

Table of Contents

| | |
|----------------------------|---------|
| Introduction | 1.1 |
| 1 微服务架构概述 | 1.2 |
| 1.1 单体应用架构存在的问题 | 1.2.1 |
| 1.2.如何解决单体应用架构存在的问题 | 1.2.2 |
| 1.3 什么是微服务 | 1.2.3 |
| 1.4 微服务架构的优点与挑战 | 1.2.4 |
| 1.5 微服务设计原则 | 1.2.5 |
| 1.6 如何实现微服务？ | 1.2.6 |
| 1.6.1 微服务技术选型有哪些 | 1.2.6.1 |
| 1.6.2 微服务架构图及常用组件 | 1.2.6.2 |
| 2 微服务开发框架——Spring Cloud | 1.3 |
| 2.1 Spring Cloud简介及其特点 | 1.3.1 |
| 2.2 Spring Cloud的版本简介 | 1.3.2 |
| 3 开始用Spring Cloud实战微服务 | 1.4 |
| 3.1 Spring Cloud实战前提 | 1.4.1 |
| 3.1.1 需要的技术储备 | 1.4.1.1 |
| 3.1.2 使用的工具及软件版本 | 1.4.1.2 |
| 3.2 创建存在调用关系的微服务 | 1.4.2 |
| 3.2.1 服务提供者与服务消费者简介 | 1.4.2.1 |
| 3.2.2 编写服务提供者 | 1.4.2.2 |
| 3.2.3 编写服务消费者 | 1.4.2.3 |
| 3.2.4 硬编码存在什么问题？ | 1.4.2.4 |
| 4 微服务注册与发现 | 1.5 |
| 4.1 服务注册与发现简介 | 1.5.1 |
| 4.2 Spring Cloud中提供的服务发现组件 | 1.5.2 |
| 4.3 Eureka简介 | 1.5.3 |
| 4.4 Eureka原理 | 1.5.4 |
| 4.5 实现一个Eureka Server | 1.5.5 |
| 4.5.1 Eureka Server代码示例 | 1.5.5.1 |
| 4.6 实现一个Eureka Client | 1.5.6 |
| 4.6.1 Eureka Client代码示例 | 1.5.6.1 |
| 4.6.2 目前存在的问题 | 1.5.6.2 |
| 4.6.3 客户端侧负载均衡——Ribbon | 1.5.6.3 |

| | |
|-------------------------------|-------------|
| 4.6.3.1 Ribbon代码示例 | 1.5.6.3.1 |
| 4.6.3.2自定义Ribbon配置 | 1.5.6.3.2 |
| 4.6.3.2.1 使用Java代码自定义Ribbon配置 | 1.5.6.3.2.1 |

序

// TODO

1 微服务架构概述

微服务架构是当前软件开发领域的一个技术热点，它在博客、社交媒体和各种会议演讲的出镜率非常之高，笔者相信大家听说过微服务这个名词。然而微服务似乎又是非常虚幻的——我们找不到微服务的完整定义，以至于很多人认为只不过是炒概念。

那么什么是微服务？微服务解决了哪些问题？微服务有哪些特点？诸多问题，本章将为您一一解答。同时，微服务理论性的内容，互联网上已经有很多了，所以本书不会过多涉及。我们尽量把篇幅花在微服务的实战上。

1.1 单体应用架构存在的问题

一个归档包（例如war格式）包含了应用所有功能的应用程序，我们通常称之为单体应用。架构单体应用的方法论，我们称之为单体应用架构。

我们以一个电影售票系统为例，它的架构图如图1-1所示。

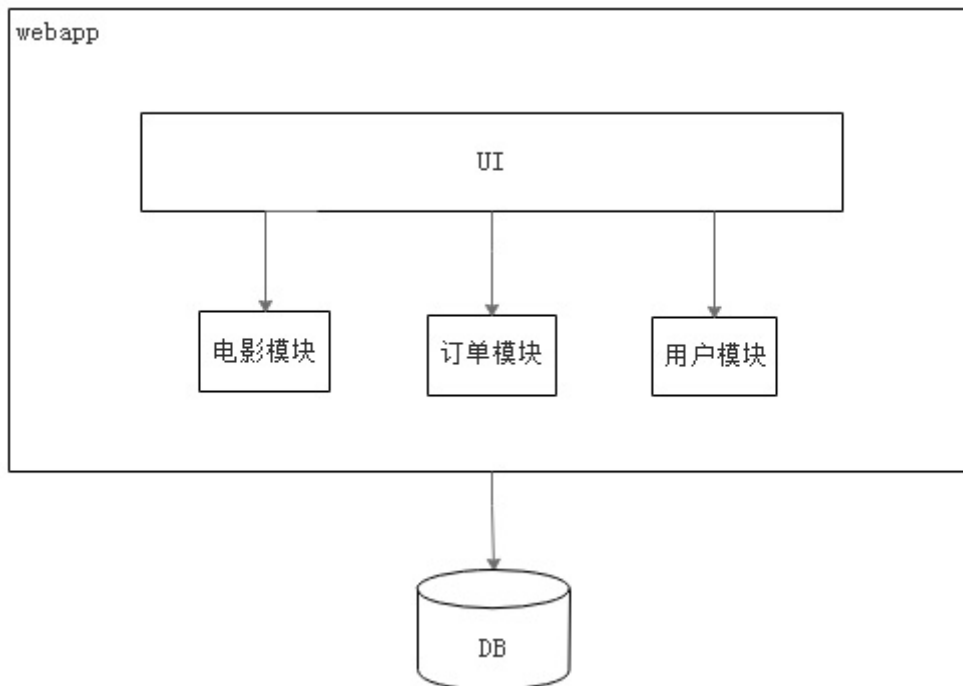


图1-1 电影售票系统单体架构示意图

由图我们可以看到，尽管该应用进行了模块化，但是UI和若干业务模块最终被打包在一个war包中，该war包包含了整个系统所有的功能，此时这个应用系统就可以被称为一个单体应用。

相信很多项目都是从单体应用开始的。单体应用比较容易部署、测试，在项目的初期，单体应用都可以很好地运行。然而，随着需求的不断增加，越来越多的人加入开发团队，代码库也在飞速地膨胀。慢慢地，单体应用变得越来越臃肿，可维护性、灵活性逐渐降低，维护成本越来越高。下面列举单体应用存在的一些问题：

- 技术债务

随着时间推移、需求变更和人员更迭，会逐渐形成应用程序的技术债务，并且越积越多。“不坏不修（Not broken, don't fix）”，这在软件开发中非常常见，在单体应用中这种思想更甚。已使用的系统设计或代码难以修改，因为应用程序的其他模块可能会以意料之外的方式使用它。

- 复杂性高

以笔者经手的一个百万行级别的单体应用为例，整个项目包含的模块非常多，模块的边界模糊，依赖关系不清晰，代码质量参差不齐，混乱地堆砌在一起……整个项目非常复杂。每次修改代码都心惊胆战，甚至添加一个简单的功能，或者修改一个BUG都会造成隐含的缺陷。

- 部署频率低

随着代码的增多，构建和部署的时间也会增加。每次功能的变更或者缺陷的修复都会导致我们需要重新部署整个应用。这种部署方式耗时长、影响的范围大、风险高，这使得单体应用项目上线部署的频率较低。而部署频率低又导致两次发布之间会有大量的功能变更和缺陷修复，出错概率比较高。

- 扩展能力受限

单体应用只能作为一个整体进行扩展，无法结合业务模块的特点进行伸缩。例如，应用中有的模块是计算密集型的，它需要强劲的CPU；有的模块是IO密集型的，需要更大的内存。由于这些模块部署在一起，我们不得不在硬件的选择上做出妥协。

- 阻碍技术创新

单体应用往往使用统一的技术平台或方案解决所有问题，每个团队成员都必须使用相同的开发语言和框架。然而，技术是不断在发展的，近几年也不断有开发效率更高、性能更好的新技术出现。如果想要尝试引入新的框架或技术会非常困难。例如：假设一个100万行代码的单体应用，原本是使用Struts 2构建的，如果现在想要使用Spring MVC，切换成本无疑是非常高的。

综上，随着业务需求的发展，功能的不断增加，单体架构很难满足互联网时代业务快速变化的需要。

如何解决单体架构存在的问题呢？

1.2.如何解决单体应用架构存在的问题

综上所述，单体应用架构存在很多的问题。有没有一种架构模式可以帮助我们解决这些问题呢？微服务就是这样的一种架构模式。

下面将着重介绍什么是微服务，使用微服务架构会给我们带来哪些好处与挑战。

1.3 什么是微服务

就目前来看，微服务本身并没有一个严格的定义，每个人对微服务的理解都不同。**Martin Fowler**在他的博客中是这样描述微服务的。

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

摘自：<http://www.martinfowler.com/articles/microservices.html>

翻译：简而言之，微服务架构风格这种开发方法，是以开发一组小型服务的方式来开发一个独立的应用系统的。其中每个小型服务都运行在自己的进程中，并经常采用**HTTP资源API**这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术。对这些微服务我们仅做最低限度的集中管理。

从中我们可以看到，微服务架构应具备以下特性：

1. 每个微服务可独立运行在自己的进程里；
2. 一系列独立运行的微服务共同构建起整个系统；
3. 每个服务为独立的业务开发，一个微服务只关注某个特定的功能，例如订单管理、用户管理等；
4. 微服务之间通过一些轻量的通信机制进行通信，例如通过**REST API**进行调用；
5. 可以使用不同的语言与数据存储技术；
6. 全自动的部署机制。

我们还以电影售票系统为例，如使用微服务来架构该应用，架构图如图1-2所示。

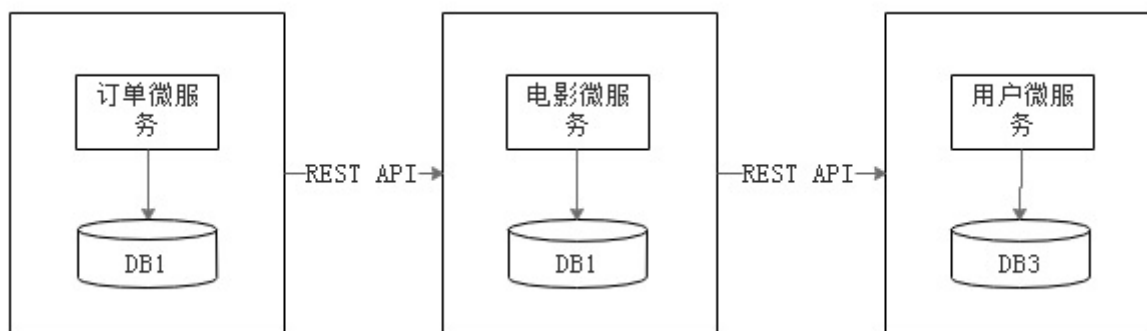


图1-2 电影售票系统微服务架构示意图

我们将整个应用分解为3个微服务。每个微服务都可以独立地运行在自己的进程中；每个微服务分别有自己的数据库；微服务之间以**REST API**进行通信。

1.4 微服务架构的优点与挑战

相对单体应用来说，微服务架构有着显著的优点；同时，微服务并不是免费的午餐，使用微服务也为我们的工作带来了一定的挑战。我们先来分析一下使用微服务有哪些优点。

微服务架构的优点

- 易于开发和维护

因为单个微服务只关注一个特定的业务功能，所以业务清晰、代码量较少。开发和维护单个微服务相对来说是比较简单的，单个微服务会一直处于一个可维护的状态；而整个应用是由若干个微服务构建而成的，所以整个应用也会处于可控状态。

- 单个微服务启动较快

单个微服务代码量较少，所以启动会比较快。

- 局部修改容易部署

单体应用只要有修改，就得重新部署整个应用，微服务则解决了这样的问题。一般来说，对某个微服务进行修改，只需要重新部署这个该服务即可。

- 技术栈不受限

在微服务中，我们可以结合项目业务及团队的特点，合理地选择技术栈。例如某些服务可使用关系型数据库MySQL；某些微服务有图形计算的需求，我们可以选择Neo4J；甚至可以根据需要，部分微服务使用Java开发，部分微服务使用NodeJS进行开发。

- 按需伸缩

我们可以根据单个微服务的需要，实现细粒度的扩展。例如：应用系统中的某个微服务遇到了瓶颈，我们可以结合这个微服务的业务特点，增加内存、升级CPU或者是增加节点。

通过以上的分析，我们可以发现，单体架构的缺点，恰恰是微服务的优点，这些优点使得微服务看起来简直完美。然而完美的东西是不存在的，就像银弹也不存在一样。下面我们来讨论使用微服务为我们带来的挑战。

微服务架构的挑战

- 运维要求较高

更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行；而在微服务中，需要保证几十甚至几百个服务的正常运行与协作，这给项目的运维带来了很大的挑战。

- 分布式固有的复杂性

使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务、异步请求等问题都给我们带来了很大的挑战。

- 接口调整成本高

微服务之间通过接口进行通信。如果修改某一个微服务的API，可能所有使用了该接口的微服务都需要做调整。

- 重复劳动

很多服务可能都会使用到某一个功能，而这个功能并没有达到可以分解为一个微服务的程度，这个时候可能不同的服务团队都会开发这一功能，从而导致代码重复。尽管我们可以使用共享库来解决这样的问题（例如可以将公共部分封装成公共组件，需要该功能的微服务引用该组件），但共享库的方式在多语言环境下就不一定行得通了。

总而言之，微服务虽然有很多吸引人的地方，但是要想使用它也是有成本的——它对开发、测试、运维都带来了一些挑战。下面我们来探讨一下微服务架构的设计原则有哪些，应该怎样去架构微服务。

1.5 微服务设计原则

和数据库设计中的N范式一样，微服务也有一定的设计原则，这些原则指导我们更加合理地架构微服务。

- 单一职责原则

单一职责原则指的是一个单元（类、方法或者服务等）应该只负责整个系统功能的一个单独的、有界限的一部分。单一职责原则可以帮助我们更优雅地开发，更敏捷地交付。

单一职责原则是SOLID原则之一。对SOLID原则感兴趣的读者可前往[https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))) 扩展阅读。

- 服务自治原则

服务自治，是指每个微服务应该具备独立的业务能力、依赖与运行环境，每个微服务可以独立地配置、更新和管理。我们知道，在微服务架构中，服务是独立的业务单元，应该与其他服务高度解耦。每个微服务从开发、测试、构建、部署，都应当可以独立运行，而不应该依赖其他的服务。

- 轻量级通信原则

微服务之间应该通过轻量级的通信机制进行交互。轻量级的通信机制应具备两点：首先是它的体量较轻；其次是该通信机制应该是跨语言、跨平台的。例如我们所熟悉的REST协议，就是一个典型的“轻量级通信机制”；而例如Java的RMI则协议就不是很符合轻量级通信机制的要求，因为它绑定了Java语言。

微服务中常用的协议有：REST、AMQP、STOMP、MQTT等。

- 接口明确原则

每个服务的对外接口应该明确定义，并尽量保持不变。这与前文我们讨论的使用微服务存在“接口调整成本高”的缺点是对应的。

1.6 如何实现微服务？

至此，我们已经知道了微服务的定义及其优缺点，并总结出了一些指导性的原则，为合理地架构微服务提供了理论支持。

下面我们来探讨一下，如何将微服务的理论付诸实践。

1.6.1 微服务技术选型有哪些

相对单体应用的交付来说，微服务应用的交付要复杂很多。所以，我们不仅需要开发框架的支持，还需要一些自动化的部署工具，以及IaaS、PaaS或CaaS的支持。

下面从开发、持续集成、运行平台三个维度考虑挑选技术选型。

开发框架的选择

我们可使用Spring Cloud作为微服务开发框架。

首先，Spring Cloud提供了开箱即用的生产特性，可大大提升开发效率；再者，Spring Cloud的文档丰富、社区活跃，遇到问题比较容易获得支持；更为可贵的是，Spring Cloud为微服务架构提供了完整的解决方案。

当然，也可以使用其他的开发框架或者解决方案实现微服务。例如Dubbo、Dropwizard、armada等。

持续集成

持续集成是一个老生常谈的话题。持续集成工具有Jenkins（原Hudson）、TeamCity、Atlassian Bamboo等。这些工具基本都能满足我们的需要，基于免费、开源和市场占有率等方面的考虑，笔者不能免俗地使用Jenkins作为持续集成的工具。

运行平台

出于轻量、灵活、应用支撑等方面的考虑，我们将应用部署在Docker上。当然，微服务并不绑定运行平台，将微服务部署在PC Server，或者阿里云、AWS等云计算平台都是可以的。

1.6.2 微服务架构图及常用组件

在进入实战之前，我们先来展望一下微服务架构图。

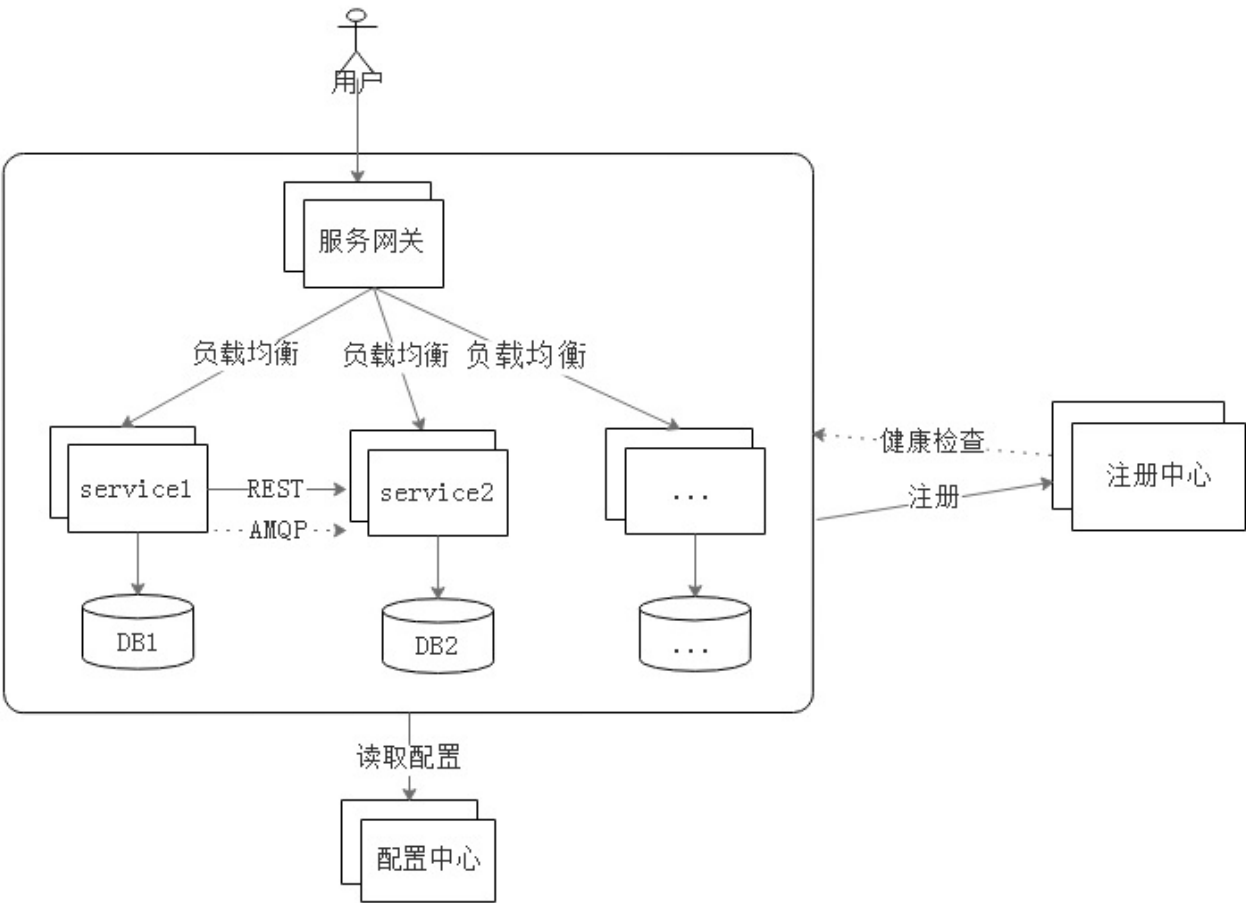


图1-3 微服务架构图

图1-3不严谨地表示了一个微服务应用的架构（之所以说不严谨，是因为配置中心可以注册到注册中心上；而注册中心也可以从配置中心读取配置信息）。

由图我们可以看到，除了service1、service2等业务相关的微服务以外，还有注册中心、服务网关、配置中心等组件。这些组件分别是什么？作用又是什呢？

读者可以带着疑问，本书将在实战的过程中进行详细的讲解。

2 微服务开发框架——Spring Cloud

2.1 Spring Cloud简介及其特点

尽管Spring Cloud中带有Cloud的字样，但是它并不是云计算的解决方案，而是在Spring Boot的基础上构建的、用于快速构建分布式系统中的一些通用模式的工具集。它具有以下特点：

1. 约定优于配置；
2. 适用于各种环境。可开发与部署在PC Server或者各种云环境（例如阿里云、AWS等）；
3. 隐藏了组件的复杂性，并提供声明式、无xml的配置方式；
4. 开箱即用，快速启动；
5. 轻量级的组件。Spring Cloud整合的组件都是比较轻量级的。例如Eureka、RabbitMQ等等，都是非常轻量级的解决方案；
6. 功能齐全，组件支持非常丰富。Spring Cloud为实现微服务提供了非常完整的支持。例如，配置管理、服务发现、断路器、服务网关等；
7. 选型中立而丰富。例如，对于服务发现组件的挑选，我们可以使用Eureka、Zookeeper或Consul，Spring Cloud对它们都有较好的支持；
8. Spring Cloud的组成部分是解耦的，开发人员可以灵活地根据需要挑选技术选型。

通俗地说，Spring Cloud是一个构建分布式系统的全家桶。它的生态圈非常完整，目前已经包含了非常多的子项目，例如：Spring Cloud Config、Spring Cloud Netflix、Spring Cloud Bus、Spring Cloud Security、Spring Cloud Sleuth、Spring Cloud Starters等。

2.2 Spring Cloud的版本简介

目前Spring大多数项目的版本都是以 主版本号.次版本号.增量版本号.里程碑版本号 这种方式命名的，例如Spring最新的稳定版本 `4.3.3.RELEASE`，或者里程碑版本 `5.0.0.M2` 等。这种命名方式也是Maven所推崇的。

我们来看一下Spring Cloud的版本：

| Spring Cloud | | |
|---------------------------------|---------------|-----|
| RELEASE | DOCUMENTATION | |
| Angel SR6 <small>GR</small> | Reference | API |
| Brixton SR6 <small>GR</small> | Reference | API |
| Brixton <small>SNAPSHOT</small> | Reference | API |
| Camden <small>SNAPSHOT</small> | Reference | API |
| Camden <small>GR</small> | Reference | API |

我们可以发现Spring Cloud的版本是以 英文单词 SRn 的方式命名的。那么单词的含义是什么呢？SR又是什么呢？

Angel、Brixton、Camden都是英国的地名，表示了主版本号的演进。SRn指的是第n次的Service Release，有些类似于Windows系统的Service Pack，表示是第n次Bug修复。

下面我们将以代码与讲解结合的方式，为大家讲解Spring Cloud为微服务提供的支持。

3 开始用**Spring Cloud**实战微服务

下面我们正式开始使用Spring Cloud实战微服务。

3.1 Spring Cloud实战前提

Spring Cloud不一定适合所有人。下面我们来探讨一下，玩转Spring Cloud需要具备什么样的技术能力，以及在实战中会使用到的工具。

3.1.1 需要的技术储备

Spring Cloud并不是面向零基础开发人员的，它有一定的学习曲线。学习Spring Cloud需要具备一定的开发素质。

语言基础

Spring Cloud是一个基于Java语言的工具套件。所以要学习Spring Cloud，需要一定的Java基础。当然，Spring Cloud同样也支持使用Scala、Groovy等语言进行开发。

本书我们所提到的示例代码都是基于Java编写的。

Spring Boot

由于Spring Cloud是基于Spring Boot构建的，它延续了Spring Boot的约定以及开发方式。如果大家对Spring Boot不熟悉，建议花一点时间入门。当然如果不了解Spring Boot也没有关系，本书会照顾到不熟悉Spring Boot的读者，由浅入深地进行讲解。

项目管理与构建工具

目前业界比较主流的项目管理与构建工具有Maven和Gradle等。考虑到国内Java程序员的情况，本书采用了更为主流的Maven。大家也可以使用Gradle实现对项目的管理与构建。同时，Maven与Gradle项目是可以互相转换的。例如，使用命令 `gradle init --type pom` 就可以将一个Maven项目转换成Gradle项目了。

3.1.2 使用的工具及软件版本

目前Spring Boot、Spring Cloud都在飞速地发展，是业界有名的“版本帝”。随着版本的演进，Spring Cloud也带来了更强大的功能与更多的特性。

新版本未必代表着完美，但老版本往往意味着过时或即将过时。鉴于这样的指导原则，我们将使用目前最新的Release版本进行实战。涉及到的新特性，笔者会尽量标记出来并做一定的讲解。

下面展示了本书所使用的各项软件版本：

JDK——JDK 1.8

Spring Cloud官方建议使用JDK 1.8。当然，Spring Cloud也支持通过通过一定的配置，支持JDK 1.7。笔者使用的是JDK 1.8。

IDE——Spring Tool Suite 3.8.2

选择一款合适的IDE往往会事半功倍。本书采用的是Spring官方提供的Spring Tool Suite 3.8.2进行讲解的，这是一个基于Eclipse的IDE。当然也可以使用IDEA等IDE进行开发。

Maven——3.3.9

本书使用Maven的最新版本3.3.9。和Spring Cloud一样，Maven 3.3.x默认也是运行在JDK 1.8之上的。如果想使用1.8以下版本的JDK，需要做一些额外的配置。

Spring Boot——1.4.1

本书使用最新的Spring Boot版本1.4.1。

Spring Cloud——Camden SR1

本书使用最新的Spring Cloud Camden SR1进行讲解。Spring Cloud版本演进非常迅速——刚开始研究学习Spring Cloud的时候，它才Release到Angel；到参加完全球微服务架构大会，开始筹备写本书的时候，Spring Cloud刚刚发布了Brixton SR5；而仅仅是过了2个月，Spring Cloud又发布了新的大版本Camden SR1。

Camden相对Brixton来说，提供了更丰富的组件，更多的特性以及更好的稳定性。

一点建议

建议大家进入学习时，尽量使用与本书相同的版本，避免踩坑。

我们知道，学习是有成本的，这个成本是时间和精力。要想降低学习的成本的重要方式之一就是少踩坑。所以建议大家使用与本书相同的版本进行学习，等到自己掌握了相应知识，具备自己排查问题的能力后，再挑选更稳定的版本用于生产。

3.2 创建存在调用关系的微服务

使用微服务构建的是分布式系统。而在分布式系统中，各个微服务之间往往存在着调用关系。我们首先创建存在调用关系的微服务。

3.2.1 服务提供者与服务消费者简介

我们使用服务提供者与服务消费者来描述微服务之间的调用关系。

以下表格简单定义了服务提供者与服务消费者：

| 名词 | 概念 |
|-------|-------------------------|
| 服务提供者 | 服务的被调用方（即：为其他服务提供服务的服务） |
| 服务消费者 | 服务的调用方（即：依赖其他服务的服务） |

我们还以电影售票系统为例：

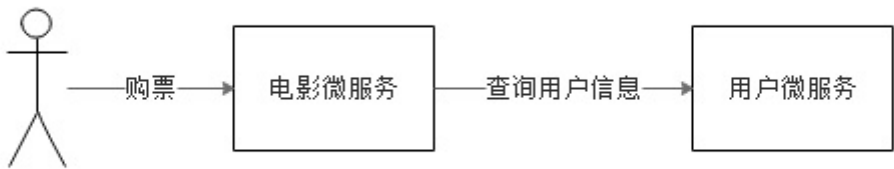


图3-1 服务提供者与服务消费者

如图3-1，用户向电影微服务发起了一个购票的请求。在进行购票的业务操作前，电影微服务需要调用用户微服务的接口，查询当前用户的余额是多少，是不是符合购票标准等。在这种场景下，用户微服务就是一个服务提供者，电影微服务则是一个服务消费者。

3.2.2 编写服务提供者

下面我们来编写一个服务提供者（用户微服务），该服务具备通过主键查询用户信息的能力。

为便于测试，我们使用spring-data-jpa作为持久层框架；使用h2嵌入式数据库引擎作为数据库。Spring Boot对h2数据库的支持非常棒，只需要将建表语句和插表语句存放在项目的classpath下，就能模拟数据库了。

我们首先准备好建表语句：在classpath下建立schema.sql，并加入如下内容：

```
drop table user if exists;
create table user (id bigint generated by default as identity, username varchar(40), name varchar(20), age int(3), balance decimal(10,2), primary key (id));
```

准备几条数据：在classpath下建立文件data.sql，并加入如下内容：

```
insert into user (id, username, name, age, balance) values (1, 'account1', '张三', 20, 100.00);
insert into user (id, username, name, age, balance) values (2, 'account2', '李四', 28, 180.00);
insert into user (id, username, name, age, balance) values (3, 'account3', '王五', 32, 280.00);
```

创建一个Maven工程，并在pom.xml文件添加如下内容：

```

<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-simple-provider-user</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>

```

实体类:

```

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column
    private String username;
    @Column
    private String name;
    @Column
    private Integer age;
    @Column
    private BigDecimal balance;
    ...
    // getters and setters
}

```

持久层:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
```

Controller:

```
@RestController
public class UserController {
    @Autowired
    private UserRepository userRepository;

    @GetMapping("/{id}")
    public User findById(@PathVariable Long id) {
        User findOne = this.userRepository.findOne(id);
        return findOne;
    }
}
```

启动类:

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

配置文件: application.yml

```
server:
  port: 7900
spring:
  jpa:
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
  datasource:
    # 指定数据源
    platform: h2
    # 指定数据源类型
    schema: classpath:schema.sql
    # 指定h2数据库的建表脚本
    data: classpath:data.sql
    # 指定h2数据库的insert脚本
  logging:
    level:
      root: INFO
      org.hibernate: INFO
      org.hibernate.type.descriptor.sql.BasicBinder: TRACE
      org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
      com.itmuch.youran.persistence: ERROR
```

这样的代码，对于接触过Spring Boot的读者来说，应该是非常简单的；对于Spring Boot新手应该也能理解其中的含义，笔者就不赘述了。

测试

访问：<http://localhost:7900/1>，获得结果：

```
{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}
```

说明我们的用户微服务已经具备通过主键查询用户信息的能力了。

TIPS

- 本书中的配置文件采用的是yaml的方式，yaml有严格的缩进，请大家注意。Spring Boot/Spring Cloud支持使用properties或者yaml文件作为配置文件，yaml相对properties的配置方式，可读性、可维护性更强。
- 代码中用到的 `@GetMapping`，是Spring 4.3为我们提供的一个新注解。`@GetMapping` 是一个组合注解，它等价于 `@RequestMapping(method = RequestMethod.GET)`，用于简化我们的开发。同理还有 `@PostMapping`、`@PutMapping`、`@DeleteMapping`、`@PatchMapping` 等。

3.2.3 编写服务消费者

我们已经成功地编写了一个服务提供者（用户微服务），下面我们来编写一个服务消费者（电影微服务）。该服务非常简单，我们使用RestTemplate 调用用户微服务的RESTful API，查询指定用户的信息。

- 首先创建一个Maven项目，在pom.xml中添加如下内容：

```
<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-simple-consumer-movie</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

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

- 实体类：

```
public class User {
    private Long id;
    private String username;
    private String name;
    private Integer age;
    private BigDecimal balance;
    ...
    // getters and setters
}
```

- Controller:

```

@RestController
public class MovieController {
    @Autowired
    private RestTemplate restTemplate;

    @Value("${userServiceUrl}")
    private String userServiceUrl;

    @GetMapping("/simple/{id}")
    public User findById(@PathVariable Long id) {
        return this.restTemplate.getForObject(this.userServiceUrl + "/" + id, User.class);
    }
}

```

- 启动类:

```

@SpringBootApplication
public class ConsumerMovieApplication {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

```

- 配置文件:

```

server:
  port: 7901
  userServiceUrl: http://localhost:7900/

```

这样，一个电影微服务就完成了，so easy!

测试

访问: <http://127.0.0.1:7901/simple/1>，获得结果:

```

{
  "id": 1,
  "username": "account1",
  "name": "张三",
  "age": 20,
  "balance": 100
}

```

结果正常，我们可以正常使用RestTemplate调用用户微服务的API。

TIPS

- `@Bean`是一个方法注解，作用是实例化一个Bean，并使用该方法的名称命名。例如：

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

等价于 `RestTemplate restTemplate = new RestTemplate();` 。

3.2.4 硬编码存在什么问题？

至此，我们已经实现了一个用户微服务和电影微服务，并在电影微服务中使用`RestTemplate`调用用户微服务中的RESTful API。一切都是那么的自然、简单，**perfect!**

那么真的完美吗？我们来分析一下上文的代码：在服务消费者`MovieController.java`中：

```
@GetMapping("/simple/{id}")
public User findById(@PathVariable Long id) {
    return this.restTemplate.getForObject(this.userServiceUrl + "/" + id, User.class);
}
```

我们可以看到：电影微服务读取配置文件 `application.yml` 中的配置项 `userServiceUrl`：
`http://localhost:7900/`，然后请求 `http://localhost:7900/1`。

也就是说，我们是把提供者的网络地址（IP和端口等）硬编码在配置文件中的，在传统的应用程序中，我们也经常这么做。这种做法有哪些问题呢？

- 适用场景有局限

如果服务提供者的网络地址（IP和端口）发生了变化，将会影响服务消费者。例如，假设用户微服务的网络地址发生了变化，会导致我们需要修改电影微服务的配置，并重新发布。这显然是不可取的。

- 无法动态伸缩

在生产环境中，每个微服务往往部署多个实例，以实现容灾和负载均衡。并且在微服务中，我们需要系统具备自动伸缩的能力，动态地增减节点，硬编码无法适应这种需求。

那么要如何解决这样的问题呢？且听下回分解。

4 微服务注册与发现

4.1 服务发现简介

通过上文，我们知道硬编码配置的方式存在一些棘手的问题。要想解决这些问题，服务消费者需要一个强大的服务发现机制——使用这种机制，服务消费者无需修改配置文件，就能知道服务提供者的网络信息。提供这种能力的，就是服务发现组件，在微服务架构中，这是一个非常重要的组件。

使用服务发现机制后，架构图大致如图4-1所示：

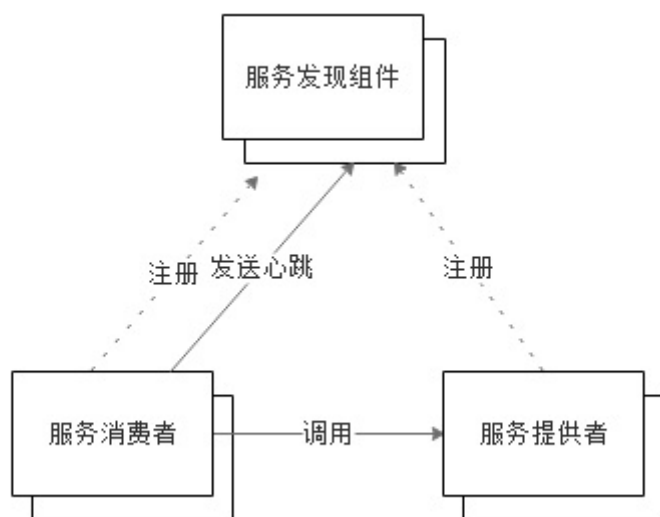


图4-1 服务发现

如图，服务的启动、使用过程大致如下：

1. 各个微服务启动时，将自己的网络地址等信息注册到服务发现组件中；
2. 服务发现组件会返回服务提供者的地址列表给服务消费者；
3. 服务消费者获得了服务提供者的网络地址，并进行调用；
4. 如果服务提供者网络地址发生了变更（例如服务提供者实例增减或者IP端口发生变化等），服务发现组件会推送给服务消费者；
5. 各微服务与服务发现组件保持心跳，如果长时间心跳断开，服务发现组件则会将该实例踢出。

综上，服务发现组件包含了以下几个功能：

- 服务注册表

服务注册表是一个记录当前可用的服务实例的网络信息的数据库，是服务发现组件的核心。服务注册表提供查询API和管理API，使用查询API获得可用的服务实例，使用管理API实现服务的注册和注销；

- 服务注册

服务注册很好理解，就是服务启动时，将服务的网络地址注册到服务注册表中；

- 健康检查

服务发现组件会通过一些机制定时检测已注册的服务，如果发现某服务无法访问了（可能是某几个心跳周期后），就将该服务从服务注册表中移除。

服务发现的好处是显而易见的。我们不再需要使用硬编码各种网络地址，而只需要服务的名称（或者其他信息）就可以使用该服务。

TIPS

- 目前市面上的书籍，服务注册、服务发现、注册中心，在多数场景下，都可以理解为是服务发现组件。
- 服务发现的方式可分为服务器端发现及客户端发现；可作为服务发现组件的技术栈选择也比较多，但是原理是类似或者相通的。

4.2 Spring Cloud中提供的服务发现组件

Spring Cloud为我们提供了多种服务发现组件的支持，例如Eureka、Consul、Zookeeper。限于篇幅，无法一一为大家讲解。本书将以Eureka为例，为大家讲解服务发现。

4.3 Eureka简介

Eureka是Netflix开发的服务发现组件，本身是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。Spring Cloud将它集成在其子项目spring-cloud-netflix中，以实现Spring Cloud的服务发现功能。

Eureka 项目相当活跃，代码更新相当频繁，目前最新的版本是1.5.5。Eureka 2.0的版本也正在紧锣密鼓地开发中，2.0将会带来更好的扩展性，并且使用细粒度的订阅模型取代了基于拉取的模型，但是由于还没有Release，故而不作讲解。

本书讲解的Spring Cloud Camden SR1使用的Eureka版本是1.4.11，还是比较新的。同时有了Eureka 1.x的基础，未来上手Eureka 2.x也会比较容易。

Eureka的Github: <https://github.com/Netflix/eureka>

4.4 Eureka原理

region、zone、Eureka集群的关系

在分析Eureka原理之前，我们需要知道region、zone、Eureka集群三者的关系。Eureka的官方文档对region、zone的描述非常抽象，新手很难理解。

图4-2简单描述了三者之间的大致关系：

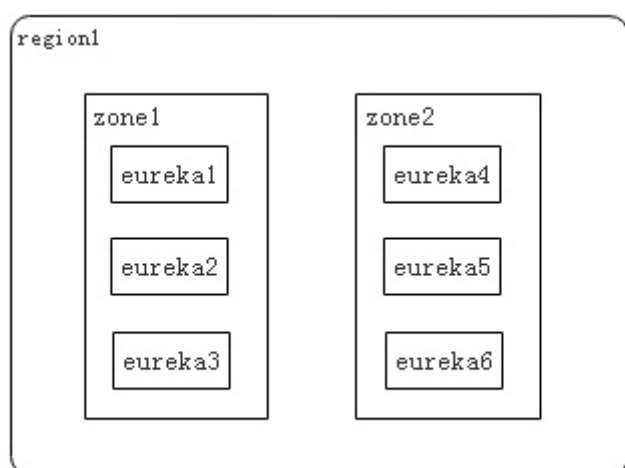


图4-2 region、zone、Eureka集群之间的关系

Eureka的region和zone（或者Availability Zone）均是AWS的概念，在非AWS环境下，我们可以简单地将region认为是我们的整个Eureka集群。

对region和zone感兴趣的读者可前往<http://blog.csdn.net/awschina/article/details/17639191> 扩展阅读，Spring Cloud中默认的region是 `us-east-1`。

Eureka架构

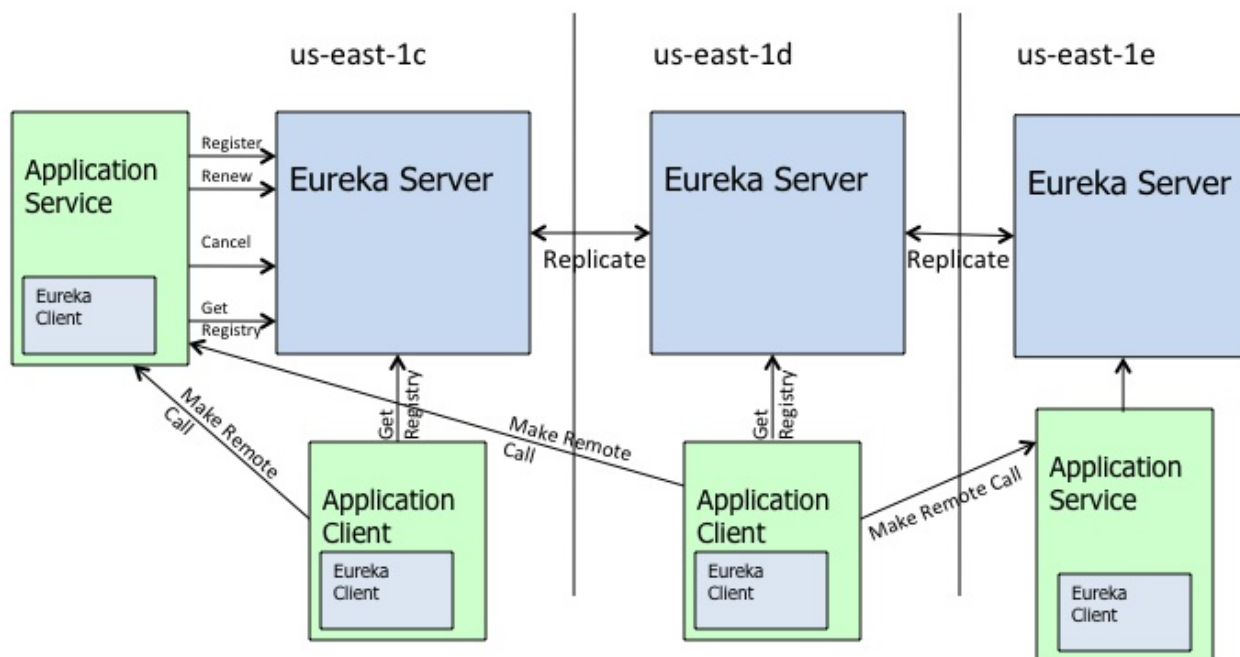


图4-3 Eureka架构图

图4-3是来自Eureka官方的架构图，大致描述了Eureka集群的工作过程。图中包含的组件非常多，可能比较难以理解，我们用通俗易懂的语言解释一下：

- Application Service 相当于本书中的服务提供者，Application Client相当于本书中的服务消费者；
- Make Remote Call，可以简单理解为调用RESTful API；
- us-east-1c、us-east-1d等都是zone，它们都属于us-east-1这个region；

由图可知，Eureka包含两个组件：Eureka Server 和 Eureka Client，它们的作用如下：

- Eureka Client是一个Java客户端，用于简化与Eureka Server的交互；
- Eureka Server提供服务发现的能力，各个微服务启动时，会通过Eureka Client向Eureka Server进行注册自己的信息（例如网络信息），Eureka Server会存储该服务的信息；
- 微服务启动后，会周期性地向Eureka Server发送心跳（默认周期为30秒）以续约自己的信息。如果Eureka Server在一定时间内没有接收到某个微服务节点的心跳，Eureka Server将会注销该微服务节点（默认90秒）；
- 每个Eureka Server同时也是Eureka Client，多个Eureka Server之间通过复制的方式完成服务注册表的同步；
- Eureka Client会缓存Eureka Server中的信息。即使所有的Eureka Server节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者。

综上，Eureka通过心跳检测、健康检查和客户端缓存等机制，提高了系统的灵活性、可伸缩性和可用性。

4.5 实现一个Eureka Server

Eureka的原理比较抽象，包含的组件也比较多。下面我们通过代码，详细讲解如何使用Eureka完成服务发现。我们先来实现一个Eurek Server。

在Spring Cloud中，实现一个Eureka Server是非常简单的一件事。

- 我们首先将Maven依赖添加到项目中：

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

- 在启动类上添加 `@EnableEurekaServer` 注解；
- 最后在配置文件application.yml中添加如下配置：

```
server:
  port: 8761 # 指定该Eureka实例的端口
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

这样就完成了一个简单的Eureka Server。我们简要说明一下application.yml中的配置项：

`eureka.client.registerWithEureka` ：表示是否将自己注册到Eureka Server，默认为true。由于当前这个应用就是Eureka Server，故而设为false。`eureka.client.fetchRegistry` ：表示是否从Eureka Server获取注册信息，默认为true。因为这是一个单点的Eureka Server，不需要同步其他的Eureka Server节点的数据，故而设为false。`eureka.client.serviceUrl.defaultZone` ：设置与Eureka Server交互的地址，默认是 `http://localhost:8761/eureka` ；多个使用，分隔。查询服务和注册服务都需要依赖这个地址。

Eureka的配置类所在类：

```
org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean
org.springframework.cloud.netflix.eureka.EurekaClientConfigBean
org.springframework.cloud.netflix.eureka.server.EurekaServerConfigBean
```


4.5.1 Eureka Server代码示例

- 创建一个Maven工程，并在pom.xml中加入如下内容：

```
<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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-discovery-eureka</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
  </dependencies>
</project>
```

- 编写启动程序：


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

- 配置文件：

```
server:
  port: 8761 # 指定该Eureka实例的端口
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
```

这样一个简单的Eureka Server就完成了。

- 启动工程后，访问：<http://localhost:8761/>。我们会发现此时还没有服务注册到Eureka上面，如下图：



[HOME](#) [LAST 1000 SINCE STARTUP](#)

System Status

| | |
|-------------|---------|
| Environment | test |
| Data center | default |

| | |
|--------------------------|---------------------------|
| Current time | 2016-09-29T17:32:31 +0800 |
| Uptime | 00:00 |
| Lease expiration enabled | false |
| Renews threshold | 1 |
| Renews (last min) | 0 |

DS Replicas

localhost

Instances currently registered with Eureka

| Application | AMIs | Availability Zones | Status |
|------------------------|------|--------------------|--------|
| No instances available | | | |

General Info

| Name | Value |
|----------------------|-------------------------------|
| total-avail-memory | 427mb |
| environment | test |
| num-of-cpus | 4 |
| current-memory-usage | 78mb (18%) |
| server-uptime | 00:00 |
| registered-replicas | http://localhost:8761/eureka/ |
| unavailable-replicas | http://localhost:8761/eureka/ |
| available-replicas | |

Instance Info

| Name | Value |
|--------|--------------|
| ipAddr | 192.168.0.59 |
| status | UP |

该页面展示了Eureka的系统状态、当前注册到Eureka Server上的服务实例、一般信息、实例信息等。我们可以看到，当前还没有任何服务被注册到Eureka Server上。

4.6 实现一个Eureka Client

实现了Eureka Server后，我们再来实现一个Eureka Client。这同样是非常简单的，我们只需要以下三步：

- 首先在pom.xml中添加 `spring-cloud-starter-eureka` 的依赖，如下：

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

- 然后在配置文件application.yml中添加如下配置：

指定应用名称是什么：

```
spring:
  application:
    name: 项目名称    # 项目名称建议小写
```

- 将该应用注册到Eureka Server中：

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/    # 指定注册中心的地址
  instance:
    preferIpAddress: true    # 表示将自己的IP注册到Eureka，如果不配置将会使用系统的hostname
```

- 在启动类上添加注解 `@EnableDiscoveryClient` 或者 `@EnableEurekaClient` 。

这样就可以完成一个Eureka Client了。

TIPS

我们知道，在Spring Cloud中，服务发现组件的选择有多个，`@EnableDiscoveryClient` 提供了各种服务发现软件的支持；`@EnableEurekaClient` 是 `@EnableDiscoveryClient` 的别名注解，比 `@EnableDiscoveryClient` 更加的语义化。

4.6.1 Eureka Client代码示例

下面我们将之前的 `microservice-simple-provider-user` 注册到Eureka Server上。

- pom.xml:

```
<?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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>microservice-provider-user</artifactId>
  <packaging>jar</packaging>

  <parent>
    <groupId>com.itmuch.cloud</groupId>
    <artifactId>spring-cloud-microservice-study</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>

  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
  </dependencies>
</project>
```

启动类:

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

配置文件:

```

server:
  port: 8000
spring:
  application:
    name: microservice-provider-user    # 项目名称建议小写
  jpa:
    generate-ddl: false
    show-sql: true
    hibernate:
      ddl-auto: none
  datasource:
    platform: h2                      # 指定数据源
    schema: classpath:schema.sql      # 指定数据源类型
    data: classpath:data.sql          # 指定h2数据库的建表脚本
    # 指定h2数据库的insert脚本
logging:
  level:
    root: INFO
    org.hibernate: INFO
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
    org.hibernate.type.descriptor.sql.BasicExtractor: TRACE
    com.itmuch.youran.persistence: ERROR
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/    # 指定注册中心的地址
  instance:
    preferIpAddress: true                    # 表示将自己的IP注册到Eureka，如果不配置将会使用系统的hostname

```

这样，就可以将用户微服务注册到Eureka Server上了。

4.6.2 目前存在的问题

至此，我们已经实现了一个Eureka Server；各个微服务启动时，通过Eureka Client将自己的网络信息注册到Eureka Server上。架构图如图4-3所示。

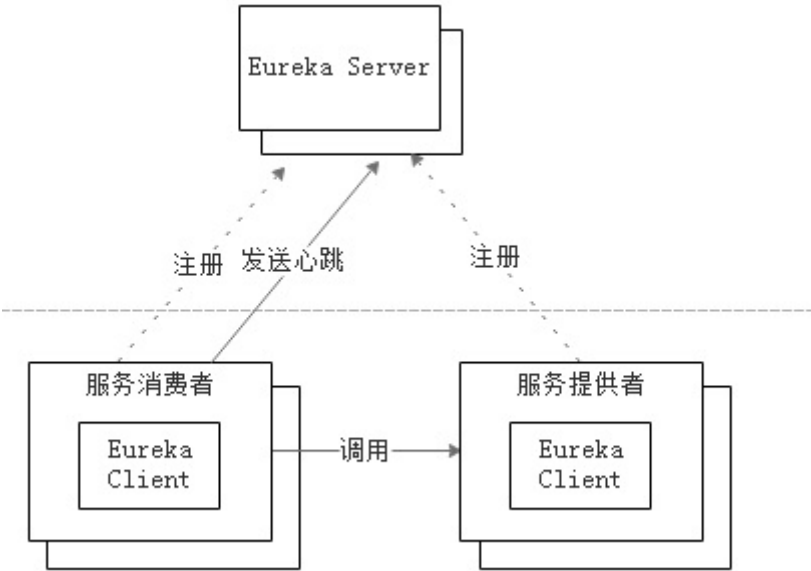


图4-3 微服务架构图

世界似乎更美好了一些。然而现实是，这样的架构离完美还有一定的距离。那么目前的架构存在什么问题呢？

负载均衡。在生产环境，服务消费者与服务提供者都是多实例的。那么在调用的时候，服务消费者如何才能够将请求分摊到多个服务提供者上呢？

4.6.3 客户端侧负载均衡——Ribbon

Ribbon是Netflix发布的一个客户端侧的负载均衡器，它可以对HTTP和TCP客户端的行为进行大量的控制。为服务消费者配置可用服务列表后，Ribbon会自动地帮助我们基于某种负载均衡算法（如轮询、随机等）去请求，我们也可以很方便地为Ribbon实现自定义的负载均衡算法。

当Ribbon与Eureka联合使用时，Ribbon会从Eureka Server中获取服务提供者列表，Ribbon从服务提供者地址列表中，基于负载均衡算法，选择一个提供者实例进行请求。

图4-3展示了Eureka使用Ribbon时候的大致架构：

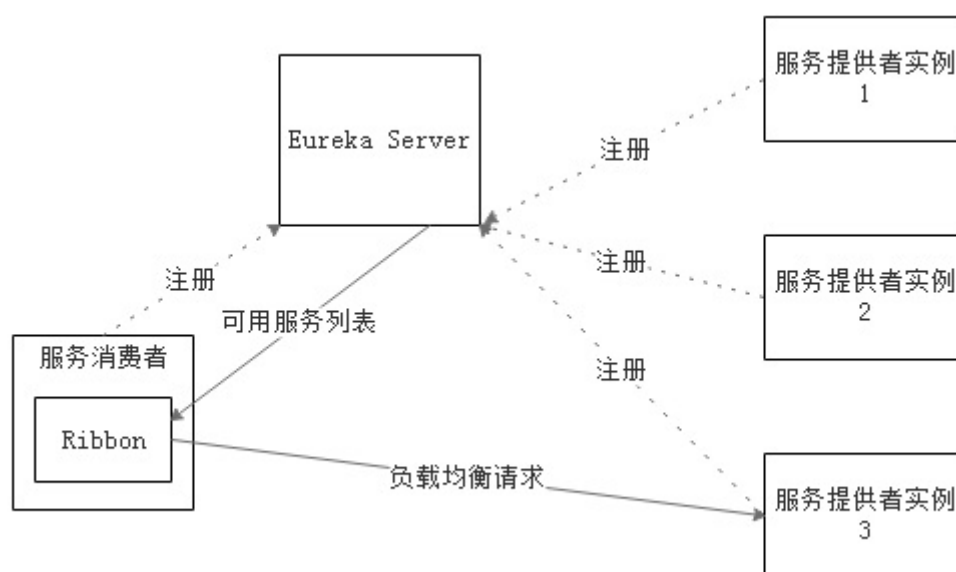


图4-3 Eureka 与 Ribbon联合使用架构图

4.6.3.1 Ribbon代码示例

在Spring Cloud中，使用Ribbon非常简单。下面我们来为之前编写的电影微服务添加Ribbon的支持。

- 引入依赖

前文将电影微服务注册到Eureka Server时，添加了 `spring-cloud-starter-eureka`，该依赖已经引入了Ribbon，所以无需手动引入Ribbon的依赖。

- 添加 `@LoadBalanced` 注解

回顾之前的代码，在启动类中，实例化RestTemplate的代码如下：

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

为这个方法添加注解 `@LoadBalanced` 注解，就可以整合Ribbon，让RestTemplate具备客户端侧负载均衡的能力。

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

- 修改调用地址：在application.yml中，找到

```
userServiceUrl: http://localhost:7900/
```

修改为：

```
userServiceUrl: http://microservice-provider-user/
```

其中 `microservice-provider-user` 是服务提供者的虚拟主机名（virtual host name），当Ribbon和Eureka联合使用时，会自动将虚拟主机名转换成服务在Eureka Server中注册的IP地址并进行调用。

TIPS

虚拟主机名与虚拟IP非常类似，如果大家接触过HAProxy或Heartbeat，理解虚拟主机名就非常容易了。如果理解不了虚拟主机名，那可以简单地认为是提供者在Eureka Server中注册的服务名称，因为默认情况下，虚拟主机名和服务名称是一致的。当然也可以通过配置项 `eureka.instance.virtual-host-name` 或者 `eureka.instance.secure-virtual-host-name` 来指定虚拟主机名。