海量,安全,低成本,高可靠的云存储服务.支持在任何应用,任何时间,任何地点存储和访问任意类型的数据.

分布式任务调度:提供秒级,精确,高可靠,高可用的定时(基于Cron表达式)任务调度服务.同时提供分布式的任务执行模型,如网格任务,网格任务支持海量子任务均匀分配到所有Worker(schedulerx-client)上执行.

github

Nacos：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

Nacos=Eureka+config+bus

```
启动
下载nacos cmd打开bin/startup.cmd
浏览器输入localhost:8848/nacos  默认用户名和密码都是nacos
```

nacos_discovery

```
1.创建服务提供者cloudalibaba-provider-payment9001
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>

spring:
  application:
```

```
    name: nacos-provider-payment
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

#用于监控，暴露出所有路径
management:
  endpoints:
    web:
      exposure:
        include: "*"

主启动类  @EnableDiscoveryClient
```

```
创建消费者 cloudalibaba-consumer-order83
spring:
  application:
    name: nacos-consumer-order
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

#消费者将要去访问的微服务名称(成功注册进nacos的微服务提供者)
service-url:
  nacos-user-service: http://nacos-provider-payment

配置类注入RestTemplate
@Bean
@LoadBalanced
public RestTemplate getRestTemplate(){
    return new RestTemplate();
}

通过微服务名调用
@RestController
public class NacosConsumerController {

    @Autowired
    private RestTemplate restTemplate;

    @Value("${service-url.nacos-user-service}")
    private String serverUrl;

    @GetMapping("/consumer/nacos/payment/{id}")
    public String orderInfo(@PathVariable("id")Long id){
        return
restTemplate.getForObject(serverUrl+"/nacos/payment/"+id,String.class);
    }
}

```

nacos支持AP和CP的切换

nacos作为服务配置中心

```
新建一个cloudalibaba-config-nacos-client
两个配置文件
bootstrap.yml
spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #Nacos服务注册中心地址
      config:
        server-addr: localhost:8848 #Nacos作为配置中心地址
        file-extension: yaml #指定yaml格式的配置


#
${spring.application.name}-${spring.profile.active}.${spring.cloud.nacos.config.file-extension}
# nacos-config-client-dev.yaml

# nacos-config-client-test.yaml   ----> config.info

application.yml
spring:
  profiles:
    active: dev #表示开发环境


```

nacos集群和持久化配置(重要)

官网

默认nacos使用嵌入式实现数据的存储.所以,如果启动多个默认配置下的nacos节点,**数据存储是存在一致性问题**的.为了解决这个问题,nacos采用了**集中式存储**的方式来支持集群化部署,目前只支持mysql的存储.

将nacos自带的数据库derby的数据迁到mysql

官网

Linux上nacos集群配置

```
备份cluster.conf.example
cp cluster.conf.example cluster.conf

vim cluster.conf
请每行配置成ip:port。（请配置3个或3个以上节点）
# ip:port
172.16.205.38:3333
172.16.205.38:4444
172.16.205.38:5555
```

```
修改nacos的启动脚本startup.sh 使其能够使用不同的端口启动不同的nacos服务
```

Sentinel实现熔断和限流

官网

github

seata处理分布式事务

官网

seata是一款开源的分布式事务解决方案,致力于在微服务架构下提供高性能和简单易用的分布式事务服务.

Transaction ID XID : 全局唯一的事务ID

3组件:

- TC(Transaction Coordinator) - 事务协调者

  维护全局和分支事务的状态，驱动全局事务提交或回滚。

- TM(Transaction Manager) - 事务管理器

  定义全局事务的范围：开始全局事务、提交或回滚全局事务。

- RM(Resource Manager) - 资源管理器

  管理分支事务处理的资源，与TC交谈以注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。