

第9章 微服务架构基础

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 了解微服务的基本概念和常见的解决方案。
- 了解**Spring Cloud**和**Spring Cloud Alibaba**的基本组件及其功能。
- 掌握微服务注册、配置和调用原理和实践方法。
- 掌握微服务容错原理和实践方法。
- 掌握微服务网关原理和实践方法。
- 掌握微服务分布式事务原理和实践方法。
- 掌握微服务链路追踪原理和实践方法。



主要内容

- 9.1 微服务概述
- 9.2 微服务注册、配置和调用
- 9.3 微服务容错
- 9.4 微服务网关
- 9.5 微服务分布式事务
- 9.6 微服务链路追踪



9.1 微服务概述

■ 9.1.1 微服务简介

■ 9.1.2 Spring Cloud简介

■ 9.1.3 Spring Cloud Alibaba简介



9.1.1 微服务简介

- 1.什么是微服务
- 2.微服务架构的常见问题
- 3.微服务架构的常见概念
- 4.微服务架构的常见解决方案



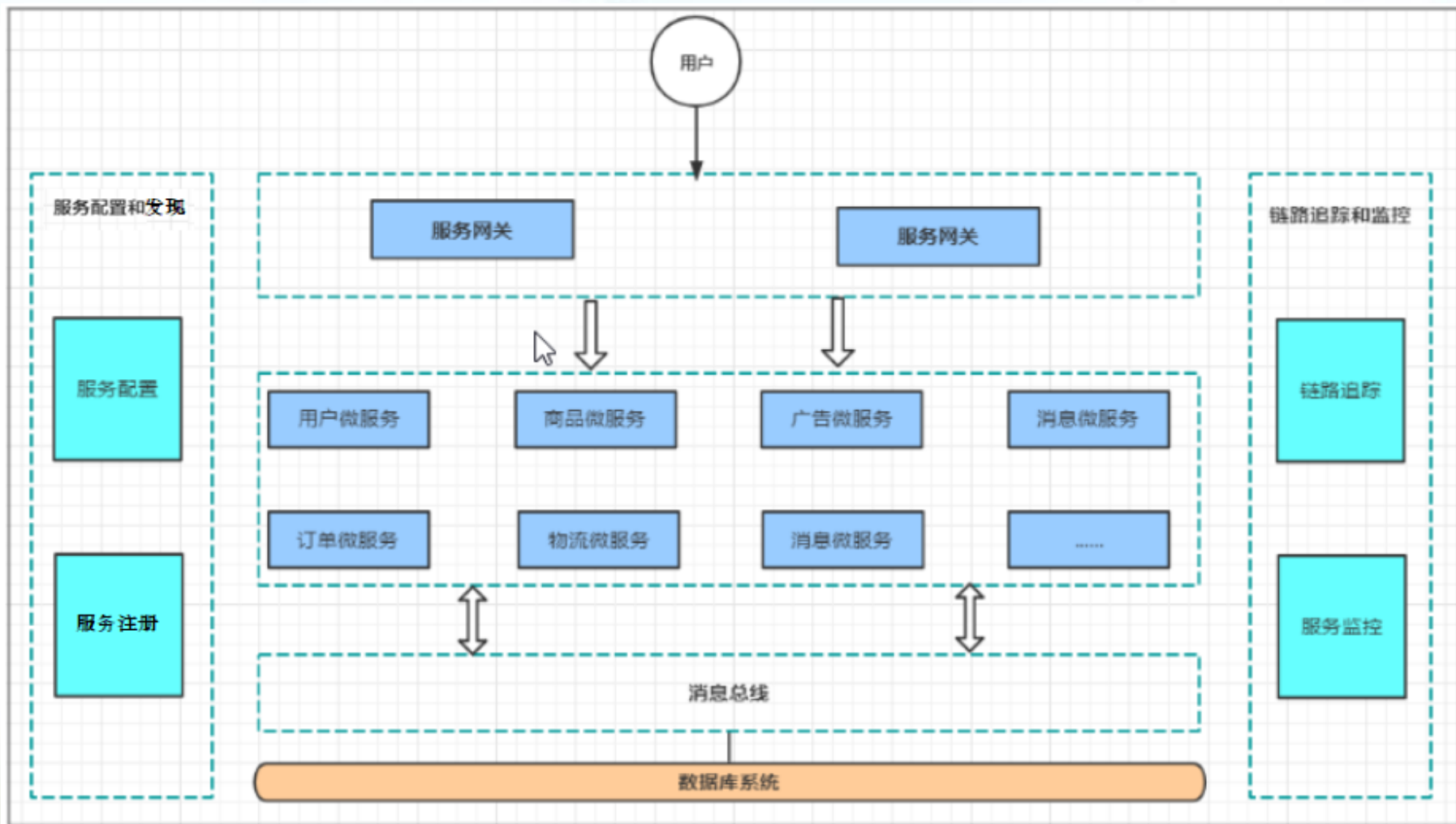
1.什么是微服务

■ 微服务是一种用于构建分布式应用的架构方案。

- ✧有别于更为传统的单体式方案，它将一个原本独立的系统拆分成了多个小型服务；
- ✧服务之间可以通过基于**HTTP/HTTPS**的**RESTful API**进行通信协作，也可以通过**RPC**协议进行通信协作；
- ✧每个功能都被称为一项服务，可以单独构建和部署；
- ✧由于使用轻量级通信协议作为基础，所以微服务可以使用不同语言来编写。



微服务架构



2.微服务架构的常见问题

■ 一旦采用微服务系统架构，就势必会遇到这样几个问题：

- ✧ 这么多小服务，如何管理他们？(服务治理)
- ✧ 这么多小服务，他们之间如何通讯？(服务调用)
- ✧ 这么多小服务，客户端怎么访问他们？(服务网关)
- ✧ 这么多小服务，一旦出现问题了，应该如何自处理？(服务容错)
- ✧ 这么多小服务，一旦出现问题了，应该如何排错？(链路追踪)

■ 对于上面的问题，是任何一个微服务设计者都不能绕过去的，因此大部分的微服务产品都针对每一个问题提供了相应的组件来解决它们。



3.微服务架构的常见概念

- 1) 服务治理
- 2) 服务调用
- 3) 服务网关
- 4) 服务容错
- 5) 链路追踪



1) 服务治理

■ 服务治理就是进行服务的自动化管理，具体包括：

- ✧ ①**服务注册与发现**：单体服务拆分为微服务后，需要将服务信息存储到某个载体（服务注册）；如果微服务之间存在调用依赖，需要得到目标服务的服务地址（服务发现）。
- ✧ ②**可观测性**：需要对众多服务间的调用关系、状态有清晰掌控，包括调用拓扑关系、监控（**Metrics**）、日志（**Logging**）、调用追踪（**Trace**）等。
- ✧ ③**流量管理**：在微服务版本更迭过程中，需要根据流量的特征（访问参数等）、百分比对微服务间调用进行控制，以完成微服务版本更迭的平滑（灰度发布、蓝绿发布、**A/B测试**）。
- ✧ ④**安全**：对于业务敏感的微服务，需要对其他服务的访问进行认证与鉴权。
- ✧ ⑤**控制**：能实时进行服务治理策略向微服务分发。



2) 服务调用

- 在微服务架构中，通常存在多个服务之间的远程调用的需求。
- 目前主流的远程调用技术有：
 - ✧ **REST(Representational State Transfer)**
 - 基于HTTP
 - 一种HTTP调用的格式，更标准，更通用，无论哪种语言都支持http协议
 - ✧ **RPC (Remote Promote Call)**
 - 基于TCP
 - 一种进程间通信方式，允许像调用本地服务一样调用远程服务



REST vs. RPC

比较项	RESTful	RPC
通讯协议	HTTP	一般使用TCP
性能	略低	较高
灵活度	高	低
应用	微服务架构	SOA架构



3) 服务网关

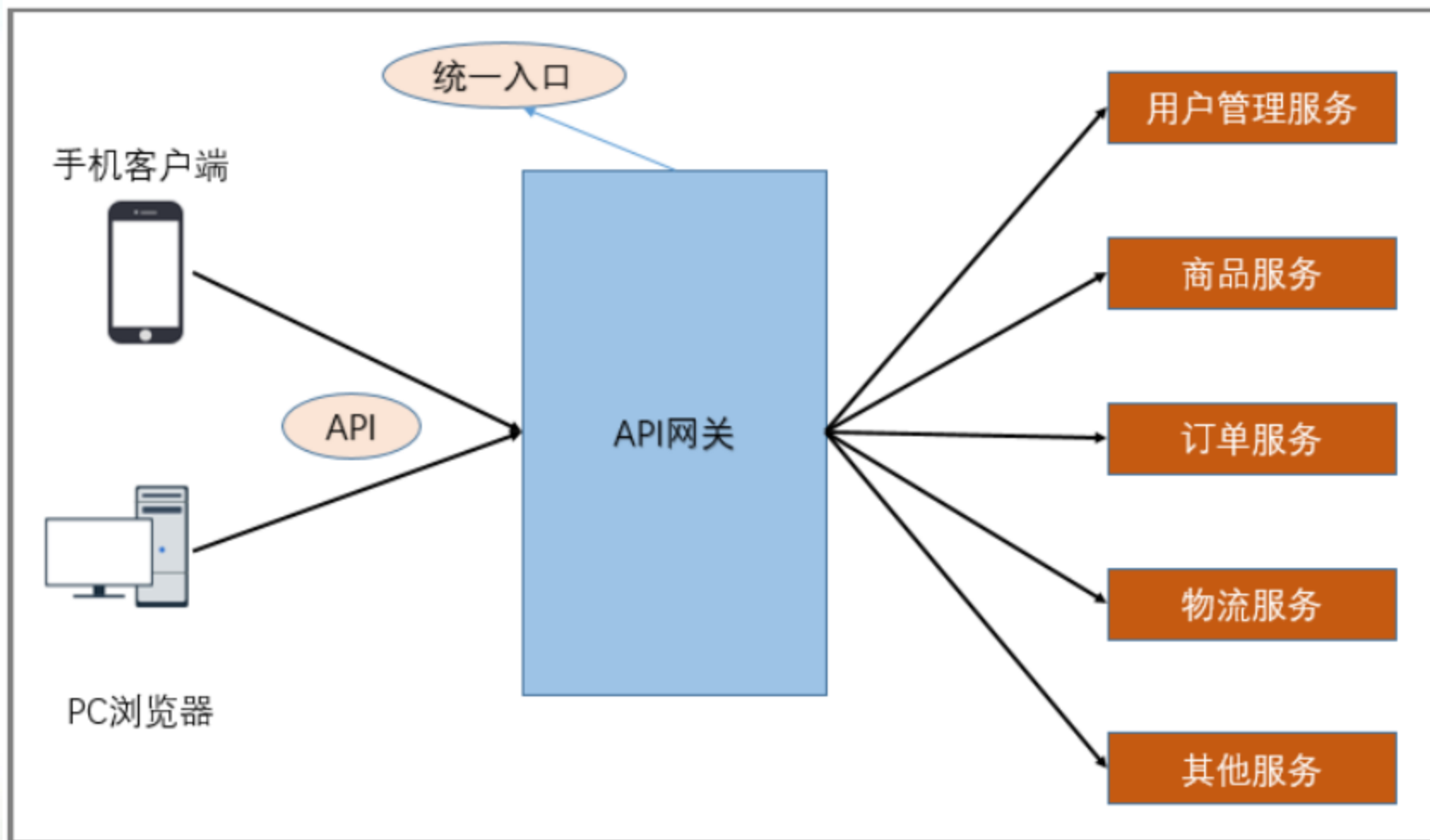
■ **API网关**字面意思是将所有**API**调用统一接入到**API网关层**，由网关层统一接入和输出。

■ 一个网关的基本功能有：

✧ 统一接入、安全防护、协议适配、流量管控、长短链接支持、容错能力。



API网关

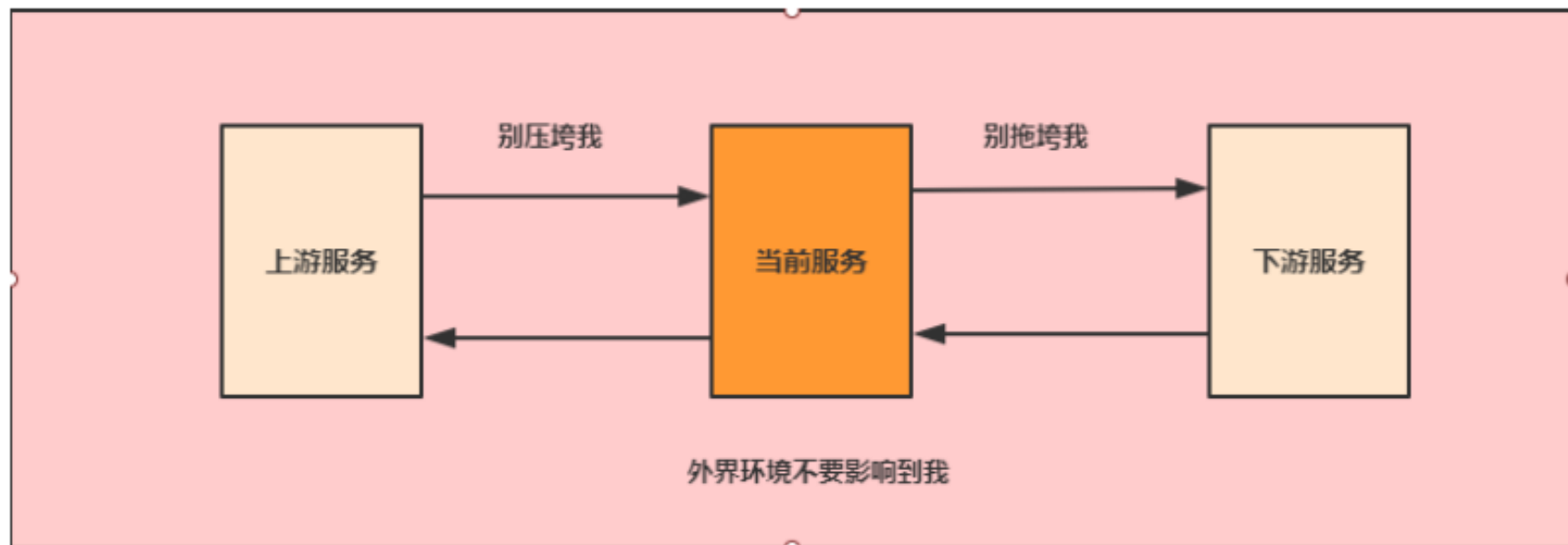


4) 服务容错

- 在微服务当中，一个请求经常会涉及到调用几个服务，如果其中某个服务不可用，没有做服务容错的话，极有可能会造成一连串的服务不可用，这就是雪崩效应。
- 服务容错的三个核心思想是：
 - ✧ 不被外界环境影响
 - ✧ 不被上游请求压垮
 - ✧ 不被下游响应拖垮



服务容错

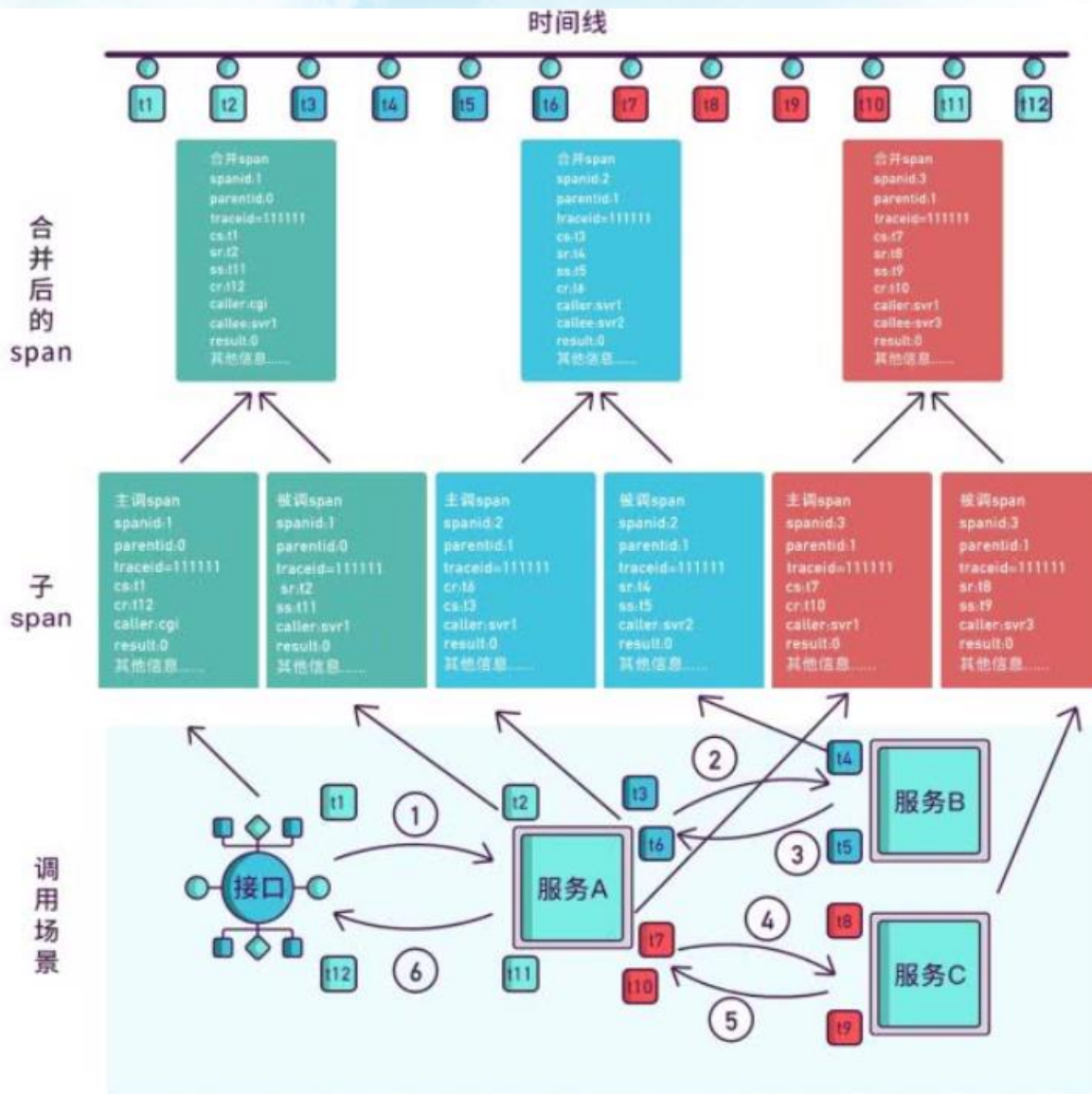


5) 链路追踪

- 随着微服务架构的流行，服务按照不同的维度进行拆分，一次请求往往需要涉及到多个服务。
- 因此，就需要对一次请求涉及的多个服务链路进行日志记录，性能监控即链路追踪。



链路追踪



4.微服务架构的常见解决方案

- 1) ServiceComb
- 2) Spring Cloud
- 3) Spring Cloud Alibaba



1) ServiceComb

- **Apache ServiceComb**，前身是华为云的微服务引擎 **CSE (Cloud Service Engine)** 云服务，是全球首个**Apache**微服务顶级项目。
- 它提供了一站式的微服务开源解决方案，致力于帮助企业、用户和开发者将企业应用轻松微服务化上云，并实现对微服务应用的高效运维管理。



2) Spring Cloud

■ **Spring Cloud**是一系列框架的集合。

✧它利用**Spring Boot**的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用**Spring Boot**的开发风格做到一键启动和部署。

✧**Spring Cloud**并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过**Spring Boot**风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

3) Spring Cloud Alibaba

■ **Spring Cloud Alibaba** 致力于提供微服务开发的一站式解决方案。

✧ 此项目包含开发分布式应用微服务的必需组件，方便开发者通过**Spring Cloud**编程模型轻松使用这些组件来开发分布式应用服务。

✧ 依托**Spring Cloud Alibaba**，您只需要添加一些注解和少量配置，就可以将**Spring Cloud**应用接入阿里微服务解决方案，通过阿里中间件来迅速搭建分布式应用系统。

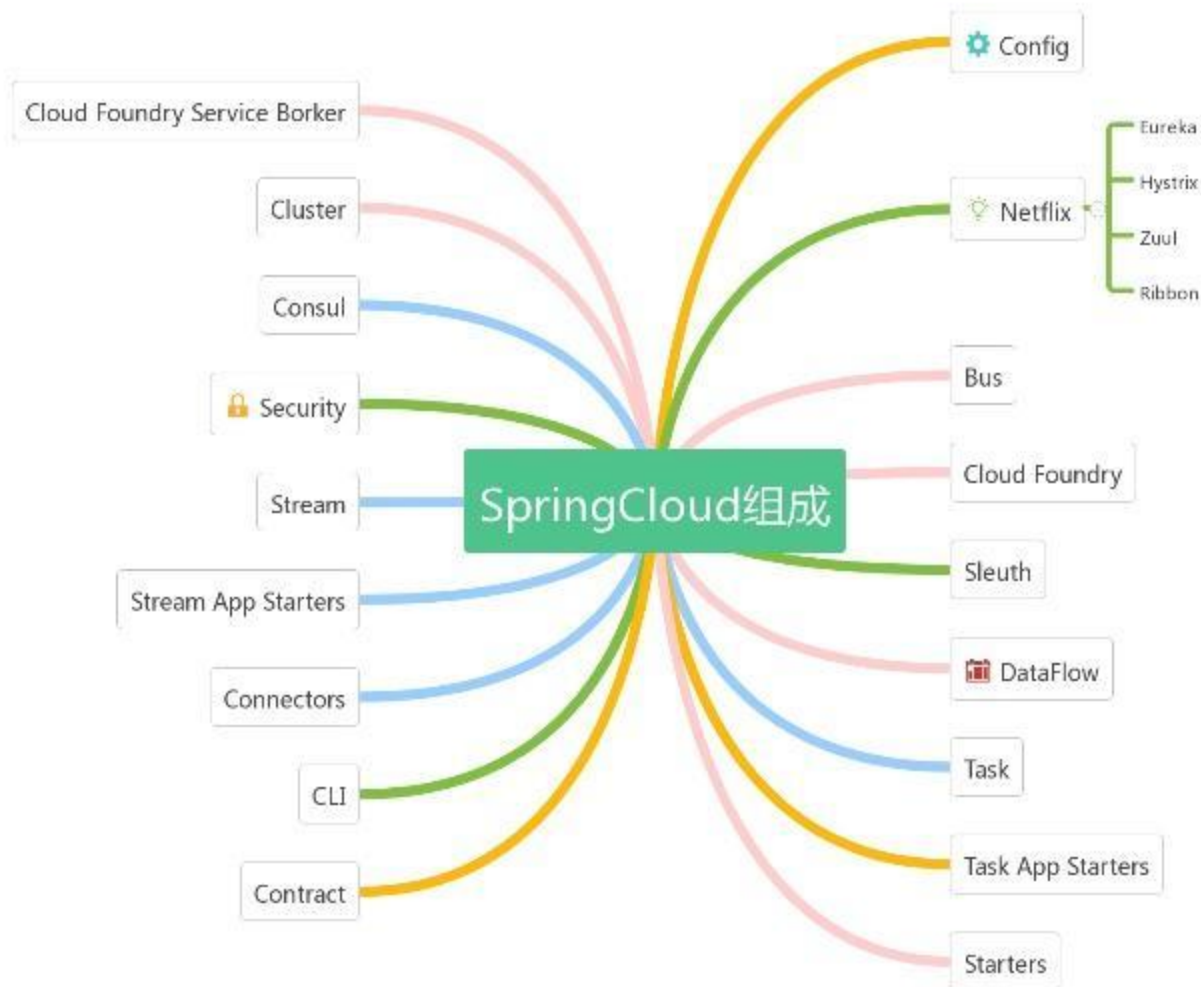


9.1.2 Spring Cloud简介

- 1) Spring Cloud组成
- 2) Spring Cloud常用组件



1) Spring Cloud组成



2) Spring Cloud常用组件

- 服务发现——Netflix Eureka
- 客户端负载均衡——Netflix Ribbon
- 断路器——Netflix Hystrix
- 服务网关——Netflix Zuul、Spring Cloud Gateway
- 分布式配置——Spring Cloud Config



9.1.3 Spring Cloud Alibaba简介

- 1) 主要功能
- 2) 主要组件



1) 主要功能

- **服务限流降级**: 默认支持 **WebServlet**、**WebFlux**、**OpenFeign**、**RestTemplate**、**Spring Cloud Gateway**、**Zuul**、**Dubbo**和**RocketMQ**限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 **Metrics** 监控。
- **服务注册与发现**: 适配 **Spring Cloud** 服务注册与发现标准，默认集成了 **Ribbon** 的支持。
- **分布式配置管理**: 支持分布式系统中的外部化配置，配置更改时自动刷新。
- **消息驱动能力**: 基于 **Spring Cloud Stream** 为微服务应用构建消息驱动能力。



主要功能

- **分布式事务**：使用 **@GlobalTransactional** 注解，高效并且对业务零侵入地解决分布式事务问题。
- **阿里云对象存储**：阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **分布式任务调度**：提供秒级、精准、高可靠、高可用的定时（基于 **Cron** 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 **Worker**（**schedulerx-client**）上执行。
- **阿里云短信服务**：覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

2) 主要组件

- **Sentinel**: 把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。
- **Nacos**: 一个更易于构建云原生应用的动态服务发现、配置管理和服务平台。
- **RocketMQ**: 一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。
- **Dubbo**: Apache Dubbo™ 是一款高性能 Java RPC 框架。
- **Seata**: 阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。



主要组件

- **Alibaba Cloud ACM:** 一款在分布式架构环境中对应用配置进行集中管理和推送的应用配置中心产品。
- **Alibaba Cloud OSS:** 阿里云对象存储服务（**Object Storage Service**，简称**OSS**），是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **Alibaba Cloud SchedulerX:** 阿里中间件团队开发的一款分布式任务调度产品，提供秒级、精准、高可靠、高可用的定时（基于**Cron**表达式）任务调度服务。
- **Alibaba Cloud SMS:** 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

9.2 微服务注册、配置和调用

■ 9.2.1 概述

■ 9.2.2 使用Nacos中间件



9.2.1 概述

- 1、服务注册和发现
- 2、服务配置
- 3、服务调用



1、服务注册和发现

■ 服务注册和发现是服务治理最核心最基本模块。

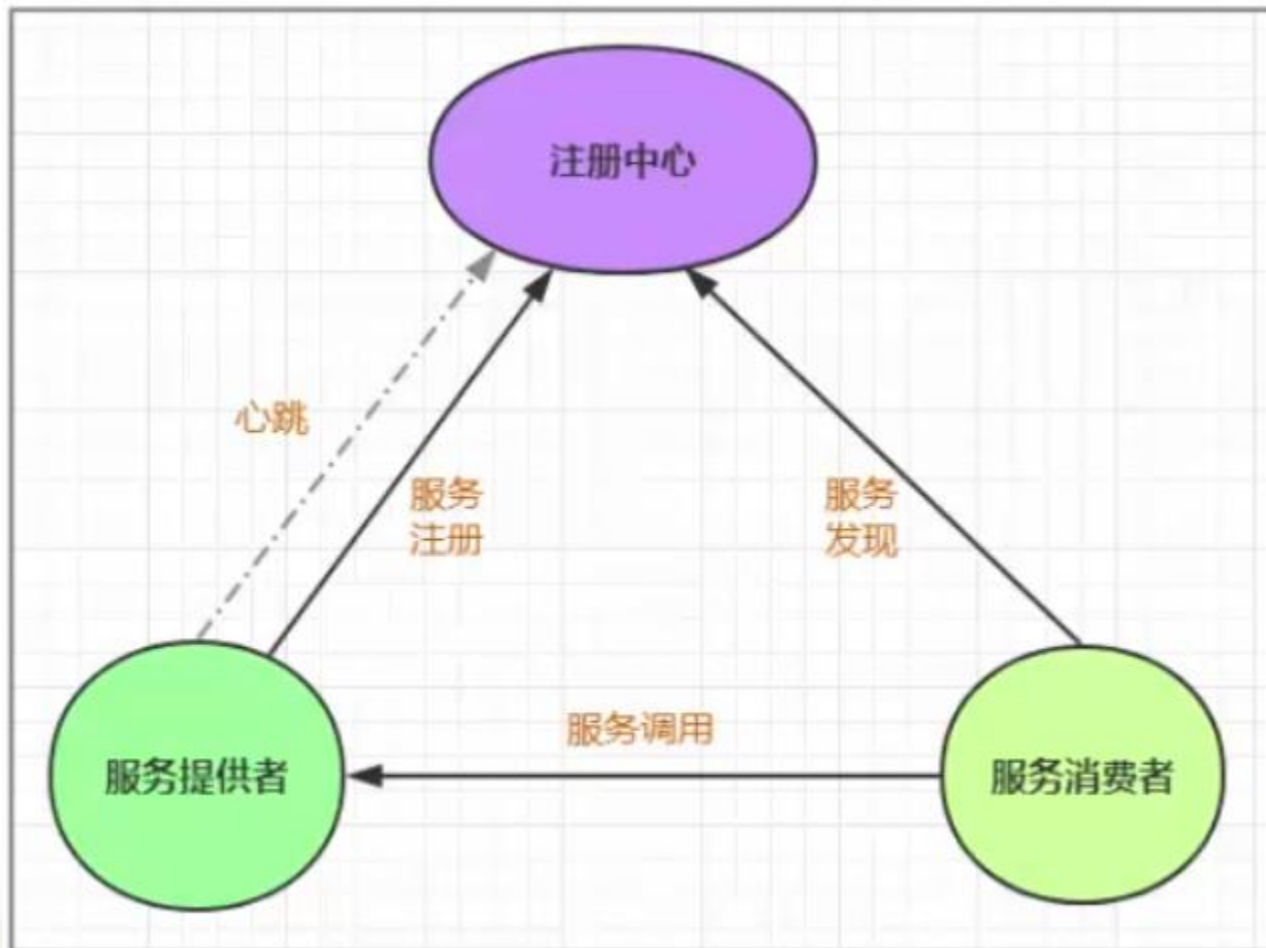
✧ 服务注册：在服务治理框架中，会构建一个**注册中心**，每个服务单元向注册中心登记自己提供服务的详细信息，并在注册中心形成一张服务的清单。

- 服务注册中心需要以心跳的方式去监测清单中服务是否可用，如果不可用，需要在服务清单中剔除不可用的服务。

✧ 服务发现：服务调用方向服务注册中心咨询服务，并获取所有服务实例清单，实现对具体实例的访问。



服务注册中心



服务注册中心

- 服务注册中心是微服务架构非常重要的一个组件，在微服务架构里主要起到了协调者的作用。
- 注册中心一般包含如下几个功能：
 - ◇ 服务发现：
 - 服务注册：保存服务提供者和服务调用者的信息
 - 服务订阅：服务调用者订阅服务提供者的信息，注册中心向订阅者推送提供者的信息
 - ◇ 服务配置：
 - 配置订阅：服务提供者和服务调用者订阅微服务相关的配置
 - 配置下发：主动将配置推送给服务提供者和服务调用者
 - ◇ 服务健康检测
 - 检测服务提供者的健康情况，如果发现异常，执行服务剔除



常见的服务注册中心

- ① Zookeeper
- ② Eureka
- ③ Consul
- ④ Nacos



①Zookeeper

■ **zookeeper**是一个分布式服务框架，是 **Apache Hadoop**的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。



②Eureka

■ Eureka是Springcloud Netflix中的重要组件，主要作用就是做服务注册和发现。但是现在已经闭源。



③ Consul

- **Consul**是基于**GO**语言开发的开源工具，主要面向分布式，服务化的系统提供服务注册、服务发现和配置管理的功能。
- **Consul**的功能都很实用，主要包括：
 - ✧ 服务注册/发现、健康检查、**Key/Value**存储、多数据中心和分布式一致性保证等特性。
- **Consul**本身只是一个二进制的可执行文件，所以安装和部署都非常简单，只需要从官网下载后，在执行对应的启动脚本即可。



④Nacos

- **Nacos**是一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。
- 它是 **Spring Cloud Alibaba** 组件之一，负责服务注册发现和服务配置，可以这样认为 **nacos=eureka+config**。



2、服务配置

■ 微服务架构下关于配置文件的一些问题：

✧ 配置文件相对分散。

- 在一个微服务架构下，配置文件会随着微服务的增多变的越来越多，而且分散在各个微服务中，不好统一配置和管理。

✧ 配置文件无法区分环境。

- 微服务项目可能会有多个环境，例如：测试环境、预发布环境、生产环境。每一个环境所使用的配置理论上都是不同的，一旦需要修改，就需要我们去各个微服务下手动维护，这比较困难。

✧ 配置文件无法实时更新。

- 我们修改了配置文件之后，必须重新启动微服务才能使配置生效，这对一个正在运行的项目来说是非常不友好的。

■ 基于上面这些问题，我们就需要配置中心的加入来解决这些问题。



配置中心的设计思路

■ 设计配置中心的思路是：

- ✧ 首先把项目中各种配置全部都放到一个集中的地方进行统一管理，并提供一套标准的接口。
- ✧ 当各个服务需要获取配置的时候，就来配置中心的接口拉取自己的配置。
- ✧ 当配置中心中的各种参数有更新的时候，也能通知到各个服务实时的过来同步最新的信息，使之动态更新。



常见的服务配置中心

- ①Apollo
- ②Disconf
- ③Spring Cloud Config
- ④Nacos



①Apollo

■ Apollo是由携程开源的分布式配置中心。

■ 特点：

- ✧ 配置更新之后可以实时生效；
- ✧ 支持灰度发布功能，并且能对所有的配置进行版本管理、操作审计等功能；
- ✧ 提供开放平台**API**，并且资料写的很详细。



②Disconf

- **Disconf**是由百度开源的分布式配置中心。
- 它是基于**Zookeeper**来实现配置变更后实时通知和生效的。



③Spring Cloud Config

- 这是**Spring Cloud**中带的配置中心组件。
- 它和**Spring**是无缝集成，使用起来非常方便，并且它的配置存储支持**Git**。
- 不过它没有可视化的操作界面，配置的生效也不是实时的，需要重启或去刷新。



④Nacos

- 这是**Sping Cloud alibaba**技术栈中的一个组件，它既是服务注册中心，也是服务配置中心。
- 作为服务配置中心具有以下功能：
 - ✧ 支持配置动态刷新
 - 只需要在需要动态读取配置类上添加**@RefreshScope**注解即可
 - ✧ 支持配置共享
 - 同一个微服务的不同环境（**dev**、**test**、**prod**）之间共享配置
 - 同组中不同微服务之间共享配置
 - ✧ 支持配置隔离
 - **Nacos**中有**Namespace**，**Namespace**下又有**Group**，**Group**下注册具体服务，不同隔离级别下的服务不可访问，做到沙箱隔离效果。



3、服务调用

■ 服务调用除了调用的协议选择问题外，还有调用的负载均衡问题。

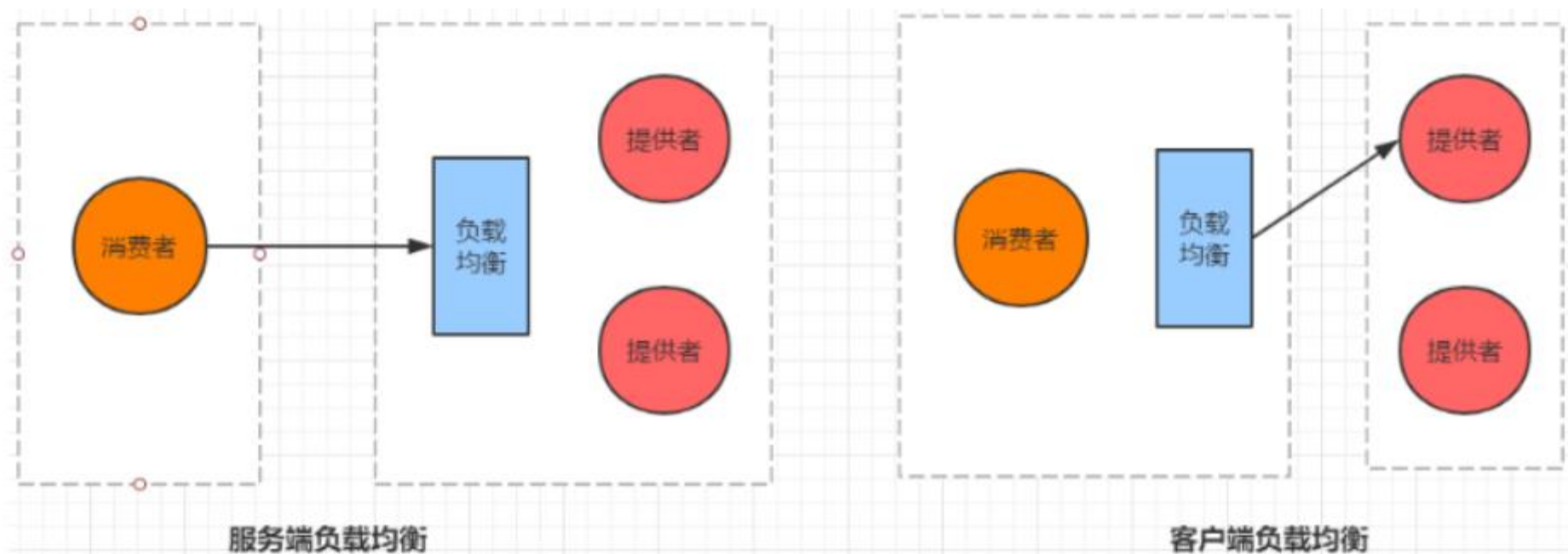
✧通俗的讲，**负载均衡**就是将负载（工作任务，访问请求）进行分摊到多个操作单元（服务器、组件）上进行执行。

✧根据负载均衡发生位置的不同，一般分为：

- 服务端负载均衡：指的是负载均衡机制在服务提供者一方，比如常见的nginx负载均衡
- 客户端负载均衡：指的是负载均衡机制在服务请求的一方，也就是在发送请求之前已经选好了由哪个实例处理请求

✧在微服务调用关系中，一般会选择客户端负载均衡，也就是在服务调用的一方来决定服务由哪个提供者执行。

负载均衡位置



基于Feign的负载均衡服务调用

■ Feign是Spring Cloud提供的一个声明式的伪Http客户端

- ✧ 调用远程服务就像调用本地服务一样，只需要创建一个调用服务的客户端接口，并在主类上添加一个@EnableFeignClients注解即可。
- ✧ Feign默认集成了Ribbon，因此就实现了负载均衡服务调用的效果。



关于Ribbon

■ **Ribbon**是**Spring Cloud**的一个组件， 它可以让使用一个注解就能轻松的搞定负载均衡

✧在**RestTemplate**的**Bean**的生成方法上添加**@LoadBalanced**注解

✧然后，通过**restTemplate**实例调用微服务。

```
@Bean
```

```
@LoadBalanced
```

```
public RestTemplate restTemplate() {
```

```
    return new RestTemplate();
```

```
}
```

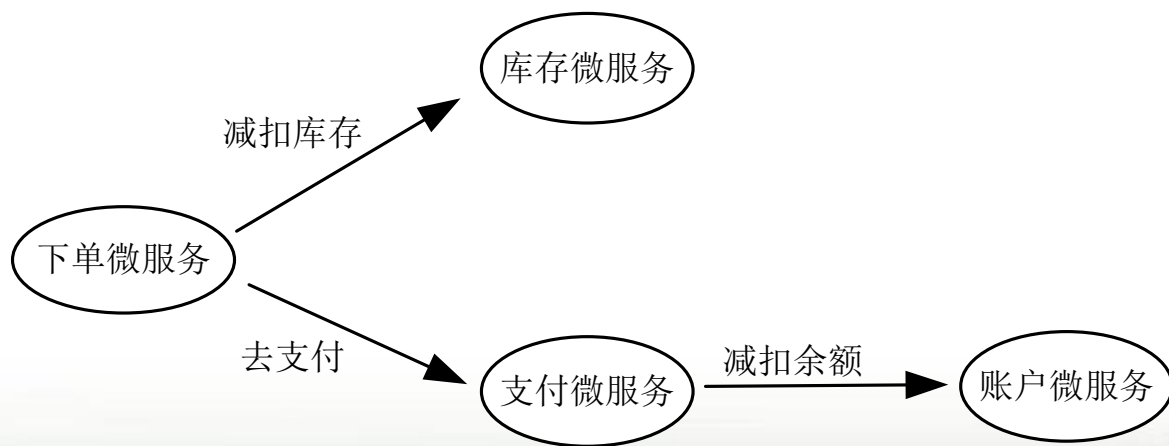
Ribbon支持的负载均衡策略

■ Ribbon内置了多种负载均衡策略，其顶级接口为**com.netflix.loadbalancer.IRule**。

策略名	策略描述	实现说明
BestAvailableRule	选择一个最小并发请求的server	逐个考察Server，如果Server被tripped了，则忽略，再选择其中ActiveRequestsCount最小的server
AvailabilityFilteringRule	过滤掉那些因为一直连接失败的被标记为circuit tripped的后端server，并过滤掉那些高并发的后端server（active connections 超过配置的阈值）	使用一个AvailabilityPredicate来包含过滤server的逻辑，其实就是检查status里记录的各个server的运行状态
WeightedResponseTimeRule	根据响应时间分配一个weight，响应时间越长，weight越小，被选中的可能性越低。	一个后台线程定期地从status里面读取评价响应时间，为每个server计算一个weight。Weight的计算也比较简单：responsetime减去每个server自己平均的responsetime就是server的权重。当刚开始运行，没有形成status时，使用roubine
RetryRule	对选定的负载均衡策略机上重试机制。	在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的
RoundRobinRule	轮询方式轮询选择server	轮询index，选择index对应位置的server
RandomRule	随机选择一个server	在index上随机，选择index对应位置的server
ZoneAvoidanceRule	复合判断server所在区域的性能和server的可用性选择server	使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。

9.2.2 使用Nacos中间件

- 本小节以Nacos为注册中心和配置中心，以Feign为负载均衡服务调用方，实践搭建一个电商项目微服务架构。
- 微服务及其调用关系如下：



搭建步骤

- 1) 搭建Nacos环境
- 2) 创建Spring Boot父项目
- 3) 创建、注册并配置下单微服务
- 4) 创建、注册并配置支付微服务
- 5) 创建、注册并配置其他微服务
- 6) 建立微服务间调用关系
- 7) 效果测试



1) 搭建Nacos环境

■ 搭建步骤:

- ✧ ① 安装Nacos中间件
- ✧ ② 启动Nacos
- ✧ ③ 访问Nacos



①安装Nacos中间件

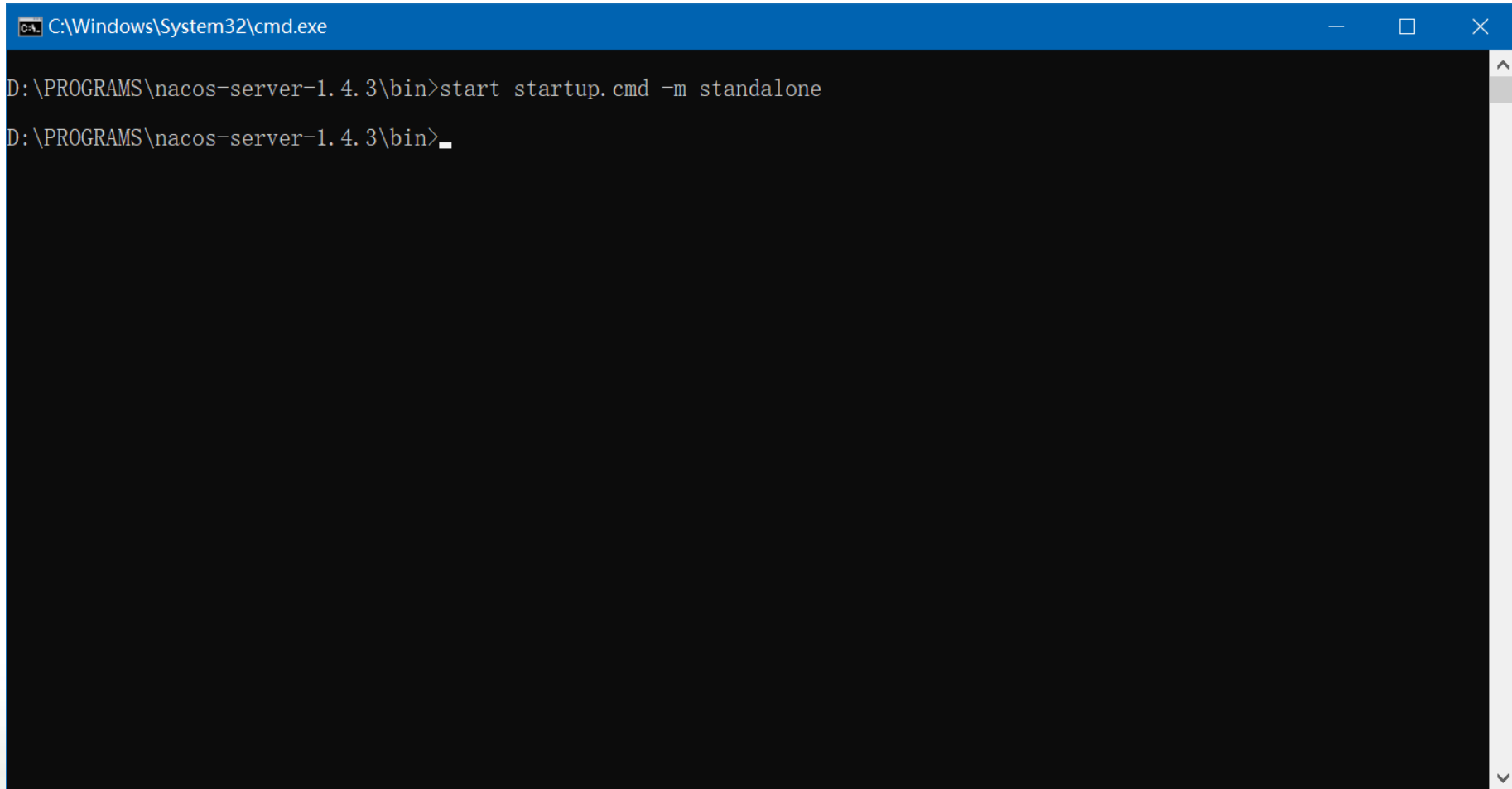
■ 下载地址:

<https://github.com/alibaba/nacos/releases>

■ 下载zip格式的安装包，然后进行解压缩操作即可



②启动Nacos



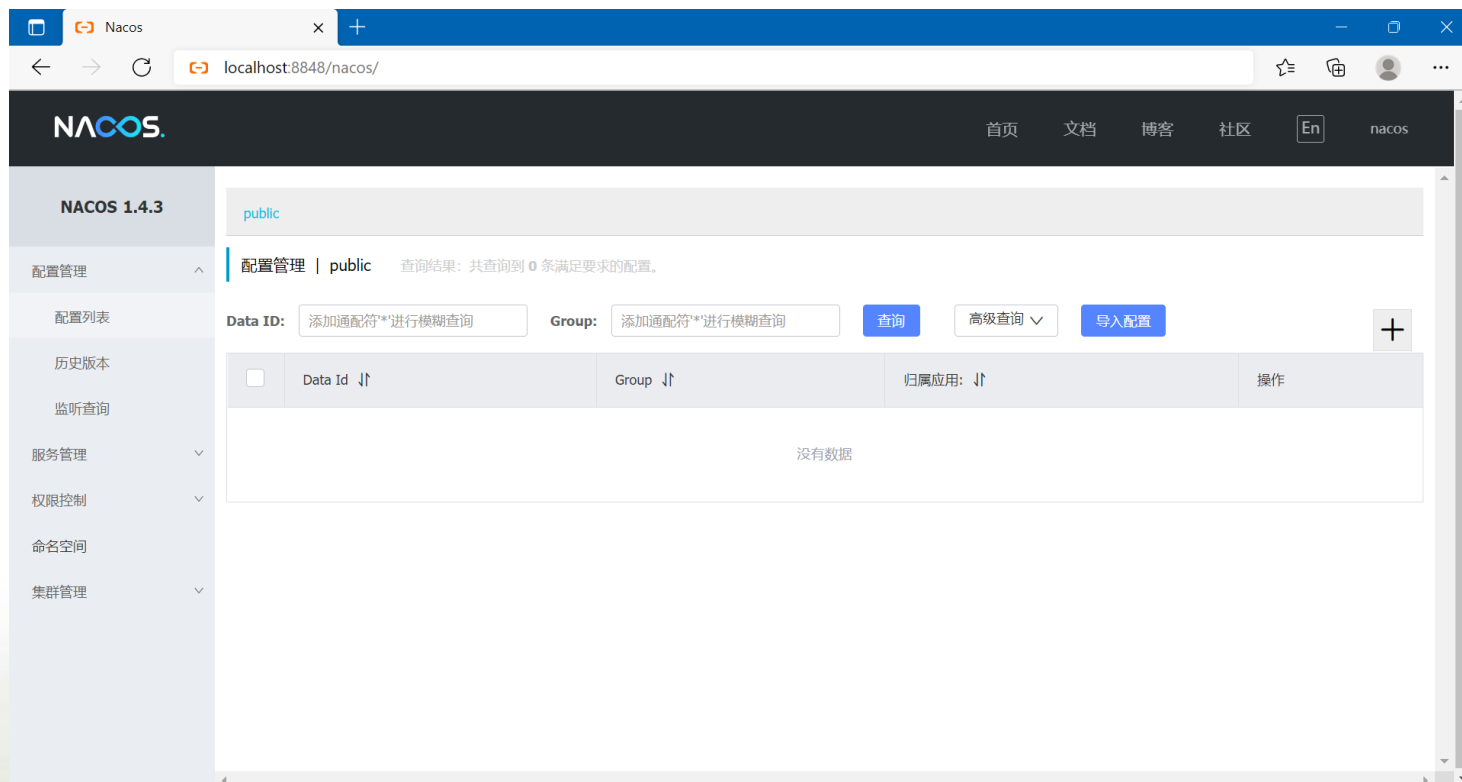
```
C:\Windows\System32\cmd.exe

D:\PROGRAMS\nacos-server-1.4.3\bin>start startup.cmd -m standalone

D:\PROGRAMS\nacos-server-1.4.3\bin>
```

③访问Nacos

■ 打开浏览器输入 **http://localhost:8848/nacos**，即可访问服务，默认密码是 **nacos/nacos**



2) 创建Spring Boot父项目

- 使用Spring Initializr方式创建一个Spring Boot项目chapter09，在Dependencies依赖选择中选择Spring Cloud Alibaba和开发工具模块中的相关依赖。

The image displays two screenshots of the Spring Initializr 'New Project' dialog. The left screenshot shows the 'Project Metadata' tab with the following fields: Group (com.scst), Artifact (chapter09), Type (Maven Project), Language (Java), Packaging (Jar), Java Version (8), Version (0.0.1-SNAPSHOT), Name (chapter09), Description (Demo project for Spring Boot), and Package (com.scst). The right screenshot shows the 'Dependencies' tab. The left pane lists categories: Alibaba Cloud, Spring Cloud Alibaba, 开发工具, Web, 模板引擎, 安全, 关系型数据库, 非关系型数据库, 消息, 输入/输出, Ops, 观测, 测试, 架构, 杂项, Spring Cloud, Spring Cloud Security, Spring Cloud Tools, Spring Cloud Config, Spring Cloud Discovery, and Spring Cloud Routing. The right pane shows a list of dependencies with checkboxes: Spring Cloud Alibaba Dubbo (unchecked), Nacos Service Discovery (checked), Spring Cloud Alibaba Sentinel (unchecked), Spring Cloud Alibaba Sentinel DataSource (unchecked), Spring Cloud Alibaba Sentinel Dubbo Adapter (unchecked), Spring Cloud Alibaba Sentinel Gateway (unchecked), Spring Cloud Alibaba Seata (unchecked), Spring Cloud Alibaba RocketMQ (unchecked), and Nacos Configuration (checked). The 'Selected Dependencies' pane on the right lists 'Spring Cloud Alibaba' (with sub-items Nacos Service Discovery and Nacos Configuration), '开发工具' (with sub-item Lombok), and 'Lombok'.

选择合适的版本组合

- 在pom.xml文件中，**Spring Boot**、**Spring Cloud Alibaba**和**Spring Cloud**要求版本对应

Spring Boot Version	Spring Cloud Alibaba Version	Spring Cloud Version
2.3.x.RELEASE	2.2.x.RELEASE	Hoxton.SR8
2.1.x.RELEASE	2.1.x.RELEASE	Greenwich
2.0.x.RELEASE	2.0.x.RELEASE	Finchley
1.5.x.RELEASE	1.5.x.RELEASE	Edgware

本项目版本组合

```
<properties>
```

```
  <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
```

```
  <spring-cloud-alibaba.version>2.2.2.RELEASE</spring-cloud-alibaba.version>
```

```
  <springcloud.version>Hoxton.SR8</springcloud.version>
```

```
</properties>
```



本项目版本组合

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

本项目版本组合

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



3) 创建、注册并配置下单微服务

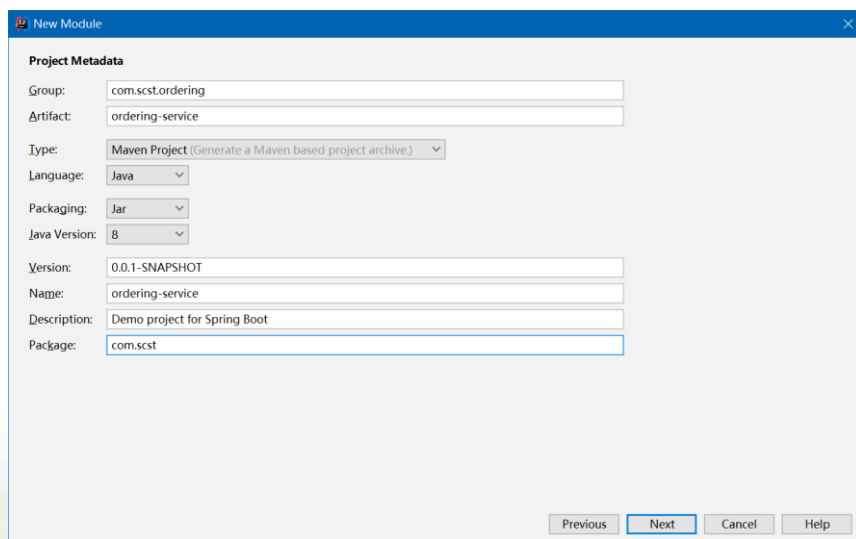
■ 搭建步骤:

- ✧①创建下单微服务模块
- ✧②配置下单微服务模块
- ✧③创建**Web**控制器类
- ✧④效果测试



①创建下单微服务模块

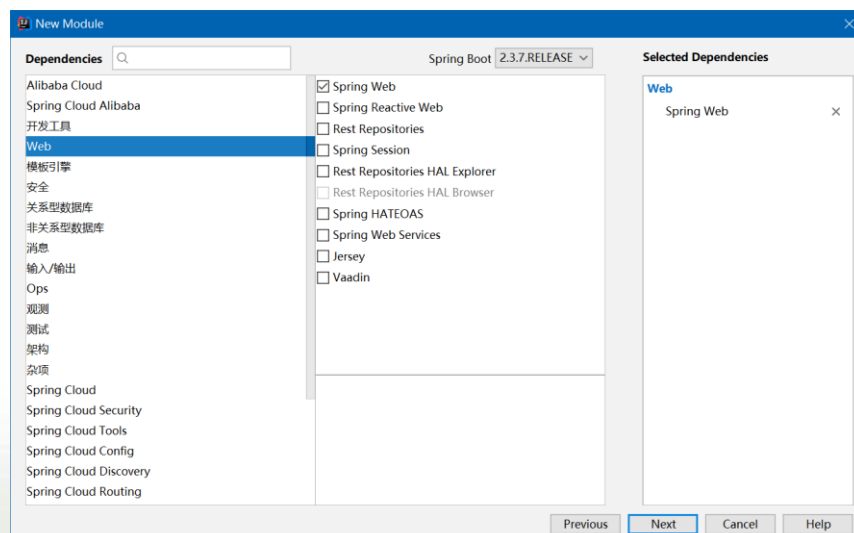
- 在chapter09项目中，使用Spring Initializr方式创建一个Spring Boot模块ordering-service，在Dependencies依赖选择中选择Web模块中的Spring Web依赖。



The 'New Module' dialog box is shown with the 'Project Metadata' tab selected. The fields are filled with the following information:

Field	Value
Group	com.scst.ordering
Artifact	ordering-service
Type	Maven Project (Generate a Maven based project archive.)
Language	Java
Packaging	Jar
Java Version	8
Version	0.0.1-SNAPSHOT
Name	ordering-service
Description	Demo project for Spring Boot
Package	com.scst

At the bottom, there are buttons for 'Previous', 'Next', 'Cancel', and 'Help'.



The 'New Module' dialog box is shown with the 'Dependencies' tab selected. The 'Spring Boot' version is set to '2.3.7.RELEASE'. The 'Web' dependency is selected in the list, and its sub-dependencies are checked in the 'Selected Dependencies' panel.

Dependencies	Selected Dependencies
<input checked="" type="checkbox"/> Spring Web	<input checked="" type="checkbox"/> Spring Web
<input type="checkbox"/> Spring Reactive Web	
<input type="checkbox"/> Rest Repositories	
<input type="checkbox"/> Spring Session	
<input type="checkbox"/> Rest Repositories HAL Explorer	
<input type="checkbox"/> Rest Repositories HAL Browser	
<input type="checkbox"/> Spring HATEOAS	
<input type="checkbox"/> Spring Web Services	
<input type="checkbox"/> Jersey	
<input type="checkbox"/> Vaadin	

At the bottom, there are buttons for 'Previous', 'Next', 'Cancel', and 'Help'.

建立父项目依赖

- 在**ordering-service**模块的**pom.xml**文件中引入对父项目**chapter09**的依赖，示例代码如下。

```
<parent>  
  <artifactId>chapter09</artifactId>  
  <groupId>com.scst</groupId>  
  <version>0.0.1-SNAPSHOT</version>  
</parent>
```



②配置下单微服务模块

■ 本模块在Nacos中创建以下配置文件：

- ✧ 开发环境（**dev**）下的配置文件**ordering-service-dev.properties**
- ✧ 测试环境（**test**）下的配置文件**ordering-service-test.properties**
- ✧ 不同环境共享的配置文件**ordering-service.properties**
- ✧ 不同服务共享的配置文件**common.properties**

■ 模块要使用上述配置还需在模块的**resources**目录下创建**bootstrap.properties**配置文件，并注解掉**application.properties**文件中的配置。

- ✧ 配置文件优先级(由高到低):**bootstrap.properties -> bootstrap.yml -> application.properties -> application.yml**



Nacos中创建配置文件

public

配置管理 | public 查询结果: 共查询到 0 条满足要求的配置。

Data ID: Group:

+

<input type="checkbox"/>	Data Id ↓↑	Group ↓↑	归属应用: ↓↑	操作
--------------------------	------------	----------	----------	----

没有数据



ordering-service-dev.properties

新建配置

* Data ID:

* Group:

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☐ YAML ☐ HTML ☒ Properties

* 配置内容:



```
1 # 应用服务 WEB 访问端口; 如果不存在JVM参数port, 则默认使用9010
2 server.port=${port:9010}
```

ordering-service-test.properties

新建配置

* Data ID: ordering-service-test.properties

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☐ YAML ☐ HTML ☒ Properties

* 配置内容:

 :

```
1 # 应用服务 WEB 访问端口; 如果不存在JVM参数port, 则默认使用9011
2 server.port=${port:9011}
```



ordering-service.properties

新建配置

* Data ID: ordering-service.properties

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☐ YAML ☐ HTML ☒ Properties

* 配置内容:

```
1 #这是同一个微服务的不同环境之间共享配置
2 config.serviceshared=This is the shared config for the same service.
```



common.properties

新建配置

* Data ID: common.properties

* Group: DEFAULT_GROUP

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☐ YAML ☐ HTML ☒ Properties

* 配置内容:



```
1 #这是同组中所有微服务的共享配置
2 config.allshared=This is the shared config for all services.
```



bootstrap.properties

应用名称

spring.application.name=ordering-service

#激活的环境： dev或test

spring.profiles.active=dev

spring.cloud.nacos.config.file-extension=properties

spring.cloud.nacos.config.server-addr=localhost:8848

#定义同组中所有微服务的共享配置； [n]的值越大， 优先级越高

spring.cloud.nacos.config.extension-configs[0].data-id=common.properties

spring.cloud.nacos.config.extension-configs[0].refresh=true

#以下是默认配置（public的命名空间ID和DEFAULT_GROUP）

spring.cloud.nacos.config.namespace=

spring.cloud.nacos.config.group=DEFAULT_GROUP

Nacos中创建的配置文件

public

配置管理 | public 查询结果: 共查询到 4 条满足要求的配置。

Data ID: Group: 查询 高级查询 ▾ 导入配置

+

<input type="checkbox"/>	Data Id ▴▾	Group ▴▾	归属应用: ▴▾	操作
<input type="checkbox"/>	common.properties	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	ordering-service-dev.properties	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	ordering-service-test.properties	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	ordering-service.properties	DEFAULT_GROUP		详情 示例代码 编辑 删除 更多

删除 克隆 导出 ▾

每页显示: ▾

< 上一页

1

下一页 >



③创建Web控制器类

- 在ordering-service模块中新建一个com.scst.controller包，并在包中新建一个Web控制类OrderingController，并在该类中实现微服务配置值的访问。



OrderingController类

```
package com.scst.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cloud.context.config.annotation.RefreshScope;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/ordering")
@RefreshScope
public class OrderingController {

    @Autowired
    private Environment environment;
```



OrderingController类

```
@GetMapping("/configs")  
public String getConfigInfo() {  
    String serverPort=environment.getProperty("server.port");  
    //同一个微服务不同环境之间的共享配置  
    String serviceShared=environment.getProperty("config.serviceshared");  
    //同组中所有微服务的共享配置  
    String allShared=environment.getProperty("config.allshared");  
    return "服务端口: "+serverPort+" 共享配置: "+serviceShared+" / "+allShared;  
}  
}
```



④效果测试

- 先后在**dev**和**test**环境下启动**ordering-service**模块，观察**Nacos**中的服务实例。
- 在浏览器中先后输入：
<http://localhost:9010/ordering/configs>和
<http://localhost:9011/ordering/configs>，访问服务的配置值。



测试结果

public

服务列表 | public

服务名称

请输入服务名称

分组名称

请输入分组名称

隐藏空服务:



查询

创建服务

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
ordering-service	DEFAULT_GROUP	1	2	2	false	详情 示例代码 订阅者 删除

每页显示:

10



< 上一页

1

下一页 >

集群: DEFAULT

集群配置

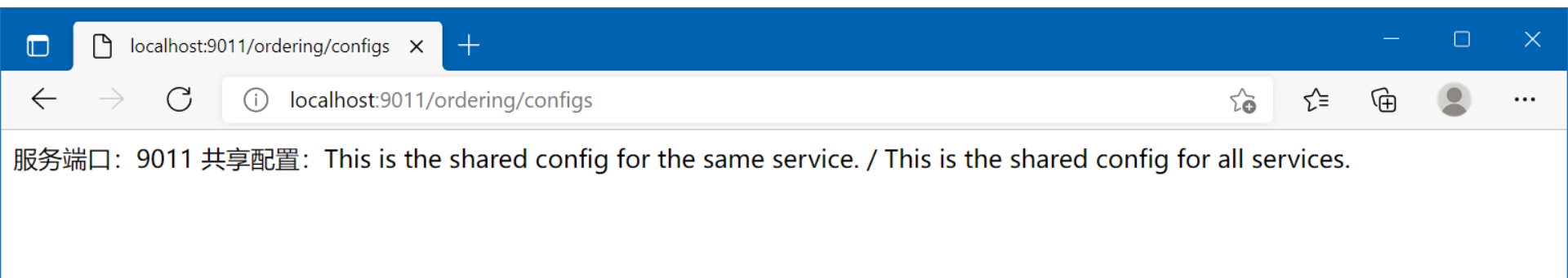
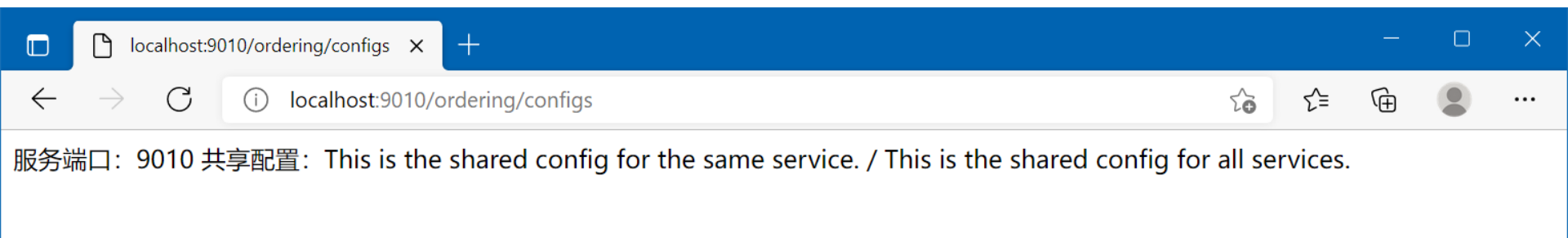
key

value



IP	端口	临时实例	权重	健康状态	元数据	操作
192.168.31.103	9010	true	1	true	preserved.register.source=SPRING_CLOUD	<button>编辑</button> <button>下线</button>
192.168.31.103	9011	true	1	true	preserved.register.source=SPRING_CLOUD	<button>编辑</button> <button>下线</button>

测试结果



4) 创建、注册并配置支付微服务

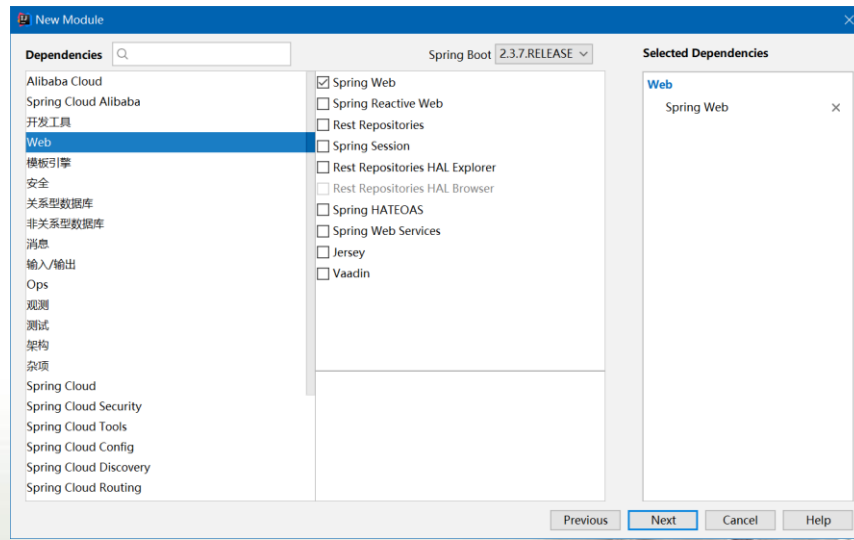
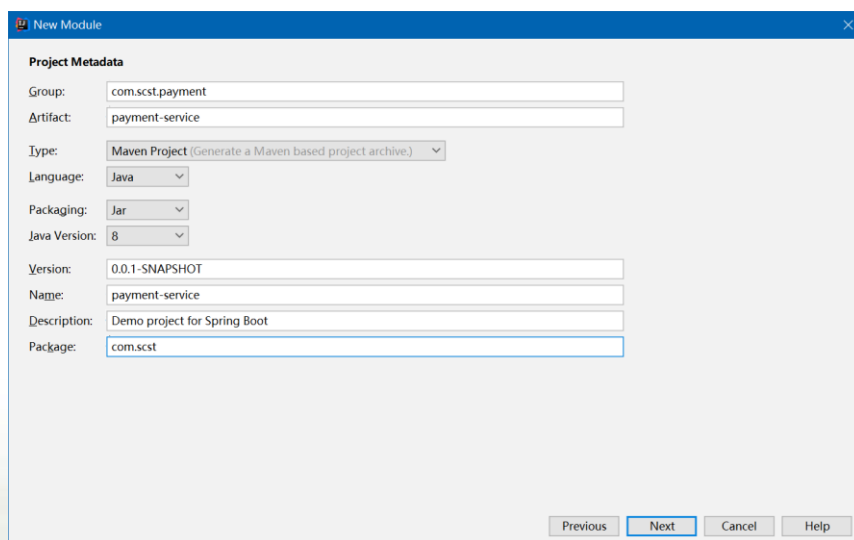
■ 搭建步骤:

- ✧①创建支付微服务模块
- ✧②配置支付微服务模块



①创建支付微服务模块

- 在chapter09项目中，使用Spring Initializr方式创建一个Spring Boot模块payment-service，在Dependencies依赖选择中选择Web模块中的Spring Web依赖。



建立父项目依赖

- 在**payment-service**模块的**pom.xml**文件中引入对父项目**chapter09**的依赖，示例代码如下。

```
<parent>
  <artifactId>chapter09</artifactId>
  <groupId>com.scst</groupId>
  <version>0.0.1-SNAPSHOT</version>
</parent>
```



②配置支付微服务模块

■ 在模块**resources**目录下的
application.properties文件中配置如下：

```
# 应用名称
spring.application.name=payment-service
# 应用服务 WEB 访问端口；如果不存在JVM参数port，则默认使用9020
server.port=${port:9020}
#配置Nacos地址
spring.cloud.nacos.discovery.server-addr=localhost:8848
```



5) 创建、注册并配置其他微服务

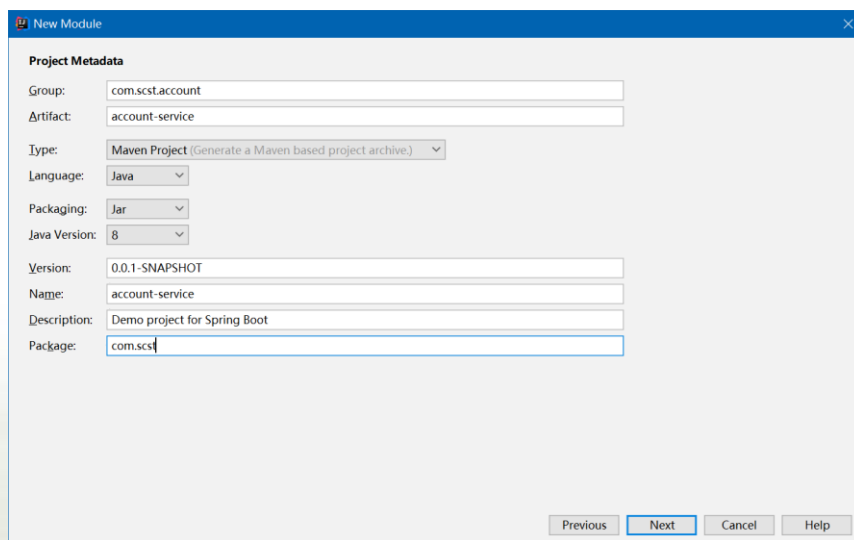
■ 搭建步骤:

- ✧①创建其他微服务模块
- ✧②配置其他微服务模块



①创建其他微服务模块

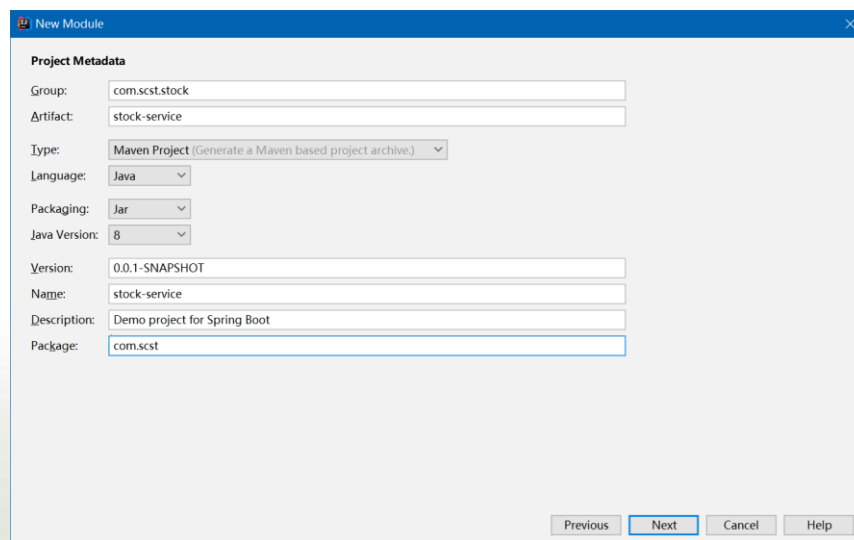
- 按照创建支付微服务模块的相同方式，在chapter09项目中，分别创建以下Spring Boot模块：**account-service**和**stock-service**，在**Dependencies**依赖选择中也都选择**Web**模块中的**Spring Web**依赖。



The screenshot shows the 'New Module' dialog box in IntelliJ IDEA. The 'Project Metadata' section contains the following fields:

- Group: com.scst.account
- Artifact: account-service
- Type: Maven Project (Generate a Maven based project archive.)
- Language: Java
- Packaging: Jar
- Java Version: 8
- Version: 0.0.1-SNAPSHOT
- Name: account-service
- Description: Demo project for Spring Boot
- Package: com.scst

At the bottom, there are buttons for 'Previous', 'Next', 'Cancel', and 'Help'.



The screenshot shows the 'New Module' dialog box in IntelliJ IDEA. The 'Project Metadata' section contains the following fields:

- Group: com.scst.stock
- Artifact: stock-service
- Type: Maven Project (Generate a Maven based project archive.)
- Language: Java
- Packaging: Jar
- Java Version: 8
- Version: 0.0.1-SNAPSHOT
- Name: stock-service
- Description: Demo project for Spring Boot
- Package: com.scst

At the bottom, there are buttons for 'Previous', 'Next', 'Cancel', and 'Help'.

建立父项目依赖

- 在**account-service**和**stock-service**模块的**pom.xml**文件中都引入对父项目**chapter09**的依赖，示例代码如下。

```
<parent>  
  <artifactId>chapter09</artifactId>  
  <groupId>com.scst</groupId>  
  <version>0.0.1-SNAPSHOT</version>  
</parent>
```



②配置其他微服务模块

- 按照配置支付微服务模块的相同方式，在 **account-service** 和 **stock-service** 模块的 **resources** 目录下的 **application.properties** 文件中分别配置如下。



其他微服务模块的配置

应用名称

spring.application.name=account-service

应用服务 WEB 访问端口；如果不存在JVM参数port，则默认使用9030

server.port=\${port:9030}

#配置Nacos地址

spring.cloud.nacos.discovery.server-addr=localhost:8848

应用名称

spring.application.name=stock-service

应用服务 WEB 访问端口；如果不存在JVM参数port，则默认使用9040

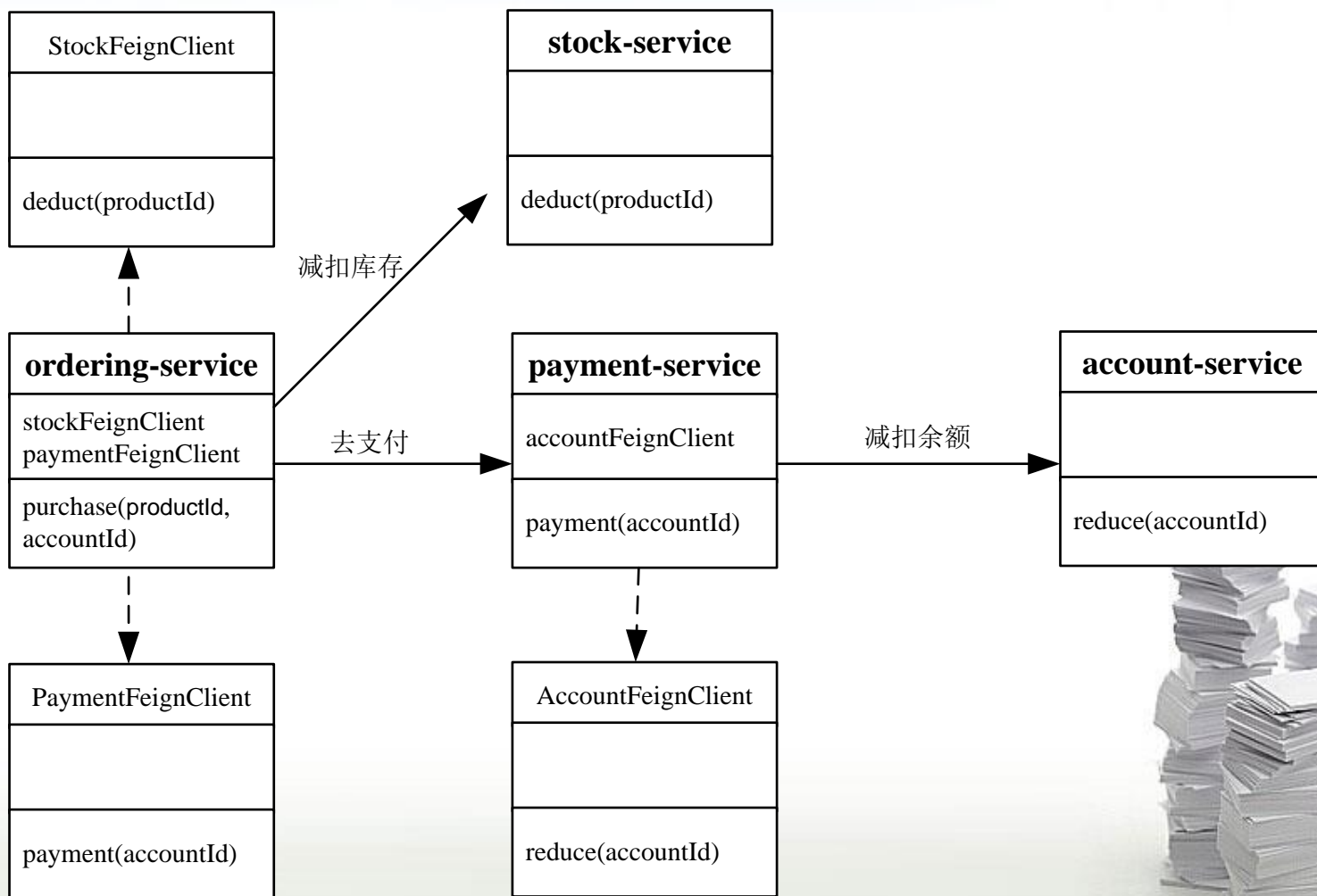
server.port=\${port:9040}

#配置Nacos地址

spring.cloud.nacos.discovery.server-addr=localhost:8848



6) 建立微服务间调用关系



服务调用客户端

■ 按照项目需求，需创建以下负载均衡微服务调用客户端，并在调用的客户端主类上添加 **@EnableFeignClients** 注解：

✧ **PaymentFeignClient**：调用 **payment-service**

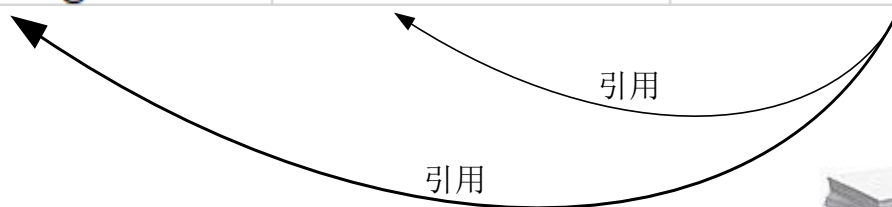
✧ **StockFeignClient**：调用 **stock-service**

✧ **AccountFeignClient**：调用 **account-service**



各微服务模块需创建的类

微服务	FeignClient	业务类	Web控制器类
①stock-service	-	StockService	StockController
②account-service	-	AccountService	AccountController
③payment-service	AccountFeignClient	PaymentService	PaymentController
④ordering-service	PaymentFeignClient 、 StockFeignClient	OrderingService	OrderingController



Feign依赖

- 同时在payment-service和ordering-service模块需引入Feign依赖。

```
<!--open feign-->
<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>
```



①stock-service模块中的类

- StockService类
- StockController类



StockService类

```
package com.scst.service;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class StockService {  
    public boolean deduct(String productId){  
        return true;  
    }  
}
```



StockController类

```
package com.scst.controller;

import com.scst.service.StockService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/stock")
public class StockController {

    @Autowired
    private StockService stockService;
```

StockController类

```
@GetMapping("/deduct/{id}")  
public String deduct(@PathVariable("id") String productId) {  
    boolean flag = stockService.deduct(productId);  
    return "商品号: "+productId+", 库存减扣成功! ";  
}  
}
```



②account-service模块中的类

- AccountService类
- AccountController类



AccountService类

```
package com.scst.service;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class AccountService {  
    public boolean reduce(String accountId){  
        return true;  
    }  
}
```



AccountController类

```
package com.scst.controller;

import com.scst.service.AccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/account")
public class AccountController {

    @Autowired
    private AccountService accountService;
```


AccountController类

```
@GetMapping("/reduce/{id}")
public String reduce(@PathVariable("id") String accountId) {
    boolean flag = accountService.reduce(accountId);
    return "账号: "+accountId+", 扣款成功!";
}
}
```



③payment-service模块中的类

- AccountFeignClient接口
- PaymentService类
- PaymentController类



AccountFeignClient接口

```
package com.scst.feignclient;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
//调用account-service微服务客户端
@FeignClient(name = "account-service")
public interface AccountFeignClient {
    //调用reduce方法:http://[account-service address]/account/reduce/{id}
    @GetMapping("/account/reduce/{id}")
    public String reduce(@PathVariable("id") String accountId);
}
```



PaymentService类

```
package com.scst.service;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class PaymentService {  
    public boolean pay(String accountId){  
        return true;  
    }  
}
```



PaymentController类

```
package com.scst.controller;

import com.scst.feignclient.AccountFeignClient;
import com.scst.service.PaymentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/payment")
public class PaymentController {
```

PaymentController类

```
@Autowired
```

```
private Environment environment;
```

```
@Autowired
```

```
private AccountFeignClient accountFeignClient;
```

```
@Autowired
```

```
private PaymentService paymentService;
```

```
@GetMapping("/pay/{id}")
```

```
public String pay(@PathVariable("id") String accountId) {
```

```
    String payResult=accountFeignClient.reduce(accountId);
```

```
    boolean flag = paymentService.pay(accountId);
```

```
    String serverPort=environment.getProperty("server.port");
```

```
    return payResult+" 支付服务端口: "+serverPort;
```

```
}
```

```
}
```


④ordering-service模块中的类

- PaymentFeignClient接口
- StockFeignClient接口
- OrderingService类
- OrderingController类



PaymentFeignClient接口

```
package com.scst.feignclient;  
  
import org.springframework.cloud.openfeign.FeignClient;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
//调用payment-service微服务客户端  
  
@FeignClient(name = "payment-service")  
public interface PaymentFeignClient {  
    //调用pay方法:http://[payment-service address]/payment/pay/{id}  
    @GetMapping("/payment/pay/{id}")  
    public String pay(@PathVariable("id") String accountId);  
}
```



StockFeignClient接口

```
package com.scst.feignclient;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
//调用stock-service微服务客户端
@FeignClient(name = "stock-service")
public interface StockFeignClient {
    //调用deduct方法:http://[stock-service address]/stock/deduct/{id}
    @GetMapping("/stock/deduct/{id}")
    public String deduct(@PathVariable("id") String productId);
}
```



OrderingService类

```
package com.scst.service;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class OrderingService {  
    public boolean purchase(String productId,String accountId){  
        return true;  
    }  
}
```



OrderingController类

```
package com.scst.controller;  
  
import com.scst.feignclient.PaymentFeignClient;  
import com.scst.feignclient.StockFeignClient;  
import com.scst.service.OrderingService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cloud.context.config.annotation.RefreshScope;  
import org.springframework.core.env.Environment;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;
```



OrderingController类

```
@RestController
@RequestMapping("/ordering")
@RefreshScope
public class OrderingController {
    @Autowired
    private Environment environment;

    @Autowired
    private PaymentFeignClient paymentFeignClient;

    @Autowired
    private StockFeignClient stockFeignClient;

    @Autowired
    private OrderingService orderingService;
```


OrderingController类

```
@GetMapping("/configs")  
public String getConfigInfo() {  
    String serverPort=environment.getProperty("server.port");  
    //同一个微服务不同环境之间的共享配置  
    String serviceShared=environment.getProperty("config.serviceshared");  
    //同组中所有微服务的共享配置  
    String allShared=environment.getProperty("config.allshared");  
    return "服务端口: "+serverPort+" 共享配置: "+serviceShared+" / "+allShared;  
}
```



OrderingController类

```
@GetMapping("/purchase/{pid}/{aid}")  
public String purchase(@PathVariable("pid") String productId, @PathVariable("aid") String  
accountId) {  
    String deductResult = stockFeignClient.deduct(productId);  
    String payResult = paymentFeignClient.pay(accountId);  
    boolean flag = orderingService.purchase(productId, accountId);  
    return deductResult+payResult;  
}  
}
```



7) 效果测试

- 启动Nacos，然后启动上述4个微服务，其中payment-service启动三个实例。
- 在浏览器中输入：
<http://localhost:9010/ordering/purchase/100/111>，并不断刷新网页，观察服务调用和负载均衡的输出效果。



测试结果

public

服务列表 | public

服务名称

分组名称

隐藏空服务:



查询

创建服务

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
account-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
ordering-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
payment-service	DEFAULT_GROUP	1	3	3	false	详情 示例代码 订阅者 删除
stock-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除

每页显示:

10 ▼

< 上一页

1

下一页 >



测试结果

