从单体架构、到SOA、再到微服务的架构设计详解

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：单体架构 SOA 微服务

题目描述

从单体架构、到SOA、再到微服务的架构设计详解

1. 面试题分析

根据题目要求我们可以知道：

1、单体架构

2、单体架构的拆分

3、SOA与微服务的区别

4、微服务的优缺点

5、微服务的消息

6、服务集成

7、数据的去中心化

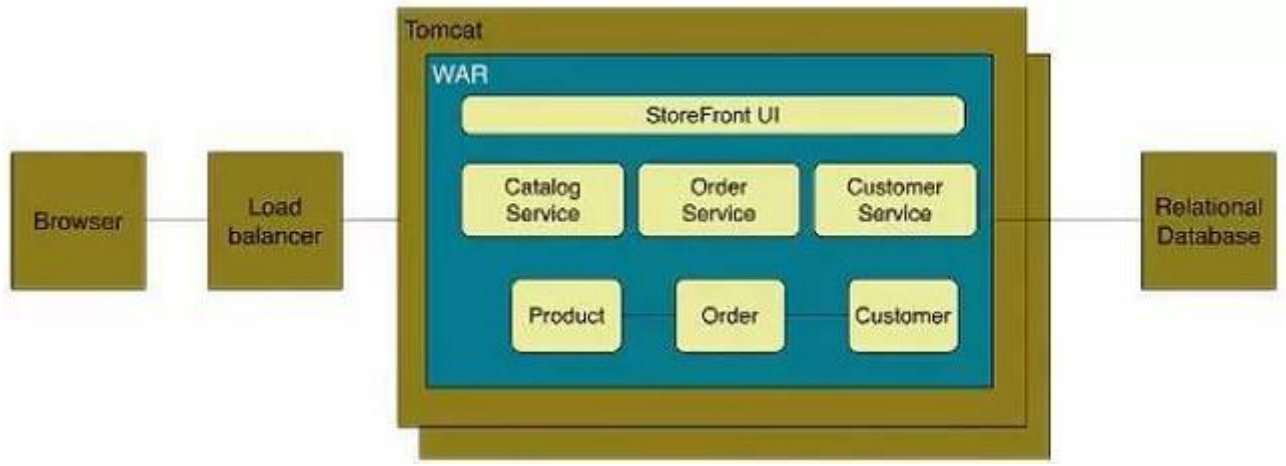
容易被忽略的坑

- 分析片面
- 没有深入

1.单体架构

Web应用程序发展的早期，大部分web工程是将所有的功能模块(service side)打包到一起并放在一个web容器中运行，很多企业的Java应用程序打包为war包。其他语言(Ruby，Python或者C++)写的程序也有类似的问题。

假设你正在构建一个在线商店系统：客户下订单、核对清单和信用卡额度，并将货物运输给客户。很快，你们团队一定能构造出如下图所示的系统。



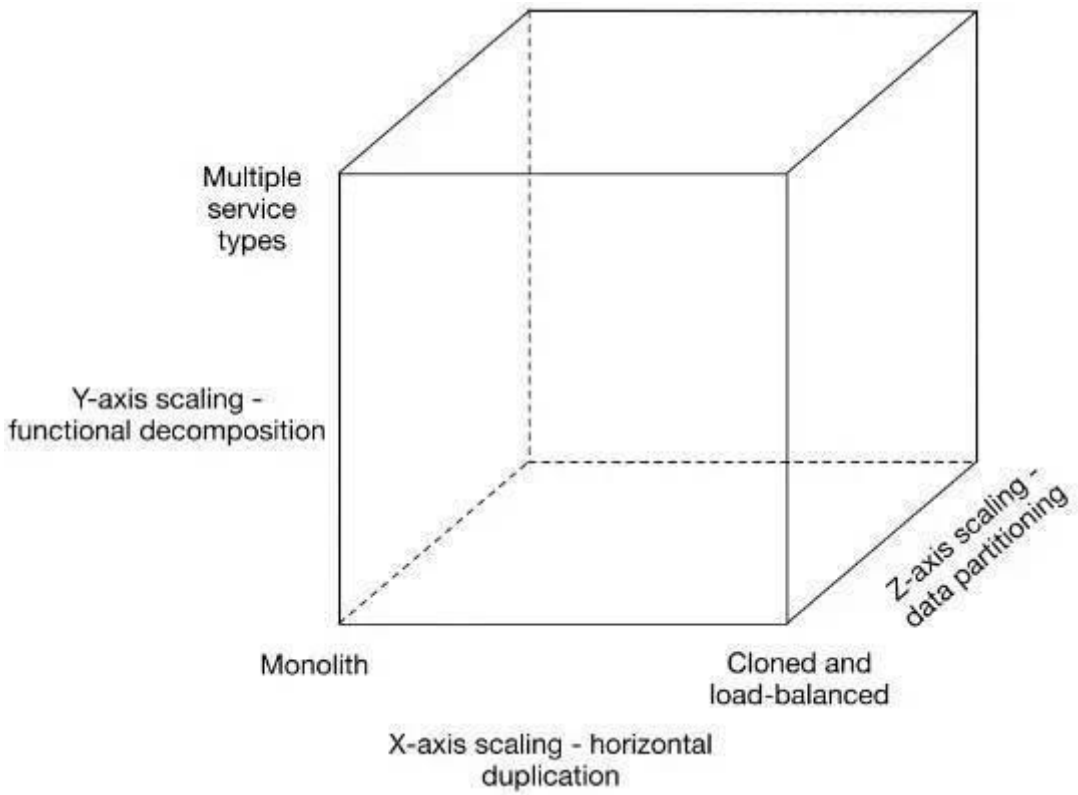
这种将所有功能都部署在一个web容器中运行的系统就叫做单体架构（也叫：巨石型应用）。

单体架构有很多好处：IDE都是为开发单个应用设计的、容易测试——在本地就可以启动完整的系统、容易部署——直接打包为一个完整的包，拷贝到web容器的某个目录下即可运行。

但是，上述的好处是有条件的：应用不那么复杂。对于大规模的复杂应用，单体架构应用会显得特别笨重：要修改一个地方就要将整个应用全部部署(PS：在不同的场景下优势也变成了劣势);编译时间过长;回归测试周期过长;开发效率降低等。另外，单体架构不利于更新技术框架，除非你愿意将系统全部重写(代价太高你愿意老板也不愿意)。

2.单体架构的拆分

详细一个网站在业务大规模爬升时会发生什么事情?并发度不够?OK，加web服务器。数据库压力过大?OK，买更大更贵的数据库。数据库太贵了?将一个表的数据分开存储，俗称“分库分表”。这些都没有问题，good job。不过，老外的抽象能力比我们强，看下图Fig2。



这张图从三个维度概括了一个系统的扩展过程：

(1)x轴，水平复制，即在负载均衡服务器后增加多个web服务器;

(2)z轴扩展，是对数据库的扩展，即分库分表(分库是将关系紧密的表放在一台数据库服务器上，分表是因为一张表的数据太多，需要将一张表的数据通过hash放在不同的数据库服务器上);

(3)y轴扩展，是功能分解，将不同职能的模块分成不同的服务。从y轴这个方向扩展，才能将巨型应用分解为一组不同的服务，例如订单管理中心、客户信息管理中心、商品管理中心等等。

3.SOA与微服务

SOA:服务导向式架构（SOA）是集成多个较大组件（一般是应用）的一种机制，它们将整体构成一个彼此协作的套件。一般来说，每个组件会从始至终执行一块完整的业务逻辑，通常包括完成整体大action所需的各种具体任务与功能。组件一般都是松耦合的，但这并非SOA架构模式的要求。

微服务：**是一种架构设计模式。**

在微服务架构中，业务逻辑被拆分成一系列小而松散耦合的分布式组件，共同构成了较大的应用。每个组件都被称为微服务，而每个微服务都在整体架构中执行着单独的任务，或负责单独的功能。每个微服务可能会被一个或多个其他微服务调用，以执行较大应用需要完成的具体任务；系统还为任务执行——比如搜索或显示图片任务，或者其他可能需要多次执行的任务提供了统一的解决处理方式，并限制应用内不同地方生成或维护相同功能的多个版本。

1）. 负责单个功能

2）. 单独部署

3）. 包含一个或多个进程

4）. 拥有自己的数据存储

5）. 一支小团队就能维护几个微服务

6）. 可替换的

相对于SOA，区别如下：

功能	SOA	微服务
组件大小	大块业务逻辑	单独任务或小块业务逻辑
耦合	通常松耦合	总是松耦合
公司架构	任何类型	小型、专注于功能交叉的团队
管理	着重中央管理	着重分散管理
目标	确保应用能够交互操作	执行新功能，快速拓展开发团队

SOA尝试将应用集成，一般采用中央管理模式来确保各应用能够交互运作。微服务尝试部署新功能，快速有效地扩展开发团队。它着重于分散管理、代码再利用与自动化执行。

4.微服务的优缺点

微服务架构模式有很多好处。

第一，通过分解巨大单体应用为多个服务方法解决了复杂性问题。

在功能不变的情况下，应用被分解为多个可管理的分支或服务。每个服务都有一个用 RPC- 或者消息驱动 API 定义清楚的边界。

第二，这种架构使得每个服务都可以有专门开发团队来开发。

开发者可以自由选择开发技术，提供 API 服务。

第三，微服务架构模式使得每个微服务独立部署，开发者不再需要协调其它服务部署对本服务的影响。**

最后，微服务架构模式使得每个服务独立扩展。你可以根据每个服务的规模来部署满足需求的实利。

微服务架构也有不足。其中一个跟他的名字类似，“微服务”强调了服务大小，实际上，有一些开发者鼓吹建立稍微大一些的，10-100 LOC服务组。尽管小服务更乐于被采用，但是不要忘了微服务只是结果，而不是最终目的。微服务的目的是有效的拆分应用，实现敏捷开发和部署。

另外一个不足之处在于，微服务应用是分布式系统，由此会带来固有的复杂性。开发者需要在 RPC 或者消息传递之间选择并完成进程间通讯机制。此外，他们必须写代码来处理消息传递中速度过慢或者不可用等局部失效问题。

另外一个关于微服务的挑战来自于分区的数据库架构。同时更新多个业务主体的事务很普遍。这种事务对于单体式应用来说很容易，因为只有一个数据库。在微服务架构应用中，需要更新不同服务所使用的不同的数据库。

测试一个基于微服务架构的应用也是很复杂的任务。另外一个挑战在于，微服务架构模式应用的改变将会波及多个服务。部署一个微服务应用也很复杂，一个单体应用只需要在复杂均衡器后面部署各自的服务器就好了。每个应用实例是需要配置诸如数据库和消息中间件等基础服务。每个服务都有多个实例，这就形成大量需要配置、部署、扩展和监控的部分。除此之外，你还需要完成一个服务发现机制（后续文章中发表），以用来发现与它通讯服务的地址（包括服务器地址和端口）

5.微服务消息

1）同步消息 – REST, Thrift

同步消息就是客户端需要保持等待，直到服务器返回应答。REST是微服务中默认的同步消息方式，它提供了基于HTTP协议和资源API风格的简单消息格式，多数微服务都采用这种方式（每个功能代表了一个资源和对应的操作）。

Thrift是另外一个可选的方案。它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。

2）异步消息 – AMQP, STOMP, MQTT

异步消息就是客户端不需要一直等待服务应答，有应到后会得到通知。某些微服务需要用到异步消息，一般采用AMQP, STOMP, MQTT。

3）消息格式 – JSON, XML, Thrift, ProtoBuf, Avro

消息格式是微服务中另外一个很重要的因素。SOA的web服务一般采用文本消息，基于复杂的消息格式（SOAP）和消息定义（xsd）。微服务采用简单的文本协议JSON和XML，基于HTTP的资源API风格。如果需要二进制，通过用到Thrift, ProtoBuf, Avro。

4）服务约定 – 定义接口 – Swagger, RAML, Thrift IDL

如果把功能实现为服务，并发布，需要定义一套约定。单体架构中，SOA采用WSDL，WSDL过于复杂并且和SOAP紧耦合，不适合微服务。

REST设计的微服务，通常采用Swagger和RAML定义约定。

对于不是基于REST设计的微服务，比如Thrift，通常采用IDL（Interface Definition Languages），比如Thrift IDL。

6.服务集成

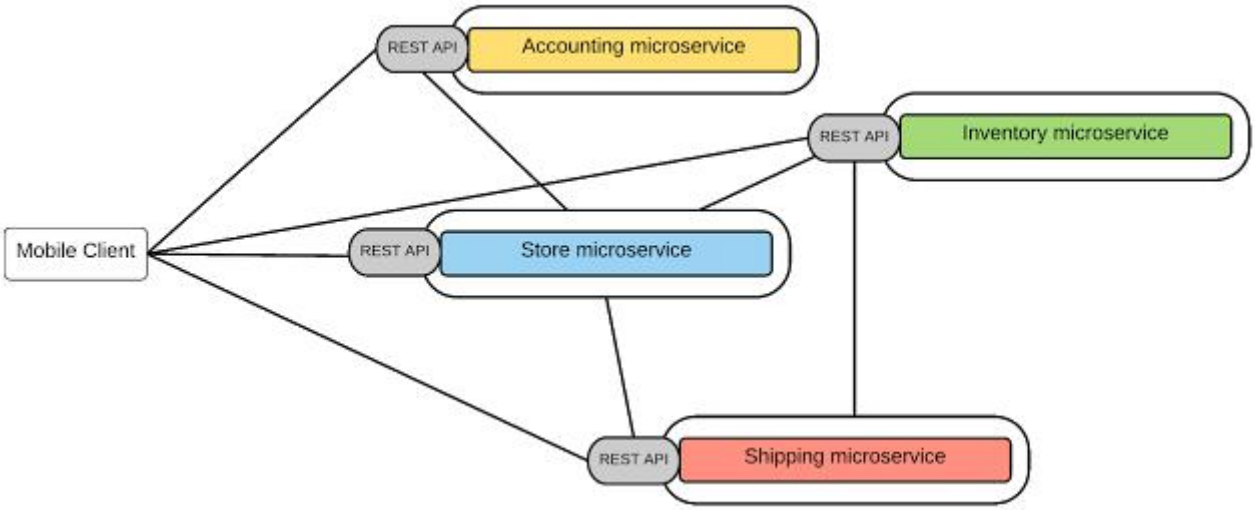
SOA体系下，服务之间通过企业服务总线（Enterprise Service Bus）通信，许多业务逻辑在中间层（消息的路由、转换和组织）。

微服务架构倾向于降低中心消息总线（类似于ESB）的依赖，将业务逻辑分布在每个具体的服务终端。

大部分微服务基于HTTP、JSON这样的标准协议，集成不同标准和格式变的不再重要。另外一个选择是采用轻量级的消息总线或者网关，有路由功能，没有复杂的业务逻辑。下面就介绍几种常见的架构方式。

1）、点对点方式 – 直接调用服务

点对点方式中，服务之间直接用。每个微服务都开放REST API，并且调用其它微服务的接口。



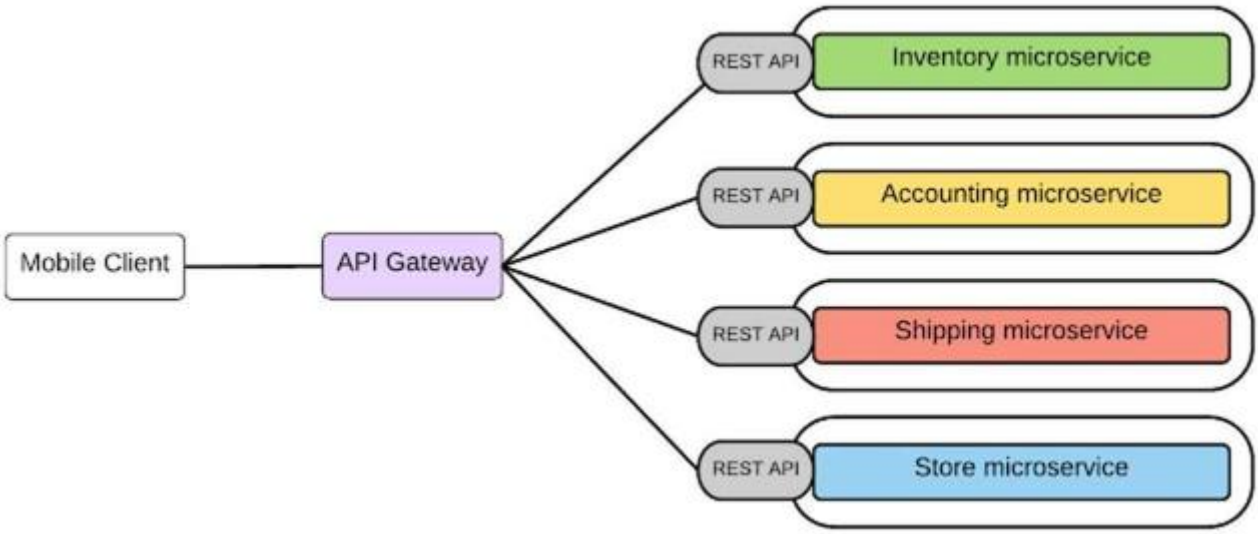
图：通过点对点方式通信

很明显，在比较简单的微服务应用场景下，这种方式还可行，随着应用复杂度的提升，会变得越来越不可维护。这点有些类似SOA的ESB，尽量不采用点对点的集成方式。

2）、API-网关方式

API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能个。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GW注册和管理服务。

图：通过API-网关暴露微服务



用我们网上商店的例子，在图5中，所有的业务接口通过API网关暴露，是所有客户端接口的唯一入口。微服务之间的通信也通过API网关。

采用网关方式有如下优势：

a.有能力为微服务接口提供网关层次的抽象。比如：微服务的接口可以各种各样，在网关层，可以对外暴露统一的规范接口。

b.轻量的消息路由、格式转换。

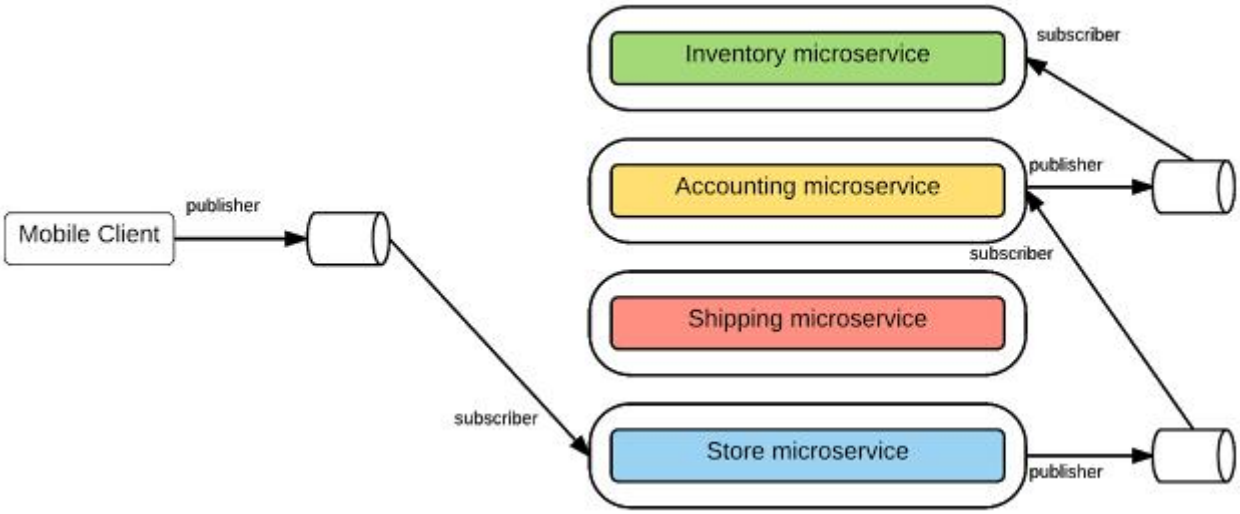
c.统一控制安全、监控、限流等非业务功能。

d.每个微服务会变得更加轻量，非业务功能个都在网关层统一处理，微服务只需要关注业务逻辑

目前，API网关方式应该是微服务架构中应用最广泛的设计模式。

3）、消息代理方式

微服务也可以集成在异步的场景下，通过队列和订阅主题，实现消息的发布和订阅。一个微服务可以是消息的发布者，把消息通过异步的方式发送到队列或者订阅主题下。作为消费者的微服务可以从队列或者主题共获取消息。通过消息中间件把服务之间的直接调用解耦。

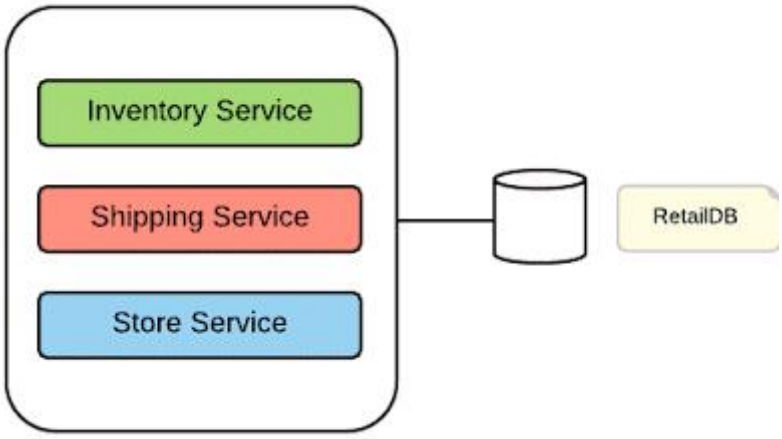


图：异步通信方式

通常异步的生产者/消费者模式，通过AMQP、MQTT等异步消息规范。

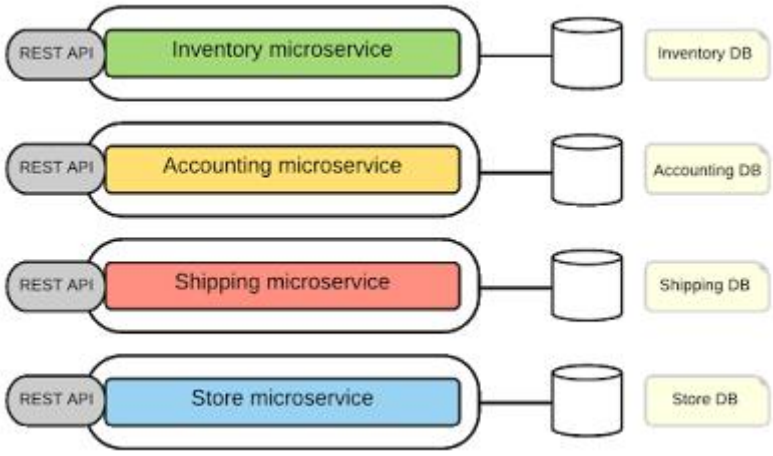
7.数据去中心化

单体架构中，不同功能的服务模块都把数据存储在某个中心数据库中。



图：单体架构，用一个数据库存储所有数据

微服务方式，多个服务之间的设计相互独立，数据也应该相互独立（比如，某个微服务的数据库结构定义方式改变，可能会中断其它服务）。因此，每个微服务都应该有自己的数据库。



图：每个微服务有自己私有的数据库，其它微服务不能直接访问。

数据去中心话的核心要点：

1）、每个微服务有自己私有的数据库持久化业务数据

2）、每个微服务只能访问自己的数据库，而不能访问其它服务的数据库

3）、某些业务场景下，需要在一个事务中更新多个数据库。这种情况也不能直接访问其它微服务的数据库，而是通过对于微服务进行操作。

数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可以采用不同的数据库技术（SQL、NoSQL等）。在复杂的业务场景下，如果包含多个微服务，通常在客户端或者中间层（网关）处理。

写在最后，提及架构，没有绝对最好的架构，只有最适合业务的架构。技术架构应该从业务中来，到业务中去，其抽象于业务而高于业务。作为技术开发的我们，不能一味的追求架构而架构，我们必须在公司业务发展，团队资源中间做一个合适的选择，然后随着业务的发展逐步重构与优化。

8. 扩展内容

- Restful、SOAP、RPC、SOA、微服务之间的区别?
- 服务注册与发现的实现原理、及实现优劣势比较?
- 微服务Dubbo和SpringCloud架构设计、优劣势比较?