Table of Contents

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，它正在博客、社交媒体和各种会议演讲的出镜率非常之高，笔者相信大家都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在炒概念。

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，由于微服务理论性的内容，互联网上已经有非常多，本书不会过多涉及，我们尽量把篇幅花在对微服务的实战上。

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，我们称之为单体架构。

我们以一个电影售票系统为例，如果采用单体架构，它的架构图如图所示。

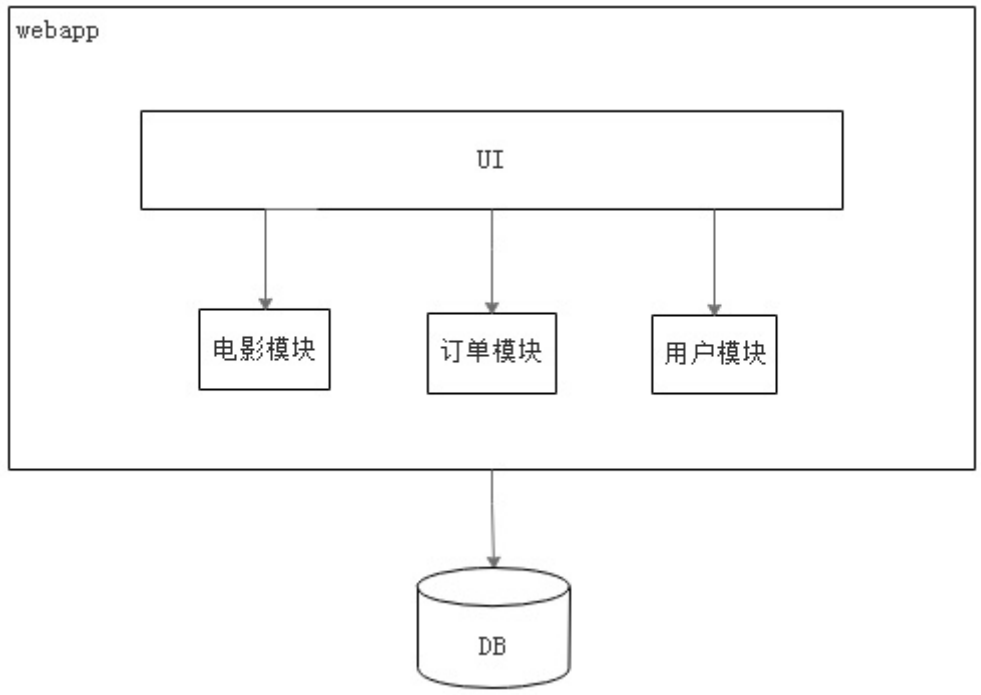


图1-1 单体架构示意图

由图我们可以看到，UI和若干业务模块都被打包在一个war包中，该war包包含了整个系统所有的功能。尽管应用本身也进行了模块化，但是它最终还是会打包并部署为一个单体应用。

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，随着项目业务的不断扩张，需求的不断增加，单体架构已经无法适应互联网时代的需求。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。在这种情况下，单体应用会变得越来越臃肿，可维护性、灵活性降低，而维护的成本却不断地在增加。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，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。

- 复杂性较高

以笔者经手的一个百万行级别的单体应用为例，整个项目非常复杂，包含的模块非常多，模块的边界模糊，依赖关系不清晰，多处存在循环依赖，代码质量还比较差，共有9000多个方法，其中只有2000个方法有注释。每次修改代码都心惊胆战，甚至改一个BUG还会造成隐含的缺陷。

- 交付周期长

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，因此，单体项目部署频率较低。而部署频率低又导致了两次发布之间有大量的功能或者缺陷需要变更，出错概率比较高。

- 扩展能力受限

单体应用作为一个整体部署，无法根据业务模块的特点进行伸缩，只能作为一个整体进行扩展。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及开发框架。然而，技术是不断在发展的，近几年也不断有开发效率更高、性能更好的技术出现。如果想要尝试使用引入新的框架或技术会非常困难。例如：假设之前使用的是Struts2代码构建了一个100万行的单体应用，如果现在想要使用Spring MVC，切换成本是是非常高的。

综上，随着业务需求的发展，功能的不断增加，单体架构很难满足业务快速变化的需要。一方面，代码的可维护性、灵活性、可扩展性在降低；另一方面运维、测试的成本在增加。

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

由上文我们看到，单体应用架构存在很多的问题。那么是不是有一种架构的方法论可以解决单体架构存在的问题呢？我们需要这种架构方式，能够做到以下几点：

- 能够降低应用的复杂性，让代码维护不是那么的困难；
- 对于局部的修改，可以部分部署或者增量部署；
- 系统启动时间快；
- 能够根据业务的特点进行伸缩；
- 不绑定技术栈。

微服务就是这样的一种架构风格。下面我们来了解一下微服务架构究竟是什么。

1.3 什么是微服务

从业界的讨论看，微服务本身并没有一个严格的定义。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

翻译一下：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，微服务架构大致具备以下特点：

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了整个系统；
3. 每个服务为独立的业务开发，一个微服务一般完成某个特定的功能，比如：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或者RPC的方式进行调用。

我们还以电影售票系统为例，使用微服务来架构该系统，架构图如图1-2所示。

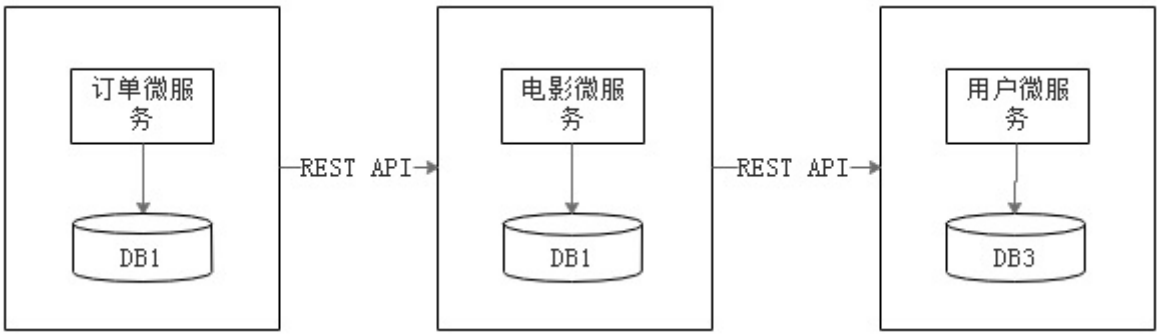


图1-2 微服务架构示意图

我们可以看到，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

微服务架构相对单体架构有着显著的优点。同时，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具有哪些优点。

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业务功能，所以业务清晰、代码量相对较小。开发和维护单个微服务相对来说是比较简单的；而整个应用是由若干个微服务构建而成，所以整个应用也会处于可控状态。

- 启动较快

对于单个微服务，启动是比较快的。

- 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。多数情况下，对某个微服务进行了修改，只需要重新部署这个服务即可。

- 技术栈不受限

在微服务中，我们可以根据项目业务及团队的情况，选择想要的技术栈。例如某些服务使用了关系型数据库MySQL；而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

我们可以按照单个微服务的需要，实现细粒度的扩展。例如：系统中的某个微服务遇到了瓶颈，那我们就根据需要扩展该服务，譬如根据它的业务特点，增加内存、提升配置或者增加节点，就可以实现对整个应用的扩展。

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。这些优势使得微服务看起来简直完美。然而完美的东西是不存在的，银弹也不存在，微服务也存在不足之处。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

在很多服务中可能都会使用到同一个功能，而这一功能点并没有大到抽离一个微服务的程度，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。

尽管我们可以在服务之间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一个公共组件，然后需要该功能的微服务引用该组件，但是在多语言环境下就不一定行得通了。

总而言之，尽管微服务有很多吸引人的地方，但是它对开发、运维都带来了一些新的挑战。下面我们来探讨一下微服务架构的设计原则有哪些，应该怎么样去合理地架构微服务。

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，这些原则指导大家更合理地架构微服务。

- 单一职责原则

单一职责原则相信大家并不陌生。它指的是一个单元（类、方法或者服务），应该只有一个职责。每个单元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，并且让交付更加敏捷。

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

- 服务自治原则

服务自治，是指每个服务应该具备独立的业务能力、依赖与运行环境，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，服务是一个独立的业务单元，应该与其他服务高度解耦。每个服务从开发、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

服务之间应该通过轻量级的通信机制以实现交互。轻量级通信机制需要具备两点：首先是体量较轻；其次这些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则协议不是很符合我们的要求，因为它绑定了Java的技术栈。

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，为我们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

微服务应用的交付相对单体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

对于开发框架，我们可使用Spring Cloud作为微服务的开发框架。

首先，Spring Cloud提供了开箱即用的生产属性，可大大提升开发效率。再者，Spring Cloud的文档丰富，社区也比较热，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，实现微服务也可以使用其他的开发框架或者解决方案，例如：Dubbo、Dropwizard、armada等。

持续集成

持续集成是一个比较老生常谈的话题。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、开源以及市场占有率等考虑，笔者不能免俗地使用了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，微服务并不绑定平台，将微服务部署在例如阿里云，AWS，或基于OpenStack、CloudStack构建的公有云、私有云上也是可以的。

1.6.2 微服务架构图及常用组件

在进入实战之前，我们首先展望一下，看一下一个微服务架构的最终架构图是什么样子的。

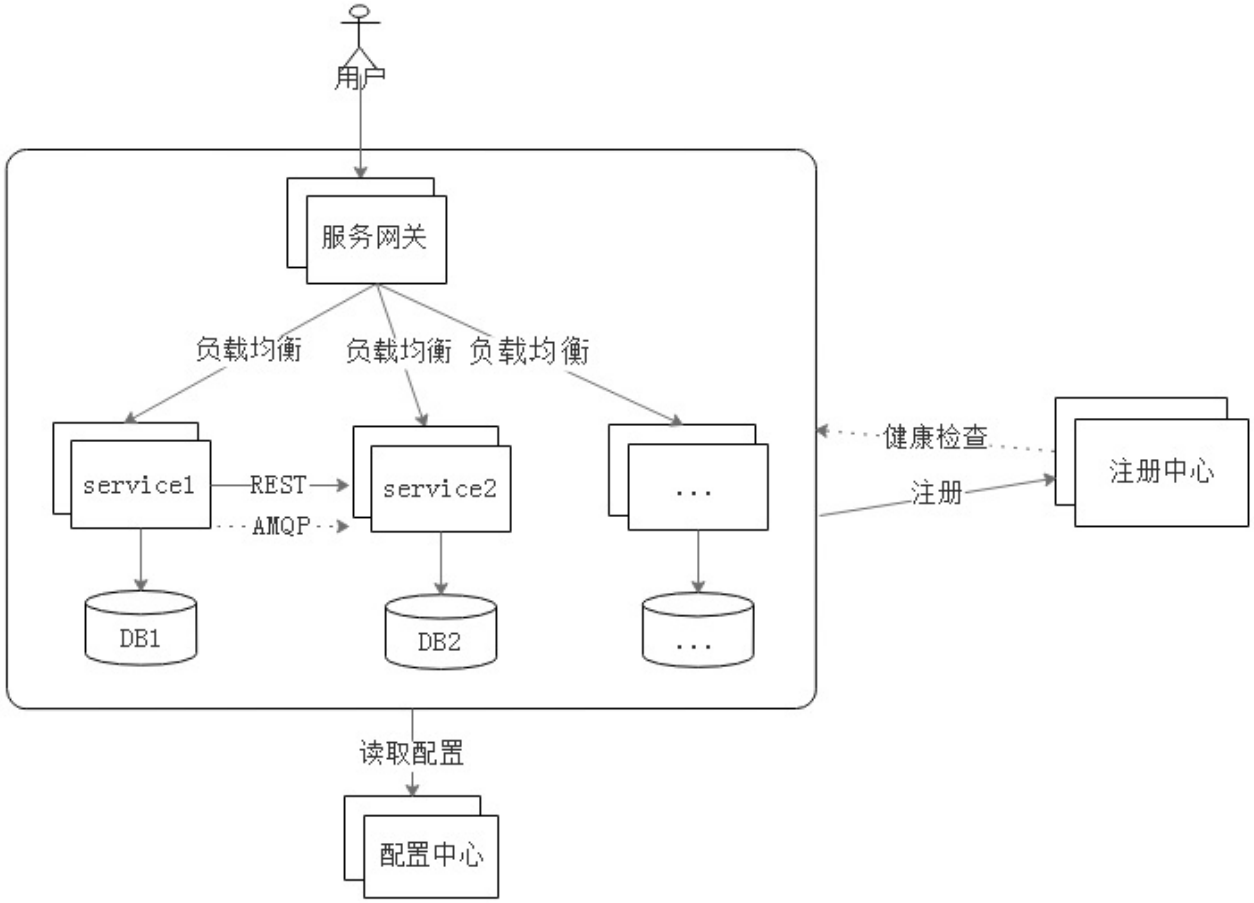


图1-3 微服务架构图

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，除了业务所需要的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。

配置中心大家很好理解，是一个用于管理配置的微服务。可注册中心、服务网关是什么？作用又是什么？我们将在实战的过程中进行具体的讲解。

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：

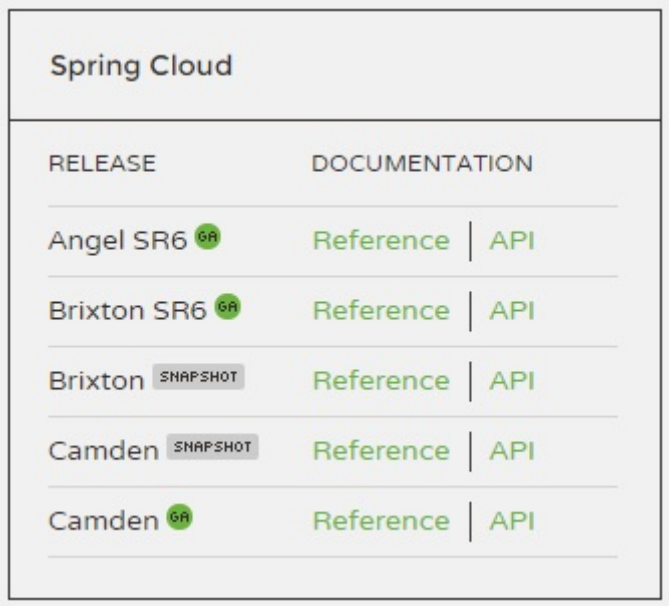
1. 约定优于配置；
2. 适用于各种环境。可开发与部署在PC服务器或者各种云环境（例如阿里云、AWS等）；
3. 隐藏了组件的复杂性，并提供声明式、无xml的配置方式；
4. 开箱即用，快速启动；
5. 轻量级的组件。Spring Cloud整合的组件都是比较轻量级的。例如Eureka、RabbitMQ等等，都是非常轻量级的解决方案；
6. 功能齐全，组件支持非常丰富。Spring Cloud为实现微服务提供了非常完整的支持。例如，配置管理、服务发现、断路器、服务网关等；
7. 选型中立而丰富。例如，对于服务发现组件的挑选，我们可以使用Eureka、Zookeeper或Consul，Spring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

简单来说，Spring Cloud是一个构建分布式系统的全家桶。它的生态圈非常强大，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绝大多数项目的版本命名方式一般是以 `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，这是Maven较为推崇的版本命名方式。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。

我们来看一下Spring Cloud的版本：



我们可以发现Spring Cloud的版本是以 `单词 SRn` 这种方式命名的。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，表明是第n次Bug修复的小版本，类似于Windows系统的Service Pack。

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，想要玩转Spring Cloud，需要什么样的技术能力；以及我们实战中所使用到的开发工具。

3.1.1 需要的技术储备

Spring Cloud并不是面向零基础开发开发人员的，学习Spring Cloud需要大家具备一定的开发素质。

语言基础

Spring Cloud是一个基于Java开发的工具套件，所以要学习它，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。

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两款：Maven和Gradle。考虑到国内Java程序员的情况，本书采用了更为主流的Maven。当然，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。

Spring Boot

由于Spring Cloud是基于Spring Boot构建的，它使用了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，当然不了解Spring Cloud也没有关系，本书的示例是由浅入深的，会照顾到不熟悉Spring Boot的读者。

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”。随着产品的不断迭代，新版本的Spring Cloud也带来了更强大的功能与更多的特性。

新的软件版本未必代表是完美的，但是旧的版本往往意味着是过时的。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，笔者会尽量标记并做一定讲解，以避免大家的不适应。

以下表格展示了我们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进行开发。当然，Spring Cloud也支持通过通过一定的配置，使用JDK 1.7进行开发。本书使用的是JDK 1.8版本进行讲解的。

IDE——Spring Tool Suite

选择一款合适、好用的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或者原生的Eclipse进行开发。不过原生的Eclipse需要安装一些Spring的插件才能进入开发。

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

我们使用最新的Spring Boot版本 1.4.1。

Spring Cloud——Camden SR1

我们使用最新的 Spring Cloud Camden SR1进行讲解。Spring Cloud版本演进非常迅速——我在研究学习Spring Cloud的时候Spring Cloud才Release到Angel；等到参加完全球微服务架构大会，开始筹备写一本使用Spring Cloud实战微服务的书的时候，Spring Cloud刚刚发行了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，提供了更加丰富的组件，更多的特性以及更好的软件稳定性。

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，那就得少踩坑。故此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，具备了自己排查问题的能力后，再挑选更加稳定、更适合公司生产的版本。

3.2 创建存在调用关系的微服务

我们已经知道，使用微服务架构出来的是分布式系统。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分布式系统中服务之间的调用关系。

下面这张表格，简单描述了服务提供者与服务消费者是什么：

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）

我们还以一个电影销售系统为例进行讲解。假设我们以微服务架构整个电影微服务系统，电影微服务承担了票务能力；用户微服务则维护用户的信息。

如上图，用户向电影微服务发起了一个购票的请求。在正式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家口，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DML语句放置在项目的classpath下，就能使用了。

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), ag
e 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：

```xml
<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：

```java
@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 stters
}
```

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
}
```

结果正常，我们的用户微服务已经可以提供通过主键查询用户信息的能力了。

TIPS

- 代码中用到了 `@GetMapping("/{id}")` 注解，这是一个组合注解，我们可以简单地理解为等价于 `@RequestMapping(value = "/id", method = RequestMethod.GET)` ，这个注解是Spring 4.3以后提供的，用于简化我们的开发。同理还有 `@PostMapping` 、 `@PutMapping` 、 `@DeleteMapping` 、 `@PatchMapping` 等注解。
- 本书中的配置文件采用的是yml的方式，yml有严格的缩进，请大家注意。Spring Cloud同样也支持使用properties作为配置文件。yml相对properties的配置方式，更易读、更易维护。

3.2.3 编写服务消费者

上文我们已经成功地创建了一个服务提供者（用户微服务），下面我们创建一个服务消费者（电影微服务）。该服务非常简单，我们在其中使用RestTemplate 请求用户微服务的API，以查询某个指定用户的信息。

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 + "1", 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。

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();
}
```

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

3.2.4 硬编码存在什么问题？

经过上文两节，我们实现了一个用户微服务，还实现了一个电影微服务，并在电影微服务中使用RestTemplate调用用户微服务中的RESTful接口。一切都是那么的自然、简单，perfect！那么上文的代码真的完美吗？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 + "1", 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，我们是把配置硬编码在配置文件中的。在传统的开发中，我们往往也是这么做的。但是这样存在什么样的问题呢？

- 适用场景有局限

如果服务提供者的网络地址发生了变化，将会影响服务消费者。例如：我们对用户服务进行了修改并重新发布，此时该服务的网络地址发生了变化，这将会导致电影微服务的异常——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，以完成容灾和伸缩。在微服务中，系统往往是自动伸缩的，硬编码无法适应这种需求。

尽管可以借助Nginx等反向代理软件对用户微服务进行反向代理，以适应这种需求，但是手动对Nginx的配置也是很麻烦的。

- 如何解决？

那么要如何解决这样的问题呢？且听下回分解。