## 原始分布式时代

半个世纪前，对于分布式架构的第一次服务化探索的得与失。

分布式架构的目标是使用多个独立的分布式服务，来共同构建一个更大型的系统。

Unix的分布式设计哲学:

Simplicity of both the interface and the implementation are more important than any other attributes of the system — including correctness, consistency, and completeness.

保持接口与实现的**简单性**，比系统的任何其他属性，包括准确性、一致性和完整性，都来得更加重要。

—— Richard P. Gabriel，The Rise of 'Worse is Better，1991

20世纪70年代末到80年代初，计算机科学经历了以大型机为主，到向以微型机为主的转变，逐渐成为了商业、家庭和个人的生产与娱乐设备。

探索中出现的对后来软件架构发展影响深远的例子：

惠普：80年代初提出的[网络运算架构](https://en.wikipedia.org/wiki/Network_Computing_System)（NCA）,远程服务调用的雏形

卡内基梅隆大学提出的AFS文件系统，分布式文件系统的最早实现

MIT提出的Kerberos协议，服务认证和访问控制的基础

制定Unix系统技术标准的 [开发软件基金会](https://zh.wikipedia.org/wiki/%E9%96%8B%E6%94%BE%E8%BB%9F%E9%AB%94%E5%9F%BA%E9%87%91%E6%9C%83)(OSF)，邀请了各个主要的研究厂商一起参与，共同制订了“[分布式运算环境](https://zh.wikipedia.org/wiki/%E5%88%86%E6%95%A3%E5%BC%8F%E9%81%8B%E7%AE%97%E7%92%B0%E5%A2%83)”（Distributed Computing Environment，**DCE**）的分布式技术体系。

DCE 包括了一整套完整的分布式服务组件的规范与实现。

OSF的背景之下，通常都会有一个预设的重要原则，也就是**在实现分布式环境中的服务调用、资源访问、数据存储等操作的时候，要尽可能地透明化、简单化，让远程的方法调用如同本地一般**，但是在当时是过于理想化的，存在许多无法完美解决的技术问题。

关于调用远程方法：

远程的服务在哪里（服务发现）、有多少个（负载均衡）、网络出现分区、超时或者服务出错了怎么办（熔断、隔离、降级）、方法的参数与返回结果如何表示（序列化协议）、如何传输（传输协议）、服务权限如何管理（认证、授权）、如何保证通信安全（网络安全层）、如何令调用不同机器的服务能返回相同的结果（分布式数据一致性）

DCE尽力做到了分布式系统的“透明化”和“简单化”，但是编码的复杂度、程序的运行效率、软件的性能都出现了巨大的挑战。

从结果来看，历史局限决定了它不可能一蹴而就地解决分布式的难题，但仅从过程来看，这个阶段的探索可以称得上是硕果累累、成绩斐然。

原始分布式时代的教训

Just because something can be distributed doesn’t mean it should be distributed. Trying to make a distributed call act like a local call always ends in tears.

某个功能能够进行分布式，并不意味着它就应该进行分布式，强行追求透明的分布式操作，只会自寻苦果。

—— Kyle Brown，IBM Fellow，Beyond buzzwords: A brief history of microservices patterns，2016

自此，通往更大规模的计算机软件系统的道路分为了两条：

1. 尽快提升单机的处理能力，以避免分布式的种种问题；
2. 找到更完美的解决方案，来应对如何构筑分布式系统的问题。

## 单体架构时代

在第一次分布式架构的尝试失败之后，20世纪80年代，摩尔定律开始稳定的发生作用，硬件在算力上对于软件规模的束缚开始松动，人们只需要使用一台或者几台计算机就可以作为服务器来支撑起工作，信息系统开始进入单体架构时代。

思维误区：单体架构是落后的系统架构风格，最终会被微服务所取代。

在微服务的书籍当中，单体系统更多指的是**大型单体系统**,或者称为**巨石系统(Monolithic System)**，这是一种事后追认。在没有微服务的时代，并没有人会认为单体是一种架构，一方面体现了单体架构本身的简单性；另一方面也体现出，在相当长的时间里，我们都已经习惯了，软件架构就应该是单体这种样子的。

#### 单体的拆分

Monolithic更有一种**自给自足**(Self-Contained)的含义，而不只是All in One Piece(铁板一块)。

纵向角度：

* 分层架构(Layered Architecture)：单体架构更加容易开发、部署和测试。

横向角度：

* 单体架构也可以支持按照技术、功能、职责等角度，把软件拆分为各种模块，以便重用和团队管理。

单体系统的**真正缺陷**实际上并不在于要如何拆分，而**在于拆分之后，它会存在隔离与自治能力上的欠缺**。

* 在单体架构中，所有的代码都运行在同一个进程空间之内，所有模块、方法的调用也都不需要考虑网络分区、对象复制这些麻烦事儿，也不担心因为数据交换而造成性能的损失。
* 如果在单体架构中，**有任何一部分的代码出现了缺陷，过度消耗进程空间内的公共资源，那所造成的影响就是全局性的、难以隔离的。**

缺陷：

1. 一旦架构中出现了内存泄漏、线程爆炸、阻塞、死循环等问题，就都将会影响到整个程序的运行，而不仅仅是某一个功能、模块本身的正常运作。
2. 动态可维护性角度：所有代码都共享着同一个进程空间，如果代码无法隔离，那也就意味着，我们无法做到单独停止、更新、升级某一部分代码。
3. 技术异构：（允许系统的每个模块，自由选择不一样的程序语言、不一样的编程框架等技术栈去实现）。单体实现起来很麻烦。

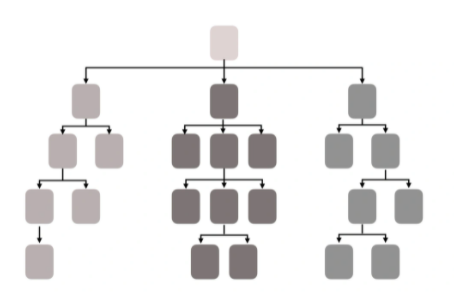
## SOA时代

**SOA 架构是第一次被广泛使用过的、通过分布式服务来构建信息系统的工程实践。**

它有完善的理论和工具，可以说，它解决了分布式系统中，几乎所有主要的技术问题。但它最终还是没能成为一种**普适的软件架构。**

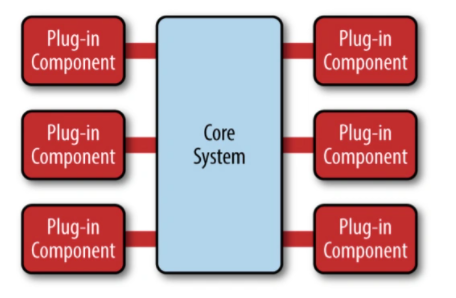
在SOA出现之前，先看一下三种代表性的服务拆分架构模式，它们是SOA演化过程的中间产物。

#### 烟囱式架构(Information Silo Architecture)



相互之间没有交互关联

#### 微内核架构(Microkernel Architecture)或 插件式架构(Plug-in Architecture)



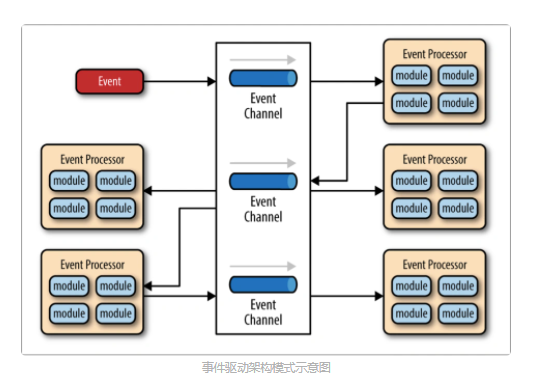
通过插件的形式提供服务的功能和进行增量开发，天然具有良好的扩展性、隔离性、灵活性。

它的局限在于：它会假设系统中各个插件模块之间是互不认识的（因为不可预知系统会安装哪些模块），这些插件会访问内核中一些公共的资源，但不会发生直接交互。 这是很多时候不符合现实的情况，很多模块是需要相互调用的。

#### 事件驱动架构(Event-Driven Architecture)

该方案解决了子系统之间的相互通讯问题。

运作方案是：在子系统之间建立一套事件队列管道（Event Queues），来自系统外部的消息将以事件的形式发送到管道中，各个子系统可以从管道里获取自己感兴趣、可以处理的事件消息，也可以为事件新增或者是修改其中的附加信息，甚至还可以自己发布一些新的事件到管道队列中去。



此时，在寻求更加完美的分布式问题的解决方案上，即远程服务调用，迎来了SOAP协议的诞生。

### SOA时代的探索

2007年，在OASIS的倡导之下，成立了SOA的官方管理机构------Open SCA。

##### 更具体

SOA针对之前分布式服务的各种问题，有了更加具体的探索。它比之前的几种架构都更具有可操作性，细节更加充实，因此可以称之为一套软件架构的基础平台。

基础平台的概念：

* 官方组织
* 清晰的设计原则
* SOAP作为远程调用协议
* ESB消息管道实现子系统之间的通讯交互，业务流程编排BPM
* 服务数据对象SDO来访问和表示数据，SCA服务组件架构定义服务的形式和运行容器

##### 更系统

因为 SOA 最根本的目标，就是希望能够总结出一套自上而下的软件研发方法论，让企业只需要跟着它的思路，就能够一揽子解决掉软件开发过程中的全套问题。

##### 问题出现

SOA 架构过于严谨精密的流程与理论，导致了软件开发的全过程，都需要有懂得复杂概念的专业人员才能够驾驭。

从 SOA 诞生的那一天起，就已经注定了它只能是少数系统的阳春白雪式的精致奢侈品：它可以实现多个异构大型系统之间的复杂集成交互，**却很难作为一种具有广泛普适性的软件架构风格来推广**。

## 微服务时代

经过了三十年的技术发展，信息系统经历了巨石、烟囱、微内核、事件驱动、SOA 等架构模式，应用受架构复杂度的牵绊却是越来越大，距离“透明”二字已经越来越远了。

Peter Rodgers 博士在 2005 年的云计算博览会（Web Services Edge 2005）上，就已经提出和使用了“Micro-Web\_Service”，即微服务，指的是一种**专注于单一职责的**、**与语言无关的**、**细粒度的 Web 服务**（Granular Web Services）

最开始的微服务，可以说是在 SOA 发展的同时被催生出来的产物，是**作为 SOA 的一种轻量化的补救方案而被提出来的。**

此后的近10年时间，都没有收到很多人的追捧。

2012年，在波兰克拉科夫举行的“33rd Degree Conference”大会上，Thoughtworks 首席咨询师 James Lewis 做了题为《Microservices - Java, the Unix Way》的主题演讲。他提到了单一服务职责、康威定律、自动扩展、领域驱动设计等原则，号召大家，应该重拾 Unix 的设计哲学（As Well Behaved Unix Services）。

到2014年，Martin Fowler 和 James Lewis 合写的文章[“Microservices: a definition of this new architectural term”](https://martinfowler.com/articles/microservices.html)里面，第一次让大多数程序员了解到微服务。它定义了现代微服务的概念：

微服务是一种通过多个小型服务的组合，来构建单个应用的架构风格，这些服务会围绕业务能力而非特定的技术标准来构建。各个服务可以采用不同的编程语言、不同的数据存储技术、运行在不同的进程之中。服务会采取轻量级的通讯机制和自动化的部署机制，来实现通讯与运维。

提出9个核心的业务与技术特征：

* 围绕业务能力构建（Organized around Business Capabilities）
* 康威定律：有怎样的结构、规模和能力的团队，就会产生出对应结构、规模、能力的产品。
* 分散治理（Decentralized Governance）
* 通过服务来实现独立自治的组件（Componentization via Services）
* 产品化思维（Products not Projects）
* 数据去中心化（Decentralized Data Management）
* 轻量级通讯机制（Smart Endpoints and Dumb Pipes）
* 容错性设计（Design for Failure）
* 演进式设计（Evolutionary Design）
* 基础设施自动化（Infrastructure Automation）

申明了微服务不是什么：微服务不是 SOA 的衍生品，应该明确地与 SOA 划清界线，不再贴上任何 SOA 的标签。

微服务追求的是更加自由的架构风格，它摒弃了 SOA 中几乎所有可以抛弃的约束和规定，提倡以“实践标准”代替“规范标准”。

在从复杂的技术标准当中获得自由的同时，微服务也失去了SOA那样统一的规范和约束，之前解决的分布式服务的问题再一次出现，如服务的注册发现、跟踪治理、负载均衡、故障隔离、认证授权、伸缩扩展、传输通讯、事务处理等问题，而在微服务当中，不再会有统一的解决方案。

微服务架构下，我们需要解决什么问题，就引入什么工具；团队熟悉什么技术，就使用什么框架。架构能力的要求变得格外重要，各种决策的权衡需要广泛的知识面才能应对。

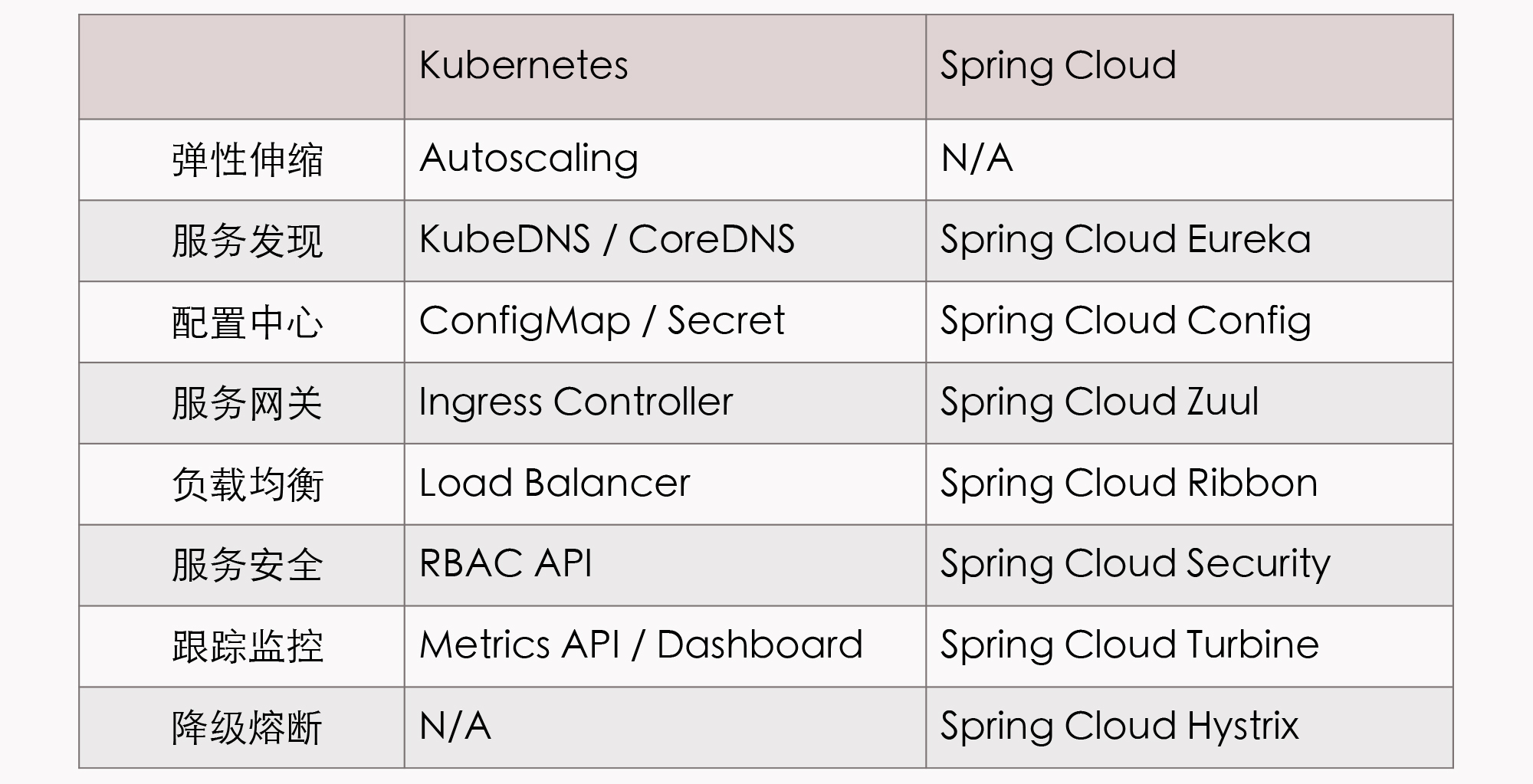
## 后微服务时代---跨越软件和硬件之间的界限

在微服务时代，我们之所以不得不在应用服务层面，而不是基础设施层面去解决这些分布式问题，完全是因为由硬件构成的基础设施，跟不上由软件构成的应用服务的灵活性。

微服务时代所取得的成就，本身就离不开以 Docker 为代表的早期容器化技术的巨大贡献。

2017年，Kubernetes成为了容器战争的胜利者，下一个时代的虚拟化基础设施。

*比较应用层的Spring Cloud与基础层的Kubernetes*



一旦硬件能够跟得上软件的灵活性，**那么这些与业务无关的技术问题，便很可能从软件的层面剥离出来，在硬件的基础设施之内就被悄悄解决掉，让软件可以只专注于业务，真正“围绕业务能力构建”团队与产品。**

进入了云原生时代，或者称为后微服务时代。

但**Kubernetes并没有完美的解决分布式的全部问题**。

从功能灵活强大来说，Kubernetes比不上应用层的Spring Cloud。一些处于应用系统与基础设施的交界处，很难完全**在基础设施层精细化的解决**好。

基础设施是针对整个容器来做整体管理的，它的粒度就相对粗犷。例如Kubernetes只能操作到Pod层次，无法直接对容器和服务来进行操作，应用层则只要合乎逻辑即可。

为了解决这些问题，引入了**服务网格**（如Istio和Envoy）的**边车代理模式**，实现既**不需要在应用层面附带额外的代码**，也提供了几乎不亚于应用代码的**精细管理能力**的目的。

但还不太成熟。

## 无服务时代

发展历史：

在工业界，2012 年，iron.io 公司率先提出了“无服务”（Serverless，应该翻译为“无服务器”才合适，但现在用“无服务”已形成习惯了）的概念；

2014 年开始，AWS 发布了命名为 Lambda 的商业化无服务应用，并在后续的几年里逐步得到了开发者的认可，发展成目前世界上最大的无服务的运行平台；

到了 2019 年，中国的阿里云、腾讯云等厂商，也发布了无服务的产品。

学术界：2019年，UC Berkeley发表了论文[“Cloud Programming Simplified: A Berkeley View on Serverless Computing”](https://arxiv.org/abs/1902.03383)，预言“无服务将会成为日后云计算的主流方式”。

特点：**简单。**

* 后端设施：是指数据库、消息队列、日志、存储等这一类用于支撑业务逻辑运行，但本身无业务含义的技术组件。这些后端设施都运行在云中，也就是无服务中的“**后端即服务**”（Backend as a Service，BaaS）
* 函数：就是业务逻辑代码。这里函数和程序编码角度的函数，的区别就在于，无服务中的函数运行在云端，不必考虑算力问题和容量规划。这就是无服务中的**“函数即服务”**（Function as a Service，FaaS）

**无服务的愿景是让开发者只需要纯粹地关注业务。**

但是它注定不会是一种普适的架构模式。

无服务架构天生的一些特点，比如**冷启动**、 **无状态**、**运行时间有限制**等等。