讲讲拆分：从单体式应用到微服务的低风险演变

2017-11-21 Christian Posta EAWorld

本文获得blog.christianposta授权翻译发表，转载需要注明来自公众号EAWorld。

作者：Christian Posta

译者：海松

原题：Low-risk Monolith to Microservice Evolution Part I

在为期两天的微服务研讨会中，我一直在思考如何向大家解释单体应用（monolith-application）分解以及它向微服务过渡后将会是什么样。本文是该主题的一小部分，但我想与大家分享，并得到反馈（在研讨会中，我们会更详细地讨论是否应该分解单体应用！）。本文中我总结了一些亲历的经验，以及在过去几年中与北美许多红帽（Red Hat）客户合作的经历。这里的第一部分主要探讨了架构，即将发布的第二部

分则会介绍一些能提供 很大帮助的技术。关注我的**Twitter（@christianposta）**或**http://blog.christianposta.com**，可以获取最近的更新和讨论。

在深入讨论之前，让我们先做以下假设：

- 微服务架构并非总是适用的（后面会详细讨论）
- 需要采用微服务架构时，我们应该确认单体式应用（monolith）会因此发生什么变化。
- 在极少数情况下，单体式应用会按照原样拆分。其它大多数情况下，要么需要构

建新的特性，要么围绕单体式应用重新实现现有的业务流程（这有点逆潮流而动）

- 在需要拆分功能或重新实现的情况下，一个不能忽略的事实是单体式应用如今仍在用于生产，并带来巨大的商业价值。
- 要设法解决这个问题，同时还必须保证不会干扰系统的整体业务价值。
- 由于单体式应用是个整体，所以变更它下面的数据模型或数据库时非常困难甚至是不可能的。
- 我们应当降低演变的风险，这过程中可能需要多次部署和发布。

一、抽取微服务

有关这个话题的会议或博文中，都提供了以下的建议：

- 围绕名词进行组织
- 做一件事，做好一件事
- 单一责任原则
- 它很难

但这些建议没什么用。

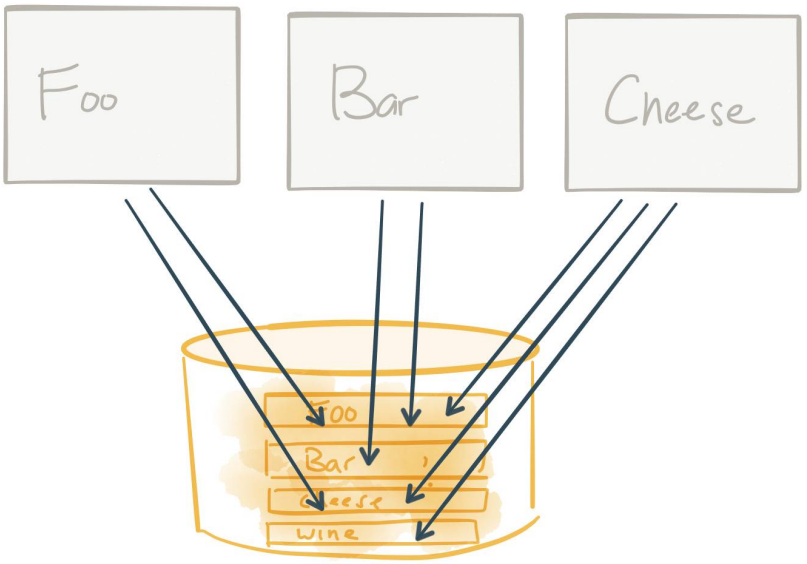
有效的建议应该像这样：

- 识别模块（现有的或是新的模块）
- 拆分出与这些模块相对应的表，并用服务进行包装

- 更新此前直接依赖数据库表的代码并用它调用新服务
- 重复上述流程（Rinse and repeat）

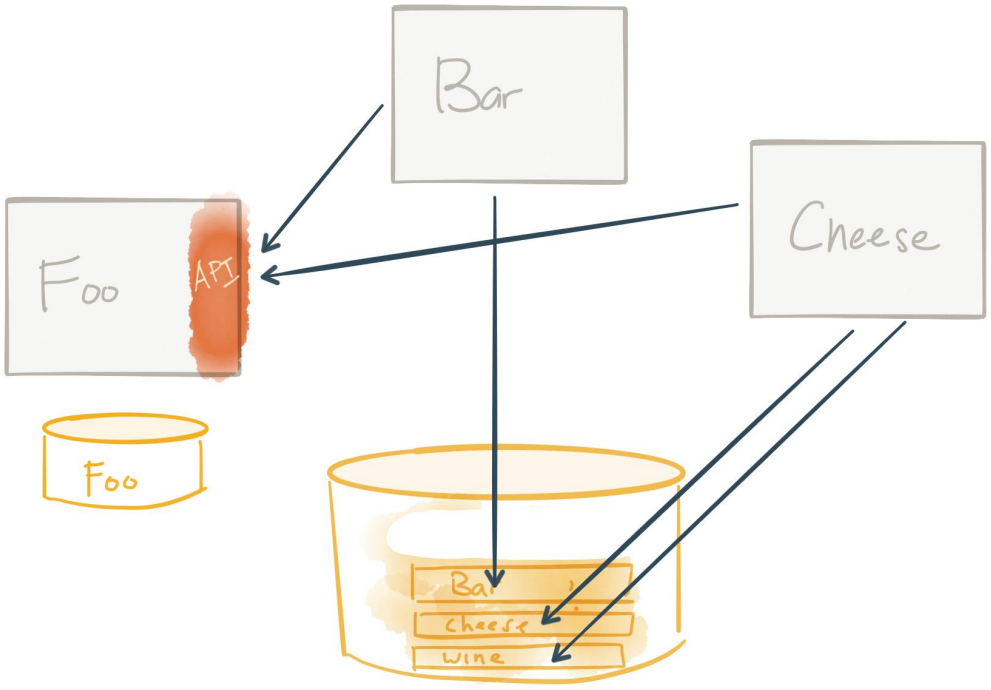
具体来说：

第1步：识别模块



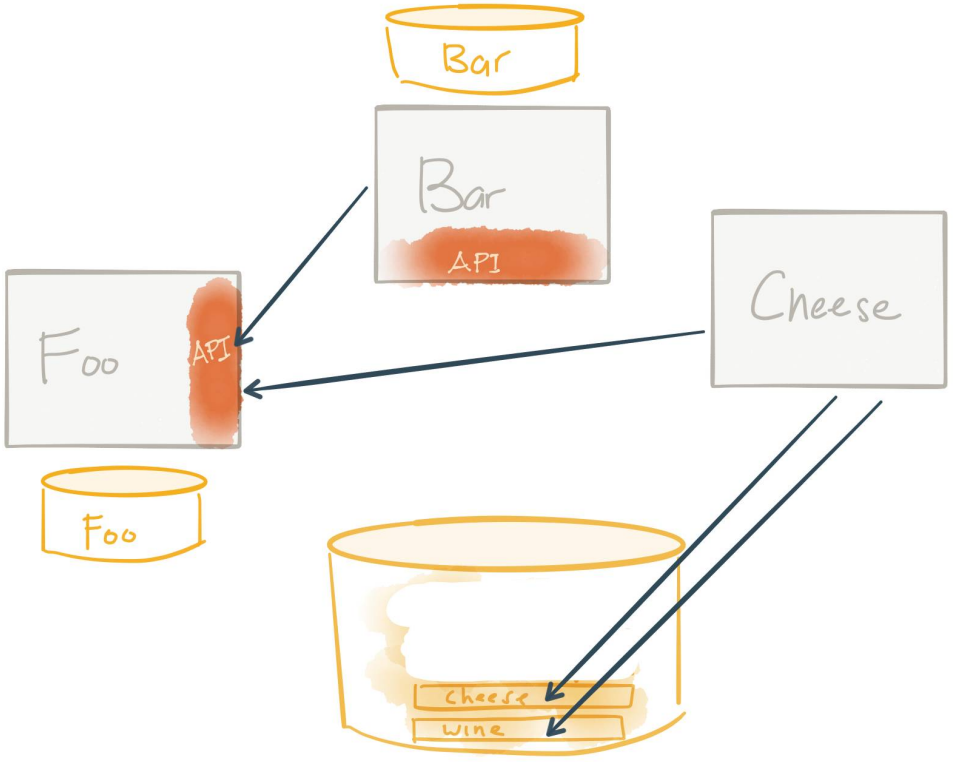
这一过程从烦人的单体式应用开始。在上图中，我简化了这一点来表示其中可能涉及到的不同模块和数据库表。我们要确定哪些模块是想从单体式应用里拆分出来的，找出涉及到的表，然后继续。当然，现实情况是单体式应用极易与模块（如果有的话）相互缠绕。

第2步：拆分数据库表，用服务包装，更新依赖关系



第二步是确定Foo 模块使用了哪些表，将它们拆分，然后加入模块自身的服务中去。该服务就成为现在唯一能访问这些Foo表的服务了。再没有别的共享表了！这是件好事。过去引用Foo的所有功能现在都必须经过新创建的服务的API。在上图中，我们更新了Bar和Cheese服务，当它们需要Foo的时候，会引用Foo服务。

第3步：重复上述流程



最后一步是重复这个过程，直到单体式应用全部消失。在上图中，我们对Bar 服务做了同样的处理，把它搬到了一个架构里，在这里，服务拥有自己的数据和开放的API，这听起来已经很像是微服务了。

通常，这算是一套不错的指导方针，**但上述步骤其实回避了许多我们不应忽略的真相**。比如我们不能要求时间暂停，然后从数据库中把表删除。同样的：

- 很少能简洁漂亮地将单体式应用模块化
- 表格间的关系可以高度规范化，而且在各实体之间表现出紧密的耦合或完整性约束
- 我们不可能完全清楚单体式应用中的某些代码到底调用了哪些表格
- 虽然我们已将表抽取到了一个新的服务中，但这并不意味着现有的业务流程停止了，我们可以让他们一个个迁移到新的服务
- 有一些烦人的迁移步骤也不会凭空消失
- 可能会存在一些收益递减的回报点，从这个点开始，把某些东西从单体式应用中拆分出来是毫无意义的

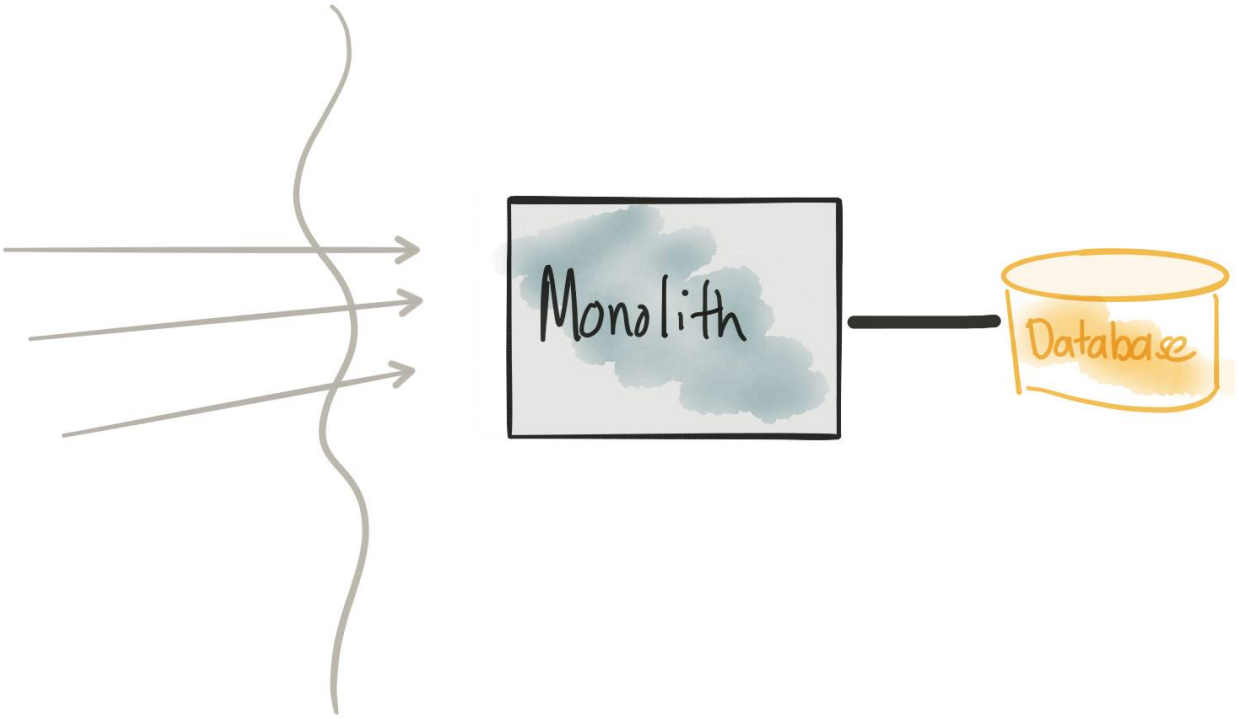
……等等等等

现在让我们来看个具体的例子，看看这个方法/模式是什么样的，以及可供的选择都有哪些。

二、具体举例

这个例子来自上面提到的研讨会。我将在分析拆分服务时做些润色，但是研讨会上谈到的更多内容，包括领域驱动设计、耦合模型以及物理或逻辑架构，这里先暂时不提。这个方法表面上似乎只能用于分解现有单体式应用的功能，但其实它同样能为单体式应用增加新功能。后者出现的概率可能更高，因为直接变更单体式应用风险是相当大的。

三、了解单体式应用



这就是单体式应用（Monolith）。它建立在developers.redhat.com上的**TicketMonster**①教程的基础上。该教程最初只是探讨如何构建一个典型的Java EE应用程序，但最终却成了一个很好的例子：它不过于复杂，而且有足够的内容让我们

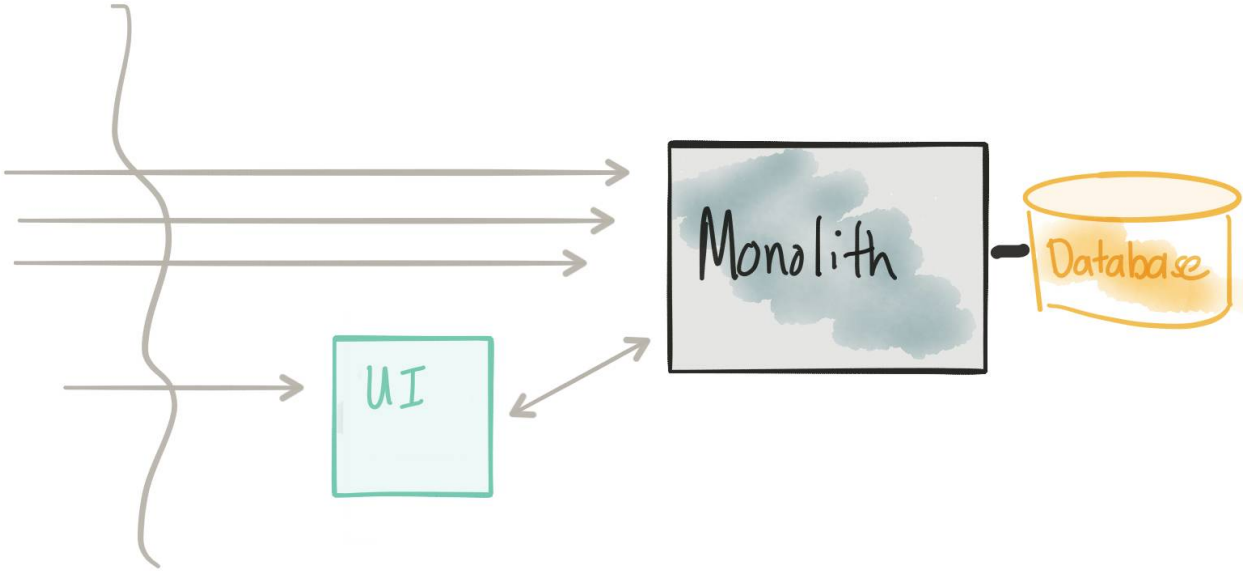
可以用来说明一些关键点。在即将发布的整个主题的第二部分中，我们将深入探讨技术框架或平台。

在这张图中，单体式应用将所有模块/组件/UI共同部署到了一个单体数据库中。当我们试图变更时，就会牵一发而动全身。试想一下，这个应用程序已经使用10多年了，所以现在变更起来难度很大（有技术原因，还有团队或组织结构的原因）。我们希望拆分出UI和关键服务，使业务变更起来更快，更独立，以交付新的客户价值和商业价值。

注意事项

- 单体式应用（代码和数据库模式）很难变更
- 变更需要整个重新部署和团队间高度的协调
- 我们需要进行大量测试来做回归分析
- 我们需要一个全自动的部署方式

四、抽取UI



在这步中，我们将从单体式应用中解耦UI。实际上在这个架构中，我们并未从中删除任何东西。为了降低风险，我们添加了一个包含UI的新部署。这个架构中的新UI组件需要非常接近单体式应用中的同一个UI（甚至完全一致），并调用它的REST API。所以这意味着单体式应用拥有一个合理的API可供外部UI使用。但是，我们可能会发

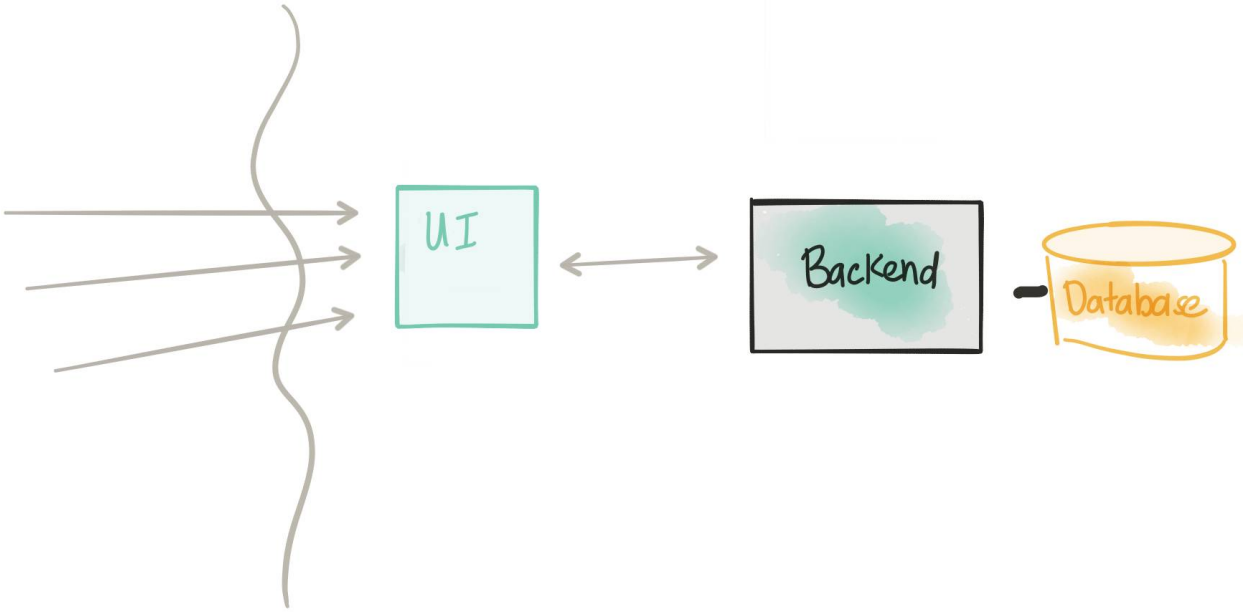
现并不是这么回事：通常这类API可能更像是“内部的”API，这里，我们需要考虑集成单独的UI组件和后端的单体式应用，以及让面向公众的API更可用。

我们可将这个新的UI组件部署到架构中，并使用平台将流量缓慢地路由到这个新架构，同时仍路由一些流量到旧的单体式应用。这样我们就不用停机。同样的，在本主题的第二部分，我们会更详细地看到如何做到这点。无论如何，**灰度上线（dark launch）/金丝雀 发布（canary）/滚动发布（rolling release）**②等概念在这里（以及后续步骤中）都非常重要。

注意事项

- 一开始，先不要变更单体式应用;只需将复制UI或者将它传到单独的组件即可
- 在UI和单体式应用间需要有一个合适的远程API—但并非所有情况下都需要
- 扩大安全面
- 需要用某种方法以受控的方式将流量路由或分离到新的UI或单体式应用，以支持灰度上线（dark launch）/金丝雀测试（canary）/滚动发布（rolling release）

五、从单体式应用中删除UI

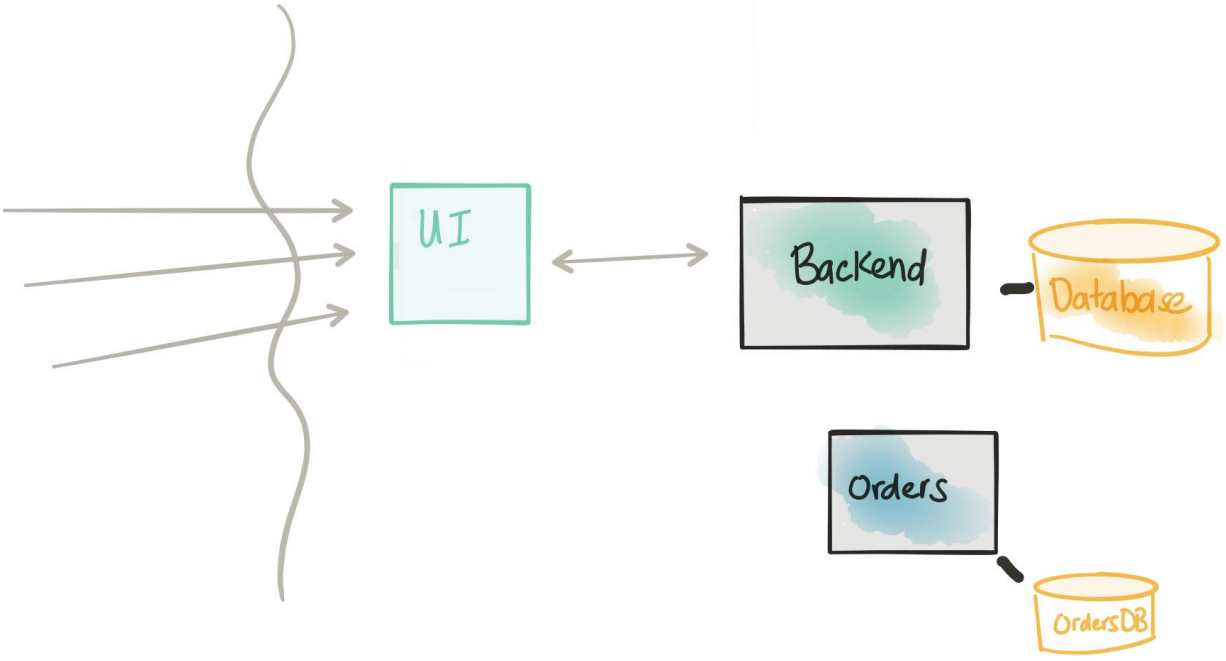


在上个步骤中，我们引入了一个UI，并缓慢地将流量转移到新的UI（它与单体式应用直接通信）。在这一步中，我们将采用一个类似的部署策略，但不同的是，UI被删之后，我们缓慢地发布了一个单体式应用的新部署。如果发现问题，我们可以慢慢地让流量流出，然后回流。在把所有的流量都送到已删除UI的单体式应用（此后称后端-Backend）中，我们就可以完全删除单体式应用部署了。通过分离UI，我们现在已对单体式应用进行了小规模的分解，并依靠灰度上线（dark launch）/金丝雀测试（canary）/滚动发布（rolling release）降低了风险。

注意事项

- 从单体式应用中删除UI组件
- 需要对单体式应用进行最小的变更（弃用/删除/禁用UI）
- 不停机的前提下，再次使用受控的路由/整流方法来引入这种变更

六、引入新的服务



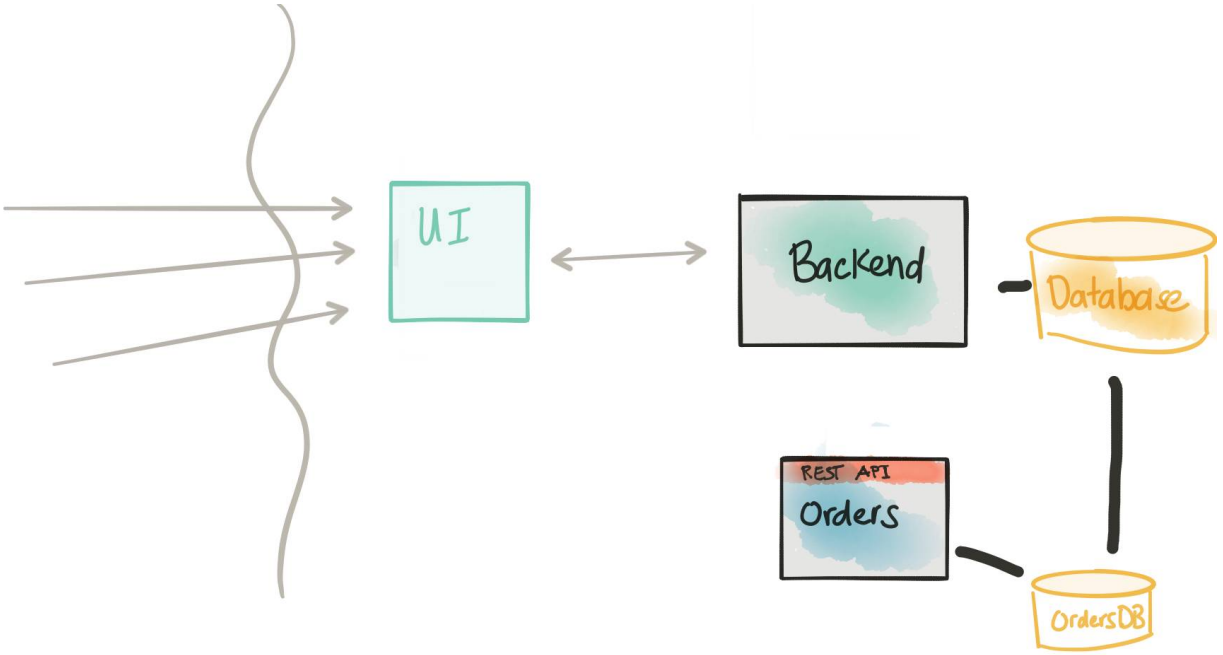
接下来的这步，跳过了耦合、领域驱动设计等细节，我们引入了一项新的服务：Orders服务。在这项关键服务里，业务部分希望比其它应用程序变更的频度更高，但同时它的编写模式相当复杂。我们也可用这个模型来探索CQRS之类的架构模式（跑题了）。

我们要根据现有Backend内的实现来关注Orders服务的边界和API。实际上，这个实现更可能是个重写而不是利用现有代码的端口，但是想法或方法都是相同的。注意在这个架构中，Orders服务有自己的数据库。这点很好，尽管还差那么几步，但离达成一个完整的解耦也已经不远了。接下来还需要考虑以下几个步骤。

同时，这也是考虑该服务在整个服务架构中所处角色的好时机，需要做的是关注于它可能发布或消耗的事件。现在是时候进行事件冲突（Event Storming）这类活动了，并思考在开始处理事务性工作负载时我们该发布的事件。这些事件在集成其它系统甚至在演变单体式应用时，都会派上用场。

- 我们要关注被抽取的服务的API设计或边界
- 可能需要重写单体式应用中的某些内容
- 在确定API后，将为该服务实施一个简单的框架（scaffolding）/place holder
- 新的Orders服务将拥有自己的数据库
- 新Orders服务目前不会承担任何流量

七、将API与实现进行对接



在这里，我们应该继续推演该服务的API和领域模型，以及如何在代码中实现模型。该服务会将新的事务性工作负载存储到其数据库中，并将数据库与其它服务分开。服务访问这些数据时必须经过API。

不能忽视的是：新服务及其数据与单体式应用中的数据关系紧密（虽然在某些地方不完全相同）。实际上这非常不方便。开始构建新服务时，需要来自Backend服务数据库的现有数据的支持。由于数据模型中的标准化、FK约束、关系，这可能会非常棘手。在单体式应用/backend上重用现有API的话，粒度可能过于粗糙，这就需要重新发明一些技巧来获取特定形式的数据。

我们要做的是通过底层API以只读模式从Backend获取数据，并重塑数据以适应新服务的领域模型。在此架构中，我们将连接到后端数据库，并且直接查询数据。这一步需要一个能反映直接访问数据库的一致性模型。

一开始，可能有些人会不敢采用这种方法。但事实是，这方法绝对可行，而且已经有在关键系统中应用成功的案例了。更重要的是，它不是最终架构（**不要认为**它可能成为最终架构）。可能你会认为连接到后端数据库、查询数据和将数据制作成新服务领域模型所需的正确形式，会牵涉到许多不成熟，堆砌而成的代码。但我认为这只是暂时的，所以在单体式应用的演化过程中，这可能是没问题的，也就是说，首先利用技术债，然后再迅速偿还它们。不过，还有个更好的办法。我会在本主题的第二部分讨论。

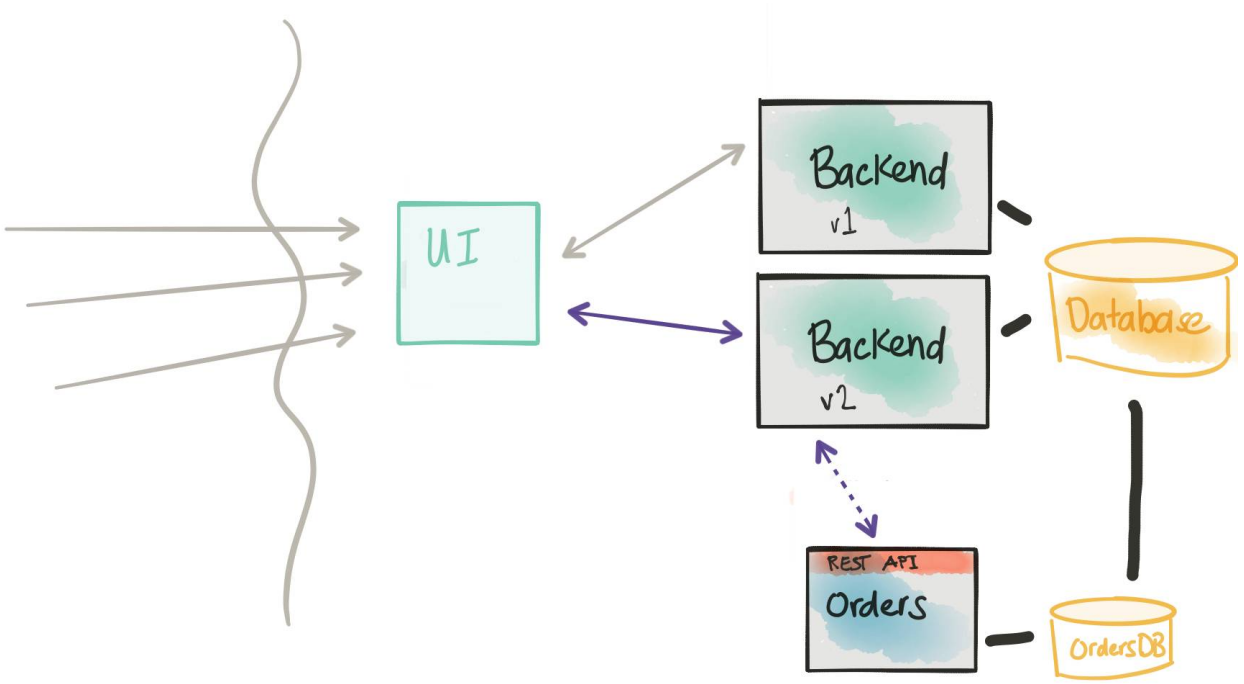
又或者，大家还会说：“好吧，只需要在后台数据库前立个REST API，然后就可以提供更低级的数据访问，再用新的服务调用它”。这也是个可行的方法，但它不是没有缺点。同样的，我也会在第二部分更详细地讨论这点。

注意事项

- 抽取的/新的服务的数据模型按照定义，是与单体式应用数据模型紧密耦合的
- 最可能的情况是，单体式应用提供的API不能在正确级别获取数据
- 即使我们获取了数据，也需要大量的代码样例来改造数据的形式
- 我们可以临时连接到Backend数据库以进行只读查询
- 单体式应用很少改变其数据库

八、发送shadow traffic到

新的微服务（dark launch）



接下来，需要将流量引入到新的微服务。注意，这不是一场重量级的发布。简单地把它扔到生产流量中显然是不行的（特别是考虑到本例中使用了接受订单的“订购（order）”服务！这个过程中我们当然不想产生任何问题！）。虽然更改底层的单体式应用数据库不是件容易的事，但如果可能，您可以小心地去尝试更改单体式应用应用程序，使其调用新的订单服务。如果你不知道哪种方式最好，我强烈推荐你看看**Michael Feather的《有效利用遗留代码》**③。Sprout Method/Class或Wrap Method/Class这样的模式也能帮到你。

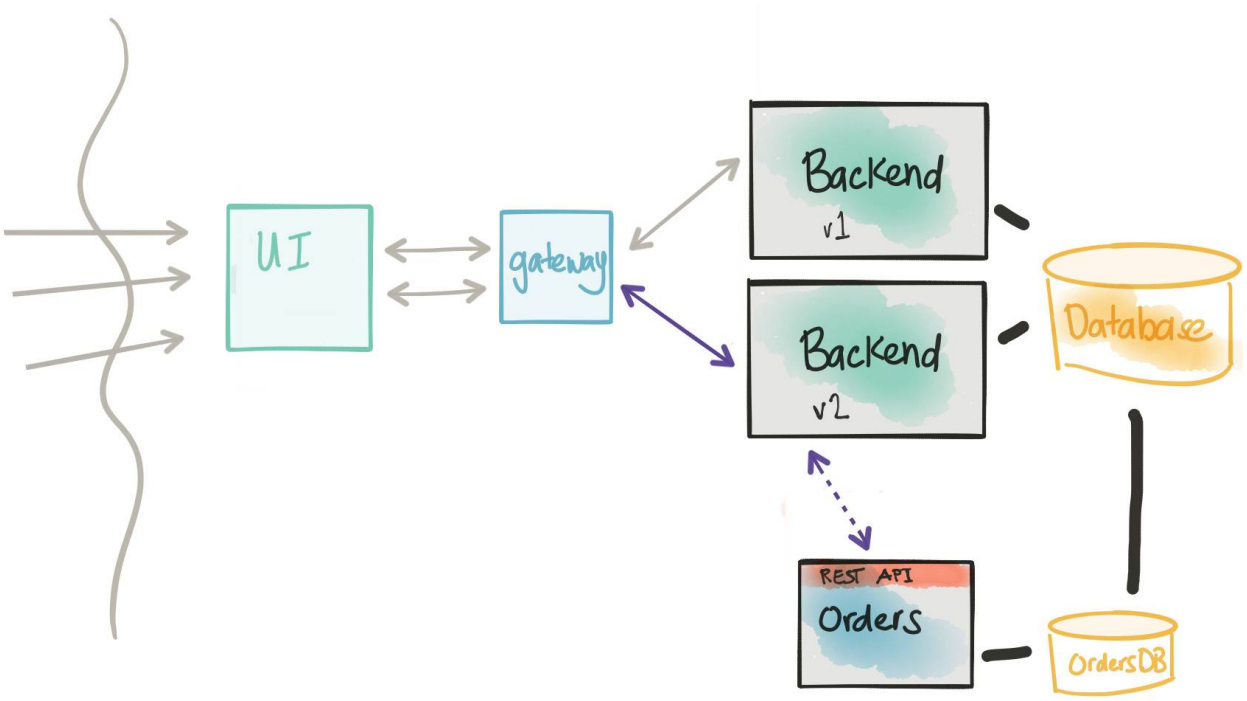
当变更单体式应用/后台时，我们希望保留旧的代码路径。这就需要加入足够的代码，让新旧代码路径都能运行，甚至并行运行。理想情况下，变更后的新版单体式应用应该允许我们在运行时，能选择是将流量发送给新的订单服务、还是使用旧的代码路径，或是两者兼顾。无论采用什么调用路径组合，我们应当了解新旧执行路径之间存在哪些潜在偏差。

另外要注意的是，若允许单体式应用将执行命令发送给旧代码路径以及用于调用新服务，我们需要某种方法来将该新服务的事务或调用标记为“合成（synthetic）”调用。如果你的新服务没有本例那么重要，且可以处理重复内容，那么识别这个合成请求可能就不那么重要。如果你的新服务倾向于更多的为服务于只读流量，可能就不用再识别哪些是合成的事务。然而，在综合交易的前提下，你会希望能够端到端地运行

整个服务，包括存储和数据库。此时您可以选择使用“合成（synthetic）”标志来标记数据并存储，或者在数据存储支持的前提下，回滚该事务。

最后需要注意的是，当我们变更单体式应用/Backend时，我们希望再次使用灰度上线（dark launch）/金丝雀测试（canary）/滚动发布（rolling release）。但基础设施必须支持它才行。在第二部分我们会详细讨论。

在这里，流量被迫回到单体式应用。我们试图不扰乱主要的调用流程，以便当canary无效时能够快速回滚。另一方面，部署网关或控制组件可能会发挥一些作用，它们能以更细的粒度控制对新服务的调用，而不是将调用强加给单体式应用。这种情况下，网关将具备控制逻辑，即能选择是否将事务发送给单体式应用、新服务还是两者都发。

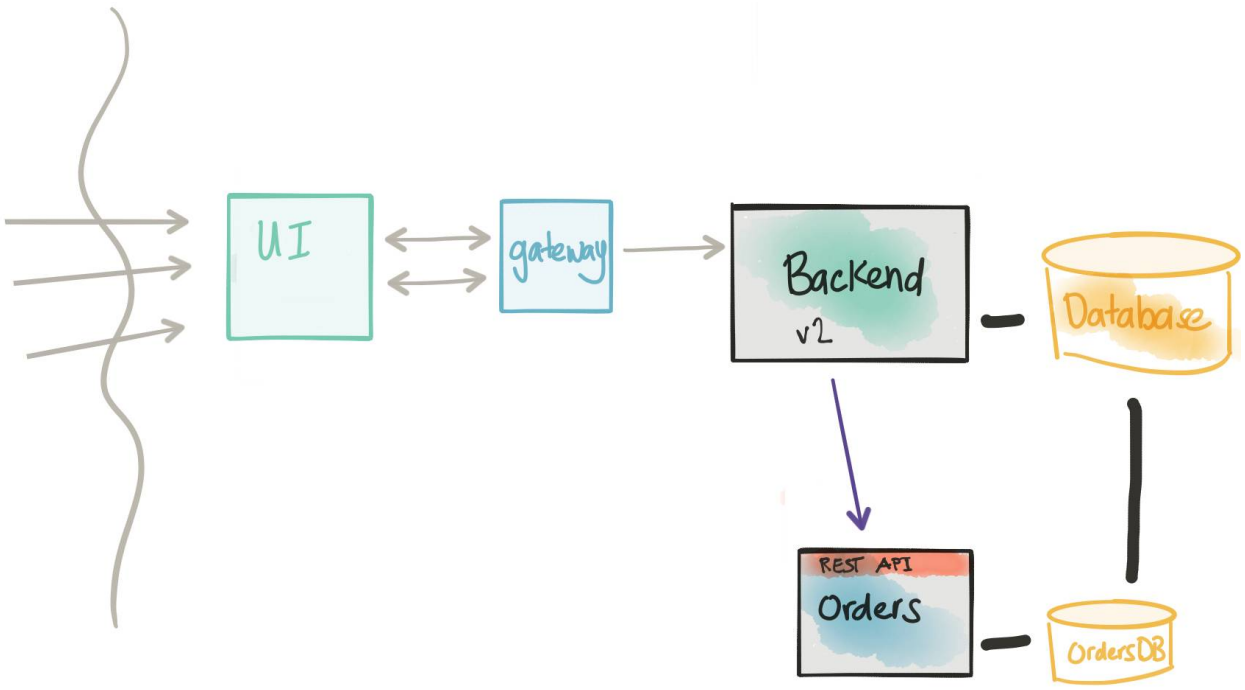


注意事项

- 将新订单服务引入代码路径有风险
- 要以可控的方式将流量发送给新服务
- 希望流量能被引到新服务以及旧代码路径
- 要测量和监控新服务的影响
- 要设法标记“合成（synthetic）”事物，以防发生比较头疼的业务一致性问题

- 希望新功能部署到特定的群组或用户

九、金丝雀测试或滚动发布新的微服务



若前面的步骤不会对事务路径产生不良影响，同时，我们有很大信心能够通过背景流量相关的测试及初期的生产实验，那么现在我们就可以将单体式应用设置为“NOT shadow”，并将流量发送到新的微服务上了。这时，要指定特定的群组或用户，让其始终转入微服务。同时，我们正在慢慢导出那些从旧代码路径通过的真实生产流量。我们可以增加Backend服务的滚动发布频率，直到所有用户都转到新的订单微服务上。

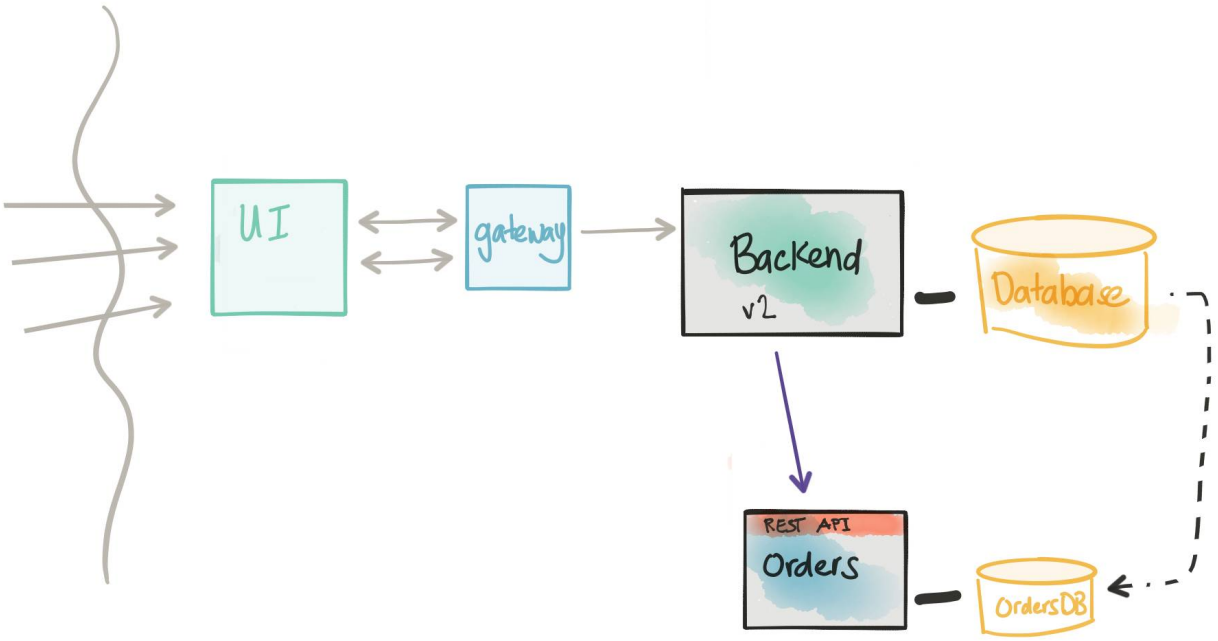
需要提醒一下，这里存在风险：当我们开始将实时流量（非影子或合成流量）滚动到微服务时，期望与群组匹配的用户总是去调用这个微服务。因为我们已经不能在新旧代码路径之间来回切换了。此时，如果我们想要实现回滚，就会牵涉到很多协调，才能使新事务从新业务移回到旧业务单元时也能使用。希望这种情况不会发生，但我们必须有所警惕并事先做好计划，有相应的测试。

注意事项

- 确定群组，并将实时事务流量发送给新的微服务

- 直接连接数据库仍然是需要的，因为在此期间，事务仍会从两条代码路径通过
- 将所有流量转到微服务后，就该放弃旧功能了
- 请注意，在将实时流量发送给微服务后，回滚到旧代码路径将遇到困难，需要协调

十、离线数据ETL/迁移

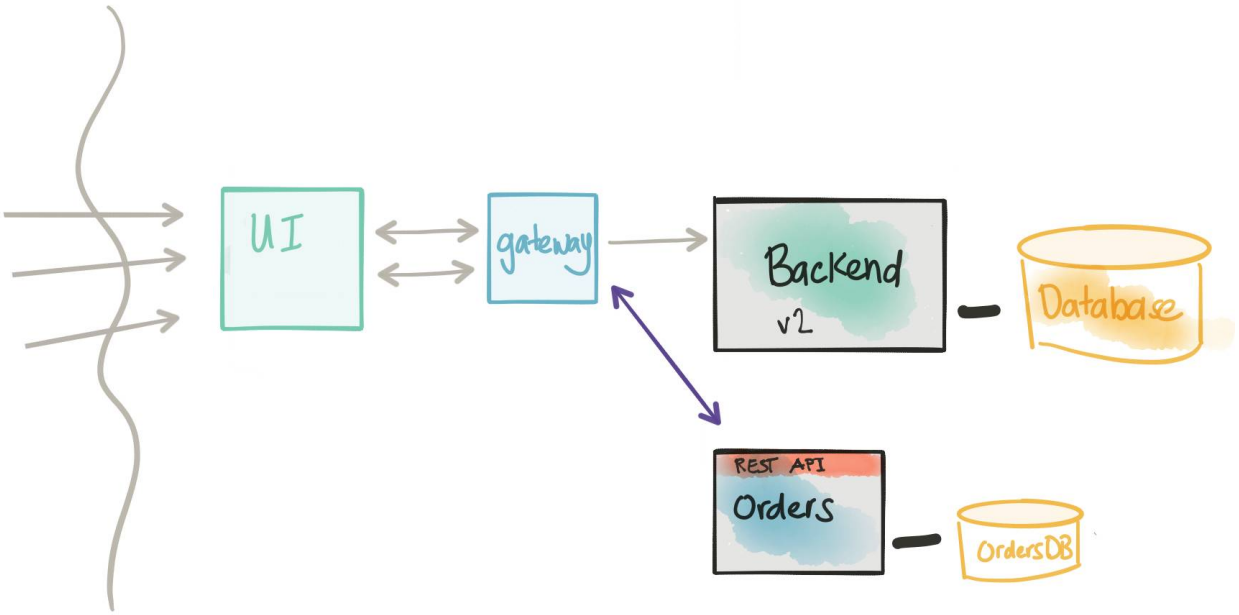


至此，订单微服务开始承载实时的生产流量了。单体式应用或Backend仍然在处理其它需求，但我们已成功地将服务功能迁出了单体式应用。接下来需要迫切关注的是，需要还清新的微服务和Backend服务之间建立直接数据库连接时产生的技术债。这很可能牵涉到从单一数据库到新服务的一次性ETL（提取转换加载）。单体式应用可能仍需要只读式地保存那些数据（比如出于合规的考虑等）。如果它们是共享的引用数据（比如只读的），这么做应该没问题。必须确保单体式应用和新的微服务中，各自的数据不共享。如果它们是的话，那么最终会出现数据或数据所有权的相关问题。

注意事项

- 我们新的订单微服务马上就要完全自治了
- 将订单服务数据库连接到Backend数据库时欠下的技术债，必须还清
- 对留在订单服务中的数据应该实施一次性的ETL
- 要注意各种数据问题

十一、解耦数据存储



完成了上一步，新的订单微服务准备就绪，可以加入到服务架构中去了。本文介绍的步骤都有各自的注意事项和优缺点。我们的目标应该是完成所有步骤，避免技术债产生利息。当然，这种模式与实际操作可能会有差异，但方法没有问题。

在接下来的后续博文中，我将展示如何使用之前提到的示例服务来完成以上步骤，并深入探讨对哪些是有帮助的工具、框架和基础设施。我们会看看Kubernetes、**Istio**④、特性标志框架、数据视图工具和测试框架等内容。请保持关注！

原文链接：

http://blog.christianposta.com/microservices/low-risk-monolith-to-microservice-evolution/

参考地址：

① https://developers.redhat.com/ticket-monster/

② http://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/

③ https://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052

④ https://istio.io/

关于EAWorld：微服务，DevOps，数据治理，移动架构原创技术分享，**长按二维码关注**

10月-11月，PWorld系列技术趴还将继续上演。目前，11月26日将在上海举行PWorld MeetUP“自服务的大数据治理行业实战分享”已启动报名，到场即有机会获得主题相关的技术书籍，戳“阅读原文”可直达报名页面，并了解更多详情~

阅读原文