**微服务架构**

目录

[微服务架构定义 3](#_Toc508352973)

[传统系统架构和微服务架构 3](#_Toc508352974)

[为什么需要微服务架构？ 4](#_Toc508352975)

[微服务架构优点 4](#_Toc508352976)

[微服务架构缺点 5](#_Toc508352977)

[微服务缺点 5](#_Toc508352978)

[需要考虑的问题 6](#_Toc508352979)

[微服务架构知识体系 7](#_Toc508352980)

[康威定律 9](#_Toc508352981)

[REST 9](#_Toc508352982)

[RPC 10](#_Toc508352983)

[Gateway 10](#_Toc508352984)

[微服务之间通信 10](#_Toc508352985)

[设计要素 11](#_Toc508352986)

[服务框架 13](#_Toc508352987)

[容器（Docker）与微服务 14](#_Toc508352988)

[开发方式影响 15](#_Toc508352989)

[微服务案例 16](#_Toc508352990)

[各语言技术框架 17](#_Toc508352991)

[微服务架构技术点 21](#_Toc508352992)

[微服务架构Spring Cloud和Dubbo 22](#_Toc508352993)

[Spring Cloud 22](#_Toc508352994)

参考资料：

<https://martinfowler.com/articles/microservices.html>

<http://www.cnblogs.com/wintersun/p/6219259.html>

<http://www.docker.org.cn/book/docker/what-is-docker-16.html>

<http://www.sohu.com/a/201844693_100038984>

<https://zhuanlan.zhihu.com/p/25450681>

<https://github.com/spring-cloud>

<https://github.com/dubbo>

<https://spring.io/>

<http://dubbo.io/>

<http://blog.csdn.net/jek123456/article/details/77440772/>

<http://microservices.io/>

知识体系搭建网站：

<http://treenpool.com/>

## 微服务架构定义

微服务架构风格[[1]](https://martinfowler.com/articles/microservices.html#footnote-etymology)是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在其自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

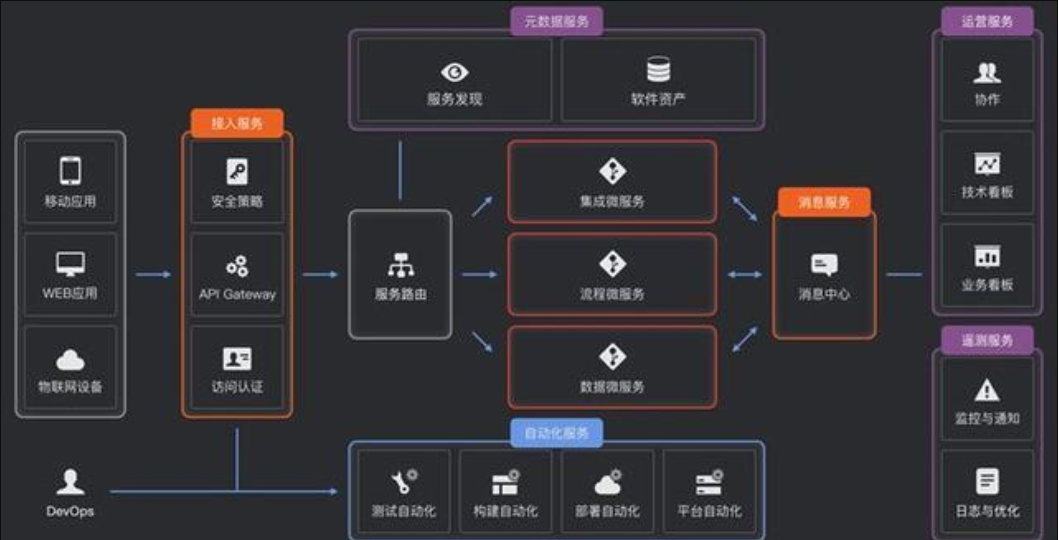
## 传统系统架构和微服务架构

传统的系统架构是单一架构模式。这种架构模式就是把应用整体打包部署，具体的样式依赖本身应用采用的语言，如果采用java语言，自然你会打包成war包，部署在Tomcat或者Jetty这样的应用服务器上，如果你使用spring boot还可以打包成jar包部署。其他还有Rails和Node.js应用以目录层次的形式打包。

微服务架构则是将单个的整体应用程序分割成更小的项目关联的独立的服务。一个服务通常实现一组独立的特性或功能，包含自己的业务逻辑和适配器。各个微服务之间的关联通过暴露api来实现。这些独立的微服务不需要部署在同一个虚拟机，同一个系统和同一个应用服务器中。

比如Amazon、eBay和NetFlix，通过采用微处理结构模式解决了上述问题。其思路不是开发一个巨大的单体式的应用，而是将应用分解为小的、互相连接的微服务。

**特点**：组件化、松耦合、自治、去中心化



## SOA与微服务架构的比较

SOA是一种能够改变整个企业的IT结构的战略创新，它将企业系统划分为不同的服务，为企业赋予了更大的灵活性……微服务必须能够独立地进行部署，而SOA服务往往是按照一体性的部署方式实现的。因此，虽然SOA与微服务技术有一定程度的相似性，但他们的本质是完全不同的。

谈论SOA的各种不足似乎已经成为了一件很普遍的事。但如果你认真地观察，就会发现SOA的缺陷中的绝大部分与微服务是相同的，只是有关他们的案例更为具体一些。而两者的优势其实也大体相同，因为从本质上看，这两种技术所做的都是同一件事：将一个较大的问题分解为多个较小的问题。

微服务只是一种为经过良好架构设计的SOA解决方案实现的面向服务的交付方案。

两者之间最关键的**区别**在于，微服务专注于**以自治的方式产生价值。**

## 为什么需要微服务架构？

单一架构模式在项目初期很小的时候开发方便，测试方便，部署方便，运行良好。可是当应用随着时间的推进，加入的功能越来越多，最终会变得巨大，一个项目中很有可能数百万行的代码，互相之间繁琐的jar包。

1、    不再适用敏捷开发，过于复杂，任何开发者都不能够完全理解，修复漏洞和实现新功能变得困难和耗时。

2、    规模越大，启动时间越长，自然会拖慢开发进度，一个小功能的修改部署起来变得困难，必须重新部署整个应用。

3、    系统的不同的模块的需要不同的特定的虚拟机环境时，由于是整体应用，那么只能折中选择。

4、    任意模块的漏洞或者错误都会影响这个应用，降低系统的可靠性

5、    还有一个如果想整体应用采用新的技术，新的框架或者语言，那是不可能的。

如果采用微服务架构模式，则可以解决单一架构模式带来的系统复杂性。主要包括以下几个好处：

1、    由于每个服务都是独立并且微小的，由单独的团队负责，仍然可以采用敏捷开发模式，自由的选择合适的技术，甚至可以重写老服务，当然都要遵守统一的API约定。

2、    每一个微服务都是独立部署的，可以进行快速迭代部署，根据各自服务需求选择合适的虚拟机和使用最匹配的服务资源要求的硬件。

3、    整体应用程序被分解成可管理的模块和服务，单个的服务可以更快的开发、更简单的理解和维护。

4、    一些需要进行负载均衡的服务可以部署在多个云虚拟机上，加入NGINX这样的负载均衡器在多个实例之间分发请求，这样不需要整个应用进行负载均衡了。

每个后端服务暴露一套REST API，大部分服务调用其他服务提供的API。每个服务都有自己的数据库模式，而不是共享单个数据库模式。尽管这会造成某些数据的冗余，但是对于微服务架构这个独立数据库模式是必要的，确保了独立服务之间的松散耦合。

以上介绍的微服务架构模式表面上类似于SOA，两种架构都包含一组服务。可以认为微服务架构是不包括Web服务规范（WS-）、企业服务总线（ESB）的SOA。基于微服务的应用倾向于使用更简单轻量级的协议，比如 REST 而不是 WS-。微服务自己实现类似 ESB 的功能并且拒绝 SOA 的其他部分，比如规范模式的概念。

## 微服务架构优点

* 每个微服务都很小，这样能聚焦一个指定的业务功能或业务需求。
* 微服务能够被小团队单独开发，这个小团队是2到5人的开发人员组成。
* 微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。
* 微服务能使用不同的语言开发。
* 微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins, bamboo 。
* 一个团队的新成员能够更快投入生产。
* 微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果。无需通过合作才能体现价值。
* 微服务允许你利用融合最新技术。
* 微服务只是业务逻辑的代码，不会和HTML,CSS 或其他界面组件混合。
* 微服务能够即时被要求扩展。
* 微服务能部署中低端配置的服务器上。
* 易于和第三方集成。
* 每个微服务都有自己的存储能力，可以有自己的数据库。也可以有统一数据库。

## 微服务架构缺点

* 微服务架构可能带来过多的操作。
* 需要DevOps技巧 (http://en.wikipedia.org/wiki/DevOps).
* 可能双倍的努力。
* 分布式系统可能复杂难以管理。
* 因为分布部署跟踪问题难。
* 当服务数量增加，管理复杂性增加。

## 微服务缺点

1、  微服务应用作为分布式系统带来了复杂性。当应用是整体应用程序时，模块之间调用都在应用之内，即使进行分布式部署，依然在应用内调用。可是微服务是多个独立的服务，当进行模块调用的时候，分布式将会麻烦。

2、  多个独立数据库，事务的实现更具挑战性。

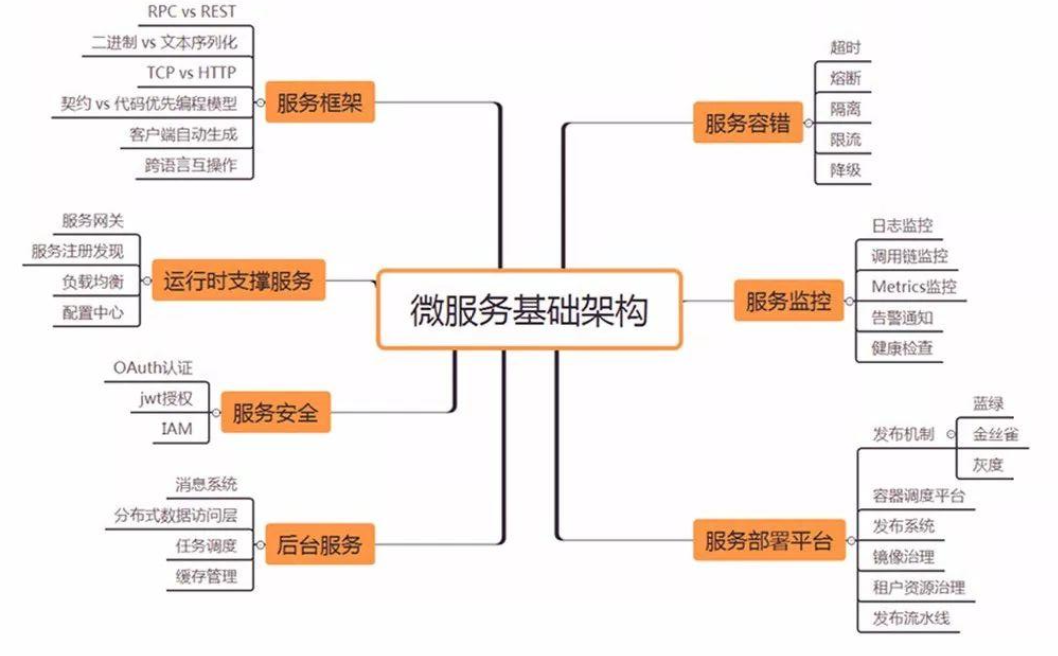
3、  测试微服务变得复杂，当一个服务依赖另外一个服务时，测试时候需要另外一个服务的支持。

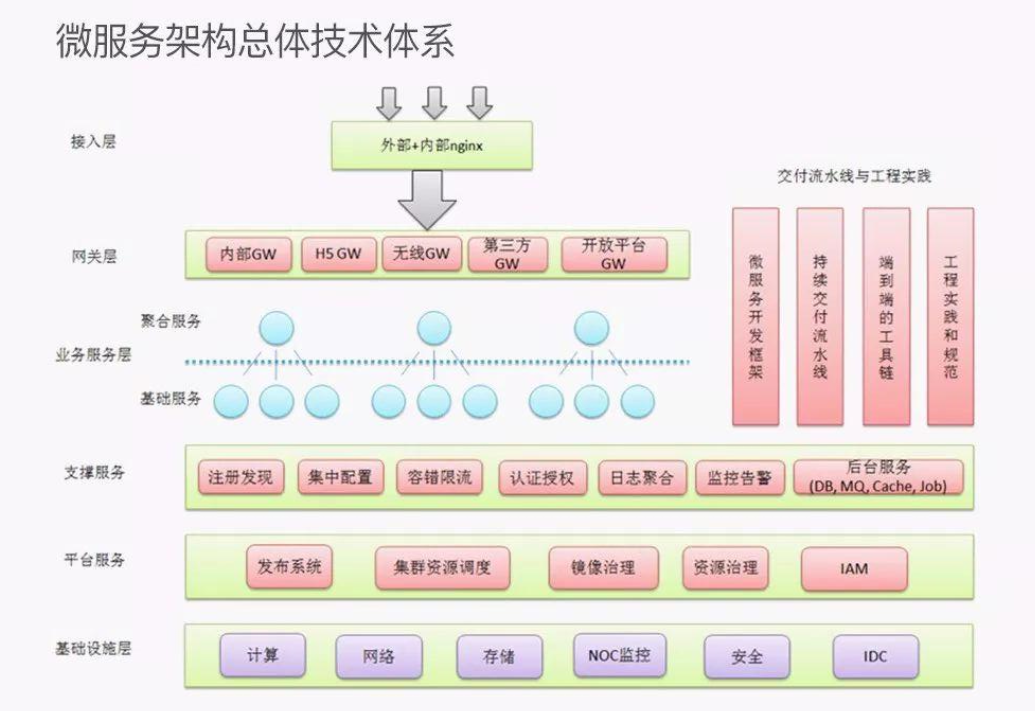
4、  部署基于微服务的应用也很复杂，整体应用程序部署只需要部署在一组相同的服务器上，在这些服务前面加入传统的负载均衡器即可。独立服务的不是讲变得复杂，需要更高的自动化形式。

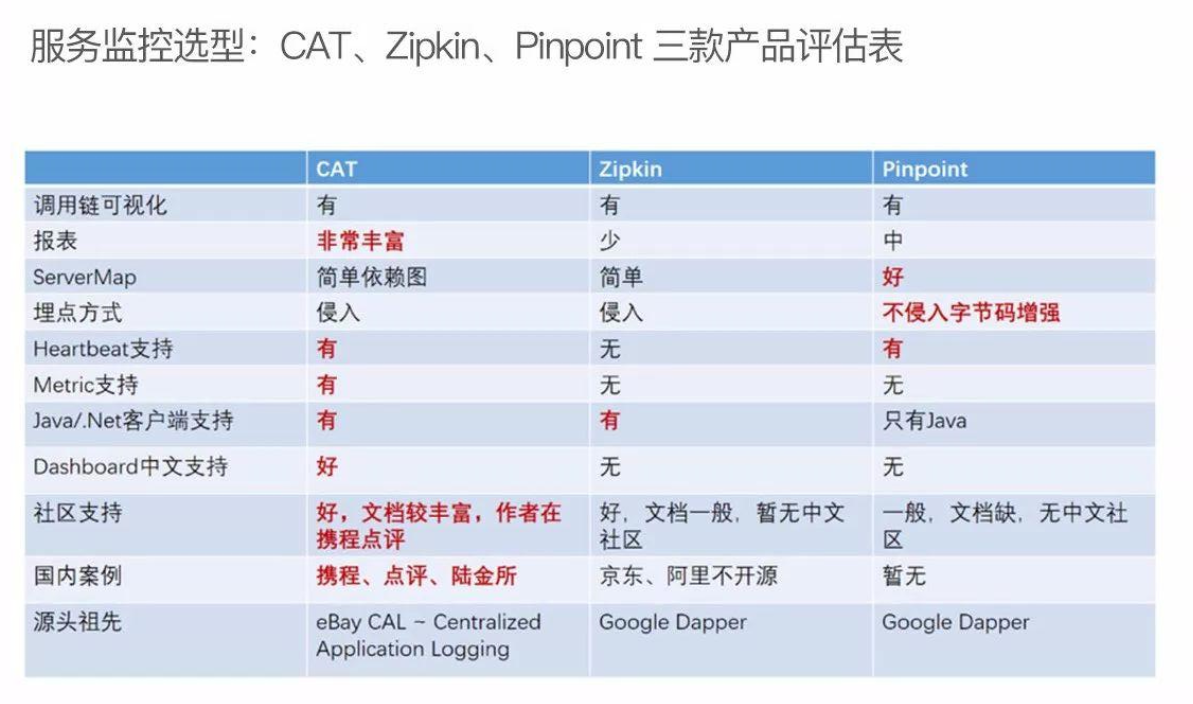
## 需要考虑的问题

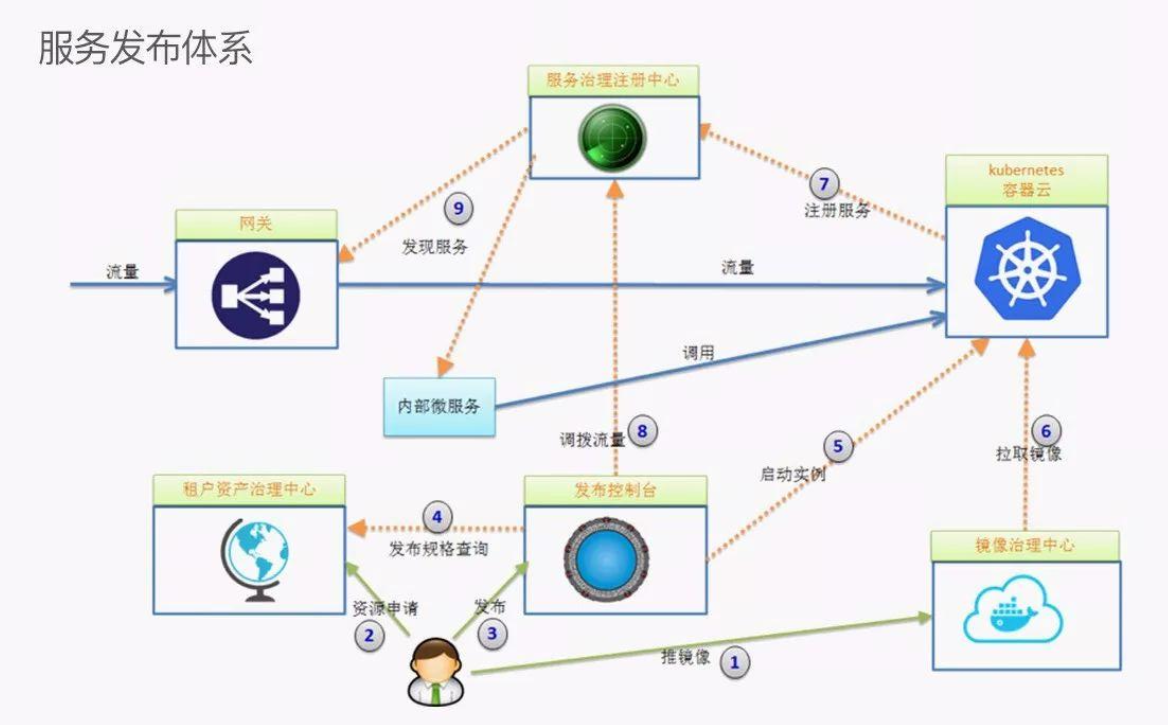
* 单个微服务代码量小，易修改和维护。但是，系统复杂度的总量是不变的，每个服务代码少了，但服务的个数肯定就多了。就跟拼图游戏一样，切的越碎，越难拼出整幅图。一个系统被拆分成零碎的微服务，最后要集成为一个完整的系统，其复杂度肯定比大块的功能集成要高很多。
* 单个微服务数据独立，可独立部署和运行。虽然微服务本身是可以独立部署和运行的，但仍然避免不了业务上的你来我往，这就涉及到要对外通信，当微服务的数量达到一定量级的时候，如何提供一个高效的集群通信机制成为一个问题。
* 单个微服务拥有自己的进程，进程本身就可以动态的启停，为无缝升级的打好了基础，但谁来启动和停止进程，什么时机，选择在哪台设备上做这件事情才是无缝升级的关键。这个能力并不是微服务本身提供的，而是需要背后强大的版本管理和部署能力。
* 多个相同的微服务可以做负载均衡，提高性能和可靠性。正是因为相同微服务可以有多个不同实例，让服务按需动态伸缩成为可能，在高峰期可以启动更多的相同的微服务实例为更多用户服务，以此提高响应速度。同时这种机制也提供了高可靠性，在某个微服务故障后，其他相同的微服务可以接替其工作，对外表现为某个设备故障后业务不中断。同样的道理，微服务本身是不会去关心系统负载的，那么什么时候应该启动更多的微服务，多个微服务的流量应该如何调度和分发，这背后也有一套复杂的负载监控和均衡的系统在起作用。
* 微服务可以独立部署和对外提供服务，微服务的业务上线和下线是动态的，当一个新的微服务上线时，用户是如何访问到这种新的服务？这就需要有一个统一的入口，新的服务可以动态的注册到这个入口上，用户每次访问时可以从这个入口拿到系统所有服务的访问地址。这个统一的系统入口并不是微服务本身的一部分，所以这种能力需要系统单独提供。
* 还有一些企业级关注的系统问题，比如，安全策略如何集中管理？系统故障如何快速审计和跟踪到具体服务？整个系统状态如何监控？服务之间的依赖关系如何管理？等等这些问题都不是单个微服务考虑的范畴，而需要有一个系统性的考虑和设计，让每个微服务都能够按照系统性的要求和约束提供对应的安全性，可靠性，可维护性的能力。

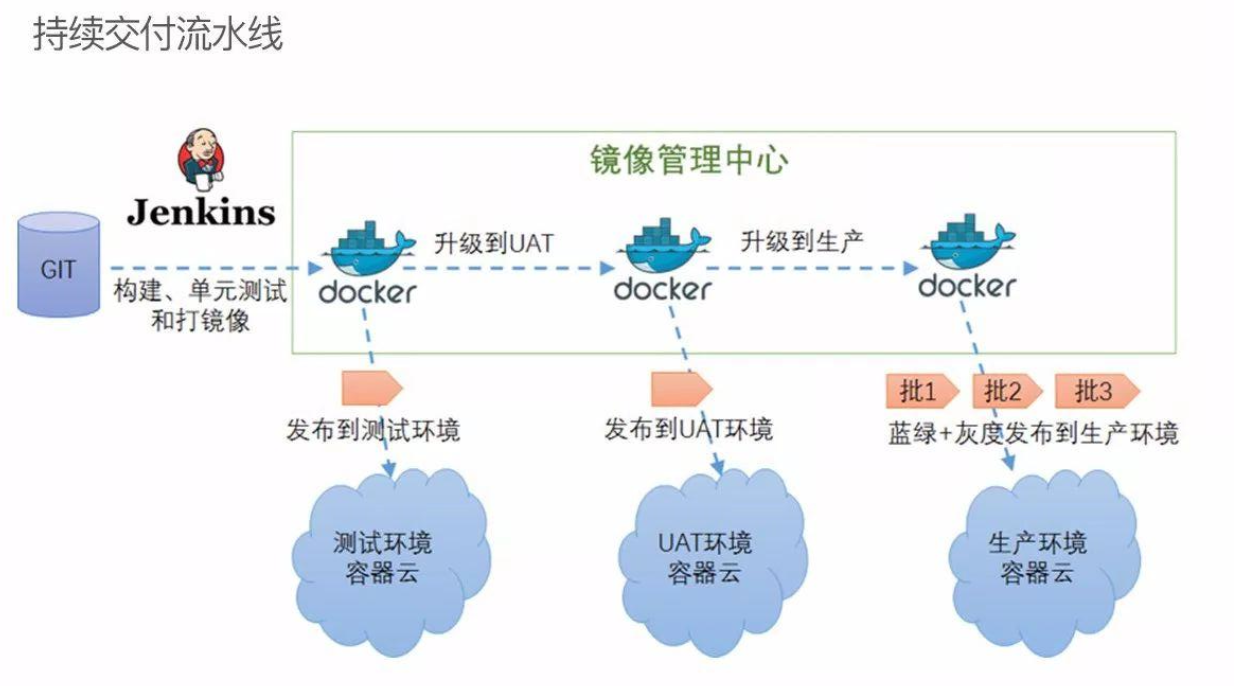
## 微服务架构知识体系











## 康威定律

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. - Melvin Conway(1967)

设计系统的组织受限于生产这些组织的通信结构的设计

组织形式等同系统设计。

这篇论文中的其他一些核心观点，如下：

**第一定律：** Communication dictates design（组织沟通方式会通过系统设计表达出来）

**第二定律：** There is never enough time to do something right, but there is always enough time to do it over（时间再多一件事情也不可能做的完美，但总有时间做完一件事情）

**第三定律：** There is a homomorphism from the linear graph of a system to the linear graph of its design organization（线型系统和线型组织架构间有潜在的异质同态特性）

**第四定律：** The structures of large systems tend to disintegrate during development, qualitatively more so than with small systems（大的系统组织总是比小系统更倾向于分解）

## REST

微服务架构采用轻量级协议—rest；

REST即表述性状态传递（英文：Representational State Transfer，简称REST）是Roy Fielding博士在2000年他的博士论文中提出来的一种[软件架构](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E6%9E%B6%E6%9E%84)风格。

REST 定义了一组体系架构原则，您可以根据这些原则设计以系统资源为中心的 Web 服务，包括使用不同语言编写的客户端如何通过 HTTP 处理和传输资源状态。 如何考虑使用它的 Web 服务的数量，REST 近年来已经成为最主要的 Web 服务设计模式。 事实上，REST 对 Web 的影响非常大，由于其使用相当方便，已经普遍地取代了基于 SOAP 和 WSDL 的接口设计。

## RPC

RPC（Remote Procedure Call）—[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8)，它是一种通过[网络](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C)从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。[RPC协议](https://baike.baidu.com/item/RPC%E5%8D%8F%E8%AE%AE)假定某些[传输协议](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI[网络通信](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1)模型中，RPC跨越了[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82)和[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82)。RPC使得开发包括网络[分布式](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F)多程序在内的应用程序更加容易。

## RESTful

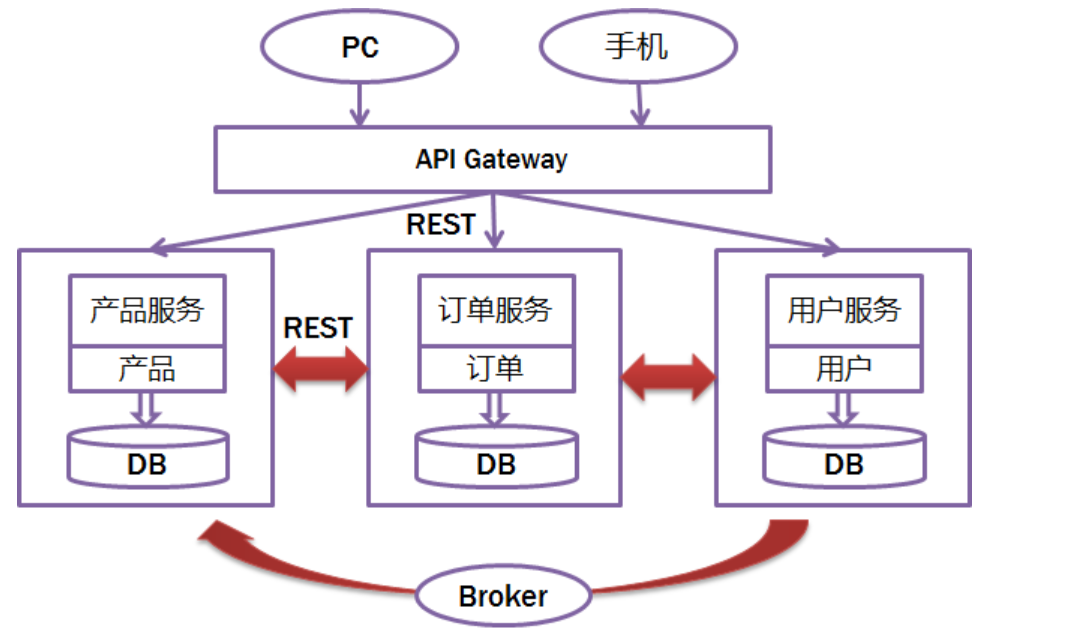
一种软件架构风格、设计风格，而**不是**标准，只是提供了一组设计原则和约束条件。它主要用于客户端和服务器交互类的软件。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

## Gateway

应用并不直接访问后台服务，而是通过API Gateway（网关）来传递中间消息。API Gateway负责负载均衡、缓存、访问控制、API 计费监控等等任务，可以通过NGINX方便实现

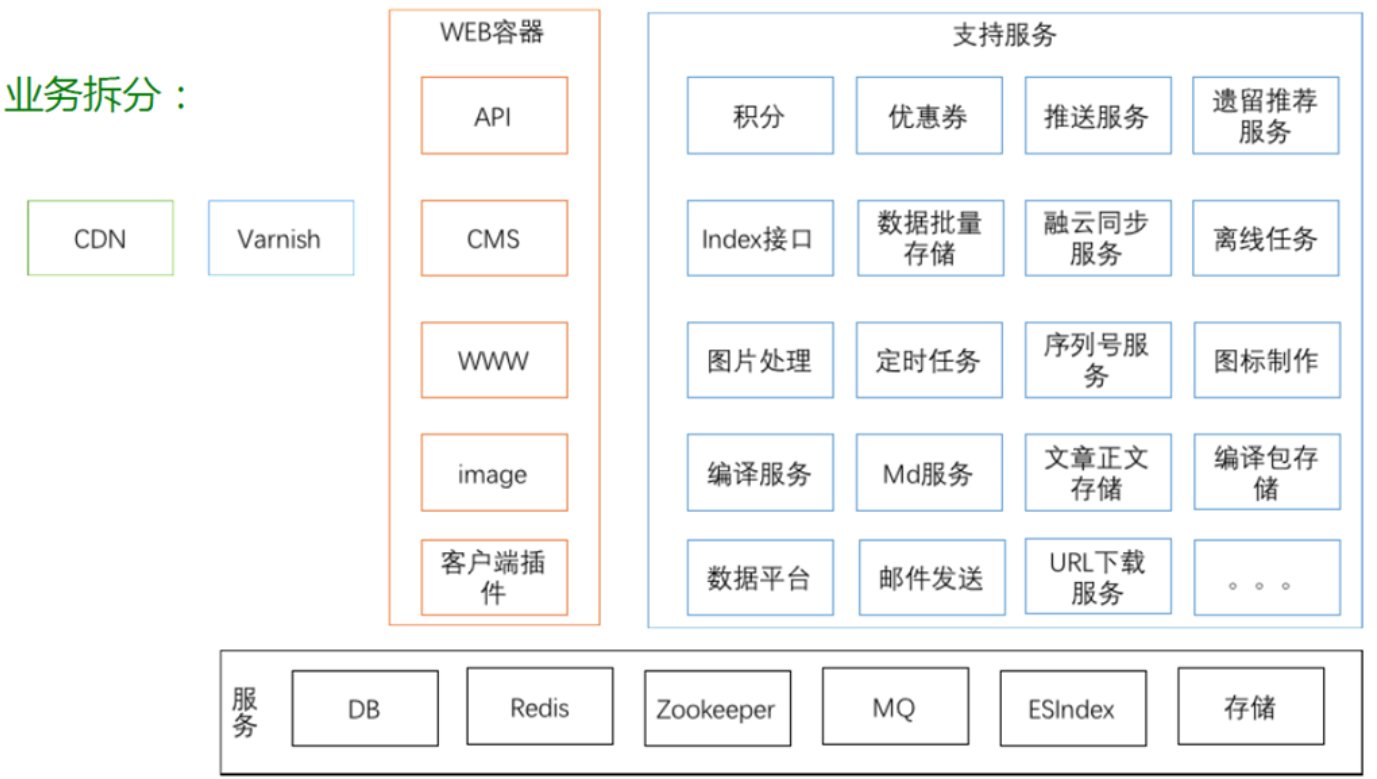
**不像传统多个服务共享一个数据库，微服务架构每个服务都有自己的数据库**

## 微服务之间通信

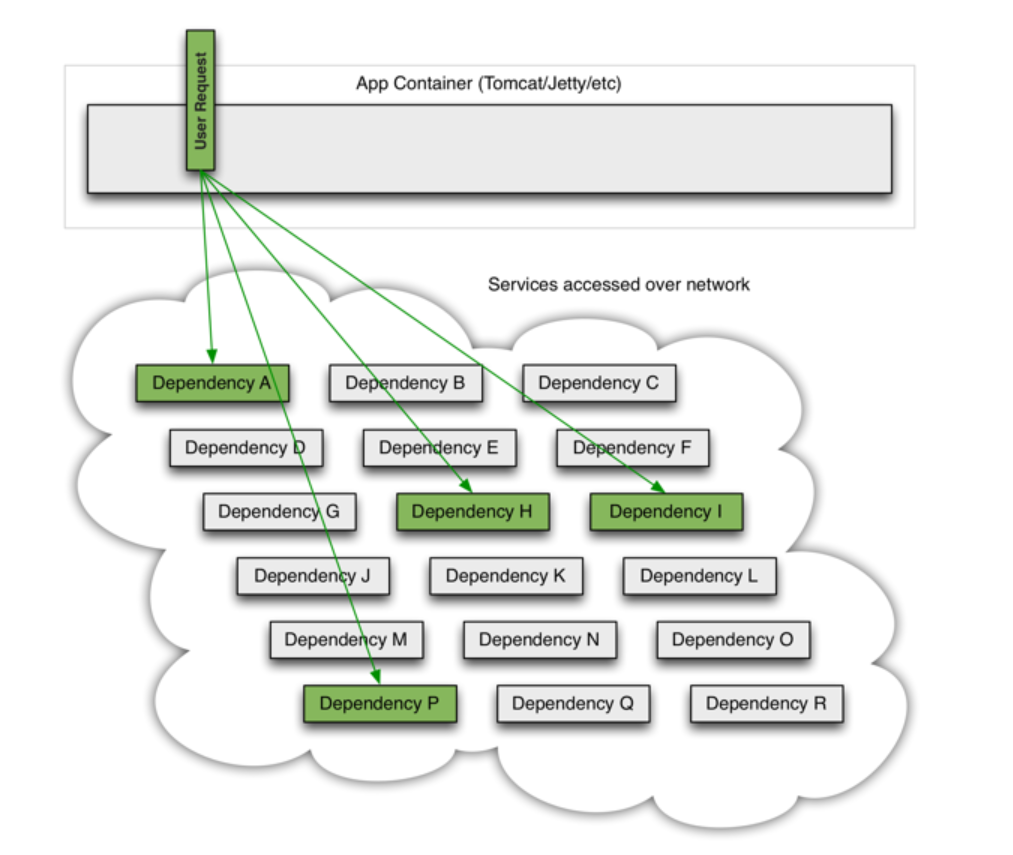


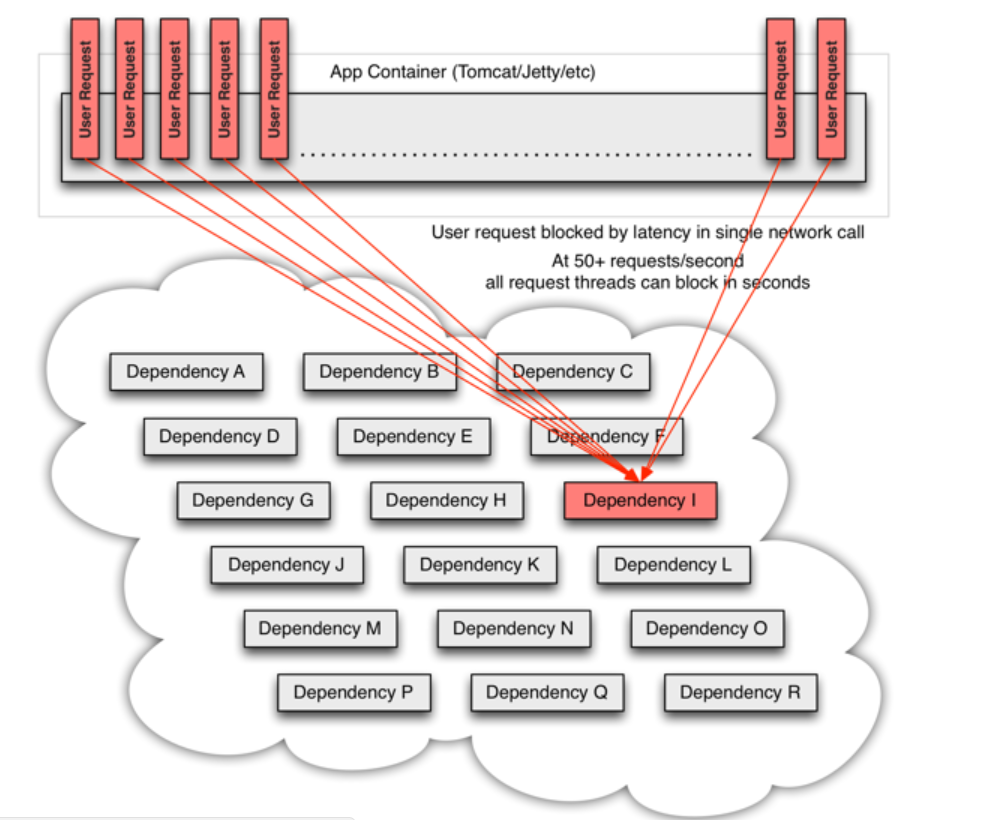
。RESTful和RPC的比较也是一个很有意 思的话题。一般REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要 求，只要封装了HTTP的SDK就能调用，所以相对使用的广一些。RPC（[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8)，它是一种通过[网络](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C)从远程计算机程序上请求服务，而不需要了解底层网络技术的协议）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。

## 设计要素



服务依赖





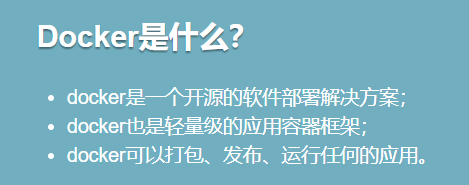
## 服务框架

1. 服务注册、发现、负载均衡和健康检查，假定采用进程内LB方案，那么服务自注册一般统一做在服务器端框架中，健康检查逻辑由具体业务服务定制，框架层提供调用健康检查逻辑的机制，服务发现和负载均衡则集成在服务客户端框架中。
2. 监控日志，框架一方面要记录重要的框架层日志、metrics和调用链数据，还要将日志、metrics等接口暴露出来，让业务层能根据需要记录业务日志数据。在运行环境中，所有日志数据一般集中落地到企业后台日志系统，做进一步分析和处理。
3. REST/RPC和序列化，框架层要支持将业务逻辑以HTTP/REST或者RPC方式暴露出来，HTTP/REST是当前主流API暴露方式，在性能要求高的场合则可采用Binary/RPC方式。针对当前多样化的设备类型(浏览器、普通PC、无线设备等)，框架层要支持可定制的序列化机制，例如，对浏览器，框架支持输出Ajax友好的JSON消息格式，而对无线设备上的Native App，框架支持输出性能高的Binary消息格式。
4. 配置，除了支持普通配置文件方式的配置，框架层还可集成动态运行时配置，能够在运行时针对不同环境动态调整服务的参数和配置。
5. 限流和容错，框架集成限流容错组件，能够在运行时自动限流和容错，保护服务，如果进一步和动态配置相结合，还可以实现动态限流和熔断。
6. 管理接口，框架集成管理接口，一方面可以在线查看框架和服务内部状态，同时还可以动态调整内部状态，对调试、监控和管理能提供快速反馈。Spring Boot微框架的Actuator模块就是一个强大的管理接口。
7. 统一错误处理，对于框架层和服务的内部异常，如果框架层能够统一处理并记录日志，对服务监控和快速问题定位有很大帮助。
8. 安全，安全和访问控制逻辑可以在框架层统一进行封装，可做成插件形式，具体业务服务根据需要加载相关安全插件。
9. 文档自动生成，文档的书写和同步一直是一个痛点，框架层如果能支持文档的自动生成和同步，会给使用API的开发和测试人员带来极大便利。Swagger是一种流行Restful API的文档方案。

## 容器（Docker）与微服务

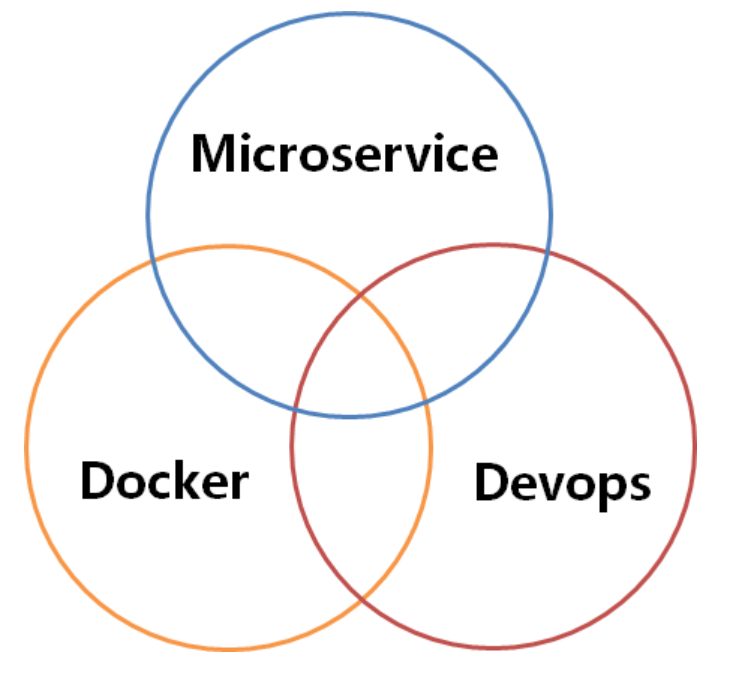
•容器够小   
–解决微服务对机器数量的诉求   
•容器独立   
–解决多语言问题   
•开发环境与生产环境相同   
–单机开发、提升效率   
•容器效率高   
–省钱   
•代码/image一体化   
–可复用管理系统   
•容器的横向与纵向扩容   
–可复制   
–可动态调节CPU与内存

Docker是一个开源的引擎，可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器。开发者在笔记本上编译测试通过的容器可以批量地在生产环境中部署，包括VMs（虚拟机）、[bare metal](http://www.whatis.com.cn/word_5275.htm)、OpenStack 集群和其他的基础应用平台。

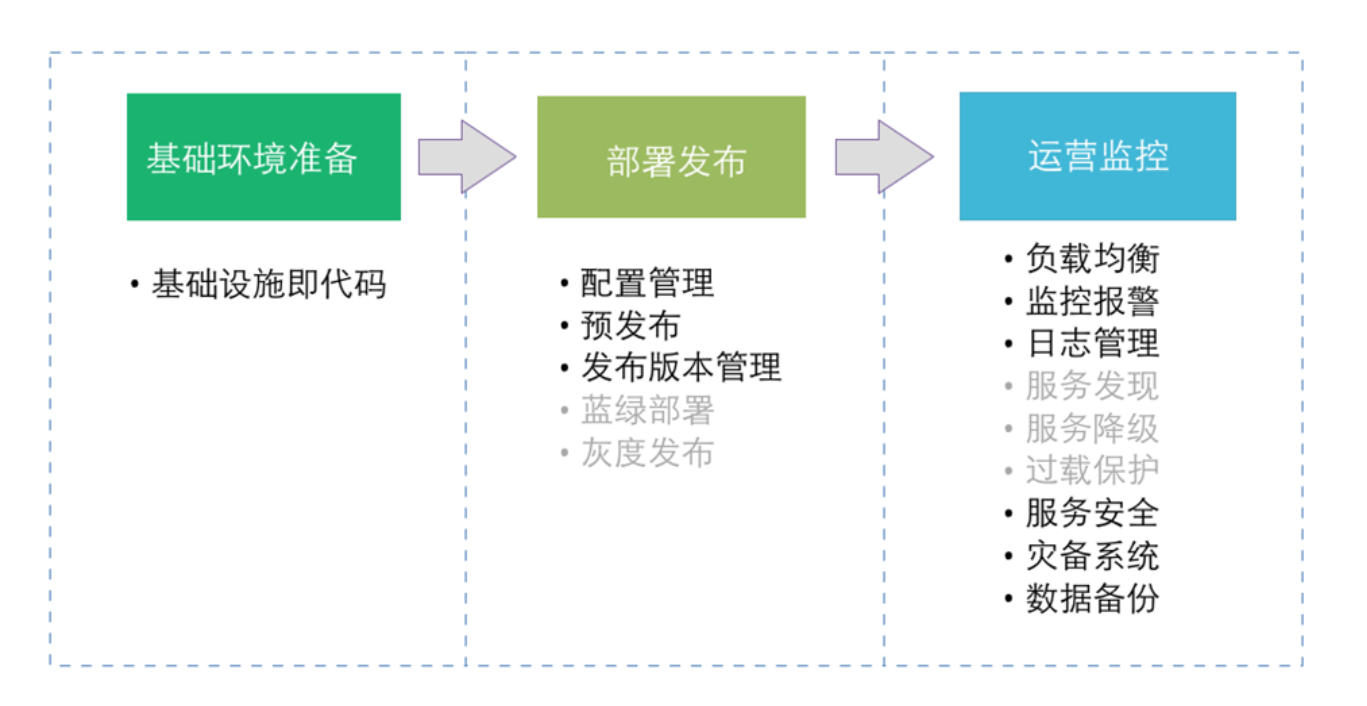


## 开发方式影响

随着持续交付概念推广以及Docker容器普及，微服务将这两种理念和技术结合起来，形成新的微服务+API + 平台的开发模式，提出了容器化微服务的持续交付概念。

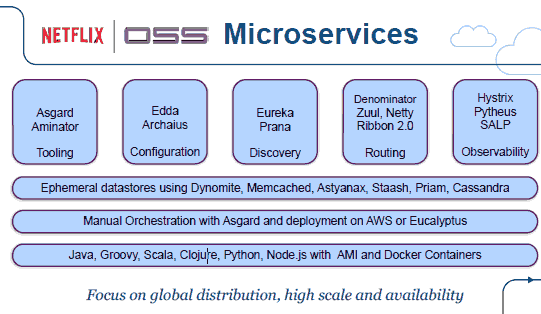


    微服务的实施是有一定的先决条件：基础的运维能力（如监控、快速配置、快速部署）需提前构建，否则就会陷入如我们般被动的局面。推荐采用[基础设施及代码](http://martinfowler.com/bliki/InfrastructureAsCode.html)的实践，通过代码来描述计算和网络基础设施的方法，使得图案度i可以快速安全的搭建和处理由新的配置代替的服务器，服务器之间可以拥有更高的一致性，降低了在“我的环境工作，而你的环境不工作”的可能，也是为后续的发布策略和运维提供更好的支撑。

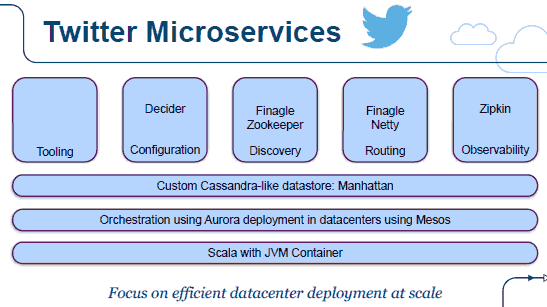


## 微服务案例

Netflix的微服务架构如下，着重全球分发 高可扩展性和可用性：

[](http://images2015.cnblogs.com/blog/15172/201612/15172-20161225120509526-1212620907.png)

Twitter的微服务架构，注重高效的可扩展的数据中心：

[](http://images2015.cnblogs.com/blog/15172/201612/15172-20161225120510542-544684455.png)

## 各语言技术框架

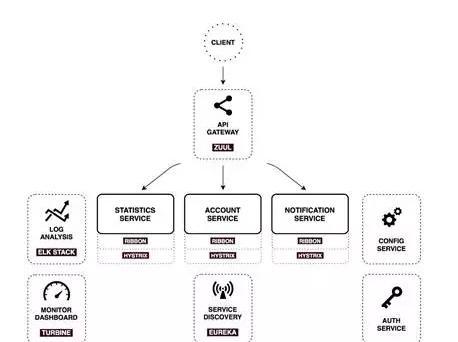
Java语言相关微服务框架

1.Spring Boot

Spring Boot的设计目的是简化新Spring应用初始搭建以及开发过程，2017年有64.4%的受访者决定使用Spring Boot，可以说是最受欢迎的微服务开发框架。利用Spring Boot开发的便捷度简化分布式系统基础设施的开发，比如像配置中心、注册、负载均衡等方面都可以做到一键启动和一键部署。

2.Spring Cloud

Spring Cloud是一个系列框架的合计，基于HTTP（s）的RETS服务构建服务体系，Spring Cloud能够帮助架构师构建一整套完整的微服务架构技术生态链。



3.Dubbo

Dubbo是由阿里巴巴开源的分布式服务化治理框架，通过RPC请求方式访问。Dubbo是在阿里巴巴的电商平台中逐渐探索演进所形成的，经历过复杂业务的高并发挑战，比Spring Cloud的开源时间还要早。目前阿里、京东、当当、携程、去哪等一些企业都在使用Dubbo。

4.Dropwizard

Dropwizard将Java生态系统中各个问题域里最好的组建集成于一身，能够快速打造一个Rest风格的后台，还可以整合Dropwizard核心以外的项目。国内现在使用Dropwizard还很少，资源也不多，但是与SpringBoot相比，Dropwizard在轻量化上更有优势，同时如果用过Spring，那么基本也会使用SpringBoot。

5.Akka

Akka是一个用Scala编写的库，可以用在有简化编写容错、高可伸缩性的Java和Scala的Actor模型，使用Akka能够实现微服务集群。

6.Vert.x/ Lagom/ ReactiveX/Spring 5

这四种框架主要用于响应式微服务开发，响应式本身和微服务没有关系，更多用于提升性能上，但是可以和微服务相结合，也可以提升性能。



.Net相关微服务框架

1. .NET Core

.NET Core是专门针对模块化微服务架构设计的，是跨平台应用程序开发框架，是微软开发的第一个官方版本。

2.Service Fabric

Service Fabric是微软开发的一个微服务框架，基于Service Fabric构建的很多云服务被用在了Azure上。

3.Surging

Surging是基于RPC协议的分布式微服务技术框架，基于.NET Core而来。

4.Microdot Framework

Microdot Framework用于编写定义服务逻辑代码，不需要解决开发分布式系统的挑战，能够很方便的进行MicrosoftOrleans集成。

Node.js相关微服务框架

1.Seneca

Seneca是Node.js的微服务框架开发工具，可以用于编写可用于产品环境的代码。

2.Hapi/ restify/ LoopBack

这三种框架的分工不同，前两种更适合开发简单的微服务后端系统，第三种更适合用在大型复杂应用开发，还可以用在现有微服务上的构建。

Go相关微服务框架

Go-Kit/Goa/Dubbogo

Go-Kit是分布式开发的工具合集，适合用于大型业务场景下构建微服务；Goa是用Go语言构建的微服务框架；Dubbogo是和阿里巴巴开源的Dubbo能够兼容的Golang微服务框架。

Python相关微服务框架

Python相关的微服务框架非常少，用的比较多的是Nameko。Nameko让实现微服务变得更简单，同时也提供了很丰富的功能，比如支持负载均衡、服务发现还支持依赖自动注入等，使用起来很方便，但是有限速、超时和权限机制不完善等缺点。

## 微服务架构技术点

**View：**  
H5、Vue.js、Spring Tag、React、angularJs

**Spring Boot/Spring Cloud：**  
Zuul、Ribbon、Feign、Turbine、Hystrix、Oauthor2、Sleuth、API Gateway、Spring Cloud、Config Eureka、SSO、Spring Cloud、  
BUS、Turbine、Zipkin、Cache、Spring Cloud Admin、API Gateway、ELK Spring Cloud Security、 Spring Cloud Stream

**Component：**  
RoketMQ、Kafka、MongoDB、OSS、Redis、Swagger、Zuul、Label、BASE、Charts、Utils

**DAO：**  
Spring Data、Mybatis、OSS、 DTO

**Data Storage：**  
RDBS DFS、NOSQL/Hadoop

**Infrastructure：**  
LogBack、BUS、Jenkins、Zipkin、Druid、Swagger、Docker

## 微服务架构Spring Cloud和Dubbo



## Spring Cloud

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中的配置管理、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操作提供了一种简单的开发方式

Spring Cloud包含了多个子项目（针对分布式系统中涉及的多个不同开源产品），比如：Spring Cloud Config、Spring Cloud Netflix、Spring Cloud CloudFoundry、Spring Cloud AWS、Spring Cloud Security、Spring Cloud Commons、Spring Cloud Zookeeper、Spring Cloud CLI等项目。