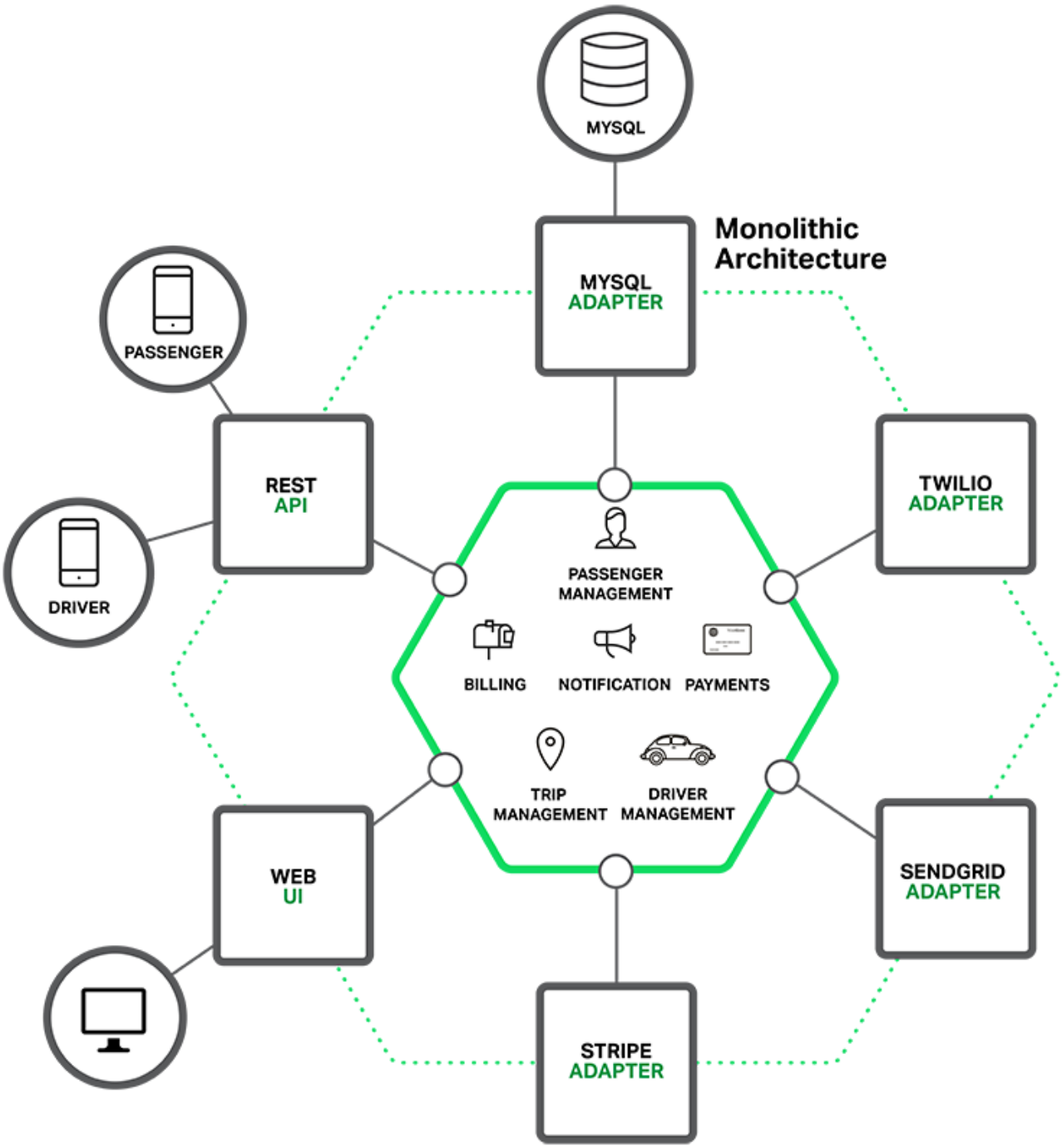
1、微服务介绍

1.1、微服务是如何演变过来的

1.1.1、开发一个单体应用

假设你正准备开发一款与 `Uber` 和 `Hailo` 竞争的出租车调度软件，经过初步会议和需求分析，你可能会手动或者使用基于 `Spring Boot` 、 `Play` 或者 `Maven` 的生成器开始这个新项目，它的六边形架构是模块化的，架构图如下：



应用核心是业务逻辑，由定义服务、域对象和事件的模块完成。围绕着核心的是与外界打交道的适配器。适配器包括数据库访问组件、生产和处理消息的消息组件，以及提供 `API` 或者 `UI` 访问支持的 `web` 模块等。

尽管也是模块化逻辑，但是最终它还是会打包并部署为单体式应用。具体的格式依赖于应用语言和框架。例如，许多Java应用会被打包为 `WAR` 格式，部署在 `Tomcat` 或者 `Jetty` 上。

1.1.2、单体式应用的不足

在做一个新项目的时候，一开始项目大多数都很小，都是「单体应用」，这是很常见的做法。

在项目规模小的时候，这种方式开发效率和运维效率都最高，项目规模逐渐变大后，单体应用就会存在以下问题：

- 复杂性高
  - 项目模块非常的多、模块边界模糊、依赖关系模糊、代码质量参差不齐
- 技术债务
  - 随着时间推移人员迭代、需求变更，会逐渐形成应用程序的技术债务
- 部署频率
  - 代码量的增加，构建和部署的时间也会增加。每次功能的变更或者缺陷修复都可能导致需要重新部署整个应用。全量部署的方式耗时长、风险大、影响范围广，使得整个应用上线部署效率低下
- 可靠性差
  - 某个应用的Bug（例如:死循环、OOM等），导致整个应用宕机
- 扩展能力受限
  - 单一应用只能作为一个整体进行扩展，无法根据业务模块的需要进行伸缩
- 阻碍技术创新
  - 单体应用一般都是使用统一的技术平台或者方案来解决所有的问题，团队中的所有人都必须使用相同的编程语言和框架，要想引入新的技术框架或者新的技术平台会非常的困难

1.1.3、如何解决

通过采用微处理结构模式解决上述问题。他的思路不是开发一个巨大的单体式的应用，而是将应用分解为小的、互相连接的「微服务」。

1.2、什么是微服务

- 微服务是由 **Martin fowler** 提出的，它是用来描述将软件应用程序设计为独立部署的服务的一种特殊方式。微服务本身没有一个严格的定义。
- **Martin fowler** 在他的 **博客** 对微服务的描述

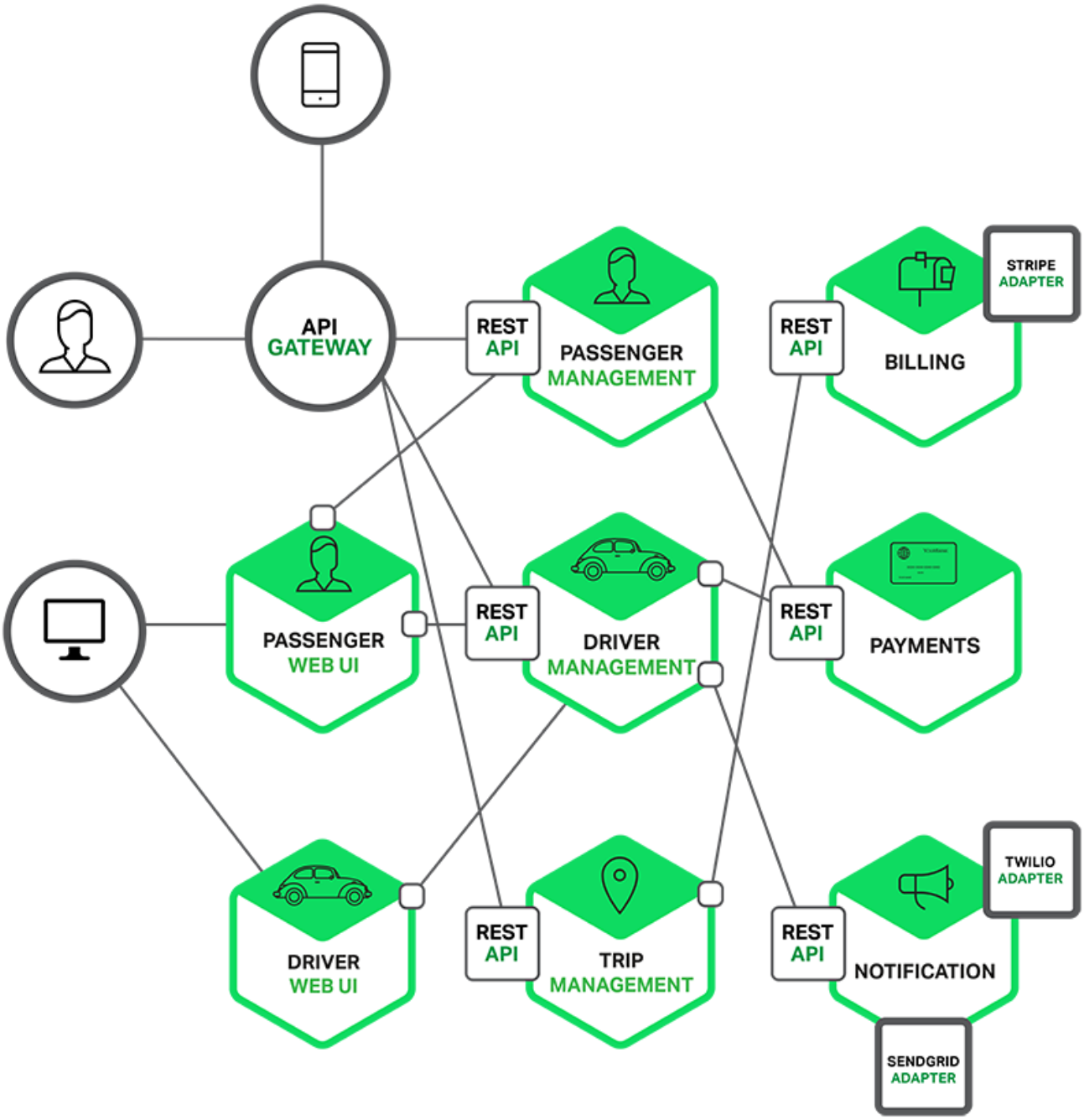
```
In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies
```

简而言之，微服务架构风格这种开发方法，是以开发一组小型服务的方式来开发一个独立的应用系统的。其中每个小型服务都运行在自己的进程中，并经常采用HTTP资源API这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术

1.3、微服务架构是怎么样的

1.3.1、单体应用拆分

一个「微服务」一般完成某个特定的功能，比如下单管理、客户管理等等。每一个「微服务」都是微型六角形应用，都有自己的业务逻辑和适配器。一些「微服务」还会发布 `API` 给其它「微服务」和应用客户端使用。其它「微服务」完成一个 `Web UI`，运行时，每一个实例可能是一个云 `VM` 或者是 `Docker` 容器。上述的单体应用示例使用「微服务」会拆分成这样：



1.3.2、微服务的优点

- 易于开发和维护
  - 一个微服务只会关注一个特定的业务功能，所以它业务清晰，代码量较少。开发和维护单个微服务相对简单。而整个应用是由若干个微服务构建而成，所以整个应用会被维持在一个可控状态

- 单个微服务启动快
  - 单个微服务代码量较少，所以启动比较快
- 局部修改容易部署
  - 单体应用只要有修改，就得重新部署整个应用。微服务解决了这个问题，一般来说修改某个微服务只需要重启某个微服务
- 技术栈不受限
  - 在微服务架构中，可以结合业务及团队的特点，合理选择技术栈。例如某些微服务可以选择 `MySQL` 。某些微服务有图形计算的需求可以选择 `Neo4J` 。甚至根据需要，部分微服务使用 `Java` 开发，部分使用 `Node JS` 开发
- 按需伸缩
  - 可根据需求，实现细粒度的扩展。例如系统中的某个微服务遇到了瓶颈，可以结合业务特点，增加内存，升级CPU或者是增加节点

1.3.3、微服务基础组件

要保证一个基于「 微服务 」架构实现的系统正常运行起来，最少需要以下几个组件:

- **服务注册**
  - 部署一个服务节点，如何让调用者知道？删除一个服务节点，如何也让调用者知道？通过**服务注册**组件来实现，服务提供者将自身的信息登记到服务注册中心，服务调用者调用服务需要先到服务注册中心查询可用的服务节点
- **服务网关**
  - 提供给外部系统调用的是统一网关。主要做授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等
- **配置中心**
  - 微服务的配置中心是用来统一管理所有微服务节点的配置信息的。因为同一个程序可能要适用于多个环境，所以在微服务实践中要尽量做到程序与配置分离，将配置进行集中管理。包括微服务节点信息、程序运行时配置、变量配置、数据源配置、日志配置、版本配置等。
- **服务框架**
  - 是指用来规范各个微服务节点之间通信标准的。服务间通信采用什么协议、数据是如何传输的、数据格式是什么样的。有了这个统一的“服务框架”就能保证各个微服务节点之间高效率的协同。
- **服务监控**
  - 微服务运行起来之后，为了能够监控节点的健康情况，保障节点的高可行，需要对各个服务节点进行收集数据指标、然后对数据进行实时处理和分析，形成监控报表和预警。
- **服务追踪**
  - 一旦使用了微服务架构，那么当有请求过来时，就会经过多个微服务节点的处理，形成了一个调用链。为了进行问题追踪和故障的定位，需要对请求的完整调用链进行记录。
    - 这里的服务追踪与上面的服务监控是不同维度的，一个是全局的，一个是微观的，发挥的作用也不一样。
- **服务治理**
  - 是指需要通过准备一些策略和方案，来保障整个微服务架构在生产环境遇到极端情况下也能正常提供服务的措施。比如 熔断、限流、隔离等等。

1.4、微服务面临的挑战

- 运维成本高
  - 更多的微服务意味着更多的运维投入。在单体架构中，只需要保证一个应用正常运行。而在微服务，需要保证几十甚至几百个微服务正常运行与协助，这给运维带来了很大的挑战（需要具备 `DevOps` ）
- 分布式固有的复杂性
  - 分布式固有的复杂性：使用微服务构建的是分布式系统，对于一个分布式系统，系统容错，网络延迟，分布式事务都会带来巨大的挑战（ `CAP` ）
- 接口调整成本高
  - 微服务之间通过接口进行通信。如果修改某个微服务API，可能所有使用了该接口的微服务都需要调整
- 重复劳动
  - 很多微服务可能都会使用到相同的功能，而这个功能并没有达到分解为一个微服务的程度，这个时候，可能各个微服务都会开发这一功能，从而导致代码重复

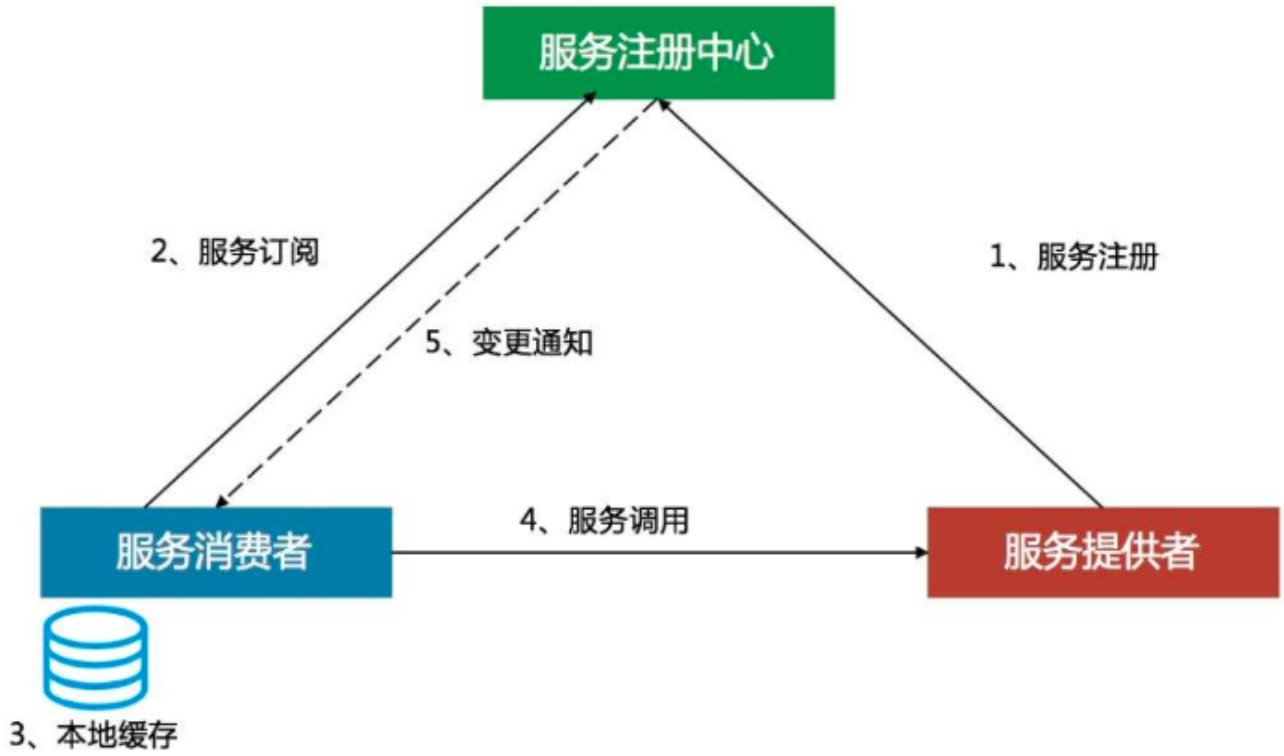
2、微服务架构组件

2.1、服务注册

2.1.1、为什么需要服务注册

【服务消费者】需要调用【服务提供者】的 `API` 来获得服务。当【服务提供者】的节点有新增或者剔除时，应该及时让【服务消费者】知晓。而大规模、集群的应用节点会很多，节点变化也很频繁，通过手动去维护这些节点的状态是不太现实的，因此才会需要服务注册中心组件来实现。

2.1.2、服务注册原理

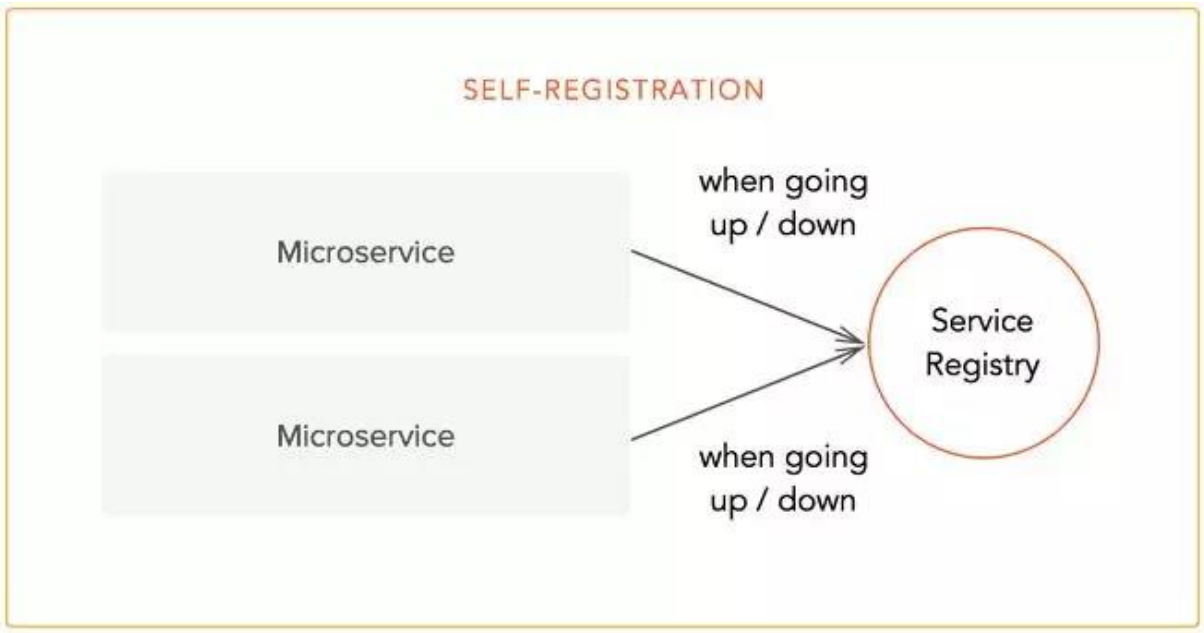


【服务提供者】需要将自身的服务信息注册到【服务注册中心】，【服务消费者】到【服务注册中心】检索服务，并将检索到的服务缓存在本地，对检索到的服务进行调用。

- **服务提供者**向**服务注册中心**注册

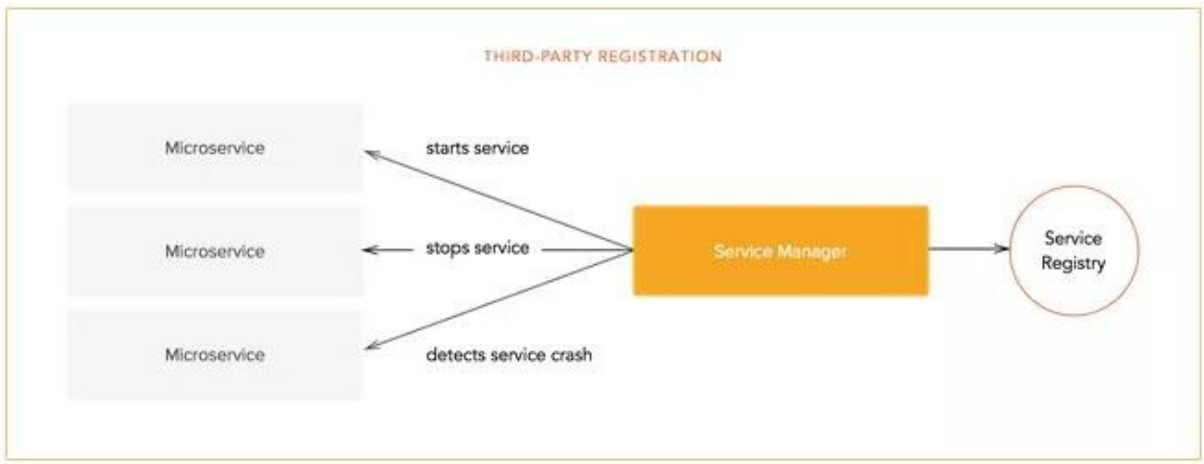
  注册分为两种分别是：自己注册和第三方注册

  - **自己注册**

    自己注册就是在服务启动的时候，自己去服务注册中心登记注册，将自身的服务信息、状态传递到服务注册中心。这种方式整体结构比较简单，对于注册中心比较省事，但是对于服务节点来说，每一个节点都需要包含一段注册的逻辑，从整体来看，架构不是完美的。

  - **第三方注册**

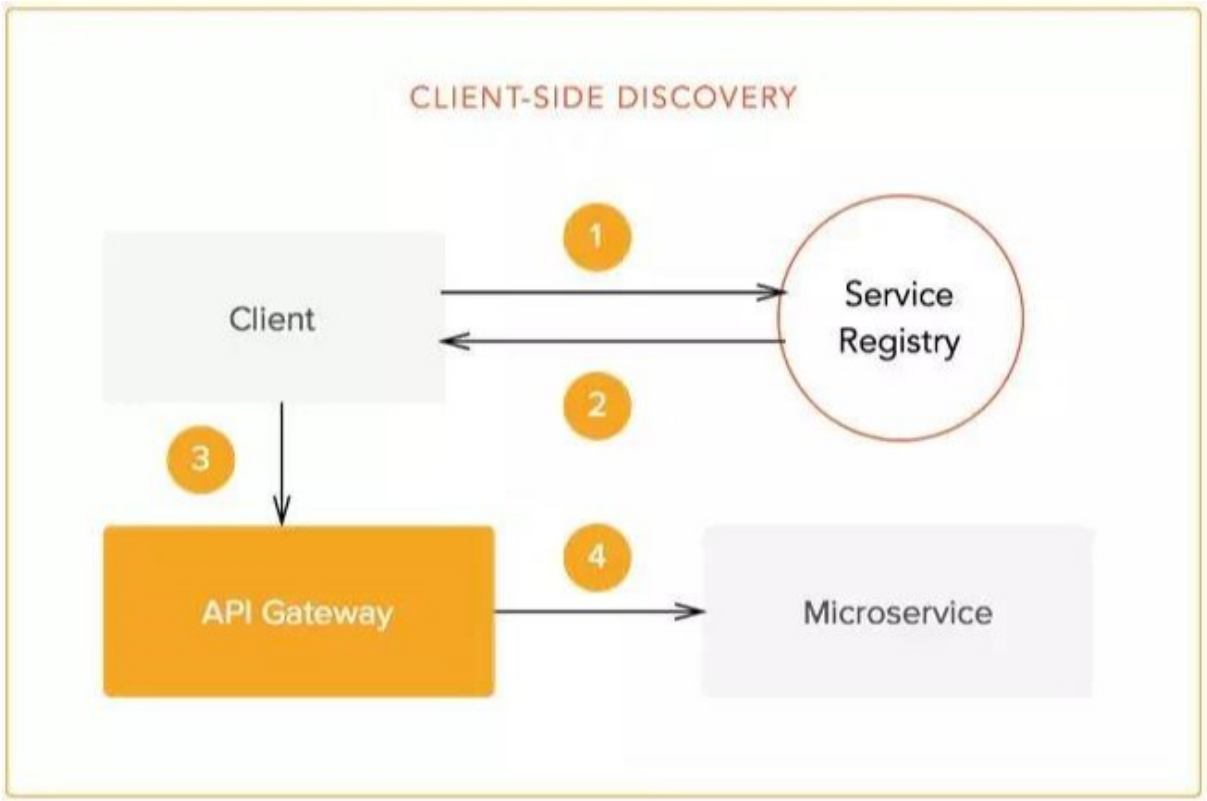
    

    第三方注册是指有一个 `Service Manager` （服务管理器），它会去管理所有的服务和进程信息，以轮询的方式（或者其他方式）去检索哪些服务正在运行，会将这些服务实例自动更新到服务注册中心。

- **服务消费者**向**服务注册中心**检索和调用服务（服务发现）

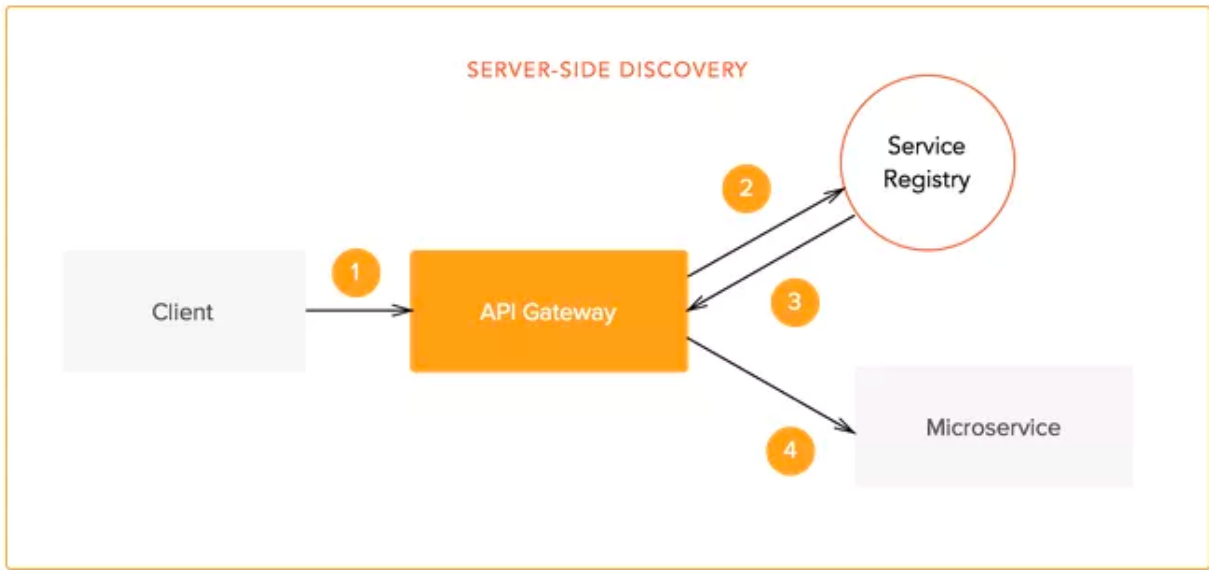
  服务调用和查询也分为两种分别是：客户端模式和代理模式

  - **客户端模式**



`Client` (服务消费者)向 `Service Registry` （服务注册中心）检索到自己需要调用的服务地址后，`Client` 会根据地址去访问 `Microservice` （微服务）（ `API Gateway` 是可选项，如果有 `API Gateway` ，那么它起到负载均衡作用；如果没有，那么需要 `Client` 自己写负载均衡策略）

- **代理模式**



`Client` （服务消费者）与 `Service Registry` （服务注册中心）中间有 `API Gateway` 组件。`Client` 只管找 `API Gateway` 访问。至于 `API Gateway` 怎么查询服务地址，以及访问服务地址的动作都由 `API Gateway` 代劳，最后 `API Gateway` 将服务结果返回给 `Client` 。

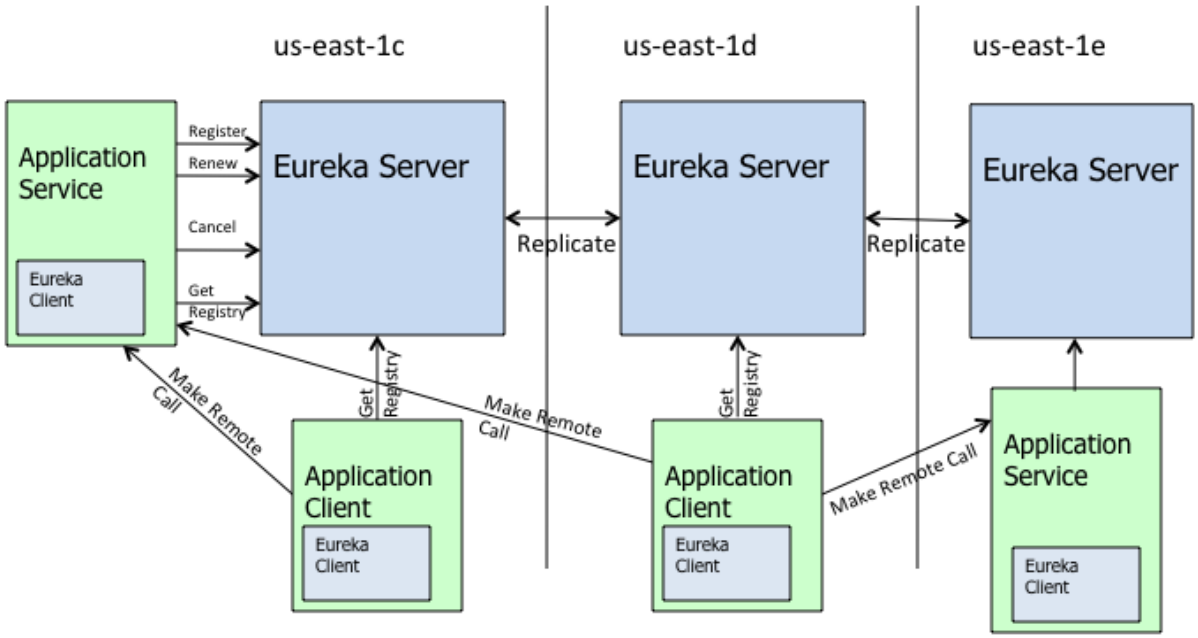
这种模式，看起来“服务消费者”省事了，但是 `API Gateway` 模块却复杂了，因为 `API Gateway` 就是整个系统的一个非常核心关键节点了，不仅需要保障自己的稳定性和性能，而且还需要处理一些负载均衡的逻辑。

2.1.3、服务注册实践

虽然可以根据原理研发一套服务注册中心，如果没有特殊需求的话，还是不用重复造轮子。毕竟市面上很多成熟的服务注册中心解决方案，并且都经历过实际场景验证，可以直接拿过来使用。

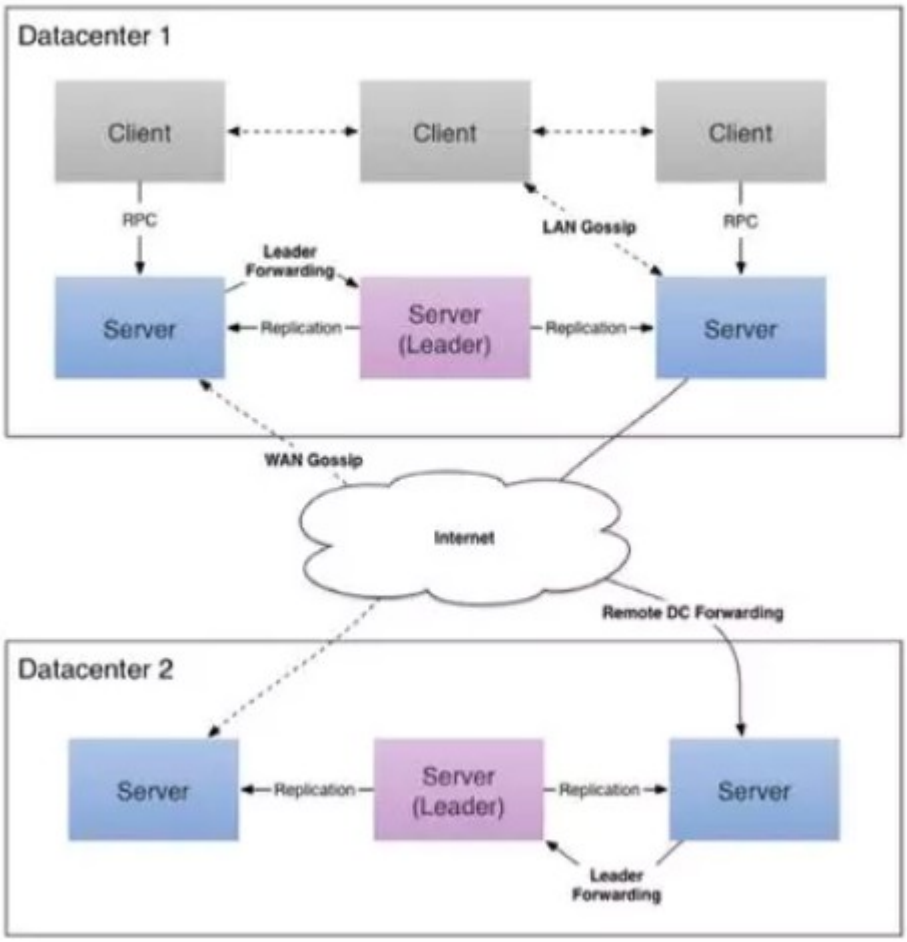
- **Eureka**：`[ju'ri:kə]`

  Eureka(**https://github.com/Netflix/eureka**)是由`Netflix`开源，其架构如下图:



  - 从图中可以看到，我们的服务（图中Application Clinet与Application Service）要使用Eureka就需要集成它的SDK（图中Eureka Client）。图中的Eureka部署在了三个异地机房，也就是说Eureka是支持多中心部署的。
  - 服务提供者（Application Service）通过Eureka Client实现服务的注册、更新和注销等。服务消费者（Application Clinet）通过Eureka Client实现服务的查询和调用。
  - Eureka支持了与Spring Cloud的集成，所以使用起来也非常方便，目前属于比较流行的方案。
- **Consul**：`['kɔnsl]`

  Consul是另外一个非常流行的开源组件，如下图:



  - Consul是在服务外进行完成一系列动作的，也就是说并不需要服务节点去依赖它的SDK，没有侵入性，所以跨语言的解决能力更强一些。它一般是在服务节点外通过一些探针的方法去检查应用是否存活，是否需要注册或注销。
  - Consul也支持Spring Cloud集成，所以使用起来也很方便，也属于比较流行的方案。
- **Etcd、Zookeeper**

  这两个也有一些公司基于它们来实现服务注册，也集成了Spring Cloud，不过不算非常广泛。

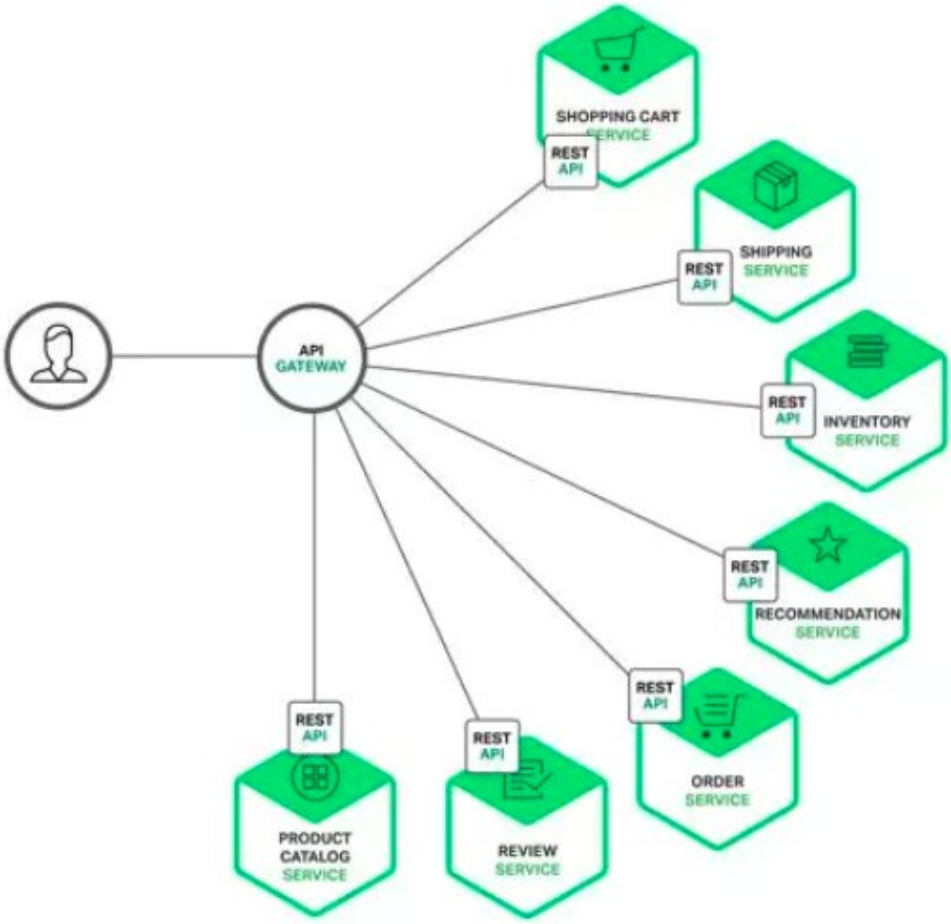
2.2、服务网关

2.2.1、为什么需要服务网关

服务网关是对外服务的一个入口，其隐藏了内部架构的实现，是微服务架构中必不可少的一个组件。

2.2.2、服务网关原理

网关架构示意图：



`API Gate` 类似一个大门，是连接外部客户端和内部服务的一个桥梁，其主要功能有：

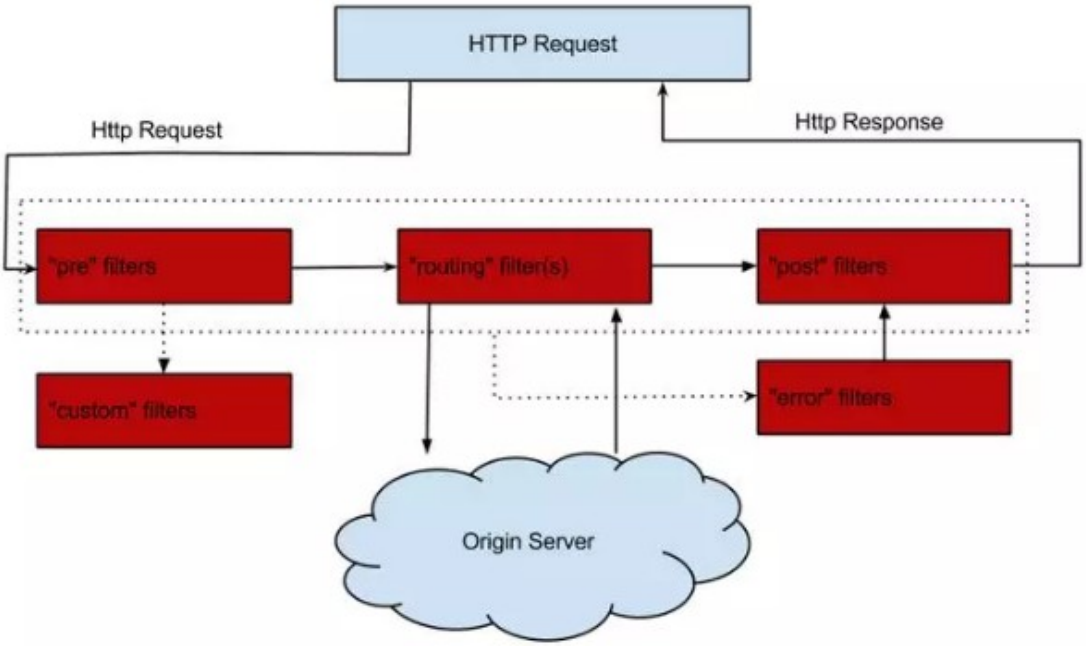
- 网关将所有服务的接口统一聚合，并统一对外暴露。外界系统调用服务接口时，都是由网关对外暴露的服务 `API` 接口，外界系统不需要知道微服务系统中各服务相互调用的复杂性。服务网关也保护了其内部微服务单元的 `API` 接口，防止其被外界直接调用，导致服务的敏感信息对外暴露。
- 集成服务注册中心，可以实现只能路由和负载均衡功能
- 网关可以用作用户身份认证和权限认证，防止非法请求操作 `API` 接口，对服务器起到保护作用
- 网关可以实现监控功能，实时日志输出，对请求进行记录
- 网关可以用来实现流量监控，在高流量的情况下，对服务进行降级
- 等等

其类似Java Web 中过滤器、拦截器，作用就是把项目中的一些非业务逻辑的功能抽离出来独立处理，避免与业务逻辑混在一起增加代码复杂度。如果让每一个节点都去实现权限认证、日志记录等，会多出很多冗余的代码。所以才会需要一个独立的功能来实现上述的需求。

2.2.3、服务网关实践

可以根据需求研发出服务网关，如果没有特殊需求可以使用市面上比较成熟的解决方案：

- **Zuul**
  - Zuul 是由 Netflix 所开源的组件，基于JAVA技术栈开发的。
  - Zuul网关的使用热度非常高，并且也集成到了 Spring Cloud 全家桶中了，使用起来非常方便。

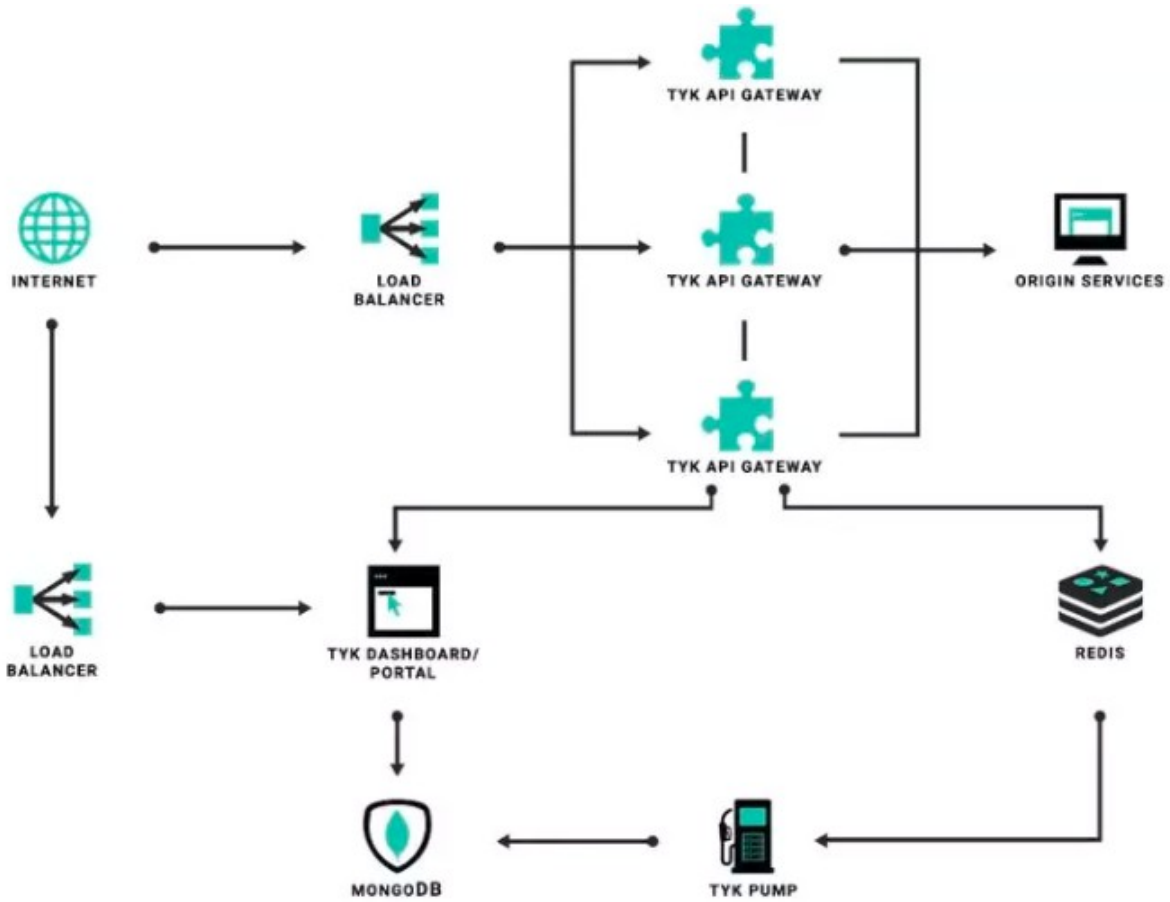


- 上图可以看到Zuul的一个简化结构，过滤器filter是整个Zuul的核心，分为前置过滤器（pre filter）、路由过滤器（routing filter）、后置过滤器（post filter）以及 错误过滤器（error filter）。
- 一个请求过来，会先执行所有的 pre filter，然后再通过 routing filter 将请求转发给后端服务，后端服务进行结果响应之后，再执行 post filter，最后再响应给客户端。在不同的filter里面可以执行不同的逻辑，比如安全检查、日志记录等等。

- **Tyk**

Tyk是一个基于GO编写的，轻量级、快速可伸缩的开源的API网关。

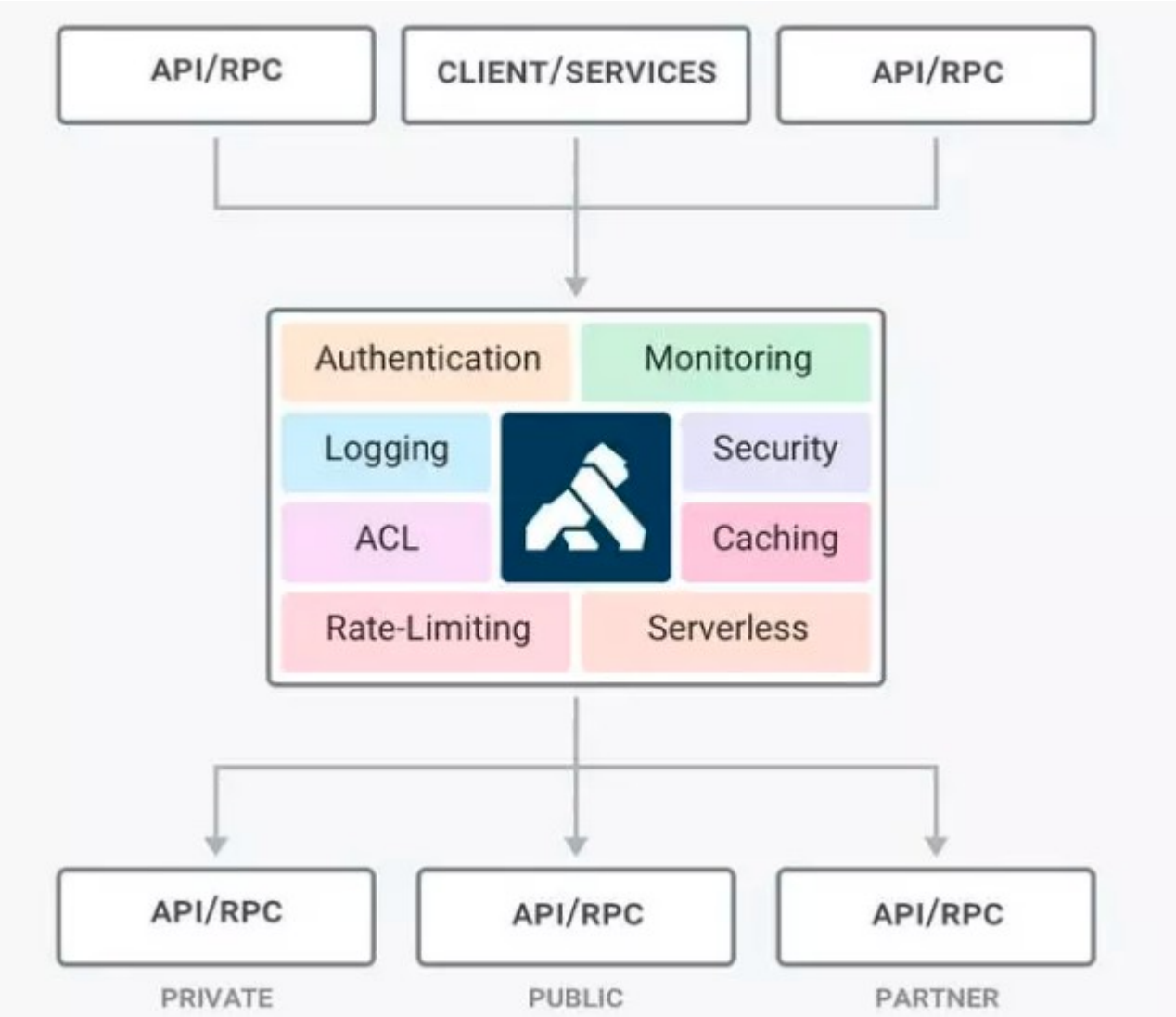
可以通过下图简单了解一下Tyk的流程原理：



- **Kong**

Kong是基于OpenResty技术栈的开源网关服务，因此其也是基于Nginx实现的。

Kong可以做到高性能、插件自定义、集群以及易于使用的Restful API管理。



- Spring Cloud Gateway

  Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul网关。

  Spring Cloud Gateway是基于Spring5(支持响应式编程)

2.3、配置中心

2.3.1、为什么需要配置中心

> 【配置中心】的思路就是把项目中各种配置、各种参数、各种开关，全部都放到一个集中的地方进行统一管理，并提供一套标准的接口。当各个服务需要获取配置的时候，就来【配置中心】的接口拉取。当【配置中心】中的各种参数有更新的时候，也能通知到各个服务实时的过来同步最新的信息，使之动态更新。

传统方式是怎么管理的配置文件的：

- 一般是静态化配置
  - 大多数在项目中单独写一个配置文件，例如 `platform.properties` ，然后将各类 参数配置、应用配置、环境配置、安全配置、业务配置 都写到这个文件里。当项目代码逻辑中需要使用配置的时候，就从这个配置文件中读取。这种做法虽然简单，但如果参数需要修改，就非常的不灵活，甚至需要重启运行中的项目才能生效。相信大多数开发同学都深有体会。
- 配置文件无法区分环境

    - 由于配置文件是放在项目中的，但是我们项目可能会有多个环境，例如：测试环境、预发布环境、生产环境。每一个环境所使用的配置参数理论上都是不同的，所以我们在配置文件中根据不同环境配置不同的参数，这些都是手动维护，在项目发布的时候，极其容易因开发人员的失误导致出错。
- 配置文件过于分散
    - 如果一个项目中存在多个逻辑模块独立部署，每个模块所使用的配置内容又不相同，传统的做法是会在每一个模块中都放一个配置文件，甚至不同模块的配置文件格式还不一样。那么长期的结果就是配置文件过于分散混乱，难以管理。
- 配置修改无法追溯
    - 因为采用的静态配置文件方式，所以当配置进行修改之后，不容易形成记录，更无法追溯是谁修改的、修改时间是什么、修改前是什么内容。既然无法追溯，那么当配置出错时，更没办法回滚配置了。

上述以静态配置文件为例，当然也有采用数据库配置的，虽然灵活一点，但是依旧不能完全解决上述问题。

配置中心应该具备的特点：

- 配置集中管理、统一标准
- 配置与应用分离
- 实时更新
- 高可用

配置中心是如何解决上述问题的：

- 采用“配置集中管理”，可以很好的解决传统的“配置文件过于分散”的问题。所有的配置都集中在配置中心这一个地方管理，不需要每一个项目都自带一个，这样极大的减轻了开发成本。
- 采用“配置与应用分离”，可以很好的解决传统的“配置文件无法区分环境”的问题，配置并不跟着环境走，当不同环境有不同需求的时候，就到配置中心获取即可，极大的减轻了运维部署成本。
- 具备“实时更新”的功能，就是用来解决传统的“静态化配置”的问题。线上系统需要调整参数的时候，只需要在配置中心动态修改即可。
- 既然配置都统一管理了，那配置中心在整个系统中的地位就非常重要了，一旦配置中心不能正常提供服务，就可能会导致项目整体故障，因此“高可用”就是配置中心又一个很关键的指标了。

2.3.2、配置中心原理

通过上述的分析配置中心的核心功能应该具备以下几项：

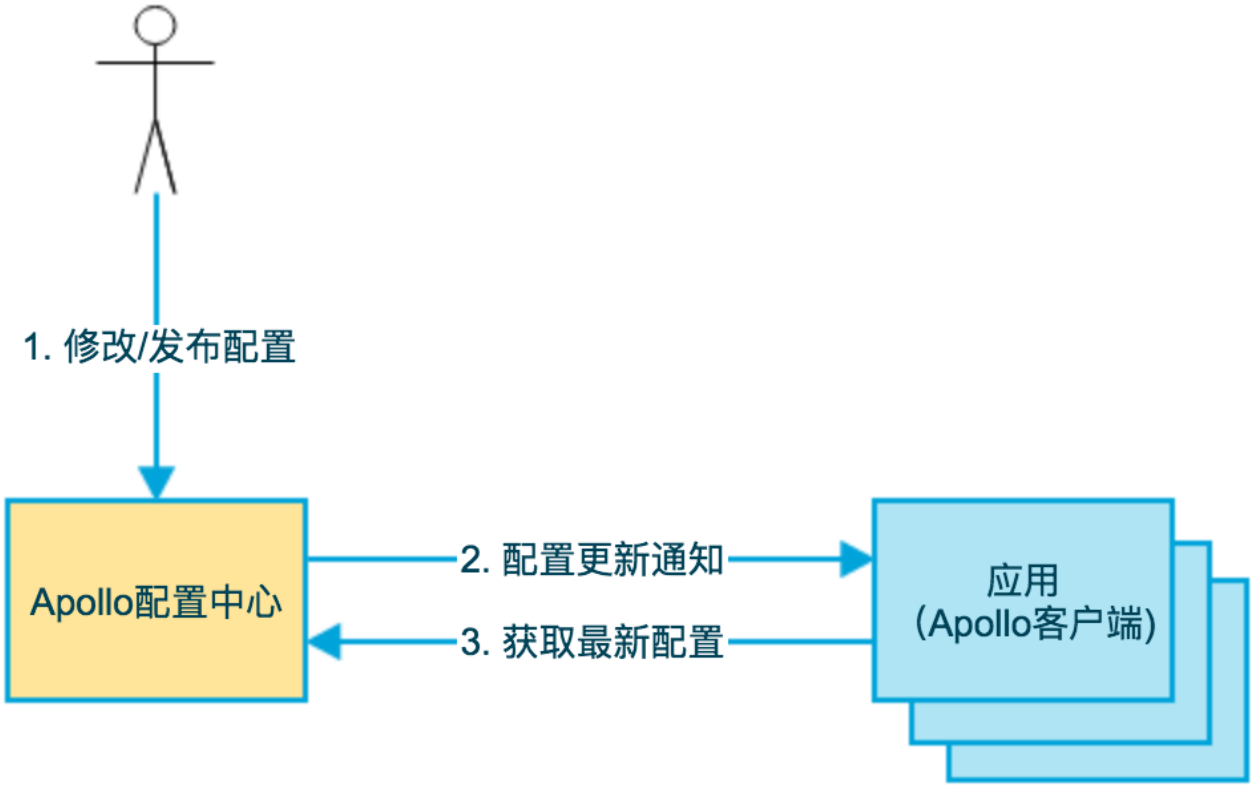
- 实现配置的记录
- 实现配置的读取、更新、取消
- 实现配置的查看

围绕这几个功能，还需要保证高可用，要实现实时更新、要能方便的使用，还希望有权限管理的功能、操作审计的功能等等，加上这些周边辅助功能之后，一个完善的【配置中心】也就不那么简单了。

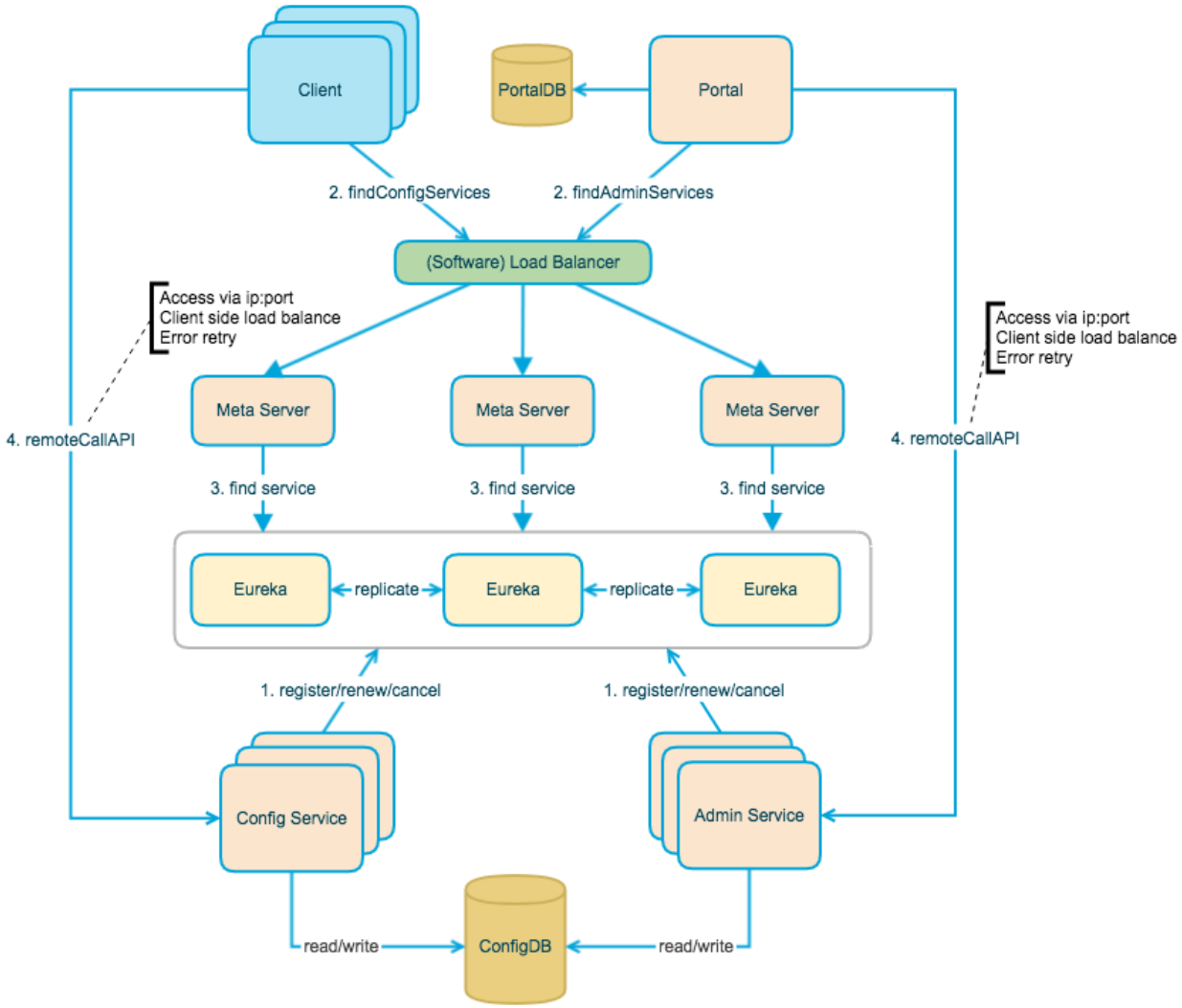
2.3.3、配置中心实践

技术选型和应用：

- **Apollo**
    - Apollo是由携程开源的分布式配置中心。能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。
    - 开源地址： https://github.com/ctripcorp/apollo
    - 基础模型图：

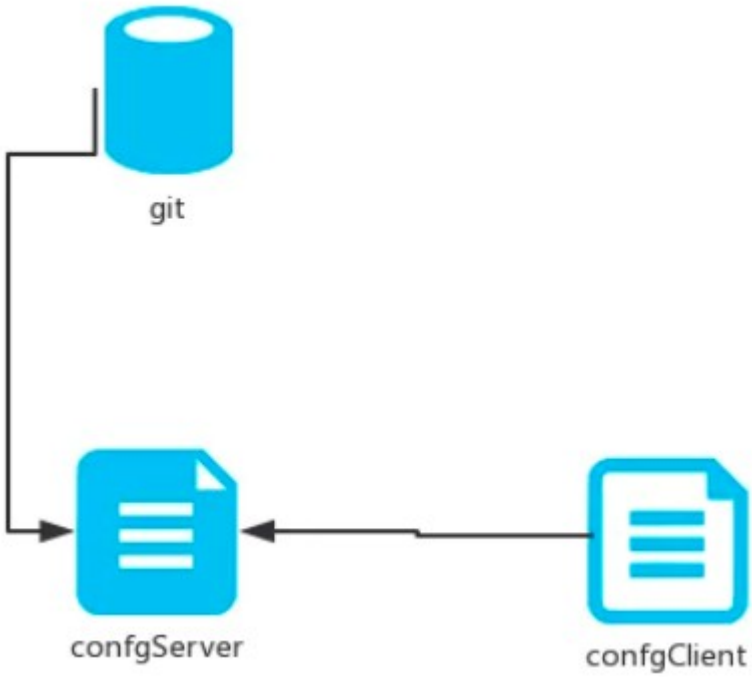


- Apollo架构图



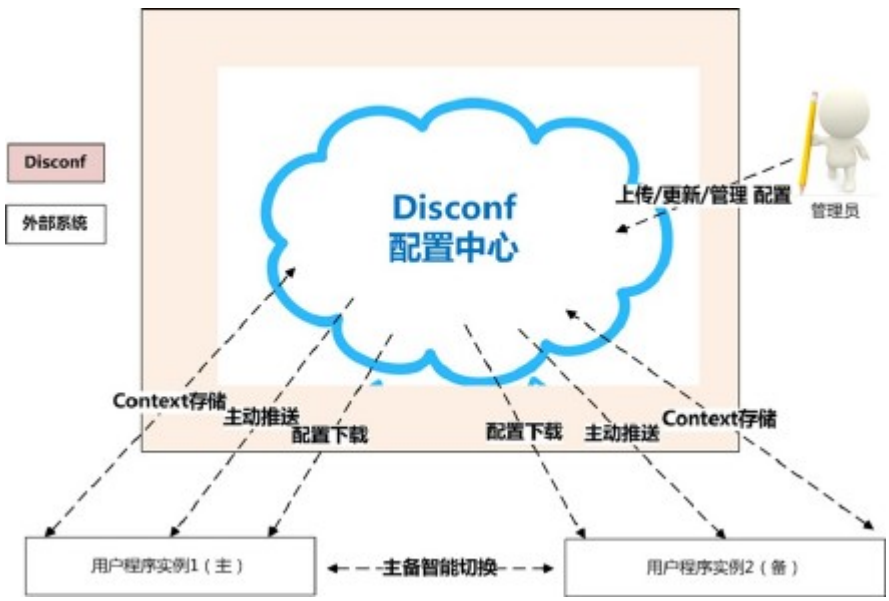
更多细节参考：**开源地址**

- **Spring Cloud Config**
  - Spring Cloud Config是一个用来为分布式系统提供配置集中化管理的服务，它分为客户端和服务端两个部分。客户端从服务端拉去配置数据，服务端负责提供配置数据。
  - Spring Cloud Config底层存储提供了多种方式，最好用的是Git来存储配置信息，还可以跟踪版本，随时恢复到指定的版本，当然也支持SVN、本地文件存储、数据库（E版支持）等方式。

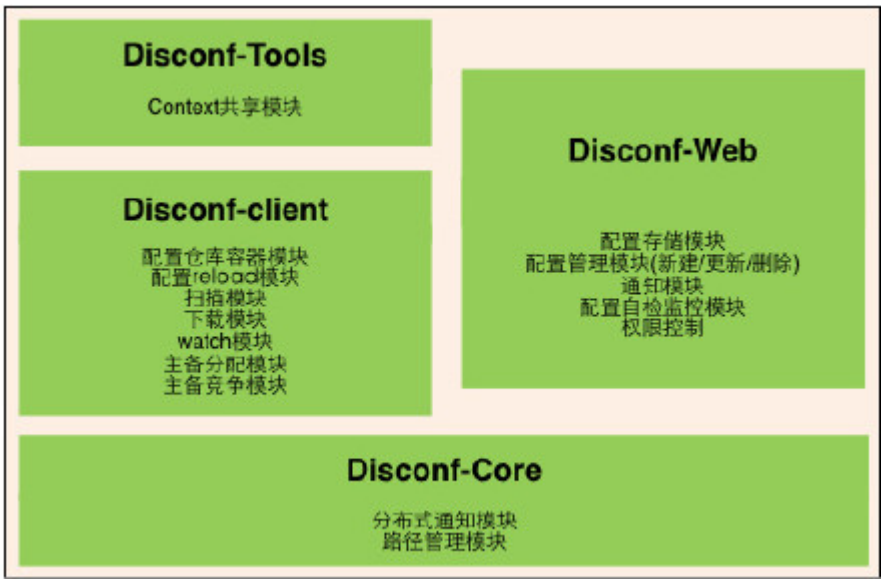


    - 支持配置的加解密，配置的自动更新，自动更新可以手动调用接口去触发，也可以利用消息总线和Git的 WebHook 来实现配置修改的自动更新。

- **Disconf**
    - Disconf是由百度开源的分布式配置中心
    - 开源地址： https://github.com/knightliao/disconf
    - 基于Zookeeper来实现配置变更后实时通知和生效的
    - 帮助文档： https://disconf.readthedocs.io/zh_CN/latest/
    - 架构设计
        - 服务集群



        - 模块架构



2.4、服务监控

2.4.1、为什么需要服务监控

在微服务的架构下，我们对服务进行了拆分，所以用户的每次请求不再是由某一个服务独立完成了，而是变成了多个服务一起配合完成。这种情况下，一旦请求出现异常，我们必须得知道是在哪个服务环节出了故障，就需要对每一个服务，以及各个指标都进行全面的监控。

2.4.2、什么是监控系统

在微服务架构中，监控系统按照原理和作用大致可以分为三类：

- **日志类**
  - 日志类比较常见，我们的框架代码、系统环境、以及业务逻辑中一般都会产出一些日志，这些日志我们通常把它记录后统一收集起来，方便在需要的时候进行查询。
  - 日志类记录的信息一般是一些事件、非结构化的一些文本内容。日志的输出和处理的解决方案比较多，大家熟知的有 **ELK Stack** 方案
- **调用链类**
  - 调用链类监控主要是指记录一个请求的全部流程。一个请求从开始进入，在微服务中调用不同的服务节点后，再返回给客户端，在这个过程中通过调用链参数来追寻全链路行为。通过这个方式可以很方便的知道请求在哪个环节出了故障，系统的瓶颈在哪儿。
  - 一般采用 **CAT** 工具 来完成，一般在大中型项目较多用到，因为搭建起来有一定的成本。后面会有单独文章来讲解这个调用链监控系统。
- **度量类**
  - 度量类主要采用 **时序数据库** 的解决方案。它是以事件发生时间以及当前数值的角度来记录的监控信息，是可以聚合运算的，用于查看一些指标数据和指标趋势。所以这类监控主要不是用来查问题的，主要是用来看趋势的。
  - Metrics一般有5种基本的度量类型：Gauges（度量）、Counters（计数器）、 Histograms（直方图）、Meters（TPS计算器）、Timers（计时器）
  - 基于时间序列数据库的监控系统是非常适合做监控告警使用的，所以现在也比较流行这个方案，如果我们要搭建一套新的监控系统，我也建议参考这类方案进行。

2.4.3、关注的对象和指标

一般做【监控系统】都是需要做分层式监控的，也就是说将我们要监控的对象进行分层，一般主要分为：

- **系统层**
  - 系统层主要是指CPU、磁盘、内存、网络等服务器层面的监控，这些一般也是运维同学比较关注的对象。
- **应用层**
  - 应用层指的是服务角度的监控，比如接口、框架、某个服务的健康状态等，一般是服务开发或框架开发人员关注的对象。
- **用户层**
  - 这一层主要是与用户、与业务相关的一些监控，属于功能层面的，大多数是项目经理或产品经理会比较关注的对象。

监控的指标一般有哪些：

- **延迟时间**
  - 主要是响应一个请求所消耗的延迟，比如某接口的HTTP请求平均响应时间为100ms
- **请求量**
  - 是指系统的容量吞吐能力，例如每秒处理多少次请求（QPS）作为指标
- **错误率**
  - 主要是用来监控错误发生的比例，比如将某接口一段时间内调用时失败的比例作为指标

2.4.4、监控系统实践

- Spring Boot Admin
- InfluxDB

2.5、调用链监控

2.5.1、为什么需要调用链监控

微服务架构是一个分布式架构，微服务系统按业务划分服务单元，一个微服务系统往往有多个服务单元。由于服务单元数量众多，业务的复杂性较高，如果出现了错误和异常，很难去定位。主要体现在一个请求可能需要调用多个服务，而内部服务调用的复杂性决定了问题难以定位。所以在微服务架构中，必须实现分布式链路追踪，去跟进一个请求到底有哪些服务参与，参与的顺序又是怎样的，从而达到每个请求的步骤清晰可见，出了问题能够快速定位的目的。

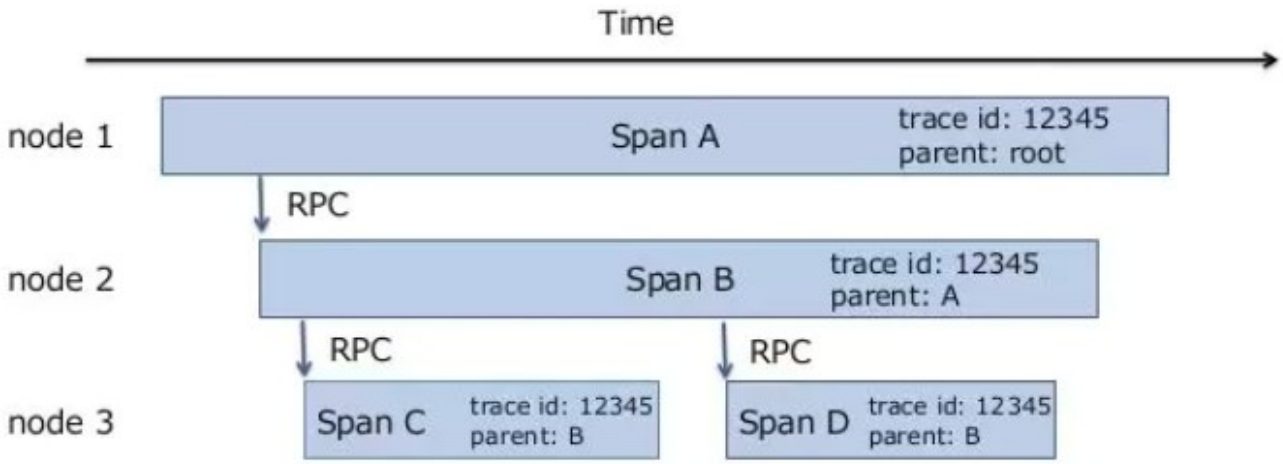
2.5.2、调用链监控原理

在调用链监控系统中几个核心的概念：

- Span
  - 基本工作单元，发送一个远程调度任务就会产生一个Span，Span是用一个64位ID唯一标识的，Trace是用另一个64位ID唯一标识的。Span还包含了其他信息，例如摘要、时间戳事件、Span的ID以及进程ID
- Trace
  - 由一系列Span组成的，呈树状结构。请求一个微服务系统的API接口，这个API接口需要调用多个服务单元，调用每个服务单元都会产生一个新的Span，所有由这个请求产生的Span组成了这个Trace
- Annotation
  - 用于记录一个事件，一些核心注解用于定义一个请求的开始和结束
    - cs-Client Sent

        - 客户端发送一个请求，这个注解描述了Span的开始
    - sr-Server Received
        - 服务端获得请求并准备开始处理它，如果将sr减去cs时间戳，便可以得到网络传输的时间
    - ss-Server Send
        - 服务端发送响应，改注解表明请求处理的完成，用ss的时间戳减去sr时间戳，便可以得到服务器处理请求的时间
    - cr-Client Receiver
        - 客户端接收响应，此时Span结束，如果cr的时间戳减去cs的时间戳，便可以得到整个请求所消耗的时间

示例



从图中可见，一次请求只有一个唯一的trace id=12345，在请求过程中的任何环节都不会改变。在这个请求的调用链中，SpanA调用了SpanB，然后SpanB又调用了SpanC和SpanD，每一次Span调用都会生成一个自己的span id，并且还会记录自己的上级span id是谁。通过这些id，整个链路基本上就都能标识出来了。

2.5.3、调用链监控实践

目前主流的调用链监控系统：

- **CAT**
    - 开源地址： https://github.com/dianping/cat

    - CAT是由大众点评开源的一款调用链监控系统，基于JAVA开发的。有很多互联网企业在使用，热度非常高。它有一个非常强大和丰富的可视化报表界面，这一点其实对于一款调用链监控系统而来非常的重要。在CAT提供的报表界面中有非常多的功能，几乎能看到你想要的任何维度的报表数据。
    - CAT有个很大的优势就是处理的实时性，CAT里大部分系统是分钟级统计
    - CAT主要提供的报表有

        - Transaction报表
            - 主要是监控一段代码运行情况，如：运行次数、QPS、错误次数、失败率、响应时间等
        - Event报表
            - 主要是监控一行代码/一个事件运行次数，如：程序中某个事件运行了多少次、错误了多少次等。Event报表的整体结构与Transaction报表几乎一样，只缺少响应时间的统计
        - Problem报表
            - 主要是统计项目在运行过程中出现的问题，根据Transaction与Event的数据分析出来系统可能出现的异常，比如访问较慢等
        - Heartbeat报表
            - 以一分钟为周期，定期向服务端汇报当前运行的一些状态，如：JVM状态、Memory、Thread等
        - Business报表
            - 业务监控报表，如订单指标、支付数据等业务指标。
- **Open Zipkin**
    - Zipkin由Twitter开源，支持的语言非常多，基于 Google Dapper 的论文设计而来，国内外很多公司都在用，文档资料也很丰富。
    - 官网： **https://zipkin.io/**
    - 开源地址： **https://github.com/openzipkin/zipkin/**
- 等等

2.6、服务治理

2.6.1、为什么需要服务治理

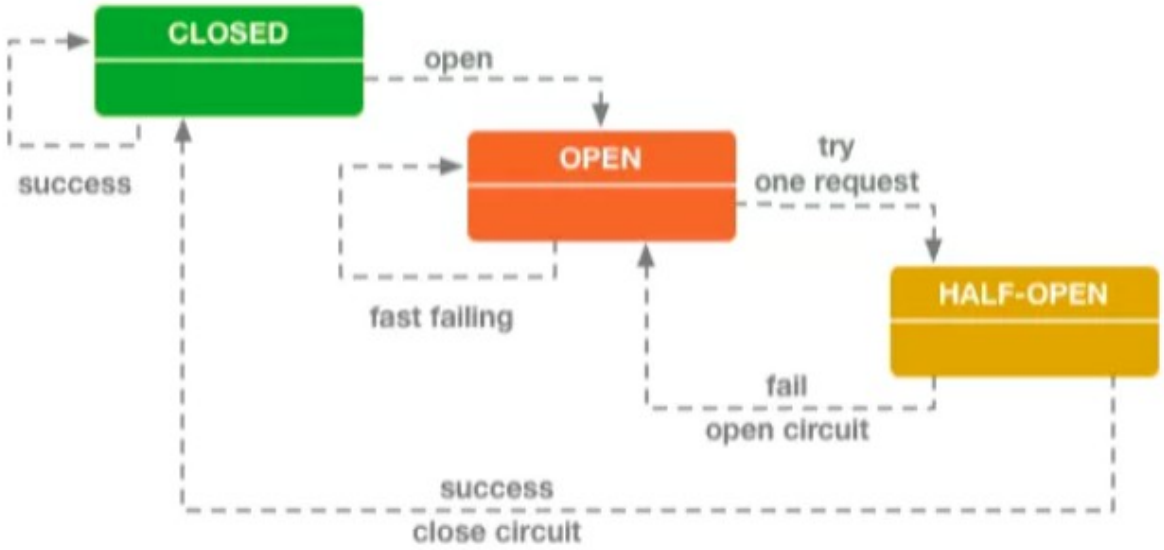
在复杂的分布式系统中，可能有几十个服务相互依赖，这些服务由于某些原因，例如机房的不可靠、网络服务商的不可靠性等，导致某个服务不可用。如果系统不隔离改不可用的服务，可能会导致整个系统不可用。这个现象也叫做服务雪崩。

假设，单个服务的故障概率是0.01%，也就是可用性是99.99%，如果一共有10个微服务，那么整体的可用性就是99.99%的十次方，得到的就是99.90%的可用性（故障概率也就是0.1%）。可见相对于之前的单体应用，整个系统可能发生故障的风险大幅提升。

2.6.2、容错隔离的措施有哪些

- **超时**
    - 这也是简单的容错方式。就是指在服务之间调用时，设置一个 主动超时时间，超过了这个时间阈值后，如果“被依赖的服务”还没有返回数据的话，“调用者”就主动放弃，防止因“被依赖的服务”的故障所影响。
- **限流**
    - 顾名思义，就是限制最大流量。系统能提供的最大并发有限，同时来的请求又太多，服务不过来啊，就只好排队限流了。
- **降级**
    - 这个与限流类似，一样是流量太多，系统服务不过来。这个时候可以可将不是那么重要的功能模块进行降级处理，停止服务，这样可以释放出更多的资源供给核心功能的去用。同时还可以对用户分层处理，优先处理重要用户的请求，比如VIP收费用户等
- **延迟处理**
    - 这个方式是指设置一个流量缓冲池，所有的请求先进入这个缓冲池等待处理，真正的服务处理方按顺序从这个缓冲池中取出请求依次处理，这种方式可以减轻后端服务的压力，但是对用户来说体验上有延迟。

- **熔断**
  - 可以理解成就像电闸的保险丝一样，当流量过大或者错误率过大的时候，保险丝就熔断了，链路就断开了，不提供服务了。当流量恢复正常，或者后端服务稳定了，保险丝会自动街上（熔断闭合），服务又可以正常提供了。这是一种很好的保护后端微服务的一种方式。
  - 熔断技术中有个很重要的概念就是：断路器，可以参考下图：



    - 断路器其实就是一个状态机原理，有三种状态：Closed（闭合状态，也就是正常状态）、Open（开启状态，也就是当后端服务出故障后链路断开，不提供服务的状态）、Half-Open(半闭合状态，就是允许一小部分流量进行尝试，尝试后发现服务正常就转为Closed状态，服务依旧不正常就转为Open状态)。

2.6.3、服务治理的应用

- Hystrix
  - 开源地址：https://github.com/Netflix/Hystrix
  - 示例程序：https://github.com/Netflix/Hystrix/wiki/How-To-Use
  - 工作原理：https://github.com/Netflix/Hystrix/wiki/How-it-Works