为什么微服务实施那么难？如何高效推进微服务架构演进

前言

笔者从 2013 年加入 ThoughtWorks 至今共 4年时间。在这 4 年的时间里，我分别以 开发人员， DevOps 工程师、 DevOps 咨询师、微服务架构师以及微服务咨询师的角色参与了共计 7 个产品和项目的微服务咨询和实施。其中有有成功，有失败，有反思，更多的是学习和总结。以下是我这些年来在微服务咨询上的经验总结，希望能给陷入微服务实施困境的人带来一些帮助。

难点1：“一步到位”的认知错觉

这些年微服务大红大紫，但是真正能够拿出来做为可实践的案例少之又少。大部分的微服务案例只能看到微服务架构的“演进结果”。但是看不到微服务架构的“演进过程”。

这就给很多架构师一个假象：微服务的架构是通过能力极高的架构师一步到位设计出来的。这和很多产品团队自上而下的架构设计风格感受和相似。于是架构师们蜂拥而至，分析和讨论此起彼伏。各种分析方法论层出不穷，讨论和分享络绎不绝。然而真正落地实施的却很少，使得微服务在网络上慢慢变成了一种“玄学”，还停留在“讲道理”的阶段。

这违反了架构的最基本原则：架构是解决当前的需求和痛点演进的。而不是预先设计出来的。因此，整体的微服务架构设计完全没有必要。如果需要一个集中化的设计，那么如何体现微服务的去中心化轻量级优势?

可以说这是某些技术咨询公司的一种把戏，通过提升新技术的应用门槛把新技术变成一种稀缺资源。

从经济学上讲，我相信技术的发展一定是向不断降低成本的方向上发展的。如果新技术没有降低成本反而提升了成本，要么这个新技术有问题，要么一定是姿势不对，走错了路。

这就引出了了第二个难点。

难点2：“架构师精英主义”

很多产品对架构师的依赖很大，即“架构师精英主义”：认为产品架构只有这个组织的“技术精英”——架构师才可以完成，而团队其它成员只需要实现架构师的设计和产品经理的决策就可以。

而微服务架构则是一种“边际革命”：即由一个不超过8个人的小团队就可以完成的工作，两个人甚至都可以完成微服务。而这种规模的团队即使从整个产品团队移除也对整体产品的研发进度没有影响。因此，即使失败了不会带来太多的损失。然而，如果第一个微服务改造成功，那么成功经验的复制带来的乘数效应却能带来很大的收益。

从架构改造投资的风险收益比来看，这是非常划算的。

因此，微服务团队完全没必要大张旗鼓，只需要两三个人就可以动工。

但是，谁也没有微服务的实践经验啊，万一失败了怎么办？

这就带来了第三个难点。

难点3：缺乏一个信任并鼓励创新的环境

面对未知的领域，失败再所难免。而面对这个不确定性频发的世界，成功和失败往往不再重要：也许今天的失败，明天再看，就是成功，反之亦然。

无论成败，我们都能从行动的过程中有所学习和反思，而这样的经验才是最有价值的。成功仅仅意味着结果符合自己的假设，而失败则意味着结果不符合自己的假设。

然而，很多组织，尤其“精英主义”的产品团队，责任和压力往往在上层，由于组织庞大，金字塔的结构往往会构建一种以“不信任对方”为基础的制度。这种制度往往营造了一种“宁可不作为，也不能犯错”的文化。由于上层则需要对失败负责，使得任何创新停留在组织的上层的想法，难以落实推进。由于组织的长期合作形成了稳定的工作习惯和思维定势，使得整个组织在面对创新的时候“卡壳”。

而解决组织“卡壳”的办法就是引入“晃动器”，需要有外部的力量（例如新招聘的高管或外部咨询师）来打破当前的工作习惯和思维定势。组织才可以继续运转下去。

难点4：微服务技术栈的“选择困难症“

由于“精英主义”的架构师需要担负很大的责任，因此架构师往往承担着很重的压力。他们必须要为微服务架构谨慎的选择技术栈。因此会在不同的技术栈之间尝试。

对于习惯了在大型组织里面“长设计，慢反馈”的人们而言。更加认为这样的节奏是理所应当的。

另一方面，微服务开源社区的快速发展滋长了“架构师焦虑”：如果采用落后的技术会被同行鄙视，被不懂技术的老板鄙视，甚至被下属鄙视。因此架构师们疲于在各种新型的

技术栈之间比较和学习。此外，不熟悉技术往往会增大风险，架构师就需要更多的时间研究。带着“一步到位”的架构幻想对微服务技术栈精挑细选。而不会采用现有低成本的方案快速迭代的解决问题。

以上四点会让大型组织面对微服务实施的时候“卡壳”，而这往往会导致微服务实施容易忽略的最重要一点，我认为也是核心的一点：

难点5：对微服务的技术变革估计过高，而对微服务带来的组织变革估计严重不足

作为架构师，永远要不要低估康威定理的威力：**“设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。”**

如果你的组织结构是去中心化的小团队结构，那么不用担心，你的应用架构会朝组织架构的方向演进。

反之，如果你不是一个去中心化的小团队结构，那么微服务的架构会和组织架构格格不入。最好的结果是组织结构随着系统架构的改变而改变，否则产品架构会给组织带来很多沟通问题。

从制度经济学角度上讲，软件产品本身就是企业内部组织（员工）和外部组织（用户）沟通的代码化制度。这个制度的发展一定是在不断缩小内部组织之间以及内外部组织沟通成本的。

那么，如何高效的推动微服务架构演进呢？

如果以上 5 点都让你膝盖中箭。那么根据我个人的经验，综合解决微服务实施难点的第一条建议就是：

步骤1：以终为始，先构建一个独立的敏捷微服务团队

我们对微服务的期待就是：可以独立开发，独立部署，独立发布，并且去中心化管理。那么，我们就先构造一只“可以独立开发，独立部署，并且去中心化管理”的团队。

这个团队为了达到这个目标，会采取各种方法（例如：DevOps，全功能团队）解决阻碍”独立开发，独立部署，独立发布 和 去中心化的问题。而根据康威定理，系统的架构会慢慢向去中心化方向发展。

一定要意识到，这个过程会打破大型系统自上而下的所有流程并采用更有生产力的方式构建新的组织结构。充分信任团队，不要用老眼光控制团队的运作，这会打击团队的士气。

管理建议：

1. 让微服务团队完全脱离之前的工作，如果分心同时做几件事，每件事都不会做到最好。
2. 给微服务团队一些特权，为了满足“全功能微服务团队的”诉求，特事特办。
3. 如果团队在执行的过程出现了依赖从而阻碍了进度。则需要把依赖标明出来。代码中的依赖容易看见，但组织中的流程依赖很难发现。
4. 为了避免团队对外部的“依赖惯性”，让团队自己想办法在内部解决依赖。

技术建议：

1. 为微服务建立一个全新的代码库，而不要从原先的代码库上克隆或者复制，避免和原团队的开发依赖。
2. 建设一个独立的持续交付流水线，最好是通过“流水线即代码技术”（例如Jenkinsfile）来自动生成流水线。

步骤2：构建微服务的“电梯演讲”

成立了微服务团队之后，接下来就是要选择第一个实现的微服务。但是这个微服务应该多大，边界在哪是个问题。我的建议是通过“电梯演讲”的方式来定义微服务。格式可以是：

> （XX微服务）用来
>
> 在（出现痛点的场景）的情况下
>
> 解决了（解决现有的某个问题）
>
> 从而（达到什么样的效果）
>
> 提升了（微服务的价值）

例如：

> （订单查询微服务）用来
>
> 在（订单查询数量快速）的情况下
>
> 解决了（访问数量迅速升高导致整体应用性能下降的问题）
>
> 从而（分离了订单查询请求）
>
> 提升了（提升了其他功能的性能）

管理建议：

1. 把微服务的电梯演讲打印出来挂到墙上，让团队成员铭记于心。这会强化组织对微服务的边界认识。
2. 随着团队的反思和学习，电梯演讲有可能会变更，但一定要让团队形成共识好和一致的意见。
3. 不要期望一次就能划分正确。划分是一个持续权衡取舍的过程。
4. 随着团队的划分。

技术建议：

1. 明确了微服务的职责和边界之后再去看代码，否则会被代码的复杂度影响。
2. 领域驱动设计（DDD）可以帮助你更好的划分微服务。领域驱动设计很好的遵循了“关注点分离”（Separation of concerns，SOC）的原则，提出了更成熟、清晰的分层架构。
3. 不会领域驱动设计（DDD）也没有关系。简单的使用“关注点分离原则”也可以帮你达到这一点。例如：从接口中分离出流量较大的接口独立部署，把读数据库和写数据库的 API 分开独立部署，把静态和动态访问分离……等等。

步骤3：以最小的代价发布出第一个微服务

这里面要注意两个关键点：一个是“最小的代价”，另一个是“发布”（Release）。

正如前文所述，微服务架构本身就觉了微服务一定是低成本低风险的渐进式演进。而最大的浪费在于：

1. 级别/职责分工明确的组织沟通结构。

2. “长时间，慢反馈”的行动习惯。

3. 先进且学习成本较高的技术栈。

因此，“最小的代价”包含了以下三个方面：

1. 最精简的独立敏捷全功能团队。

2. 最快的时间。

3. 代价最小的技术栈。

此外，很多微服务的“爱好者”由于害怕失败，因此将微服务技术始终放在“实验室”里。要勇于面对失败，在生产环境中面对真实的问题，但要采取一些规避风险的措施。

管理建议：

1. 尽量让现有微服务团队自己学习解决问题，成为全功能团队。如无必要，绝不增添新的人手。
2. “扯破嗓子不如甩开膀子”，先动起来，在前进中解决问题。
3. 先考虑最后如何发布，根据发布流程倒推。

技术建议：

1. 根据当前技术采用的情况选择代价较小的技术栈。
2. 采用动态特性开关（Feature Toggle），在发布后可以在生产环境动态的控制微服务的启用，降低失败风险。
3. 如果采用了特性开关，一定要设立删除特性开关和对应旧代码的时间，一般不超过两个月。否则后面大量的特性开关会带来管理成本的提升和代码的凌乱。
4. 由于团队比较小，功能比较单一，不建议采用分支来构建微服务，而应该采用单主干方式开发。

步骤4：取得快速胜利（Quick wins），演示驱动开发

刚开始进行微服务改造的时候一定会是一个试错的过程。如果目标定得太大，会让团队倍感压力，从而士气低落。而制定每日的短期目标，赢得快速胜利则会不断激励团队的士气。通过设定当天结束的产出来确定今天需要做什么是一个非常有效的办法。

每日演示（Daily Showcase）就是一种推进产出的做法。每天向团队分享今天的工作内容，使小组能够共同学习。并且以当天或者明天的 showcase 作为目标。每个人 showcase 的内容一般不超过20分钟，一天的 showcase 时间不超过一小时。可以早上 showcase，也可以下班后 showcase。

常见的快速胜利如下：

1. 构造出第一个微服务 Hello Word。
2. 构造第一条微服务流水线。
3. 构造出第一个微服务自动化测试。

而以下的目标不适合作为快速胜利的目标：

1. 构造出微服务 DevOps 平台。
2. 完成产品的微服务架构拆分。
3. 构造微服务自动化运维体系。

管理建议：

1. 要防止团队画大饼，完成好每日和每周的工作目标即可。微服务开发本身就没有很长周期。
2. 强迫团队有所产出，这样才能用关键产出驱动开发。产出不一定是代码或者基础设施，一篇总结，或者学习的文章分享，甚至是踩过的坑和遇到的问题都可以展示，目的是要打造自治学习的团队。
3. 贵在坚持，不要计划太远。超过一个月，就要对目标是不是范围过大进行反思。
4. 以天为单位拆分任务，超过一天的必须要拆分。无法在一天完成的工作需要拆分出阶段性产出。
5. 如果能结对，并且能够每天交换结对，showcase 不必要。
6. 可视化所有任务，用敏捷看板来管理任务是了解现状的最好方式。

技术建议：

1. 除了让第一个 HelloWord 微服务尽快发布到生产环境，其它的不要想太多。
2. 完成了 HelloWord 的发布，然后要考虑如何对发布流程进行改进。而不是上线业务。

步骤5：代码未动，DevOps 先行

我把微服务的技术架构问题比作“搭台唱戏”：首先需要建立好微服务交付和运行的平台，然后让微服务上台“唱戏”。

这个平台一开始不需要很完善，只需要满足生产上线的必要要求即可。而在很多企业里，这个部分是由 Ops 团队在交付流程的末尾把关的。因此，把最后一道关卡的确认工作放到最前面考虑可以减少后期的返工以及不必要的浪费。

以前，软件的开发和测试过程是分开的。然而，随着 DevOps 运动的兴起和各种自动化运维工具的兴起，这之间的必要性不如从前，只要有足够的自动化测试做质量保证，就可以很快的将微服务快速部署和发布到生产环境上。

最开始的时候，哪怕是发布一个 Hello World 程序，都表明微服务的持续交付和运行的平台已经搭建好。流程已经打通。

DevOps 不光是一系列技术，更是一种工作方式。

从技术交付产物来说，DevOps 主要交付两点：

1. 持续交付流水线。
2. 微服务运行平台。

为了保证微服务交付的高效，需要把这二者通过自动化的方式有机的结合起来，而不是各为其主。让开发和运维的矛盾变成“自动化的开发运维矛盾”

从团队工作方式来说，DevOps 要做到：

1. 要让 Dev 和 Ops 共同参与决策，设计，实现和维护。
2. 团队完全独立自主，打破对现有流程的依赖。
3. 不断的追求改进，让团队行程改进的团队文化。

管理建议：

1. 给团队继续前进最大的动力就是新程序快速投入生产。
2. 如果你的组织是 Dev 和 Ops 分离的组织，先咨询一下 Ops 工程师的意见。最好是能够给微服务团队里面配备一名 Ops 工程师。
3. 如果不具备实施 DevOps 的条件，微服务架构就要从运维侧，而不是开发侧开始进行。

技术建议：

1. 微服务的平台一开始可以很简单，可以以后慢慢增强和扩展。但是一定要部署到生产环境里使用。
2. 如果想使用现成的微服务平台可以参考 Spring Cloud。
3. 微服务运行平台可以通过反向代理和生产环境并行运行。
4. 采用灰度发布技术在生产环境中逐步提升微服务的使用占比。
5. 基础设施即代码是 DevOps 核心实践，可以帮助开发人员迅速在本机构建生产环境相似的开发环境，减少环境的不一致性。可以采用 Docker，Ansible，Vagrant 等工具来完成。
6. 基础设施对微服务应该是透明的。微服务不应该也没必要知道运行环境的细节。只要能够正常启动并执行业务就完成了它的任务。因此，基础设施代码要和微服务业务代码分开，且微服务不应该告诉平台自己如何部署。
7. 服务注册和发现是微服务架构的核心部分。consul 和 Eureka 是这方面的佼佼者。
8. 部署（Deploy）和发布（Release）要分开。

步骤6：除了提交代码和发布，微服务平台一切都应当自动化

在完成了微服务的基础设施之后，就可以开始实现微服务的业务了。这时候需要依据电梯演讲划分出来的微服务进行业务逻辑的开发。在以 DevOps 的方式工作一段时间之

后，团队应该养成了一些自动化的习惯，如果没有，就应该检查一下自己的自动化程度。最佳的自动糊理想的状态就是除了代码提交和发布。在这之间的每一个流程和环节都应当由自动化的手段来完成。

当然，也有不能自动化的部分。根据我的经验，不能自动化的原因主要来自于流程管理的制度要求，而非技术困难。这往往是组织没有依据微服务进行流程变革导致的。这时候需要检讨不能自动化的部分是不是有存在的必要。

另一方面，虽然自动化可以大量缩短微服务交付时间，提升微服务交付效率。但是自动化的同时需要考虑到安全因素和风险，不能顾此失彼。对于生产来说，可用性和安全性是最重要的部分。

关键的自动化：

- 自动化功能性测试（UI/集成/单元/回归）
- 自动化构建
- 自动化部署
- 自动化性能测试
- 自动化安全扫描

管理建议：

1. 鼓励团队成员自发的进行自动化的改进，这会给未来微服务批量开发带来很多裨益。
2. 不要一开始就追求全面的自动化，自动化需要花费一定时间。根据团队的进度视情况适度进行。

技术建议：

1. 采用 TDD 的方式开发不光可以提升质量，也完成了测试的自动化。
2. 注意自动化的安全隐患。机密信息需要独立管理。
3. 关键步骤需要准备自动手动两种方式，必要时可以干预自动过程。
4. 采用 git 的 hook 技术，在代码 push 之前就可以完成测试和静态检查，提升 CI 的成功率。

步骤7：总结并复制成功经验，建立起微服务交付的节奏

当完成了第一个微服务，不要着急开始进行下一个微服务的开发。而是需要进行一次关于可复制经验的总结，识别微服务开发中的经验教训并总结成可复制的经验和产出。

以下是一些需要总结出来的关键产出：

1. 微服务开发到发布的端到端流程规范。
2. 微服务开发的技术质量规范。
3. 团队合作中的坚持的最佳实践。
4. 常见技术问题总结。

有了以上的关键产出，就可以对微服务开发团队进行扩张。这时候有了微服务开发的老司机，带着刚加入的同事一起开发，风险会相对低很多。

管理建议：

1. 刚开始的时候可以每周进行一个回顾会议，团队需要快速的反馈和调整。
2. 不要急于扩张团队，要在成功经验稳定并形成模式之后再快速扩充。
3. 避免微服务良好的开发氛围被稀释，刚开始的时候扩充团队可以慢一点。新老成员的配比不要超过1:1。
4. 虽然微服务平台趋于稳定，但在微服务没有上规模之前，不要让团队里缺少 Ops 成员。
5. 注意知识的传递和人员的培养。

技术建议：

1. 不要急于在微服务应用规模不大的时候形成微服务模板，否则会限制未来微服务的开发和扩展。
2. 在微服务不成规模的时候不要放松对微服务平台的改进。

参考书目

《微服务设计》 是一本微服务各个方面技术的综合参考材料。如果你在实施微服务的过程中碰到了问题，它就是一个解决方案的分类汇总。

《持续交付》汇集了很多交付最佳实践，当你的微服务实施碰到阻碍时，里面的建议能够让你解决当前的困境。

《领域驱动设计》和《实现领域驱动设计》为拆分微服务提供了方法论，当团队之间对于微服务的拆分有困难的时候，采用领域驱动的方法往往会得到更好的效果。