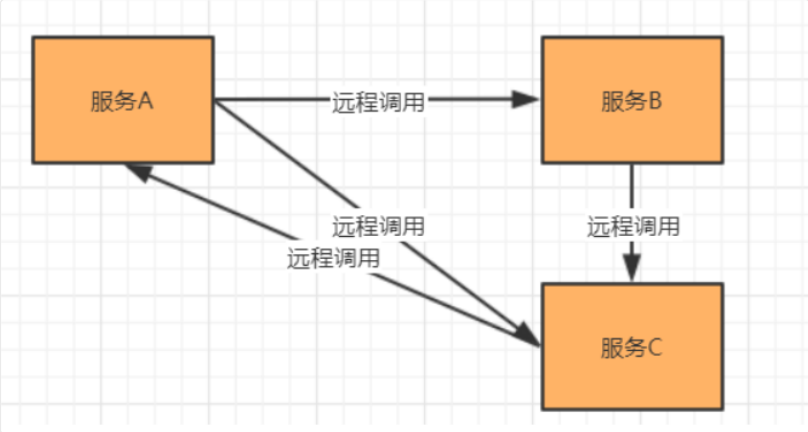
服务调用| Http、Ribbon、OpenFeign

服务调用，即在分布式系统中一个服务对另一个服务进行远程调用。如果调用者调用注册中心上的服务，那么调用者需要配置为该注册中心的客户端。

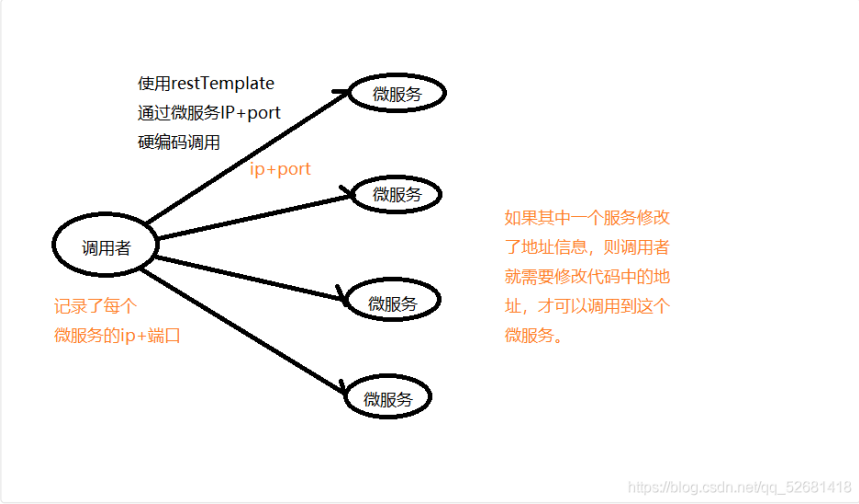


Http 请求调用

在分布式系统中，不同的服务间通讯可以借助Http异步请求的方式来实现不同服务之间的通讯，常见的Http客户端有如下：

- SpringBoot自带的RestTemplate对象
- Unirest
- HttpClient
- Hutool 包的HttpUtil
- OkHttp

上面的这些工具都可以在java程序中发送http请求，从而获取其它服务接口响应的数据。



在这里插入图片描述

这种方式有个很明显的问题：调用时目标服务地址写在代码中，当系统体量变大，服务变多后这些地址将难以维护。

构建消费者

SpringBoot内置了HTTP请求模板RestTemplate，因此我们能够之间使用。

注入RestTemplate：

```java
@SpringBootApplication
public class MainApplication {
```

```java
  @Bean//将RestTemplate注册到容器
  public RestTemplate RestTemplate(){
      return new RestTemplate();
  }

  public static void main(String[] args) {
     SpringApplication.run(MainApplication.class,args);
  }

}
```

调用其它服务：

```java
@Autowired//注入RestTemplate
private RestTemplate template;

@RequestMapping("/img/{id}")
public Img getImg(@PathVariable long id){

   // 调用图片服务接口，并接收响应数据
   Img img = template.getForObject("http://localhost:9092/img/findimg/"+id,Img.class);
   return img;

}
```

Ribbon 调用

Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。Eureka、Consul、甚至接下来的OpenFeign均集成了Ribbon。

构建消费者

引入ribbon

```xml
<!--Ribbon依赖：Eureka、Consul均集成了，因此这些依赖实际上已经被导入了，不用手动添加 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
    <version>2.2.1.RELEASE</version>
</dependency>
```

注入RestTemplate

```java
@LoadBalanced     // 实现负载均衡
@Bean             // 将RestTemplate注册到容器
public RestTemplate RestTemplate(){
   return new RestTemplate();
}
```

从注册中心调用服务：img-service

```java
@Autowired//注入RestTemplate
private RestTemplate rt;

//Ribbon获取服务
@RequestMapping("/img3/{id}")
public Img ribbonimg(@PathVariable long id){
   Img img=rt.getForObject("http://img-service/img/findimg/"+id,Img.class);//服务名
   return img;
}
```

负载均衡

将某一服务部署成多个同样的节点，大量请求被随机分发到这些节点上，从而避免单节点压力过大。负载均衡的作用就是使请求均匀地分布在每一个节点上。

通常使用默认均衡策略即可，但我们仍可修改该策略：

```yaml
```

```yaml
#修改Ribbon负载均衡策略：服务名.ribbon.NFLoadBalancerRuleClassName.策略
img-service:
 ribbon:
   NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

请求重试

在某一个服务的多个节点中某个节点挂了，那么访问该节点的请求就会进行请求重试，转而访问其他节点。

为调用者服务配置请求重试：

XML

```xml
<!--spring 重试组件-->
<dependency>
   <groupId>org.springframework.retry</groupId>
   <artifactId>spring-retry</artifactId>
</dependency>
```

添加配置

YAML

```yaml
spring:
  cloud:
    loadbalancer:
      retry:
        enabled: true #开启springCloud的重试功能，默认开启
img-service:
  ribbon:
    ConnectTimeout: 250           #ribbon连接的超时时间
    ReadTimeout: 1000             #ribbon数据读取的超时时间
    OkToRetryOnAllperations: true #是否对所有操作都重试
    MaxAutoRetriesNextServer: 1   #切换实例的重试次数
    MaxAutoRetries: 1             #当前实例的重试次数
```

OpenFeign 调用

SpringCloud 对 feign 组件进行了增强，使其支持 SpringMVC 注解、整合了 Ribbon 和 Eureka。openFeign 是一个十分值得推荐的调用方式。

构建消费者

引入依赖：

XML

```xml
<!--SpringCloud整合的openFeign-->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

<!--引入EurekaClient-->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

注解激活 OpenFeign

Java

```java
@SpringBootApplication
@EnableFeignClients    // 激活Fegin，注意写在配置类上会启动出错
public class MainApplication {

  public static void main(String[] args) {
       SpringApplication.run(MainApplication.class,args);
   }
}
```

创建接口并在接口上添加注解来实现调用：

```java
@Component
@FeignClient(name="img-service")  // 声明要调用的微服务名，name值为服务名
public interface ImgFeignClient {

    // 配置需要调用的微服务接口
    @RequestMapping(value = "/img/findimg/{id}",method = RequestMethod.GET)
    public Img findid( @PathVariable long id);

}
```

将进行调用的接口作为业务层使用

```java
@Autowired
private ImgFeignClient ifc;  //此处报红线，不用担心，是加载过慢

//feign获取服务
@RequestMapping("/img4/{id}")
public Img fignimg(@PathVariable long id){
    Img img=ifc.findid(id);
    return img;
}
```

负载均衡

Feign 中本身已经集成了 Ribbon 依赖和自动配置，因此我们不需要额外引入依赖.也不需要额外配置。

服务降级

服务降级是指某一服务请求到达上限后，提供一个低配的服务来处理多余的请求，以此来使用户有一个良好的体验。

OpenFeign 内部集成了 Hystrix，通过 Hystrix，可以实现服务的降级操作。

配置 hystrix 信息

```yaml
feign:
  hystrix: #开启对hystrix的支持
    enabled: true
```

实现要保护的接口类，并将实现类注册到容器：假定调用服务创建的接口为 ImgFeignClient 。

```java
@Component
public class ImgFeignClientCallBack implements ImgFeignClient {

    @Override // 熔断降级的方法
    public Img findid(long id) {
        Img img=new Img();
        img.setName("触发降级方法");
        return img;
    }
}
```

调用服务的接口上添加注解，指定用来降级的实现类(因为此接口实现不止一个)：

```java
//  fallback:指定服务降级方法，即实现类
@FeignClient(name="img-service",fallback= ImgFeignClientCallBack.class)
public interface ImgFeignClient {

    // 配置需要调用的微服务接口
    @RequestMapping(value = "/img/findimg/{id}",method = RequestMethod.GET)
    public Img findid( @PathVariable long id);

}
```

为什么说此接口实现类不止一个呢，通过接口加注解来调用服务，不就相当于内置提供了一个实现吗。

请求压缩

SpringCloud Feign 支持对请求、响应进行 GZIP 压缩，减少性能损耗。

请求压缩配置：

```yaml
feign:
  compression:
    request:
      enabled: true #开启请求压缩
    response:
      enabled: true #开启响应压缩
```

对请求的数据类型、触发压缩的大小下限。

```yaml
feign:
  compression:
    request:
      enabled: true #开启请求压缩
      mime-types: text/html,application/xml,application/json  #设置压缩的数据类型
      min-request-size: 2048 #设置触发压缩的大小下限
```

日志配置

```yaml
feign:
client:
 config:
  img-service: #此处为服务名
    loggerLevel: FULL
        #NONE：不输出日志(性能最好)。
                #BASIC：适用于生产环境追踪问题
        #HEADERS:在BASIC的基础上记录请求和响应头信息
        #FULL:记录所有
logging:
  level:
    com.feign.ImgFeignClient: debug1  #接口全类名
```

自定义 Feign

从Spring Cloud Edgware开始，Feign支持使用属性自定义Feign.

```yaml
feign:
   client:
     config:
       feignName: ##定义FeginClient的名称
         connectTimeout: 5000  # 相当于Request.Options
         readTimeout: 5000   # 相当于Request.Options
         # 配置Feign的日志级别，相当于代码配置方式中的Logger
         loggerLevel: full
         # Feign的错误解码器，相当于代码配置方式中的ErrorDecoder
         errorDecoder: com.example.SimpleErrorDecoder
         # 配置重试，相当于代码配置方式中的Retryer
         retryer: com.example.SimpleRetryer
         # 配置拦截器，相当于代码配置方式中的RequestInterceptor
         requestInterceptors:
           - com.example.FooRequestInterceptor
           - com.example.BarRequestInterceptor
         decode404: false

```