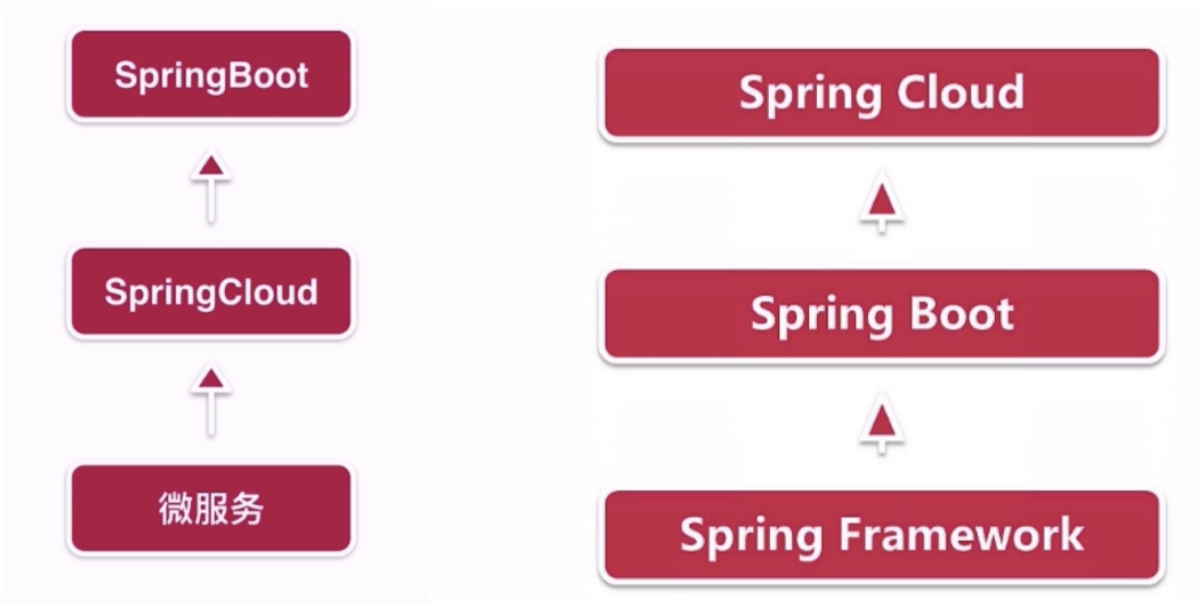
第一章 课程介绍

导学

Spring, SpringBoot, SpringCloud, 微服务 的关系



技术储备

- SpringBoot 的基础知识数量账务
- 对 Linux 和 Docker 的基本用法数量掌握

课程重点

SpringCloud 构建

微服务改造探讨

主要内容

- Eureka(服务注册) 服务注册和发现, 高可用服务注册中心
- Config(配置中心) 常规的 Server 和 Client 使用 Bus 结合RabbitMQ 实现自动刷新
- Ribbon(服务通讯) RestTemplate Feign Ribbon通讯方式
- Zuul(动态路由)
- Hystrix 熔断机制
- 容器编排: Docker + Rancher

第二章 微服务介绍

微服务和其他常见架构

微服务的提出

James Lewis & Martin Fowler 2014 年提出

微服务介绍

微服务是一种架构风格

微服务的特点

- 一系列微小的服务共同组成
- 跑在自己的进程里
- 每个服务为独立的业务开发
- 独立部署
- 分布式的管理

服务技术的演进

单一应用架构(ORM) -> 垂直应用架构(MVC) -> 分布式服务架构(RPC) -> 流动计算架构(SOA)

单体架构

优点

1. 容易测试
2. 容易部署

缺点

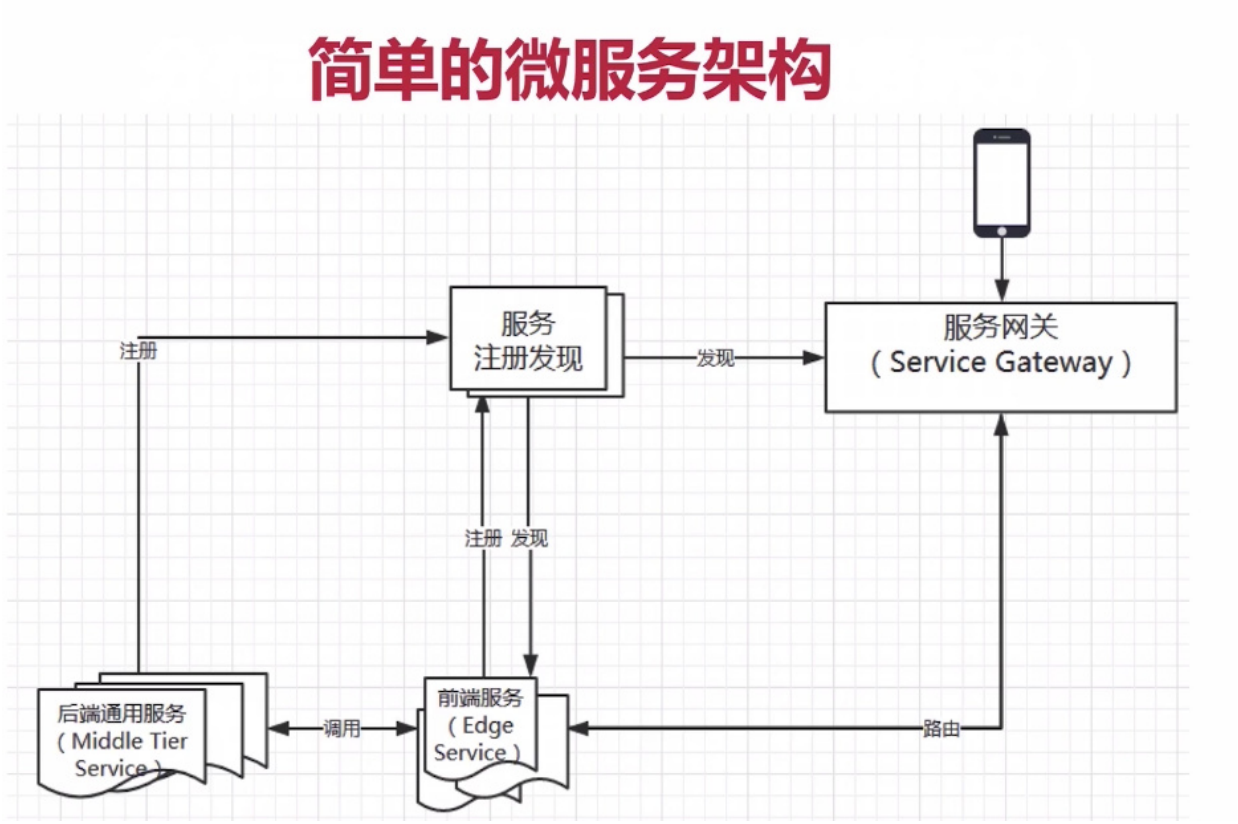
1. 开发效率低
2. 代码维护困难
3. 部署不灵活(构建时间长)
4. 稳定性不高
5. 扩展性不够(高并发和)

分布式架构

旨在支持应用程序和服务的开发, 可以利用物理架构, 由多个**自治的处理元素**, 不共享主内存, 但**通过网络发送消息**合作.

— Leslie Lamport

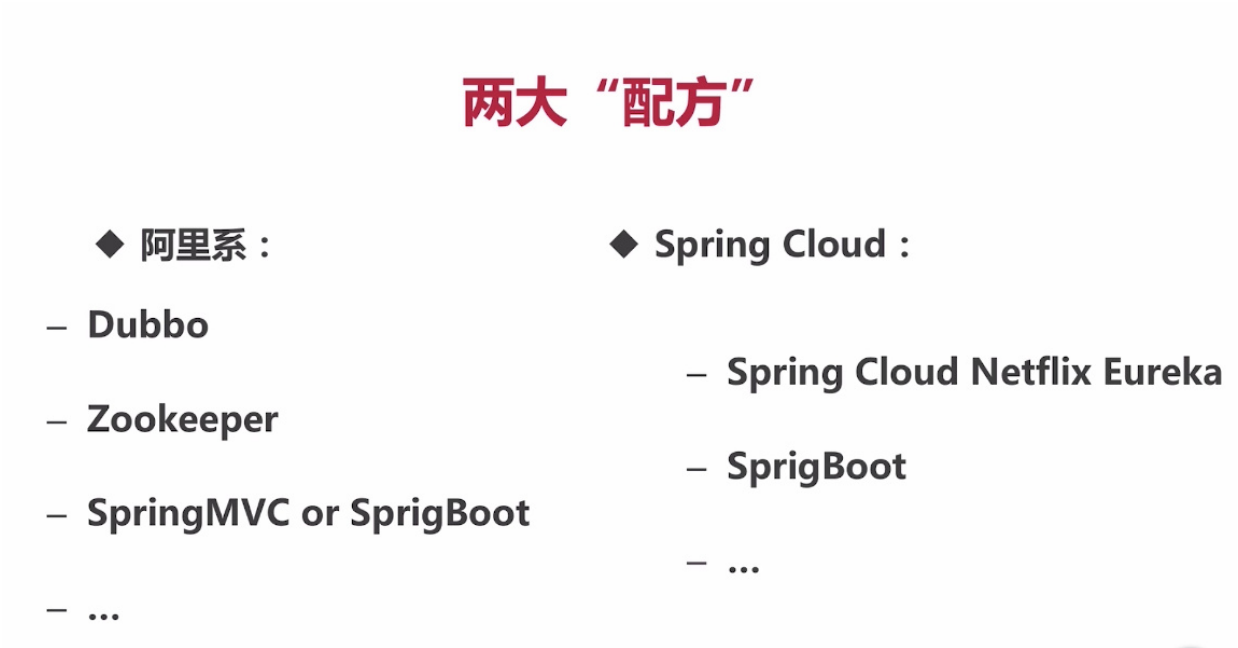
从一个极简的微服务框架开始



微服务架构的基础框架/组件

- 服务注册发现
- 服务网关
- 后端通用服务
- 前端服务

我服务的实现方式(两大配方)



SpringCloud 是什么?

SpringCloud 是一个开发工具, 含了多个子项目,

-利用SpringBoot的开发便利

-主要是基于对Netflix开源组件的进一步封装(Netflix是微服务界的翘楚)

SpringCloud 的作用

简化了分布式服务开发, 极大的降低了微服务开发的门槛

第三章 Eureka

简介

- 基于 Netflix Eureka 做了二次封装,
- 由两个组件组成
  - Eureka Server 注册中心
  - Eureka Client 注册服务

注册中心 Eureka Server

SpringCloud 和 SpringBoot 对应的关系

https://spring.io/projects —> SpringCloud —> 最下面的对应表格

开发

1. 引入相应的 jar 包
2. 在启动类上添加 `@EnableEurekaServer` 表示这是一个注册中心服务的项目, 同时此注解也包含 `Client` 注解, 所以添加了 `@EnableEurekaServer` 后, 即表示当前项目即时 Server 又是 Client

Eureka Client

开发

1. 选择相应的 jar 包
2. 在启动类上添加 `@EnableDiscoveryClient` 注解
3. 配置注册的地址

实现 Eureka 的高可用

新开启一个或多个 Eureka 服务, 使其互相注册

这时, 如果我们把 client 注册在其中一个 server 上时, client 会注册在其相互注册的其他 server 中. 当 client 重启后, 其他未被注册的 server 便不能再次被注册.

服务注册中心的原理和地位

在分布式系统中, 服务注册中心是最重要的基础部分

客户端发现

定义

当客户端 A 需要服务 B 的服务时, 回去注册中心调用, 注册中心会吧所有 B 的地址告诉 A, 由 A 去选择调用那个 B

优点

可控性强, 对服务 B 的控制度高

缺点

需要自己实现逻辑去挑选 B

代表

Eureka

服务端发现

定义

由代理在众多可用的 B 里边挑选出一个, 让 A 去使用,

优点

不需要调用者 A 实现调用逻辑, 同时由于代理的接入, 服务提供者和注册中心对 A 是透明不可见的.

代表

Nginx

Zookeeper

Kubernetes

异构

可以使用不同的语言不同的数据库来构建整体服务, 从这一点上, 就能比较好的理解, 为什么 SpringCloud 的调用方式选择 Http 的 restfull 接口的形式调用, 而不是像 dubbo 那样选择 RPC 的调用方式.

由于 rest 调用的客户端实现比较简单, 其他语言实现相关的调用也很方便

第四章 服务拆分

业务拆分的起点和终点

业务形态不适合做微服的项目

- 系统中包含很多强事物场景
- 业务相对稳定, 迭代周期长
- 访问压力不大, 可用性不高

康威定律和微服务

任何组织在设计一套系统(广义概念上的系统)时, 所交付的设计方案在结构上都与该组织的沟通结构保持一致.

> 沟通的问题, 会影响系统的设计
>
> 业务架构, 总是和团队内部的组织架构相匹配

点餐业务的服务拆分

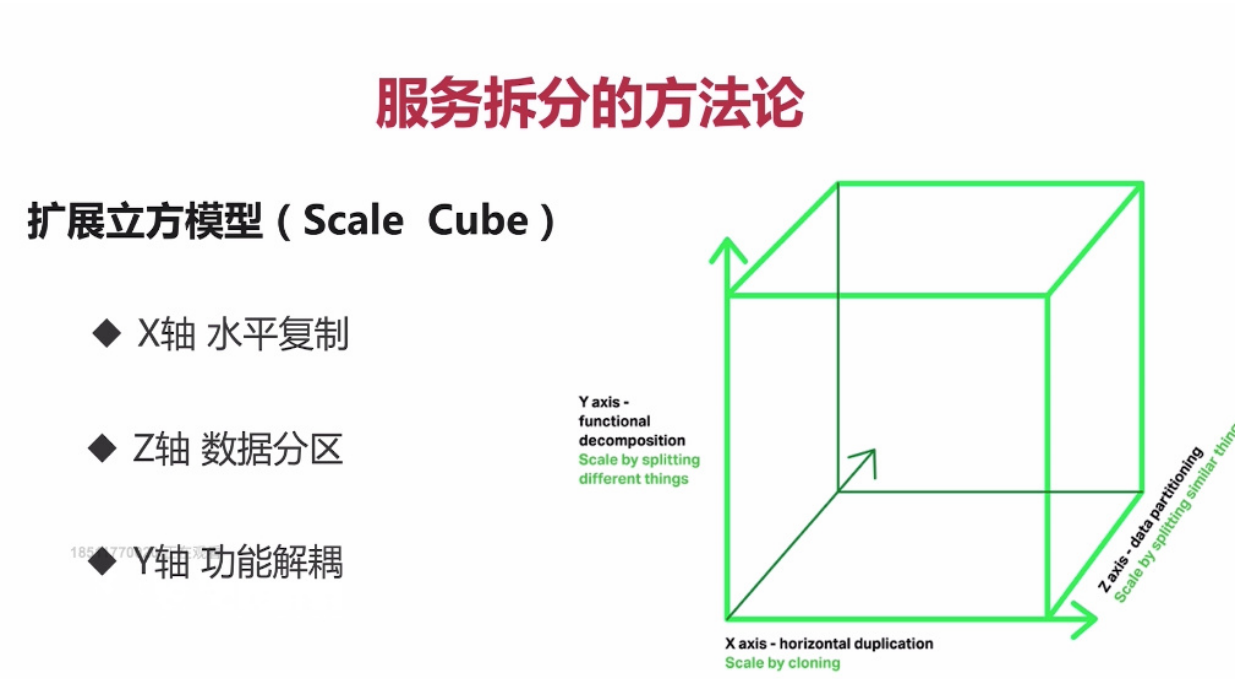
服务的拆分方式

1. 根据技术栈拆分
2. 根据业务拆分

拆分时需要考虑的因素

- 起点: 现有业务的情况, 现有项目的结构, 业务的发展预期
- 团队结构: 团队人员质量, 团队人数
- 沟通方式: 沟通效率和流畅度

服务拆分方法论



如何拆分功能

- 单一职责, 松耦合, 高内聚
- 关注点分离

    - 按职责 (容易拆分的业务)
    - 按通用性 (基础组件)
    - 按粒度级别

服务和数据的关系

- 优先考虑业务功能, 再考虑数据
- 无状态服务 (状态: 如果一个数据需要被多个服务共享才能完成一个请求, 则此数据被称为状态, 进而依赖这个状态的服务称为有状态服务, 反之称为无状态服务)
- 拆分之前, 要把有状态的业务服务, 改为无状态的业务服务

商品服务业务编码

技术点

lombok 工具插件的使用

ProductController 中 lambda 表达式封装集合的方法

Junit 测试类, 继承方式的使用

springdata jpa in 语句查询

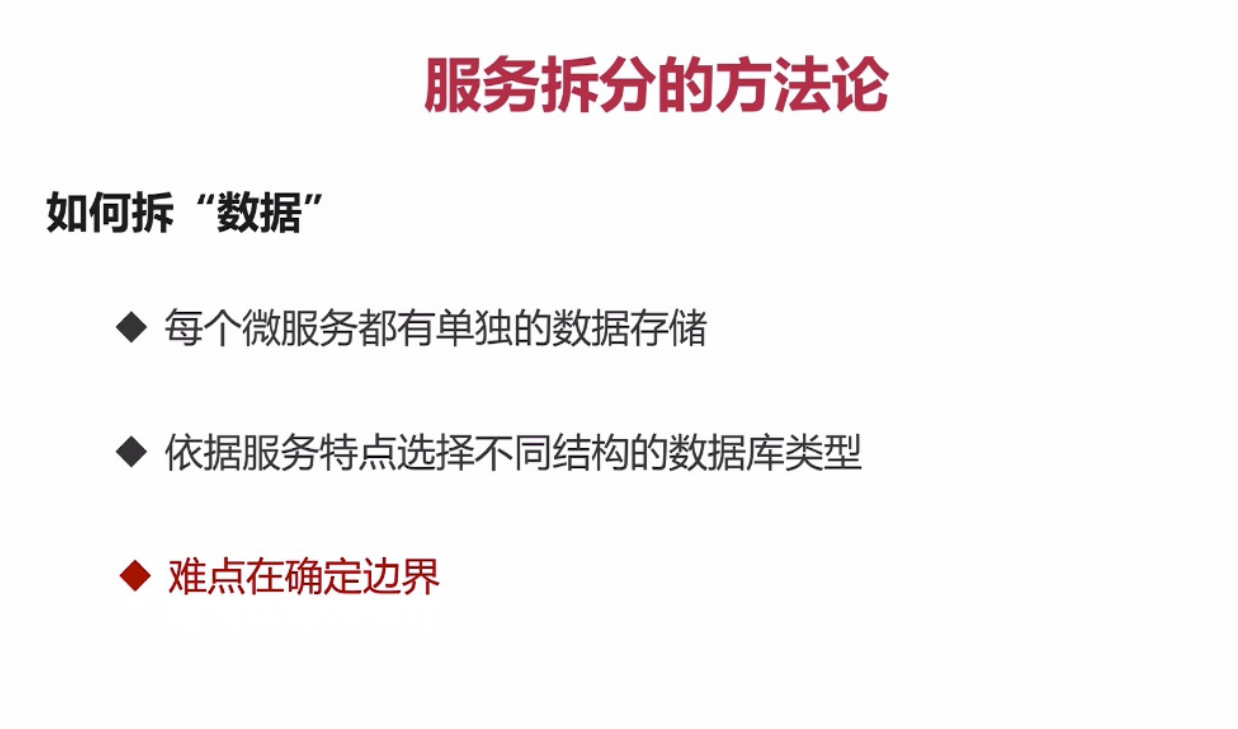
订单服务编码

技术点

`OrderForm2OrderDTOConverter` 中使用 gson 进行对象的转换

在预期可能出错的地方需要些 try catch , 此时需要抛出异常, 并记录日志

拆数据



第五章 应用通讯

HTTP vs RPC

应用间的通讯, 当前有两种方式 `http` 和 `rpc` 其代表分别为 `SpringCloud` 和 `dubbo`

dubbo 的优势: 服务治理集成上非常完善, 提供了服务注册发现, 负载均衡, 路由, 还设计了面向测试开发的 moke 泛化调用的机制. 提供服务治理和监控的可视化平台.

SpringCloud 的优势: 使用 http restfull 交互, 本身轻量易用,适用性强, 跨语言, 跨平台

Dubbo 和 SpringCloud 服务对比:

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task
.....	.....	.....

RestTemplate 的三种调用方式

直接通过路径发起请求: 指定 ip 和端口号以及请求路径.

详情请见 order Controller 中的 ClientController

负债均衡器 Ribbon

RestTemplate, Feign, Zuul 都使用了 Ribbon 的负载均衡

Ribbon 实现负载均衡的核心

- 服务发现: 依据服务的名字, 把该服务下所有的事例均找出来
- 服务选择规则: 如何选择
- 服务监听: 检测失效的服务

主要组件

ServerList

IRule

ServerListFilter

通过 ServerList 获取所有的可用列表 通过 ServerListFilter 过滤掉一部分地址, 通过 IRule选择一个实例, 作为最终实例结果

Ribbon 选择服务的规则

默认获取服务的规则是轮询, 可以在客户端通过配置改变相应的规则

```
# 修改获取服务的方式，默认是轮询
# 服务的名称(可自己定义)
PRODUCT:
  # ribbon 获取服务
  ribbon:
    # 获取富的方式的类
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

Feign 的使用

- 调用方添加依赖
- 调用方启动主类上添加注解 `EnableFeignClients`
- 定义好需要调用的接口

Feign 介绍

- 声明式 REST 客户端 (伪 RPC)
- 采用了基于接口的注解
- 内部也使用了 Ribbon 做负载均衡

使用时的注意点:

- 路径, 一定要是完整的全路径
- 当传递参数时使用了 @RequestBody 注解, 则需要使用 POST 请求

  > 无参, @RequestParam 传递单个参数, @PathVariable 路径的方式传递参数时, 才可以使用 Get 请求

项目改造成多模块

多模块的推荐课程 : Spring Boot 2.0 深度实践-初遇Spring Boot —— 小马哥

同步 or 异步

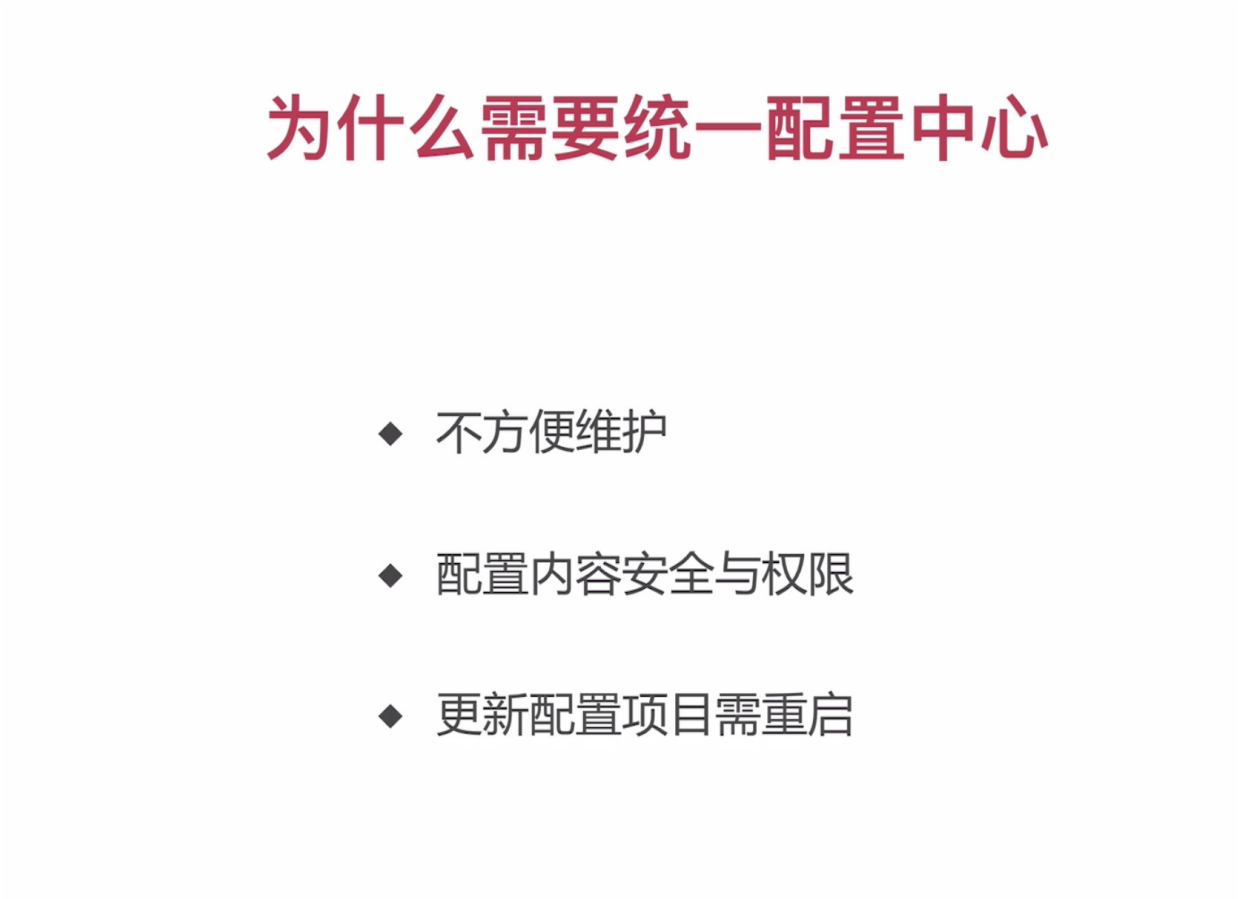
消息中间件

消息中间件推荐课程: Java 消息中间件

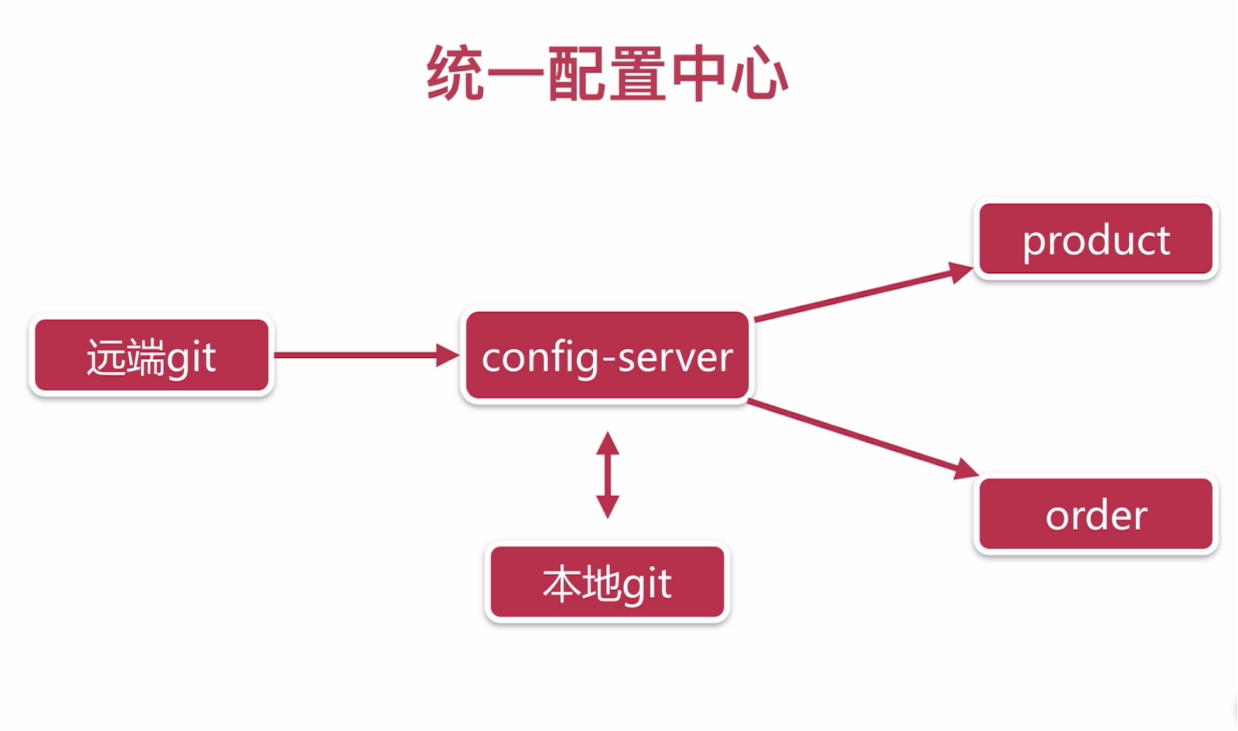
第六章 统一配置中心

配置中心概述

为什么需要统一配置中心?



实现的架构



编码 & 测试

Server 端开发

- 创建 `SpringBoot` 项目
  - 添加 `eureka` 客户端相关 `jar` 包: 作为 `SpringCloud` 项目的一个服务项目
  - 添加 `config server` 相关 `jar` 包: 是 `SpringCloud` 的 `config` 服务端
  - 在启动类上添加 `@EnableConfigServer` 注解: 标明当前项目是 `Config` 的 `Server` 项目
  - 在 `yml` 文件中配置注册中心, 配置 `config` 文件的 `git` 厂库地址(注: http路径, 并非 `git clone` 地址)
- 维护配置文件
  - 创建配置文件项目: 项目中写入所需的配置文件
- 测试方式
  - 直接请求 `Config Server` 的 `服务地址[:端口] 相关的配置文件的全称`, 若能看到配置文件的内容, 说明测试成功
  - 请求路径解释: `/{label}/{name}-{profiles}.yml`
    - name: 文件的名称
    - profiles: 环境
    - label: 分支
  - 注意: 文件名称的结构是 `文件名-profiles.yml/json/properties` 格式可以自动转换

Client 端开发

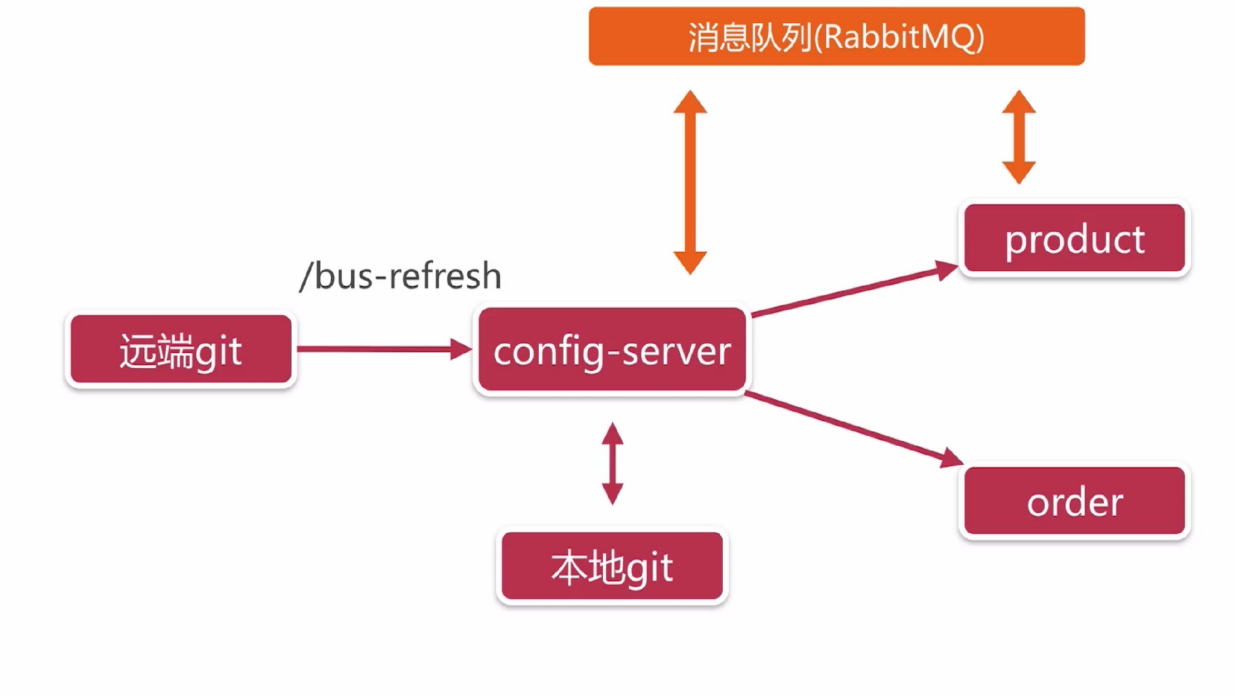
- 引入注解

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-client</artifactId>
</dependency>
```

- 在 `.yml` 文件中添加相应的配置信息.
- 修改 `application.yml` 为 `bootstrap.yml` .
- 注意:
  - 统一注册中心等前置配置, 需要写在本地
  - 客户端在加载时如果想要获取的是 `{name}-{profiles}.yml` 服务器在返回数据的时候会联同 `{name}.yml` 文件一同返回,

SpringCloud Bus 自动更新配置

原理



- 准备远端 `git` 项目, 用于提供配置资源
- `config-server` 用于配置的调度,
  - 从远端获取配置内容存储到本地,
  - 从本地获取配置内容
  - 发送配置信息给统一配置的客户端
  - 发送 MQ 消息给客户端, 让客户端调用新的配置文件
- 本地 `git` 在没有网络的时候, 用于保证项目正常运行的缓存配置内容
- 消息队列: 用于配置中心的 `server` 端和 `client` 端的通讯

编码 & 测试

Config Server 编码

添加相关 `jar`

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

<!--springcloud 2.0 正式版中没有提供 monitor 的相关接口-->
<!--ps: 此接口用于 git 的 webhooke 调用, 用于通知 config server 发送消息给客户端,-
->
<!--以实现配置的自动刷新-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-monitor</artifactId>
</dependency>
```

添加后启动项目, 会发现 RabbitMQ 中会自动添加一个队列

> 如果 RabbitMQ 有密码, 需要配置相关的密码

在 `config server` 启动类上添加注解 `@EnableConfigServer`

Config Client 编码

添加相关的 `jar`

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

不需要添加其他注解

刷新配置

访问 `config` 的地址 `/actuator/bus-refresh` 即可完成刷新.

同时, 如果想要 config 客户端起效果, 还需要在对应的配置文件的类上添加 `@RefreshScope` 注解.

例如:

```
/**
 * @author summer
 * @date 2018/4/9 下午6:18
 */
@Data
@Component
@ConfigurationProperties("girl")
@RefreshScope
public class GirlConfig {
```

```
    private String name;

    private Integer age;
}
```

git 自动刷新配置

如果想使用 `git` 刷新配置, 则需要用到 `git` 的 `Webhooks` 功能. SringCloud 提供给 `git Webhook` 的地址是 `/monitor`

第七章 消息和异步

异步和消息

异步的常见形态

- 通知: 单项请求
- 请求/异步相应: 客户端调用服务端, 服务端不会立刻相应, 默认相应不会立刻送达
- 消息: 可实现一对多的模式

MQ 应用场景

- 异步处理
- 流量削峰
- 日志处理 代表者 kafka
- 应用解耦

RabbitMQ 的简单使用

SpringCloud Stream 的使用

添加相关 `jar` 文件

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

Spring Cloud Stream