# 微服务

除非你生活在石器时代，不然你一定知道「微服务」是当世最流行的架构。我们Segment早在2015年就开始实践这一架构。这让我们在一些方面上吃了不少甜头，但很快我们发现：在其他场景，他时不时让我们吃了苦头。

简而言之，微服务的主要宣传点在于：模块化优化，减少测试负担，更好的功能组成，环境独立，而且开发团队是自治的（因为每一个服务的内部逻辑是自洽且独立的）。而另一头的单体应用：「巨大无比且难以测试，而且服务只能作为一个整理来伸缩（如果你要提高某一个服务的性能，只能把服务器整体提高）」

2017 早期，我们陷入了僵局，复杂的微服务树让我们的开发效率骤减，并且每一个开发小组都发现自己每次实现都会陷入巨大的复杂之中，此时，我们的缺陷率也迅速上升。

最终，我们不得不用三个全职工程师来维护每一个微服务系统的正常运行。这次我们意识到改变必须发生了，本文会讲述我们如何后退一步，让团队需要和产品需求完全一致的方法。

## 为什么微服务曾经可行？

Segment 的客户数据基础设施吸收每秒成百上千个事件，将每一个伙伴服务的API 请求结果一个个返回给对应的服务端的「目的地」。而「目的地」有上百种类别，例如Google Analytics， Optimizely，或者是一些自定义的webhook。

几年前，当产品初步发布，当时架构很简单。仅仅是一个接收事件并且转发的消息队列。在这个情况下，事件是由Web或移动应用程序生成的JSON对象，例子如下：

**{**

"type"**:** "identify"**,**

"traits"**:** **{**

"name"**:** "Alex Noonan"**,**

"email"**:** "anoonan@segment.com"**,**

"company"**:** "Segment"**,**

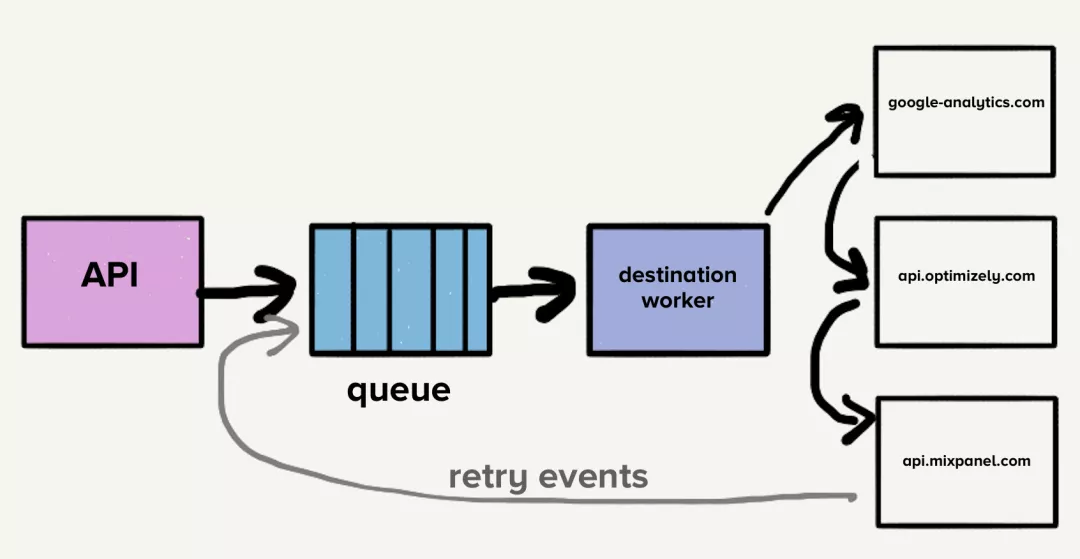
"title"**:** "Software Engineer"

**},**

"userId"**:** "97980cfea0067"**}**

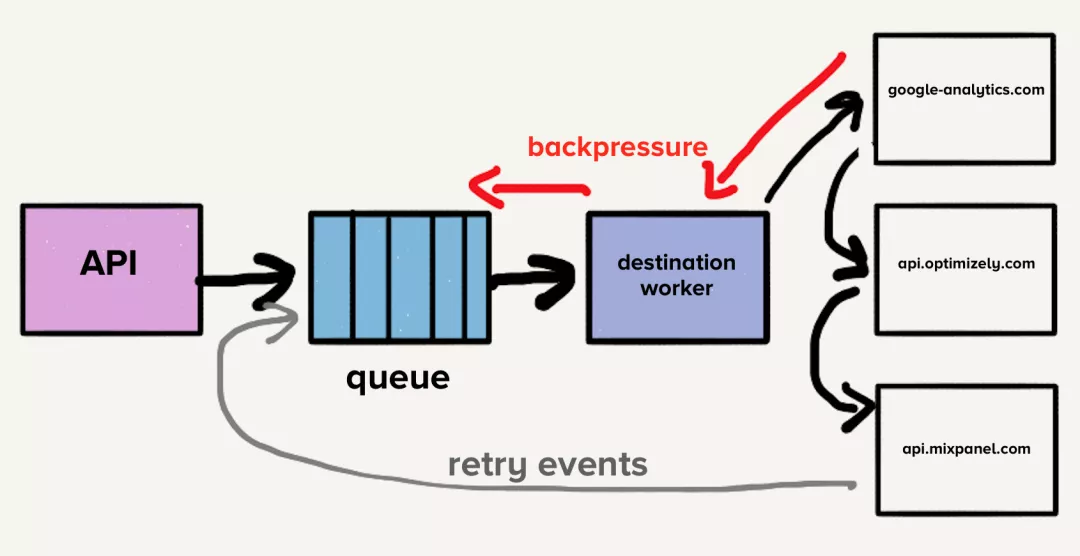
事件是从队列中消耗的，客户的设置会决定这个事件将会发送到哪个目的地。这个事件被纷纷发送到每个目的地的API，这很有用，开发人员只需要将他们的事件发送到一个特定的目的地——也就是Segment 的API，而不是你自己实现几十个项目集成。

如果一个请求失败了，有时候我们会稍后重试这个事件。一些失败的重试是安全的，但有些则不。可重试的错误可能会对事件目的地不造成改变，例如：50x错误，速率限制，请求超时等。不可重试的错误一般是这个请求我们确定永远都不会被目的地接受的。例如：请求包含无效的认证亦或是缺少必要的字段。

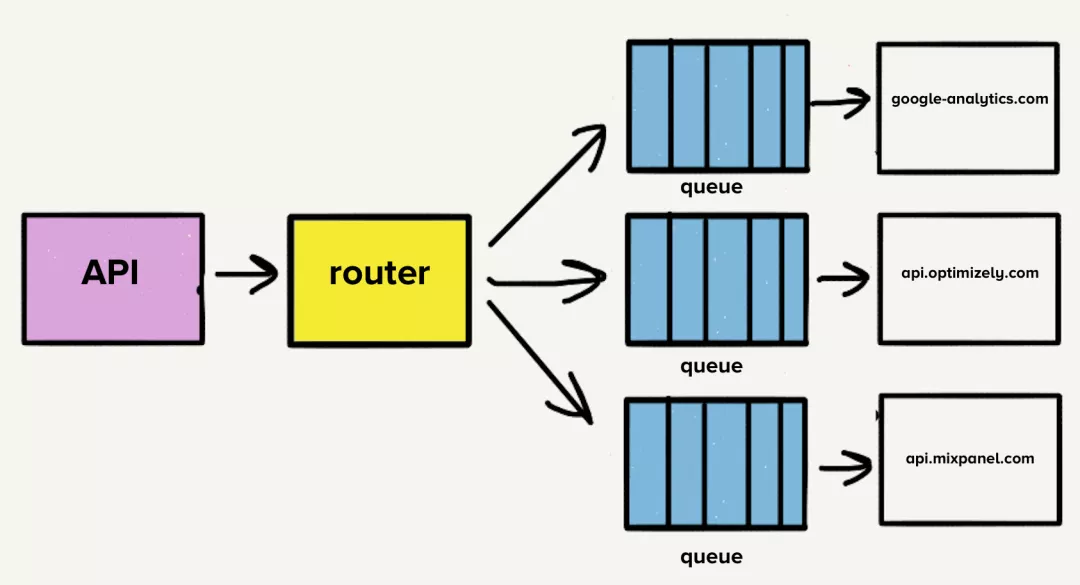


此时，一个简单的队列包含了新的事件请求以及若干个重试请求，彼此之间事件的目的地纵横交错，会导致的结果显而易见：队头阻塞。意味着在这个特定的场景下，如果一个目的地变慢了或者挂掉了，重试请求将会充斥这个队列，从而整个请求队列会被拖慢。

想象下我们有一个 目的地 X 遇到一个临时问题导致每一个请求都会超时。这不仅会产生大量尚未到达目的地 X的请求，而且每一个失败的事件将会被送往重试的队列。即便我们的系统会根据负载进行弹性伸缩，但是请求队列深度突然间的增长会超过我们伸缩的能力，结果就是新的时间推送会延迟。发送时间到每一个目的地的时间将会增加因为目的地X 有一个短暂的停止服务（因为临时问题）。客户依赖于我们的实时性，所以我们无法承受任何程度上的缓慢。



为了解决这个队头阻塞问题，我们团队给每一个目的地都分开实现了一个队列，这种新架构由一个额外的路由器进程组成，该进程接收入站事件并将事件的副本分发给每个选定的目标。现在如果一个目的地有超时问题，那么也仅仅是这个队列会进入阻塞而不会影响整体。这种「微服务风格」的架构分离把目的地彼此分开，当一个目的地老出问题，这种设计就显得很关键了。



## **个人Repo 的例子**

每一个目的地的API 的请求格式都不同，需要自定义的代码去转换事件来匹配格式。一个简单的例子：还是目的地X，有一个更新生日的接口，作为请求内容的格式字段为 dob ，API 会对你要求字段为 birthday，那么转换代码就会如下:

const traits = {}

traits.dob = segmentEvent.birthday

许多现代的目的地终点都用了Segment 的请求格式，所以转换会很简单。但是，这些转换也可能会十分复杂，取决于目的地API 的结构。

起初，目的地分成几个拆分的服务的时候，所有的代码都会在一个repo 里。一个巨大的挫折点就是一个测试的失败常常会导致整个项目测试无法跑通。我们可能会为此付出大量的时间只是为了让他像之前一样正常运行通过测试。为了解决这个问题,我们把每一个服务都拆分成一个单独的repo，所有的目的地的测试错误都只会影响自己，这个过渡十分自然。

拆分出来的repo 来隔离开每一个目的地会让测试的实现变得更容易，这种隔离允许开发团队快速开发以及维护每一个目的地。

## **伸缩微服务和Repo 们**

随着时间的偏移，我们加了50多个新的目的地，这意味着有50个新的repo。为了减轻开发和维护这些codebase 的负担，我们创建一个共享的代码库来做实现一些通用的转换和功能，例如HTTP 请求的处理，不同目的地之间代码实现更具有一致性。

例如：如果我们要一个事件中用户的名字，event.name() 可以是任何一个目的地里头的调用。共享的类库会去尝试判断event 里的 name 或者 Name 属性，如果没有，他会去查 first name，那么就回去查找first\_name 和 FirstName，往下推：last name 也会做这样的事情。然后吧first name 和last name 组合成full name.

Identify.prototype.name = function() {

var name = this.proxy('traits.name');

if (typeof name === 'string') {

return trim(name)

}

var firstName = this.firstName();

var lastName = this.lastName();

if (firstName && lastName) {

return trim(firstName + ' ' + lastName)

}

}

共享的代码库让我们能快速完成新的目的地的实现，他们之间的相似性带给我们一致性的实现而且维护上也让我们减少了不少头疼的地方。

尽管如此，一个新的问题开始发生并蔓延。共享库代码改变后的测试和部署会影响所有的目的地。这开始让我们需要大量时间精力来维护它。修改或者优化代码库，我们得先测试和部署几十个服务，这其中会带来巨大的风险。时间紧迫的时候，工程师只会在某个特定的目的地去更新特定版本的共享库代码。

紧接着，这些共享库的版本开始在不同的目标代码库中发生分歧。微服务起初带给我们的种种好处，在我们给每一个目的地都做了定制实现后开始反转。最终，所有的微服务都在使用不同版本的共享库——我们本可以用自动化地发布最新的修改。但在此时，不仅仅是开发团队在开发中受阻，我们还在其他方面遇到了微服务的弊端。

这额外的问题就是每一个服务都有一个明确的负载模式。一些服务每天仅处理寥寥几个请求，但有的服务每秒就要处理上千个请求。对于处理事件较少的目的地，当负载出现意外峰值时，运维必须手动伸缩服务以满足需求。（编者注，肯定有解决方案，但原作者突出的还是复杂度和成本。）

当我们实现了自动伸缩的实现，每个服务都具有所需CPU和内存资源的明显混合，这让我们的自动伸缩配置与其说是科学的，不如说更具有艺术性（其实就是蒙的）。

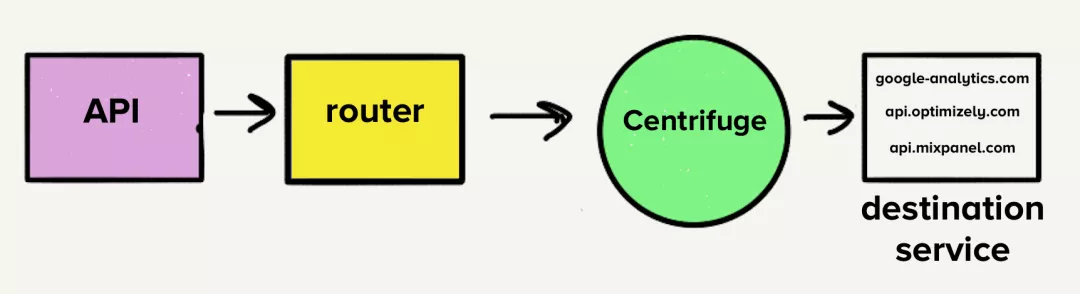
目的地的数量极速增长，团队以每个月三个（目的地）的速度增长着，这意味着更多的repo，更多的队列，更多的服务。我们的微服务架构的运维成本也是线性地增长着。因此，我们决定退后一步，重新考虑整个流程。

## **深挖微服务以及队列**

这时列表上第一件事就是如何巩固当前超过140个服务到一个服务中，管理所有服务的带来的各种成本成了团队巨大的技术债务。运维工程师几乎无眠，因为随时出现的流量峰值必须让工程师随时上线处理。

尽管如此，当时把项目变成单一服务的架构是一个巨大的挑战。要让每一个目的地拥有一个分离的队列，每一个 worker进程需要检查检查每一队列是否运行，这种给目的地服务增加一层复杂的实现让我们感到了不适。这是我们「离心机」的主要灵感来源，「离心机」将替换我们所有的个体队列，并负责将事件发送到一个单体服务。

译者注：「离心机」其实就是Segment 制作的一个事件分发系统。 相关地址



## **搬到一个单体Repo**

所以我们开始把所有的目的地代码合并到了一个repo，这意味着所有的依赖和测试都在一个单一的repo 里头了，我们知道我们要面对的，会是一团糟。

120个依赖，我们都提交了一个特定的版本让每一个目的地都兼容。当我们搬完了目的地，我们开始检查每一个对应的代码是否都是用的最新的依赖。我们保证每一个目的地在最新的依赖版本下，都能正确运行。

这些改变中，我们再也不用跟踪依赖的版本了。所有目的地都使用同一版本，这显著地减小了codebase 的代码复杂度。维护目的地变得快捷而且风险也变小了。

另一方面我们也需要测试能简单快速地运行起来，之前我们得出的结论之一就是：「不去修改共享库文件主要的阻碍就是得把测试都跑一次。」

幸运的是，目的地测试都有着相似的架构。他们都有基础的单元测试来验证我们的自定义转换逻辑是否正确，而且也能验证HTTP 的返回是否符合我们的期望值。

回想起我们的出新是分离每一个目的地的codebase 到各自的repo 并且分离各自测试的问题。尽管如此，现在看来这个想法是一个虚假的优势。HTTP 请求的发送仍然以某种频率失败着。因为目的地分离到各自的repo，所以大家也没有动力去处理这类失败的请求。这也让我们走进了某种令人沮丧的恶性循环。本应只需几个小时的小改动常常要花上我们几天甚至一周的时间。

## **构建一个弹性测试套件**

给目的地发送的HTTP 请求失败是我们主要的失败测试原因，过期凭证等无关的问题不应该使测试失败。我们从中也发现一些目的地的请求会比其他目的地慢不少。一些目的地的测试得花上5 分钟才能跑完，我们的测试套件要花上一小时时间才能全部跑完。

为了解决这个问题，我们制作了一个「Traffic Recorder」，「Traffic Recorder」是一个基于yakbak 实现的工具，用于记录并且保存一些请求。无论何时一个测试在他第一次跑的时候，对应的请求都会被保存到一个文件里。后来的测试跑的时候，就会复用里头的返回结果。同时这个请求结果也会进入repo，以便在测试中也是一致的。这样一来，我们的测试就不再依赖于网络HTTP请求，为了接下来的单一repo 铺好了路。

记得第一次整合「Traffic Recorder」后，我们尝试跑一个整体的测试，完成 140+ 目的地的项目整体测试只需几毫秒。这在过去，一个目的地的测试就得花上几分钟，这快得像魔术一般。

## **为何单体应用可行**

只要每个目的地都被整合到一个repo，那么他就能作为一个单一的服务运行。所有目的地都在一个服务中，开发团队的效率显著提高。我们不因为修改了共享库而部署140+ 个服务，一个工程师可以一分钟内重新完成部署。

速度是肉眼可见地被提升了，在我们的微服务架构时期，我们做了32个共享库的优化。再变成单体之后我们做了46个，过去6个月的优化甚至多过2016年整年。

这个改变也让我们的运维工程师大为受益，每一个目的地都在一个服务中，我们可以很好进行服务的伸缩。巨大的进程池也能轻松地吸收峰值流量，所以我们也不用为小的服务突然出现的流量担惊受怕了。

## **坏处**

尽管改变成单体应用给我们带来巨大的好处，尽管如此，以下是坏处：

故障隔离很难，所有东西都在一个单体应用运行的时候，如果一个目的地的bug 导致了服务的崩溃，那么这个目的地会让所有的其他的目的地一起崩溃（因为是一个服务）。我们有全面的自动化测试，但是测试只能帮你一部分。我们现在在研究一种更加鲁棒的方法，来让一个服务的崩溃不会影响整个单体应用。

内存缓存的效果变低效了。之前一个服务对应一个目的地，我们的低流量目的地只有少量的进程，这意味着他的内存缓存可以让很多的数据都在热缓存中。现在缓存都分散给了3000+个进程所以缓存命中率大大降低。最后，我们也只能在运维优化的前提下接受了这一结果。

更新共享库代码的版本可能会让几个目的地崩溃。当把项目整合的到一起的时候，我们解决过之前的依赖问题，这意味着每个目的地都能用最新版本的共享库代码。但是接下来的共享库代码更新意味着我们可能还需要修改一些目的地的代码。在我们看来这个还是值得的，因为自动化测试环节的优化，我们可以更快的发现新的依赖版本的问题。

## **结论**

我们起初的微服务架构是符合当时的情况的，也解决了当时的性能问题还有目的地之间孤立实现。尽管如此，我们没有准备好服务激增的改变准备。当需要批量更新时，我们缺乏适当的工具来测试和部署微服务。结果就是，我们的研发效率因此出现了滑坡。

转向单体结构使我们能够摆脱运维问题，同时显着提高开发人员的工作效率。我们并没有轻易地进行这种转变，直到确信它能够发挥作用。

我们需要靠谱的测试套件来让所有东西都放到一个repo。没有它，我们可能最终还是又把它拆分出去。频繁的失败测试在过去损害了我们的生产力，我们不希望再次发生这种情况。

我们接受一些单体架构的固有的坏处而且确保我们能最后得到一个好的结果。我们对这个牺牲是感到满意的。

在单体应用和微服务之间做决定的时候，有些不同的因素是我们考虑的。在我们基础设施的某些部分，微服务运行得很好。但我们的服务器端，这种架构也是真实地伤害了生产力和性能的完美示例。但到头来，我们最终的解决方案是单体应用。