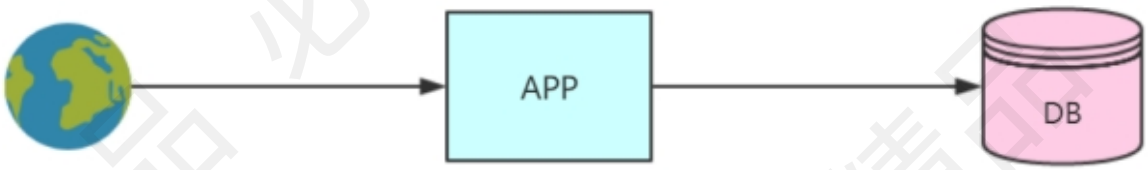
从架构演变谈微服务框架在分布式中的作用

1、架构演变过程

1.1、单体架构

我们最先接触的单体架构，整个系统就只有一个工程，打包往往是打成了war包，然后部署到单一tomcat上面，这种就是单体架构，如图：



这种架构适合比较小型的系统，比如以前的老系统，比如公司的内部OA系统等等，是一个典型的传统项目的结构。假如系统按照功能划分了**商品模块，购物车模块，订单模块，物流模块**等等模块。那么所有模块的代码都会在一个工程里面，这就是单体架构。

单体应用架构也就是大家在早期所学习的JavaEE知识SSH或者SSM架构模式，会采用分层架构模式：**数据库访问层**、**业务逻辑层**、**控制层**，从前端到后台所有的代码都是一个小的开发团队去完成。

该架构模式没有对我们业务逻辑代码实现拆分，所有的代码都写入到同一个工程中里面，适合于小公司开发团队或者个人开发。

这种架构模式最大的缺点，如果该系统一个模块出现不可用、会导致整个系统无法使用。

单体架构

优点

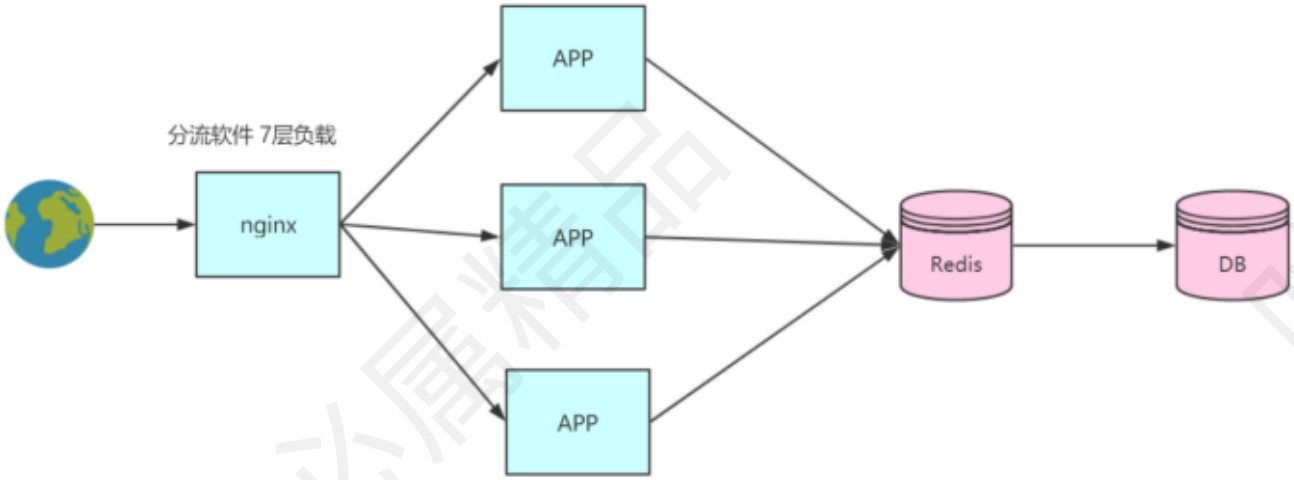
1. 结构简单，部署简单
2. 所需的硬件资源少
3. 节省成本

缺点

1. 版本迭代慢，往往改动一个代码会影响全局
2. 不能满足一定并发的访问
3. 代码维护困难，所有代码在一个工程里面，存在被其他人修改的风险

1.2、水平扩展架构

随着业务的拓展，公司的发展，单体架构慢慢的不能满足我们的需求，我们需要对架构进行变动，我们能够想到的最简单的办法就是加机器，对应用横向扩展。



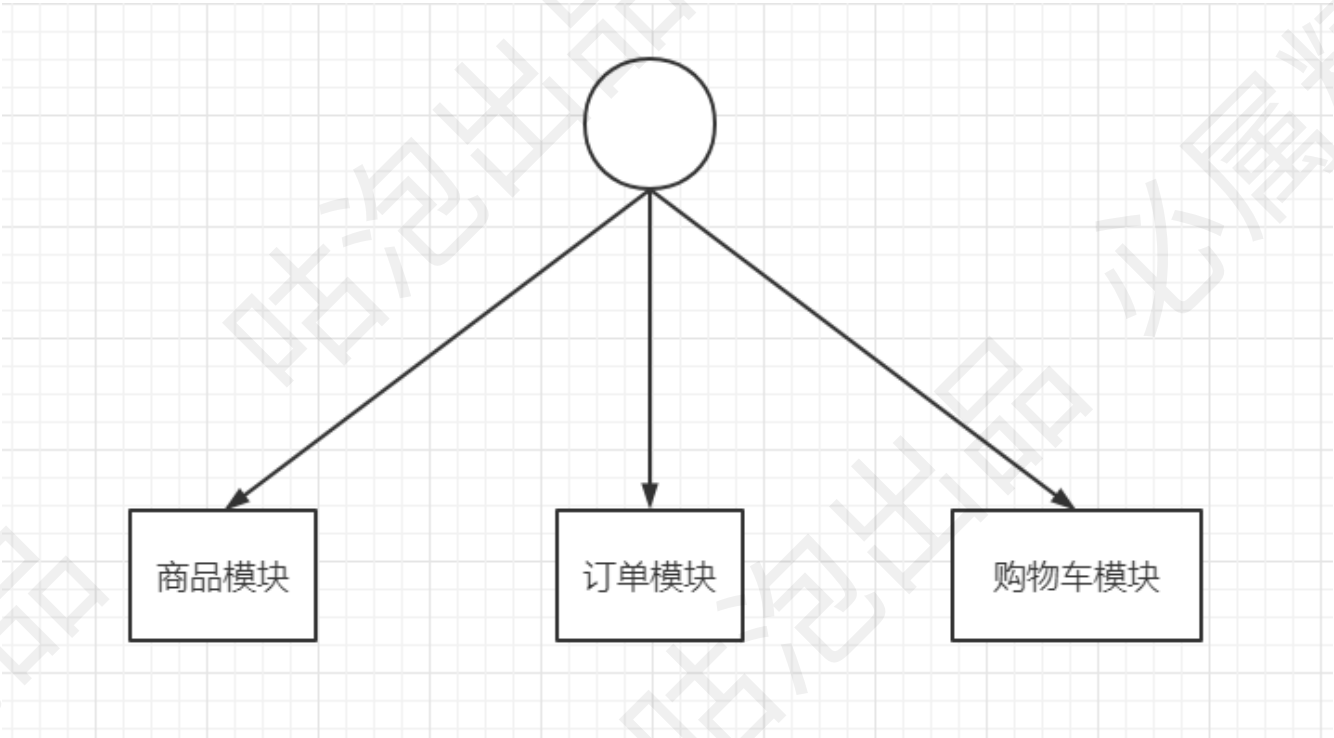
这种架构貌似暂时解决了我们的问题，但是用户量慢慢增加后，我们只能通过横向加机器来解决，还是会存在版本迭代慢，代码维护困难的问题。而且用户请求往往是读多写少的情况，所以可能真正需要扩容的只是商品模块而已，而现在是整个工程都扩容了，这无形中是一种资源的浪费，因为其他模块可能根本不需要扩容就可以满足需求。所以我们有必要对整个工程按照模块进行拆分

1.3、垂直应用架构

随着访问量的逐渐增大，单一应用只能依靠增加节点来应对，但是这时候会发现并不是所有的模块都会有比较大的访问量。

这时候单体应用的架构就不能够满足我们的需求，我们需要将系统里面的模块拆分开来，这样对于需要水平扩展的节点，是比较友好的。

比如，我的商品系统用户访问量变大，我就只需要增加商品系统的节点进行水平扩展就可以了。



优点

- 系统拆分实现了流量分担，解决了并发问题，而且可以针对不同模块进行优化和水平扩展

- 一个系统的问题不会影响到其他系统，提高容错率

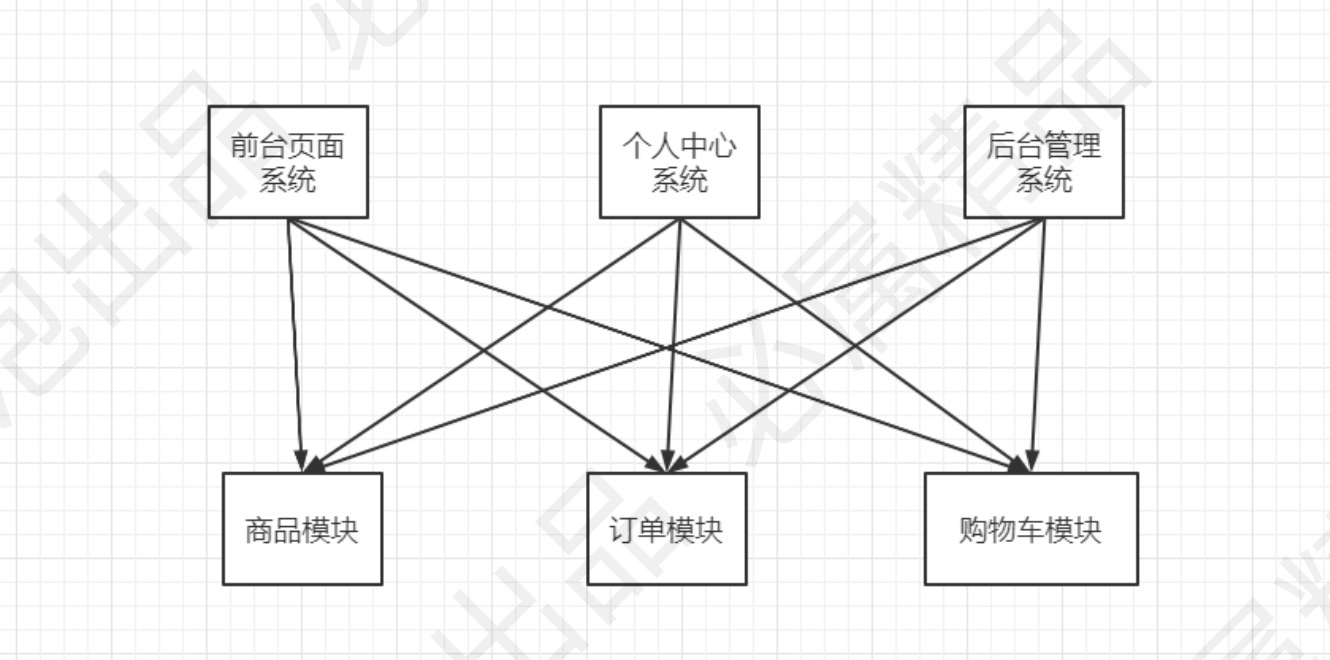
缺点

- 系统之间相互独立， 无法进行相互调用
- 系统之间相互独立， 会有重复的开发任务

1.4、分布式架构

分布式架构 也是基于垂直应用架构演变而来，当垂直应用越来越多，重复的业务代码就会越来越多。这时候，我们就思考可不可以将重复的代码抽取出来，做成统一的业务层作为独立的服务，然后由前端控制层调用不同的业务层服务呢?

这就产生了新的分布式系统架构。它将把工程拆分成表现层和服务层两个部分，服务层中包含业务逻辑。表现层只需要处理和页面的交互，业务逻辑都是调用服务层的服务来实现



优点

- 抽取公共的功能为服务层，提高代码复用性

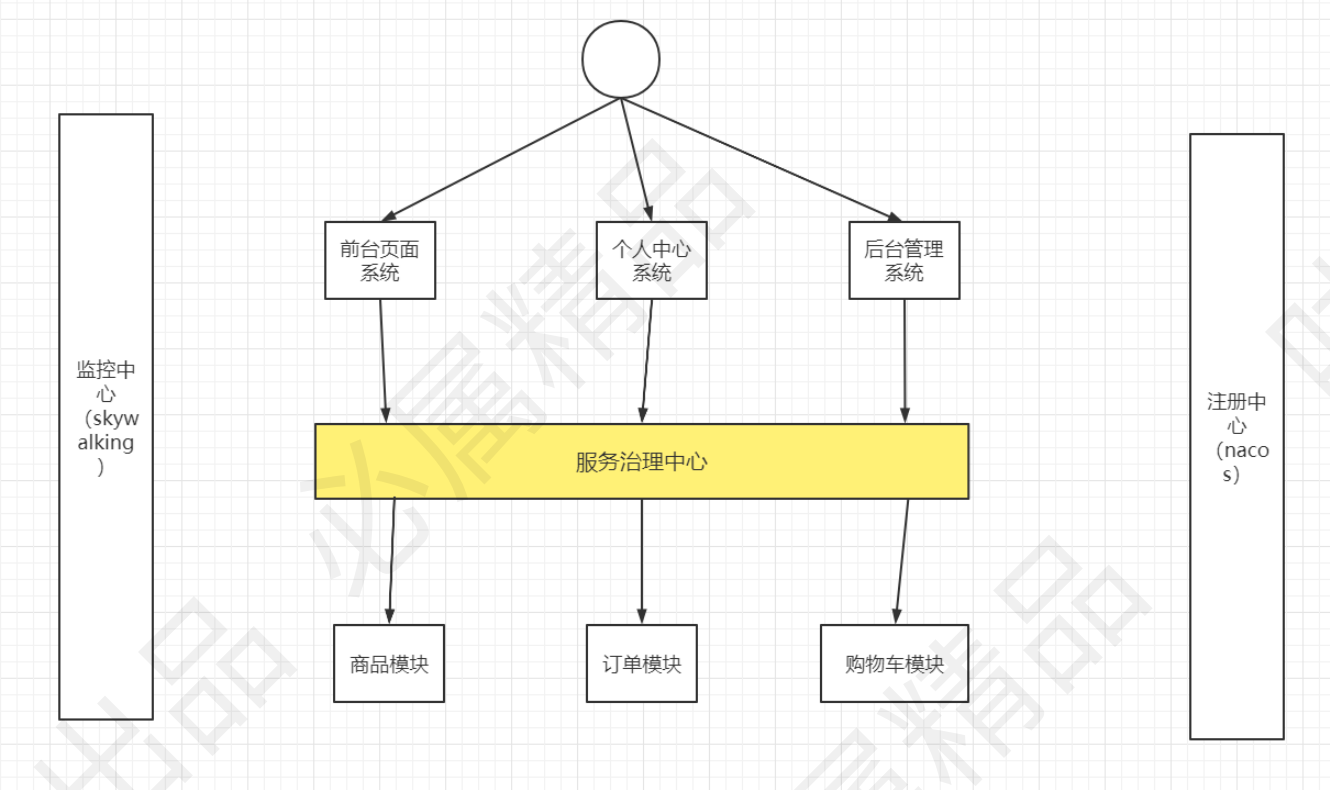
缺点

- 系统间耦合度变高，调用关系错综复杂，难以维护

1.5、微服务架构

在分布式架构下，当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心对集群进行实时管理，比如，上面分布式架构中的前台页面系统，需要访问下面多个服务，十几二十个呢？下面的用户服务也被十几二十个调用者调用?

此时，用于资源调度和治理中心(SOA Service OrientedArchitecture，面向服务的架构)是关键。



基于注册中心的SOA框架，扩展是非常方便的，因为不需要维护分流工具，但我们启动应用的时候就会把服务通过http的方式注册到注册中心。

在SOA框架中一般会有三种角色：1、注册中心 2、服务提供方 3、服务消费方

1、注册中心

在注册中心维护了服务列表

2、服务提供方

服务提供方启动的时候会把自己注册到注册中心

3、服务消费方

服务消费方启动的时候，把获取注册中心的服务列表，然后调用的时候从这个服务列表中选择某一个去调用。

微服务工程的特点：

1、扩展灵活

2、每个应用都规模不大

3、服务边界清晰，各司其职

打包应用变多，往往需要借助CI持续集成工具

2、微服务架构需要解决的问题

上面我们分析了架构的演变过程，随着系统变得越来越复杂，我们的服务之间的调用变得越来越复杂我们引入了微服务架构，那么在微服务架构下我们要解决什么问题呢？

2.1、服务注册发现

在一个分布式系统中，服务注册与发现组件主要解决两个问题：服务注册和服务发现。

服务注册：服务实例将自身服务信息注册到注册中心。这部分服务信息包括服务所在主机IP和提供服务的Port，以及暴露服务自身状态以及访问协议等信息。

服务发现：服务实例请求注册中心获取所依赖服务信息。服务实例通过注册中心，获取到注册到其中的服务实例的信息，通过这些信息去请求它们提供的服务。

除此之外，服务注册与发现需要关注监控服务实例运行状态、负载均衡等问题。

监控：微服务应用中，服务处于动态变化的情况，需要一定机制处理无效的服务实例。一般来讲，服务实例与注册中心在注册后通过心跳的方式维系联系，一旦心跳缺少，对应的服务实例会被注册中心剔除。

2.2、远程服务调用

服务消费者称为客户端，服务提供者称为服务端，两者通常位于网络上两个不同的地址，要完成一次 RPC 调用，就必须先建立网络连接。建立连接后，双方还必须按照某种约定的协议进行网络通信，这个协议就是通信协议。双方能够正常通信后，服务端接收到请求时，需要以某种方式进行处理，处理成功后，把请求结果返回给客户端。为了减少传输的数据大小，还要对数据进行压缩，也就是对数据进行序列化。

2.3、负载均衡

负载均衡建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

负载均衡*（Load Balance）*其意思就是分摊到多个操作单元上进行执行，例如Web服务器、FTP服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

2.4、熔断降级

在分布式架构中，各个服务节点一定需要满足高可用，所以对于服务本身来说，一方面是在有准备的前提下做好充足的扩容。另一方面，服务需要有熔断、限流、降级的能力。

当一个服务调用另外一个服务，可能因为网络原因、或者连接池满等问题导致频繁出现错误，需要有一种熔断机制，来防止因为请求堆积导致整个应用雪崩。

当发现整个系统的确负载过高的时候，可以选择降级某些功能或某些调用，保证最重要的交易流程的通过，以及最重要的资源全部用于保证最核心的流程。

在设置了熔断以及降级策略后，还有一种手段来保护系统，就是限流算法。

我们能够通过全链路压测了解到整个系统的吞吐量，但实际上的流量可能会超过我们预期的值，比如存在恶意攻击、或者突然的高峰流量。在这种情况下可以通过限流来保护系统不崩溃，但是对于部分用户来说，会出现被限流导致体验不好的情况。

2.5、分布式消息

2.6、配置中心

服务拆分以后，服务的数量非常多，如果所有的配置都以配置文件的方式放在应用本地的话，非常难以管理，可以想象当有几百上千个进程中有一个配置出现了问题，是很难将它找出来的，因而需要有统一的配置中心，来管理所有的配置，进行统一的配置下发。

在微服务中，配置往往分为几类，一类是几乎不变的配置，这种配置可以直接打在容器镜像里面，第二类是启动时就会确定的配置，这种配置往往通过环境变量，在容器启动的时候传进去，第三类就是统一的配置，需要通过配置中心进行下发，例如在大促的情况下，有些功能需要降级，哪些功能可以降级，哪些功能不能降级，都可以在配置文件中统一配置。

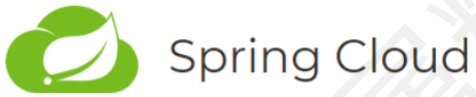
链路监控

3、springcloud的简介

2.1、什么是SpringCloud?

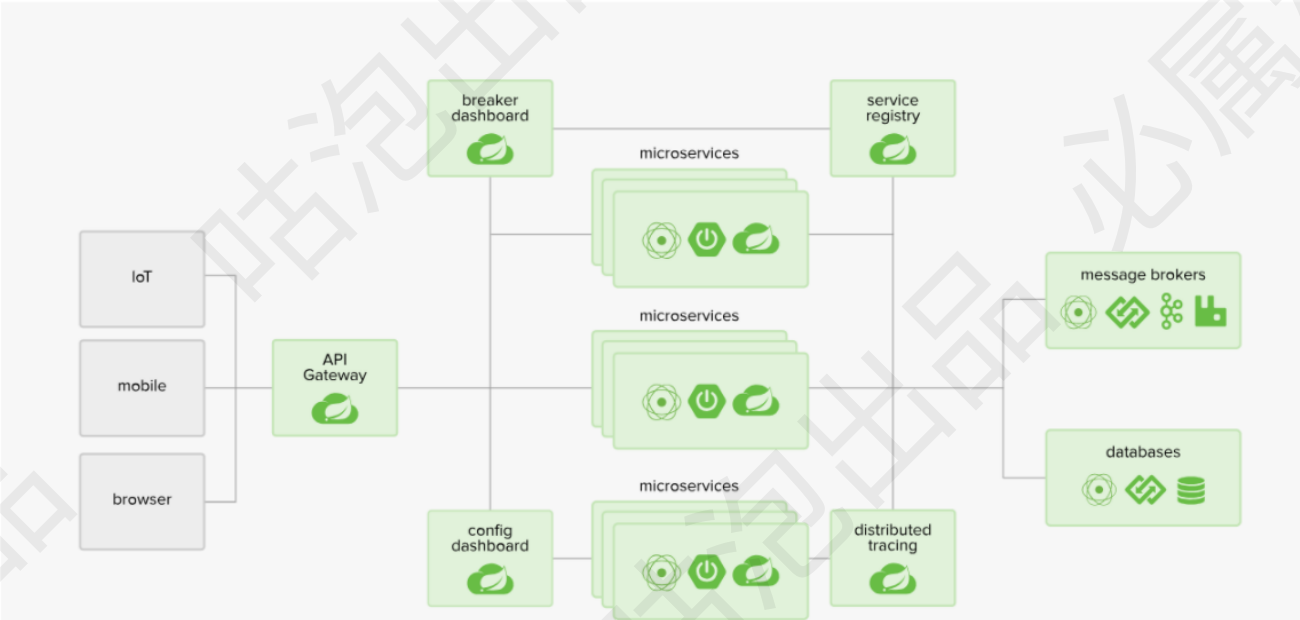
SpringCloud是分布式**微服务架构下的一站式解决方案**，是各个**微服务架构落地技术的集合体**，俗称**微服务全家桶。**

springCloud是基于SpringBoot的一整套实现微服务的框架。他提供了微服务开发所需的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等组件。最重要的是，跟spring boot框架一起使用的话，会让你开发微服务架构的云服务非常好的方便。 SpringBoot旨在简化创建产品级的 Spring 应用和服务，简化了配置文件，使用嵌入式web服务器，含有诸多开箱即用微服务功能。

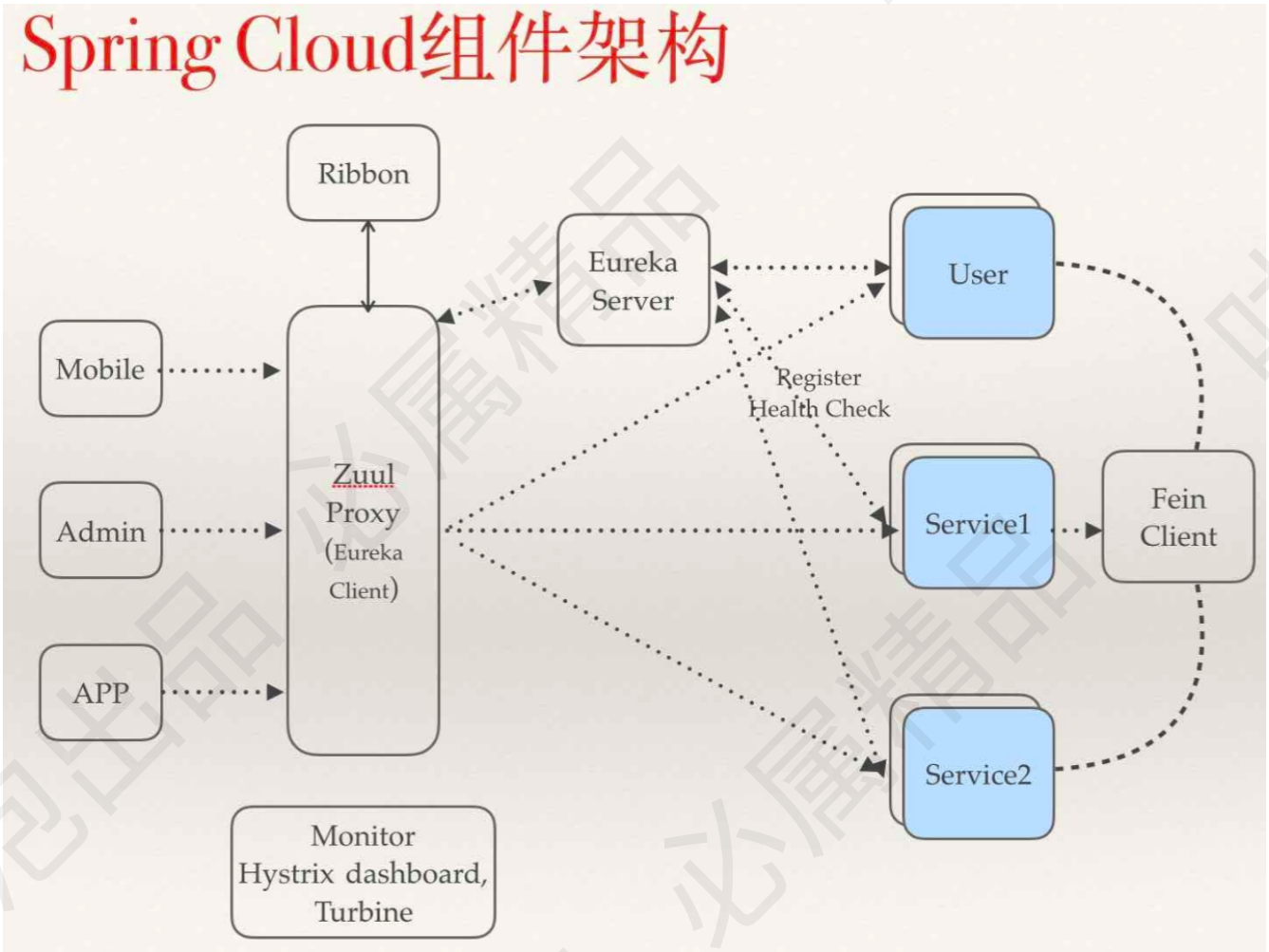


COORDINATE ANYTHING: DISTRIBUTED SYSTEMS SIMPLIFIED

Building distributed systems doesn't need to be complex and error-prone. Spring Cloud offers a simple and accessible programming model to the most common distributed system patterns, helping developers build resilient, reliable, and coordinated applications. Spring Cloud is built on top of Spring Boot, making it easy for developers to get started and become productive quickly.



SpringCloud组件架构：



2.2、SpringCloud与SpringBoot的区别

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系.

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

2.3、SpringCloud VS Dubbo

技术栈对比:

微服务组件	Dubbo	SpringCloud
服务注册中心	Zookeeper	Netflix Eureka
服务调用方式	RPC	Restful
服务监控	Dubbo-monitor	SpringBoot Admin
熔断器	不完善	Netflix Hystrix
服务网关	无	Netflix Zuul
分布式配置中心	无	SpringCloud Config
服务跟踪	无	SpringCloud Sleuth
消息总线	无	SpringCloud Bus
数据流	无	SpringCloud Stream
任务调度	无	SpringCloud Task

通信方式对比：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的Restful方式。两者方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

服务生态对比：SpringCloud的功能比Dubbo更加强大，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而SpringCloud就像品牌机，在SpringSource的整合下，做了大量的兼容性测试，保证了服拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

社区支持与更新力度对比：Dubbo停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

2.4、SpringCloud微服务架构组件

名称	描述
Spring Cloud Config	配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion
Spring Cloud Bus	事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。
Eureka	云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。
Hystrix	熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。
Zuul	Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。
Archaius	配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。
Consul	封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。
Spring Cloud for Cloud Foundry	通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。
Spring Cloud Sleuth	日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。
Spring Cloud Data Flow	大数据操作工具，作为Spring XD的替代产品，它是一个混合计算模型，结合了流数据与批量数据的处理方式
Spring Cloud Security	基于spring security的安全工具包，为你的应用程序添加安全控制
Spring Cloud Zookeeper	操作Zookeeper的工具包，用于使用zookeeper方式的服务发现和配置管理。
Spring Cloud Stream	数据流操作开发包，封装了与Redis,Rabbit、Kafka等发送接收消息。
Spring Cloud CLI	基于 Spring Boot CLI，可以让你以命令行方式快速建立云组件
Ribbon	提供云端负载均衡，有多种负载均衡策略可供选择，可配合服务发现和断路器使用
Turbine	Turbine是聚合服务器发送事件流数据的一个工具，用来监控集群下hystrix的metrics情况
Feign	Feign是一种声明式、模板化的HTTP客户端
Spring Cloud Task	提供云端计划任务管理、任务调度
Spring Cloud Connectors	便于云端应用程序在各种PaaS平台连接到后端，如：数据库和消息代理服务。
Spring Cloud Cluster	提供Leadership选举，如：Zookeeper, Redis, Hazelcast, Consul等常见状态模式的抽象和实现
Spring Cloud Starters	Spring Boot式的启动项目，为Spring Cloud提供开箱即用的依赖管理

Ribbon，客户端负载均衡，特性有区域亲和、重试机制。

Hystrix，客户端容错保护，特性有服务降级、服务熔断、请求缓存、请求合并、依赖隔离。

Feign，声明式服务调用，本质上就是Ribbon+Hystrix

Stream，消息驱动，有Sink、Source、Processor三种通道，特性有订阅发布、消费组、消息分区。

Bus，消息总线，配合Config仓库修改的一种Stream实现，

Sleuth，分布式服务追踪，需要搞清楚TraceID和SpanID以及抽样，如何与ELK整合。

Ø **独自启动不需要依赖其它组件，单枪匹马都能干。**

Eureka，服务注册中心，特性有失效剔除、服务保护。

Dashboard，**Hystrix**仪表盘，监控集群模式和单点模式，其中集群模式需要收集器Turbine配合。

Zuul，API服务网关，功能有路由分发和过滤。

Config，分布式配置中心，支持本地仓库、SVN、Git、Jar包内配置等模式

2.5、SpringCloud版本说明

英文	中文	终结版本	boot大版本	boot代表	说明
Angel	安吉尔	SR6	1.2.X	1.2.8	GA
Brixton	布里克斯顿	SR7	1.3.X	1.3.8	GA
Camden	卡梅登	SR7	1.4.X	1.4.2	GA
Dalston	达斯顿	SR5	1.5.X	*	GA
Edgware	艾奇韦尔	SR5	1.5.X	1.5.19	GA
Finchley	芬奇利	SR2	2.0.X	2.0.8	GA
Greenwich	格林威治	RC2	2.1.X	2.1.2	GA
hoxton	霍克斯顿	RC2	2.2.X	2.2.6	GA

其他说明 cloud版本都是伦敦地铁站的名词， 在SR版本发布之前，会先发布一个Release版本，如 Camder RELEASE

Edgware 有 1.5.9也是比较经典的

Finchley有Finchley.RELEASE 2018年6月19日发布 在Finchley SR2之前，一直都是 RC2 然后到了2.0.3.RELEASE，还是Finchley.RELEASE 直到2.0.8.RELEASE 才变成Finchley.SR2 Finchley的boot的2.0.9快照版是Finchley.BUILD-SNAPSHOT

2019年1月21日 CURRENT 版本为 Greenwich RC2 而最新的boot 2.2.0.BUILD-SNAPSHOT 对应：Greenwich.BUILD-SNAPSHOT 版本

版本说明 SNAPSHOT 快照版，可以稳定使用，且仍在继续改进版本。

PRE preview edition， 预览版,内部测试版. 主要是给开发人员和测试人员测试和找BUG用的，不建议使用；

RC Release Candidate，发行候选版本，基本不再加入新的功能，主要修复bug。是最终发布成正式版的前一个版本，将bug修改完就可以发布成正式版了。

SR Service Release，表示bug修复 修正版或更新版，修正了正式版推出后发现的Bug。

GA General Availability，正式发布的版本，官方开始推荐广泛使用，国外有的用GA来表示release版本。

Table 1. Release train Spring Boot compatibility

Release Train	Boot Version
2020.0.x aka Ilford	2.4.x, 2.5.x (Starting with 2020.0.3)
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

从这里可以看到大版本的对应关系

https://spring.io/projects/spring-cloud#overview

2.6、SpringCloud相关网站

官网：https://spring.io/projects/spring-cloud

API：

https://springcloud.cc/spring-cloud-netflix.html http://cloud.spring.io/spring-cloud-static/Dalston.SR1/ 社区：

SpringCloud中国社区：http://springcloud.cn/ SpringCloud中文网：https://springcloud.cc/

4、服务网格架构

4.1、业务团队的痛点

1. 对于业务开发团队而言，最强的是技术吗？一定不是，业务团队最强的一定是对于业务的理解和熟悉程度。
2. 而业务应用的核心价值，就是为了实现业务场景，而不是写微服务，微服务只是一种实现业务的手段。
3. 业务团队除了关心业务之外，他们所面临的最大的挑战在于，如何保证系统的稳定性何可扩展性、如何设计一个安全的open api。如果对服务进行拆分、如何保证跨库的数据一致性。以及对于旧系统的改造。
4. 于公司层面而言，业务团队的压力还来自于时间人力的投入，我们用于被各种deadline赶着走。所以作为一个业务程序员，如果在这个deadline之前还需要花更多的时间投入在spring cloud这些工具的学习上，那无疑是雪上加霜。公司对于业务团队的考核，永远只看结果！

4.2、微服务存在的技术问题

首先在了解服务网格之前我们先来谈谈微服务架构的技术问题

Spring Cloud则是通过集成众多的组件的形式实现了相对完整的微服务技术栈，但是Spring Cloud的实现方式代码侵入性较强，而且只支持Java语言，无法支持其他语言开发的系统。Spring Cloud全家桶包括的内容较多，学习成本也相对较高，对老系统而言，框架升级或者替换的成本较高，导致一些开发团队不愿意担负技术和时间上的风险与成本，使得微服务方案在落地时遇到了诸多的困难。springcloud的技术问题主要表现在3个方面：

1、技术门槛高：开发团队在实施微服务的过程中，除了基础的服务发现、配置中心和鉴权管理之外，团队在服务治理层面面临了诸多的挑战，包括负载均衡、熔断降级、灰度发布、故障切换、分布式跟踪等，这对开发团队提出了非常高的技术要求。 2、多语言支持不足：对于互联网公司，尤其是快速发展的互联网创业公司，多语言的技术栈、跨语言的服务调用也是常态，但目前开源社区上并没有一套统一的、跨语言的微服务技术栈，而跨语言调用恰恰是微服务概念诞生之初的要实现的一个重要特性之一。 3、代码侵入性强：Spring Cloud、Dubbo等主流的微服务框架都对业务代码有一定的侵入性，技术升级替换成本高，导致开发团队配合意愿低，微服务落地困难。

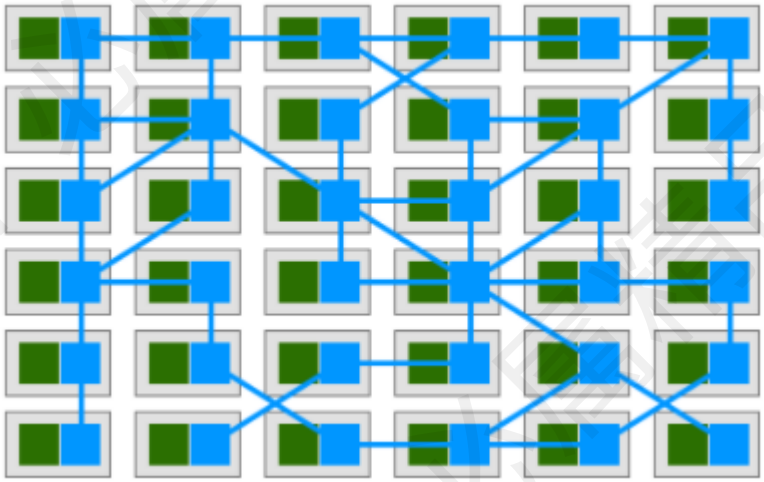
4.3、服务网格ServiceMesh

在介绍Service Mesh概念之前，我们先来了解一下Sidecar。Sidecar是以第一次世界大战时活跃在战场上的军用边斗车命名（也是我们在抗日神剧中最常见的道具之一）。Sidecar是Service Mesh中的重要组成部分，Sidecar在软件系统架构中特指边斗车模式，这个模式的精髓在于实现了数据面（业务逻辑）和控制面的解耦。 在Service Mesh架构中，给每一个微服务实例部署一个Sidecar Proxy。该Sidecar Proxy负责接管对应服务的入流量和出流量，并将微服务架构中的服务订阅、服务发现、熔断、限流、降级、分布式跟踪等功能从服务中抽离到该Proxy中。

Sidecar以一个独立的进程启动，可以每台宿主机共用同一个Sidecar进程，也可以每个应用独占一个Sidecar进程。所有的服务治理功能，都由Sidecar接管，应用的对外访问仅需要访问Sidecar即可。当该Sidecar在微服务中大量部署时，这些Sidecar节点自然就形成了一个服务网格。

在新一代的ServiceMesh架构中(下图上方)，服务的消费方和提供方主机(或者容器)两边都会部署代理SideCar。ServiceMesh比较正式的术语也叫数据面板(DataPlane)，与数据面板对应的还有一个独立部署的控制面板(ControlPlane)，用来集中配置和管理数据面板，也可以对接各种服务发现机制(如K8S服务发现)

下图，每个主机上同时居住了业务逻辑代码(绿色表示)和代理(蓝色表示)，服务之间通过代理发现和调用目标服务，形成服务之间的一种网络状依赖关系，控制面板则可以配置这种依赖调用关系，也可以调拨路由流量。如果我们把主机和业务逻辑剥离，就出现一种网格状架构(上图右下角)，服务网格由此得名。



Service Mesh为业务开发团队降低了门槛，提供了稳定的基础设施，最重要的是，让业务开发团队从微服务实现的具体技术细节中解放出来回归到业务。