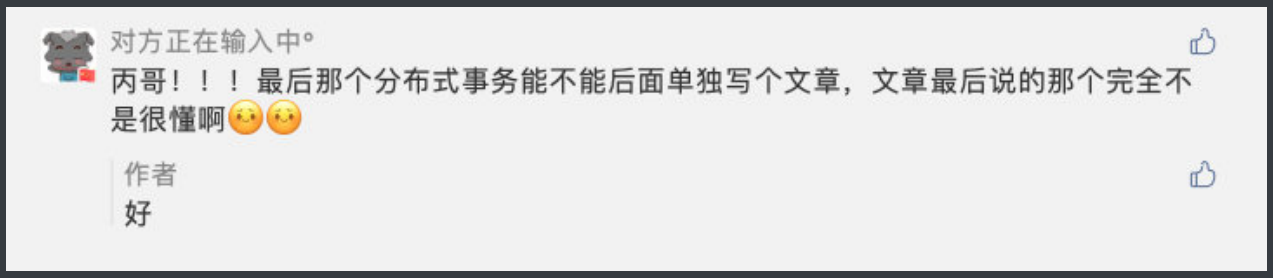
> 点赞再看，养成习惯，微信搜一搜【**三太子敖丙**】关注这个文绉绉的程序员。
>
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言

之前写了一篇秒杀系统的文章，最后给自己埋了分布式事务的坑，然后很多读者就要求我去写分布式事务，那作为程序员届的暖男，我一向是有求必应的，就算是不睡觉我都要写给你们看的！



因为分布式事务是：分布式 + 事务 = 分布式事务。

理所当然的要先谈谈分布式，而分布式又得谈谈这个概念是如何演进得来的，因此作为创作鬼才的我，就先来讲讲架构的演进，什么叫分布式？什么是集群？SOA、微服务这两个东西的关系和区别，下篇再讲分布式事务。

因为我发现我读者大多都是学生或者跟我一样刚毕业不久，那一直听分布式估计都听腻了，估计都还不知道分布式的一些细节和架构演进路线吧。

先来说个题外话，我一直追崇一个技术点不仅要理解其原理，还需要知道这个技术点解决了哪些痛点，这些痛点又是由什么引发的？这个技术还未诞生的时候是如何解决的？

也就是整体的演进过程，历史脉络。

比如为何需要HTTP？HTTP0.9为何需要演进到HTTP1.0？ 进而又向1.1、2演进到最新要将Google开发的基于UDP的QUIC协议应用到HTTP3中。

搞懂这些来龙去脉相信你不仅仅对HTTP会有更深层次的理解，对网络也会有更加深刻的认识。当然也不是说一样东西一来就得全盘理清，有些东西还是比较复杂的，只是说我们要往这个方向去学。

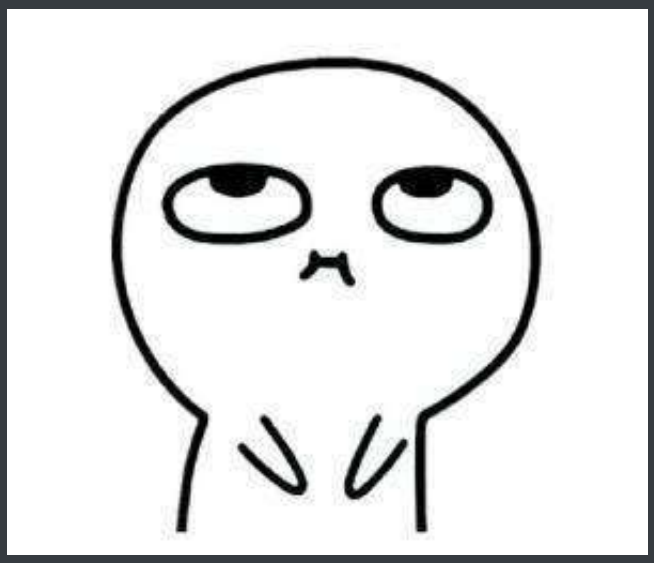
这也是我一直觉得很重要的一个学习思想，知其然知其所以然，会让大家更好的理解很多东西。

一般架构的演进过程

好了，让我们回到今天的主题，咱们先来看看一般架构的演进过程。

为什么说一般呢？举个例子，比如某线下龙头企业，现在想开展网上相关业务，那么有大批线下忠实用户的支撑，并且自身钱包鼓鼓，人力财力都不缺。

那么线上的软件架构就得考虑清楚了，几乎不可能按照初始阶段来，当然也不能用力过猛，实际得靠架构师以及团队实力进行权衡。



单体应用架构

什么是单体应用？简单的说就是不管啥功能都往一个应用里写，比如电商系统。用户功能、商品功能、订单功能等等，都往一个应用里写。

这有什么好处?

在项目初期，小公司人力财力不足，急于拓宽市场，这种单体应用架构**简单粗暴**，将所有的功能都打包在一个的应用中，直接部署。

本地开发调试方便，直接起一个项目，调试也是在一个进程内，没有冗长跨进程的调用链，出错可快速定位。

本地的函数调用，没有网络调用的开销。

线上出了问题回滚这一个应用即可（这一点其实在某种程度上看是优点，某种程度上看是缺点）。

总结的说就是开发、测试、部署方便，本地调用对于远程调用性能较好。

有什么坏处?

系统耦合性高，导致开发效率低下。

一开始可能模块结构还很清晰，随着需求日益增长，不断的添加新功能，代码量巨增，模块之间的边界开始模糊，调用关系开始混乱，整体的代码质量非常依赖个人水平。

假使某个同事水平较差，实现的代码冗余，逻辑混乱，这时候要在上面添加新功能或者修改老功能其实是一件很困难的事情，**你不能保证你修改的功能模块不会影响到其他功能**。

而且代码会有**“破窗效应”**（这里其实不仅仅是单体架构，对于所有架构来说都是如此，只是单体应用更大的庞大，业务界限不清晰，因此这种问题更容易被放大）。

有些人看到这就可能会说，这上面还说开发方便，这就又效率低下了？是的，过犹不及。

再比如一个新需求上线例如短信相关的，并且订单也做了一些改造，但是短信功能出了 bug，**需要回滚的是整个应用**，订单模块冤啊，陪着一起回滚。

我在老东家做电商活动，我们上线一个需求可能涉及6、7个服务，回滚也得全部回滚，而且都是负载均衡的，那机器可能就是上百台了。

语言单一，不能根据场景选择更加合适的语言，例如要实现数据分析，应用的语言是 Java，那么就不能利用到 Python 丰富的类库。

系统的整体可靠性不高，什么意思呢？还是拿短信功能说事，新上的短信功能写的有 bug，不管是堆栈溢出还是死循环等等，核心的订单等功能都会受到影响。 因此你上线的功能有问题影响的不仅仅是这个功能模块，可能是整体系统的瘫痪。

系统不易于扩展部署，假设你发现你们的商品查询的流量特别大，顶不住就得加机器。因为是单体应用所以为了商品查询这一个功能，你需要在新加的机器上部署这一个应用，没法单独为这一个功能做定制化部署，对硬件资源有一定的浪费。

总结的说缺点就是随着需求不断增长，代码结构日益复杂，各功能掺杂在一起，系统耦合性高，模块之间边界维护非常依赖开发者的个人水平。

模块之间经常会有公共功能难以划分清楚，添加或修改功能困难，不确定是否会影响到其他模块，所有功能都在一个进程内，某个功能出问题可能影响的就是整个应用，而且无法根据特点场景选择更加合适的语言去实现功能，技术单一。

随着用户的增长，无法做到热点功能单独扩展，只能整体应用部署。

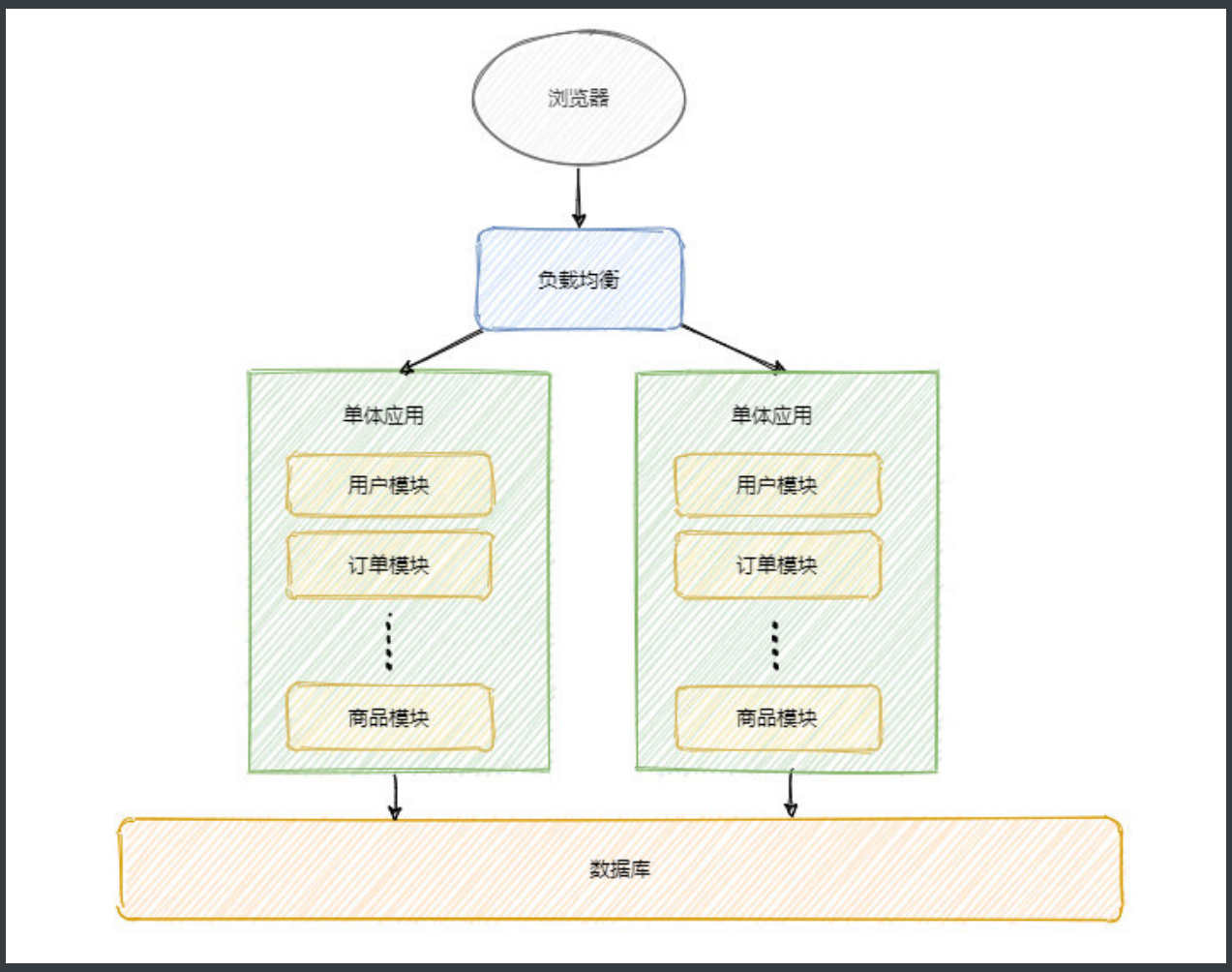
至此我们已经明白了单体应用架构的优缺点，可以看出初始阶段单体应用优点突出，随着需求和用户的增长渐渐的单体应用顶不住，缺点在不断的放大。

也就是说你的产品需要发展到一定的阶段，单体应用才会顶不住。在这之前单体应用是你的最佳选择。

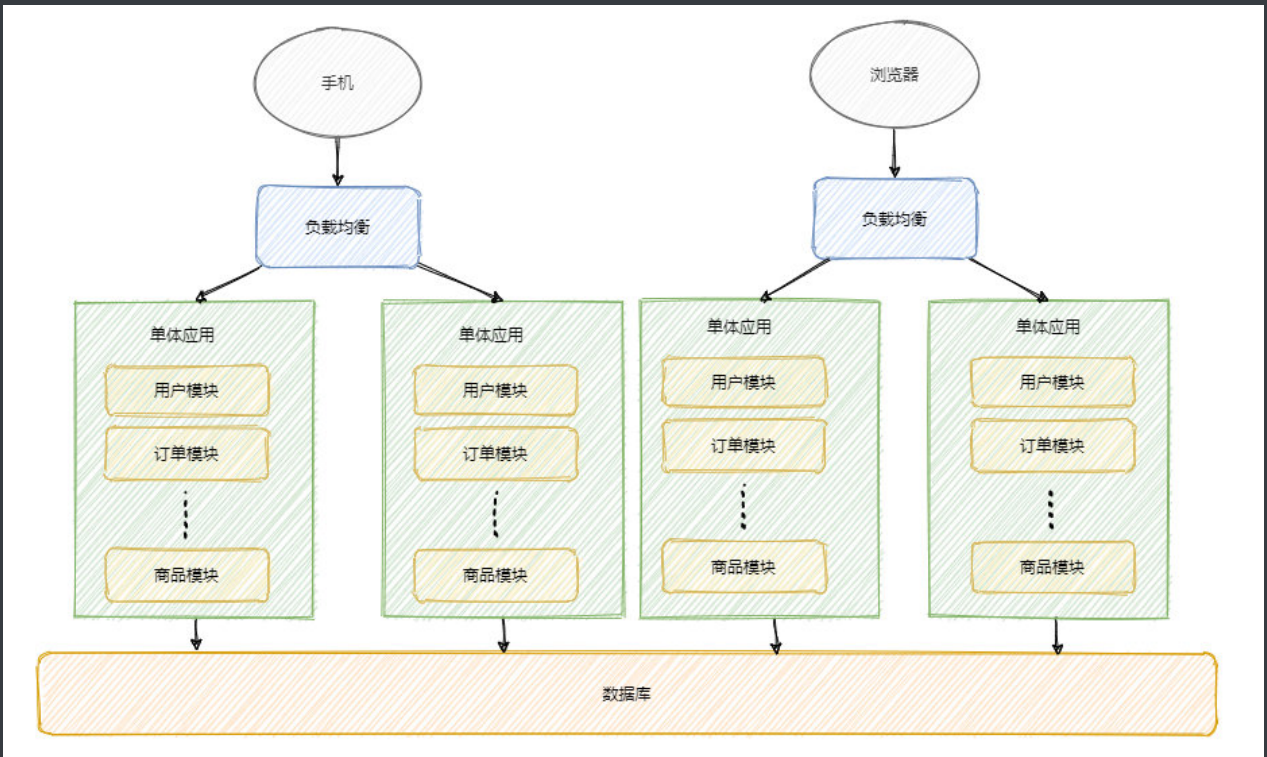
你要是说我的产品肯定顶的，所以一开始就大刀阔斧的设计，单体应用太 low 坚决不用。

可以的，秀出你的花样，你有你的 Young。

我们再来看下单体应用一般的架构图，注意单体应用不是真的就线上部署一个，好歹得两台，互相 backup 下，不能太虎一台顶。



又过了一段时间，你发现你们还需要开发手机版。于是你们的架构又变成下图所示的样子。



没错，为了让每个端不会相互影响，粗暴的拷贝现有的应用，稍加修改即可为手机版和小程序提供服务，你会发现很多功能代码都是重复的。这时候来个需求你改的就是多份代码了。

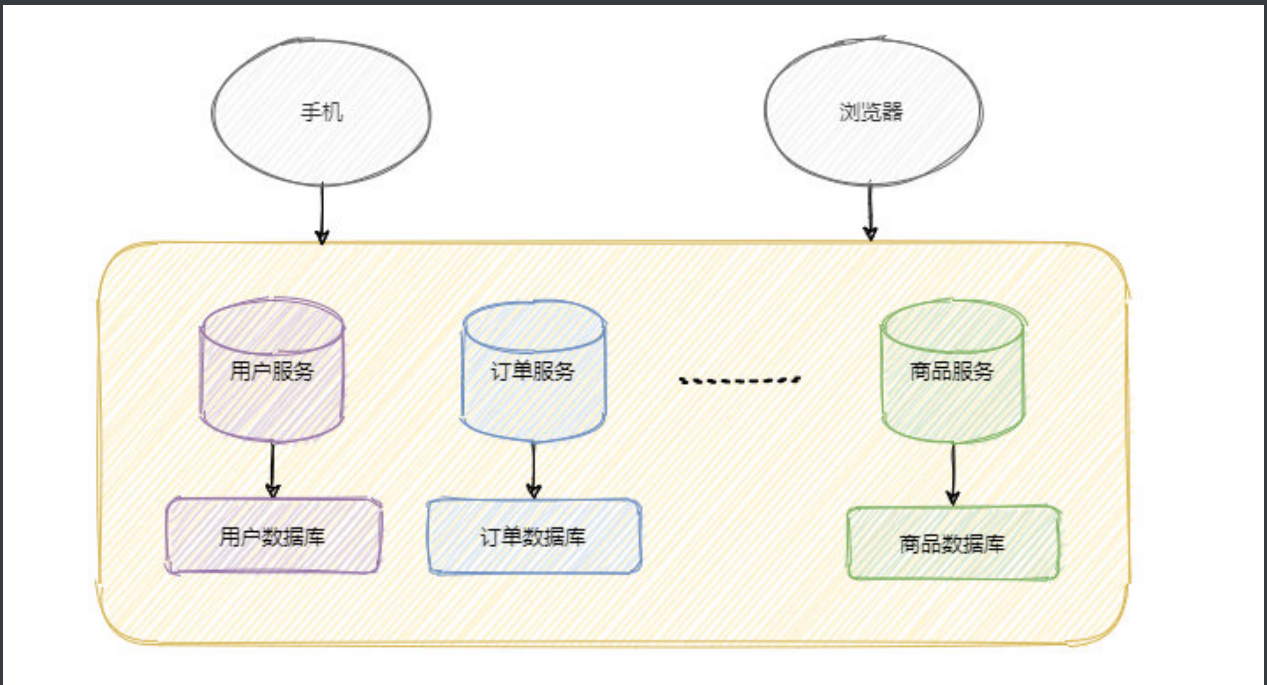
微服务架构

又过了段时间你已经强烈感受到单体应用所带来的痛点，这证明你的产品发展的不错，一开始肯定会忍，继续忍，终有一天你会一拍桌子！来开个会咱们得还债了。

理所当然的你会根据不同业务拆分出不同的服务，并且会整理出当前公共的功能变成一个公共服务，每个服务独立部署，独立运行，代码进行了物理隔离，一个小团队维护一个服务或多个服务。

并且一般而言服务化了，数据库也会拆分出来每个服务维护自己的数据库，数据库之间的数据通过接口传递，而不再是直接访问。

此时你的系统变成了下图所示的样子。



那现在解决了单体应用什么问题?

系统的耦合度降低，模块之间的边界清晰，都按业务物理隔离了。

在一定的措施下（下文会提到），**系统整体可靠性变高**。

技术选型丰富，不同的服务可以利用不同的技术或语言实现，例如数据分析服务可以用 Python 实现，一些底层的服务团队说我要用 GO，那就用 GO 呗。

可根据服务扩展部署，商品服务访问量特大，那我们就单单给商品服务扩容，增加机器，其他服务照旧。

这就是微服务了。

好像微服务架构解决了单体应用的所有痛点啊？别急上面只是微服务的一部分，真正的微服务架构还需要包含很多东西，**微服务是解决的单体应用的痛点，但是也引入了新的痛点！**

服务化了之后上线某个需求，如果是单个服务内的不影响其它服务的你会感到很舒服，如果这个改动是接口层面的改动，涉及到多个服务，你就会觉得有点难受了。

上线之前需要定制好服务上线的顺序，定制好每个服务的回滚计划，涉及到每个团队之间的合作，上线不再是一个简单的打包、部署的过程。

出了问题也不是一个简单回滚的过程，而可能是各个团队分别回滚各自的服务。如果你自己的服务出了问题你会很焦虑，别的团队都等着。如果别的团队出了问题你也焦虑，怎么还没好啊。

你还会发现本地开发如果依赖别的服务会异常的难受，特别是你依赖的服务还依赖别人的服务，调试、测试将变的复杂。

而且你会发现调用链路变长，调用增加了网络的开销，性能变差。而且出错难以定位问题来源。因此你需要**引入分布式链路追踪服务来定位问题**。

还需要**引入ELK来方便日志的查看，分析问题**。

为了能够动态扩容，你的服务需要自动注册且能被自动发现，因此**需要个注册中心**。

网络之间的调用较为不可靠，因此还需要让**调用有重试机制**，防止其他服务出 bug 或其他原因疯狂调用你的服务，还需要有**限流措施**。为了防止一个服务挂了导致整体的雪崩需要有**熔断措施**。

为了在特殊时候例如大促的时候让出硬件资源给核心功能，还需要有**降级策略**。

上面说的重试、限流、熔断、降级就是上文提到的一定措施下，可靠性变高。

而且每个服务都需要配置，因此还得有个**配置中心**，来做统一管理。

服务太多了，调用关系复杂为了对调用者更加的友好，并且还需要对调用进行权限等控制，因此**需要有个网关**，对外暴露统一的接口，当然想限流什么的可以在网关实现。

当然**整体的监控是必不可少的**，对所有的服务都需要做到全面的监控。

其他的**还有啥DevOps、容器等等**。

可以看到服务化之后需要引入太多太多的东西，有人可能说你这也就才几个服务啊，我上面的服务其实可以再细分。

例如商品的修改和写入动作相比较于商品的浏览访问量肯定少很多，那我就将商品的浏览再剥离出来单独做一个服务，这样便于扩容。

这用户量上来，访问量增加这样的服务剥离你会发现越来越多，服务的数量到时候就上来了，而且需求也会不断增加，推荐服务啊、搜索服务啊等等很多很多，只是为了简便都没列出来。

上面提到的那些服务于微服务的组件也得部署，也得保证可靠性...你看这系统就越来越复杂了，所以**服务化之后解耦了业务，却又融入了非业务相关的东西**。

不过服务化其实是一个自然的结果，就像我们平时去的办事大厅会根据服务类别划分成不同的服务窗口，为了办一件事情我能可能需要在各个窗口之间来回走动，这对应的不就是调用链路长嘛？走动的耗时等于我们调用的网络开销。

所以说微服务架构是发展到一个阶段自然而然的演进产物，早在微服务这个概念被提出之前，很多公司就已经是这样干的了。

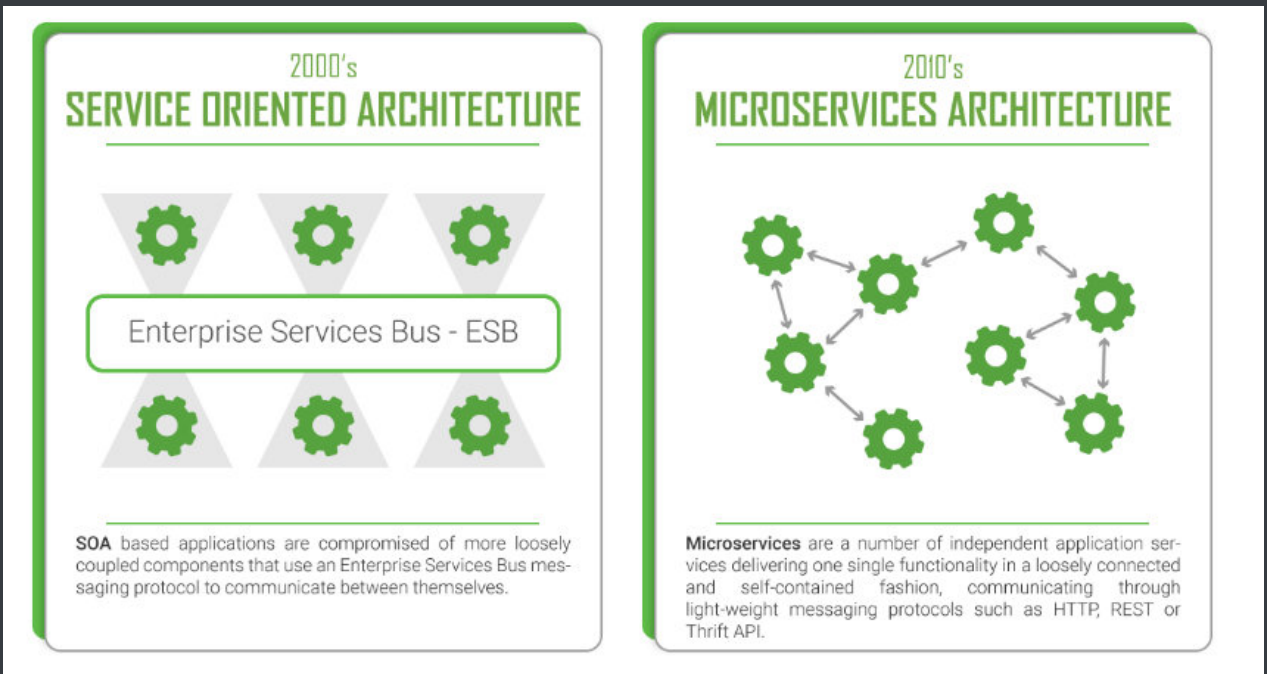
我上面提到的其实是微服务1.0架构，而微服务2.0就是为了将非业务功能剥离出来而提出的，将服务治理的功能放在 SideCar 即边车上，使得开发者专注于应用业务的开发，进而演进出 Service Mesh 即服务网格架构。

SOA和微服务

谈到微服务你会发现 SOA 这个名词经常伴随着出现。

关于SOA和微服务我查阅了很多资料，不过对于这两个名词的解释都各执一词，没有一个统一的答案。今天我就说说我的理解，抛砖引玉，有纰漏之处，敬请指正。

SOA，全称 `Service-Oriented Architecture` 即面向服务的架构。说到SOA就离不开 ESB，全称 `Enterprise Service Bus` 。SOA和微服务一样都是面向服务的。



可以看到 SOA 架构通过企业服务总线进行交互，也就是说中心化，需要按照总线的标准进行开发改造，而微服务是去中心化的。

其实我们可以抓到关键字企业，**SOA 我认为是企业级别的面向服务概念，而微服务是应用级别的概念。**

两种都是面向服务，只是 **SOA 注重的是企业资源的重复利用，把企业的各个应用通过 ESB 进行整合。**

而微服务注重的是应用级别的服务划分，使得应用内服务边界清晰，易扩展。

这两者其实是两个方向的面向服务，互不冲突。还能是包容的结构，如下图所示

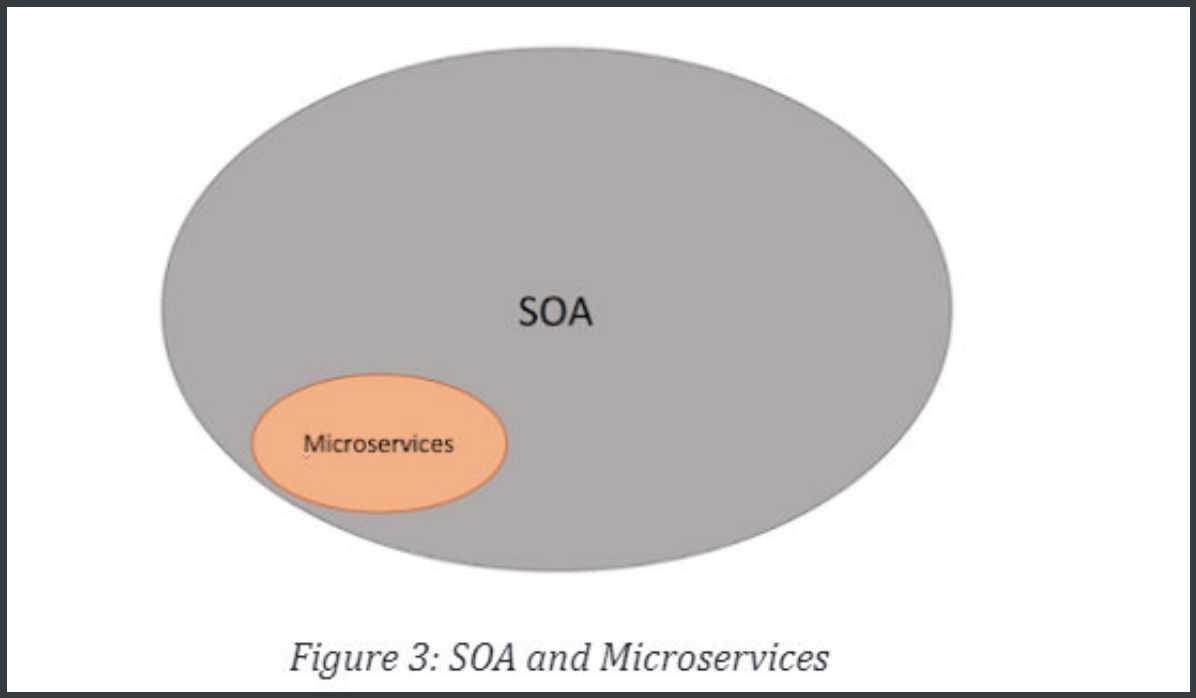


Figure 3: SOA and Microservices

分布式和集群

分布式可以认为是通过网络连接多个组件而形成的系统。

广义上说前后分离的应用就能算分布式，前端的 js 代码在浏览器跑着，后端的代码在服务器跑着，两种不同的组件合力对外提供服务构成分布式。

而我们**常提到的分布式是狭义上的，指代不同的组件通过协作构成的系统**。

而**集群常指的同一个组件多实例而构成逻辑上的整体**。

这两个概念不冲突，分布式系统里面可以包含集群，像我们的商品服务就可以是集群部署。

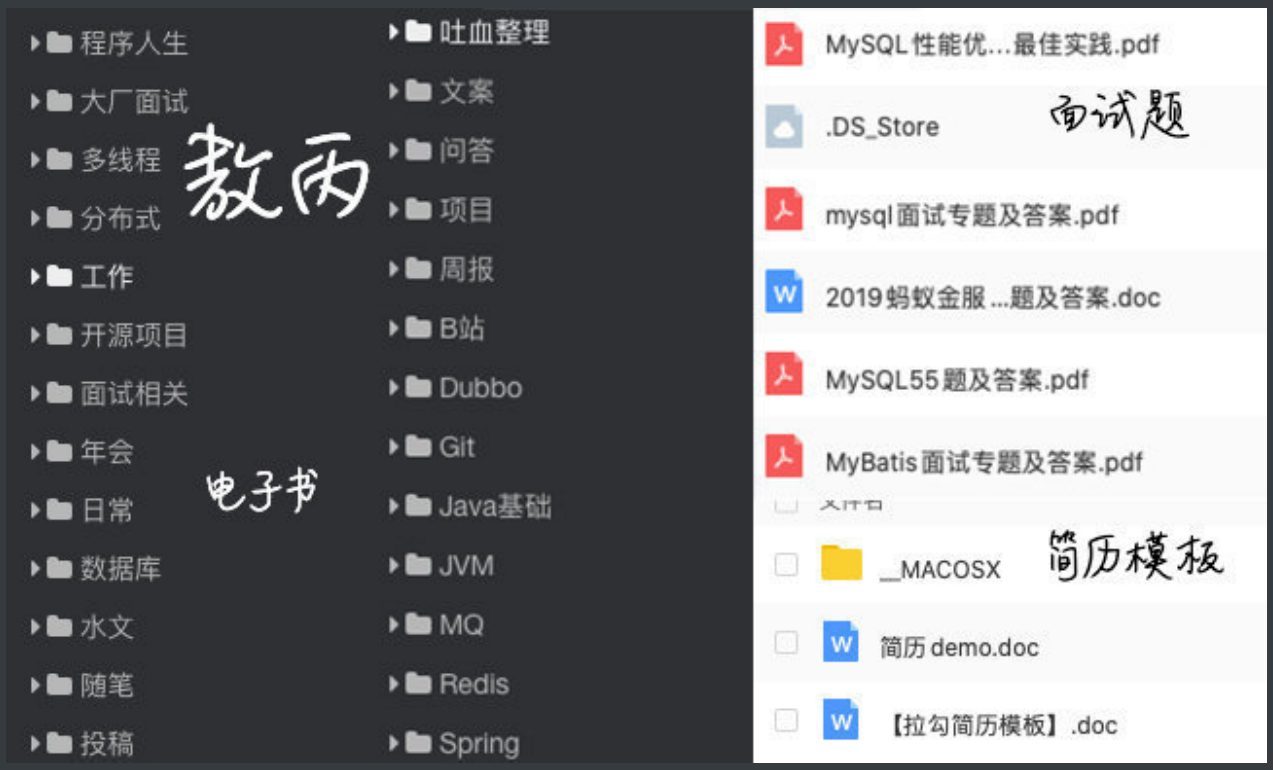
絮叨

今天主要简述了下架构的演进，单体应用的优缺点以及微服务的优缺点。

再谈了谈SOA 和微服务之间的区别，以及分布式和集群的区别。

说了这么多，也不知道有没有说清楚，个人能力有限，如果有纰漏，敬请指正，分布式事务也在疯狂爆肝中，我们下篇文章见。

另外，敖丙把自己的面试文章整理成了一本电子书，共 1630页！目录如下，还有我复习时总结的面试题以及简历模板



现在免费送给大家，在我的公众号**三太子敖丙**回复 **【888】** 即可获取。

我是敖丙，你知道的越多，你不知道的越多，我们下期见！

人才们的 **【三连】** 就是敖丙创作的最大动力，如果本篇博客有任何错误和建议，欢迎人才们留言！

> 文章持续更新，可以微信搜一搜「 **三太子敖丙** 」第一时间阅读，回复【**资料**】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。