

# 第十一章 微服务

赖永炫 博士/教授  
厦门大学 软件工程系

# 前言

---

- **微服务**是近年来出现的一种新型的架构风格，它提倡将应用程序划分为一组细粒度的服务，服务间采用轻量级的通信机制进行交互。
- 在微服务架构中，每个微服务都是具有单一职责的小程序，能够被独立地部署、扩展和测试。通过将这些独立的服务进行组合可以完成一些复杂的业务。
- 本章介绍了微服务的概念、架构体系、流行框架以及适用场景等，特别是较为详细说明了**Spring Cloud**微服务架构。

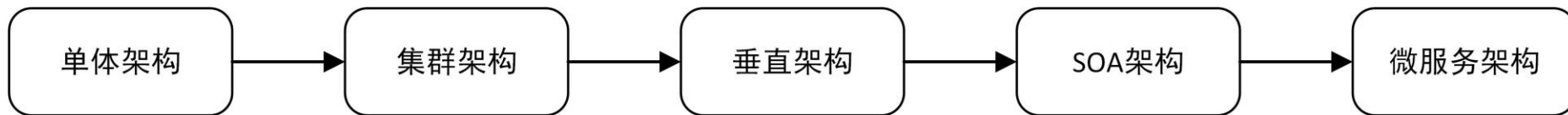
# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
- 微服务开发模式

# 软件服务架构的发展

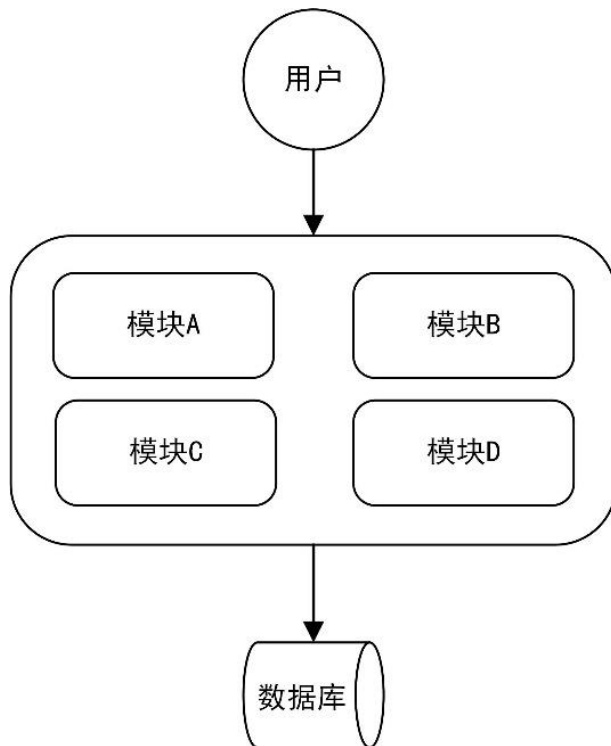
- 随着互联网和电子商务的跨越式发展，出现了一些大型网站和系统，日活用户量都在千万级别，软件系统架构逐步变得不堪重负。如何打造一个高可用性、高性能、易扩展、可伸缩且安全的网站，一直是业界关注的焦点。
- **软件的服务架构**也是随着业务的需求逐步发展的。如下图所示，大概可分为**5**个阶段：单体架构、集群架构、垂直架构、面向服务架构和微服务架构。



# 软件服务架构的发展

## ➤ 1) 单体结构

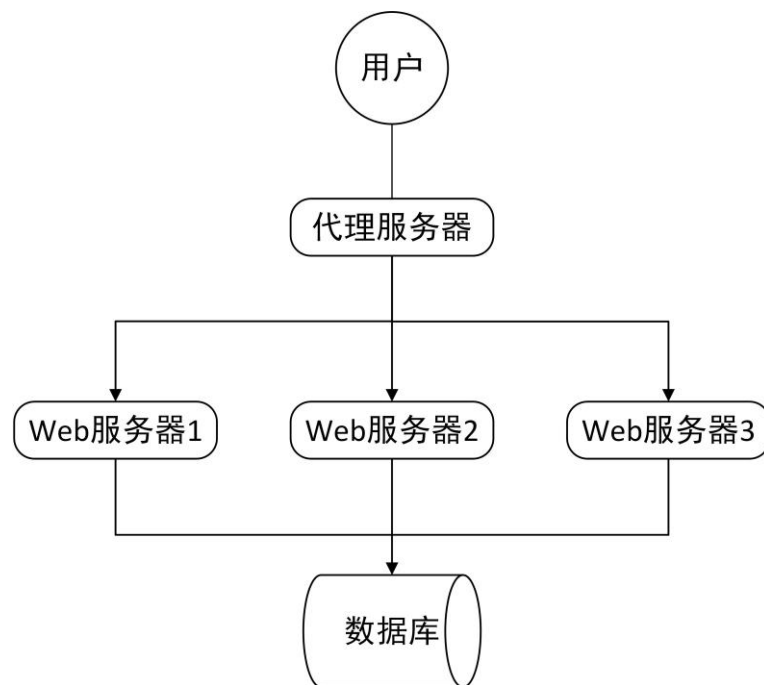
- 单体应用就是指将所有功能集中在一个项目上，不可分开部署的应用。
- 单体应用具有部署便捷、调试便捷和共享便捷等优点。
- 但缺点体现在复杂性高、稳定性差、可维护性差等方面。单体应用在一个项目中实现所有功能，这经常导致逻辑复杂、模块耦合、代码臃肿、修改难度大。



# 软件服务架构的发展

## ➤ 2) 集群结构

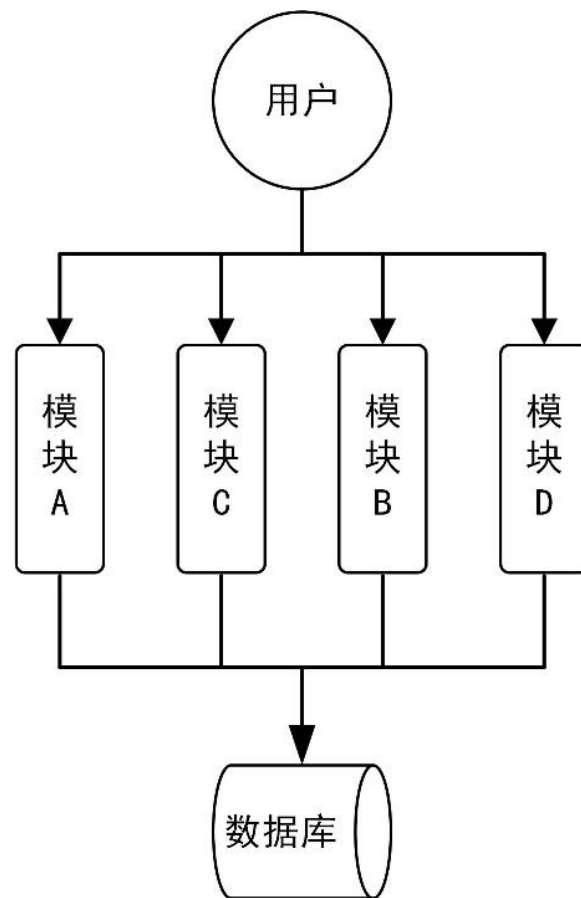
- 集群结构可看作单体结构的延伸。所有代码仍然在一个工程内，不带来额外的开发工作。但可部署多个服务器形成集群，进行横向扩展并引入反向代理做负载均衡。
- 同时，当流量增长到一定阶段，集群的瓶颈会变成后端对应的状态存储，比如数据库。



# 软件服务架构的发展

## ➤ 3) 垂直架构

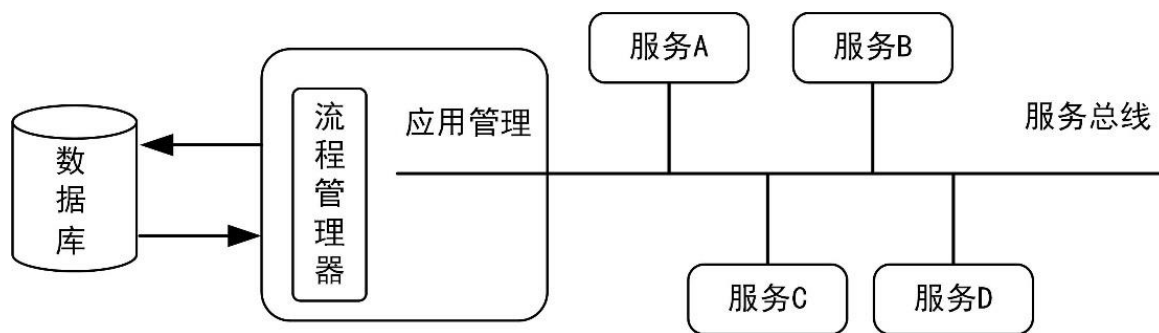
- 垂直架构是根据业务功能通过对单体架构进行垂直拆分得到的，实现了流量分担，解决了并发问题。相对于单体架构，垂直架构具有水平扩展性高、负载均衡、容错率高等优点。
- 缺点是复杂应用的开发维护成本变高，部署效率逐渐降低，公共功能重复开发，代码重复率高。



# 软件服务架构的发展

## ➤ 4) 面向服务架构

- 面向服务架构**SOA**是一个组件模型，它将应用程序的不同功能单元（称为服务）进行拆分，并通过这些服务之间定义良好的接口和协议联系起来。
- **SOA**架构能够提高开发效率，降低系统之间的耦合，具有良好的扩展性。但**SOA**架构里面比较依赖企业服务总线（**ESB**），所有的服务都集中在一个 **ESB** 上。

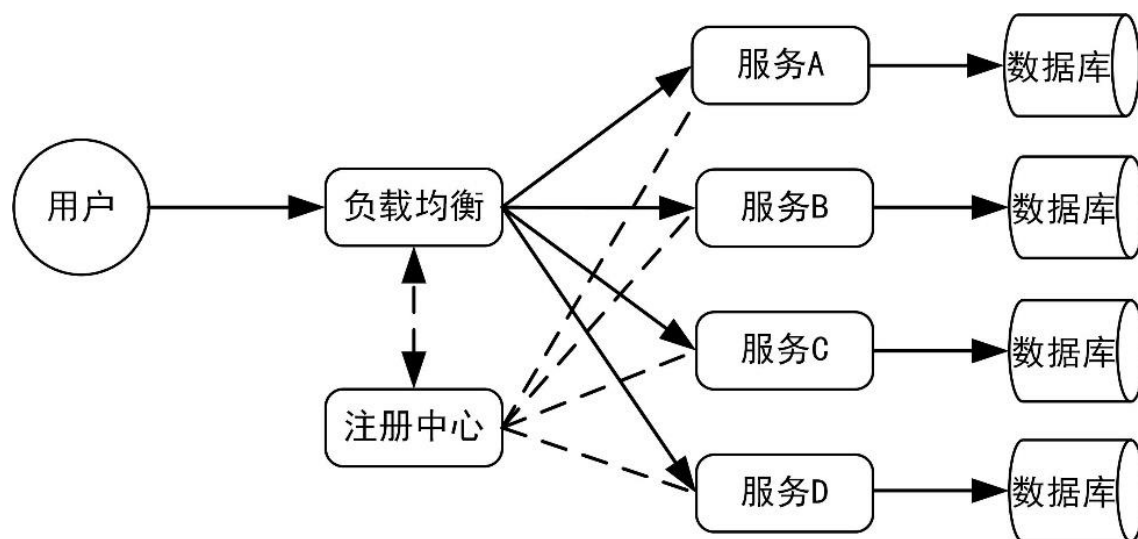




# 软件服务架构的发展

## ➤ 5) 微服务架构

- 微服务架构采用一组服务的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻量级交互机制来通信。



# 软件服务架构的发展

---

- 微服务是 **SOA** 的子集。微服务架构与**SOA**架构主要有以下3点区别：
  - ✓ **1) 架构划分不同**。**SOA**强调按水平架构划分。微服务强调按垂直架构划分，按业务能力划分。
  - ✓ **2) 技术平台选择不同**。**SOA**应用倾向于使用统一的技术平台。微服务可以针对不同业务特征选择不同技术平台，去中心统一化，发挥各种技术平台的特长。
  - ✓ **3) 系统间边界处理机制不同**。**SOA**架构强调的是异构系统之间的通信和解耦合。微服务架构强调的是系统按业务边界做细粒度的拆分和部署。

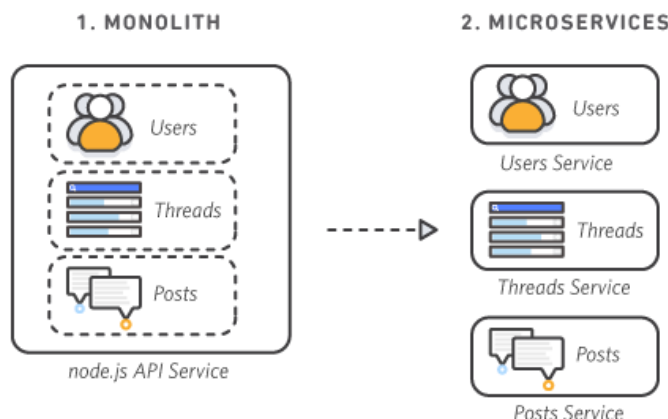
# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
- 微服务开发模式

# 微服务的概念

- **微服务**是面向服务体系结构（**SOA**）架构样式的一种变体。**SOA**架构将应用拆分成多个核心功能。而微服务架构更进一步，服务粒度更细，**每个功能都称为一项服务**。
- 微服务围绕着具体业务进行构建，并且能够被**独立地部署**到生产环境。各项服务出现故障时，不会相互影响。微服务中几乎不存在集中管理，服务之间采用**轻量级的通信机制**（如**HTTP、REST或Thrift API**）互相沟通。



# 微服务的概念

---

➤ 微服务具有以下特点:

- ✓ **1) 微服务之间是松耦合的。**微服务的功能可分为业务功能和技术功能，各服务之间耦合度较低。每个微服务都是单一职责的。
- ✓ **2) 各服务之间是独立部署的。**当改变一个特定的微服务时，只需要将该微服务的变更部署到生产环境中，而无需部署或触及系统的其他部分。
- ✓ **3) 每个微服务有独立的数据存储，易于维护。**采用独立的数据存储，能有效避免微服务间在数据库层面的耦合。可以根据微服务的业务和需求选择合适的数据库技术。

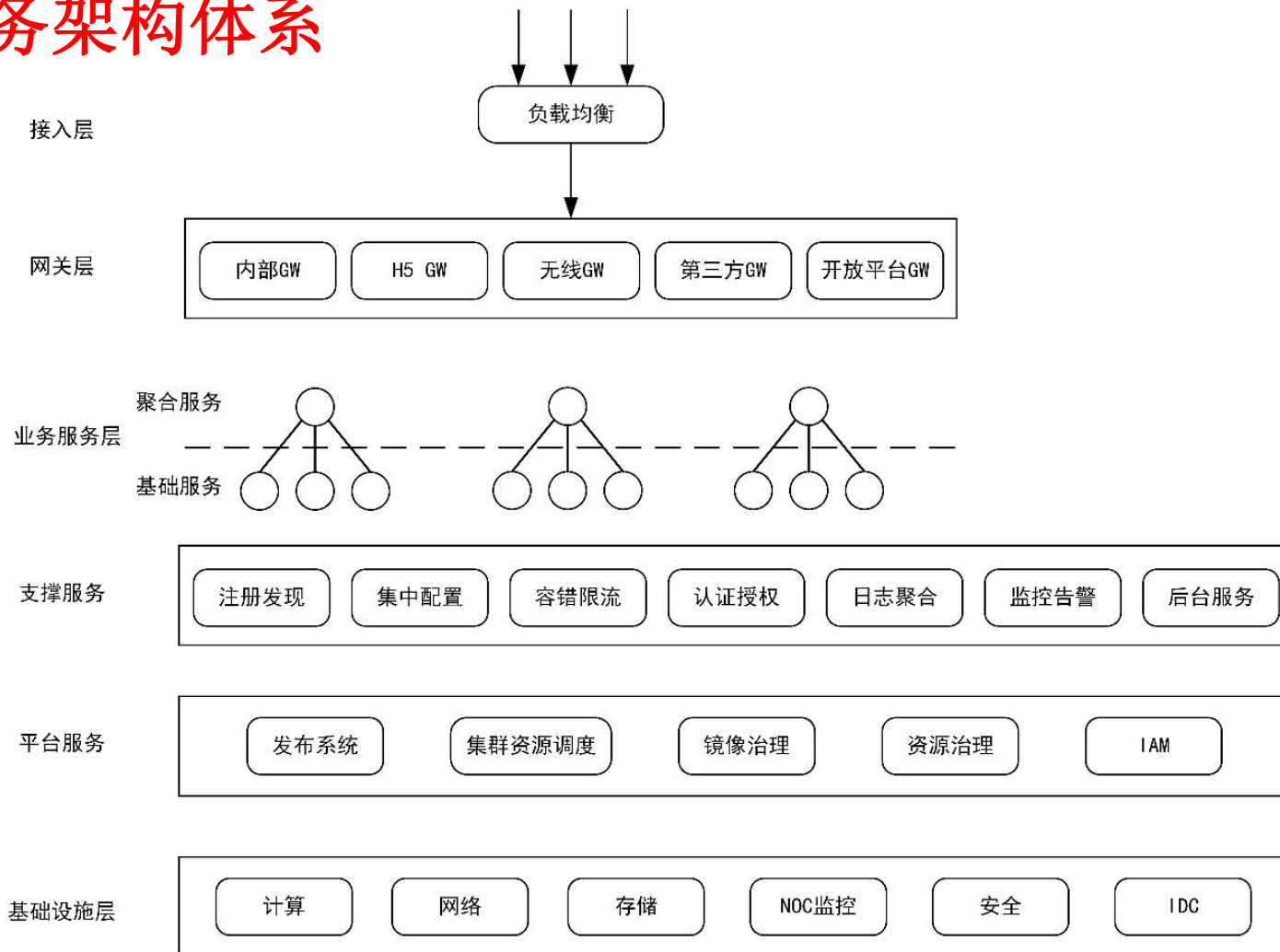
# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
- 微服务开发模式

# 微服务架构

## ➤ 微服务架构体系



# 微服务架构

层次	功能
接入层	通过负载均衡接入请求到内部平台
网关层	业务层接收外部流量的屏障，保障后台服务安全；识别请求的权限；对请求进行分类
业务服务层	微服务的核心层，包含了系统核心的业务逻辑。可划分为聚合服务层与基础服务层
支撑服务层	提供非业务功能，以支撑业务服务层和网关层软件的正常运行
平台服务层	站在系统平台的角度上，有处理系统发布、资源调度整合等功能
基础设施层	提供支撑系统需要的硬件资源，包括计算、网络、存储、监控、安全、互联网数据中心等



# 微服务架构

---

- 微服务架构的**设计理念**是各服务间保持隔离、自治、独立部署、异步通信，但各独立的服务可进行组合以满足业务的需求。下面介绍的是几种常用的组合方式。

## 微服务设计模式

聚合器设计模式

代理设计模式

链式设计模式

分支设计模式

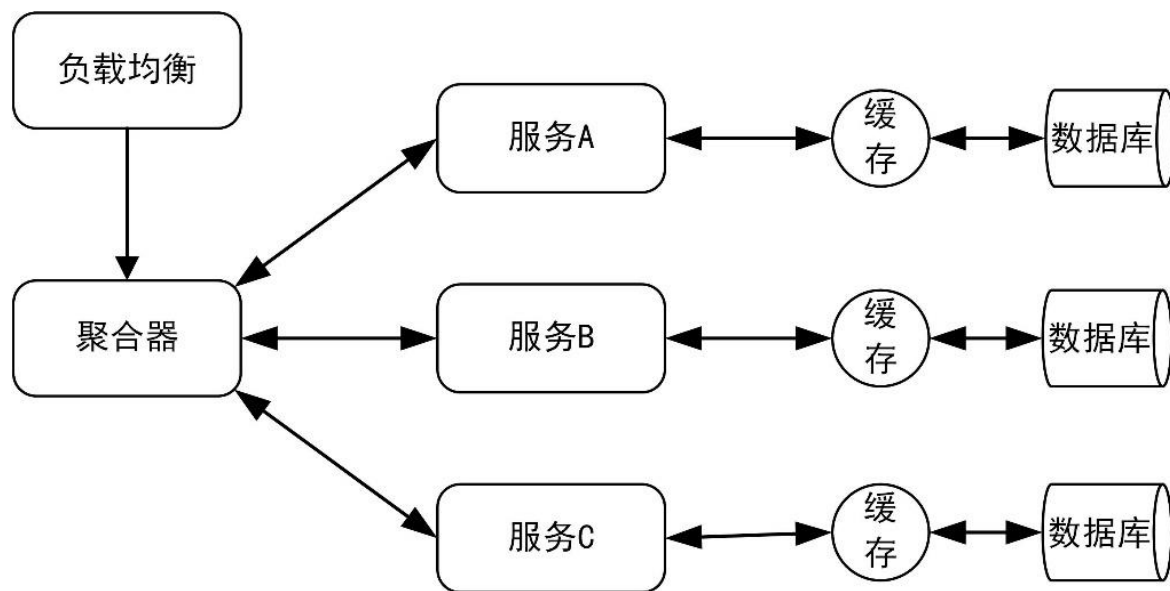
数据共享设计模式

异步消息设计模式

# 微服务架构

## ➤ 1) 聚合器设计模式

- 聚合器根据业务流程处理的需要，以一定的顺序调用依赖的多个微服务，对依赖的微服务返回的数据进行组合、加工和转换，最后以一定的形式返回给使用方。

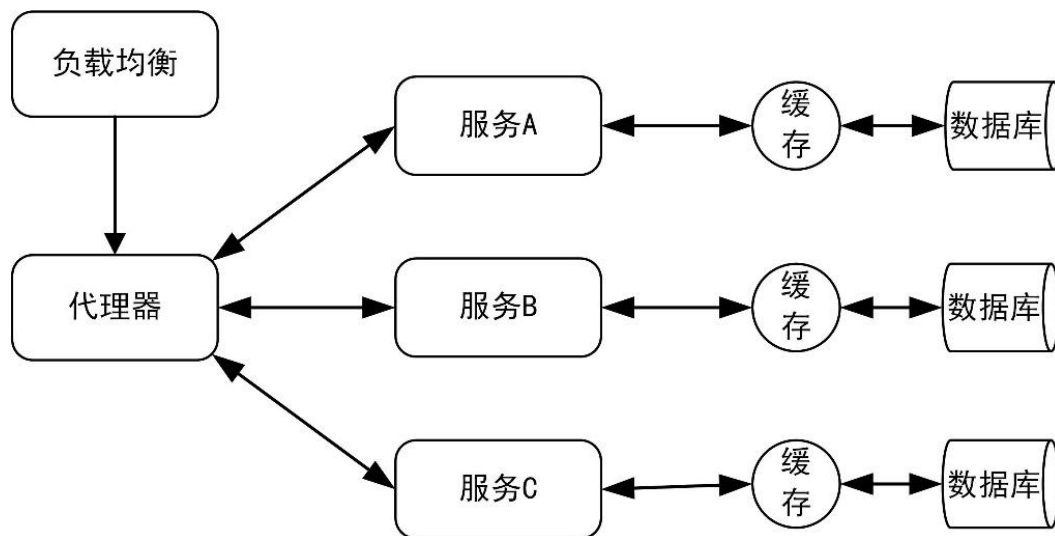


# 微服务架构

## ➤ 2) 代理设计模式

- 代理设计模式是聚合器模式的一个变体。客户端并不使用聚合器聚合数据，而是使用代理根据业务需求的差别调用不同的微服务。代理可以仅委派请求，也可以进行数据转换工作。

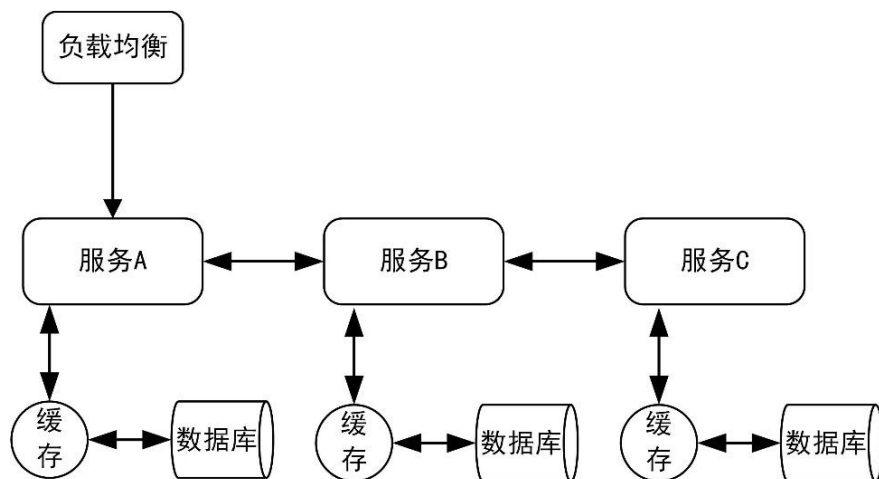
。



# 微服务架构

## ➤ 3) 链式设计模式

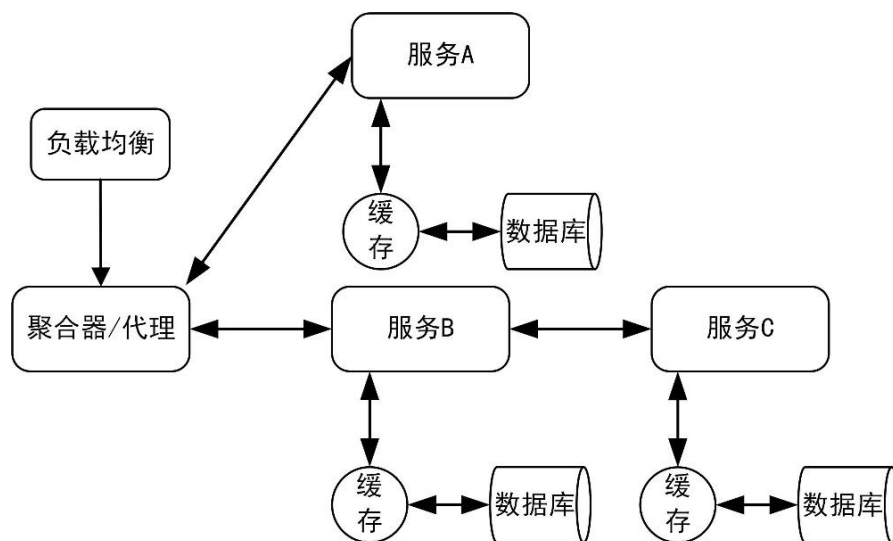
- 链式设计模式在接收到请求后会产生一个经过合并的响应。服务A接收到请求后会与服务B进行通信；类似地，服务B会同服务C进行通信。所有服务都使用同步消息传递。在整个链式调用完成之前，客户端会一直阻塞。因此，服务调用链不宜过长，以免客户端长时间等待。



# 微服务架构

## ➤ 4) 分支设计模式

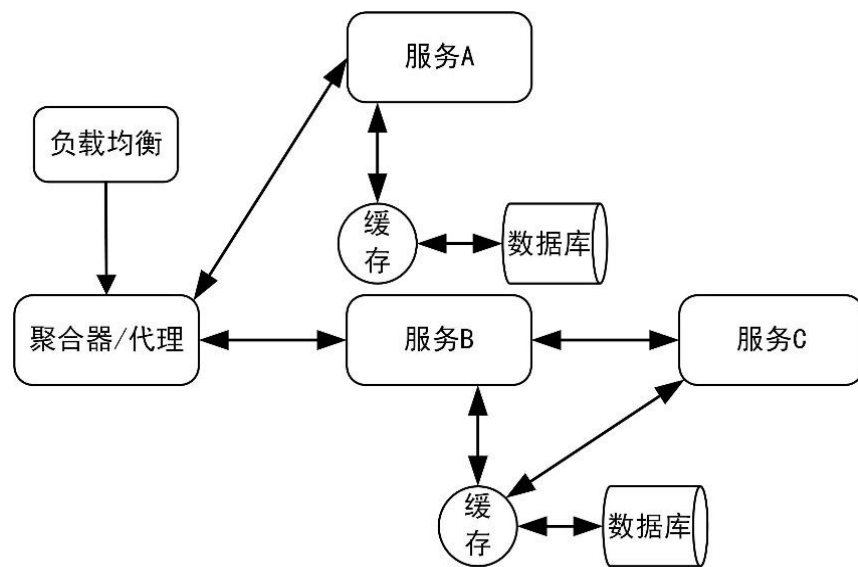
- 分支设计模式是聚合模式、代理模式和链式模式结合的产物。该模式下，分支服务可以拥有自己的数据库存储，调用多个后端服务或者服务串联链，然后将结果进行组合处理再返回给客户端。也可以使用代理模式。



# 微服务架构

## ➤ 5) 数据共享设计模式

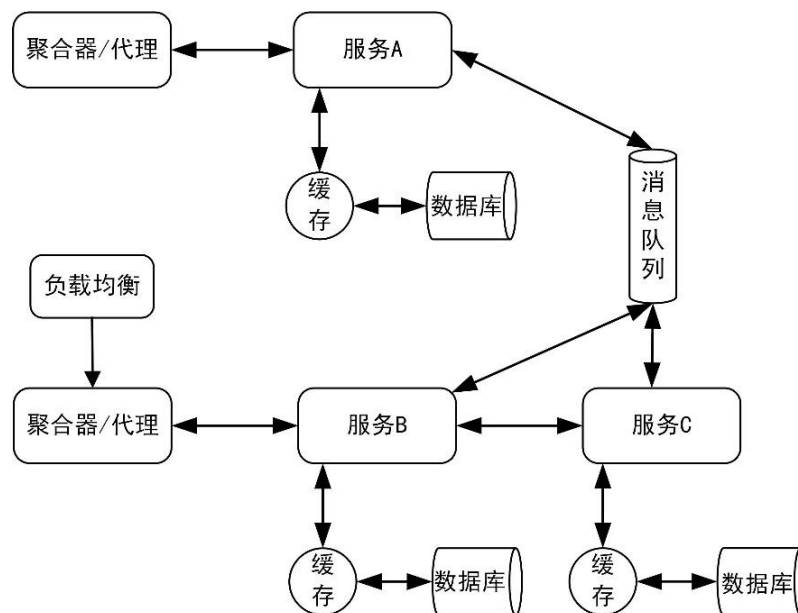
- 在单体应用到微服务架构的过渡阶段，可以使用数据共享模式。数据共享模式下部分微服务可以共享缓存和数据库存储。然而，该模式只适用于两个服务之间存在强耦合关系的情况。



# 微服务架构

## ➤ 6) 异步消息设计模式

➤ 在代理调用微服务过程中，同步模式在调用过程中会出现线程阻塞问题。可将微服务划分为多个集合，集合内的微服务使用同步调用模式，不同集合间的服务使用异步消息队列实现调用。服务B和服务C之间为同步调用模式，而服务A与服务B、服务C之间为异步调用模式。



# 微服务架构

---

- 常用的微服务架构方案
- 主流的微服务框架有Spring Cloud、ZeroC IceGrid、Dubbo、Service Fabric、Docker Swarm和基于消息队列的框架等。
- **Spring Cloud**是基于Spring Boot的一整套实现微服务的框架，简化了分布式系统基础设施的开发，用Spring Boot的开发风格做到一键启动和部署。
- 下面的部分将详细介绍Spring Cloud架构。



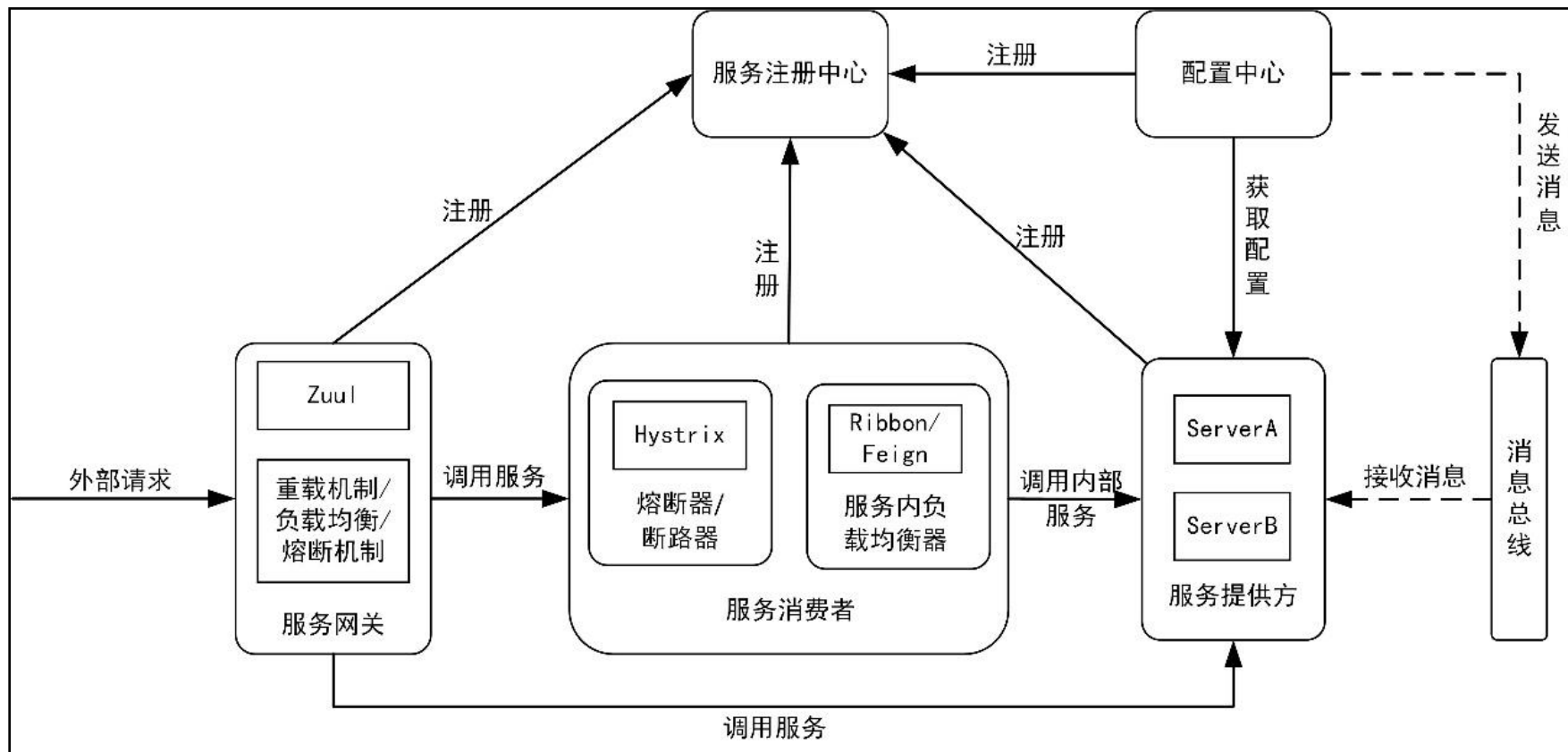
# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
- 微服务开发模式

# 基于Spring Cloud架构的开发

## ➤ Spring Cloud框架



Spring Cloud组件架构

# 基于Spring Cloud架构的开发

---

## ➤ Spring Cloud框架

- Spring Cloud包括了5大核心组件：服务发现组件Eureka、客户端负载均衡Ribbon、断路器Hystrix、服务网关Zuul和分布式配置Spring Cloud Config。

## ➤ 架构中各组件运行流程

- 1) 请求统一通过API网关（Zuul）来访问内部服务；
- 2) 网关接收到请求后，从注册中心（Eureka）获取可用服务
- 3) 由Ribbon进行均衡负载后，分发到后端具体实例；
- 4) 微服务之间通过消息总线进行通信处理业务；
- 5) Hystrix负责处理服务超时熔断。

# 基于Spring Cloud架构的开发

---

- **1) 服务发现框架Eureka**
- Eureka是基于REST的服务发现框架，主要用于定位服务，以实现负载均衡和中间层服务器的故障转移。
- Eureka由两个部分组成：**Eureka服务端**和**Eureka客户端**。服务端用作服务注册中心，支持集群部署；客户端是一个Java客户端，用来处理服务注册与发现，负责把服务的信息注册到服务端。客户端还具有一个内置的负载均衡器，可以执行基本的循环负载平衡。

# 基于Spring Cloud架构的开发

---

## ➤ 1) 服务发现框架Eureka

➤ Eureka的主要功能有：

- 1) 服务注册：Eureka客户端向Eureka Server注册时需提供自身的元数据，比如IP地址、端口、主页等；
- 2) 服务续约：客户端每隔30秒发送续约消息，若服务端在90秒没有收到客户端续约，将实例从其注册表中删除。
- 3) 获取注册列表信息：客户端从服务器获取注册表信息，并将其缓存在本地，用于查找其他服务，进行远程调用。
- 4) 服务下线：客户端在程序关闭时向Eureka服务器发送取消请求，从服务器的实例注册表中删除。

# 基于Spring Cloud架构的开发

---

## ➤ 2) 负载均衡Ribbon

- 负载均衡在系统架构中是一个非常重要的部分，它是提高系统的可用性、缓解网络压力和处理能力扩容的重要手段之一。
- Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，基于Netflix Ribbon实现。Spring Cloud对Ribbon进行了一些封装以更好的使用Spring Boot的自动化配置理念。

# 基于Spring Cloud架构的开发

---

## ➤ 2) 负载均衡Ribbon

➤ Ribbon的工作分为两步：

➤ 1) 选择Eureka Server，通常优先选择在同一个Zone且负载较少的Server；

➤ 2) 根据用户指定的策略，在从Server取到的服务注册列表选择一个地址。地址的挑选其实是在进行负载均衡。

➤ Ribbon中有多种[负载均衡算法](#)，包括轮询策略、随机策略、重试策略。默认使用轮询策略。

# 基于Spring Cloud架构的开发

---

## ➤ 3) 熔断器Hystrix

- 分布式系统环境下，一个业务调用通常依赖多个基础服务。当系统中部分服务故障时，调用该服务的线程受到阻塞，导致系统崩溃。
- Hystrix是Netflix开源的一个针对分布式系统的延迟和容错库。通过添加延迟容忍和容错逻辑，来控制这些分布式服务之间的交互。Hystrix能够对来自依赖的延迟和故障进行防护和控制，并阻止故障的连锁反应。
- 作用类似于电路中的保险丝，避免故障的“雪崩效应”。



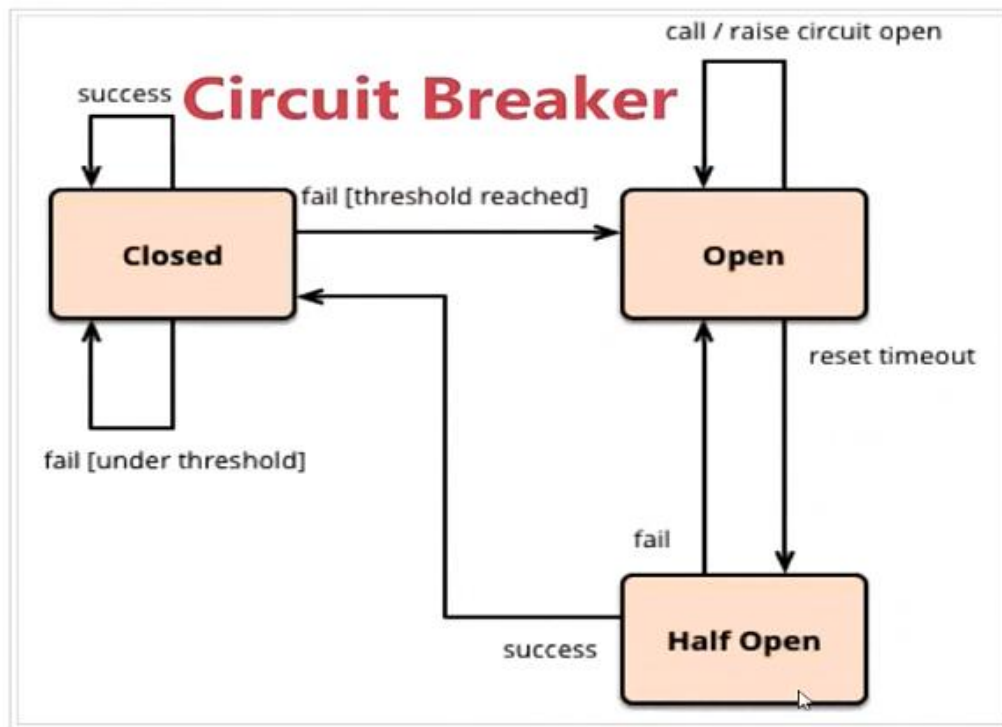
# 基于Spring Cloud架构的开发

---

- Hystrix工作流程如下：
- 1) 构造一个HystrixCommand或HystrixObservableCommand对象，用于封装请求，并在构造方法配置请求被执行需要的参数；
- 2) 执行命令，Hystrix提供了4种执行命令的方法；
- 3) 判断是否使用缓存响应请求，若启用了缓存，且缓存可用，直接使用缓存响应请求；
- 4) 判断熔断器是否打开，如果打开，跳到8；
- 5) 判断线程池/队列/信号量是否已满，已满则跳到8；
- 6) 执行HystrixObservableCommand.construct()或HystrixCommand.run()方法，如果执行失败或者超时，跳到8；否则，跳到9；
- 7) 统计熔断器监控指标；
- 8) 走Fallback备用逻辑；
- 9) 返回请求响应。

# 基于Spring Cloud架构的开发

- 具体地，当Hystrix Command请求后端服务失败数量超过一定比例（默认50%），熔断器会切换到**开路状态（Open）**。这时所有请求会直接失败而不会发送到后端服务。
- 熔断器保持在开路状态一段时间后（默认5秒），自动切换到**半开路状态（HALF-OPEN）**。这时会判断下一次请求的返回情况，如果请求成功，断路器切回**闭路状态（CLOSED）**，否则重新切换到开路状态（OPEN）。



# 基于Spring Cloud架构的开发

---

## ➤ 4) 微服务网关Zuul

- Zuul网关是系统唯一对外的入口，介于客户端与服务器端之间，用于对请求进行鉴权、限流、路由、监控等功能。
- Zuul的核心是一系列的filters，在zuul把请求路由到用户处理逻辑的过程中，这些filter对请求进行过滤处理。
- Zuul主要有以下几个功能：身份验证和安全性、洞察和监控、动态路由、压力测试、请求卸载、静态响应处理。

# 基于Spring Cloud架构的开发

## ➤ 4) 微服务网关Zuul

- Zuul需要通过向Eureka 进行注册获取系统中服务消费者和服务提供者的信息，从而实现请求的路由映射。下面是Zuul注册的配置文件：

```
server.port=8085↵  
spring.application.name=zuul-gateway↵  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

↵

注册后需要在 Zuul 启动类上标注@EnableZuulProxy: ↵

```
@SpringBootApplication↵  
@EnableZuulProxy↵  
@EnableEurekaClient↵  
public class ZuulApplication {↵  
    public static void main(String args) {↵  
        SpringApplication.run(ZuulApplication.class, args);↵  
    }↵  
}↵
```

↵

# 基于Spring Cloud架构的开发

---

## ➤ 4) 微服务网关Zuul

- 在Zuul中主要有三种过滤器，[前置过滤器](#)、[路由过滤器](#)和[后置过滤器](#)。
- 前置过滤器在请求之前进行过滤，路由过滤器根据路由策略进行过滤，而后置过滤器在响应之前进行过滤。

# 基于Spring Cloud架构的开发

---

## ➤ 5) 配置中心Config

- Spring Cloud Config是一个解决分布式系统的配置管理方案。包含了Client和Server两个部分。Server提供配置文件的存储、以接口的形式将配置文件的内容提供出去，Client通过接口获取数据、并依据此数据初始化自己的应用。
- 服务器存储后端的默认实现使用git，可轻松支持标签版本的配置环境并访问用于管理内容的各种工具。除了git外还可以用数据库、svn、本地文件等作为存储。

# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- **微服务案例**
- 微服务开发模式

# 微服务案例

---

- 本章节将快速构建一个可用的Spring Cloud微服务项目，项目构建使用的ide平台为JetBrain公司的idea，jdk版本设置为1.8。
- 首先需要建立一个Spring Cloud工程的父工程，随后的模块将在这个工程的基础上进行添加。
- 父工程下的pom文件需要添加响应的spring-boot的依赖，父模块主要规定spring boot的版本，设置spring boot启动器。
  -



# 微服务案例

- 父工程下的pom文件部分如下：

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

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Eureka服务中心

- 1) 新建eureka-server子模块
- 新建模块命名为eureka-server，并配置eureka-server模块下的pom文件内容，指明该模块为spring-cloud-starter-netflix-eureka-server模块。
- eureka-server模块下的pom文件内容部分如下：

```
<dependency>␣  
    <groupId>org.springframework.cloud</groupId>␣  
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>␣  
</dependency>␣  
  
<dependency>␣  
    <groupId>org.springframework.boot</groupId>␣  
    <artifactId>spring-boot-starter-test</artifactId>␣  
    <scope>test</scope>␣  
</dependency>␣
```

# 微服务案例——搭建Eureka服务中心

- 2) 配置eureka-server子模块
- 在eureka-server模块下的src.main.resources子文件夹下新建Resources Bundle类型，命名为application.properties，设置eureka-server中心的访问端口等信息，application.properties文件中的内容如下：

```
spring.application.name=eureka-server↵
server.port=8080# 服务端口号↵
eureka.instance.hostname=localhost↵
eureka.client.register-with-eureka=false#是否将 eureka 服务本身登记到注册中心↵
eureka.client.fetch-registry=false#是否从 eureka 注册中心获取注册中心↵
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/
#配置暴露给 eureka client 的请求地址↵
```

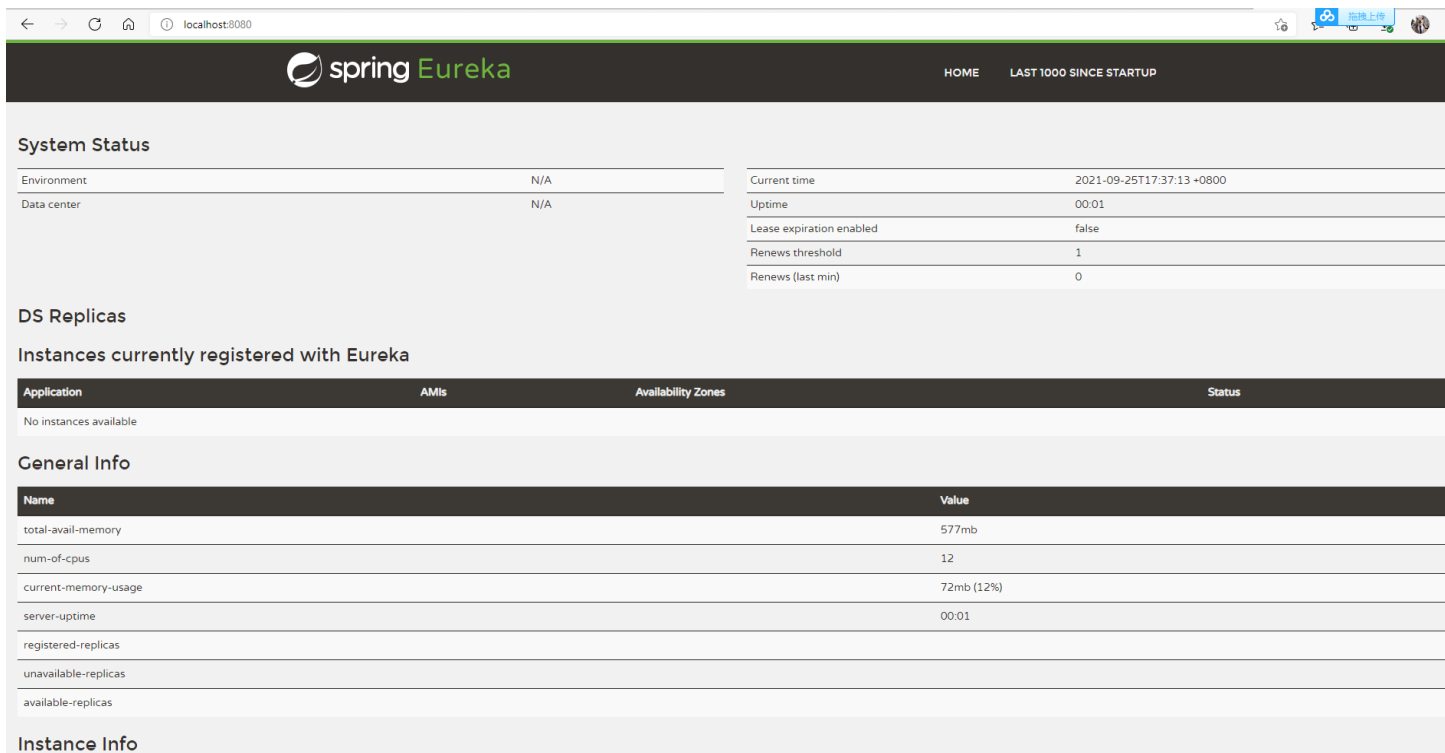
# 微服务案例——搭建Eureka服务中心

- 3) 编写启动类
- 在eureka-server模块下的src.main.java文件夹下新建包，命名为com.eureka.server，在该包下新建类ServerApplication.java作为Spring Boot启动类。

```
@EnableEurekaServer//指定为 eureka server，激活 server 服务↵
@SpringBootApplication//指定为 spring-boot 应用↵
public class ServerApplication {↵
    ↵
    public static void main(String[] args) {↵
        SpringApplication.run(ServerApplication.class, args);↵
    }↵
    ↵
}
```

# 微服务案例——搭建Eureka服务中心

- 4) 浏览器查看
- 打开浏览器，键入properties文件中设置的服务地址，localhost:8080（或者127.0.0.1:8080），可以看到eureka的配置中心已经配置完成了。



The screenshot displays the Spring Eureka web interface in a browser window. The address bar shows 'localhost:8080'. The interface has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'.

**System Status**

Environment	N/A	Current time	2021-09-25T17:37:13 +0800
Data center	N/A	Uptime	00:01
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

**DS Replicas**

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

**General Info**

Name	Value
total-avail-memory	577mb
num-of-cpus	12
current-memory-usage	72mb (12%)
server-uptime	00:01
registered-replicas	
unavailable-replicas	
available-replicas	

**Instance Info**

# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Eureka-Client客户端

- 1) 新建eureka-client子模块
- 在父工程下新建模块命名为eureka-client，配置pom.xml文件设置坐标指明本模块是Spring-Cloud下的eureka-client服务。pom文件内容部分如下：

```
<dependency>␣  
    <groupId>org.springframework.cloud</groupId>␣  
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>␣  
</dependency>␣  
  
<dependency>␣  
    <groupId>org.springframework.boot</groupId>␣  
    <artifactId>spring-boot-starter-test</artifactId>␣  
    <scope>test</scope>␣  
</dependency>␣  
<dependency>␣  
    <groupId>org.springframework.boot</groupId>␣  
    <artifactId>spring-boot-starter-web</artifactId>␣  
</dependency>␣  
</dependencies>
```



# 微服务案例——搭建Eureka-Client客户端

- 2) 配置eureka-server子模块
- 在src.main.resources下同样新建application.properties文件，添加服务的相关信息，告诉client，eureka-server的注册中心的地址，application.properties中内容如下：

```
spring.application.name=eureka-provider↵  
server.port=8081↵  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

# 微服务案例——搭建Eureka-Client客户端

---

- **3) 编写启动类和控制类**
- 在src.java文件下新建包com.eureka.client，并在该包下新建两个java类，第一个是ClientApplication服务提供者的启动类，第二个是ClientController作为控制类。
- 在启动类中通过SpringBootApplication注解来指明该模块可以作为服务单独启用，EnabelEurekaClient注解指明该服务是Eureka注册中心的服务提供者。
- 控制类中配置访问路径和提供的服务内容

# 微服务案例——搭建Eureka-Client客户端

```
@SpringBootApplication↵
@EnableEurekaClient//以 Client 的形式进行激活，这一条在新版本的 SpringCloud 中可以不
写↵
public class ClientApplication {↵
    ↵
    public static void main(String[] args) {↵
        SpringApplication.run(ClientApplication.class, args);↵
    }↵
```

```
@RestController↵
@RequestMapping(value = "/hello")↵
public class ClientController {↵
    ↵
    public String hello() {↵
        return "hello world";//服务内容仅让浏览器页面打印 hello world↵
    }↵
}↵
```

# 微服务案例——搭建Eureka-Client客户端

- 4) 浏览器查看
- 启动Client模块，将服务注册到Server端，在浏览器中键入Eureka-Server路径（127.0.0.1:8080）查看，能看到新注册进来的服务。

## DS Replicas

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-PROVIDER	n/a (1)	(1)	UP (1) - 192.168.1.2:eureka-provider:8081

# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Ribbon负载均衡器

---

- 具有同一功能或者类似功能的微服务提供者将服务共同注册到服务中心时，服务消费者通过[负载均衡器](#)的配置能够有秩序地借助负载均衡策略访问微服务的提供者，防止单点访问压力过大。
- 1) 新建eureka-client2服务提供者
- 参照上一部分中的Eureka-Client配置，构建一个新的服务提供者Eureka-Client2，功能保持与之前的Client基本一致。
- application.properties中需要另外指定[端口](#)，区别于之前的client，[服务名称spring.application.name](#)需要与之前的client保持一致，表示两个模块提供的是同一种微服务

# 微服务案例——搭建Ribbon负载均衡器

- 2) 搭建服务使用者模块
- 以maven的方式新建eureka-consumer模块。编辑pom.xml文件配置好所需要的spring包，在src.main.java文件加下新建包com.eureka.consumer，在该包下新建控制类和启动类，同时在src.main.resources文件夹下新建文件application.properties文件，配置模块的端口：

```
spring.application.name=eureka-consumer↵  
server.port=8083↵  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

# 微服务案例——搭建Ribbon负载均衡器

- 2) 搭建服务使用者模块
- 启动类ConsumerApplication:

```
@EnableEurekaClient
@EnableDiscoveryClient //允许消费者发现并调用 Eureka 中心已经注册的服务内容
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    @LoadBalanced //使用内置的 Ribbon 模块进行负载均衡
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

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



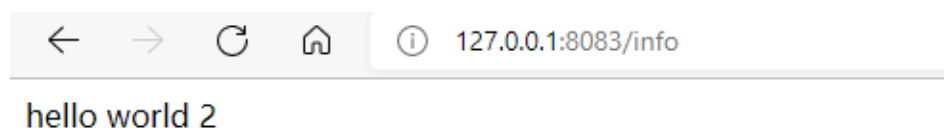
# 微服务案例——搭建Ribbon负载均衡器

- 2) 搭建服务使用者模块
- 控制类ConsumerController:

```
@RestController<
public class ConsumerController {<
<
    @Autowired<
    private RestTemplate restTemplate; //使用 RestTemplate 模板并对服务进行地址映射<
<
    @GetMapping("/info")<
    public String getInfo() {<
        return this.restTemplate.getForEntity("http://eureka-provider/hello", String.class).getBody();<
    }<
}<
```

# 微服务案例——搭建Ribbon负载均衡器

- 3) 测试
- 打开浏览器访问eureka-consumer的页面（<http://localhost:8083/info>）内容显示hello world, 此时刷新页面，由于负载分担机制服务消费者会调用另外一个服务提供者的服务内容，页面显示hello world 2。



# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Hystrix熔断器

---

- 下面的例子中，将通过关闭其中的一项服务来模拟服务出现故障的情况，并通过配置和代码编写设置出现服务调用失败时上游服务提供的服务内容。
- 1) 新建hystrix子模块
- 首先新建hystrix模块，并配置pom.xml，指明该模块为hystrix模块。

# 微服务案例——搭建Hystrix熔断器

- 2) 配置Hystrix子模块
- 在application.properties配置文件中指明服务的名称端口号，以及Eureka注册中心的地址，application.properties文件内容如下：

```
spring.application.name=eureka-consumer-hystrix↵  
server.port=8084↵  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

# 微服务案例——搭建Hystrix熔断器

- 3) 编写启动类和控制类
- 在HystrixApplication类中使用@EnableHystrix注解激活Hystrix模块，HystrixApplication类文件内容如下：

```
@EnableEurekaClient<
@EnableDiscoveryClient<
@SpringBootApplication<
public class ConsumerApplication {<
<
    @Bean<
    @LoadBalanced<
    RestTemplate restTemplate() {<
        return new RestTemplate();<
    }<
<
    public static void main(String[] args) {<
        SpringApplication.run(ConsumerApplication.class, args);<
    }<
```

# 微服务案例——搭建Hystrix熔断器

- 3) 编写启动类和控制类
- 在HystrixController类文件中通过@GetMapping注解指明访问域名，通过@HystrixCommand注解指明发生服务熔断时调用的方法，HystrixController内容如下：

```
@RestController
public class HystrixController {
    private Logger log = LoggerFactory.getLogger(this.getClass());

    @Autowired
    private RestTemplate restTemplate;

    @GetMapping("/info")
    @HystrixCommand(fallbackMethod = "getDefault")//getDefault 为熔断时调用的方法名，注意正常服务调用的返回值要和发生 fallback 情况下调用方法的返回值相同
    public String getInfo() {
        return this.restTemplate.getForEntity("http://eureka-provider/hello", String.class).getBody();
    }//正常调用时调用之前的 eureka-consumer 服务内容

    public String getDefault() {
        return "the default";
    }//发生熔断时页面返回 the default 字符串
```

# 微服务案例——搭建Hystrix熔断器

- 4) 测试
- 启动Hystrix模块并访问127.0.0.1: 8080查看eureka-server页面可以看到之前配置的所有模块的内容:

## DS Replicas

### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-CONSUMER	n/a (1)	(1)	UP (1) - 192.168.1.2:eureka-consumer:8083
EUREKA-CONSUMER-HYSTRIX	n/a (1)	(1)	UP (1) - 192.168.1.2:eureka-consumer-hystrix:8084
EUREKA-PROVIDER	n/a (2)	(2)	UP (2) - 192.168.1.2:eureka-provider:8081 , 192.168.1.2:eureka-provider:8082



# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Zuul网关

---

- Zuul网关能为微服务云平台提供动态路由、压力测试、负载分配、身份认证等等功能，其本质是一系列的过滤器，本小节将构建一个简单的zuul网关模块实现简单的动态路由功能。
- 1) 新建zuul子模块
- 构建zuul模块，在项目中新建模块命名为zuul，同样先配置好模块的pom文件，添加依赖。

# 微服务案例——搭建Zuul网关

- 2) 配置zuul子模块
- 在resources下新建配置文件application.properties，为zuul设置路由规则，即将其捕获到的URL映射到真正想要访问的微服务当中，下面的配置文件中表示zuul模块的访问端口为8085，将所有/api-a/\*\*的路径映射到前面eureka-server中已经注册的eureka-provider服务当中

```
server.port=8085↵
spring.application.name=zuul-gateway↵
zuul.routes.api-a.path=/api-a/**↵
zuul.routes.api-a.service-id=eureka-provider↵
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

# 微服务案例——搭建Zuul网关

- 3) 编写启动类和控制类
- 新建包`com.zuul.gate`，并在该包下新建启动类和控制类。  
启动类中通过注解`@EnableZuulProxy`标明这是一个zuul应用服务，在控制类中编写页面显示的内容，启动类`GateApplication`的内容如下：

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

# 微服务案例——搭建Zuul网关

---

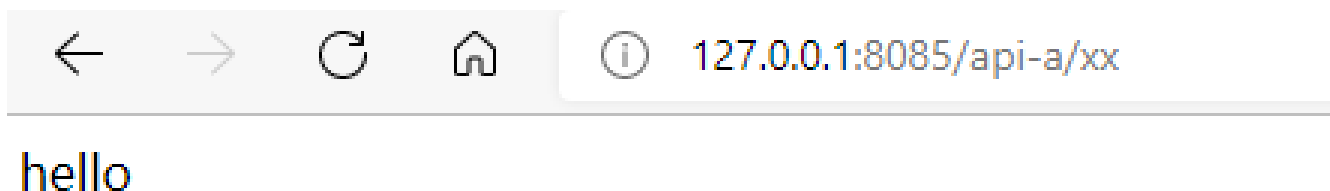
- 3) 编写启动类和控制类
- 控制类TestController内容如下：

```
@RestController
public class TestController {

    @GetMapping("/api-a/**") //URL 地址与 application.properties 配置的路径对应,**表示任意字符
    public String hello() {
        return "hello";
    }
}
```

# 微服务案例——搭建Zuul网关

- 4) 测试
- 浏览器地址中键入配置好的URL(127.0.0.1:8085/api-a/xx)其中xx可以使用任意字符替换，我们可以看到页面正常显示：



# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
  - ✓ 搭建**Eureka**服务中心
  - ✓ 搭建**Eureka-Client**客户端
  - ✓ 搭建**Ribbon**负载均衡器
  - ✓ 搭建**Hystrix**熔断器
  - ✓ 搭建**Zuul**网关
  - ✓ 搭建**Config**配置中心
- 微服务开发模式

# 微服务案例——搭建Config配置中心

---

- Spring Cloud Config是一个解决分布式系统的配置管理方案，将项目中所使用到的所有配置文件进行统一管理，它分为Client端和Server端两个部分，Server端提供配置文件的存储和内容发布，而Client端则通过Server服务的接口获取配置文件数据用以初始化自身应用。



# 微服务案例——搭建Config配置中心

- 1) 构建配置中心服务器config-server
- 同之前小节中的内容一样分为三步：pom文件编写、主程序启动类编写、服务配置文件编写。
- 在pom文件中添加依赖标明模块功能为spring-cloud-config-server。
- 启动类ServerApplication如下：

```
@EnableConfigServer //指定模块启用 config 配置中心功能↵
@EnableDiscoveryClient↵
@SpringBootApplication↵
public class ServerApplication {↵
    ↵
    public static void main(String[] args) {↵
        SpringApplication.run(ServerApplication.class, args);↵
    }↵
    ↵
}
```

# 微服务案例——搭建Config配置中心

- 1) 构建配置中心服务器config-server
- 配置文件的编写中有两部分，首先需要在resources下新建application.properties配置文件配置config-server服务本身的相关信息：

```
spring.application.name=config-server↵  
server.port=8888↵  
spring.cloud.config.discovery.enabled=true↵  
spring.profiles.active=native#表示配置文件存储在本地↵  
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

- 新建configtest.properties配置文件充当client端所请求的配置文件，仅仅做一个字符串的映射：

```
word=hello world↵
```

# 微服务案例——搭建Config配置中心

- 2) 构建配置访问客户端config-client
- 新建模块config-client用作读取配置中心提供的配置文件的客户端，修改pom.xml文件内容进行依赖添加。
- 为了读取配置中心的配置文件，client需要指定自己访问的配置中心地址和所需要获取的配置文件名称，在resources下新建文件bootstrap.properties，并添加如下信息：

```
spring.cloud.config.name=configtest# 指定配置文件的名称↵
spring.cloud.config.profile=native# 指定获取配置文件的方法是在 server 端本地获取↵
spring.cloud.config.label=master# 获取配置文件的分支，默认是 master。如果是本地获取的话，则无用↵
spring.cloud.config.discovery.enabled=true # 开启配置信息可被发现↵
spring.cloud.config.discovery.service-id=config-server# 绑定 config-server 服务名称,这个名称需要与 config-server 端的配置文件中一致↵
eureka.client.serviceUrl.defaultZone=http://localhost:8080/eureka/↵
```

# 微服务案例——搭建Config配置中心

---

- 2) 构建配置访问客户端`config-client`
- 同时`client`服务提供页面访问的功能，所以还需要指定自己的访问端口和服务名称，在`resources`文件夹下新建文件`application.properties`指明`client`端服务的名称和端口号：

```
spring.application.name=config-client↵  
server.port=8087↵
```

# 微服务案例——搭建Config配置中心

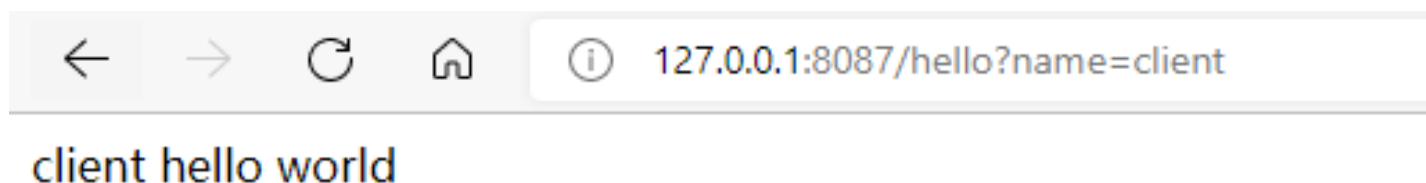
- 2) 构建配置访问客户端config-client
- 编写Client的启动类和控制类，在config-client模块的src.main.java下新建包com.config.client，在该包下新建两个类：ClientApplicationl.java与ClientController.java

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

```
@RestController<
public class ClientController {<
    <
    @Value("${word}")<
    private String word;<
    <
    @RequestMapping("/hello")<
    public String index(@RequestParam String name) {<
        return name+", "+this.word;<
    }<
    //@RequestParam String name 注解获取 URL 后的访问参数<
}<
```

# 微服务案例——搭建Config配置中心

- 3) 测试
- 启动config-server, config-client启动类浏览器访问127.0.0.1:8087/hello?name=client可以看到页面中显示client hello world, 表示client端服务可以正常的拿到server端上的配置文件的内容(word = hello world)。



# 大纲

---

- 软件服务架构的发展
- 微服务的概念
- 微服务架构
- 基于**Spring Cloud**架构的开发
- 微服务案例
- 微服务开发模式

# 微服务开发模式

---

- ▶ 微服务架构的开发模式不同于传统方式，它倡导围绕应用程序为核心，按业务能力来划分为不同的团队。每个团队都要求能够对每个服务，将其对应的业务领域的全部功能实现。比如，团队需负责某业务需求的更改，从用户体验界面到业务逻辑实现，再到数据的存储和迁移等。
- ▶ 微服务架构这种开发模式，得益于微服务响应速度快、易于集成、开发效率高等特点。随着业务复杂性的增加和团队规模的扩大，微服务的松散耦合自治特性能有效提升系统的开发效率。



# 微服务开发模式

---

- ▶ 在构建微服务应用时需要将应用拆分为多个粒度适中的服务。在微服务拆分过程中需遵循以下原则：

## 服务的拆分

高内聚低耦合，服务粒度适中  
围绕业务概念建模  
演进式拆分

# 微服务开发模式

---

- **1) 高内聚低耦合，服务粒度适中**
- 在微服务架构中，每个微服务都应围绕具体的业务进行构建，服务**大小应该适中**。在拆分微服务过程中，应尽量**降低服务间的耦合**。服务拆分是为了横向扩展，因而应该**横向拆分**。
- 微服务的纵向拆分最多三层。分别是：**1) 基础服务层、2) 组合服务层、3) 控制层**。

# 微服务开发模式

---

- 2) 围绕业务概念建模
- 围绕业务建模是指以软件模型方式描述系统业务所涉及的对象和要素，以及对象的属性、行为和对象间的关系。围绕业务建模强调以体系的方式来理解、设计和构建应用系统。
- 在构建微服务系统过程中，使用业务模型可以确定微服务的功能边界。根据业务模型设计的微服务系统能有效提升微服务的凝聚力和自主性。

# 微服务开发模式

---

## ➤ 3) 演进式拆分

- 在拆分微服务应用时，难以一次给出合适的拆分粒度，可以使用“先粗后细”的方式拆分。在刚开始拆分时，可以将应用划分为几个粗粒度的子服务。当对服务有了更多认识后，会不断调整粒度，进行服务的进一步拆分、合并。

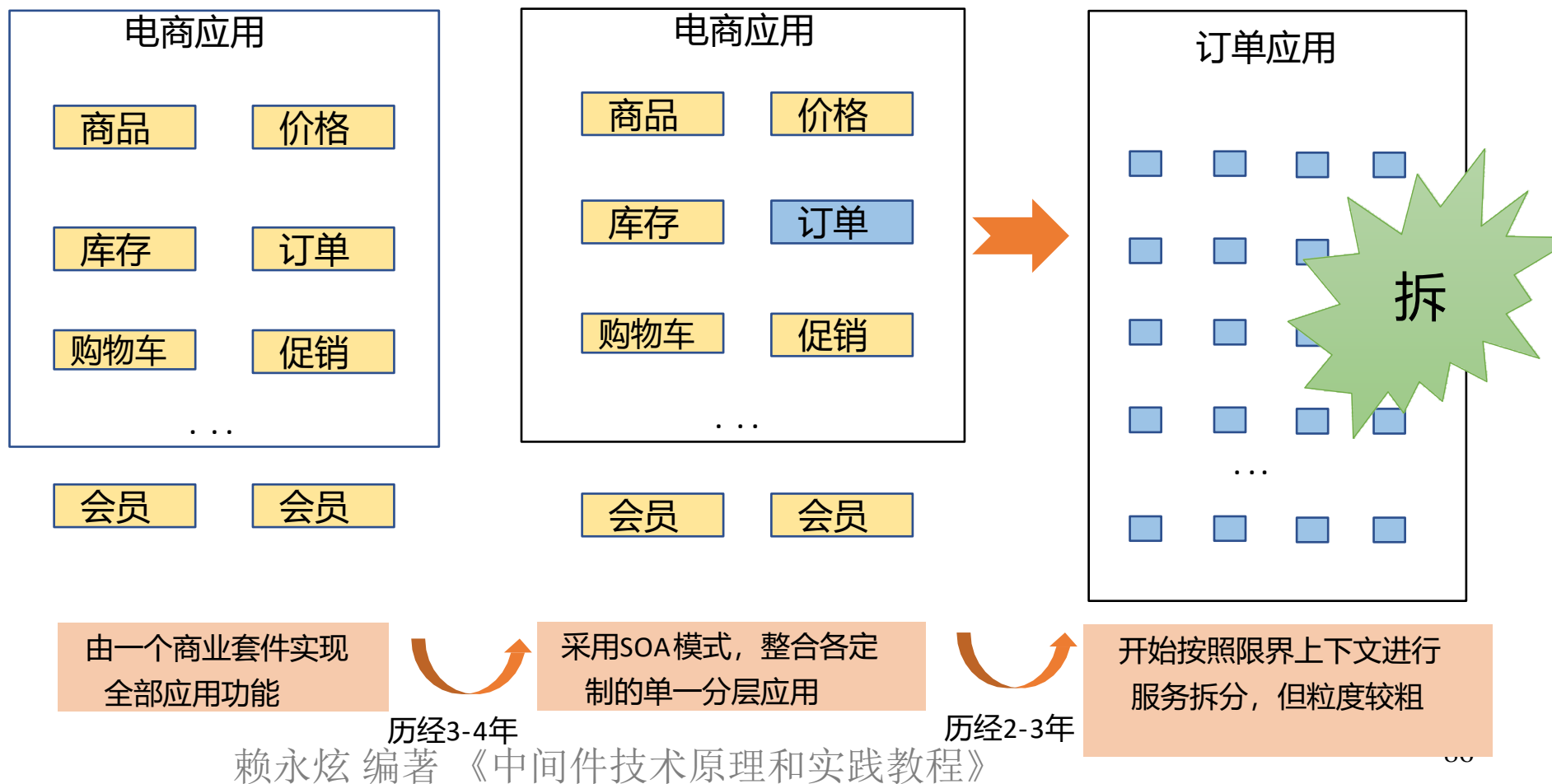
# 微服务开发模式

---

- 在刚开始构建系统时，系统业务较为简单，随着系统业务复杂度提升，需要将单体应用重构为微服务架构以提升系统服务效率。**重构单体应用的基本步骤**如下：
  - ✓ **1) 将新加特性构建为微服务。**用微服务构建新增功能，并隔离旧应用以提升系统的扩展性。
  - ✓ **2) 对旧应用进行分解。**首先需要围绕业务对系统进行建模，然后不断地提取微服务，直到应用中全部的“限界上下文”都提取为微服务，或其中所剩内容已无必要再提取。
  - ✓ **3) 提取组件为服务标准。**在提取组件过程中，需要识别整体架构内的“限界上下文”，把不一致概念分开。

# 微服务开发模式

因应业务发展而不断演变!



# 微服务开发模式

## ➤ 服务拆分举例

### 交易服务

- 下单
- 拆单
- 校验
- 支付
- ...

### 履约服务

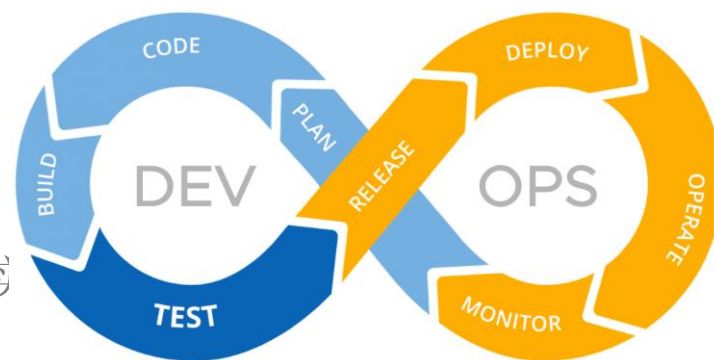
- 库存调度
- 物流调度
- 售后服务调度
- ...

### 查询服务

- 按用户查询
- 按营业员查询
- 按手机号查询
- ...

# 微服务开发模式

- **DevOps** (Development, Operations) 是一组过程、方法与系统的统称，用于促进开发（应用程序/软件工程）、技术运营和质量保障（**QA**）部门之间的沟通、协作与整合。它是一种重视**软件开发人员（Dev）**和**IT运维技术人员（Ops）**之间**沟通合作**的文化、运动或惯例。
- 构建基于微服务的应用时，如果不采用**DevOps**等自动化工具，工作量之大，复杂度之高将难以估量。因此，微服务的实施需要开发部门与运维部门的协作，**DevOps是微服务实施的充分必要条件**。





# 微服务开发模式

---

- 微服务架构具有松耦合、部署独立、数据存储独立等特点，但同时具有以下缺点：
  - ✓ 运维成本过高
  - ✓ DevOps是必须的
  - ✓ 接口不匹配
  - ✓ 代码重复
  - ✓ 分布式系统的复杂性
- 微服务架构有很多吸引人的地方，不过在拥抱微服务之前，需要认清它所带来的挑战。

# 本章小结

---

- 微服务架构能够将复杂臃肿的单体应用进行细粒度的服务化拆分，每个拆分出来的服务各自独立打包部署，并交由小团队进行开发和运维，从而极大地提高应用交付的效率。
- 本章首先介绍了软件服务架构的发展，介绍了微服务的概念、架构体系、设计模式和常用的微服务架构方案，然后介绍了经典的微服务开发框架Spring Cloud，包括Eureka、Ribbon、Hystrix、Zuul、Config等5大核心组件，并给出了实际编程案例；最后介绍了微服务的开发模式，包括服务拆分的准则等。