单体架构和微服务架构到底哪个好？

腾讯程序员 腾讯技术工程 2024年11月18日 18:03 广东

作者：jeanwu

> 单体和微服务谁是毒瘤？单体、分布式、微服务、SOA到底是什么关系？我的系统该用什么架构？最近终于下定决心研究这个问题并且有所收获，欢迎一起讨论。

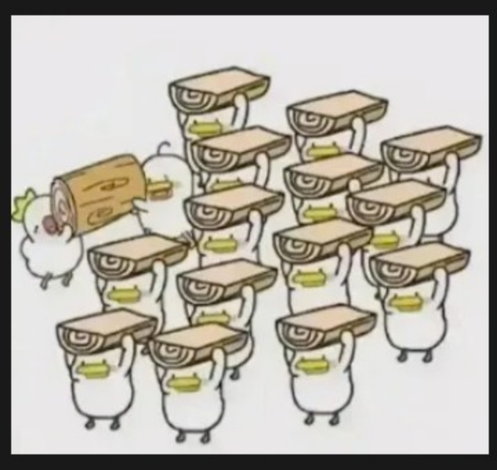
一、架构的发展历程

我坚定的认为要深刻的理解一项技术光靠网上一两张按照各项维度对比的表格是不够的，而是要了解这些技术出现的历史背景：他们的出现到底是解决了什么问题，又带来了什么新的问题，最后又因何而被淘汰。下面这部分内容参考《凤凰架构》以及Martin Fowle等人一些文章进行整理，一起来看下历史的浪潮是如何推动架构的演进。

1.1、原始分布式时代

首先介绍的是竟然是“分布式”而不是“单体”这有些反常识，然而事实上分布式确实出现的比单体早，“单体”这个名称是在微服务开始流行之后“事后追认”所形成的概念，**在单体出现之前分布式早已流行**，并且成果斐然。

1971年Intel公司设计了世界上第一台微型计算机MCS-4，开创了微型计算机的新时代，计算机逐步从专业的科研设备转变为企业的生产设备。但是微型计算机用于商业生产面临一个非常大的问题：**计算机硬件有限的运算处理能力，直接限制在单台计算机上信息系统软件能够达到的最大规模。**如果你生在这个时代，相信也能自然而然想到这一问题的解决思路：“人多力量大”、”众人拾柴火焰高“，朴素的真理适用于任何地方，当时高校、研究机构、企等不约而同的开始探索**“使用多台计算机共同协作来支撑同一套软件系统”**的方案，并取得了一系的成果。谈分布式必然绕不开远程调用，所以下面以远程调用为例谈一下这一时期的探索成果有什么局限性，又产生了哪些深渊的影响。



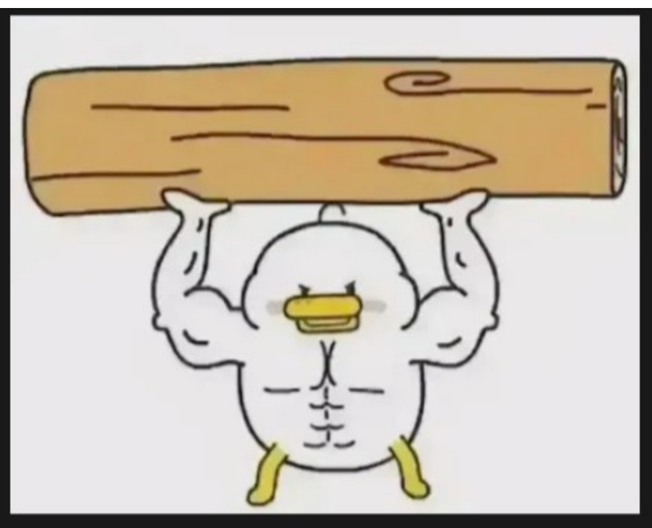
通过多台计算机分布式协同支撑提升系统规模

大家应该了解“UNIX系统的版本战争”这一故事，为了避免相同的“战争”重演，负责制定 UNIX 系统技术标准的开放软件基金会与主流计算机厂商共同制订了DCE/RPC这一影响深远的远程服务调用规范，它也是公认的现代 RPC 鼻祖之一。因为开放软件基金会本身就负责UNIX 系统技术标准的制定，在这个背景下，DCE/RPC带着浓厚的UNIX**“简单优先原则”**的设计哲学，预设**分布式环境中的服务调用、资源访问等操作尽可能透明，使开发人员不必过于关注他们访问的方法或资源是位于本地还是远程。**

然而这个过于理想化的目标在当时面临着太多的技术难题，“远程调用”与“本地调用”相比复杂程度完全不可同日而语：服务发现、负载均衡、熔断、隔离、降级、认证、授权等一系列的问题亟待解决。令人敬佩的是面对重重困难，DCE从无到有构建了大量的协议来解决这些问题，真的做到了相对“透明”，但是分布式还有一个致命的问题——网络所带来的性能问题。

我们来推演一下：硬件性能不足——>采用分布式服务——>分布式的远程调用导致性能降低（**与解决硬件性能不足的初心相悖**）——>通过合并多个请求等方式刻意（**开发人员需要意识到自己在写分布式程序，与DCE透明简单相悖**）降低网络性能损耗——>人的能力成为软件规模的约束。这时候分布式从结果来看并不成功，设计向性能妥协让简单透明成为一句空话。

当我们玩游戏打BOSS时，喊人解决不了问题还有另外个方法——氪金，20世纪80年代摩尔定律稳定发挥，微型计算机的性能以每两年即增长一倍的惊人速度提升，既然分布式充满了矛盾与妥协，那就加钱换更好的机器，单体时代到来。



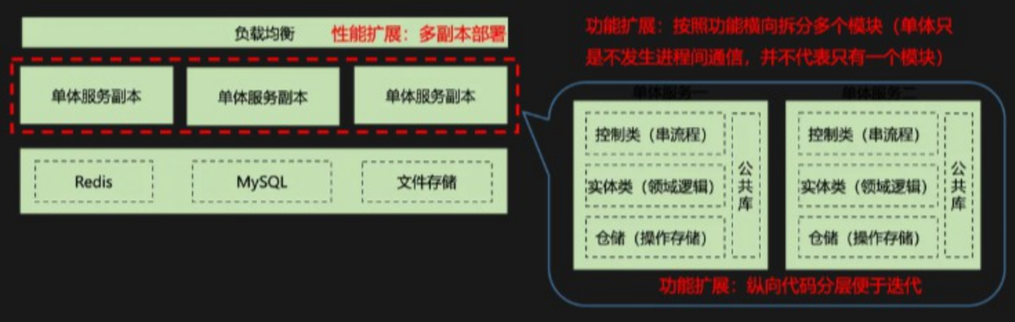
通过提升单台计算机性能提升系统规模

1.2、单体架构时代

在微服务出现之前“单体”都不认为是架构，因为他太“简单”、太“自然”了，以至于我想找到一本关于单体最佳实践的书籍都没有找到。现在很多书籍把单体当做“毒瘤”，甚至会出现微服务比单体先进的观点，然而事实上真的如此吗？

首先，我们先要明确单体是什么：**“单体”只是表明系统中主要的过程调用都是进程内通信，不会发生进程间通信，仅此而已。**那么进程内调用是罪恶吗？是毒瘤吗？那肯定不是的。现实中不会有人对你说：你这个"Hello, World!"程序不能用单体，因为单体架构是毒瘤。

有个“单体不便于扩展“的论调非常流行，我们着重讨论下这个观点是否准确。我们从性能扩展和功能扩展两个方便说。先说性能扩展：这太简单了，把一个服务部署多个副本，前面加一个负载均衡服务器来均分流量，这是不是就实现了扩展？再说功能扩展：纵向上我从来没见到哪个大型系统代码是不分层的，用过Spring都知道写一个服务基本上默认按照MVC模式分层以便于扩展；而横向上我们也可以从功能等维度拆分为不同模块（模块之间不进行通信或者进行少量的通信，**注意：拆分模块不代表不是单体服务，单体服务也不表示系统只有一个模块**），各模块完全可以独立迭代。从这个角度来看单体服务在“可扩展”这一点上也不输与微服务，甚至在开发、调试等方面更优。当前单体服务也有其局限性，比如有个C++实现的系统要实现部分AI功能，明显Python更有优势，C++中执行Python虽然可以实现，但是显然不如微服务独立一个Python模块来的优雅。**现在回来看“单体服务不便于扩展”这个观点并不完全正确。**



单体也有良好的扩展性

“罪恶的单体”是“大型的单体系统”，而“大单体服务”的主要罪恶在于隔离性。举一个例子，不小心写了一个BUG会导致程序崩溃，如果是微服务只会导致一个模块崩溃，影响的范围也仅仅是依赖这个模块的功能，可是如果是单体服务，所有代码都共享着同一个进程空间，某一处崩溃直接导致整个进程崩溃，那影响的范围就大了。到这里我们可以简单的说：**单体服务在同一进程更简单、高效，其代价是损失了隔离能力以及技术扩展性。**至于这两点谁轻谁重不可一概而论，要看你的系统到底是一个小卖店还是一个大超市了。

既然单体和微服务各有优劣，为什么后来微服务潮流滚滚而来？互联网是不断发展的，随着微型计算机的普及，软件系统的功能越来越多，对应的开发团队规模也越来越大，我们逐步进入了“大兵团作战”时代，这时候“墨菲定律”+“黑天鹅事件”对系统可用性影响越来越大。

构建一个大规模但依然可靠的软件系统非常难。根据墨菲定律可能发生的事（BUG）就一定会发生，再叠加不可预测的“黑天鹅事件”，随着研发团队的规模不断扩大，系统不可用的概率会被无限放大，举个极端点的例子：假如一个公司有10万人，每个人写出系统的可用性都能达到99.999%，可是当他们共同协作写一个单体服务时，那么系统的可用性就是99.999%的10万次方，约等于36.8%，这个系统简直不可用，更何况人是不可靠的，这时候单体服务隔离性差的缺点凸显。微服务把**构筑可靠系统观点从“追求尽量不出错”转变为“出错是必然”**，通过拆分服务以减少异常的影响范围，关于微服务为什么可以提升系统的可用性可以用一个例子帮助理解：人体系统由一个个的不可靠的细胞（微服务）组成，大多数情况下一个或者一群微服务（细胞）的崩溃并不会影响我们的生命，我们的人体系统就是用不可靠部件构造的可靠的系统例子。

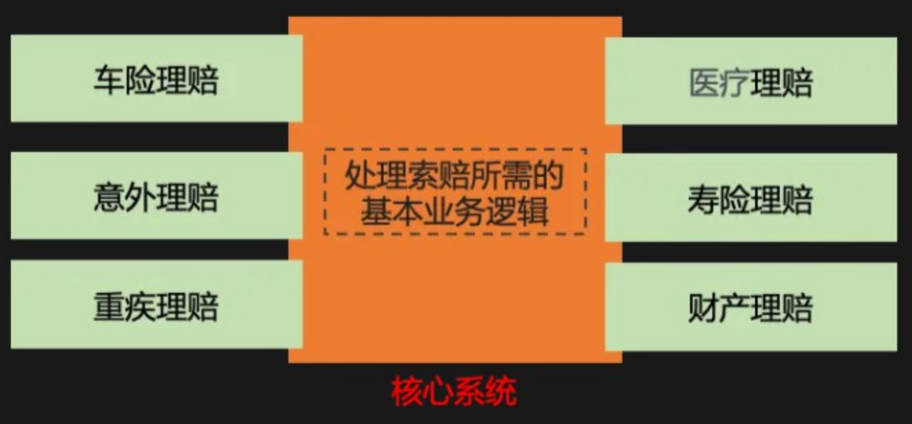
随着系统规模扩大单体拆分势不可挡，关于大单体如何拆分前辈们也进行了大量的探索，其中比较有代表性的有三种：

烟囱式架构：其实这里并没有什么高深的理论，无非就是随着系统的迭代，把一个系统拆分多个，拆分后的系统相互独立没有业务交互，因此使用这种架构系统又称为孤岛式信息系统，问题是这种情况真的存在吗，如果存在大概率一开始就设计为两个系统，既然是两个系统也就不会放在一个单体里面了，放在一起那必然是因为这两个系统共享了某些资源的，例如存储、权限、账号等等。

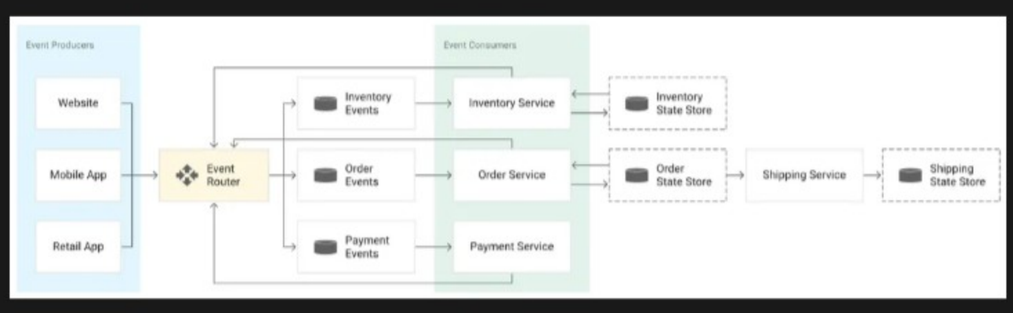
微内核架构：烟囱式架构在实际很少存在，因为拆分后的系统与系统大概率是有资源共享的，既然如此就大方的把数据、资源、公共服务集中到一块成为一个被所有业务系统共同依赖的核心，具体的业务系统以插件模块（Plug-in Modules）的形式存在，注意微内核架构核心在于微，**核心只负责通用业务逻辑，不包括针对特殊情况、特殊规则或复杂条件处理的自定义代码**，不是插件化实现就一定是微内核架构，比如MySQL只是存储引擎做成了插件，而MySQL的Server层是核心功能，不能算微内核架构。

这种架构至今仍然存在，Eclipse IDE就是一个典型代表，下载基本的Eclipse产品只会提供一个简单的编辑器，一旦添加插件它就能变成一个高度可定制和实用的软件开发工具。当然微内核架构也有其局限性，由于无法预先知道有哪些插件，因此实现一个插件时并不能默认的可以与另一个插件交互，插件只能和内核交互。

目前保险行业也常用微内核架构，由于不同地区、时间、事件索赔规则非常复杂，理赔规则会发展成一个复杂的大泥球，更改一条规则会影响其他规则，开发测试极其麻烦。使用微内核架构模式可以解决其中许多问题：



事件驱动架构：事件驱动解决了微内核架构插件不能相互交互的问题，子系统之间建立一套事件队列管道，子系统可以发布消息到队列，也可以订阅队列中的消息。系统之间完全解耦，好的架构历久弥新，迄今为止电商系统依然有事件驱动架构的影子：下单、支付、发货、结算，完全是由事件驱动。



图源Google Cloud

1.3、SOA架构时代

在原始分布式时代就提到单体和分布式并行发展，单体架构在探索如何拆分、演化到事件驱动架构时，分布式也在曲折发展，SOA即将登上架构的舞台。

刚接触到“SOA治理”时我并不是很清楚是到底治理什么，SOA看起来似乎和微服务是一样的，并没有感受到明显区别。SOA（Service-Oriented Architecture）是面向服务的架构……很遗憾定义实在太多了，我没有找到权威的、明确的解释，但是通过总结起码有两点我觉得是各方达成共识的。

1）面向服务的架构中的“服务”是指包含执行完整的、独立业务功能所需的代码和数据的，注意是“代码”和“数据”，也就是说服务是一个独立的功能单元，这里的服务和事件驱动架构中的子系统非常类似，要比今天我们所说的微服务中的“服务”粒度要粗的多。

2）SOA的目的是复用，并为此制定了一系列标准，服务之间使用公共接口标准和架构模式，因此可以快速整合到新应用中。这让应用开发人员无需像之前那样重新开发现有功能，因为有公共的接口标准开发人员也不必了解如何连接现有功能 。

这么看SOA并不是解决新的问题，依然是在解决分布式服务刚被提出时就预见的困难，但是SOA针对这些问题的解决方案更具体、操作性更强，**并形成一揽子规范标准**，相比前面的烟囱架构、微内核架构、事件驱动架构，SOA就显得具体的多，举几个例子：SOA明确了采用SOAP 作为远程调用的协议、明确使用企业服务总线（ESB）来实现各个子系统之间的通信交互等等，从技术角度SOA成功地解决了分布式环境下出现的主要技术问题，那SOA为什么逐步被微服务替代呢?

因为SOA太重了，过于严格的**一揽子**规范定义带来过度的复杂性，落地成本非常之高。我们以远程服务调用（Remote Procedure Call，RPC）为例：**RPC 出现的最初目的是为了让调用远程方法和调用本地方法一样简单，**直到今天还有人这么认为，实际上今天的大多数 RPC 技术已经不再追求这个目标，因为通信不是无成本的。Andrew Tanenbaum 教授曾在1987年发布一篇论文，核心论点是本地调用与远程调用当做一样处理，这是犯了方向性的错误，把系统间的调用做成透明，反而会增加程序员工作的复杂度，这个观点在后几年一直有争论，直到大名鼎鼎的Sun公司一众大佬总结“通过网络进行分布式运算的八宗罪”该观点才被广泛认可，明确**RPC是语言层次的特征，而不是系统层次的特征。**

既然是RPC是语言层次的特征，编程语言那么多，最理想的是有一套统一的规范来解决RPC中如何表示数据、传递数据、确定方法三个基本问题（事实上直到现在RPC协议都是混乱的，数一数现在流行的RPC协议有多少个）。这个规范的发展很曲折，我们重点关注一个历史性的时刻：W3C在1998年发布SOAP 1.0来实现Web Service。SOAP当时风头一时无两，虽然现在来看他在性能上和易用性上并不完美，但是这并不是他没落的原因，而是因为过于“理想主义”的期望解决解决分布式计算中可能遇到的所有问题，为此定义庞大Web Service 协议家族，学习、使用成本非常高，脱离人民群众的做法导致SOAP必然逐渐淹没在历史的长河中。SOAP作为SOA登上架构舞台的重要条件，他的没落也意味着SOA架构的没落。

回顾“原始分布式时代”这一讲中：Unix DCE提出的分布式服务的主旨是“让开发人员不必关心服务是远程还是本地，都能够透明地调用服务或者访问资源”。但是现在SOA已经背离了这一初心，于是微服务时代到来。

1.4、微服务架构时代

微服务真正崛起实于2014年，这时候的微服务已经和9年前刚刚被提出时有了明显的区别。最开始微服务是SOA的轻量版，既然SOA过于繁重，那么**微服务就尽可能的抛弃SOA 里可以抛弃的所有约束和规范**，回归透明简单的初心；当然现在微服务已经脱离SOA成为一整独立的架构风格，Martin Fowler在《Microservices: A Definition of This New Architectural Term》一文中对微服务解释已经足够详细，这里不再赘述。

那么微服务和SOA到底有什么不同？讨论这个问题实在意义不大，因为他们之间的关系实在太微妙了，有人说ESB是SOA的特点，这其实是过于武断了，微服务抛弃了SOA中所有可以抛弃的标准并不代表微服务一定不用，别忘了微服务是自由的。我们只需要知道：**微服务提倡以“实践标准”代替“规范标准”，针对SOA解决的那些分布式服务的问题在微服务中不再会有统一的解决方案，开发者可以结合实际情况自由选择。**

微服务是普通开发者的狂欢，可以随心所欲；微服务是架构者的噩梦，对架构师的能力要求提升到史无前例。**微服务并不是什么银弹**，没有必要过分的神话他，分布式架构中出现的问题如注册发现、跟踪治理、负载均衡、传输通信等微服务一个都没有避免，该需要解决还是需要解决。但是在云时代下，微服务迎来新的契机：**云技术让微服务从软件层面独力应对分布式问题发展到软、硬协同应对**，也称为“后微服务时代”。

当问题不局限于采用软件方式那很多问题都简单了。例如软件可以通过命令伸缩扩容服务，硬件难道就不可以通过敲键盘就变出相应的应用服务器、负载均衡器等基础设施吗？氪金买云服务一样可以解决问题，虚拟化技术让软件、硬件的边界开始模糊，一旦虚拟化硬件变得足够灵活，那么和业务无关的技术性问题都可以解决于基础设施内，让研发专注于业务开发。后面也催生出了云原生等许多新的概念，但是从本质上讲他们所追求的目标和微服务没有什么区别。**微服务时代所取得的成就离不开以虚拟化技术的巨大贡献。**

微服务+云让普通程序员也同样能写出可用于生产的软件产品，但是对于架构师提出更高的技术要求，这会导致开发者的分层，形成大部分普通程序员加上小部分软件设计专家的金字塔结构。

1.5、无服务时代

回想分布式架构之所以存在，最开始就是因为单台机器的性能无法满足复杂系统的需要。那么如果单台机器如果性能无限所有问题都解决了。以前不敢做这样的假设，但是现在云计算技术的发展实际已经让这个畅想接近现实，而所谓无服务实际主要涉及两点：后端基础设施和函数。后端基础设施提供了日志、存储等等这一类用于支撑业务逻辑运行，而函数负责实现业务逻辑。我们不用考虑容量和算力问题，函数就是服务。现在云计算提供的能力已经让无服务初见端倪。

到这里架构演变的历史我们已经理清，相信你也理解了每种架构出现的意义与淘汰的原因，接下来我们一起讨论该如何选择架构来解决今天我们所面临的现实问题。

二、单体还是微服务

我们花了大量的篇幅去讲架构演进是为了在历史的浪潮中，理解每种架构出现的意义和淘汰原因，解决今天我们所面对的现实问题。直到现在关于架构好坏之争依然存在，最激烈的讨论就是的单体还是微服务。先说结论，假如一个业务发展良好，**当系统复杂度增加到一定程度必然是从单体迁移到微服务的，但在这之前，我推荐单体。**下面我们来看下是什么驱动复杂系统从单体向微服务迁移。

2.1、为什么要使用微服务

天下熙熙，皆为利来；天下攘攘，皆为利往。有收益才有付出，上文已经介绍既然单体是如此简单、自然为什么还要迁移到微服务呢？为了更好的性能？微服务可以扩缩容自如，很好的应对业务发展所带来的性能压力，问题是随着云计算的发展，单体服务同样可以集群化部署、同

样可以很方便的扩缩容，而且硬件的价格越来越便宜，甚至单体服务还减少了RPC所带来的性能损耗，这个理由看似正确，实际上是站不住脚的。仅仅为了性能而选择分布式的话，那应该是 40 年前“原始分布式时代”所追求的目标。**需求是不这样不行，而不是这样也行，同理复杂系统迁移微服务必然也有着非迁不可的理由。**

2.1.1、当团队内个人能力因素成为系统发展的明显制约

这个问题很好解释，前面我已经举了一个极端例子：有10万人写一个单体服务，假如每个人写出系统的可用性都能达到99.999%，那么这个单体服务的可用性只能达到36.8%。更何况没有人能保证招到的人全部都专业、靠谱，即使可以人也具有不稳定性，受到情绪、健康状况、人际关系影响也可能干出各种不靠谱的事，难以做出整体可靠的大型系统。**由于在单体架构下系统中“整体”与“部分”的关系没有物理的划分，一旦发生错误缺乏有效的阻断手段**，而少量的专家也没有办法控制某个人研发在某行不起眼的代码生成一个BUG并造成全局影响。

这种情况下，团队人员水平参差不齐，或者团队已经发展的非常大，微服务都是一个更好的选择。在单体架构下，系统质量只能靠研发与项目管理措施来尽可能地保障，而微服务已经假定系统一定会出错，架构本身就追求独立、自治，**让高水平专家来开发和维护关键的基础设置和业务服务**，至于其他非核心功能发生错误可以依靠基础设置的自愈能力恢复，退一万步来讲即使不能恢复影响也控制在一个小的范围，而系统整体依然是高可用的。

2.1.2、当语言或者技术框架成为系统发展的明显制约

语言没有好坏，但有适不适合，举一个非常简单的例子：近几年人工智能开展的如火如荼，所有的软件都在考虑怎么结合人工智能提供能优秀的服务，让业务进入第二增长曲线。但是稍微一调研就会发现Python对于AI训练的支持是其他所有语言所无法比拟的，如果原来的系统使用C++实现，这时候对不同技术栈、不同框架实现的功能进行分布式部署并不是你想或者不想的问题，而是没有选择、无可避免的。当然非要在C++里执行Python也可以，可是使用这些奇技淫巧所带来的维护成本是无法估量的。

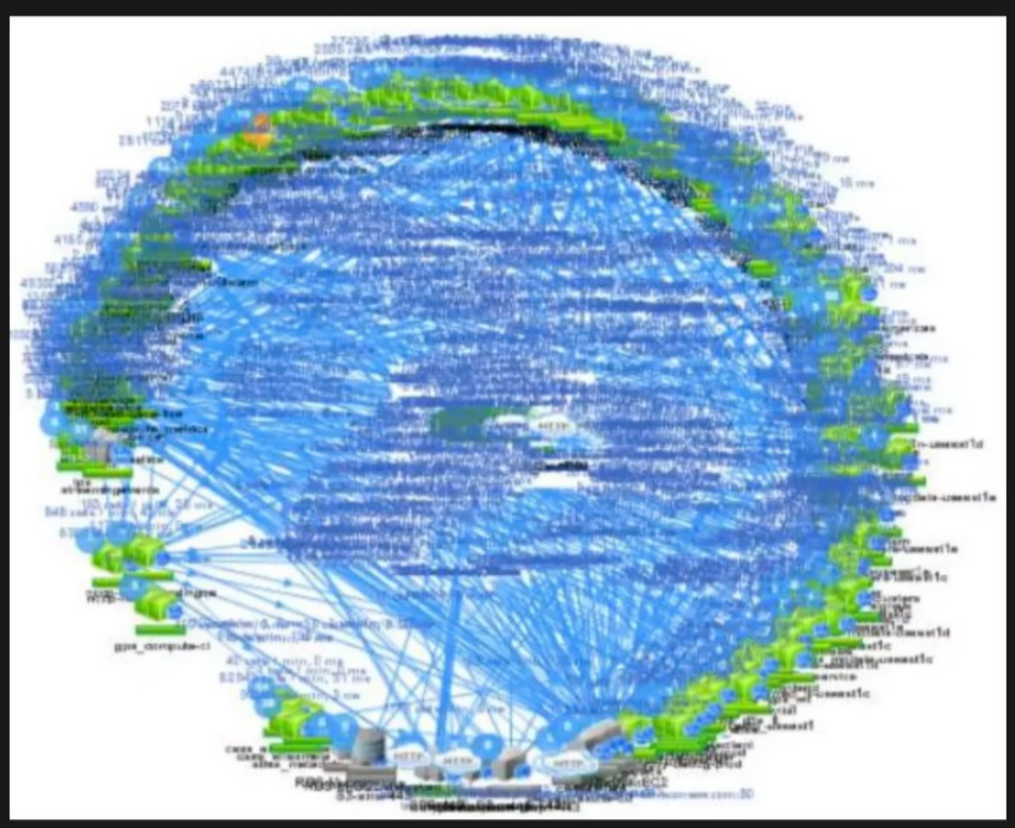
2.1.3、当组织的人员架构成为系统发展的明显制约

康威定律大家应该都了解，这里不做额外的赘述。变化是永恒的，随着业务规模的不断扩大，开发该系统的团队也必然不断扩大，组织架构随之调整是自然而然的事。我们对抗不了康威定律，随着组织架构的调整系统架构必然随之调整。举个简单的例子：我要对某个模块进行发布，该找谁审批，必然是自己的领导？但是如果是单体服务很可能涉及到多个团队，让每个团队的领导都来审批会导致发布效率低下，这是很难容忍的。

那如何识别组织架构已经成为制约了呢：**当所有的决策都要依赖某个人或者某个会议，对某个服务或者功能没有人愿意承担责任，或者愿意承担责任的人成为决策单点时，这时候就该意识到该拆分服务了。**

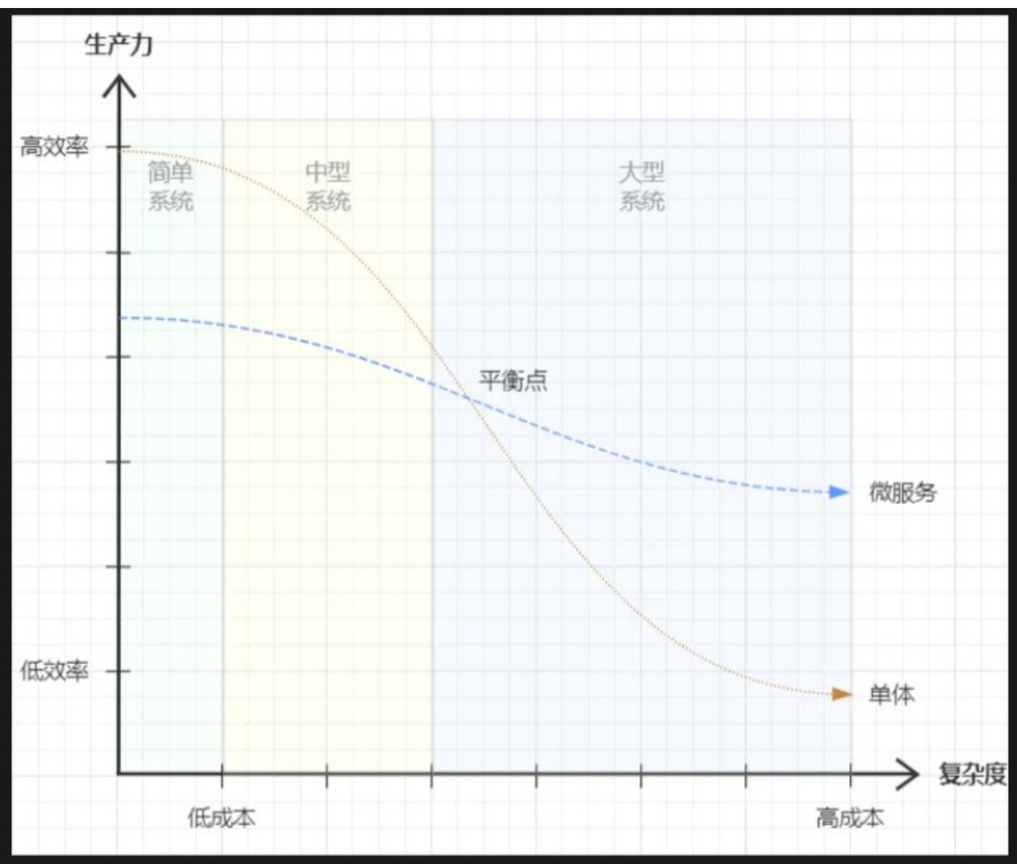
2.1.4、小结

我们要认识到微服务是有成本的，这里如果用一张表格来从各个纬度对比来说微服务需要解决哪些哪些问题可能并不会让你有直接的感受，下面是网上一张著名的图：Netflix公司在2014年公开的微服务调用关系图。我相信Netflix中没有一个人可以理清楚该系统的服务间调用关系，这是一个黑洞，如果某个微服务需要修改协议，很难理清楚到底需要周知哪些依赖方进行兼容修改。



Netflix公司在2014年公开的微服务调用关系图

微服务核心价值是拆分之后的系统能够让局部的单个服务敏捷开发，最主要的目的是对系统进行有效的拆分以进行物理隔离，**通过一系列的自制手段用不可靠的微服务构造可靠的系统，**然而微服务在解决单体问题的同时又会带来额外的很多问题，这里如果权衡不好反而会导致生产力下降，而系统进行任何改造的根本动力都是“这样做收益大于成本”，因此**“对于中小型系统单体架构就是最好的架构”，**只有在业务已经发展到一定的程度，单体架构与微服务架构的生产力曲线已经到达交叉点，此时开始进行微服务化才是有收益的。至于这个是否出现了这个交叉点如何判断，可以审查一下是否出现了以上三点。



图源《凤凰架构》

2.2、使用微服务的必备条件

即使我们有了充分的理由要迁移微服务也不能立刻迁移，还要看当前团队是否具备迁移微服务的条件，贸然迁移很可能会适得其反。

2.2.1、组织中具备对微服务有充分理解的专家

讲微服务时代时候就提到，微服务给予编码人员充分的自由，但对于架构师却提出了史无前例的高要求。在实际研发团队中并不是每个人都是架构师，也不是每个人都会去探究扩展、认证、隔离等和具体业务无关的问题，大多人普通程序员还是更聚焦于业务代码的编写。可是使用微服务却不关注这些问题那带来的灾难将是巨大的。举几个例子：微服务之间如何路由？微服务之间如何进行权限认证？微服务超时间如何配置才能规避重试带来的雪崩？流量变化以怎样的策略进行扩缩容？而这些问题不解决很可能影响整个系统的可用性。使用微服务架构的团队经常会遇到这中情况：微服务划分不合理导致高扇入、高扇出，加一个字段要修改N个模块、测试N个接口，极其低效。因此要想享受微服务带来的好处，首先就需要有专家认识到其中的风险，设计靠谱的架构。**如果整个团队中缺乏能够在微服务架构中撑起系统主干的专家，强行迁移微服务带来的收益不足以抵消复杂性增加而导致的成本。**

2.2.2、系统应具有以自治为目标的自动化与监控度量能力

我们期望团队具备这样一个资产管理系统，可以清晰的看到所有的资产、服务运行情况以及系统还有多少风险项没有消除。微服务是由大量松耦合服务互相协作构成的系统，一旦切换为微服务意味着资产成倍增长。我们再来回顾下：微服务是正视问题，认为“错误一定会发生”，因此要有一系列的手段来**用不可靠的微服务构造可靠的系统**。这个前提系统能够监控自身的健康状态以实现自动恢复，也就是我们需要奔着自动化运维目标前进。

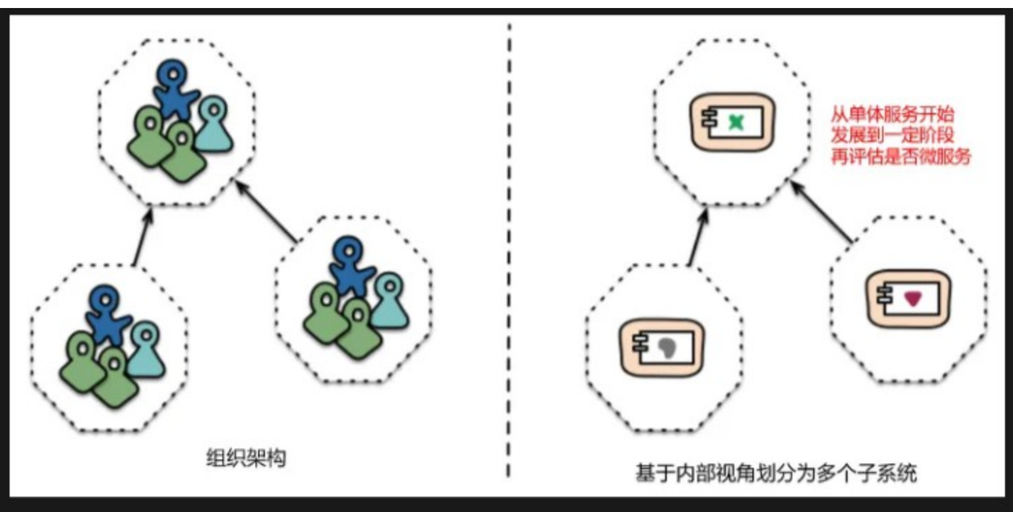
在团队缺少基础设施的情况下使用微服务是冒着极大风险的，想象一下系统故障，原来使用单体服务只需要查看少数模块就可以定位问题发生点，假如拆分为数百个模块，如果没有有效的监控系统，该需要多少时间才能定位到异常原因。

2.3、如何实践微服务

如何实践微服务这个话题太大了，远远不是一篇文章所能够承载的，光介绍微服务的书就有《微服务设计》、《微服务架构设计模式》、《微服务分布式构架开发实战》 等等很多，这里我只是简单提几个点与大家一起讨论：

如果团队规模较小，在开始一个新的系统的时我认为应该坚定不移的选择单体服务。这时候希望团队对于做出使用的单体的架构师有充分的信任。现实中往往在立项之初就会给业务定下长远的目标，但是从技术角度还是应该立足当下，然后随着业务不断发展对架构进行及时调整。切不可当业务发展到一定规模时需要拆分微服务时再来指责架构师：“当初我就说应该使用微服务，而不应该使用单体这种落后的架构”。客观的想一下，与一开始就用微服务比，前期使用单体所节约的工作量，比起迁移所带来工作是划算的。我们也应该有这样的意识：**我们没有办法设计出一个永远不变的完美的架构，架构一定是随着业务发展而不断变化的。**

如果团队规模较大，我相信没有哪个系统一开始就由一个“大团队”开发，而是随着业务发展系统规模越来越大，团队所需的人员也不断增长最终变成“大团队”。这种情况下应该通过领域建模后基于内部视角将系统划分为多个子系统，结合康威定律将这些子系统分配给不同的小团队负责。通常情况下此阶段系统已经变的较为复杂，达到了2.1.4小节图中效率和成本的平衡点，各个小团队经过评估后可以将所负责的子系统部署为相对独立的服务，这些服务之间的交互使用统一的规范。而且推荐数据去中心化，每个服务管理自己的数据库，即使这些数据在不同的子系统都可能出现，但是每个子系统所需要的属性多少有差异，还是推荐自己维护以便于团队独立迭代。实际上就是每个小团队维护了一个小的子系统，那么参考第一点“如果团队规模较小”时的结论，各个小的团队构建服务也从单体服务架构开始，没有必要从一开始就再拆分多个更小的服务。我所在团队也一直在探索和实践单体服务，某业务的核心功能到现在依然是单体，该业务已经发展三年，到现在为止尚没有因为使用单体而在某一点上明显拖慢效率。



三、尾巴

回到导语中的问题，单体和微服务谁是毒瘤，都不是，他们只是在某个历史阶段下适应潮流而生的一种普通的架构，并没有高低贵贱、先进落后之分；而现在，也只是我们结合业务和团队实际情况进行利益取舍后，供我们选择的方案之一罢了。

参考资料

1、《凤凰架构：构建可靠的大型分布式系统》周志明

2、《Microservices: A Definition of This New Architectural Term》 Martin Fowler、James Lewis

🎁 互动福利 🎁

评论区留言关于你对单体架构和微服务的理解

抽取1位粉丝送出100元京东卡~