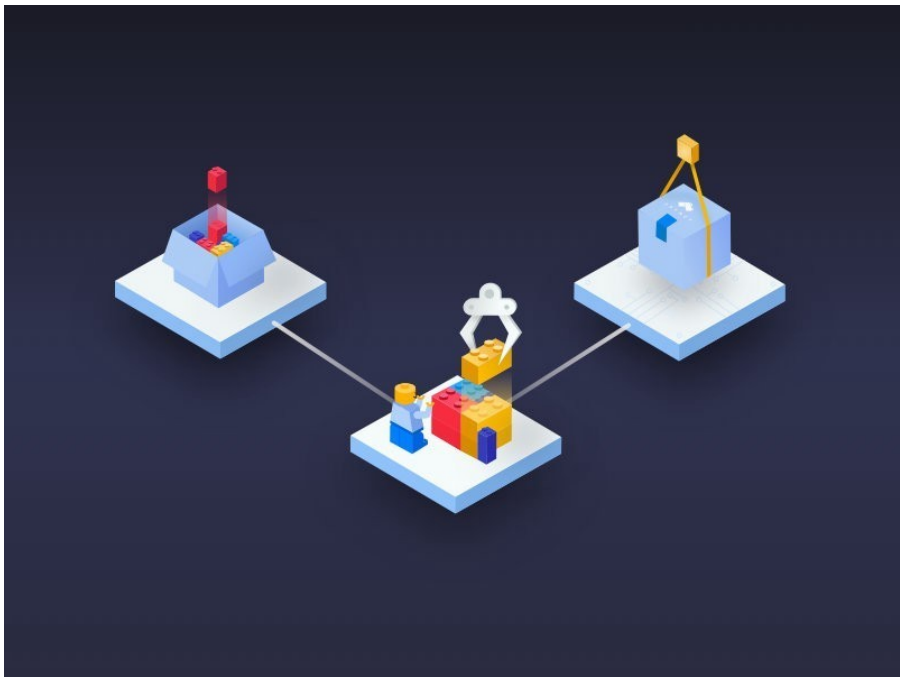
微服务编排

StoneDemo 发表于 小石不识月



在 Jexia 中，我们相信微服务架构是组织我们的后端云的最佳方式 —— 它可以很好地进行关注分离（Separation of concerns），并为特定任务提供明确的职责边界。通过使用微服务架构，我们可以根据每一后端应用程序的特定部分的负载，对其进行缩放（Scale）。最后同样重要的一点是，对我们开发人员来说，拥有自己的 “偏好型项目（Pet project）” 是很好的，在这种项目中我们可以按照自己的方式进行构建。

这非常棒，但在实践中，事情就开始变得更加复杂了：对于某些任务，我需要调用哪个服务来执行呢？我的任务涉及到以特定的顺序使用多个服务，但是要以何种顺序使用呢？如果其中有一个操作失败了，我们该如何回滚呢？

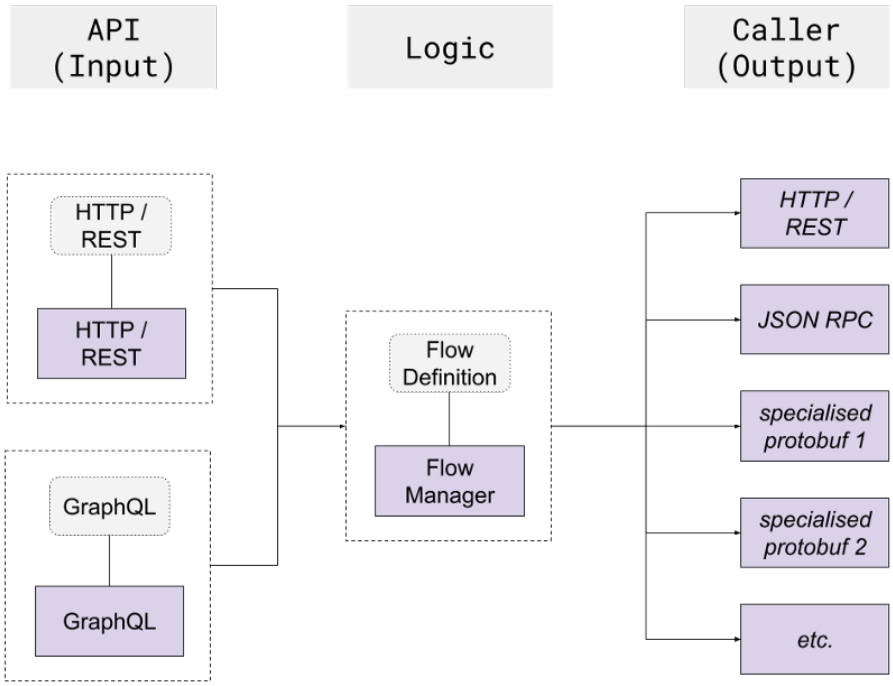
当然，我们不希望我们的前端和 SDK（软件开发程序包）团队被这些琐碎的事情困扰。还有一些 '假设' 的情景，其中文档尚未编写完整 —— 但这又是一个潜在的绊脚石。

因此我们引入了一个编排服务（Orchestration service）。对此编排服务的单次调用会引发对后端微服务的一个或多个请求。这种编排服务需要是快速的、简单的、动态的、小型的、可配置的、易于使用的，以及可运转的等等。基本上没有任何权衡（Trade-off），并且所有好东西已包括在内。我们还选择使用 Golang 语言编写它，以防止后端团队出现混淆，因为其他所有的服务都已经用 Golang 编写了。

我们非常喜欢 Netflix Conductor 的想法，但对于我们的预期用例（并且不是用 Golang 编写的）来说，这个想法过于庞大又复杂。一些框架和库提供了（一部分）我们的需求，但它们缺乏灵活性，复杂性，质量等等。所以我们必须拥有自己的方式。

设计选型

我们的主要设计目标是灵活性 —— 我们希望避免一些不得不修改代码的情况（每次添加新服务，或有新的请求新任务时）。下图所显示的模块化设计或许是我们的最佳选择。



Microservice Orchestrator 的架构

它包含三个层级：

- **API（输入）模块：**这一层将系统与外界连接起来。此处实现了 HTTP REST、GraphQL、Web Sockets 等协议 —— 无论需要什么，都有！这些请求随后都会以内部格式转发给逻辑模块。
- **逻辑模块：**这一部分将转换后的请求转变为对微服务的一次或多次调用。然后收集调用结果，并将其回报给 API 模块。
- **调用者（输出）模块**：此处会处理与实际微服务的交互。交互由调用者的逻辑执行，并提供所需的参数和细节。每个调用者会处理自己的协议。 这个协议可以是通用的协议，例如用于与多个服务通信的 HTTP JSON 协议，或使用一个专用协议（例如使用 protobuf）与特定服务进行通信。

逻辑流程

逻辑层可以使用称为 Flows（流，以下都用英文表述） 的自定义 DSL 进行配置（没错，我们的创意是最好的！），我们可以随时读取它以更新逻辑（目前我们只在启动时读取它）。Flow DSL 由 ***服务***、***中间件***，以及 ***流程*** 模块组成，如下所示。

```
flows {
    service "security" {
        identifier: "auth"
    }
    service "item-manager" {
        identifier: "http"
        address: "http://item-manager.service.cloud/"
    }
    service "image-service" {
```

```
        identifier: "http"
        address "http://images.micro.cloud/"
    }
    service "data-store" {
        identifier: "json-rpc"
        name: "DataStoreService"
    }
    middleware "Authorize Token" {
        // returns an error when token is not valid,
        // aborting the rest of the flow
        service = "security"
        method = "TokenValid"
        request = {
            "token" = "{{{input_token}}}"
       }
    }
    flow "item details" {
        input = ["id", "token"]
        middleware = ["Authorize Token"]
        order = ["details", "image URL"]
        call "details" {
            type = "fork"
            call "basic details" {
                service = "item-manager"
                method = "Item.Details"
                request = {
                    "id": "{{{input_id}}}"
                }
                response = ["name", "description", "imageId"]
            }
            call "relations" {
                service = "data-store"
                method = "Item.Relations"
                request = {
                    "id": "{{{input_id}}}"
                }
                response = ["relatedItems"]
            }
        }
        call "image URL" {
            type="sync"
            service = "image service"
            method = "URL"
            request = {
                "id": "{{{basic details:imageId}}}"
            }
            response = ["imageURL", "width", "height"]
        }
    }
    output = {
        "name": "{{{basic details:name}}}"
        "description": "{{{basic details:description}}}"
        "image": "{{{image URL:URL}}}"
        "relatedItems": "{{{relations:relatedItems}}}"
    }
}
```

请注意，示例中有些字段没有展示出来，如超时、重试等等，这是为了缩减篇幅。

这些服务对调用者（我猜，我们可以更类似地命名它们）进行配置，并将它们提供给 Flows。每个服务都可能会包含提供给调用者的参数，以便它与后端服务进行通信。

Flows DSL 的主要部分是 Flows。它定义了为完成某一特定任务，我们应该执行的服务调用。在紧接着调用列表的，是它们的一些属性。这些属性包括所需的输入和输出、所需的中间件、调用顺序，以及超时信息等等。

每个调用都有一个类型，目前我们支持 ***同步（Synchronous）*** 和 ***分叉（Forked）*** 调用。同步调用就是，在开始下一个调用之前，将会等待直到上一次的调用结束为止。分叉调用则会同时产生多个子调用，并等待所有调用完成（即 Joined，主线程等待所有子线程执行结束。因此命名为分叉）。

正如 Flows 一样，调用也在类型之后紧跟着一组属性。这其中包括从其他调用传递的请求和响应参数、关于如何调用服务的细节、超时信息等。为了获得更大的灵活性，我们在请求参数中加入了基于 Mustache 的模板引擎。它可以将静态值与一个或多个变量组合在一起。

从根本上来说，中间件其实是一个可以重复使用的同步调用，能使所有需要它们的流都可以使用一些常见任务。例如，检查处理非公开信息的所有 Flows 是否需要包含有效令牌。此任务将会大量使用在应用程序中，用户需要登录才能访问某些功能或信息。在我们的例子中，中间件的调用流程总是在它的常规调用列表之前执行。

Flows DSL 有点广泛（或者说，有些长了），但在读取到内存中后，它会被转换为数组和映射（Map），以便快速查找与执行。目前所选的格式是为了让开发人员尽可能简单地使用。由于这是人类可读的格式，我们可以轻易地从其内容中明白 Flow 中发生了什么。这使得我们无需深入研究代码就可以轻松地调整 Flow。

API 供应商

如前所述，API 的主要任务是将请求与支持的协议的 Flow 匹配。

例如，HTTP REST API 可识别到 `https://myapp.jexia.com/item/1` ，并将其与 *项目详细信息* 的 Flow （它收集来自多个服务的信息）进行匹配。然后，将这些信息作为单一资源呈现给 API，API 则将响应反馈给用户。另一个 API（例如 GraphQL 的实现），它处理 `https://myapp.jexia.com/grapql?query={item(id:1){name,description,image,relatedItems}` ，同时也可以激活 *项目详细信息* 的 Flow。（是的，我知道仅抓取请求的数据会更有效率，但这只是一个简单的例子。我想不出更合理的东西了。）。

当 Flow 完成时，其输出被 API 接收。然后，它们被用于为其实现的协议创建适当的响应格式。

API 模块通常会打开自己的侦听接口，以最适合的协议、优先级和其他需求的方式处理传入的请求。

每个 API 模块都有自己的配置文件，其中包含所有的路由以及 Logical Flows（逻辑流）的映射。这样，添加新的路由或者 Flow 就不需要添加更多代码、重新编译等。在我们的用例下，这些都可以通过修改配置来完成。请注意，这与 Flows DSL 是相似的。

服务调用者

服务调用者通过 Flow 获取信号，用以向实际的微服务发送请求并等待响应。不同的协议有不同的调用者。调用者可以是通用的，就像 HTTP JSON 调用者，它可以与任何接受 JSON 请求的 HTTP 服务进行通信。调用者也可以是特定的，如使用 protobuf 定义与服务进行通信的调用者。

调用者的类型可以根据项目需求（出于安全原因，服务与编排器紧密耦合）以及需要处理的情况（例如现有服务的可用性）进行选择。

路线图 / 引人深思之事

我们所描述的 Flows 目前正在实现，并进行内部测试。

当然，我们也注意到，通过一些附加功能，它可以成为更好的编排器。文章的最后部分描述了其中的一些内容。

实时通信

提供一个通道（Channel）而不是实际结果，这为我们的后端开辟了新的可能性。更新图表、健康/状态概述、应用程序/网站活动等，这些都只是我想到的一些潜在特性。

除了这些无足轻重的例子之外，它对于长时间运行的工作来说也是很好的，例如处理这样一个请求 —— 统计您在应用程序中获得的所有资金。

通常情况下，请求并不会在几分钟内打开（是的，我们有非常快的硬件，所以统计所有的钱不需要几周时间），取而代之的是打开通道，它非常适合用于过一段时间再报告结果。

从人类的角度来看，等待一分钟可能太长了。然而，编排器可能会被其他不介意等待的应用程序和服务使用。

开放的通道通过编排器（或者直接）将客户端连接到（后端）服务。此处的决策受到下列因素影响：

- 数据需要过滤（安全性，客户利益等等）
- 该服务真正地支持实时通信
- 该服务是一个消息队列
- 诸如此类

除了便利性以外，这还可以减少后端和编排器的负载。其原因是，请求的更新现在是被推送的，而不是后端反复轮询。因此，终端用户会拥有更愉快的体验。

基准

纵观我们的设计，简单和疯狂的编程技巧，我们认为我们的编排器将会有极佳的表现。但我们的 CTO（以及我认为其他开发团队也是一样）想要证明……（现在，你的同事的信任在哪里？！）

因此，一些基准测试计划用于衡量性能，稳健性和可伸缩性等等。

目前我们尚未提供相关的详细信息，因为这些仍处在规划阶段。不过，我们承诺会在这些机制可用的时候，提供一篇关于它们的好文章。

而现在，你必须信任我们的能力（或者，如果有必要的话，你不信的话，请保持跟进这个博客……或者顺道拜访办公室来喝杯咖啡）。

原子性 / 事务

该机制规定，所有的服务调用都应该是 —— 成功的，或者什么都不应该改变的。通过引入这一机制，我们确保我们的微服务环境不会由于架构的分布式特性而被打断。

例如，如果一个 Flow 包含了 3 个调用，其中第一个调用成功，而第二个调用失败：此时第一个调用需要回滚（或尚未永久化），第三个调用则不应调用。

这是一个非常复杂的 *挑战*，对此我们有一些想法，不同于手动修复数据库（合适的解决方案，因为没有过错误/疯狂的编码技巧，所以这并 *不会* 要求一大群无人机或间接任何其他形式的开销）的共识算法（大量的开销……）。第一个已经实现（包括一些无人机，但它们只能飞行，目前情况良好），如果所有其他解决方案都不按要求运行，后者将被实现。

一种更可能的中间方法是令我们的平台足够强大，以便在不中断的情况下处理部分更改的数据。

这可以通过更新时态数据库或表格，并在所有服务报告其成功时切换……或通过以 “向后兼容” 的方式进行更改来完成。正如你所看到的，我们目前正在尝试一些想法。

监测，日志记录以及追踪

持续追踪后端的使用情况，在出现故障时提供信息（这不太可能是我们编码导致的问题，但 CTO 需要它……），或在不希望出现的情况下做出响应，这是非常重要的。现在，微服务编排器从根本上就是内部云与公共世界之间的通道。这使它成为了添加这些功能的一个非常方便的所在。你不会感到惊讶，这正是我们打算做的。

传入逻辑层中的任务使用同样的（内部）格式，因此以特定格式记录它们很容易。如果有需要，可以在 API 模块中监控实际请求和潜在的不当行为。随后，这些日志（事件）可以用来自动响应不希望的不当行为。这些行为的范围，可以从限制速率或吞吐量直到完全拒绝访问。

这种格式也可用于添加追踪信息，因此可以轻松地对每个请求过滤所有并发请求和操作的日志消息。我们希望在调试这个并发环境中的一些（罕见）问题时，它能给我们缩短许多调试时间。

最后一点重要内容是，这些信息可用于展示办公室仪表盘上的应用程序使用情况。这将使开发人员更有动力.....

总结

如果阅读了这篇文章而不会昏昏欲睡，那么我现在有一个内幕消息要告诉你 —— 请确保 *密切关注这个博客*，因为我们可能会开源我们的微服务编排器以供大家使用（很快就可以了 ;-))

发表于 2018-07-04

微服务

上一篇：妈呀，女生换个季有这么多戏吗?
下一篇：微信小游戏将变「大游戏」，「高抽成」超苹果该被吐槽吗?

相关文章

来自专栏 嵌入式程序猿

Bootloader需要你的精心设计

嵌入式产品，我们一般都需要一个bootloader来更新固件和修复bug，一般常用的接口有，UART, CAN, USB, Ethernet，有的还有无线接口，...

127 3

来自专栏 小狼的世界

基于Mapabc API的周边查询应用

现在，越来越多的 Location Based 应用，或者Geolocation的应用出现在网络、手机等各种各样的终端上，为人们的日常生活、出行和工作都提供了不...

95 2

来自专栏 张善友的专栏

9

分享

Google SiteMap Protocol协议

在新浪看到这样的新闻Google雅虎微软联手支持网页手工提交标准, Google、微软和雅虎认为，统一标准有助于从整体上改进站点地图，从而搜索引擎可以将更广泛的...

221 10

来自专栏 黑白安全

ASLRay：一个可以绕过ASLR的工具

ASLR（Address Space Layout Randomization，即地址空间格局随机化）是指利用随机方式配置数据地址，一般现代系统中都加设这一机制...

64 1

来自专栏 Golang语言社区

Redis在游戏服务器中的应用

排行榜 游戏服务器中涉及到很多排行信息，比如玩家等级排名、金钱排名、战斗力排名等。 一般情况下仅需要取排名的前N名就可以了，这时可以利用数据库的排序功能，或者自...

476 12

来自专栏 野路子程序员

徒手解剖composer，简单了解其实现过程

289 6

来自专栏 進无尽的文章

聊聊程序设计思想之面向接口编程IOP

我们在一般实现一个系统的时候，通常是将定义与实现合为一体，不加分离的，但是有时候最为理想的系统设计规范应是所有的定义与实现分离，尽管这可能对系统中的某些情况有点...

117 2

来自专栏 张善友的专栏

beagle MONO 应用的desktop search

beagle是linux的desktop search软件，跟winows下的google desktop search类似的东西，它可以搜索各种各...

196 7

来自专栏 FreeBuf

ASLRay：一个可以绕过ASLR的工具

ASLR（Address Space Layout Randomization，即地址空间格局随机化）是指利用随机方式配置数据地址，一般现代系统中都加设这一机制...

227 8

来自专栏 大数据人工智能

ZStack--工作流引擎

在IaaS软件中的任务通常有很长的执行路径，一个错误可能发生在任意一个给定的步骤。为了保持系统的完整性，一个IaaS软件必须提供一套机制用于回滚先前的操作步骤。...

472 4