第 2 章　范例：helloapp 项目

阿云：“我对微服务的概念和原理已经略有所知。俗话说，百闻不如一见。用 Java 语言实现的微服务到底长什么样呢？”

答主：“本章会创建一个简单的 helloapp 项目，隆重邀请微服务登台亮相。”

helloapp 项目包括两个模块：hello-provider 模块和 hello-consumer 模块。1.1.3 小节已经介绍了微服务的提供者和消费者的关系是相对的。本范例为了便于清晰地演示提供者和消费者的通信过程，固定了这两个模块的角色，由 hello-provider 充当提供者，hello-consumer 充当消费者。

如图 2.1 所示，hello-provider 提供打招呼的微服务，hello-consumer 向 hello-provider 发送一个用户名，hello-provider 就会返回字符串“Hello, 用户名”。

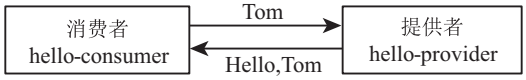

图 2.1　hello-consumer 访问 hello-provider 提供的打招呼微服务


扫一扫，看视频

2.1　提供者和消费者的通信及实现原理

阿云：“在分布式的运行环境中，各台主机上运行着各自的微服务。hello-consumer 如何在微服务的网络世界里找到 hello-provider 呢？”

答主：“在现实生活中，用户会通过 114 查询系统来查找一个机构的地址，获得了地址后，就能顺利地去访问特定机构了。114 查询系统之所以无所不知，是因为这些机构预先向 114 查询系统做了注册。Nacos 服务器就像 114 查询系统，hello-consumer 通过 Nacos 服务器获得 hello-provider 的网络地址。”

如图 2.2 所示，hello-provider 在启动时，会向 Nacos 服务器自报家门，注册自身。hello-consumer 通过 Nacos 服务器获得 hello-provider 的地址信息，然后访问 hello-provider。

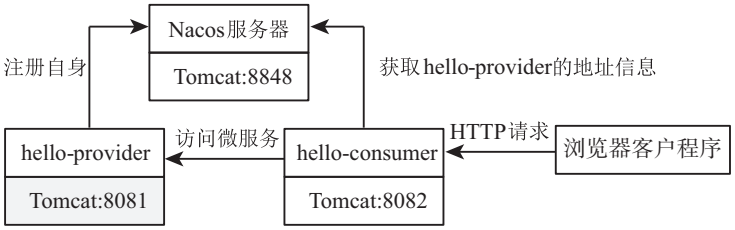

图 2.2　Nacos 服务器是微服务的注册中心

在图 2.2 中，Nacos 服务器、hello-provider 和 hello-consumer 都有相同的“坐骑”：Tomcat。由于它们的“坐骑”都是 Tomcat，就能方便地通过 HTTP 协议进行通信。

阿云：“Nacos 服务器的 Tomcat 是内置的，那么 hello-provider 和 hello-consumer 的 Tomcat 从何而来呢？”

答主：“在通过 IDEA 创建 hello-provider 和 hello-consumer 模块时，只要为它们声明了 Spring Web 依赖，Spring Boot 就会为这两个模块配备 Tomcat。”

把 Tomcat 看作微服务的“坐骑”，是形象的比喻。实际上，Tomcat 充当了 hello-provider 和 hello-consumer 的容器，hello-provider 和 hello-consumer 都是运行在 Tomcat 中的 Spring Web 应用。因此，浏览器客户程序也能访问 hello-provider 和 hello-consumer 中的 Controller 控制器组件。

阿云：“既然 hello-provider 和 hello-consumer 只是普通的 Spring Web 应用，它们又是如何摇身一变，成了微服务的提供者和消费者呢？”

答主：“借助 Spring Boot 的强大整合能力，只要给 hello-provider 和 hello-consumer 增加一些与微服务相关的配置、依赖类库和注解，它们就变成了微服务的提供者和消费者。”

2.2　在 IDEA 中创建 helloapp 项目

扫一扫，看视频

在 IDEA 中选择菜单 File → New → Project → Spring Initializr，创建基于 Spring Boot 的 helloapp 项目，在配置窗口中设置项目的名字（Name）、根路径（Location）、包的名字（Package name）、JDK 的版本（JDK）等信息，如图 2.3 所示。

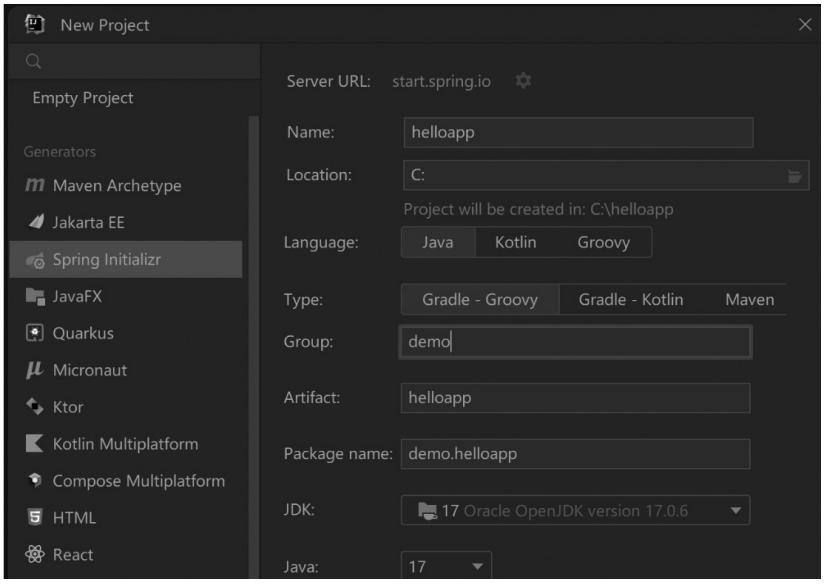

图 2.3　设置 helloapp 项目的基本信息


扫一扫，看视频

2.3 创建 hello-provider 模块

hello-provider 模块会提供打招呼的微服务，该模块主要包括以下内容。

（1）Maven 的配置文件 pom.xml：在该文件中配置与 Spring Cloud Alibaba 相关的依赖。

（2）HelloProviderApplication 类：hello-provider 模块的启动类，由 Spring Boot 自动创建。

（3）HelloProviderController 类：一个控制器类，提供打招呼的微服务。

（4）application.properties 配置文件：设置模块自身的 Tomcat 服务器以及 Nacos 服务器的信息。

2.3.1 在 IDEA 中创建 hello-provider 模块

在 IDEA 中创建 hello-provider 模块的步骤如下。

（1）在 IDEA 中选择菜单 File → New → Module → Spring Initializr，为 helloapp 项目增加一个 hello-provider 模块，在配置窗口中设置模块的名字（Name）、根路径（Location）、包的名字（Package name）、JDK 的版本（JDK）等信息，如图 2.4 所示。

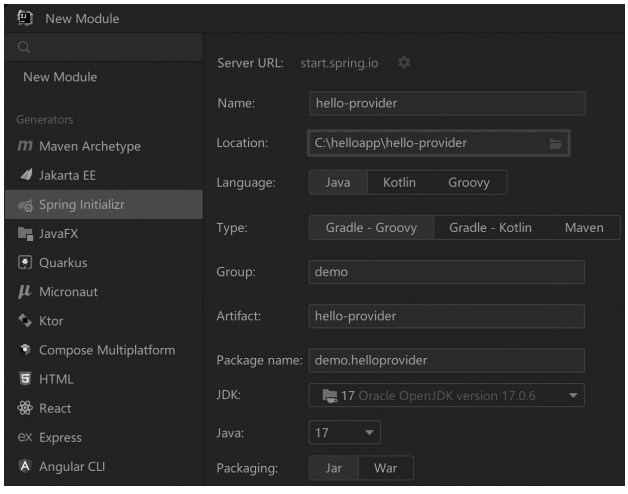

图 2.4 配置 hello-provider 模块

（2）在依赖配置窗口中添加 Spring Web 和 Cloud Bootstrap 依赖，如图 2.5 所示。

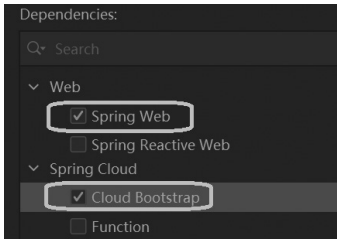

图 2.5 添加 Spring Web 和 Cloud Bootstrap 依赖

添加了 Spring Web 和 Cloud Bootstrap 依赖后，IDEA 会自动在 hello-provider 模块的 Maven 配置文件 pom.xml 中加入以下内容。

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

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter</artifactId>
  <version>3.1.1</version>
</dependency>
```

在以上代码中，spring-boot-starter-web 启动器会为 hello-provider 模块自动装配 Spring Web 应用的开发和运行环境，引入 Spring Web 的类库，配备 Tomcat 容器；spring-cloud-starter 启动器会为 hello-provider 模块自动装配 Spring Cloud 框架的开发和运行环境，引入 Spring Cloud 框架的类库。

2.3.2　在 pom.xml 文件中添加 Spring Cloud Alibaba 依赖

下面修改 hello-provider 模块的 Maven 配置文件 pom.xml，参见例程 2.1，粗体字部分是新增加的配置代码。

例程 2.1　hello-provider 模块的 pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project …>
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.8</version>
    <relativePath/>
  </parent>

  <groupId>demo</groupId>
  <artifactId>hello-provider</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>hello-provider</name>
  <description>hello-provider</description>

  <properties>
```

```
        <java.version>17</java.version>
        <spring-cloud.version>
          2022.0.0
        </spring-cloud.version>
        <spring-cloud-alibaba.version>
          2022.0.0-RC2
        </spring-cloud-alibaba.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</artifactId>
          <version>3.1.1</version>
        </dependency>

        <!-- Nacos Discovery 组件 -->
        <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-actuator
            </artifactId>
        </dependency>

        <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
        </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>
```

1.4 节介绍了各种软件的版本具有相应的匹配关系。在本范例中，JDK 的版本为 17，Spring Boot 的版本为 3.0.8，Spring Cloud 的版本为 2022.0.0，Spring Cloud Alibaba 的版本为 2022.0.0.0-RC。

在以上粗体字代码中，spring-cloud-alibaba-dependencies 的作用是为 hello-provider 模块引入 Spring Cloud Alibaba 框架的类库。

spring-cloud-starter-alibaba-nacos-discovery 启动器会自动装配 Nacos Discovery 组件，并为模块引入相关的依赖类库。Nacos Discovery 组件是 Nacos 服务器的客户端组件，微服务模块通过 Nacos Discovery 组件与 Nacos 服务器通信。

spring-boot-starter-actuator 启动器会自动装配 Spring Boot 的 Actuator 监控器。虽然 Actuator 监控器不属于 Spring Cloud Alibaba 框架，但是会负责监控微服务的特定端点（EndPoint）。

对于早期的 Spring Cloud Alibaba 框架，为了启用 Nacos Discovery 组件，还需要在 HelloProviderApplication 启动类中加入 @EnableDiscoveryClient 注解。

```
// 对于新版的 Spring Cloud Alibaba 框架，可以省略该注解
@EnableDiscoveryClient
```

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

对于新版的 Spring Cloud Alibaba 框架，不需要在启动类中显式地加入 @EnableDiscoveryClient 注解。因为 spring-cloud-starter-alibaba-nacos-discovery 启动器会自动启用 Nacos Discovery 组件。

Nacos Discovery 组件是 hello-provider 模块与 Nacos 服务器通信的桥梁，如图 2.6 所示。该组件的主要功能如下：

（1）在启动 hello-provider 模块时向 Nacos 服务器注册该模块。

（2）从 Nacos 服务器中订阅所有已经注册的微服务的列表信息。

图 2.6　Nacos Discovery 组件是 hello-provider 模块与 Nacos 服务器通信的桥梁

阿云：“pom.xml 文件中冗长的依赖配置代码靠死记硬背根本记不住，而且这些代码不是一成不变的，如果软件升级换代，它的配置代码也会发生更新。有什么窍门可以准确地编写配置代码吗？”

答主：“首先要理解依赖配置代码的作用；然后在各个软件的官方网站的使用文档中查看软件的依赖配置代码。”

例如，在 Spring 的官网（http://spring.io）中，选择菜单 Projects → Spring Cloud → Spring Cloud Alibaba → LEARN，就会显示 Spring Cloud Alibaba 框架的使用文档，如图 2.7 所示。在这些文档中提供了 Spring Cloud Alibaba 框架的依赖配置代码。

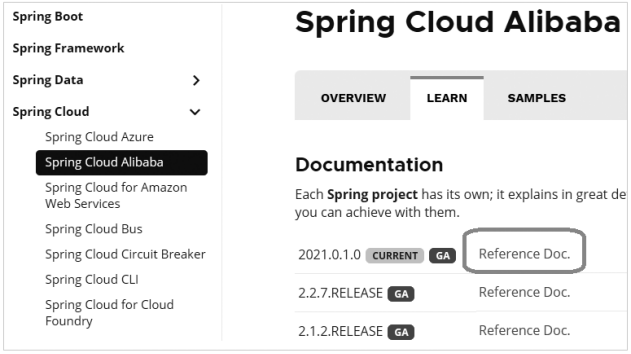

图 2.7　Spring Cloud Alibaba 框架的使用文档

阿云:“在 pom.xml 文件中设定了 hello-provider 模块所依赖的各个软件的版本，如何正确设置这些版本，确保软件互相兼容呢?”

答主:“1.4 节列出了各种软件之间的版本匹配。”

2.3.3　创建控制器类 HelloProviderController

HelloProviderController 类利用 Spring Web API 中的 @RestController 注解，声明自身是一个遵循 RESTFul 风格的控制器类，参见例程 2.2。

例程 2.2　HelloProviderController.java

```
@RestController
public class HelloProviderController {
  @GetMapping(value = "/greet/{username}")
  public String greet(@PathVariable String username) {
    return "Hello," + username;
  }
}
```

HelloProviderController 类只是一个普通的基于 Spring Web 的控制器组件，它的 greet() 方法具有打招呼的功能。如图 2.8 所示，用户通过浏览器就能直接访问 greet() 方法，访问该方法的 URL 为 http://localhost:8081/greet/Tom ，其中 Tom 是发送给 greet() 方法的 username 参数的值。

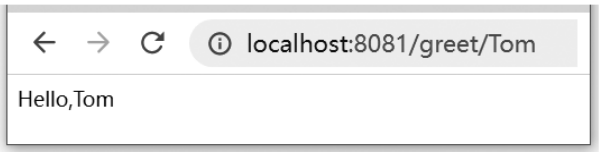

图 2.8　通过浏览器访问 greet() 方法

阿云:“HelloProviderController 类可以向浏览器返回 HTTP 响应，又如何为 hello-consumer 模块提供微服务呢?”

答主:“当 HelloProviderController 类置于 Spring Cloud Alibaba 框架中时，就变成了微服务的提供者。hello-consumer 模块无须借助浏览器，也能远程调用 HelloProviderController 类的 greet() 方法。”

在图 2.9 中，浏览器程序和 hello-consumer 模块都会访问 HelloProviderController 类的 greet() 方法。浏览器程序和 HelloProviderController 类之间进行的是最原始的 HTTP 通信，而 hello-consumer 模块通过 Spring Cloud Alibaba 框架远程调用 greet() 方法，使 greet() 方法成为提供微服务的方法。

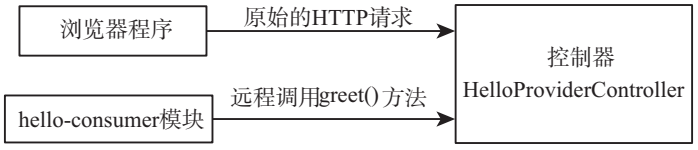

图 2.9　浏览器程序和 hello-consumer 模块以不同的方式访问 HelloProviderController

阿云：“在实际应用中，微服务都是由控制器类来实现的吗？”

答主：“确切地说，控制器类只是微服务的调用入口。在 Spring Web MVC（Model-View-Controller，模型－视图－控制器）框架中，微服务通常由模型层的业务逻辑组件来实现，而控制器类会调用业务逻辑组件的特定方法对外提供服务。”

2.3.4 在 application.properties 文件中配置微服务

阿云：“如何配置 hello-provider 模块所在的 Tomcat 容器以及所注册的 Nacos 服务器？”

答主：“需要在 application.properties 配置文件中设定 hello-provider 模块的 Tomcat 容器所监听的端口，以及 Nacos 服务器的网络地址。做好这些配置后，Spring Cloud Alibaba 框架就会负责 hello-provider 模块与 Nacos 服务器的通信。”

提示

Spring 框架还支持 YAML 格式的配置文件，它的配置代码比较简洁，这也是目前很流行的配置格式。本书后面章节中的一些范例会使用 YAML 格式的配置文件。

例程 2.3 是 application.properties 文件的配置代码。

例程 2.3 application.properties 文件

```
#hello-provider 模块的 Tomcat 容器所监听的端口
server.port=8081

#hello-provider 模块的应用名字，也是微服务的默认名字
spring.application.name=hello-provider-service

#Nacos 服务器的地址
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848

# 向 Spring Boot Actuator 监控器暴露所有的端点
management.endpoints.web.exposure.include=*
```

2.3.5 启动 hello-provider 模块

首先按照 1.5.3 小节中的步骤启动 Nacos 服务器，然后运行 HelloProviderApplication 启动类，就启动了 hello-provider 模块。hello-provider 模块在启动的过程中，会向 Nacos 服务器注册自身。

hello-provider 模块启动后，通过浏览器访问 Nacos 服务器的管理平台 http://localhost:8848/nacos，选择菜单“服务管理”→“服务列表”，显示注册成功的 hello-provider-service 微服务，如图 2.10 所示。

图 2.10　在 Nacos 服务器中注册了 hello-provider-service 微服务

在图 2.10 中，服务名 hello-provider-service 是由 application.properties 配置文件中的 spring.application.name 属性指定的。

提示

在 application.properties 配置文件中，spring.application.name 属性值会作为微服务的默认名字。此外，还可以通过 spring.cloud.nacos.discovery.service 属性显式地指定微服务的名字。

2.4　创建 hello-consumer 模块


扫一扫，看视频

hello-consumer 模块也是一个微服务模块，它会作为消费者访问 hello-provider 模块提供的微服务。如图 2.11 所示，由 Spring Cloud 框架提供的 OpenFeign 组件是 hello-consumer 模块与 hello-provider 模块之间通信的桥梁。OpenFeign 封装了与 hello-provider 模块之间的 RESTFul 风格的 HTTP 通信细节，使得 hello-consumer 模块可以按照 RPC（Remote Procedure Call，远程过程调用）模式访问 hello-provider 模块。

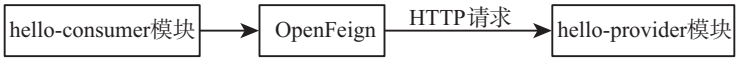

图 2.11　hello-consumer 通过 OpenFeign 组件远程访问 hello-provider 模块

hello-consumer 模块主要包括以下内容。

（1）Maven 配置文件 pom.xml：在该文件中配置与 Spring Cloud Alibaba 框架相关的依赖。

（2）HelloConsumerApplication 类：hello-consumer 模块的启动类，由 Spring Boot 自动创建，还需要手工添加 @EnableFeignClients 注解。

（3）HelloConsumerController 类：一个控制器类，会访问 hello-provider 模块的微服务。

（4）application.properties 配置文件：配置模块自身的 Tomcat 服务器以及 Nacos 服务器的信息。

2.4.1　在 IDEA 中创建 hello-consumer 模块

参考 2.3.1 小节和 2.3.2 小节中的步骤创建 hello-consumer 模块。在 IDEA 中创建 hello-

consumer 模块时，需要添加 Spring Web、Cloud Bootstrap、OpenFeign、Cloud LoadBalancer 依赖，如图 2.12 所示。

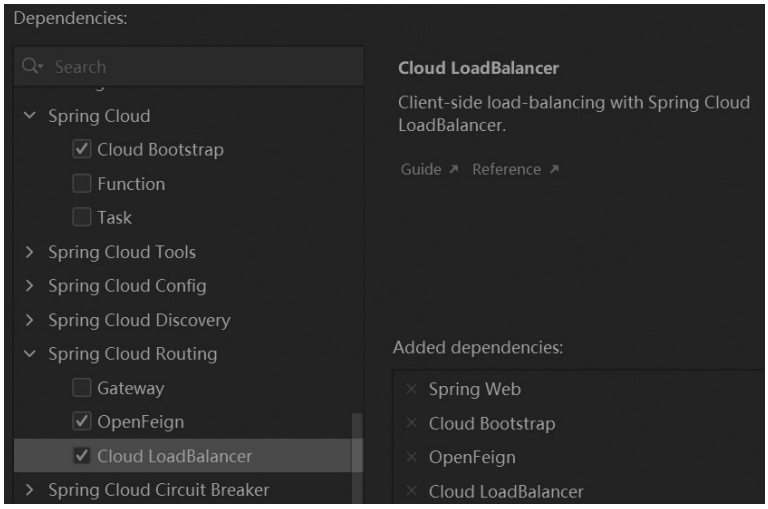

图 2.12　添加 hello-consumer 模块的依赖

添加了 OpenFeign 和 Cloud LoadBalancer 依赖后，IDEA 会自动在 pom.xml 文件中加入它们的依赖配置代码。

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

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

OpenFeign 依靠 Cloud LoadBalancer 管理访问微服务的负载均衡。早期版本的 Spring Cloud 通过 Ribbon 管理负载均衡，新的版本提倡使用 Cloud LoadBalancer。在 pom.xml 文件中，需要去除 Ribbon 依赖。

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
```

```
    <artifactId>
      spring-cloud-starter-alibaba-nacos-discovery
    </artifactId>

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

2.4.2　在启动类中加入 @EnableFeignClients 注解

@EnableFeignClients 注解的作用是告诉 Spring 框架，启动时扫描所有用 @FeignClient 注解标识的 FeignClient 组件，并把它们注册到 Spring 框架中。2.4.3 小节会介绍 FeignClient 组件的作用。

在例程 2.4 的 HelloConsumerApplication 启动类中使用了 @EnableFeignClients 注解。

例程 2.4　HelloConsumerApplication.java

```
@EnableFeignClients
@SpringBootApplication
public class HelloConsumerApplication {
  public static void main(String[] args) {…}
}
```

2.4.3　创建 HelloFeignService 接口

如图 2.13 所示，HelloConsumerController 类通过 HelloFeignService 接口来访问 hello-provider 模块的 HelloProviderController 类。开发人员只需创建 HelloFeignService 接口，而其具体的实现类由 OpenFeign 提供。

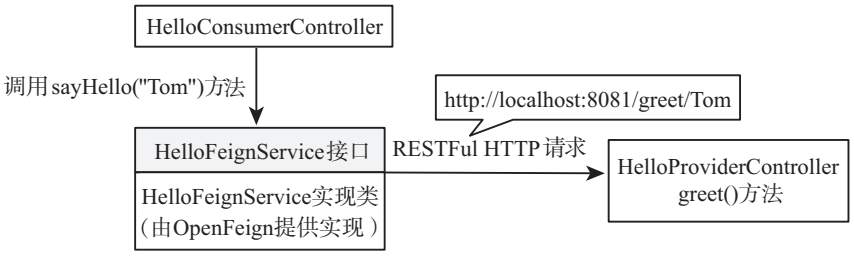

图 2.13　HelloConsumerController 类通过 HelloFeignService 接口访问 HelloProviderController 类

例程 2.5 是 HelloFeignService 接口的源代码，它通过 @FeignClient 注解把自己标识为 FeignClient 客户端接口。

例程 2.5　HelloFeignService.java

```
@FeignClient(value="hello-provider-service")
public interface HelloFeignService {
  @RequestMapping(method = RequestMethod.GET,
                  value = "/greet/{username}")
  String sayHello(
            @PathVariable("username")String username);
}
```

以上代码中的 @FeignClient 注解指定访问的微服务的名字是 hello-provider-service，HelloFeignService 接口的实现会根据该微服务名字找到它的网络地址（http://localhost:8081）。

提示

在 HelloFeignService 接口的实现中，实际上是通过 Nacos Discovery 组件从 Nacos 服务器中获得 hello-provider-service 微服务的地址信息的。

HelloFeignService 接口的 sayHello() 方法中有一个 @RequestMapping 注解，它会将 sayHello() 方法映射到 hello-provider-service 微服务的一个相对 URL，为 /greet/{username}。这个相对 URL 的完整路径为 http://localhost:8081/greet/{username}，该 URL 与 hello-provider 模块中的 HelloProviderController 类的 greet() 方法对应。

HelloFeignService 接口的 sayHello() 方法会请求访问 http://localhost:8081/greet/{username}，该请求对应 HelloProviderController 类的 greet() 方法。因此，hello-provider 模块所在的 Tomcat 容器会执行 greet() 方法，提供相应的远程服务。

2.4.4　创建控制器类 HelloConsumerController

例程 2.6 的 HelloConsumerController 类只需调用本地 HelloFeignService 接口的 sayHello() 方法，HelloFeignService 接口可以远程访问 hello-provider-service 微服务的打招呼服务。

例程 2.6　HelloConsumerController.java

```
@RestController
public class HelloConsumerController {
  @Autowired
  private HelloFeignService helloFeignService;

  @GetMapping(value = "/enter/{username}")
  public String sayHello(@PathVariable String username) {
    return helloFeignService.sayHello(username);
  }
}
```

2.4.5　在 application.properties 文件中配置微服务

在例程 2.7 的 application.properties 文件中，指定 hello-consumer 模块的 Tomcat 容器监听 8082 端口，而且也设置了 Nacos 服务器的地址。

例程 2.7　application.properties 文件

```
server.port=8082
spring.application.name=hello-consumer-service
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
management.endpoints.web.exposure.include=*
```

2.4.6　启动和访问 hello-consumer 模块

首先启动 Nacos 服务器和 hello-provider 模块，然后在 IDEA 中运行 HelloConsumerApplication 类，就会启动 hello-consumer 模块。通过浏览器访问 http://localhost:8082/enter/Tom，就会访问 HelloConsumerController 类，它返回的网页如图 2.14 所示。

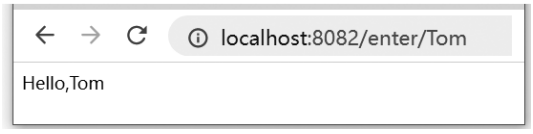

图 2.14　HelloConsumerController 类返回的网页

图 2.15 所示为当通过浏览器访问 HelloConsumerController 类时，hello-consumer 模块与 hello-provider 模块间的通信过程。

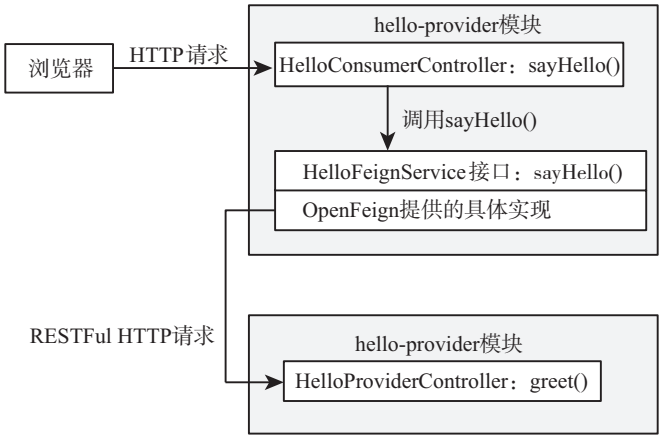

图 2.15　hello-consumer 模块与 hello-provider 模块间的通信过程

在图 2.15 中，通过浏览器发出的 HTTP 请求的 URL 为 http://localhost:8082/enter/Tom。HelloFeignService 接口发出的 HTTP 请求的 URL 为 http://localhost:8081/greet/Tom。

HelloFeignService 接口的 sayHello() 方法会发出符合 RESTFul 风格的 HTTP 请求，该请求由

HelloProviderController 类的 greet() 方法处理。

2.4.7 HelloFeignService 接口的默认方法

在 HelloFeignService 接口中，除了包含抽象方法，还可以包含默认方法。例如，在 HelloFeignService 接口中再添加以下不带参数的 sayHello() 方法。

```
default String sayHello(){           // 默认方法
    return sayHello("Stranger");     // 调用带参数的 sayHello()方法
}
```

在 HelloConsumerController 类中也加入一个不带参数的 sayHello() 方法。

```
@GetMapping(value = "/enter")
public String sayHello() {
    return helloFeignService.sayHello();
}
```

当用户通过浏览器访问 http://localhost:8082/enter 时，就会由 HelloConsumerController 类的不带参数的 sayHello() 方法处理用户请求。该方法会调用 HelloFeignService 接口的 sayHello() 默认方法，最后返回 Hello,Stranger。


扫一扫，看视频

2.5 启动微服务的多个实例

阿云：“假如 hello-provider-service 微服务只有一个实例在运行，却要同时为几十万名消费者提供服务，这是会力不从心的。”

答主：“消费者可以同时启动 hello-provider-service 微服务的多个实例，每个实例都运行在独立的进程中，这样就构成了微服务集群。负载均衡器为这些实例分配负载，共同为消费者提供服务。”

阿云：“这让我想到了大闹天宫的孙悟空，如果单打独斗会应接不暇，就拔一撮毫毛，变成无数小孙悟空，一起和天兵天将作战。”

如图 2.16 所示，hello-provider-service 微服务有两个实例，它们运行在各自的 Tomcat 容器中，分别监听 8081 和 8091 端口。消费者只需按照 hello-provider-service 微服务的名字访问服务，至于到底访问微服务的哪个实例，对消费者是透明的，由负载均衡器 LoadBalancer 来调度。

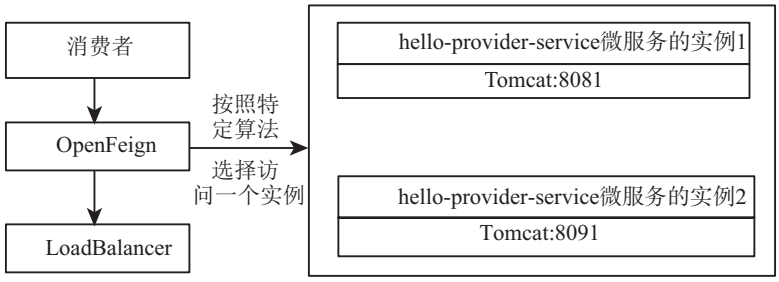

图 2.16　由 LoadBalancer 决定调用 hello-provider-service 微服务的哪个实例

在 IDEA 中启动 hello-provider-service 微服务的两个实例的步骤如下。

（1）在 IDEA 中选择菜单 Run → Edit Configurations，选择 HelloProviderApplication 类的启动配置，把启动配置的名字改为 HelloProviderApplication1，如图 2.17 所示。

图 2.17　修改 HelloProviderApplication 类的启动配置

（2）在图 2.17 所示的窗口中单击复制图标，为 HelloProviderApplication 类再创建一个启动配置，把启动配置的名字设为 HelloProviderApplication2，并且增加启动参数 -Dserver.port=8091，如图 2.18 所示。

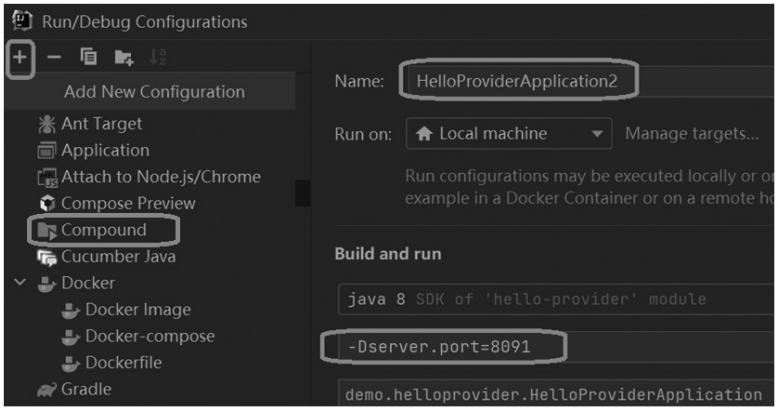

图 2.18　为 HelloProviderApplication 类再创建一个启动配置

图 2.18 中设置的启动参数 -Dserver.port=8091 将会覆盖 application.properties 配置文件中的 server.port 属性，使内置 Tomcat 监听 8091 端口。

（3）在图 2.18 中，单击“+”图标，在弹出的下拉菜单中选择菜单 Compound，创建一个批处理组合，用于按照两种启动配置分别运行 HelloProviderApplication 类。这个批处理组合的名字为 HelloProviderApplication，它包括 HelloProviderApplication1 和 HelloProviderApplication2 这两个启动配置，如图 2.19 所示。

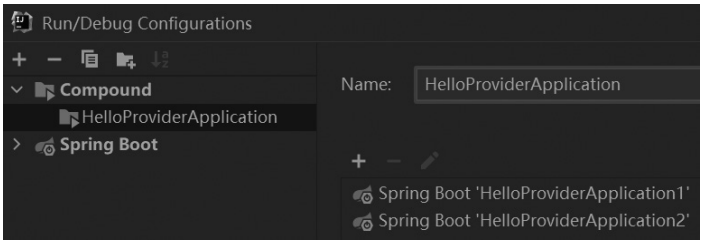

图 2.19　创建 HelloProviderApplication 批处理组合

（4）在 IDEA 中选择菜单 Run → Run HelloProviderApplication，就会运行 HelloProvider-Application 批处理组合，如图 2.20 所示。IDEA 会依据 HelloProviderApplication1 和 HelloProvider-Application2 这两个启动配置，启动 hello-provider-service 微服务的两个实例。

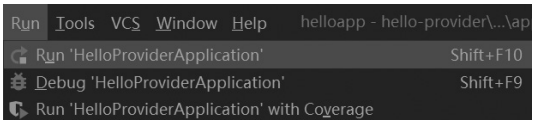

图 2.20　运行 HelloProviderApplication 批处理组合

通过浏览器访问 Nacos 服务器的管理平台 http://localhost:8848/nacos，在 Nacos 服务器管理平台中显示注册的 hello-provider-service 微服务有两个实例，如图 2.21 所示。

图 2.21　在 Nacos 服务器的管理平台中查看微服务实例

通过浏览器访问 hello-consumer 模块的 HelloConsumerController 控制器，URL 为 http://localhost:8082/enter/Tom，可以得到正常的响应结果。HelloConsumerController 远程访问 hello-provider-service 微服务时，无须考虑到底访问它的哪个实例，这是由 LoadBalancer 负载均衡器决定的，2.6 节还会对此做进一步的介绍。

提示

在本范例中，hello-provider-service 微服务的两个实例都运行在同一台主机上，监听不同的端口。在实际应用中，会把两个实例部署到不同的主机上，让它们各自获得更加充足的软件和硬件资源。

启动 hello-provider-service 微服务的两个实例的另一种方法是复制 hello-provider 模块的所有代码，再创建一个 hello-provider2 模块，把该模块的 application.properties 文件中的 server.port 属性设为 8091。分别运行 hello-provider 模块和 hello-provider2 模块的 HelloProviderApplication 类，就会启动微服务的两个实例。

2.6　LoadBalancer 负载均衡器

扫一扫，看视频

Spring Cloud 提供的 LoadBalancer 负载均衡器的官方文档的网址参见本书技术支持网页的【链接 6】。在 LoadBalancer API 中，ReactorLoadBalancer 接口表示客户端的负载均衡器，它有两个具体的实现类：

- RandomLoadBalancer：采用随机算法，从微服务列表中随机选择一个微服务实例。
- RoundRobinLoadBalancer：采用轮询算法，从微服务列表中轮流选择一个微服务实例。

ReactorLoadBalancer 接口的默认实现类为 RoundRobinLoadBalancer。为了演示 RoundRobinLoadBalancer 类的作用，对 HelloProviderController 类做一些修改，在它的 greet() 方法中输出 server.port 配置属性和 spring.application.name 配置属性参见例程 2.8。

例程 2.8　HelloProviderController.java

```
@RestController
public class HelloProviderController {
  // 把 servicePort 变量与 server.port 配置属性绑定
  @Value("${server.port}")
  private String servicePort;

  // 把 serviceName 变量与 spring.application.name 配置属性绑定
  @Value("${spring.application.name}")
  private String serviceName;

  @GetMapping(value = "/greet/{username}")
  public String greet(@PathVariable String username) {
    return "Hello," + username
      +"<br>Service Name:" + serviceName
      +"<br>Service Port:"+servicePort;
  }
}
```

按照 2.5 节中的步骤启动 hello-provider-service 微服务的两个实例，再启动 hello-consumer-service 微服务。通过浏览器访问 hello-consumer 模块的 HelloConsumerController 控制器，URL 为 http://localhost:8082/enter/Tom，得到如图 2.22 所示的网页。