11 服务注册与消费

更新时间：2019-06-19 17:55:45

> 不想当将军的士兵，不是好士兵。
>
> ——拿破仑

前面几篇文章向读者介绍了 Eureka 和 Consul 的基本用法，并对原理做了细致地分析。对于服务注册与调用，在前面的文章中也有涉及，但是没有细说，本文我们就来看看微服务中的服务注册与消费。

系统架构

其实，在微服务出现之前，跨服务调用的需求就有了，只不过以前大多数人可能会采用直连的方式去调用另一个服务。例如有 Server A 和 Server B 两个服务，当 Server A 需要去调用 Server B 的时候，直接在 Server A 中写上 Server B 的地址去调用就行了，像下面这样：

这种硬编码的方式缺陷很明显，Server B 的地址一旦被写死了，就必须一动不动地呆在一个地方，Server B 如果想要上线新的集群，还得修改 Server A 的代码。在微服务架构下，系统调用将不再是这种方式了，请看下图：

此时的服务调用将分为两个过程：

1. Server A 去服务注册中心拿到 Server B 的地址，如果 Server B 是单机部署，这个地址就只有一个，如果 Server B 是集群化部署，这个地址就有多个；
2. 拿到 Server B 的地址之后，Server A 再去调用 Server B。

这样做，看着比上面的方式更麻烦了，但是却带来了更多的好处——例如服务的之间调用时，互相之间的地址不用写死，需要的时候直接去服务注册中心获取，服务之间也可以方便进行集群化部署、使用负载均衡功能等。

本文我们就先来搭建图二这样一个简单的微服务架构。

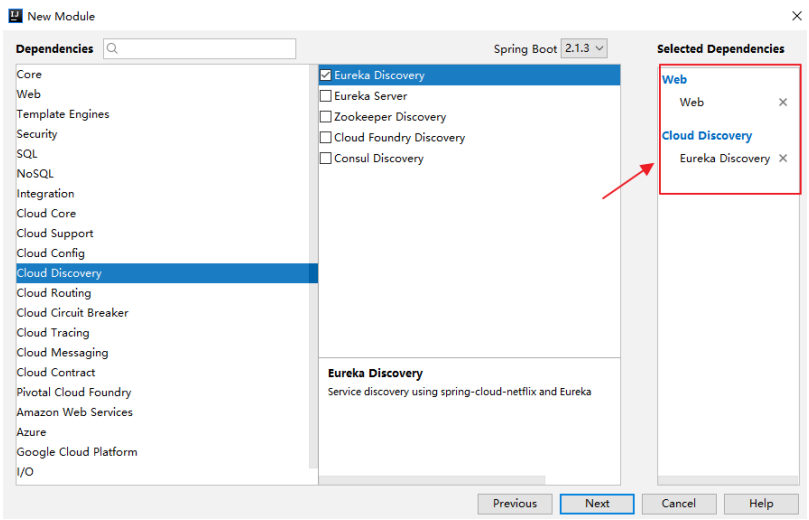
服务注册与消费

1. 首先启动 Eureka

首先我们创建一个名为 ServiceRegistry 的空的 maven 项目，然后创建一个 Eureka 服务注册中心作为子模块并启动，Eureka 的配置和前面文章的一致，这里我们只需要启动一个单机的 Eureka 即可。关于 Eureka 的配置和启动，如果读者有不懂的地方，可以参考本专栏前面的文章，这里我就不再赘述。

2. 创建 provider

接下来创建一个 Provider 作为消息提供者，这就是一个普通的 Spring Boot 工程，创建时候注意添加Web依赖以及 Eureka Discovery 依赖：



项目 pom 文件完整依赖如下：

```
<properties>
    <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

项目创建成功后，在 application.properties 中添加如下配置：

```
spring.application.name=provider
server.port=4001
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

三行配置含义分别如下：

- spring.application.name 表示当前服务的名字，这个名字将作为服务的标记存储在Eureka上，当其他服务需要调用这个服务的时候，都是通过这个名字来查找服务。
- server.port 当前服务的端口。
- 最后一个地址表示当前服务需要注册到的服务注册中心地址，这里需要注意，如果服务注册中心是一个集群，这里也可以只写集群中的一个节点，Eureka 集群会自动进行服务同步。

在微服务中，只要当前项目的 classpath 下存在 `spring-cloud-starter-netflix-eureka-client` 依赖，并且提供了 eureka 注册中心的地址，该服务就会自动注册到 Eureka Server 上。

然后在 provider 中提供一个 `/hello` 接口，供其他服务调用，如下：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(String name) {
        return "hello " + name + " !";
    }
}
```

至此，我们的服务提供者就算有了。

3. 创建 consumer

有了服务提供者，接下来我们再来创建服务消费者 consumer。首先创建一个普通的 Spring Boot 工程，创建时依然记得添加 web 和 eureka discovery 依赖；创建工程后，在 application.properties 中的配置和 provider 基本一致，只是服务端口变了而已，如下：

```
spring.application.name=consumer
server.port=4002
eureka.client.service-url.defaultZone=http://localhost:1111/eureka
```

这段配置的含义参考 provider 配置，不再赘述。

接下来，在 consumer 的启动类中添加一个 RestTemplate 的实例。RestTemplate 是一个 Spring 提供的 HTTP 请求工具，关于这个 RestTemplate 的详细用法下篇文章会给大家详细介绍，这里就先简单了解下，提供 RestTemplate 实例的代码如下：

```
@SpringBootApplication
public class ConsumerApplication {

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

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

最后，再在 consumer 中添加一个 UseHelloController ，在这里去调用 provider 提供的服务，代码如下：

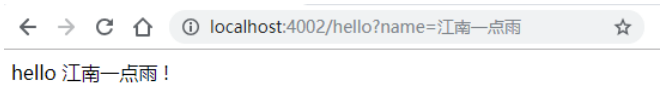
```
@RestController
public class UseHelloController {
    @Autowired
    DiscoveryClient discoveryClient;
    @Autowired
    RestTemplate restTemplate;
    @GetMapping("/hello")
    public String hello(String name) {
        List<ServiceInstance> list = discoveryClient.getInstances("provider");
        ServiceInstance instance = list.get(0);
        String host = instance.getHost();
        int port = instance.getPort();
        String s = restTemplate.getForObject("http://" + host + ":" + port + "/hello?name={1}", String.class, name);
        return s;
    }
}
```

这段代码对应的解释如下：

- 首先注入一个 DiscoveryClient，DiscoveryClient 可以用来从 Eureka 或者 Consul 上根据服务名查询一个服务的详细信息，注意 DiscoveryClient 的全路径是 `org.springframework.cloud.client.discovery.DiscoveryClient` ，大家不要导错包。至于这个 DiscoveryClient 从哪里来，下个小节和大家细说。
- 注入一个 RestTemplate 的实例，这个 RestTemplate 将用来发送 HTTP 请求，这个工具发送 HTTP 请求比较省事一些，当然读者在这里也可以选择使用 Java 自带的 HttpUrlConnection 或者其他第三方工具来发送 HTTP 请求。
- 在方法中，首先调用 DiscoveryClient 实例的 getInstances 方法，方法的参数就是要调用的微服务的名字，返回结果是一个 List 集合中放着 ServiceInstance 。之所以是 List 集合，那是因为 provider 可能是单机部署，也可能是集群部署，如果是集群部署的话，provider 实例就会有多个。
- ServiceInstance 中则保存了一个实例的详细信息，例如 host、port、schema、instanceId 等，ServiceInstance 也是一个接口，它有多个实现，这里使用的实现类是 EurekaServiceInstance。
- 由于这里我的 provider 实例只有一个，因此这里使用 list.get(0) 来获取实例。
- 从 ServiceInstance 中提取出微服务的关键地址信息 host 和 port ，然后拼接成一个请求地址。
- RestTemplate 的 getForObject 方法接收三个参数。第一个参数是请求地址，请求地址中的 {1} 表示一个参数占

位符，第一个参数 String.class 表示返回的参数类型，第三个参数则是一个第一个占位符的具体值。

好了，经过如上步骤之后，此时我们分别启动 Eureka、provider 以及 consumer，然后在浏览器中访问 consumer 的 hello 接口，结果如下：



可以看到，我们已经成功在 consumer 中调用 provider 提供的服务了。

4. DiscoveryClient从哪儿来？

在上面一小节中，我们提到一个东西叫做 DiscoveryClient，那么这个东西到底是怎么来的呢？ 根据注释的描述，这个 DiscoveryClient 可以用来从 Eureka 或者 Consul 中查到一个服务的实例，不过这个 DiscoveryClient 只是一个接口而已，具体的还得一个类去实现，这个具体的实现类就是 CompositeDiscoveryClient ,当我们一个微服务启动时，在 CompositeDiscoveryClientAutoConfiguration 类中会自动配置一个 CompositeDiscoveryClient 的实例，CompositeDiscoveryClientAutoConfiguration 类源码如下：

```
@Configuration
@AutoConfigureBefore(SimpleDiscoveryClientAutoConfiguration.class)
public class CompositeDiscoveryClientAutoConfiguration {

 @Bean
 @Primary
 public CompositeDiscoveryClient compositeDiscoveryClient(
  List<DiscoveryClient> discoveryClients) {
 return new CompositeDiscoveryClient(discoveryClients);
 }

}
```

当我们满心欢喜地来到 CompositeDiscoveryClient 类中，以为服务查找就是调用 CompositeDiscoveryClient 的 getInstances 方法，却发现事情好像没那么简单！ 真正的调用类是 CompositeDiscoveryClient 类中的 discoveryClients 属性提供的 DiscoveryClient，而 discoveryClients 属性默认集合中只有一条数据， 那就是 EurekaDiscoveryClient ，最终在 EurekaDiscoveryClient 类中，通过它里边的 EurekaClient 来获取了一个微服务的详细信息。

小结

本文通过一个简单的案例，向大家展示了微服务中服务之间的调用过程。大家在学习过程中主要把握两个过程：

1. Server A 先根据 Server B 的名字去 Eureka 上获取 Server B 的地址；
2. Server A 有了 Server B 的地址后，再去调用 Server B。

本质上，调用过程就是一个 HTTP 请求，看着简单，其实并不简单，因为真正的调用过程中涉及到的问题非常多，例如负载均衡、服务容错、服务降级、异常处理、请求合并等等，这些问题，我们在后面的文章中都会向大家一一介绍。

本文作者：纯洁的微笑、江南一点雨