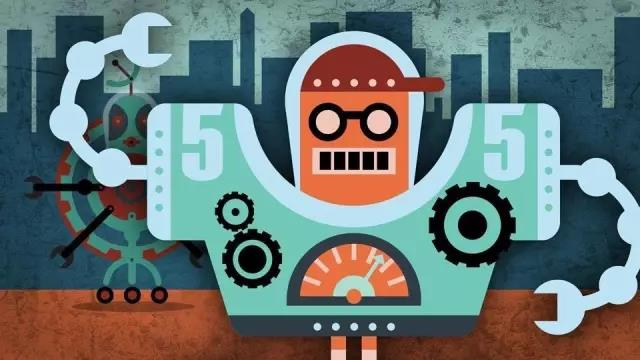
基于微服务和Docker容器技术的PaaS云平台架构设计



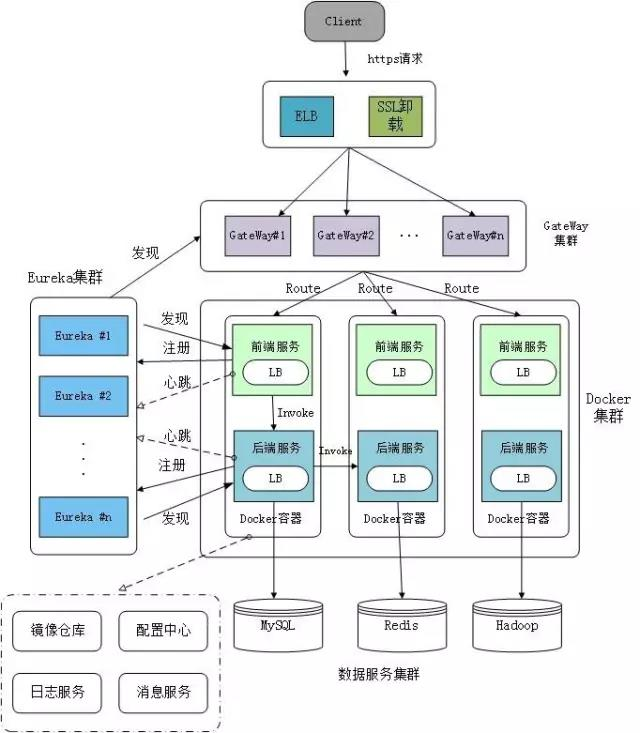
基于微服务架构和Docker容器技术的PaaS云平台建设**目标**是**给我们的开发人员提供一套服务快速开发、部署、运维管理、持续开发持续集成的流程。平台提供基础设施、中间件、数据服务、云服务器等资源，开发人员只需要开发业务代码并提交到平台代码库，做一些必要的配置，系统会自动构建、部署，实现应用的敏捷开发、快速迭代。**在系统架构上，**PaaS云平台主要分为微服务架构、Docker容器技术、DveOps**三部分，这篇文章重点介绍**微服务架构的实施**。

实施微服务需要投入大量的技术力量来开发基础设施，这对很多公司来说显然是不现实的，别担心，业界已经有非常优秀的开源框架供我们参考使用。目前业界比较成熟的微服务框架有Netflix、Spring Cloud和阿里的Dubbo等。

Spring Cloud是基于Spring Boot的一整套实现微服务的框架，它提供了开发微服务所需的组件，跟Spring Boot一起使用的话开发微服务架构的云服务会变的很方便。Spring Cloud包含很多子框架，其中Spring Cloud Netflix是其中的一套框架，在我们的微服务架构设计中，就使用了很多Spring Cloud Netflix框架的组件。

Spring Cloud Netflix项目的时间还不长，相关的文档资料很少，博主当时研究这套框架啃了很多英文文档，简直痛苦不堪。对于刚开始接触这套框架的同学，要搭建一套微服务应用架构，可能会不知道如何下手，接下来介绍我们的微服务架构搭建过程以及需要那些框架或组件来支持微服务架构。

为了直接明了的展示微服务架构的组成及原理，博主画了一张系统架构图，如下：



从上图可以看出，微服务访问大致路径为：外部请求 → 负载均衡 → 服务网关（GateWay）→ 微服务 → 数据服务/消息服务。**服务网关和微服务都会用到服务注册和发现来调用依赖的其他服务，各服务集群都能通过配置中心服务来获得配置信息。**

服务网关（GateWay）

网关是外界系统（如：客户端浏览器、移动设备等）和企业内部系统之间的一道门，**所有的客户端请求通过网关访问后台服务。为了应对高并发访问，服务网关以集群形式部署，这就意味着需要做负载均衡**，我们采用了亚马逊EC2作为虚拟云服务器，采用ELB（Elastic Load Balancing）做负载均衡。

EC2具有自动配置容量功能，当用户流量达到尖峰，EC2可以自动增加更多的容量以维持虚拟主机的性能。ELB弹性负载均衡，在多个实例间自动分配应用的传入流量。为了保证安全性，客户端请求需要使用https加密保护，这就需要我们进行SSL卸载,使用Nginx对加密请求进行卸载处理。外部请求经过ELB负载均衡后路由到GateWay集群中的某个GateWay服务，由GateWay服务转发到微服务。**服务网关作为内部系统的边界，它有以下基本能力**：

1. **动态路由**：动态的将请求路由到所需要的后端服务集群。虽然内部是复杂的分布式微服务网状结构，但是外部系统从网关看就像是一个整体服务，网关屏蔽了后端服务的复杂性。
2. **限流和容错**：为每种类型的请求分配容量，当请求数量超过阀值时抛掉外部请求，限制流量，保护后台服务不被大流量冲垮；当内部服务出现故障时直接在边界创建一些响应，集中做容错处理，而不是将请求转发到内部集群，保证用户良好的体验。
3. **身份认证和安全性控制**：对每个外部请求进行用户认证，拒绝没有通过认证的请求，还能通过访问模式分析，实现反爬虫功能。
4. **监控**：网关可以收集有意义的数据和统计，为后台服务优化提供数据支持。
5. **访问日志**：网关可以收集访问日志信息，比如访问的是哪个服务？处理过程（出现什么异常）和结果？花费多少时间？通过分析日志内容，对后台系统做进一步优化。

**我们采用Spring Cloud Netflix框架的开源组件Zuul来实现网关服务**。Zuul使用一系列不同类型的过滤器（Filter），通过重写过滤器，使我们能够灵活的实现网关（GateWay）的各种功能。

服务注册与发现

由于微服务架构是由一系列职责单一的细粒度服务构成的网状结构，服务之间通过轻量机制进行通信，这就引入了服务注册与发现的问题，服务的提供方要注册报告服务地址，服务调用放要能发现目标服务。我们的微服务架构中使用了**Eureka组件来实现服务的注册与发现**。

**所有的微服务（通过配置Eureka服务信息）到Eureka服务器中进行注册，并定时发送心跳进行健康检查，Eureka默认配置是30秒发送一次心跳，表明服务仍然处于存活状态**，发送心跳的时间间隔可以通过Eureka的配置参数自行配置，Eureka服务器在接收到服务实例的最后一次心跳后，需要等待90秒（默认配置90秒，可以通过配置参数进行修改）后，才认定服务已经死亡（即连续3次没有接收到心跳），在Eureka自我保护模式关闭的情况下会清除该服务的注册信息。

所谓的自我保护模式是指，出现网络分区、Eureka在短时间内丢失过多的服务时，会进入自我保护模式，即一个服务长时间没有发送心跳，Eureka也不会将其删除。自我保护模式默认为开启，可以通过配置参数将其设置为关闭状态。

**Eureka服务以集群的方式部署**，**集群内的所有Eureka节点会定时自动同步微服务的注册信息，这样就能保证所有的Eureka服务注册信息保持一致。**那么在Eureka集群里，Eureka节点是如何发现其他节点的呢？我们通过**DNS服务器来建立所有Eureka节点的关联**，在部署Eureka集群之外还需要搭建DNS服务器。

当网关服务转发外部请求或者是后台微服务之间相互调用时，会去Eureka服务器上查找目标服务的注册信息，发现目标服务并进行调用，这样就形成了服务注册与发现的整个流程。Eureka的配置参数数量很多，多达上百个，博主会在另外的文章里详细说明。

微服务部署

微服务是一系列职责单一、细粒度的服务，是将我们的业务进行拆分为独立的服务单元，伸缩性好，耦合度低，**不同的微服务可以用不同的语言开发，每一个服务处理的单一的业务。微服务可以划分为前端服务（也叫边缘服务）和后端服务（也叫中间服务），前端服务是对后端服务做必要的聚合和剪裁后暴露给外部不同的设备（PC、Phone等），所有的服务启动时都会到Eureka服务器进行注册，服务之间会有错综复杂的依赖关系。**

**当网关服务转发外部请求调用前端服务时，通过查询服务注册表就可以发现目标服务进行调用，前端服务调用后端服务时也是同样的道理，一次请求可能涉及到多个服务之间的相互调用。由于每个微服务都是以集群的形式部署，服务之间相互调用的时候需要做负载均衡，因此每个服务中都有一个LB组件用来实现负载均衡。**

**微服务以镜像的形式，运行在Docker容器中**。Docker容器技术让我们的服务部署变得简单、高效。传统的部署方式，需要在每台服务器上安装运行环境，如果我们的服务器数量庞大，在每台服务器上安装运行环境将是一项无比繁重的工作，一旦运行环境发生改变，就不得不重新安装，这简直是灾难性的。

而使用Docker容器技术，我们只需要将所需的基础镜像（jdk等）和微服务生成一个新的镜像，将这个最终的镜像部署在Docker容器中运行，这种方式简单、高效，能够快速部署服务。每个Docker容器中可以运行多个微服务，Docker容器以集群的方式部署，使用Docker Swarm对这些容器进行管理。我们创建一个镜像仓库用来存放所有的基础镜像以及生成的最终交付镜像，在镜像仓库中对所有镜像进行管理。

服务容错

微服务之间存在错综复杂的依赖关系，一次请求可能会依赖多个后端服务，在实际生产中这些服务可能会产生故障或者延迟，在一个高流量的系统中，一旦某个服务产生延迟，可能会在短时间内耗尽系统资源，将整个系统拖垮，因此一个服务如果不能对其故障进行隔离和容错，这本身就是灾难性的。**我们的微服务架构中使用了Hystrix组件来进行容错处理。**Hystrix是Netflix的一款开源组件，**它通过熔断模式、隔离模式、回退（fallback）和限流等机制对服务进行弹性容错保护，保证系统的稳定性。**

1. **熔断模式**：熔断模式原理类似于电路熔断器，当电路发生短路时，熔断器熔断，保护电路避免遭受灾难性损失。当服务异常或者大量延时，满足熔断条件时服务调用方会主动启动熔断，执行fallback逻辑直接返回，不会继续调用服务进一步拖垮系统。熔断器默认配置服务调用错误率阀值为50%，超过阀值将自动启动熔断模式。服务隔离一段时间以后，熔断器会进入半熔断状态，即允许少量请求进行尝试，如果仍然调用失败，则回到熔断状态，如果调用成功，则关闭熔断模式。
2. **隔离模式**：Hystrix默认采用线程隔离，不同的服务使用不同的线程池，彼此之间不受影响，当一个服务出现故障耗尽它的线程池资源，其他的服务正常运行不受影响，达到隔离的效果。例如我们通过andThreadPoolKey配置某个服务使用命名为TestThreadPool的线程池，实现与其他命名的线程池隔离。
3. **回退（fallback）**：fallback机制其实是一种服务故障时的容错方式，原理类似Java中的异常处理。只需要继承HystixCommand并重写getFallBack()方法，在此方法中编写处理逻辑，比如可以直接抛异常（快速失败），可以返回空值或缺省值，也可以返回备份数据等。当服务调用出现异常时，会转向执行getFallBack()。有以下几种情况会触发fallback：

程序抛出非HystrixBadRequestExcepption异常，当抛出HystrixBadRequestExcepption异常时，调用程序可以捕获异常，没有触发fallback，当抛出其他异常时，会触发fallback；

程序运行超时；

熔断启动；

线程池已满。

1. **限流**： 限流是指对服务的并发访问量进行限制，设置单位时间内的并发数，超出限制的请求拒绝并fallback，防止后台服务被冲垮。

Hystix使用命令模式HystrixCommand包装依赖调用逻辑，这样相关的调用就自动处于Hystrix的弹性容错保护之下。调用程序需要继承HystrixCommand并将调用逻辑写在run()中，使用execute()（同步阻塞）或queue()（异步非阻塞）来触发执行run()。

动态配置中心

微服务有很多依赖配置，某些配置参数在服务运行期间可能还要动态修改，比如：根据访问流量动态调整熔断阀值。传统的实现信息配置的方法，比如放在xml、yml等配置文件中，和应用一起打包，每次修改都要重新提交代码、打包构建、生成新的镜像、重新启动服务，效率太低，这样显然是不合理的，因此我们需要搭建一个动态配置中心服务支持微服务动态配置。

我们使用Spring Cloud的**configserver服务帮我们实现动态配置中心的搭建。**我们开发的微服务代码都存放在Git服务器私有仓库里面，所有需要动态配置的配置文件存放在Git服务器下的configserver（配置中心，也是一个微服务）服务中，部署到Docker容器中的微服务从Git服务器动态读取配置文件的信息。

当本地Git仓库修改代码后push到Git服务器仓库，Git服务端hooks(post-receive，在服务端完成代码更新后会自动调用)自动检测是否有配置文件更新，如果有，Git服务端通过消息队列给配置中心（configserver，一个部署在容器中的微服务）发消息，通知配置中心刷新对应的配置文件。这样微服务就能获取到最新的配置文件信息，实现动态配置。

以上这些框架或组件是支撑实施微服务架构的核心，在实际生产中，我们还会用到很多其他的组件，比如日志服务组件、消息服务组件等等，根据业务需要自行选择使用。在我们的微服务架构实施案例中，参考使用了很多Spring Cloud Netflix框架的开源组件，**主要包括Zuul（服务网关）、Eureka（服务注册与发现）、Hystrix（服务容错）、Ribbon（客户端负载均衡）等。**这些优秀的开源组件，为我们实施微服务架构提供了捷径。