基于Spring Boot+Cloud构建微云架构

原文地址：https://www.cnblogs.com/williamjie/p/11167852.html

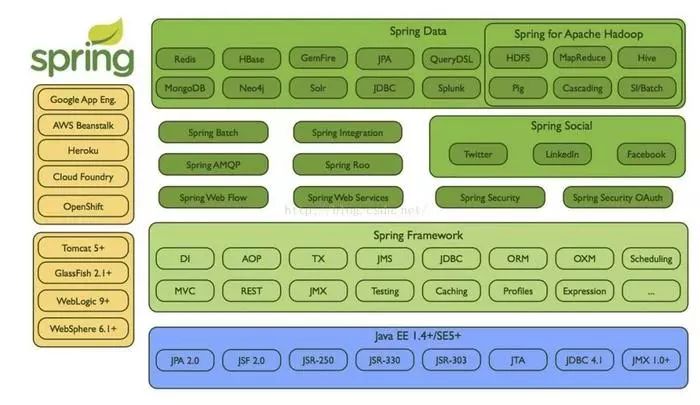
前言

首先，最想说的是，当你要学习一套最新的技术时，官网的英文文档是学习的最佳渠道。因为网上流传的多数资料是官网翻译而来，很多描述的重点也都偏向于作者自身碰到的问题，这样就很容易让你理解和操作出现偏差，最开始我就进入了这样误区。官网的技术导读真的描述的很详细，虽然对于我们看英文很费劲，但如果英文不是很差，请选择沉下心去读，你一定能收获好多。

我的学习是先从Spring boot开始的，然后接触到微服务架构，当然，这一切最大的启迪还是感谢我的一个老师，是他给我指明了新的道路，让我眼前一亮，再次感谢。

Spring 顶级框架

谈及微服务，作为当前主流的企业框架Spring，它提供了一整套相关的顶级项目，能让开发者快速的上手实现自己的应用，今天就介绍下Spring旗下各个顶级项目：



Spring IO platform:用于系统部署，是可集成的，构建现代化应用的版本平台，具体来说当你使用maven dependency引入spring jar包时它就在工作了。

Spring Boot: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，可以和spring cloud联合部署。

Spring Framework:即通常所说的spring 框架，是一个开源的Java/Java EE全功能栈应用程序框架，其它spring项目如spring boot也依赖于此框架。

Spring Cloud：微服务工具包，为开发者提供了在分布式系统的配置管理、服务发现、断路器、智能路由、微代理、控制总线等开发工具包。

Spring XD：是一种运行时环境（服务器软件，非开发框架），组合spring技术，如spring batch、spring boot、spring data，采集大数据并处理。

Spring Data：是一个数据访问及操作的工具包，封装了很多种数据及数据库的访问相关技术，包括：jdbc、Redis、MongoDB、Neo4j等。

Spring Batch：批处理框架，或说是批量任务执行管理器，功能包括任务调度、日志记录/跟踪等。

Spring Security：是一个能够为基于Spring的企业应用系统提供声明式的安全访问控制解决方案的安全框架。

Spring Integration：面向企业应用集成（EAI/ESB）的编程框架，支持的通信方式包括HTTP、FTP、TCP/UDP、JMS、RabbitMQ、Email等。

Spring Social：一组工具包，一组连接社交服务API，如Twitter、Facebook、LinkedIn、GitHub等，有几十个。

Spring AMQP：消息队列操作的工具包，主要是封装了RabbitMQ的操作。

Spring HATEOAS：是一个用于支持实现超文本驱动的 REST Web 服务的开发库。

Spring Mobile：是Spring MVC的扩展，用来简化手机上的Web应用开发。

Spring for Android：是Spring框架的一个扩展，其主要目的在乎简化Android本地应用的开发，提供RestTemplate来访问Rest服务。

Spring Web Flow：目标是成为管理Web应用页面流程的最佳方案，将页面跳转流程单独管理，并可配置。

Spring LDAP：是一个用于操作LDAP的Java工具包，基于Spring的JdbcTemplate模式，简化LDAP访问。

Spring Session：session管理的开发工具包，让你可以把session保存到redis等，进行集群化session管理。

Spring Web Services：是基于Spring的Web服务框架，提供SOAP服务开发，允许通过多种方式创建Web服务。

Spring Shell：提供交互式的Shell可让你使用简单的基于Spring的编程模型来开发命令，比如Spring Roo命令。

Spring Roo：是一种Spring开发的辅助工具，使用命令行操作来生成自动化项目，操作非常类似于Rails。

Spring Scala：为Scala语言编程提供的spring框架的封装（新的编程语言，Java平台的Scala于2003年底/2004年初发布）。

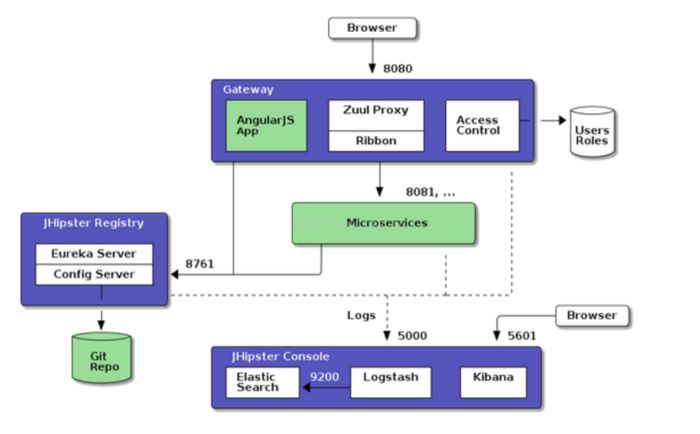
Spring BlazeDS Integration：一个开发RIA工具包，可以集成Adobe Flex、BlazeDS、Spring以及Java技术创建RIA。

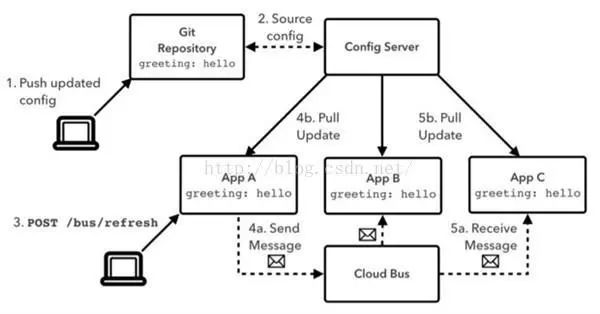
Spring Loaded：用于实现java程序和web应用的热部署的开源工具。

Spring REST Shell：可以调用Rest服务的命令行工具，敲命令行操作Rest服务。

Spring cloud子项目

目前来说spring主要集中于spring boot（用于开发微服务）和spring cloud相关框架的开发，我们从几张图着手理解，然后再具体介绍：





spring cloud子项目包括：

**Spring Cloud Config**：配置管理开发工具包，可以让你把配置放到远程服务器，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。

**Spring Cloud Netflix**：针对多种Netflix组件提供的开发工具包，其中包括Eureka、Hystrix、Zuul、Archaius等。

**Netflix Eureka**：云端负载均衡，一个基于 REST 的服务，用于定位服务，以实现云端的负载均衡和中间层服务器的故障转移。

**Netflix Hystrix**：容错管理工具，旨在通过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。

**Netflix Zuul**：边缘服务工具，是提供动态路由，监控，弹性，安全等的边缘服务。

**Netflix 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。

**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,Zipkin和HTrace操作。

**Spring Cloud Data Flow**：大数据操作工具，通过命令行方式操作数据流。

**Spring Cloud Security**：安全工具包，为你的应用程序添加安全控制，主要是指OAuth2。

**Spring Cloud 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。

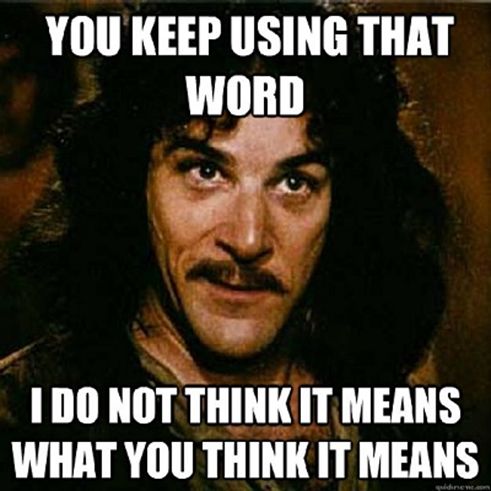
**Spring Cloud Zookeeper**：操作Zookeeper的工具包，用于使用zookeeper方式的服务注册和发现。

**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。

**WHAT - 什么是微服务**

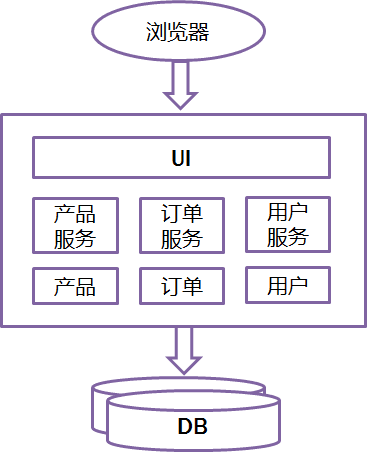
**微服务简介**



微服务的流行，Martin功不可没，这老头也是个奇人，特别擅长抽象归纳和制造概念，我觉的这就是最牛逼的markting啊，感觉这也是目前国人欠缺的能力。

Martin Fowler是国际著名的OO专家，敏捷开发方法的创始人之一，现为ThoughtWorks公司的首席科学家.福勒（Martin Fowler），在面向对象分析设计、UML、模式、软件开发方法学、XP、重构等方面，都是世界顶级的专家，现为Thought Works公司的首席科学家。Thought Works是一家从事企业应用开发和集成的公司。早在20世纪80年代，Fowler就是使用对象技术构建多层企业应用的倡导者，他著有几本经典书籍：《企业应用架构模式》、《UML精粹》和《重构》等。—— 百度百科

先来看看传统的web开发方式，通过对比比较容易理解什么是Microservice Architecture。和Microservice相对应的，这种方式一般被称为Monolithic（比较难传神的翻译）。所有的功能打包在一个 WAR包里，基本没有外部依赖（除了容器），部署在一个JEE容器（Tomcat，JBoss，WebLogic）里，包含了 DO/DAO，Service，UI等所有逻辑。



Monolithic比较适合小项目，优点是：

1、开发简单直接，集中式管理

2、基本不会重复开发

3、功能都在本地，没有分布式的管理开销和调用开销

它的缺点也非常明显，特别对于互联网公司来说（不一一列举了）：

1、开发效率低：所有的开发在一个项目改代码，递交代码相互等待，代码冲突不断

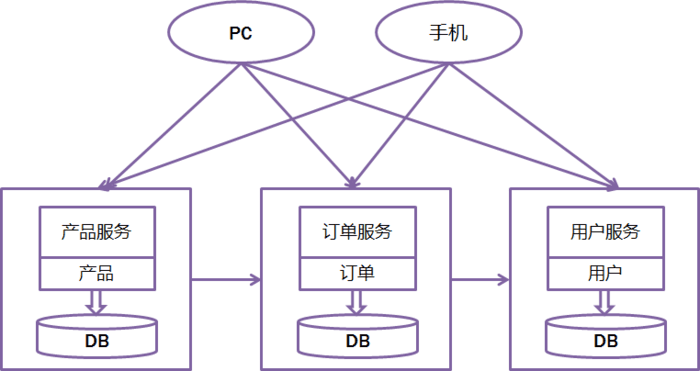
2、代码维护难：代码功能耦合在一起，新人不知道何从下手

3、部署不灵活：构建时间长，任何小修改必须重新构建整个项目，这个过程往往很长

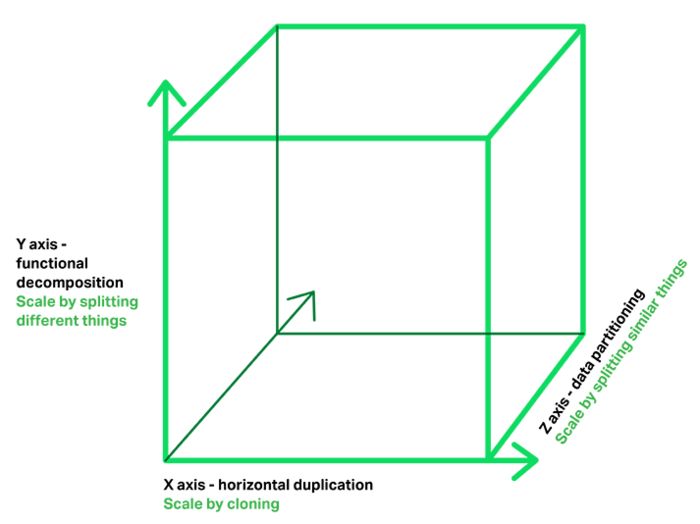
4、稳定性不高：一个微不足道的小问题，可以导致整个应用挂掉

5、扩展性不够：无法满足高并发情况下的业务需求

所以，现在主流的设计一般会采用Microservice Architecture，就是基于微服务的架构。简单来说，微服务的目的是有效的拆分应用，实现敏捷开发和部署。



用《The art of scalability》一书里提到的scale cube比较容易理解如何拆分。你看，我们叫分库分表，别人总结成了scale cube，这就是抽象的能力啊，把复杂的东西用最简单的概念解释和总结。X轴代表运行多个负载均衡器之后运行的实例，Y轴代表将应用进一步分解为微服务 （分库），数据量大时，还可以用Z轴将服务按数据分区（分表）



**微服务的具体特征**

先看看最官方的定义吧

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are \*\*built around business capabilities\*\* and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services , which may be written in different programming languages and use different data storage technologies.

-- James Lewis and Martin Fowler

把Martin老头的定义大概的翻译一下就是下面几条，这个定义还是太抽象是不是，那就对了，就是要务虚，都说明白了谁还找他付费咨询啊，这么贵。

1. 一些列的独立的服务共同组成系统

2. 单独部署，跑在自己的进程里

3. 每个服务为独立的业务开发

4. 分布式的管理

Martin自己也说了，每个人对微服务都可以有自己的理解，不过大概的标准还是有一些的。

1、分布式服务组成的系统

2、按照业务而不是技术来划分组织

3、做有生命的产品而不是项目

4、Smart endpoints and dumb pipes（我的理解是强服务个体和弱通信）

5、自动化运维（DevOps）

6、容错

7、快速演化

**SOA vs Microservice**

除了Smart endpoints and dumb pipes都很容易理解对吗？相信很多人都会问一个问题，这是不是就是SOA换了个概念，挂羊头卖狗肉啊，有说法把Microservice叫成 Lightway SOA。也有很多传统砖家跳出来说Microservice就是SOA。其实Martin也没否认SOA和Microservice的关系。

我个人理解，Microservice是SOA的传承，但一个最本质的区别就在于Smart endpoints and dumb pipes，或者说是真正的分布式的、去中心化的。Smart endpoints and dumb pipes本质就是去ESB，把所有的“思考”逻辑包括路由、消息解析等放在服务内部（Smart endpoints），去掉一个大一统的ESB，服务间轻（dumb pipes）通信，是比SOA更彻底的拆分。

**HOW - 怎么具体实践微服务**

听上去好像都不错，具体怎么落地啊？这需要回答下面几个问题：

1、客户端如何访问这些服务？

2、服务之间如何通信？

3、这么多服务，怎么找?

4、服务挂了怎么办？

5、客户端如何访问这些服务？

原来的Monolithic方式开发，所有的服务都是本地的，UI可以直接调用，现在按功能拆分成独立的服务，跑在独立的一般都在独立的虚拟机上的 Java进程了。客户端UI如何访问他的？后台有N个服务，前台就需要记住管理N个服务，一个服务下线/更新/升级，前台就要重新部署，这明显不服务我们 拆分的理念，特别当前台是移动应用的时候，通常业务变化的节奏更快。另外，N个小服务的调用也是一个不小的网络开销。还有一般微服务在系统内部，通常是无 状态的，用户登录信息和权限管理最好有一个统一的地方维护管理（OAuth）。

所以，一般在后台N个服务和UI之间一般会一个代理或者叫API Gateway，他的作用包括

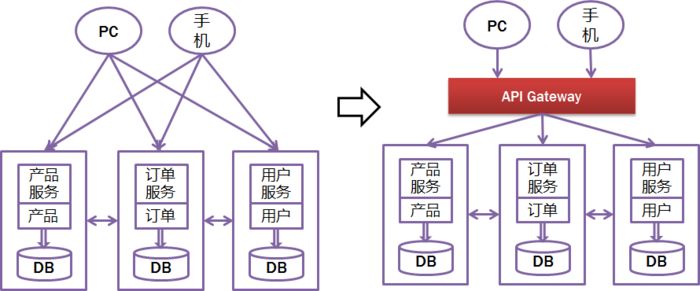
1、提供统一服务入口，让微服务对前台透明

2、聚合后台的服务，节省流量，提升性能

3、提供安全，过滤，流控等API管理功能

我的理解其实这个API Gateway可以有很多广义的实现办法，可以是一个软硬一体的盒子，也可以是一个简单的MVC框架，甚至是一个Node.js的服务端。他们最重要的作 用是为前台（通常是移动应用）提供后台服务的聚合，提供一个统一的服务出口，解除他们之间的耦合，不过API Gateway也有可能成为单点故障点或者性能的瓶颈。

一般用过Taobao Open Platform的就能很容易的体会，TAO就是这个API Gateway。



**服务之间如何通信？**

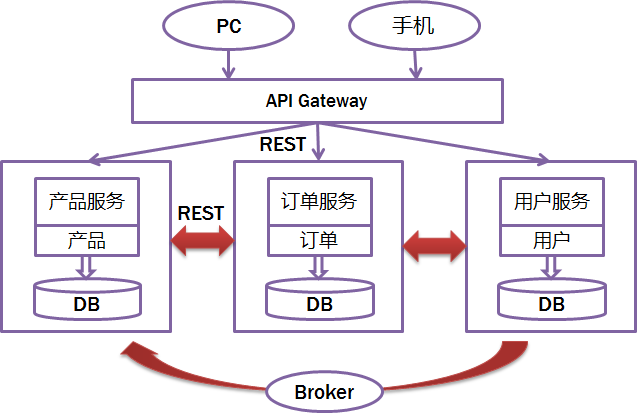
因为所有的微服务都是独立的Java进程跑在独立的虚拟机上，所以服务间的通行就是IPC（inter process communication），已经有很多成熟的方案。现在基本最通用的有两种方式。这几种方式，展开来讲都可以写本书，而且大家一般都比较熟悉细节了， 就不展开讲了。

1、同步调用

2、REST（JAX-RS，Spring Boot）

3、RPC（Thrift, Dubbo）

4、异步消息调用(Kafka, Notify, MetaQ)



一般同步调用比较简单，一致性强，但是容易出调用问题，性能体验上也会差些，特别是调用层次多的时候。RESTful和RPC的比较也是一个很有意 思的话题。

一般REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要 求，只要封装了HTTP的SDK就能调用，所以相对使用的广一些。RPC也有自己的优点，传输协议更高效，安全更可控，特别在一个公司内部，如果有统一个 的开发规范和统一的服务框架时，他的开发效率优势更明显些。就看各自的技术积累实际条件，自己的选择了。

而异步消息的方式在分布式系统中有特别广泛的应用，他既能减低调用服务之间的耦合，又能成为调用之间的缓冲，确保消息积压不会冲垮被调用方，同时能 保证调用方的服务体验，继续干自己该干的活，不至于被后台性能拖慢。

不过需要付出的代价是一致性的减弱，需要接受数据最终一致性；还有就是后台服务一般要 实现幂等性，因为消息发送出于性能的考虑一般会有重复（保证消息的被收到且仅收到一次对性能是很大的考验）；最后就是必须引入一个独立的broker，如 果公司内部没有技术积累，对broker分布式管理也是一个很大的挑战。

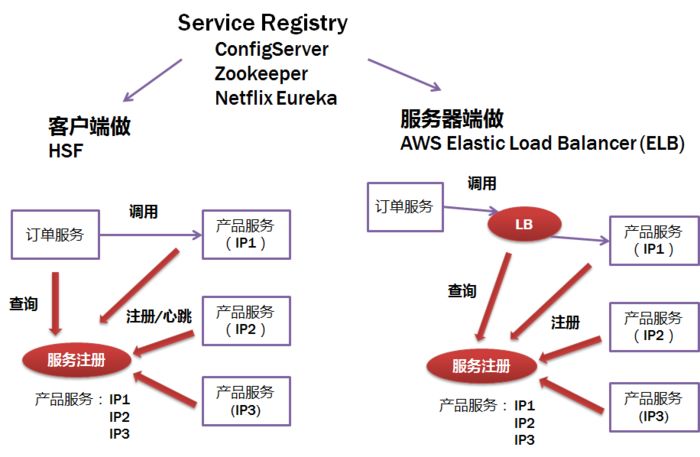
**这么多服务，怎么找?**

在微服务架构中，一般每一个服务都是有多个拷贝，来做负载均衡。一个服务随时可能下线，也可能应对临时访问压力增加新的服务节点。服务之间如何相互 感知？服务如何管理？这就是服务发现的问题了。

一般有两类做法，也各有优缺点。基本都是通过zookeeper等类似技术做服务注册信息的分布式管理。当 服务上线时，服务提供者将自己的服务信息注册到ZK（或类似框架），并通过心跳维持长链接，实时更新链接信息。服务调用者通过ZK寻址，根据可定制算法， 找到一个服务，还可以将服务信息缓存在本地以提高性能。当服务下线时，ZK会发通知给服务客户端。

1、客户端做：优点是架构简单，扩展灵活，只对服务注册器依赖。缺点是客户端要维护所有调用服务的地址，有技术难度，一般大公司都有成熟的内部框架支持，比如Dubbo。

2、服务端做：优点是简单，所有服务对于前台调用方透明，一般在小公司在云服务上部署的应用采用的比较多。



这么多服务，服务挂了怎么办？

前面提到，Monolithic方式开发一个很大的风险是，把所有鸡蛋放在一个篮子里，一荣俱荣，一损俱损。而分布式最大的特性就是网络是不可靠 的。通过微服务拆分能降低这个风险，不过如果没有特别的保障，结局肯定是噩梦。

我们刚遇到一个线上故障就是一个很不起眼的SQL计数功能，在访问量上升 时，导致数据库load彪高，影响了所在应用的性能，从而影响所有调用这个应用服务的前台应用。所以当我们的系统是由一系列的服务调用链组成的时候，我们 必须确保任一环节出问题都不至于影响整体链路。相应的手段有很多：

1、重试机制

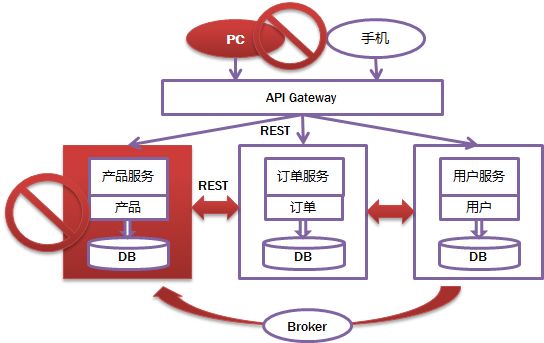
2、限流

3、熔断机制

4、负载均衡

5、降级（本地缓存）

这些方法基本上都很明确通用，就不详细说明了。



**WHY - 微服务的应用**

这里有一个图非常好的总结微服务架构需要考虑的问题，包括

1、API Gateway

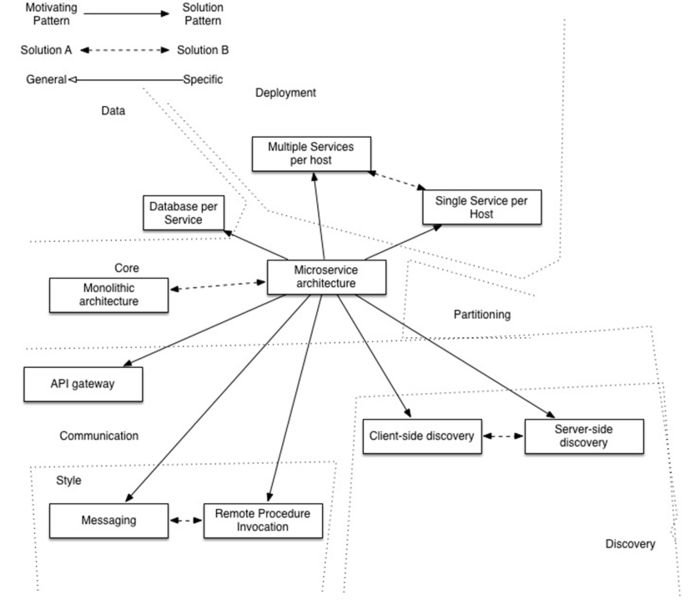
2、服务间调用

3、服务发现

4、服务容错

5、服务部署

6、数据调用



微服务的优点和缺点（或者说挑战）一样明显。

1、优点

2、开发简单

3、技术栈灵活

4、服务独立无依赖

5、独立按需扩展

6、可用性高

7、缺点（挑战）

8、多服务运维难度

9、系统部署依赖

10、服务间通信成本

11、数据一致性

12、系统集成测试

13、重复工作

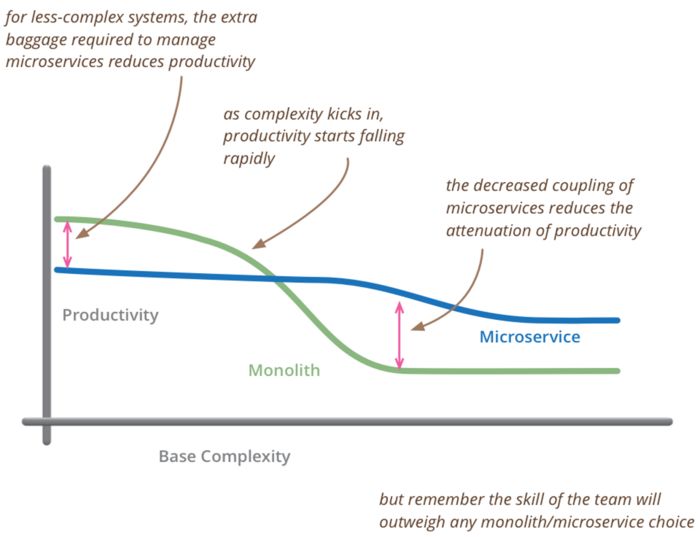
14、性能监控

没有最好的，只有适合自己的。



1、对于大的互联网公司，微服务架构是血液，是习惯，每家公司都有自己的套路和架构，细节有不同，但是核心理念是通的。

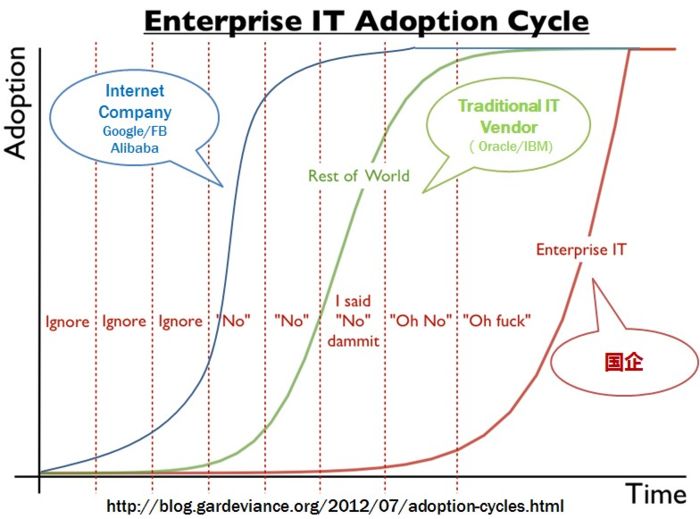
2、对于一般的公司而言，实践微服务有非常大的技术挑战，于是乎才有了这么多IT供应商考虑这里的商机。微服务比较适合未来有一定的扩展复杂度，且有 很大用户增量预期的应用，说人话就是新兴的互联网公司。创业初期，不可能买大量的机器或者很贵的机器，但是又必须考虑应对成功后的巨量的用户，微服务架构 成了最好的选择。



**So What - 思考**

看到上面的图，不是不觉得特别的熟悉？其实我们N年前就用的滚瓜烂熟了好不好？裤子都拖了。

其实本来所谓的微服务就是对互联网在应用技术的一个总结归纳，IT厂商鼓吹所有概念无非是为了生意（business），SOA是，Cloud是，Microservice也是。下面玩笑很有意思的概括了这个情况（我加了第一条线，原图点击阅读原文查看）



所以微服对我们的思考我觉得更多的是思维上的，对已微服务架构，技术上不是问题，意识比工具重要。

1、按照业务 或者客户需求组织资源（这是最难的）

2、做有生命的产品，而不是项目

3、头狼战队，全栈化

4、后台服务贯彻Single Responsibility Principle

5、VM->Docker （to PE）

6、DevOps (to PE)

同时，对于开发同学，有这么多的中间件和强大的PE支持固然是好事，我们也需要深入去了解这些中间件背后的原理，知其然知其所以然，设想下，如果我们是一个小公司的CTO，离开的阿里的大环境，在有限的技术资源如何通过开源技术实施微服务？

最后，一般提到微服务都离不开DevOps和Docker，理解微服务架构是核心，devops和docker是工具，是手段。下次在抽时间再学习整理下。

