**新项目别一上来就用微服务**

<https://mp.weixin.qq.com/s/wfTp-grcb0wsRtFrMvzecQ>

微服务变得越来越理所当然，似乎我们一直生活在微服务的世界中。很多时候，我们常常讨论微服务采用与否、如何选型等问题。但本文作者 Arnold Galovics 想讨论的是，为什么一个全新的项目从开始就使用微服务通常是个坏主意。很长一段时间以来，他都在思考这个问题。

最近，当他与其他开发人员交谈并询问他们如何启动一个全新项目时，答案几乎都是：这儿用一个微服务，那儿用一个微服务，用户管理用一个微服务，身份验证用一个微服务，鉴权用一个微服务，session 管理用一个微服务等等。因此，关于微服务，Arnold 想基于他过去工作项目的一手经验，讲讲别人没有讲过的一些东西，他撰写了一篇题为《不要从微服务开始，单体架构才是你的朋友》（Don’t Start With Microservices – Monoliths Are Your Friend）的文章。

Arnold 的文章很快在技术社区引发热议。有持赞同意见的人直言，微服务对于大多数普通需求来说是一种“矫枉过正”，还有人提出微服务有个 Arnold 没提到的更严重的缺点——将事情分成模块需要时间，并且涉及做出我们可能不知道答案的决定。“启动新产品时，最重要的是尽快启动并运行产品，以便人们可以试用并提供反馈。而根据收到的反馈，我们往往可能会意识到需要构建与现有的完全不同的东西。我见过很多工程劣质的成功产品，也见过很多设计精良的产品失败。产品的成功与其设计的好坏无关。速度往往是最重要的因素。”

另外，有条热门评论提出了个有意思的问题：为什么没有人谈论介于这两者之间的架构——模块化单体？

对此，有回复说道“因为没有新发明的架构称为‘模块化单体’——单体从一开始就应该是模块化的。”该回复进一步指出，微服务不是单体应用不好而诞生的答案，人们的理解在某些地方出现了问题，这也可能是因为很多人不知道应该在代码中构建模块，然后大量的单体最终成为“意大利面条式”代码，就像现在许多微服务架构那样。

有人对此表示赞同并表示，“模块化单体”只是“代码中关注点的适当分离”，其实从一开始就存在。但也有人提出反对意见，认为“模块化单体”要比“分离关注点”要更复杂，并不完全是一回事。

一千个人眼中有一千个哈姆雷特，我们将 Arnold Galovic 的这篇文章翻译出来，希望能为读者带来一些参考价值。以下是他的分享内容：

**理想中的微服务**

我们先来看看，多数文章提到的一些微服务的主要优势有哪些：

* 故障隔离
* 消除技术锁定
* 更容易理解
* 更快的部署
* 可伸缩性

是的，这些都不是书中的虚假承诺，但我也必须对你说实话，使用微服务的话，你的系统很难实现这些承诺。下面让我一一列举这些优势。

**故障隔离**。由于应用程序由多个服务组成，因此如果其中一个服务宕机或出现问题，则只会影响系统的那个部分。以 Netflix 为例，当你观看节目时，你并不关心推荐。因此，如果它们有一个服务来处理当前观众，为他们提供视频流；它们有另一个服务来处理个人用户推荐。如果推荐服务宕机，系统中最重要的功能（例如观看节目）不会受到影响。故障被隔离了。

**消除技术锁定**。想想单体应用。它是一个巨大的应用程序，有成百上千个 API，管理数百个数据库表。这个应用程序是用 Java 写的，团队花了 5 年时间开发它。一种奇特的新语言出现了，纸面上带来更好的性能、提供更好的安全性等等。这可能是 Go 或 Rust，团队想试验下该语言及其技术栈。他们如何在一个单体应用中做到这一点呢？他们做不到，因为这是一个单独的部署包。你可以将应用程序的一部分切换到不同的语言，但这并不容易做到。

使用微服务时，不同的服务可以使用不同的技术栈。服务 A 可以用 Java 写，服务 B 可以用 Go 写，服务 C 可以用Whitespace写，如果你有勇气这么做的话。

**更容易理解**。当你有多个服务负责整个功能的一小部分时，这个服务本质上会更小，因此更容易理解。

**更快的部署**。在常规的单体应用系统中，要么完全部署，要么根本不部署。你需要部署一个包，这是一个要么全有要么全无的场景。使用微服务，你有机会独立部署，这意味着，如果你想要部署推荐服务的一次升级（回到 Netflix 的例子），你可以部署单个服务并节省大量时间。

**可伸缩性**。我的最爱。你可以通过启动多个实例来增加特定功能的容量，从而扩展服务。像前面举的例子，如果人们在 Netflix 上查看大量推荐，它们可以很容易地启动多个推荐服务的实例来应对负载。在单体应用环境中，你要么扩展应用程序的每一个部分，要么什么都不扩展。

## ****现实生活中的微服务****

我要用残酷的事实打击你，我的朋友。**我并不是说这些优势无法实现，但是你、你的项目、你的组织必须非常努力才有可能实现这些优势。**

### 基础设施要求

下面让我从微服务的一个最大困难——基础设施开始讲起。

你曾经部署过单体应用吗？当然，我们可以将其复杂化，但在常规情况下，如果将应用程序部署到云上，单体应用就是你所需要的形式。让我们以一个简单的在线商店应用程序为例：

* 应用程序的一个负载均衡器
* 运行应用程序的一个计算实例
* 应用程序的一个（关系型）数据库
* 用于日志聚合的 Kibana 如果你用的是微服务：
* 一个 Kubernetes 集群
* 一个负载均衡器
* 运行应用程序和托管 K8S 集群的多个计算实例
* 一个或多个（关系型）数据库，取决于你是否每个服务用一个数据库
* 一个用于服务间通信的消息系统，例如 Kafka
* 用于持续集成（持续部署）的 Jenkins
* 用于日志聚合的 Kibana
* 用于监控的 Prometheus
* 用于分布式跟踪的 Jaeger/Zipkin 而且这只是一个高层级的概览。微服务确实
* 可以产生价值，但问题是：代价是什么？

尽管这些承诺听起来很好，但你的架构中有更多活动的部件，这自然会导致更多的失败。如果你的消息系统挂了怎么办？如果你的 K8S 集群出现问题怎么办？如果 Jaeger 宕机，而你无法跟踪错误怎么办？如果指标没有进入 Prometheus 怎么办？

显然，你将花费更多时间（和金钱）来构建和运转这个复杂的系统。

### 更快的部署?

我将讲讲优势列表中的第一点：更快的部署。当你想到 Netflix、Facebook、Twitter，并且观看他们的会议演讲，他们描述他们正在运行的微服务数量，以及他们如何向 Git 提交内容，并在数小时内将其投入生产。这是不是好得难以置信？

在我看来，这绝对是可以实现的，但我承认我从未参与过这样的微服务项目。我并不是说这是不可能的，只是从稳定性、基础设施和团队文化角度来看，这真的很难实现。

让我分享一下我是如何从我的经历中得出这个结论的。在对一个全新的项目进行编码之前，你通常会先研究如何将产品转变成技术方案。你会设计系统，设计微服务，会有多少个微服务，每个微服务的职责等等。

有一个真正的教学项目，我们在这个项目中练手，我们最终做了 80+微服务，用了多长时间呢，4 个月？

这 80+微服务的现实意义是，与其将这 80+微服务一起组合成一个单体应用并部署这个单体应用，部署单个微服务绝对会更快，但是.......这 80+微服务太小了，以至于一个开发单元——敏捷领域的叙事——永远不可能只涉及一个服务。系统从根本上被破坏了，更快的部署的承诺立即消失了。我们不再拥有更快的部署，而是相反，更慢的部署。而且慢得多。

另外，我会反复思考这个问题。部署过程中活动的部件越多，意味着潜在故障越多。很多时候，基础实施不够稳定，部署会随机失败，因为：

* 在下载/上传软件包时，Artifactory/Nexus/Docker 仓库短时间内不可用；
* Jenkins 构建器随机卡住。这只是其中的一部分。产品必须分解为微服务。每个服务都必须对其自己的事情负责。例如，Netflix 中的一个推荐服务应该负责向用户提供推荐。

\*\*不是谁都是 Netflix，也不是所有东西都容易分解成合适的大小和职责。\*\*这是领域驱动设计（Domain Driven Design，DDD）和有界上下文可以提供帮助的地方，但这实践起来并不容易，有时甚至没有足够的时间/开发性来试验这些方法。

### 更快的部署?

我将讲讲优势列表中的第一点：更快的部署。当你想到 Netflix、Facebook、Twitter，并且观看他们的会议演讲，他们描述他们正在运行的微服务数量，以及他们如何向 Git 提交内容，并在数小时内将其投入生产。这是不是好得难以置信？

在我看来，这绝对是可以实现的，但我承认我从未参与过这样的微服务项目。我并不是说这是不可能的，只是从稳定性、基础设施和团队文化角度来看，这真的很难实现。

让我分享一下我是如何从我的经历中得出这个结论的。在对一个全新的项目进行编码之前，你通常会先研究如何将产品转变成技术方案。你会设计系统，设计微服务，会有多少个微服务，每个微服务的职责等等。

有一个真正的教学项目，我们在这个项目中练手，我们最终做了 80+微服务，用了多长时间呢，4 个月？

这 80+微服务的现实意义是，与其将这 80+微服务一起组合成一个单体应用并部署这个单体应用，部署单个微服务绝对会更快，但是.......这 80+微服务太小了，以至于一个开发单元——敏捷领域的叙事——永远不可能只涉及一个服务。系统从根本上被破坏了，更快的部署的承诺立即消失了。我们不再拥有更快的部署，而是相反，更慢的部署。而且慢得多。

另外，我会反复思考这个问题。部署过程中活动的部件越多，意味着潜在故障越多。很多时候，基础实施不够稳定，部署会随机失败，因为：

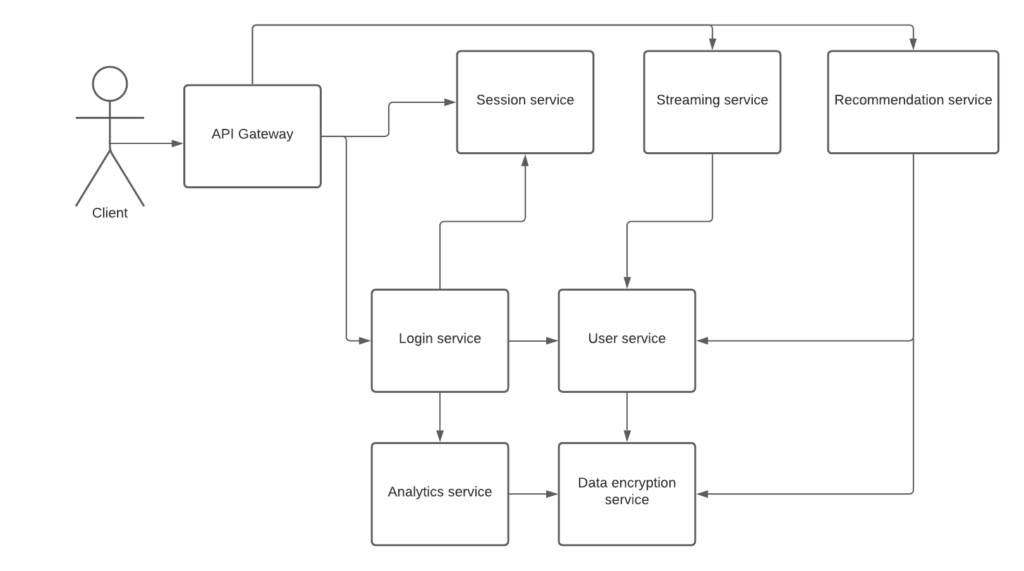
* 在下载/上传软件包时，Artifactory/Nexus/Docker 仓库短时间内不可用；
* Jenkins 构建器随机卡住。这只是其中的一部分。产品必须分解为微服务。每个服务都必须对其自己的事情负责。例如，Netflix 中的一个推荐服务应该负责向用户提供推荐。

\*\*不是谁都是 Netflix，也不是所有东西都容易分解成合适的大小和职责。\*\*这是领域驱动设计（Domain Driven Design，DDD）和有界上下文可以提供帮助的地方，但这实践起来并不容易，有时甚至没有足够的时间/开发性来试验这些方法。

### 更好的故障隔离

这一点自然是正确的。如果一个服务挂了，只有那个服务会受影响，对吧？

虽然确实如此，但这并不是绝对的。让我给你展示 Netflix 的一个虚拟架构图——我对其进行了简化：



假设用户想要看推荐。请求转到推荐服务，它查询用户数据来了解用户详情，并将推荐存储在其数据库中（不在图片上），而且由于这是用户相关的数据，所以它们可能需要将其加密。

现在，如果数据加密服务挂了会发生什么？我们还能做推荐吗？肯定不能，因为我们不能加密用户数据，所以我们自然会说，嘿，伙计，我们现在不能给你推荐，请 5 分钟后再试。这个故障影响到系统中的推荐服务，系统会以无法立即提供推荐的事实来优雅地做出回应。

但是你知道要优雅地处理这类情况需要做多少工作吗？非常多。

让我们再举一个例子。用户尝试使用登录服务来登入系统。数据加密服务仍在故障，登录服务调用分析服务来获取在一个时间区间内有多少用户正在尝试登入的指标，以及其它一些虚构的指标。不过，分析服务也在与数据加密服务通信，因为这些数据也需要加密。

现在，编写分析服务的团队正忙着，没有时间来实现适当的异常处理，因此数据加密服务的问题会转而影响到登录服务。显然，登录服务是在几个月前完成的，这个服务没有准备好处理来自分析服务的底层错误，因此即使不关键的分析服务失败也会导致用户登录被拒绝。

而且我知道你是怎么想的。是的，实现登录服务的团队不负责为处理这种情况做准备，但是如果他们认为分析服务会优雅地处理这个异常呢？这已经写在分析服务的 API 合约中，但它没有那样生效。

那么，当你在一个单体应用程序中，会发生什么呢？一个服务崩溃在这个上下文中并没有真正的意义，但假定由于某种原因，连接到数据加密的数据库表不可访问了。

在这种情况下，异常处理非常简单，因为你只需要准备一个 exception 就可以了。但是在过分赞扬单体应用前，需要说的是，单体应用也有缺点，如果单体应用挂了，什么东西都不可用了。**因此，这是一个平衡问题，需要问问你自己。是实现一个 try-catch 代码块更容易，还是处理一个同步 HTTP 调用异常或异步消息异常更容易？**

我记得，对于 80+微服务，标准化异常处理是非常了不起的事情，它需要一个团队花费数月来完成。这甚至不意味着在每个地方都引入异常处理，而只是将现有的异常用我们使用的一个自定义库重写，这样我们就可以减少未来异常处理场景所需的繁琐工作。