微服务架构及设计模式

吴佳兴 译 分布式实验室 2020-02-06

微服务能够对企业产生积极影响。因此，了解如何处理微服务架构（MSA）以及一些微服务设计模式，一个微服务架构的一些通用目标或者设计原则是很有价值的。下面是在微服务架构方案中值得考虑的四个目标。

1. 缩减成本：MSA 将会降低设计、实现和维护IT服务的总体成本
2. 加快发布速度：MSA 将会加快服务从想法到部署的落地速度
3. 增强弹性：MSA 将会提升我们服务网络的弹性
4. 开启可见性：MSA 支持为服务和网络提供更好的可见性

你需要了解建设微服务架构背后的几个设计原则：

- 可扩展性
- 可用性
- 韧性
- 灵活性
- 独立自主性，自治性
- 去中心化治理
- 故障隔离
- 自动装配
- 通过 DevOps 持续交付

听取上述原则，在你实施的解决方案或系统付诸实践的同时，这也会带来一些挑战和问题。这些问题在许多解决方案中也很常见。使用正确及匹配的设计模式可以克服这些问题。微服务有

一些设计模式，这可以大体分为五类。每类都包含许多具体的设计模式。下图展示了这些设计模式。

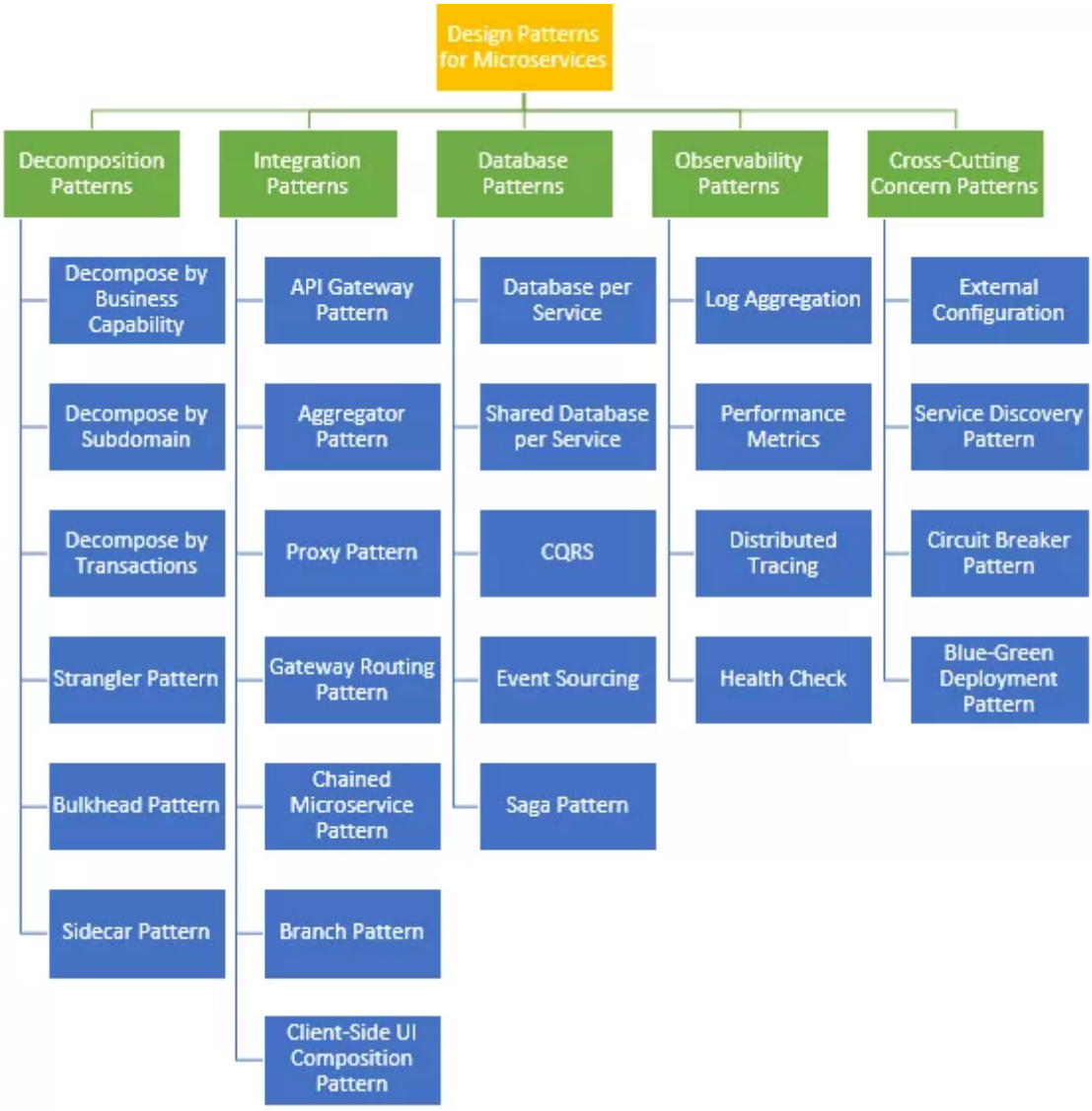

图1 微服务设计模式

分解模式

按业务功能进行分解

说白了，微服务就是要应用单一职责原则，把服务改造成松耦合式的。它可以按照业务功能进行分解。定义和业务功能相对应的服务。业务功能是一个来自业务架构建模的概念。它是一个企业为了创造价值而要去做的某些事情。一个业务功能往往对应于一个业务对象，比如：

- 订单管理负责订单
- 客户管理则是负责客户

按问题子域进行分解

按照业务功能来分解一个应用程序可能会是一个不错的开始，但是你终将会遇到所谓的“神类”，它很难再被分解。这些类将在多个服务之间都是通用的。可以定义一些和领域驱动设计（DDD）里面的子域相对应的服务。DDD 把应用程序的问题空间 —— 也即是业务 —— 称之为域。一个域由多个子域组成。每个子域对应业务的各个不同部分。

子域可以分为如下几类：

- 核心 —— 业务的核心竞争力以及应用程序最有价值的部分
- 支撑 —— 和业务有关但并不是一个核心竞争力。这些可以在内部实现也可以外包
- 通用 —— 不特定于业务，而且在理想情况下可以使用现成的软件实现

一个订单管理的子域包括：

- 产品目录服务
- 库存管理服务
- 订单管理服务
- 配送管理服务

按事务/两阶段提交（2pc）模式进行分解

你可以通过事务分解服务。然后，这样一来系统里将会存在多个事务。事务处理协调器是分布式事务处理的重要参与者之一。分布式事务包括两个步骤：

- 准备阶段 —— 在这个阶段，事务的所有参与者都准备提交并通知协调员他们已准备好完成事务
- 提交或回滚阶段 —— 在这个阶段，事务协调器向所有参与者发出提交或回滚命令

2PC 的问题在于，和单个微服务的运行时间相比，它显得相当慢。即便这些微服务跑在相同的网络里，它们之间的事务协调也确实会减慢系统速度，因此这种方法通常不适用于高负载情况。

绞杀者模式（Strangler Pattern）

上面三种，我们看到的这几个设计模式都是用来分解绿场（Greenfield）的应用程序，但是往往我们所做的工作中有 80% 是针对灰场（brownfield）应用程序，它们是一些大型的单体应

用程序（历史遗留的代码库）。绞杀者模式可以解决这类问题。它会创建两个单独的应用程序，它们并排跑在同一个 URI 空间里。随着时间的流逝，直到最后，新重构的应用程序会“干掉”或替换原有的应用程序，此时就可以关掉那个老的单体应用程序。绞杀应用程序的步骤分别是转换，共存和消除：

- 转换（Transform） —— 使用现代方法创建一个并行的全新站点。
- 共存（Coexist） —— 让现有站点保留一段时间。把针对现有站点的访问重定向到新站点，以便逐步实现所需功能。
- 消除（Eliminate） —— 从现有站点中删除旧功能。

隔舱模式（Bulkhead Pattern）

让一个应用程序的元素和池子相对隔离，这样一来，其他应用程序将可以继续正常工作。这种模式被称为“隔舱”，因为它类似于船体的分段分区。根据使用者负载和可用性要求，将服务实例分成不同的组。这种设计有助于隔离故障，并允许用户即使在故障期间仍可为某些使用者维持服务。

边车模式

该模式将一个应用程序的组件部署到一个单独的处理器容器里以提供隔离和封装。它还允许应用程序由异构的组件和技术组成。这种模式被称为边车模式（Sidecar），因为它类似于连接到摩托车的侧边车。在该模式中，侧边车会附加到父应用程序，并为该应用程序提供功能支持。Sidecar 还与父应用程序共享相同的生命周期，并与父应用程序一起创建和退出。Sidecar 模式有时也称为 sidekick 模式，这是我们在文章中列出的最后一个分解模式。

集成模式

API 网关模式

当一个应用程序被分解成多个较小的微服务时，这里会出现一些需要解决的问题：

- 存在不同渠道对多个微服务的多次调用
- 需要处理不同类型的协议
- 不同的消费者可能需要不同的响应格式

API 网关有助于解决微服务实现引发的诸多问题，而不仅限于上述提到的这些。

- API 网关是任何微服务调用的单一入口点
- 它可以用作将请求路由到相关微服务的代理服务
- 它可以汇总结果并发送回消费者
- 该解决方案可以为每种特定类型的客户端创建一个细粒度的 API
- 它还可以转换协议请求并做出响应
- 它也可以承担微服务的身份验证/授权的责任

聚合器模式（Aggregator Pattern）

将业务功能分解成几个较小的逻辑代码段后就有必要考虑如何协同每个服务返回的数据。不能把这个职责留给消费者。

聚合器模式有助于解决这个问题。它讨论了如何聚合来自不同服务的数据，然后将最终响应发送给消费者。这里有两种实现方式：

1. 一个组合微服务将调用所有必需的微服务，合并数据，然后在发送回数据之前对其进行转换合成
2. 一个 API 网关还可以将请求划分成多个微服务，然后在将数据发送给使用者之前汇总数据

如果要应用一些业务逻辑的话，建议选择一个组合式的微服务。除此之外，API 网关作为这个问题的解决方案已经是既定的事实标准。

代理模式

针对 API 网关，我们只是借助它来对外公开我们的微服务。引入 API 网关后，我们得以获得一些像安全性和对 API 进行分类这样的 API 层面功能。在这个例子里，API 网关有三个 API 模块：

1. 移动端 API，它实现了 FTGO 移动客户端的 API
2. 浏览器端 API，它实现了在浏览器里运行的 JavaScript 应用程序的 API
3. 公共API，它实现了一些第三方开发人员需要的 API

网关路由模式

API 网关负责路由请求。一个 API 网关通过将请求路由到相应的服务来实现一些 API 操作。当 API 网关接收到请求时，它会查询一个路由映射，该路由映射指定了将请求路由到哪个服

务。一个路由映射可以将一个 HTTP 方法和路径映射到服务的 HTTP URL。这种做法和像 NGINX 这样的 Web 服务器提供的反向代理功能一样。

链式微服务模式（Chained Microservice Pattern）

单个服务或者微服务将会有多级依赖，举个例子：Sale 的微服务依赖 Product 微服务和 Order 微服务。链式微服务设计模式将帮助你提供合并后的请求结果。microservice-1 接收到请求后，该请求随后与 microservice-2 进行通信，还有可能正在和 microservice-3 通信。所有这些服务都是同步调用。

分支模式

一个微服务可能需要从包括其他微服务在内的多个来源获取数据。分支微服务模式是聚合器和链式设计模式的混合，并允许来自两个或多个微服务的同时请求/响应处理。调用的微服务可以是一个微服务链。分支模式还可用于根据你的业务需求调用不同的微服务链或单个链。

客户端UI组合模式

通过分解业务功能/子域来开发服务时，负责用户体验的服务必须从多个微服务中提取数据。在一个单体世界里，过去只有一个从 UI 到后端服务的调用，它会检索所有数据然后刷新/提交 UI 页面。但是，现在不一样了。对于微服务而言，我们必须把 UI 设计成一个具有屏幕/页面的多个板块/区域的框架。每个板块都将调用一个单独的后端微服务以提取数据。诸如 AngularJS 和 ReactJS 之类的框架可以帮助我们轻松地实现这一点。这些屏幕称为单页应用程序（SPA）。每个团队都开发一个客户端 UI 组件，比如一个 AngularJS 指令，该组件实现其服务的页面/屏幕区域。UI 团队负责通过组合多个特定服务的 UI 组件来实现构建页面/屏幕的页面框架。

数据库模式

给微服务定义数据库架构时，我们需要考虑以下几点：

1. 服务必须是松耦合的。这样它们可以独立开发，部署和扩展
2. 业务事务可能会强制跨越多个服务的不变量
3. 一些业务事务需要查询多个服务的数据
4. 为了可扩展性考虑，数据库有时候必须是可复制和共享的
5. 不同服务存在不同的数据存储要求

每个服务一套数据库

为了解决上述问题，必须为每个微服务设计一个数据库。它必须仅专用于该服务。应当只能通过微服务的 API 访问它。其他服务无法直接访问它。比如，针对关系型数据库，我们可以采用每个服务使用单独的专用表（private-tables-per-service），每个服务单独的数据库模式（schema-per-service）或每个服务单独的数据库服务器（database-server-per-service）。

服务之间共享数据库

我们已经说过，在微服务里，为每个服务分配一套单独的数据库是理想方案。采用共享数据库在微服务里属于反模式。但是，如果应用程序是一个单体应用而且试图拆分成微服务，那么反正规化就不那么容易了。在后面的阶段里，我们可以转到每个服务一套数据库的模式，直到我们完全做到了这一点。服务之间共享数据库并不理想，但是对于上述情况，它是一个切实可行的解决方案。大多数人认为这是微服务的反模式，但是对于灰场应用程序，这是将应用程序分解成更小逻辑部分的一个很好的开始。值得一提的是，这不应当应用于绿场应用程序。

命令和查询职责分离（CQRS）

一旦实现了每个服务分配单独一套数据库（database-per-service），自然就会产生查询需求，这需要联合来自多个服务的数据。然而这是不可能的。CQRS 建议将应用程序分成两部分——命令端和查询端。

- 命令端处理创建，更新和删除请求
- 查询端通过使用物化视图来处理查询部分

这通常会搭配事件驱动模式（event sourcing pattern）一起使用，一旦有任何数据更改便会创建对应的事件。通过订阅事件流，我们便可以让物化视图保持更新。

事件驱动

绝大多数应用程序需要用到数据，典型的做法就是应用程序要维护当前状态。例如，在传统的创建，读取，更新和删除（CRUD）模型中，典型的数据流程是从存储中读取数据。它也包含了经常使用事务导致锁定数据的限制。

事件驱动模式定义了一种方法，用于处理由一系列事件驱动的数据操作，每个事件都记录在一个 append-only 的存储中。应用程序代码向事件存储发送一系列事件，这些事件命令式的描

述了对数据执行的每个操作，它们会被持久化到事件存储。每个事件代表一组数据更改（例如，AddedItemToOrder）。

这些事件将保留在充当记录系统的一个事件存储里。事件存储发布的事件的典型用途是在应用程序触发的一些动作更改实体时维护这些实体的物化视图，以及与外部系统集成。例如，一个系统可以维护一个用于填充 UI 部分所有客户订单的物化视图。当应用程序添加新订单，添加或删除订单中的项目以及添加运输信息时，描述这些更改的事件将会得到处理并用于更新物化视图。下图展示了该模式的一个概览。

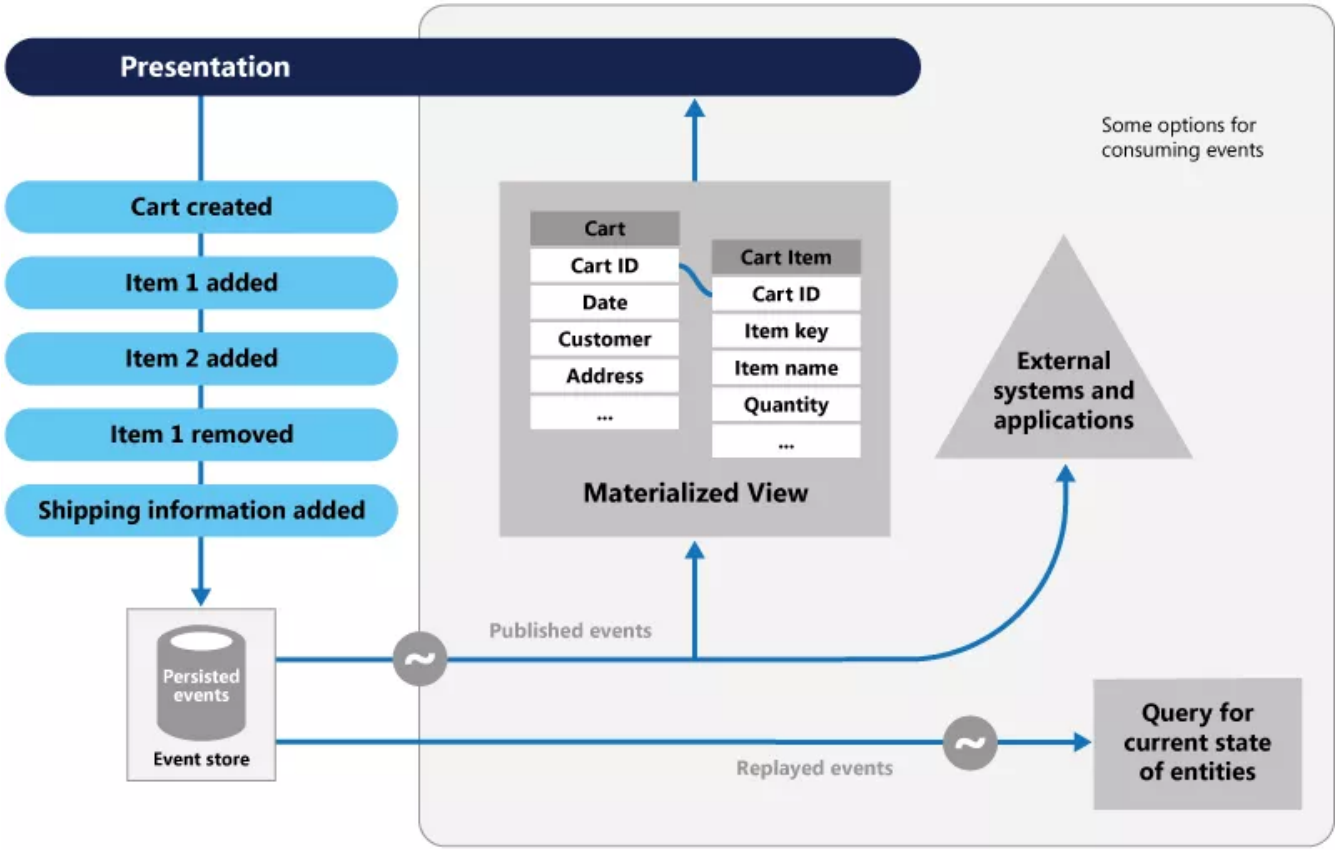

图2 事件驱动模式

Saga模式

当每个服务都有它们自己的数据库，并且一个业务事务跨越多个服务时，我们该如何确保各个服务之间的数据一致性呢？每个请求都有一个补偿请求，它会在请求失败时执行。这可以通过两种方式实现：

1. 编舞（Choreography）—— 在没有中央协调的情况下，每个服务都会生成并侦听另一个服务的事件，并决定是否应该采取措施。编舞是一种指定两个或多个参与方的方案。任何一方都无法控制对方的流程，或者对这些流程有任何可见性，无法协调他们的活动和流

程以共享信息和值。当需要跨控制/可见性域进行协调时，请使用编舞的方式。参考一个简单场景，你可以把编舞看作和网络协议类似。它规定了各方之间可接受的请求和响应模式。

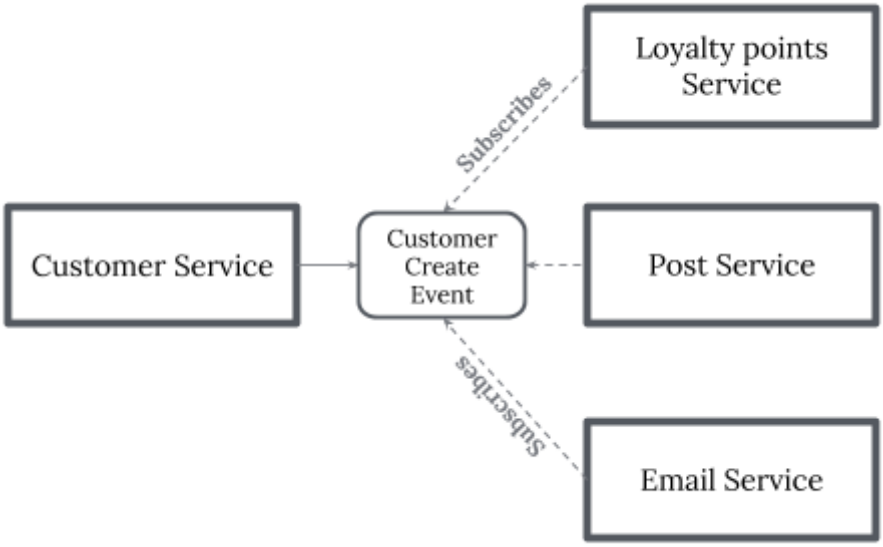

图3 Saga模式 —— 编舞

2. 编排（Orchestration）—— 一个编排器（对象）会负责 saga 的决策和业务逻辑排序。此时你可以控制流程中的所有参与者。当它们全部处于一个控制域时，你可以控制该活动的流程。当然，这通常是你被指派到一个拥有控制权的组织里制定业务流程。

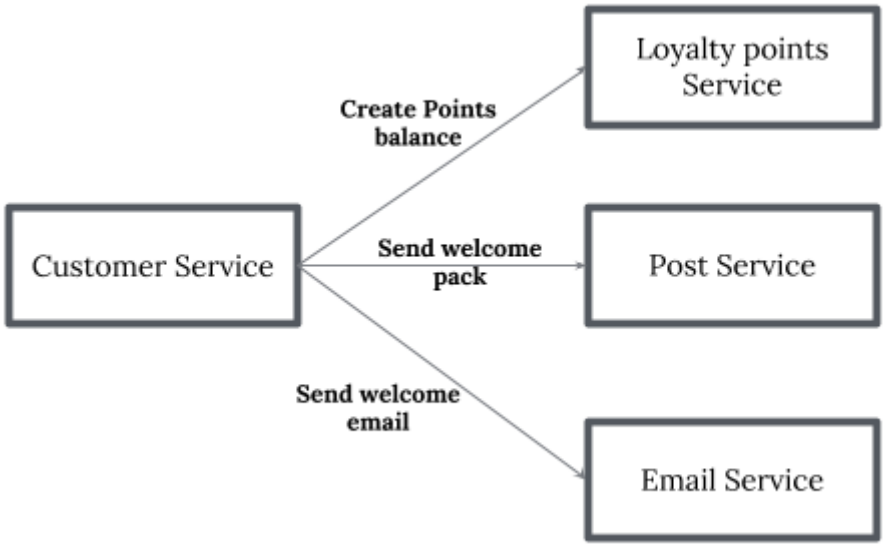

图4 Saga模式 —— 编排

可观测性模式

日志聚合

考虑一个应用程序包含多个服务的用例。请求通常跨越多个服务实例。每个服务实例均采用标准格式生成日志文件。我们需要一个集中式的日志记录服务，该服务可以汇总每个服务实例的日志。用户可以搜索和分析日志。他们可以配置在某些消息出现在日志中时触发告警。例如，PCF 就有日志聚合器，它在应用侧从 PCF 平台的每个组件（router、controller、diego 等）收集日志。AWS Cloud Watch 也是这样做的。

性能指标

当服务组合由于引入了微服务架构而增加时，保持对事务的监控就变得尤为关键了，如此一来就可以监控这些模式，而当有问题发生时便会发送告警。

此外，需要一个度量服务来收集有关单个操作的统计信息。它应当聚合一个应用服务的指标数据，它会用来报告和告警。这里有两种用于汇总指标的模型：

- 推送——服务将指标推送到指标服务，例如 NewRelic，AppDynamics
- 提取——指标服务从服务中提取指标，例如 Prometheus

分布式链路追踪

在微服务架构里，请求通常跨越多个服务。每个服务通过跨越多个服务执行一个或多个操作来处理请求。在排障时，有一个 Trace ID 是很有帮助的，我们可以端对端地跟踪一个请求。

解决方案便是引入一个事务ID。可以采用如下方式：

- 为每个外部请求分配一个唯一的外部请求ID
- 将外部请求ID传递给处理该请求链路的所有服务
- 在所有日志消息中加入该外部请求ID

健康检查

实施微服务架构后，服务可能会出现启动了但是无法处理事务的情况。每个服务都需要有一个可用于检查应用程序运行状况的 API 端点，例如 /health。该 API 应该检查主机的状态，与其他服务/基础设施的连接以及任何其他特定的逻辑。

横切关注点模式（Cross-Cutting Concern Patterns）

外部配置

一个服务通常还会调用其他服务和数据库。对于 dev，QA，UAT，Prod 等每个环境而言，API 端点的 URL 或某些配置属性可能会有所不同。这些属性中的任何一个更改都可能需要重新构建和重新部署服务。

为避免代码修改，可以使用配置。把所有配置放到外面，包括端点 URL 和证书。应用程序应该在启动时或运行时加载它们。这些可以在启动时由应用程序访问，也可以在不重新启动服务器的情况下进行刷新。

服务发现模式

在微服务出现时，我们需要在调用服务方面解决一些问题。

借助容器技术，IP地址可以动态地分配给服务实例。每次地址更改时，消费端服务都会中断并且需要手动更改。

对于消费端服务来说，它们必须记住每个上游服务的 URL ，这就变成紧耦合了。

为此，需要创建一个服务注册中心，该注册表将保留每个生产者服务的元数据和每个服务的配置。服务实例在启动时应当注册到注册中心，而在关闭时应当注销。服务发现有两种类型：

- 客户端：例如：Netflix Eureka
- 服务端：例如：AWS ALB

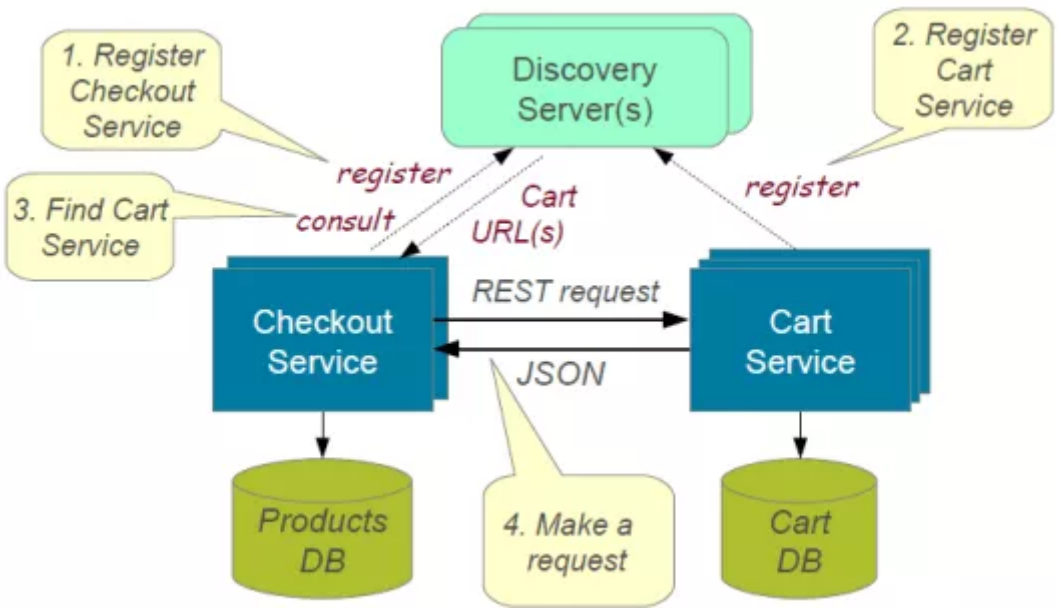

图5 服务发现

熔断器模式

一个服务通常会通过调用其他服务来检索数据，而这时候下游服务可能已经挂了。这样的话，有两个问题：首先，请求将继续抵达挂了的服务，耗尽网络资源，并且降低性能。其次，用户体验将是糟糕且不可预测的。

消费端服务应通过代理来调用远程服务，该代理的表现和一个电流断路器类似。当连续的故障数超过阈值时，断路器将跳闸，并且在超时期间内，所有调用远程服务的尝试都会立即失败。超时到期后，断路器将允许有限数量的测试请求通过。如果这些请求成功，断路器则将恢复正常运行。否则，如果发生故障的话，超时时间则将再次重新开始计算。如果某些操作失败概率很高的话，采取此模式有助于防止应用程序在故障发生后仍然不断尝试调用远程服务或访问共享资源。

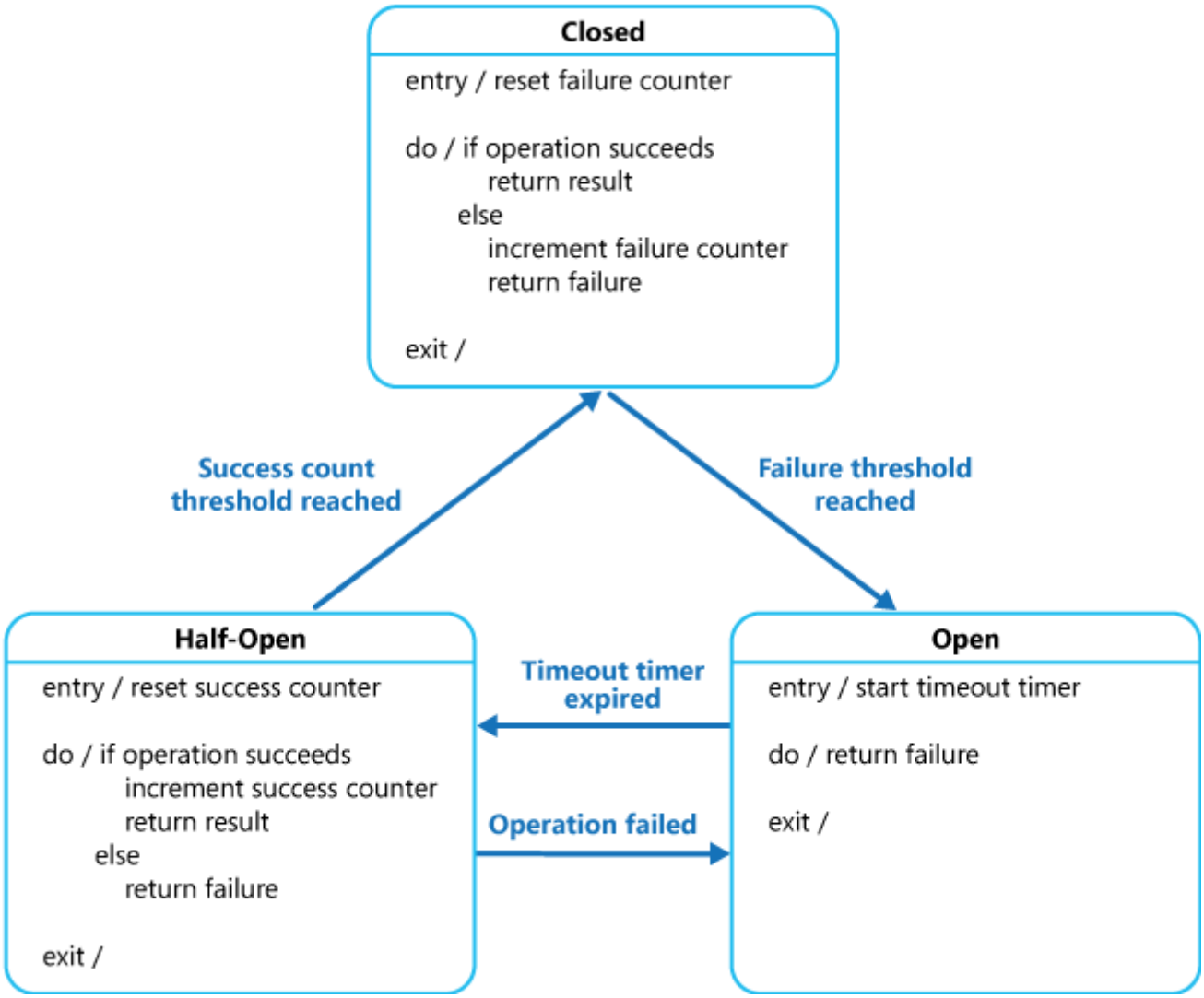

图6 熔断器模式

蓝绿部署模式

使用微服务架构时，一个应用可以被拆分成许多个微服务。如果我们采用停止所有服务然后再部署改进版本的方式的话，宕机时间将是非常可观的，并且会影响业务。同样，回滚也将是一场噩梦。蓝绿部署模式可以避免这种情况。

实施蓝绿部署策略可以用来减少或消除宕机。它通过运行两个相同的生产环境，Blue 和Green 来实现这一目标。假设 Green 是现有的活动实例，Blue 是该应用程序的新版本。在任何时候，只有一个环境处于活动状态，该活动环境为所有生产流量提供服务。所有云平台均提供了用于实施蓝绿部署的选项。

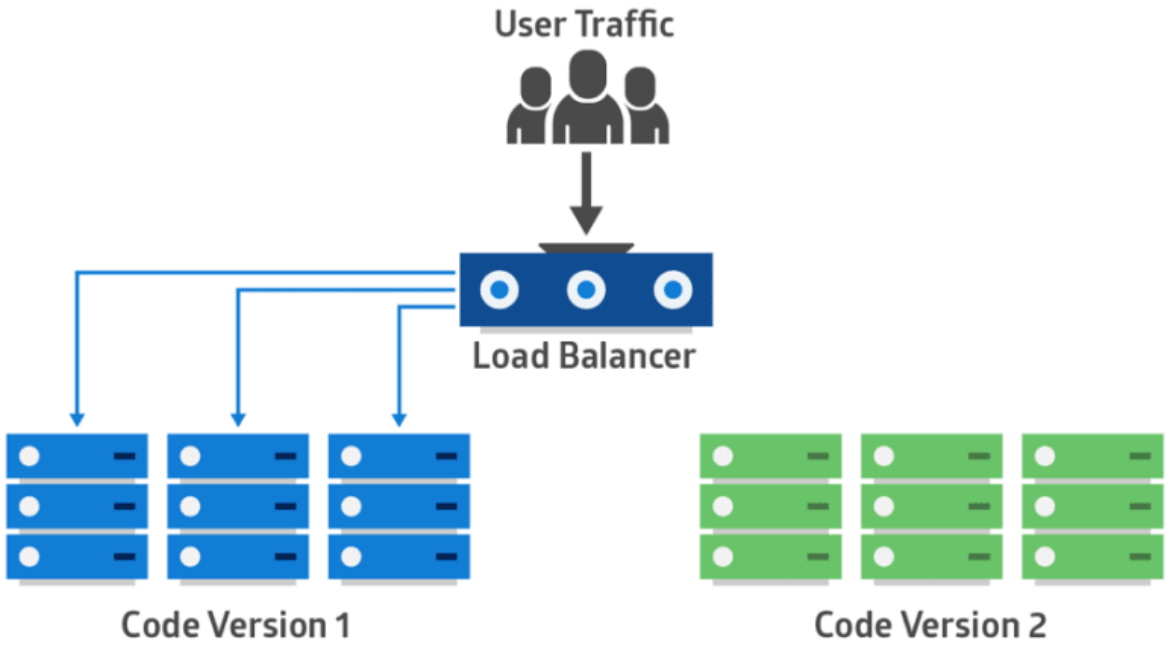

图7 蓝绿部署模式

原文链接：https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a

Kubernetes入门与实战培训

Kubernetes入门与实战培训将于2020年2月28日在北京开课，3天时间带你系统掌握Kubernetes，学习效果不好可以继续学习。本次培训包括：Docker基础、容器技术、Docker镜像、数据共享与持久化、Docker实践、Kubernetes基础、Pod基础与进阶、常用对象操作、服务发现、Helm、Kubernetes核心组件原理分析、Kubernetes服务质量保证、调度详解与应用场景、网络、基于Kubernetes的CI/CD、基于Kubernetes的配置管理等等，点击下方图片或者阅读原文链接查看详情。

阅读原文

喜欢此内容的人还喜欢

全面解析 Netflix 的微服务架构设计

InfoQ

微服务+异步工作流+Serverless，Netflix 决定弃用稳定运行7年的旧平台

架构之家