<https://mp.weixin.qq.com/s/8qo-mBepuY7TCuKZGU7DHg>

# 12张手绘图，终于搞懂了微服务架构

微服务的概念最早在 2012 年提出，在 Martin Fowler 的大力推广下，微服务在 2014 年后得到了大力发展。今天我们通过一组手绘图来梳理下微服务的核心架构。

### 什么是微服务？

微服务 Microservices 之父，马丁.福勒，对微服务大概的概述如下：

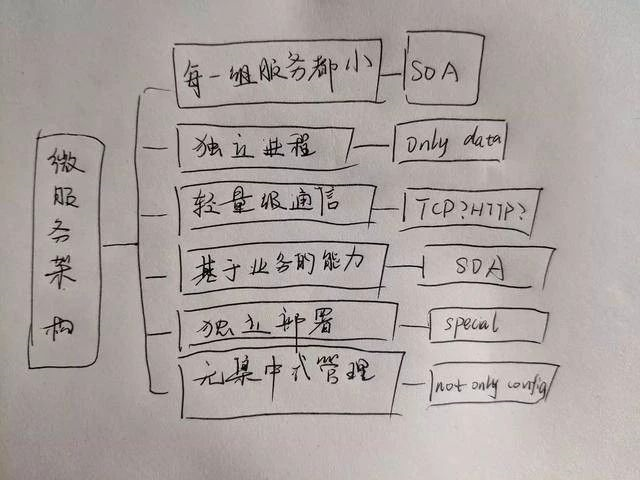
就目前而言，对于微服务业界并没有一个统一的、标准的定义（While there is no precise definition of this architectural style ) 。

但通常在其而言，**微服务架构是一种架构模式或者说是一种架构风格，它提倡将单一应用程序划分成一组小的服务，每个服务运行独立的自己的进程中，服务之间互相协调、互相配合，为用户提供最终价值。**

服务之间采用轻量级的通信机制互相沟通（通常是基于 HTTP 的 RESTful API ) 。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。

另外，应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建，可以有一个非常轻量级的集中式管理来协调这些服务。可以使用不同的语言来编写服务，也可以使用不同的数据存储。

根据马丁.福勒的描述，我总结了以下几点：



**①小服务**

小服务，没有特定的标准或者规范，但他在总体规范上一定是小的。

**②进程独立**

每一组服务都是独立运行的，可能我这个服务运行在 Tomcat 容器，而另一个服务运行在 Jetty 上。可以通过进程方式，不断的横向扩展整个服务。

**③通信**

过去的协议都是很重的，就像 ESB，就像 SOAP，轻通信，这意味着相比过去更智能更轻量的服务相互调用，就所谓 smart endpoints and dumb pipes。

这些 Endpoint 都是解耦的，完成一个业务通信调用串起这些 Micro Service 就像是 Linux 系统中通过管道串起一系列命令业务。

过去的业务，我们通常会考虑各种各样的依赖关系，考虑系统耦合带来的问题。微服务，可以让开发者更专注于业务的逻辑开发。

**④部署**

不止业务要独立，部署也要独立。不过这也意味着，传统的开发流程会出现一定程度的改变，开发的适合也要有一定的运维职责。

**⑤管理**

传统的企业级 SOA 服务往往很大，不易于管理，耦合性高，团队开发成本比较大。

微服务，可以让团队各思其政的选择技术实现，不同的 Service 可以根据各自的需要选择不同的技术栈来实现其业务逻辑。

### 微服务的利与弊

为什么用微服务呢？因为好玩？不是的。下面是我从网络上找到说的比较全的优点：

* 每个服务足够内聚，足够小，代码容易理解这样能聚焦一个指定的业务功能或业务需求。
* 开发简单、开发效率提高，一个服务可能就是专一的只干一件事。
* 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。
* 易于和第三方集成，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，Hudson，bamboo。
* 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。
* 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。

总的来说，微服务的优势，就是在于，面对大的系统，可以有效的减少复杂程度，使服务架构的逻辑更清晰明了。  
但是这样也会带来很多问题，就譬如分布式环境下的数据一致性，测试的复杂性，运维的复杂性。

### 什么组织适合使用微服务？

**什么组织适合使用微服务？**

①墨菲定律（设计系统）和康威定律（系统划分）

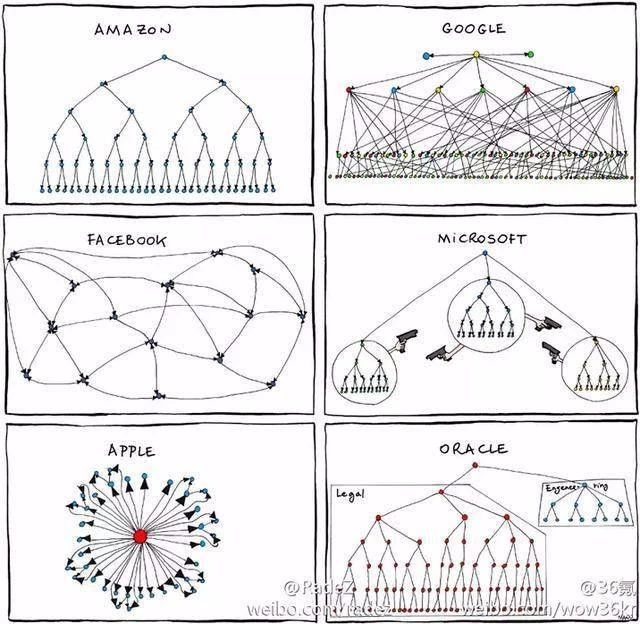
康威定律，是一个五十多年前就被提出来的微服务概念。在康威的这篇文章中，最有名的一句话就是：

Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

-Melvin Conway(1967)

中文直译大概的意思就是：设计系统的组织，其产生的设计等同于组织之内、组织之间的沟通结构。

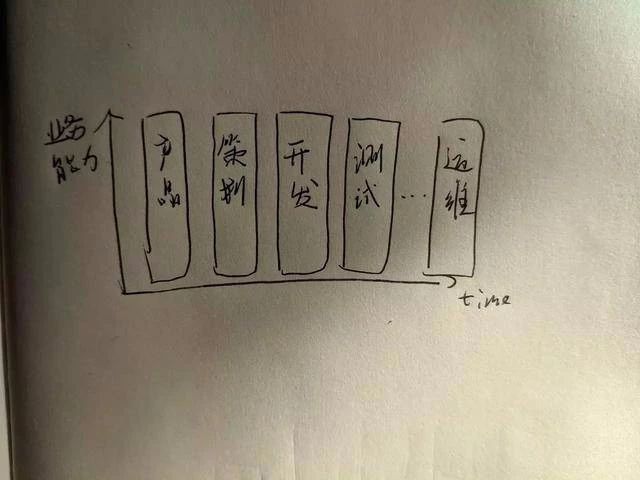
看看下面的图片，再想想 Apple 的产品、微软的产品设计，就能形象生动的理解这句话。



感兴趣的各位可以研究一下！

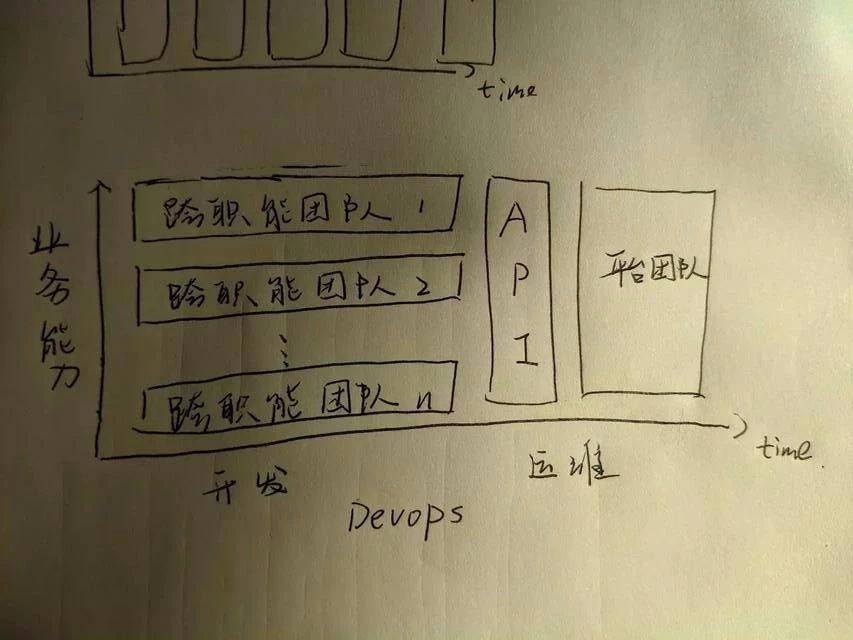
**②架构演化**

架构是不断演化出来的，微服务也是这样，当从各大科技公司，规模大到一定程度，完全需要演化成更进一步管理的技术架构体系。

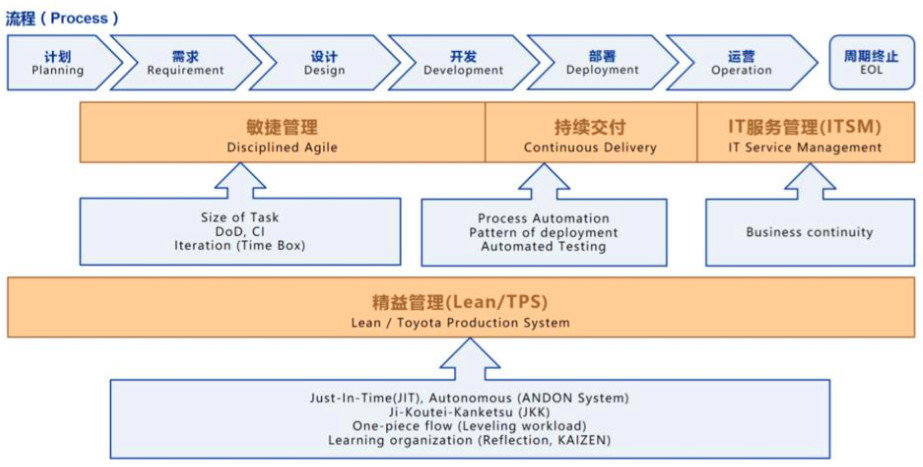


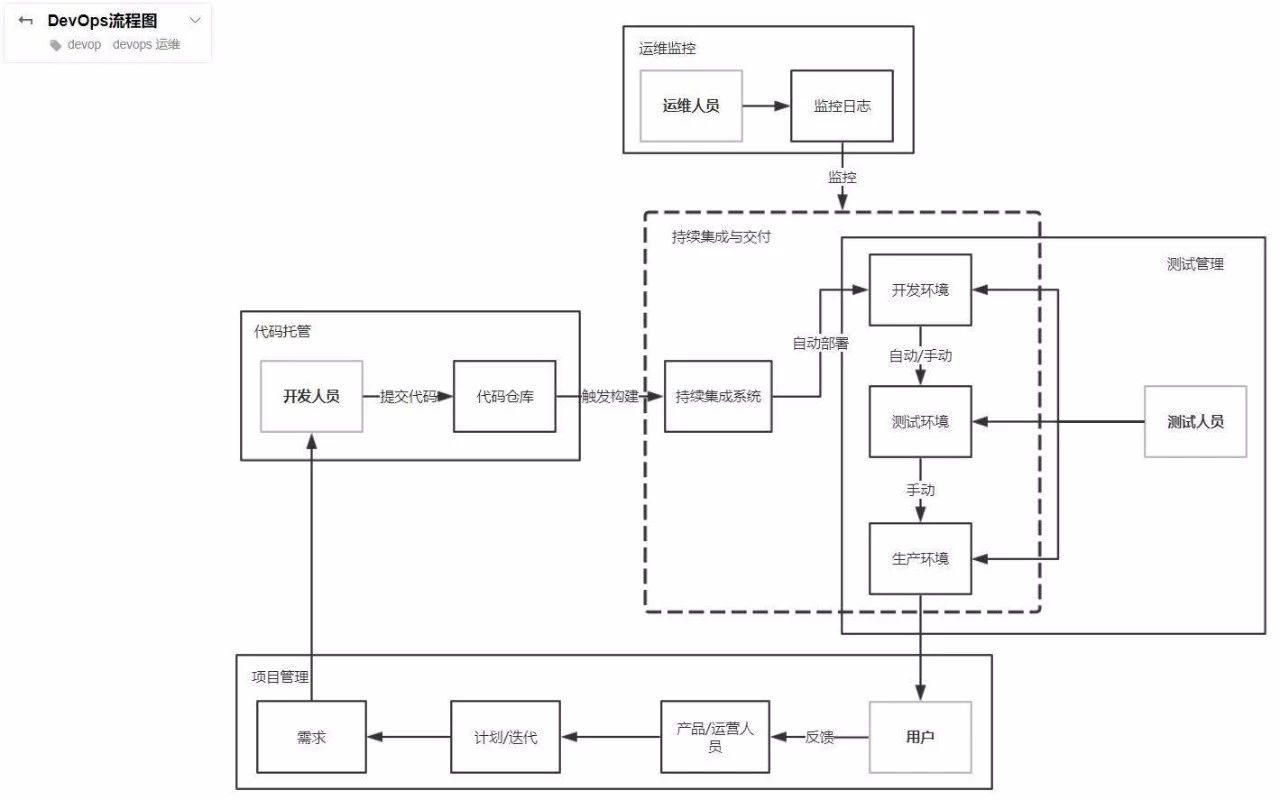
传统的团队，都是面向过程化的，产品想完了去找策划，策划完了找开发，接着顺着一步一步找。

我们做技术都是为了产品的，一旦过程出来了什么问题，回溯寻找问题会非常耗时。



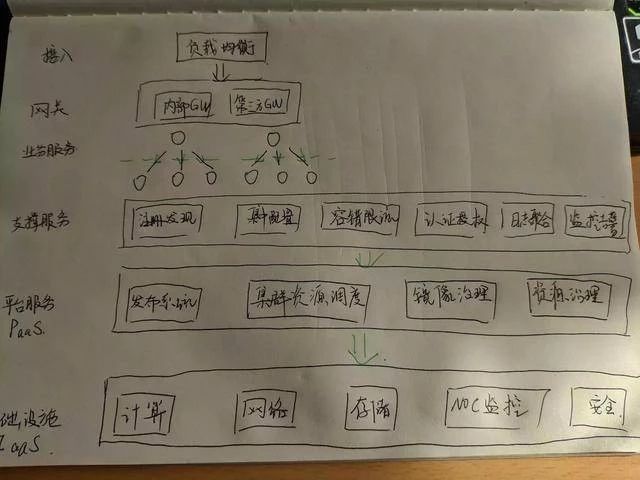
使用了微服务架构体系，团队组织方式需要转变成跨职能团队，即每个团队都有产品专家，策划专家，开发专家，运维专家，他们使用 API 方式发布他们的功能，而平台使用他们的功能发布产品。





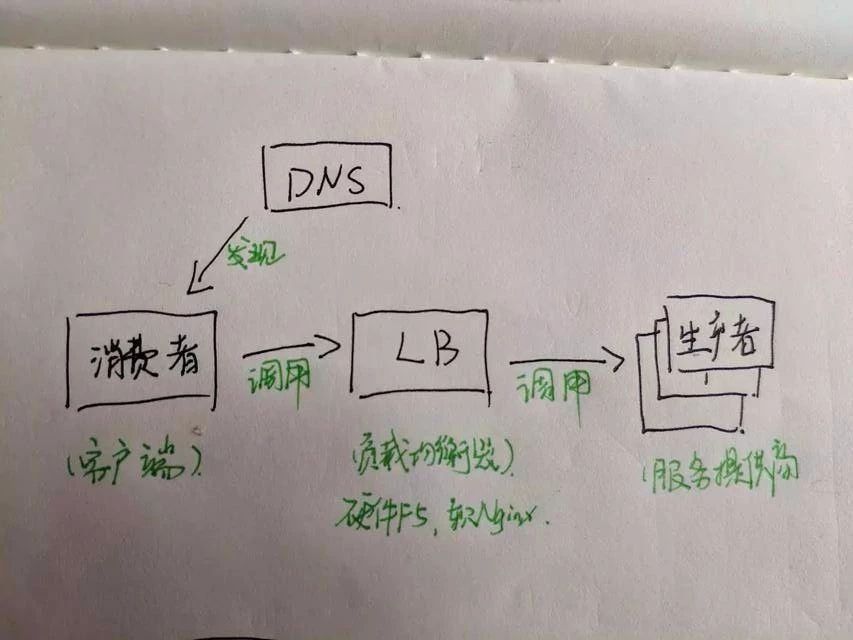
### 微服务技术架构体系

下面我分享一下大部分公司都使用的微服务技术架构体系：



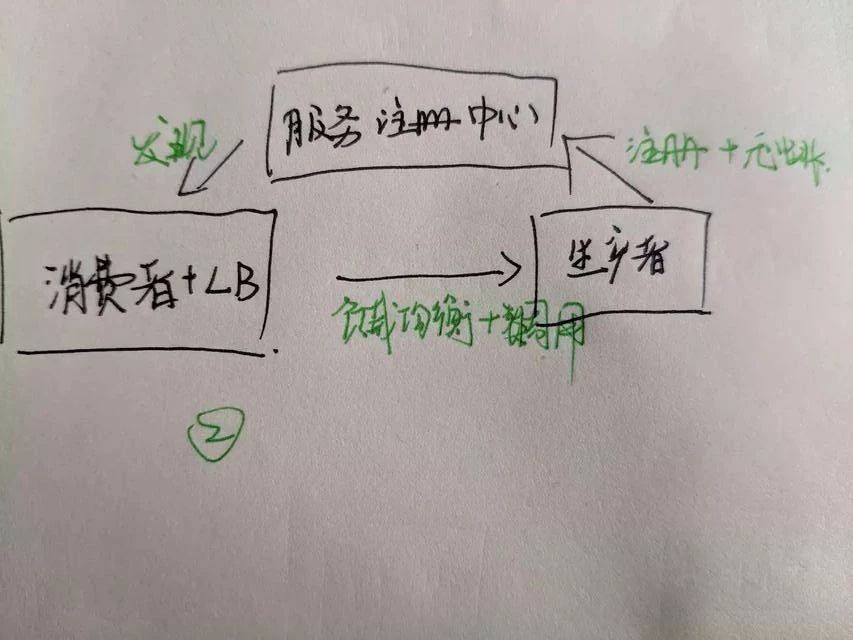
#### 服务发现

主流的服务发现，分为三种：



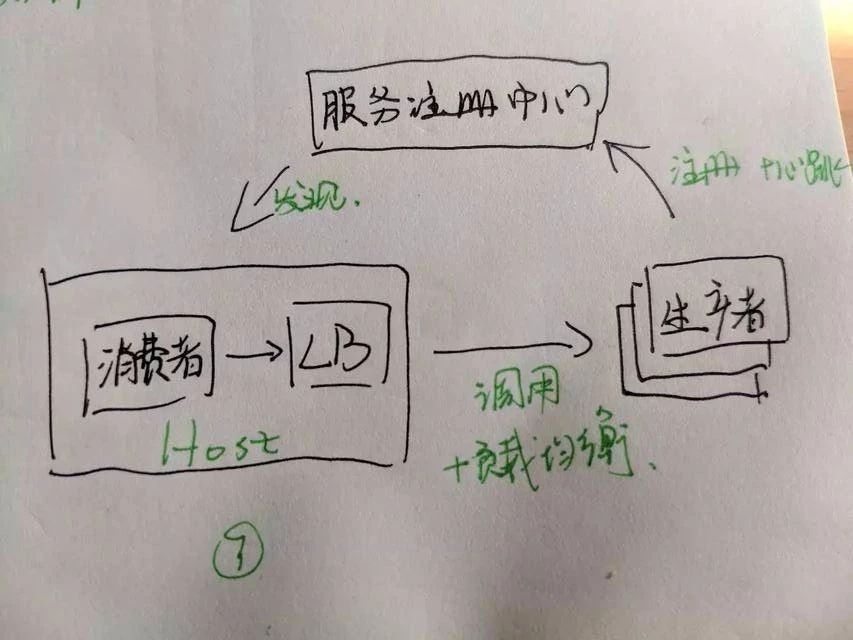
第一种，开发人员开发了程序以后，会找运维配一个域名，服务的话通过 DNS 就能找到我们对应的服务。

缺点是，由于服务没有负载均衡功能，对负载均衡服务，可能会有相当大的性能问题。



第二种，是目前普遍的做法。可以参考 Zuul 网关，每一个服务都通过服务端内置的功能注册到注册中心，服务消费者不断轮询注册中心发现对应的服务，使用内置负载均衡调用服务。

缺点是，对多语言环境不是很好，你需要单独给消费者的客户端开发服务发现和负载均衡功能。当然了，这个方法通常都是用在 Spring Cloud 上的。

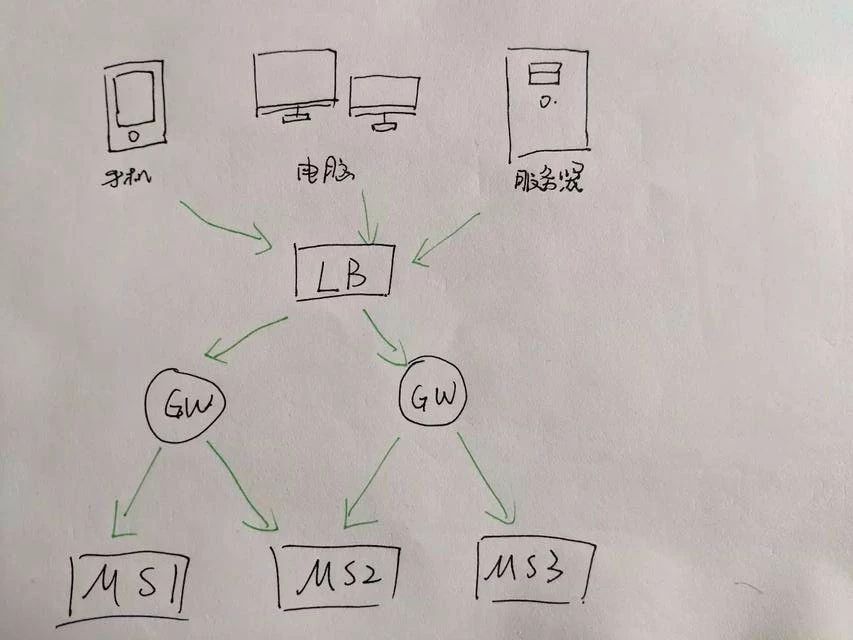


第三种，是将客户端和负载均衡放在同一个主机，而不是同一个进程内。  
这种方法相对第一种第二种方法来说，改善了他们的缺点，但是会极大增加运维成本。

#### 网关

微服务的网关是什么？我们可以联系生活实际想一下。每一个大的公司，都会有一偏属于自己的建筑区，而这建筑区内，都有不少的门卫。如果有外来人员进入公司，会先和门卫打好招呼，才能进去。

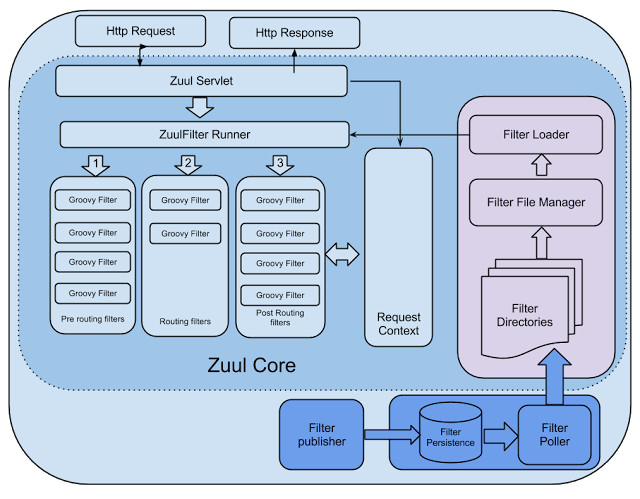
将生活实际联系到微服务上，就不难理解网关的意思了：



网关的作用如下：

* **反向路由：**很多时候，公司不想让外部人员看到我们公司的内部，就需要网关来进行反向路由。即将外部请求转换成内部具体服务调用。
* **安全认证：**网络中会有很多恶意访问，譬如爬虫，譬如黑客攻击，网关维护安全功能。
* **限流熔断：**当请求很多服务不堪重负，会让我们的服务自动关闭，导致不能用服务。限流熔断可以有效的避免这类问题。
* **日志监控：**所有的外面的请求都会经过网关，这样我们就可以使用网关来记录日志信息。
* **灰度发布，蓝绿部署。**是指能够平滑过渡的一种发布方式。在其上可以进行 A/B testing。即让一部分用户继续用产品特性 A，一部分用户开始用产品特性 B，如果用户对 B 没有什么反对意见，那么逐步扩大范围，把所有用户都迁移到 B 上面来。

开源网关 Zuul 架构：



Zuul 网关核心其实是一个 Servlet，所有请求都会经过 Zuul Servlet 传到 ZuulFilter Runner，然后分发到三种过滤器。  
先说说架构图左半部分，分别是使用 Groovy 实现的前置路由过滤器，路由过滤器，后置路由过滤器。  
一般请求都会先经过前置路由过滤器处理，一般的自定义 Java 封装逻辑也会在这里实现。

路由过滤器，实现的是找到对应的微服务进行调用。调用完了，响应回来，会经过后置路由过滤器，通过后置路由过滤器我们可以封装日志审计的处理。  
可以说 Zuul 网关最大的特色就是它的三层过滤器。架构图右半部分，是 Zuul 网关设计的自定义过滤器加载机制。

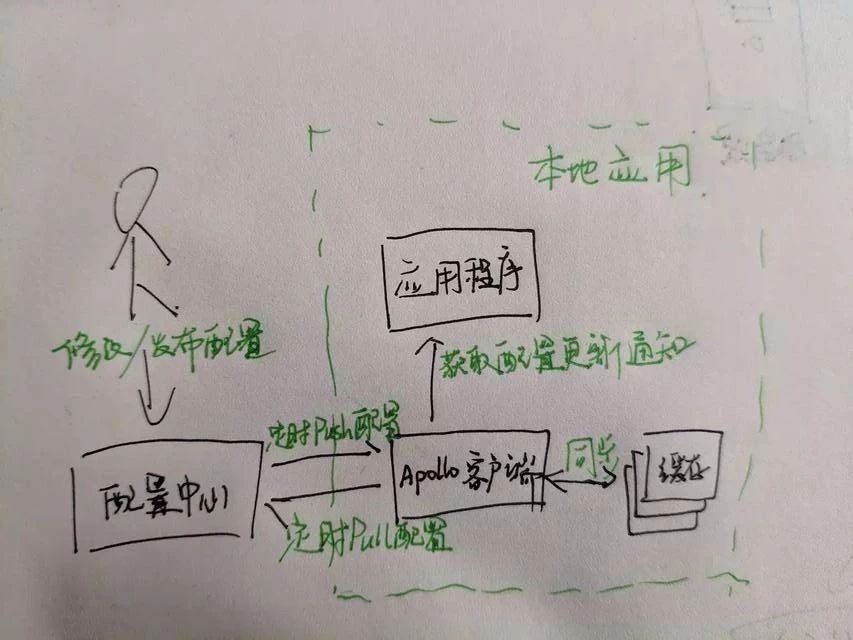
网关内部会有生产者消费者模型，自动的将过滤器脚本发布到 Zuul 网关读取加载运行。

#### 配置中心

以前，开发人员把配置文件放在开发文件里面，这样会有很多隐患。譬如，配置规范不同，无法追溯配置人员。

一旦需要大规模改动配置，改动时间会很长，无法追溯配置人员，从而影响整个产品，后果是我们承担不起的。  
因此就有配置中心这个喽！现在的开源中心有百度配置中心 Disconf，Spring Cloud Config，Apollo。

今天重点说说现在应用质量不错的配置中心，携程开源的阿波罗（Apollo）：



Apollo 的配置中心规模比较大，本地应用会有响应的配置中心客户端，可以定时同步配置中心里的配置。如果配置中心怠机，会使用缓存来进行配置。

#### 通讯方式

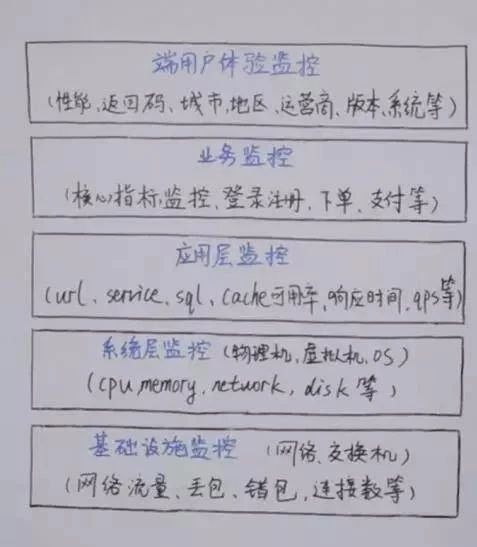
关于通讯方式，一般市面也就是两种远程调用方式，我整理了一个表格：



#### 监控预警

监控预警对于微服务很重要，一个可靠的监控预警体系对微服务运行至关重要。

一般监控分为如下层次：

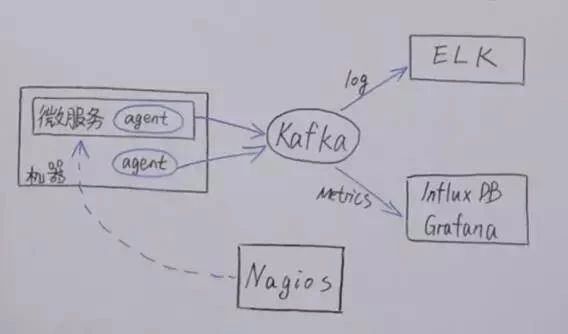


从基础设施到用户端，层层有监控，全方位，多角度，每一个层面都很重要。  
总体来说，微服务可分为 5 个监控点：

* **日志监控**
* **Metrics 监控**
* **健康检查**
* **调用链检查**
* **告警系统**

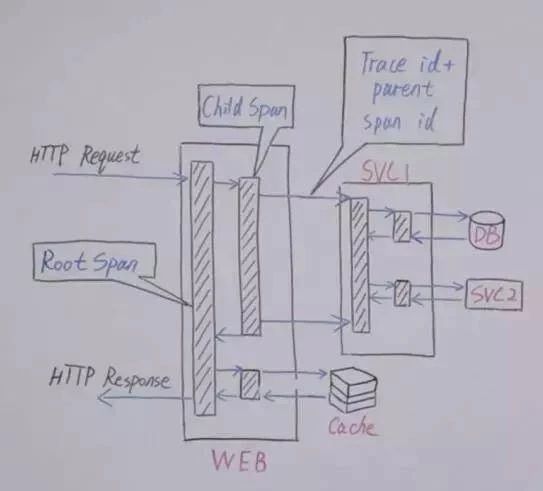
**①监控架构**  
下面的图是大部分公司的一种监控架构图。每一个服务都有一个 Agent，Agent 收集到关键信息，会传到一些 MQ 中，为了解耦。

同时将日志传入 ELK，将 Metrics 传入 InfluxDB 时间序列库。而像 Nagios，可以定期向 Agent 发起信息检查微服务。



**②调用链监控 APM**

很多公司都有调用链监控，就譬如阿里有鹰眼监控，点评的 Cat，大部分调用链监控（没错，我指的 Zipkin）架构是这样的：

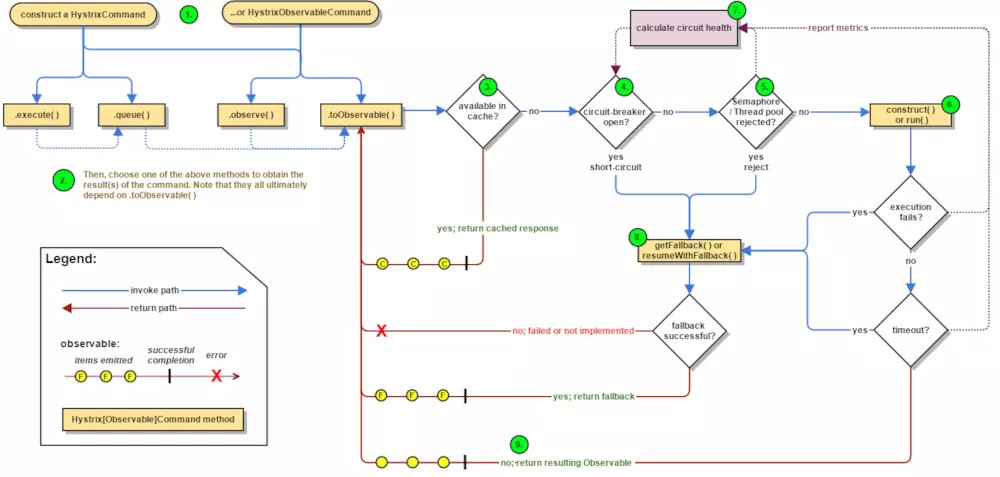


当请求进入 Web 容器的时候，会经过创建 Tracer，连接 Spans（模拟潜在的分布式工作的延迟，该模块还包含在系统网络间传递跟踪上下文信息的工具包，如通过 HTTP Headers）。  
Spans 有一个上下文，其中包含 Tracer 标识符，将其放在表示分布式操作的树的正确位置。  
当我们把图中的各种 Span 放到后端的时候，我们的服务调用链会动态的生成调用链。

下面是一些市场上用的比较多的调用链监控对比：



#### 熔断、隔离、限流、降级 面对巨大的突发流量下，大型公司一般会采用一系列的熔断（系统自动将服务关闭防止让出现的问题最大化）、隔离（将服务和服务隔离，防止一个服务挂了其他服务不能访问）、限流（单位时间内之允许一定数量用户访问）、降级（当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，我们可以将一些不重要或不紧急的服务或任务进行服务的延迟使用或暂停使用）措施。

每一个微服务调用时，都会使用 Hystrix 的 Command 方式（上图的左上角那个），然后使用 Command 同步的，或者是响应式的，或者是异步的，判断电路是否熔断（顺着图从左往右看），如果断路则走降级 Fallback。

如果这个线闭合着，但是线程资源没了，队列满了，则走限流措施（看图的第 5 步）。  
如果走完了，执行成功了，则走 run() 方法，获取 Response，但是这个过程如果出错了，则继续走降级 Fallback。  
同时，看图最上面有一个后缀是 Health 的，这是一个计算整个链路是否健康的组件，每一步操作都被它记录着。

#### 容器与服务编排引擎

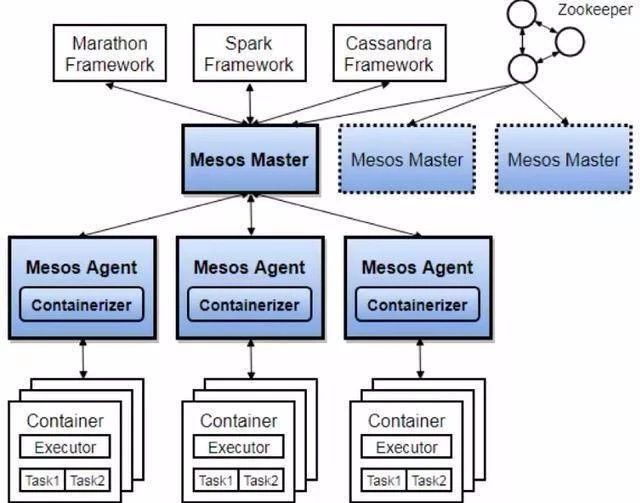
从物理机到虚拟机，从虚拟机到容器；从物理集群到 OpenStack，OpenStack 到 Kubernetes；科技不断的变化，我们的认知也没刷新。  
我们从容器开始说起，它首先是一个相对独立的运行环境，在这一点有点类似于虚拟机，但是不像虚拟机那样彻底。   
虚拟机会将虚拟硬件、内核（即操作系统）以及用户空间打包在新虚拟机当中，虚拟机能够利用“虚拟机管理程序”运行在物理设备之上。  
虚拟机依赖于 Hypervisor，其通常被安装在“裸金属”系统硬件之上，这导致 Hypervisor 在某些方面被认为是一种操作系统。  
一旦 Hypervisor 安装完成， 就可以从系统可用计算资源当中分配虚拟机实例了，每台虚拟机都能够获得唯一的操作系统和负载（应用程序）。  
简言之，虚拟机先需要虚拟一个物理环境，然后构建一个完整的操作系统，再搭建一层 Runtime，然后供应用程序运行。  
对于容器环境来说，不需要安装主机操作系统，直接将容器层（比如 LXC 或 Libcontainer）安装在主机操作系统（通常是 Linux 变种）之上。  
在安装完容器层之后，就可以从系统可用计算资源当中分配容器实例了，并且企业应用可以被部署在容器当中。  
但是，每个容器化应用都会共享相同的操作系统(单个主机操作系统)。容器可以看成一个装好了一组特定应用的虚拟机，它直接利用了宿主机的内核，抽象层比虚拟机更少，更加轻量化，启动速度极快。  
相比于虚拟机，容器拥有更高的资源使用效率，因为它并不需要为每个应用分配单独的操作系统——实例规模更小、创建和迁移速度也更快。这意味着相比于虚拟机，单个操作系统能够承载更多的容器。  
云提供商十分热衷于容器技术，因为在相同的硬件设备当中，可以部署数量更多的容器实例。  
此外，容器易于迁移，但是只能被迁移到具有兼容操作系统内核的其他服务器当中，这样就会给迁移选择带来限制。  
因为容器不像虚拟机那样同样对内核或者虚拟硬件进行打包，所以每套容器都拥有自己的隔离化用户空间，从而使得多套容器能够运行在同一主机系统之上。  
我们可以看到全部操作系统层级的架构都可实现跨容器共享，惟一需要独立构建的就是二进制文件与库。

正因为如此，容器才拥有极为出色的轻量化特性。我们最常用的容器是 Docker。

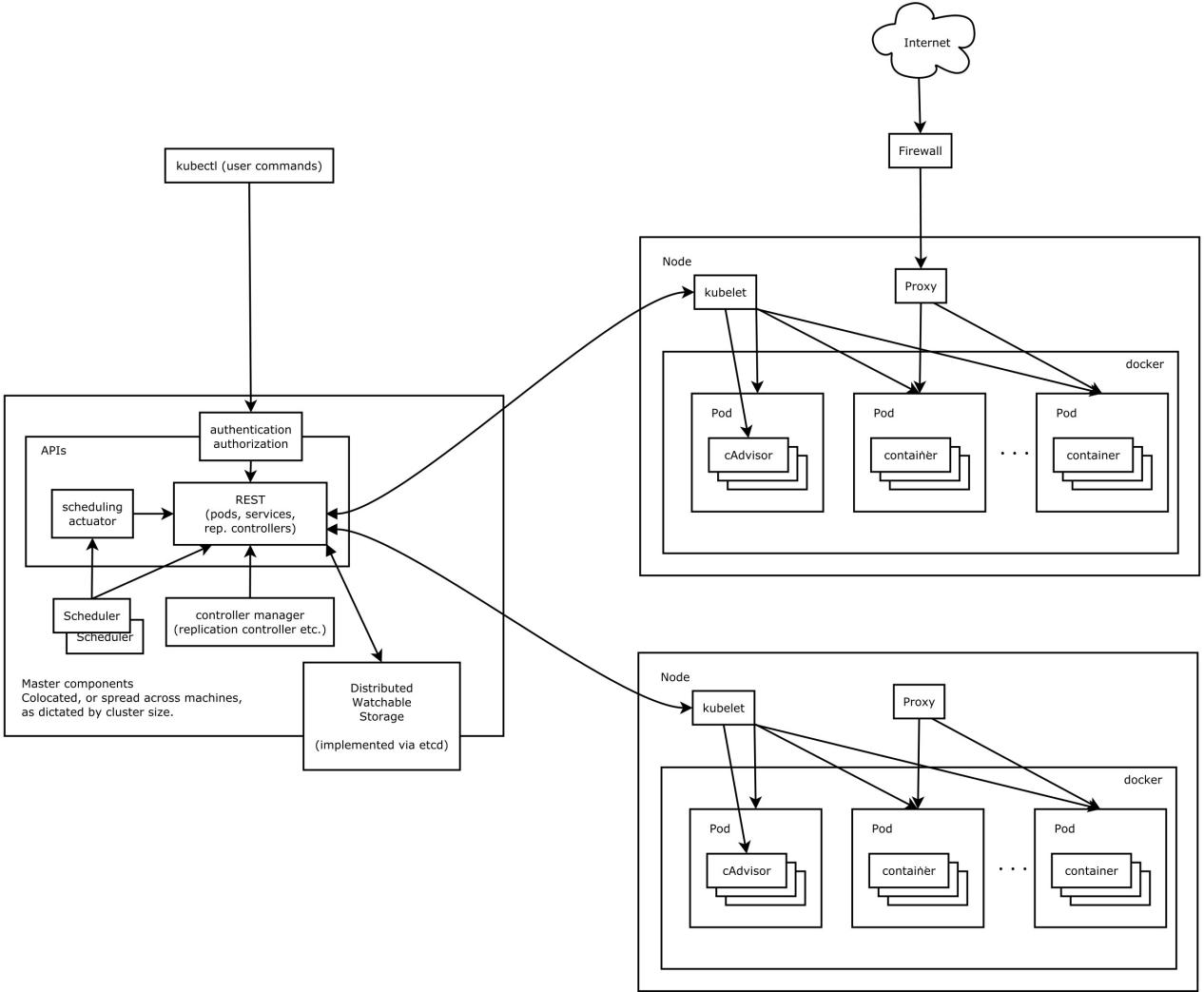
**①容器编排**  
过去虚拟机可以通过云平台 OpenStack 管理虚拟化，容器时代如何管理容器呢？这就要看看容器编排引擎了。

**Apache Mesos：**Mesos 是基于 Master，Slave 架构，框架决定如何利用资源，Master 负责管理机器，Slave 会定期的将机器情况报告给 Master，Master 再将信息给框架。Master 是高可用的，因为 ZK，也有 Leader 的存在。

下面是架构图：



**Kubernetes：**Kubernetes 是最近十分火热的开源容器编排引擎：



Kubernetes 设计理念和功能其实就是一个类似 Linux 的分层架构，先说说每一个 Kubernetes 节点内部，kubelet 管理全局全局 pod，而每一个 pod 承载着一个或多个容器，kube-proxy 负责网络代理和负载均衡。  
Kubernetes 节点外部，则是对应的控制管理服务器，负责统一管理各个节点调度分配与运行。  
**②服务网格化**

关于服务网络化，后面会更加深入的为大家进行讲解。