SpringCloud-2

1 服务调用的负载均衡

1.1 问题引出

学习完了注册中心相关知识，在微服务架构中，我们已经可以实现服务的注册与自动发现了。但是再来看看我们的代码

```
        // 服务发现
        List<ServiceInstance> instances = discoveryClient.getInstances("服务名");
        // 选择一个服务提供者
        URI uri = instances.get(0).getUri();
        // 向选择的服务提供者发起请求
        ResponseEntity<String> response = template.getForEntity(uri.toString() +
"/nacos/registry/hello?name={1}", String.class, name);
```

服务是可以有集群的，在发现了一个服务所有的实例之后，在一次服务调用过程中，我们还需要选择其中一个服务实例，发起调用请求，所以发起调用之前还存在着一个选择过程，这就涉及到了选择的策略问题，该按照何种策略选择出集群中的一个实例呢？在SpringCloud中有一个由Ribbon帮我们完成这一选择过程。

1.2 Ribbon负载均衡

Ribbon是一个客户端负载均衡器，能够给HTTP客户端带来灵活的控制。其实现的核心功能，就是一组选择策略，帮助我们在一个服务集群中，选择一个服务实例，并向该实例发起调用请求。它所支持的负载均衡策略如下:

策略	实现类	描述
随机策略	RandomRule	随机选择server
轮训策略	RoundRobinRule	轮询选择
重试策略	RetryRule	对选定的负载均衡策略(轮训)之上重试机制，在一个配置时间段内当选择服务不成功，则一直尝试使用该策略选择一个可用的服务；
最低并发策略	BestAvailableRule	逐个考察服务，如果服务断路器打开，则忽略，再选择其中并发连接最低的服务
可用过滤策略	AvailabilityFilteringRule	过滤掉因一直失败并被标记为circuit tripped的服务，过滤掉那些高并发链接的服务（active connections超过配置的阈值)

策略	实现类	描述
响应时间加权重策略	WeightedResponseTimeRule	根据server的响应时间分配权重，响应时间越长，权重越低，被选择到的概率也就越低。响应时间越短，权重越高，被选中的概率越高，这个策略很贴切，综合了各种因素，比如：网络，磁盘，io等，都直接影响响应时间
区域权重策略	ZoneAvoidanceRule	综合判断服务所在区域的性能，和服务的，轮询选择server并且判断一个AWS Zone的运行性能是否可用，剔除不可用的Zone中的所有server

1.2.1 RestTemplate整合Ribbon

我们希望，在使用RestTemplate发起请求的时候，能"自动选择"其所请求的服务实例，因此我们需要将RestTemplate与Ribbon进行整合。

首先，理论上需要在服务消费者工程中，添加依赖

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
        </dependency>
```

但是，因为nacos-discovery已经自己整合了ribbon依赖，所以实际上我们并不需要去添加该依赖

接着，我们需要修改RestTemplate的配置类，添加@LoadBalance注解

```
@Configuration
public class ClientConfig {

    @Bean
    @LoadBalanced
    public RestTemplate template() {
        return new RestTemplate();
    }
}
```

然后在使用RestTemplate发起调用的时候，直接使用服务名进行调用即可

```
@RestController
@RequestMapping("/call")
public class RegistryConsumerController {
    @Autowired
    RestTemplate template;
```

```
    @GetMapping("/nacos")
    public String consumeNacos(String name) {
        // 注意这里调用的ip地址，使用的是服务名称，而不是真实的ip
        ResponseEntity<String> response = template.getForEntity( "http://nacos-
provider-8002/nacos/registry/hello?name={1}", String.class, name);
        String result = response.getBody();
        return result;
    }
}
```

1.2.2 指定Ribbon负载均衡策略

Ribbon中包含多种负载均衡策略，我们在使用Ribbon的时候，可以指定其负载均衡策略，指定的方式有两种，即配置文件和配置类。

```
# 这里的users是我们的服务名称
users:
  ribbon:
   # 这一行配置的就是实现具体负载均衡策略实现类的全类名
    NFLoadBalancerRuleClassName:    com.netflix.loadbalancer.RandomRule
```

除了使用配置文件的方式，我们还可以使用代码的方式，指定我们所使用的所使用的负载均衡策略

定义配置类

```
@Configuration
public class FooConfiguration {

    // 这里的xxxRule对应的就是
    @Bean
    public IRule ribbonRule() {
        return new xxxRule();
    }
}
```

定义Ribbon客户端配置

```
@Configuration
// 这里的foo即name属性的值表示的是被调用的服务的名称
@RibbonClient(name = "foo", configuration = FooConfiguration.class)
public class MyRestClient {
}
```

但是切记有一个地方需要注意，**我们自己定义的配置类(比如上面的FooConfiguration配置类)，不能被@ComponentScan扫描到**，所以我们可以将其放在一个独立的，与扫描路径无重叠的包里，或者指明不被@ComponentScan注解扫描到，因为这样一来导致的结果就是，对所有服务调用的负载均衡都用的是同一个我们指定的，被扫描到的这个负载均衡策略

当然在实际开发过程中，我们可根据自己的需要，去定义自己的负载均衡策略，我们只需要自己实现IRule接口的实现类，在接口实现中，实现我们自己的负载均衡策略，并用类似于前面代码的配置方式，使我们自定义负载均衡策略生效。

```
public class MyBalanceRule extends AbstractLoadBalancerRule {

    public MyBalanceRule() {
```

```
    }

    @Override
    public void initWithNiwsConfig(IClientConfig clientConfig) {

    }
    /*
         在该方法里实现负载均衡
     */
    @Override
    public Server choose(Object key) {

        List<Server> allServers = getLoadBalancer().getAllServers();
        // 简单粗暴的负载均衡策略
        return allServers.get(0);
    }
}
```

2 面向接口的服务调用

2.1 问题引出

现在我们的服务调用过程，又变得简单了一些，因为Ribbon帮助我们解决了，服务调用过程中的选择问题。再来看一下我们的服务调用代码

```
@RestController
@RequestMapping("/call")
public class RegistryConsumerController {
    @Autowired
    RestTemplate template;

    @GetMapping("/nacos")
    public String consumeNacos(String name) {
        // 注意这里调用的ip地址，使用的是服务名称，而不是真实的ip
        ResponseEntity<String> response = template.getForEntity( "http://nacos-
provider-8002/nacos/registry/hello?name={1}", String.class, name);
        String result = response.getBody();
        return result;
    }
}
```

我们会发现，因为我们是使用RestTemplate这个Http客户端发起的Http协议的服务调用请求，因此在发起请求的时候，我们得自己构建请求url，请求参数，获取响应体数据等等，导致我们的代码和Restful风格的Http请求紧密耦合。

那么有没有办法，让我们在服务调用的时候与Restful的请求“解耦”，直接以Java代码中接口调用的方式，来完成服务的调用呢?

2.2 OpenFeign 客户端

OpenFeign就可以帮助我们实现，让服务调用代码与Restful风格的Http请求解耦的功能。OpenFeign是一个实现Java代码和Http客户端绑定的绑定器，通俗的来解释，它可以帮助我们以统一的方式，将接口"翻译"成Restful风格的请求。

2.2.1 OpenFeign的使用

因为OpenFeign本身，充当着一个“翻译”的角色，可以将我们的Java接口翻译为对应的Http APIs，所以对于我们来说，OpenFeign也可以理解为一种服务调用的客户端，正因为是服务调用的客户端，所以只在服务消费者一端使用。

虽然，OpenFeign本身仅仅只是在客户端使用，但是因为使用了OpenFeign意味着服务的调用是面向Java接口的，而非HTTP API的，调用方式发生了改变，所以我们服务提供者工程的代码结构也要发生改变

其中，feign-provider-api这个工程中，主要放一些公共的接口请求参数，响应类，很显然，feign-provider-8005工程依赖feign-provider-api，因为它需要实现服务对外暴露的接口。

在feign-provider-8005工程中，导入feign-provider-api的依赖

```
 <dependencies>
        <!--SpringCloud ailibaba nacos -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

    </dependencies>
```

定义接口实现类

```
@RestController
public class FeignDemoController {
    @GetMapping("/feign/hello")
    public String sayHello(String name) {
        return "hello, " + name;
    }
}
```

同时，对于服务消费者而言，因为服务消费者需要调用服务提供者暴露出来的接口，所以服务消费者也需要依赖feign-provider-api工程，以及openfeign依赖。

```


    <dependencies>

        <!--SpringCloud ailibaba nacos -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
        </dependency>
        <!--openfeign-->
```

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
        <!--web-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.cskaoyan.micro</groupId>
            <artifactId>feign-provider-api</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>
```

服务消费者，需要以面向接口的方式调用其他服务，需要用到OpenFeign，所以需要定义Feign客户端

```
// 注意，这里FeignClient的名字是调用的服务的名称
@FeignClient("feign-provider-8005")
public interface DemoServiceClient{
    @GetMapping("/feign/hello")
    String sayHello(@RequestParam(name = "name")String name);
}
```

在启动类上加注解@EnableFeignClients，才能让我们定义的FeignClient生效

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class FeignConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(FeignConsumerApplication.class, args);
    }
}
```

2.2.2 FeignClient日志输出

当我们调用FeignClient发出请求的时候，如果我们希望能看到其发出的具体Http请求，我们可以通过配置来实现。

```
# 这里的xxx表示我们自己的定义的FeignClient所在包的包名(比如:
com.cskaoyan.feign.consumer.api)
logging:
  level:
    xxx: debug
```

定义配置类，在配置类中，配置Feign的日志输出级别

```
@Configuration
public class FeignConfig {

    @Bean
    public Logger.Level logLevel() {
        return Logger.Level.FULL;
    }
}
```

这样当我们，通过在对应的FeignClient对象上，调动方法，发起http请求的时候，对应的请求就会打印在控制体

```
2022-06-10 16:43:33.020 DEBUG 21164 --- [nio-7301-exec-2] c.c.f.consumer.api
.FeignProviderClient    : [FeignProviderClient#sayHello] ---> GET
http://feign-provider/say/hello?n=deco HTTP/1.1
```

2.2.3 服务调用的超时设置

通常，一次远程调用过程中，服务消费者不可能无限制的等待服务提供者返回的结果，正常情况下，服务提供者的一次调用执行过程也不会执行很长时间(除非出现网络故障，或者服务提供者宕机等问题)，所以为防止，在非正常情况下服务消费者在调用过程中的长时间阻塞等待，对于一次服务调用过程，我们会设置其超时时间。一次服务调用，超时未返回即认为调用失败。在使用Feign的时候，我们可以配置其超时时间。

```
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
  ReadTimeout: 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
  ConnectTimeout: 5000
```

3 配置中心

3.1 问题引出

设想一下，如果每个服务都有自己的配置，比如服务访问的数据库地址等，但是某一天，数据库部署的服务器地址变了，此时会发生什么呢？为了让服务能够正确访问到数据库，我们得停止每一个服务，重新修改每一个服务的配置文件，然后在重新启动每个服务，在这个过程中就会出现两个问题：

- 修改配置文件的工作繁琐，工作量大，尤其当服务数量较多的时候
- 要让新的配置生效，得重启服务

3.2 配置中心

如果要解决以上问题，那么在我们微服务架构的项目中，我们就得引入一个新的角色——配置中心来解决这个问题了，类似于注册中心，配置中心的实现也有多种，而Nacos同时也实现了配置中心的角色。

- 使用配置中心可以让您以中心化、外部化和动态化(动态化即可以实时刷新配置)的方式管理所有环境的应用配置和服务配置。
- 动态配置消除了配置变更时重新部署应用和服务的需要，让配置管理变得更加高效和敏捷。

3.3 Nacos 配置中心

Nacos除了可以作为服务注册中心之外，还可以实现服务配置中心的功能。

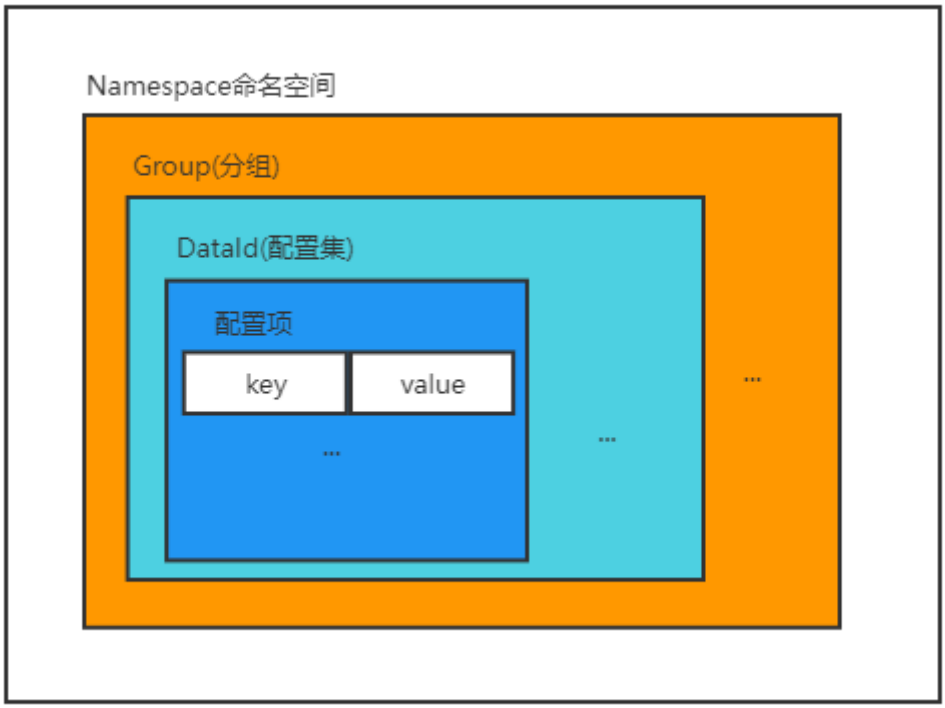
在使用Nacos配置中心之前，我们必须对于注册中心的配置信息有一个清楚的认识：

- 配置中心中的配置，主要是以键值对的形式存在的，即每条配置都以key-value的形式存储，key是配置的名称，value才是配置的值
- 所以，很明显，不同配置的key值应该有所区别，或者即使key值相同，我们也应该有办法区分他们，即给key值划分不同的维度。

所以接下来，我们得介绍一下Nacos中定义的基本概念：

- **配置项**: 一个具体的可配置的参数与其值域，通常以 param-key=param-value 的形式存在。例如我们常配置系统的日志输出级别（logLevel=INFO|WARN|ERROR） 就是一个配置项。
- **配置集**：一组相关或者不相关的配置项的集合称为配置集。在系统中，一个配置文件通常就是一个配置集，包含了系统各个方面的配置，每一个配置集都对应一个唯一的DataId，DataId必须由我们自己定义。
- **配置分组**：Nacos 中的一组配置集，是组织配置的维度之一，每一个分组都有一个唯一的组名，如果我们未定义，则默认使用DEFAULT-GROUP分组
- **命名空间**：用于进行用户粒度的配置隔离，每一个命名空间都有一个唯一的Id值，如果我们未定义，则默认使用public命名空间

以上几个概念其实就是在告诉我们区分不同配置项的维度，Nacos提供多个维度帮助我们区分不同的配置，它们的关系如下图所示



有了以上不同的配置项的划分维度，我们就可以灵活定义我们的配置项了。其中

- 配置项中的key值，以及配置分组的组名都由我们自己根据场景去定义
- 命名空间的Id值，在我们定义命名空间的时候，由Nacos帮我们生成
- 在一个服务启动的时候，默认读取的配置集id即data_id和该服务的配置有关，按照如下公式计算:

```yaml
spring:
  application:
  name: xxx
  profiles:
    active: dev
  cloud:
    nacos:
      config:
        #namespace: xxx
        #group: xxx
        server-addr: 127.0.0.1:8848
        file-extension: yaml
```

```
${prefix}-${spring.profiles.active}.${file-extension}
```

- `prefix` 默认为 `spring.application.name` 的值，也可以通过配置项 `spring.cloud.nacos.config.prefix` 来配置。
- `spring.profiles.active` 即为当前环境对应的 profile，**注意：当 `spring.profiles.active` 为空时，对应的连接符 `-` 也将不存在，dataId 的拼接格式变成 `${prefix}.${file-extension}`**
- `file-exetension` 为配置内容的数据格式，可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。目前只支持 `properties` 和 `yaml` 类型。

这里要主要，我们的SpringBoot项目在启动的时候，就会根据${prefix}-${spring.profiles.active}.${file-extension}生成配置集名称，并自动去读取该配置集名称对应的配置。

- 除此之外，如果有多个服务具有一些共享的配置，我们可以在配置文件中指定读取某个共享的配置集

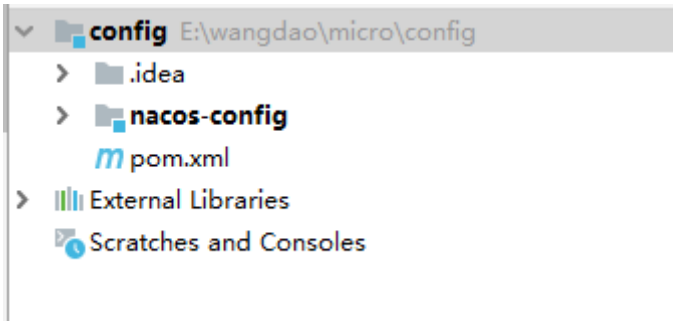
```yaml
spring:
  cloud:
      config:
        server-addr: 127.0.0.1:8848
        shared-configs:
          - data-id: common.yaml
            group: xxx
            refresh: true #是否支持动态刷新
```

注意:

- 如果相同key的配置同时出现在profile粒度的配置集(即${prefix}-${spring.profiles.active}.${file-extension}配置集) 和 shared-config配置集中，它们的优先级关系是：profile配置 > shared-config配置
- 配置中心的优先级配置，都高于本地配置

3.2.1 Nacos配置中心的使用

工程结构如下



项目仍然采用父子工程，父工程 中包含依赖

```
    <dependencyManagement>
        <dependencies>
            <!--SpringCloudAlibaba-->
            <dependency>
                <groupId>com.alibaba.cloud</groupId>
                <artifactId>spring-cloud-alibaba-dependencies</artifactId>
                <version>2.1.0.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--springCloud的依赖-->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Hoxton.SR1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
            <!--SpringBoot-->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.2.2.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

nacos-config子工程中添加如下依赖

```
 <dependencies>
        <!--SpringCloud ailibaba nacos -->
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
        </dependency>
        <!--<dependency>-->
            <!--<groupId>com.alibaba.cloud</groupId>-->
            <!--<artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>-->
        <!--</dependency>-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
```

```
        </dependency>
    </dependencies>
```

在nacos-config的目录下添加两个配置文件，分别是bootstrap.yml和application.yml，在bootstrap配置文件中必须有应用名，以及nacos服务器地址等，而在application.yml配置文件中，配置自己项目中的其他配置

```yaml
# 项目中的其他配置都包含在application.yml文件中
server:
  port: 3377
```

```yaml
# bootstrap.yml文件内容
spring:
  application:
    name: nacos-config-client
  profiles:
    active: dev # 表示开发环境
  cloud:
    nacos:
      config:
        #Nacos作为配置中心地址
        server-addr: localhost:8848
        #指定yaml格式的配置，如果是yml文件，注意这里写的是yaml!
        file-extension: yaml
        # 配置所属的配置分组
        # group: DEV_GROUP
        # 配置所属的命名空间
        # namespace: 7d8f0f5a-6a53-4785-9686-dd460158e5d4
        # shared-configs:
        #  - data-id: common.yaml
        #    group: xxx
        #    refresh: true #是否支持动态刷新
```

测试代码如下：

```java
@RestController
@RequestMapping("config")
// 通过 Spring Cloud 原生注解 @RefreshScope 实现配置自动更新
@RefreshScope
public class ConfigController {

    @Value("${nacos.config}")
    String config;

    @Value("${shared.config}")
    String sharedConfig;

    @GetMapping("/nacos/config")
    public String sayHello() {
        return config;
    }
```

```
    @GetMapping("/shared/config")
    public String sharedConfigHello() {
        return sharedConfig;
    }
}
```

5.2.2 Nacos 配置的持久化

我们在Nacos服务器上写入的配置，会被持久化保存到Nacos自带的一个嵌入式数据库derby中，因此当我们重启Nacos之后，仍然可以看到之前的配置信息。但是，使用嵌入式数据库实现数据的存储，不方便观察数据存储的基本情况，因此，Nacos还支持将配置信息写入Mysql中：

- 在数据库中，创建名为nacos的数据库
- 在nacos数据库中，执行数据库初始化文件：nacos-mysql.sql(改文件在conf目录下已经提供)
- .修改conf/application.properties文件，增加支持mysql数据源配置（目前只支持mysql），添加mysql数据源的url、用户名和密码。

```
spring.datasource.platform=mysql
db.num=1
# 这里的url要改成你自己的mysql数据库地址，并在你的mysql中创建名为nacos的数据库
db.url.0=jdbc:mysql://11.162.196.16:3306/nacos?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
# 这里要改成你自己登录mysql的用户名和密码
db.user.0=nacos_devtest
db.password.0=youdontknow
```

在配置了mysql数据库之后，我们会发现，之前配置中心的配置信息全部消失了，那是因为我们之前使用的是nacos的内嵌数据库derby，现在切换到mysql之后数据存储在nacos这个数据库中，而该数据库现在是没有数据的。

但是当我们，在nacos的控制台重新添加配置数据之后，我们就可以在mysql中看到了