第3章 chapter 3

注册和发现服务

本章学习目标

- 了解注册和发现服务的基本概念及 Nacos 的优势
- 熟悉 Spring Cloud Nacos 的使用方式，包括注册服务、发现服务、配置中心等
- 理解 Spring Cloud 调用服务的方式，包括 Feign、Ribbon、OpenFeign、LoadBalancer、RestTemplate 和 WebClient
- 掌握在 Spring Cloud Nacos 中调用服务的方法和技巧

本章准备工作

开发者需要提前准备的开发环境和开发工具包括 IDEA、JDK 11＋、Maven 3.0＋、Nacos 2.1.0＋、MySQL 5.6.5＋。

本章将介绍在 Spring Cloud 中使用 Nacos 注册和发现服务的方式。首先介绍 Nacos 的概述和功能特性，其次将演示使用 Spring Cloud Nacos 注册和发现服务，最后将介绍使用 Nacos 实现负载均衡和高可用性服务的方法。本章旨在为读者提供全面的 Nacos 使用指南，以及在实际项目中使用 Nacos 解决一些常见问题的经验。

3.1 背 景 介 绍

注册和发现是一种重要的微服务架构模式，它是在微服务之间实现通信和协作的核心机制之一。在微服务架构中，系统会被拆分成多个服务单元，每个服务单元都具有独立的代码库和数据存储，服务之间通过网络进行通信和协作，其中一个服务可能需要调用另一个服务提供的功能。在这种情况下，服务调用方需要知道服务提供者的网络地址和端口号，这就需要有一个注册中心记录所有服务的地址信息。

注册和发现机制的核心是注册中心。每个服务在启动时会向注册中心注册自己的信息，包括服务名、IP 地址、端口号等。注册中心将这些信息存储起来，当服务调用方需要调用另一个服务时，它会向注册中心发起请求，请求服务的信息。注册中心将返回所

有符合条件的服务实例信息给服务调用方，服务调用方再根据自己的负载均衡策略选择其中一台服务器实现调用。

注册和发现机制的优势在于可以实时感知服务实例的动态变化（如新增或下线）。这可以使微服务架构更加灵活和可靠。例如，某个服务实例不可用，那么注册中心可以将其从可用服务列表中删除，从而避免了服务调用方无法正确调用的情况发生。此外，注册发现机制还可以支持服务的版本管理和灰度发布等高级功能。

不同的注册和发现实现方式有所不同。目前比较流行的注册和发现框架包括ZooKeeper、Consul、Etcd、Eureka 和 Nacos 等。其中，Nacos 是一个由阿里巴巴开源的、新型的服务发现、配置中心和元数据中心，具有高可用、可扩展、支持多种协议等优点，故其被越来越多的企业和开发者使用。


Nacos 的安装与配置

3.2 Nacos 的安装与配置

3.2.1 Nacos 的下载与安装

首先，访问 Nacos 的 GitHub 仓库——alibaba/nacos，下载 Nacos 的二进制包或源码包。下载完成后，解压缩即可，如图 3-1 所示。

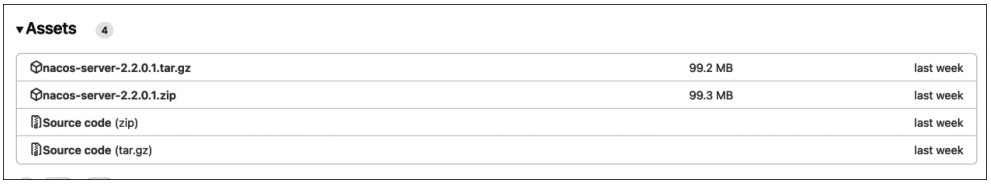

图 3-1 GitHub 仓库中的 releases

图 3-1 中有四个文件链接，分两种扩展名（gz 和 zip），文件内容都差不多，Windows 系统用户建议选择 zip 格式，Linux 类系统用户建议选择 gz 格式。

由于是压缩包，因此开发者无须安装，直接解压即可，解压完之后，切换到 bin 目录下。

Nacos 依赖 Java 运行时环境。故开发者应正确安装 JDK 11＋并正确配置环境变量。

1. 必要配置

找到 Nacos 根目录下的 conf 目录，打开 application.properties 文件，修改以下几处配置。

```
#启用鉴权，需要用户名、密码才能登录
nacos.core.auth.system.type=nacos
nacos.core.auth.enabled=true
```

```
#2.2.0.1 版本后没有默认值,要求开发者自定义
#推荐将配置项设置为 Base64 编码的字符串,且原始密钥长度不得低于 32 个字符

nacos.core.auth.plugin.nacos.token.secret.key=
MTIzNDU2NzgxMjM0NTY3ODE yMzQ1Njc4MTIzNDU2Nzg=
```

2. 启动服务

(1) 在 Linux/UNIX/macOS 系统下运行。直接执行“启动命令”(standalone 代表单机模式运行,非集群模式)。

```
sh startup.sh -m standalone
```

如果使用的是 Ubuntu 系统,或者运行脚本报错提示“[[”符号找不到,那么可尝试以如下方式运行。

```
bash startup.sh -m standalone
```

(2) 在 Windows 系统下运行。打开“命令提示符”,执行“启动命令”(standalone 代表单机模式运行,非集群模式)。

```
startup.cmd -m standalone
```

3. 启动成功

启动成功后,Nacos 的日志文件会存放在 logs/start.out 文件中。如果日志中没有异常,最后输出的将是下面这行提示。

```
Nacos started successfully in stand alone mode. use embedded storage
```

4. 关闭服务

(1) 在 Linux/UNIX/macOS 系统下运行,命令如下。

```
sh shutdown.sh
```

(2) 在 Windows 系统下运行,命令如下。或者双击 shutdown.cmd 运行文件,效果相同。

```
shutdown.cmd
```

3.2.2 Nacos 的管理界面

本示例在本机成功安装了 Nacos,并且按照 3.2.1 节讲述的步骤成功启动了 Nacos。

本地环境下默认的访问地址是 http://localhost:8848/nacos，登录界面如图 3-2 所示。

图 3-2 Nacos 登录界面

开发者可以使用默认的用户名和密码登录，默认用户名为 nacos，默认密码为 nacos。登录成功之后，看到的 Nacos 控制台界面如图 3-3 所示。

图 3-3 Nacos 控制台界面

Nacos 服务的注册

3.3 服务的注册和发现

3.3.1 服务的注册

在微服务架构中，服务的注册是指服务实例将自己的信息（如服务名、IP 地址、端口号等）发送到注册中心实现注册。下面以一个简单的示例说明服务的注册过程。

假设现在有一个服务提供者（provider）和一个服务消费者（consumer）。服务提供者提供了一个计算两个整数相加之和的服务（addService），服务消费者需要使用该服务计算两个整数的和。

这个示例将使用 Nacos 作为注册中心。首先,需要在服务提供者的项目中添加 Nacos 客户端依赖,然后在服务提供者的启动类中添加以下代码实现注册服务。

1. 修改项目的 pom.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.7</version>
    <relativePath/><!--lookup parent from repository -->
  </parent>
  <groupId>com.etoak.tutorial.nacos</groupId>
  <artifactId>provider</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>client</name>
  <description>Nacos 注册和发现示例中的提供者</description>

  <properties>
    <java.version>1.8</java.version>
    <spring-cloud-alibaba.version>2021.1</spring-cloud-alibaba.version>
    <spring-cloud.version>2021.0.3</spring-cloud.version>
  </properties>

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

  <dependencyManagement>
```

```
    <dependencies>
      <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>${spring-cloud.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-alibaba-dependencies</artifactId>
        <version>${spring-cloud-alibaba.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
    <build>
      <plugins>
        <plugin>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
      </plugins>
  </build>
</project>
```

上面这段代码的核心内容如下。这个依赖给微服务提供了与 Nacos-Server 对接的能力，从而实现了注册服务和发现服务。

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

2. 修改项目的 application.yml 文件

```
server:
    port: 8081

  spring:
    application:
```

```
    name: provider
  #Nacos 注册和发现的配置
  cloud:
    nacos:
      discovery:
        #Nacos-Server 的地址
        server-addr: 127.0.0.1:8848
        #默认的命名空间
        namespace: public
        username: nacos
        password: nacos
```

3. 修改项目启动类 App.java

```
package com.etoak.tutorial.nacos;

  import org.springframework.boot.SpringApplication;
  import org.springframework.boot.autoconfigure.SpringBootApplication;

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

4. 添加类 AddController.java

```
package com.etoak.tutorial.nacos;

  import org.springframework.web.bind.annotation.RequestMapping;
  import org.springframework.web.bind.annotation.RequestParam;
  import org.springframework.web.bind.annotation.RestController;

  @RestController
  public class AddController {
    @RequestMapping("/add")
    public String add(@RequestParam int a, @RequestParam int b) {
      int result =a +b;
      return "The result is " +result +" from remote service [provider]";
    }
  }
```

上面代码中的 add()方法需要两个整型数值作为参数，其返回值为两个参数相加的结果(String 类型)，add()方法将被用于接下来验证服务消费者。

接下来启动服务，启动成功后，日志中有一句重要的提示表示注册的动作已经完成。

```
nacos registry, DEFAULT_GROUP provider 192.168.1.104:8081 register finished
com.etoak.tutorial.nacos.App: Started App in 5.171 seconds (JVM running for 7.423)
```

通过 Nacos 控制台可以看到 provider 服务的注册情况，如图 3-4 所示。

图 3-4　服务的注册情况

单击“详情”按钮可以看到服务注册的详细情况，如图 3-5 所示。

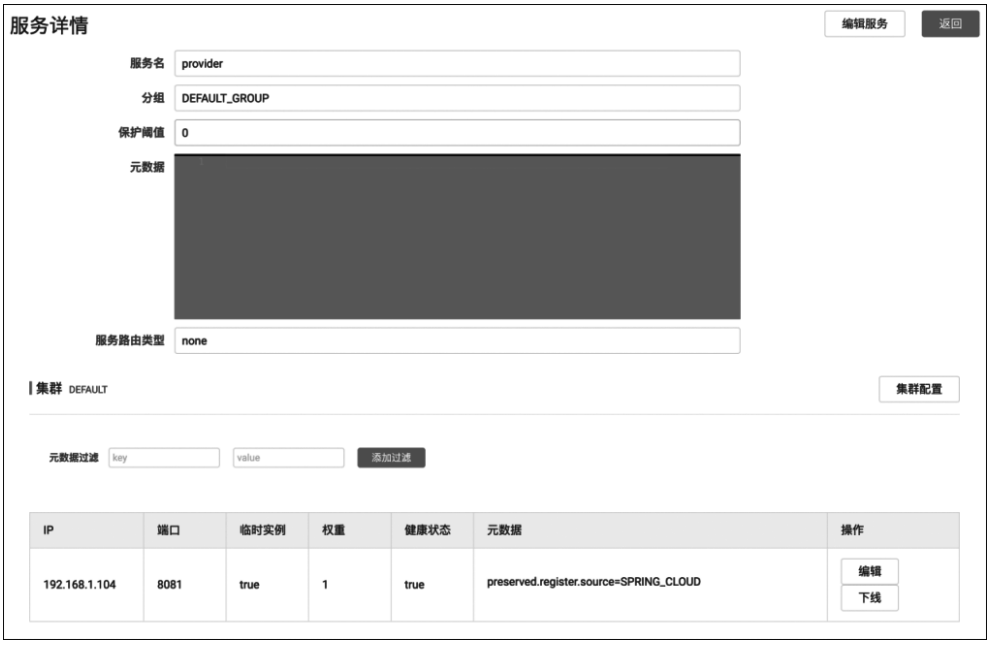

图 3-5　服务注册的详细情况

从图 3-5 中可以看出，提供者的服务已经通过注册动作把服务名、IP、端口注册到注册中心了。

3.3.2 服务的发现


Nacos 服务的发现

在微服务架构中，发现服务指客户端从注册中心获取服务实例的信息，以便能访问该服务。下面以一个简单的示例说明发现服务的过程。

假设现在服务消费者需要访问服务提供者的 addService。服务消费者需要从注册中心获取可用的服务实例列表，然后选择其中一个服务实例进行访问。

这个示例同样使用 Nacos 作为注册中心。首先，需要在服务消费者的项目中添加 Nacos 客户端依赖，然后，在服务消费者的启动类中添加以下代码以获取服务实例列表。

1. 修改项目的 pom.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.7</version>
    <relativePath/><!--lookup parent from repository -->
  </parent>
  <groupId>com.etoak.tutorial.nacos</groupId>
  <artifactId>consumer</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>client</name>
  <description>Nacos 注册和发现示例中的消费者</description>

  <properties>
    <java.version>1.8</java.version>
    <spring-cloud-alibaba.version>2021.1</spring-cloud-alibaba.version>
    <spring-cloud.version>2021.0.3</spring-cloud.version>
  </properties>

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

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

    <dependencyManagement>
      <dependencies>
        <dependency>
          <groupId>org.springframework.cloud</groupId>
          <artifactId>spring-cloud-dependencies</artifactId>
          <version>${spring-cloud.version}</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
        <dependency>
          <groupId>com.alibaba.cloud</groupId>
          <artifactId>spring-cloud-alibaba-dependencies</artifactId>
          <version>${spring-cloud-alibaba.version}</version>
          <type>pom</type>
          <scope>import</scope>
        </dependency>
      </dependencies>
    </dependencyManagement>
</project>
```

2. 修改项目的 application.yml 文件

```
server:
    port: 8080

  spring:
    application:
      name: consumer
      #Nacos 注册和发现的配置
  cloud:
    nacos:
      discovery:
        #Nacos-Server 的地址
        server-addr: 127.0.0.1:8848
        #默认的命名空间
        namespace: public
        username: nacos
        password: nacos
```

3. 修改项目启动类 App.java

```
package com.etoak.tutorial.nacos;

  import java.util.List;
  import org.springframework.boot.SpringApplication;
  import org.springframework.boot.autoconfigure.SpringBootApplication;
  import org.springframework.cloud.client.ServiceInstance;
  import org.springframework.cloud.client.discovery.DiscoveryClient;
  import org.springframework.context.ConfigurableApplicationContext;

  @SpringBootApplication
  public class App {
    public static void main(String[] args) {
       ConfigurableApplicationContext context = SpringApplication.run(App.
       class, args);
      //提供服务提供者的名称
      //与提供者项目 application.yml 中 spring.application.name 是对应的
      String serviceName ="provider";
      //通过 Spring 获取用于发现(查询)服务的客户端对象
       DiscoveryClient discoveryClient = context.getBean(DiscoveryClient.
       class);
      //通过服务名从注册中心获取真实的提供服务的实例
      //这里将返回一个列表,因为大多数场景是集群部署
      List<ServiceInstance> serviceInstances =discoveryClient.getInstances
      (serviceName);
      if (!serviceInstances.isEmpty()) {
        //遍历获取到的所有提供服务的实例对象
        //这里输出核心要素,即 IP 地址和端口号
        for (ServiceInstance serviceInstance : serviceInstances) {
          String serviceUrl ="http://" +serviceInstance.getHost() +
          ":" +serviceInstance.getPort();
          System.out.println(serviceUrl);
        }
      }else{
        //通过 serviceName 未能从注册中心查询到结果
        System.out.println("没有找到[" +serviceName +"]对应的服务");
      }
    }
  }
```

通过注册中心获取具体提供服务的实例地址后,就可以使用 httpclient 客户端调用远程服务。这里使用 RestTemplate 示意,代码如下所示。

```
ServiceInstance serviceInstance =serviceInstances.get(0);
String serviceUrl ="http://" +serviceInstance.getHost() +":" +
serviceInstance.getPort();
int a =20;
int b =23;
String url ="http://" +serviceInstance.getHost() +":" +
serviceInstance.getPort() +"/add?a=" +a +"&b=" +b;
RestTemplate restTemplate =new RestTemplate();
// 调用服务提供者提供的 add()方法
String response =restTemplate.getForObject(url, String.class);
System.out.println("远程服务响应结果: " +response);
```

上面这段代码调用成功后，会显示如下内容。

```
远程服务响应结果: The result is 43 from remote service [provider]
```

注意：为了快速演示，以上示例的代码直接被写到了 App.java 启动类中，在实际项目中要结合具体业务场景对代码位置进行调整。

3.3.3 订阅服务

订阅服务是指客户端在注册中心获取服务的变化情况，以便及时感知服务实例的变化。客户端可以通过订阅机制获取注册中心服务实例的变化信息，如服务实例的上线、下线、状态变化等，从而保证客户端及时调整访问服务的策略和方式。订阅机制还可以支持服务的版本管理和灰度发布等高级功能。

灰度发布(gray release)：软件开发中的一种部署策略，也称渐进式发布(progressive release)或金丝雀发布(canary release)。它是一种控制新功能或更新的方式，可以降低潜在风险并获得更好的用户反馈。在灰度发布中，新的软件版本或功能被逐步引入生产环境中的一小部分用户中，而不是立即对所有用户进行全面推广。这样可以让开发团队逐步检查新功能的稳定性、性能和用户体验，同时减少潜在问题对整个用户群体的影响。

为了避免客户端主动轮询服务实例列表，服务注册中心通常会支持服务的订阅功能。当服务实例列表发生变化时，注册中心会主动通知客户端，客户端可以及时更新服务实例列表，从而实现服务的高可用和负载均衡。下面以 Nacos 为例说明服务的订阅过程。

在 Nacos 中，服务的订阅和发布是通过命名空间和分组实现的。命名空间是实现逻辑隔离的单位，可以将不同的业务或应用程序隔离开；而分组则可以将同一应用程序的服务划分到不同的组中。

下面通过示例介绍订阅服务的过程。

1. 客户端发起订阅

在 App.java 中加入如下代码。

```
@PostConstruct
public void subscribeDemo() throws NacosException {
    // 参数 nacosDiscoveryProperties.getNacosProperties() 方法可以获取
    // YAML 文件的注册和发现的配置
    NamingService namingService =NamingFactory.createNamingService
    (nacosDiscoveryProperties.getNacosProperties());
    // provider 是指要订阅的微服务名
    namingService.subscribe("provider", new EventListener() {
      @Override
      public void onEvent(Event event) {
        //输出服务名
        System.out.println("serviceName:" +((NamingEvent)event).getServiceName());
        //这个服务名对应的所有实例
          System.out.println("instances:" +((NamingEvent)event).getInstances());
        }
    });
}
```

上述代码可以通过@PostConstruct 注解简化演示的过程。这个注解将在 Spring 初始化完成后执行。

订阅方应有一个需要两个参数的方法和一个需要三个参数的方法，示例使用的是两个参数的方法，如下所示。

```
void subscribe(String serviceName, EventListener listener) throws NacosException;

void subscribe (String serviceName, String groupName, EventListener listener)
throws NacosException;
```

这里的 serviceName 是要订阅的服务名，groupName 是要订阅的服务所在的分组，默认是 DEFAULT_GROUP，listener 则为订阅时注入的事件监听，程序通过其回调方法 onEvent(Event event)可及时获取被订阅服务的变更内容。

启动成功后，日志中已经有相应的事件输出了，如下所示。

```
serviceName:DEFAULT_GROUP@@provider
instances:[Instance{instanceId='192.168.1.104#8081#DEFAULT#DEFAULT_GROUP
@@provider', ip='192.168.1.104', port=8081, weight=1.0, healthy=true,
enabled=true, ephemeral=true, clusterName='DEFAULT', serviceName='DEFAULT_
GROUP@@provider', metadata={preserved.register.source=SPRING_CLOUD}}]
```

2. 触发订阅监听的执行

在 Nacos 管理页面/服务管理/服务列表中找到名为 provider 的服务，单击后面的“详情”按钮进入 provider 服务的详情页面，如图 3-6 所示。

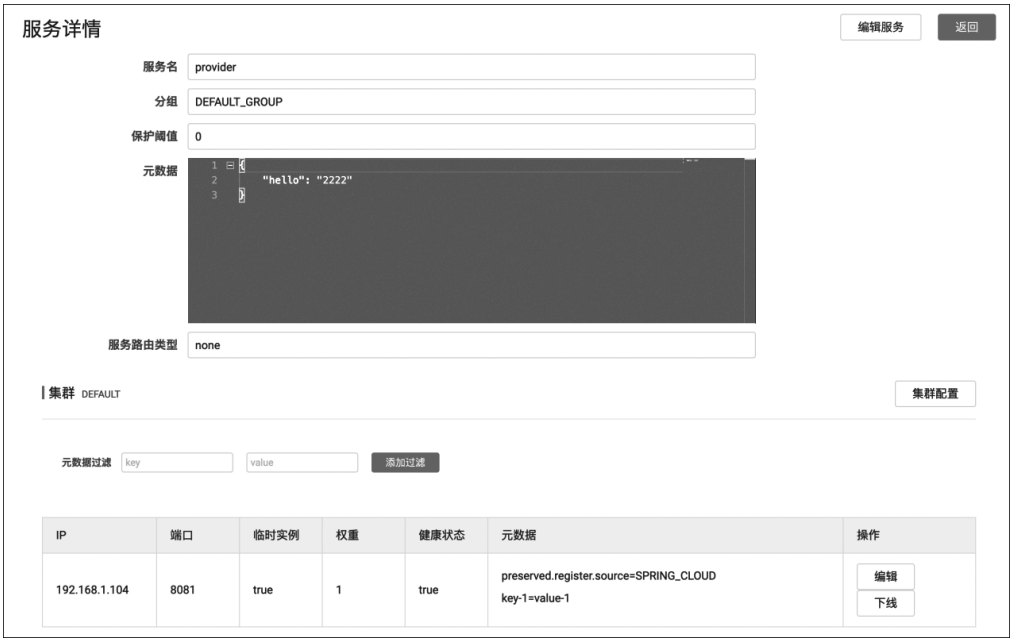

图 3-6　provider 服务的详情页面

单击“下线”按钮后，通过日志输出可以看到事件监听代码已经被执行，日志如下。

```
serviceName:DEFAULT_GROUP@@provider
instances:[]
```

通过日志可以看出，由于之前一共启动了一个服务实例，所以 instances 集合为空。再单击“上线”按钮，再次观察日志，日志如下。

```
serviceName:DEFAULT_GROUP@@provider
instances:[Instance{instanceId='192.168.1.104#8081#DEFAULT#DEFAULT_GROUP@@provider', ip='192.168.1.104', port=8081, weight=1.0, healthy=true, enabled=true, ephemeral=true, clusterName='DEFAULT', serviceName='DEFAULT_GROUP@@provider', metadata={preserved.register.source=SPRING_CLOUD}}]
```

可以发现，通过上线操作，监听代码已经输出了刚上线的实例。

再通过服务实例看监听程序是否能接收刚才修改的数据，在图 3-6 中单击“上线”按钮、“下线”按钮上方的“编辑”按钮，打开的服务实例编辑页面如图 3-7 所示。

如图 3-7 所示，修改一下元数据，添加一对 key、value 值如下。

图 3-7 provider 服务其中一个实例

```
"key-1": "value-1"
```

单击“确定”按钮后保存，日志输出如下。

```
instances:[Instance{instanceId='192.168.1.104#8081#DEFAULT#DEFAULT_GROUP
@@provider', ip='192.168.1.104', port=8081, weight=1.0, healthy=true,
enabled=true, ephemeral=true, clusterName='DEFAULT', serviceName='DEFAULT_
GROUP@@provider', metadata={preserved.register.source=SPRING_CLOUD, key-1
=value-1}}]
```

从日志中可以看出，事件监听已经获取到了新的元数据的值。

3.4 服务的负载均衡

3.4.1 负载均衡的原理

微服务架构通常会有多个服务提供者提供同一种服务，例如，某个服务的接口实现会被部署到多个服务提供者上，这些服务提供者可能被部署在不同的物理机器上，而服务消费者需要从这些服务提供者中选择一个进行访问。负载均衡就是解决这个问题的一种技术手段。

负载均衡的原理是将服务请求分摊到多个服务提供者上，从而避免单一的服务提供者压力过大，提高系统的可用性和吞吐量。当一个服务消费者需要访问某个服务时，负载均衡器会根据负载均衡算法选择一个服务提供者，并将请求转发给该服务提

供者。

负载均衡的实现方式有很多种，其中比较常用的是基于软件的负载均衡和基于硬件的负载均衡。基于软件的负载均衡主要是通过在客户端和服务端之间增加一个负载均衡器实现的。负载均衡器可以根据一定的算法将请求分发到多个服务器上，从而实现负载均衡。

具体来说，基于软件的负载均衡包括以下几个步骤。

（1）收集服务提供者信息。负载均衡器首先需要获取服务提供者的信息，包括服务地址、端口、权重等。

（2）根据策略选择。根据负载均衡策略选择一个合适的服务提供者，常见的负载均衡策略有轮询、随机、最小连接数等。

（3）转发请求。将请求转发给选中的服务提供者。在转发时，软件需要考虑服务提供者的可用性、处理请求的时间等因素，以便更好地实现负载均衡。

（4）检查健康。定期检查服务提供者的健康状态，将不可用的服务提供者从负载均衡器的可用列表中剔除。

基于硬件的负载均衡是通过专门的硬件设备实现负载均衡的。这种方式的优点是效率高、性能稳定、处理能力强，特别是在高负载的情况下，基于硬件的负载均衡能够提供更好的性能和可靠性。常见的基于硬件的负载均衡设备包括 F5、Cisco 等，它们具有高稳定性、高性能，但缺乏灵活的扩展性，资金成本也比较高。

目前微服务架构一般采用的是基于软件的负载均衡。

3.4.2 负载均衡的算法

负载均衡的算法指负载均衡器在选择服务提供者时所采用的算法，常见的负载均衡算法有以下几种。

（1）随机算法：随机选择一个服务提供者。

（2）轮询算法：轮流选择每个服务提供者，循环往复。

（3）最少连接数算法：将请求分配给连接数最少的服务器。

（4）最小连接数算法：选择当前连接数最小的服务提供者。

（5）最少活跃数算法：选择处理请求最少的服务提供者。

（6）带权重的随机算法：根据服务提供者的权重随机选择一个服务提供者。

（7）带权重的轮询算法：根据服务提供者的权重轮流选择服务提供者，按照权重分配请求。

（8）IP 哈希算法：根据服务消费者的 IP 地址选择服务提供者，从而保证同一客户端的请求始终被转发到同一台服务提供者。

（9）带权重的最少活跃数算法：根据服务提供者的权重选择处理请求最少的服务提供者。

3.5 在 Nacos 中如何实现负载均衡

3.5.1 Nacos 的负载均衡机制概述

作为一个服务发现和配置管理平台，Nacos 的负载均衡机制目的是将客户端请求分发到多个服务实例上，从而提高系统的可用性和稳定性。

Nacos 支持多种负载均衡策略，如随机、轮询、最小连接数等。通过对请求进行负载均衡，Nacos 可以将请求平均地分发到多个服务实例上，以提高服务的可用性和性能。Nacos 提供了两种负载均衡方式：一种是服务端负载均衡，即服务提供者在处理请求时进行负载均衡；另一种是客户端负载均衡，即服务消费者在发起请求前进行负载均衡。通常的负载均衡是指后者，本章提到的负载均衡也是客户端负载均衡。

服务端负载均衡是通过 Cluster 组件实现的。Cluster 组件会管理同一集群内的所有实例，并通过各种算法选择其中一台实例作为服务的提供者。下面是实现服务端负载均衡的具体步骤。

（1）注册集群：将同一集群内的实例注册到同一个 Cluster 下面。

（2）同步集群：各个结点将通过心跳同步集群信息，包括当前集群下的实例列表、结点健康状况等。

（3）选择实例：根据指定的负载均衡算法，在集群内选择一台健康的实例作为服务提供者，将请求转发给该实例处理。

需要注意的是，在服务端负载均衡模式下，请求不会由客户端进行负载均衡选择，而是在服务端选择处理请求的实例，这样能够减少客户端的负担，并提高整体的请求处理效率。

Nacos 客户端负载均衡主要是由 Spring Cloud LoadBalancer 实现的。Spring Cloud LoadBalancer 是一个 Spring Cloud 子项目，它提供了一组负载均衡的抽象，不依赖任何具体的负载均衡实现，因此可以轻松地在不同的负载均衡算法之间切换，如 Random、Round Robin、Weighted Response Time 等。

在 Nacos 中，Spring Cloud LoadBalancer 与 Nacos Discovery 的集成实现了客户端的负载均衡。它通过监听 Nacos Server 中注册的服务实例信息获取服务实例的列表，并根据特定的负载均衡策略选择服务实例，从而实现客户端负载均衡。

3.5.2 基于 Spring Cloud LoadBalancer 实现的 Nacos 负载均衡

在 Spring Cloud 早期版本中，Ribbon 是负载均衡的核心组件，它是一个客户端负载均衡器，可以作为 HTTP 和 TCP 客户端的负载均衡器，主要功能是提供负载均衡算法和对服务实例列表的管理。

但是，从 Spring Cloud 2020.0.0 版本开始，Spring Cloud 官方推荐使用 Spring Cloud LoadBalancer 作为负载均衡器。Spring Cloud LoadBalancer 是一个基于 Ribbon 的轻量

级负载均衡器，它提供了一个可扩展的架构，允许开发者定义自己的负载均衡策略。

Spring Cloud LoadBalancer 的目标是提供一个更加灵活、简单、可扩展的负载均衡器，以取代 Ribbon。在未来的版本中，Ribbon 将会被 Spring Cloud LoadBalancer 替代，并且 Spring Cloud 社区也计划将 Ribbon 转移到 Spring Cloud 外。

需要注意的是，虽然取代了 Ribbon，但是 Spring Cloud LoadBalancer 仍然支持 Ribbon 的所有负载均衡算法，并且提供了一些新的负载均衡算法。此外，Spring Cloud LoadBalancer 还提供了一个更加简单、灵活的 API，可以让开发者更加方便地自定义负载均衡策略。

接下来重点说一下 Spring Cloud Load，本章使用的 Nacos 版本是 2021 版本，已经没有自带 Ribbon 的整合，需要引入另一个支持的 jar 包——LoadBalancer。

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

接下来配置负载均衡器，如下所示。

```
package com.etoak.tutorial.nacos;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

  @Configuration
  /* 指定自定义配置文件，配置 LoadBalancer 的负载均衡算法。如果未配置，那么默认就
是 RoundRobinLoadBalancer 轮询策略 */
  public class RestTemplateConfig {
    @Bean
    @LoadBalanced              //添加负载均衡支持
    public RestTemplate restTemplate() {
      System.out.println("初始化 restTemplate");
      return new RestTemplate();
  }
}
```

上面的配置可以实现基本的负载均衡功能，负载均衡的默认配置为轮询配置。Spring 5 之后，上面的代码还可以以如下方式书写。

```
@Bean
@LoadBalanced
```

```
public WebClient.Builder loadBalancedWebClientBuilder() {
    return WebClient.builder();
}
```

WebClient 是在 Spring 5 中最新引入的，读者可以将其理解为 reactive 版的 RestTemplate。其支持响应式编程方式，支持非阻塞特性，这部分内容非本章重点内容，读者可自行学习这方面的知识。

LoadBalancer 只提供了两种负载均衡器，默认用的是轮询方式。

（1）RandomLoadBalancer 随机策略。

（2）RoundRobinLoadBalancer 轮询策略(默认)。

为了体现有负载均衡和效果，需要增加 provider 服务实例的个数，具体可以通过下面命令行实现。首先，启动三个 provider，其中 provider-0.0.1-SNAPSHOT.jar 是通过 mvn package 命令打的 Spring Boot 可运行的 jar 包。

```
java -Dserver.port=8081 -jar provider-0.0.1-SNAPSHOT.jar
java -Dserver.port=8082 -jar provider-0.0.1-SNAPSHOT.jar
java -Dserver.port=8083 -jar provider-0.0.1-SNAPSHOT.jar
```

项目中三个不同实例可能在不同机器上，如果不具备多机条件或简化测试环境，那么可以在本机通过变换端口号的方式启动三个实例。

然后，通过 Nacos 的服务管理页面的服务列表验证 provider 服务是否启动了 3 个实例，如图 3-8 所示。

图 3-8 服务列表

为了在调用 add 服务时能体现来自不同的服务实例的效果，须改变 add()方法的代码，如下所示。

```
package com.etoak.tutorial.nacos;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
```

```
import org.springframework.web.bind.annotation.RestController;

  @RestController
  public class AddController {
    @Autowired
    private Environment env;
    @RequestMapping("/add")
    public String add(@RequestParam int a, @RequestParam int b) {
      int result =a +b;
      //本地端口号
      String port =env.getProperty("server.port");
      return "The result is " +result +" from remote service [provider], local
      port:" +port;
   }
}
```

上面代码通过“env.getProperty("server.port");”语句获取本服务实例启动时所使用的端口号，在客户端发起调用返回时通过输出响应信息就能得知这个调用是来自哪个服务实例。

接下来看客户端使用负载均衡。

```
ConfigurableApplicationContext context = SpringApplication.run(App.class,
args);
RestTemplate restTemplate =context.getBean(RestTemplate.class);
//调用 20 次,通过观察返回结果看负载均衡策略的效果
for(int i =0; i <21; i++) {
     ResponseEntity < String > resp  = restTemplate. getForEntity ( " http://
     provider/add?a=10&b=20", String.class);
    if(resp.getStatusCode().is2xxSuccessful()) {
      System.out.println(resp.getStatusCode().value() +" :: " +resp.getBody());
    } else {
      System.out.println(resp.getStatusCode().value() +" :: " +resp.getBody());
    }
}
```

运行效果如下。

```
200 :: The result is 30 from remote service [provider], local port:8083
200 :: The result is 30 from remote service [provider], local port:8081
200 :: The result is 30 from remote service [provider], local port:8082
200 :: The result is 30 from remote service [provider], local port:8083
200 :: The result is 30 from remote service [provider], local port:8081
```