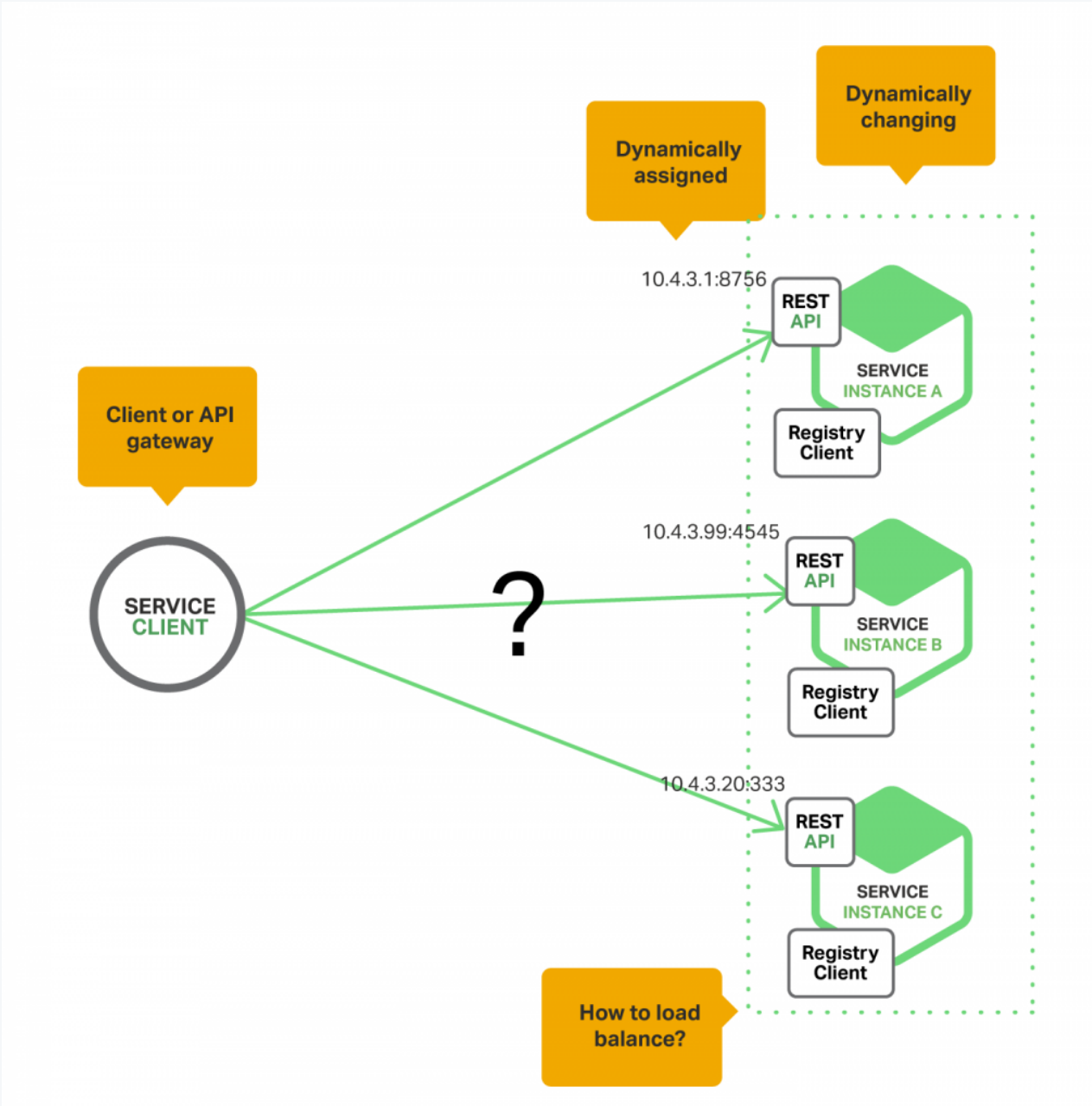
17
APR 2016

微服务系统中的服务发现机制

IN 后端编程,架构

为什么要使用服务发现？

我们可以想象一下，当我们需要远程的访问REST API或者Thrift API时，我们必须得知道服务的网络地址（IP Address和port）。传统的应用程序都是运行在固定的物理机器上，IP Address和端口号都是相对固定的。可以通过配置文件方式来实现不定期更新的Ip Address和端口号。但是，在基于云的微服务应用中，这是一个非常难以解决的问题。如下图所示：

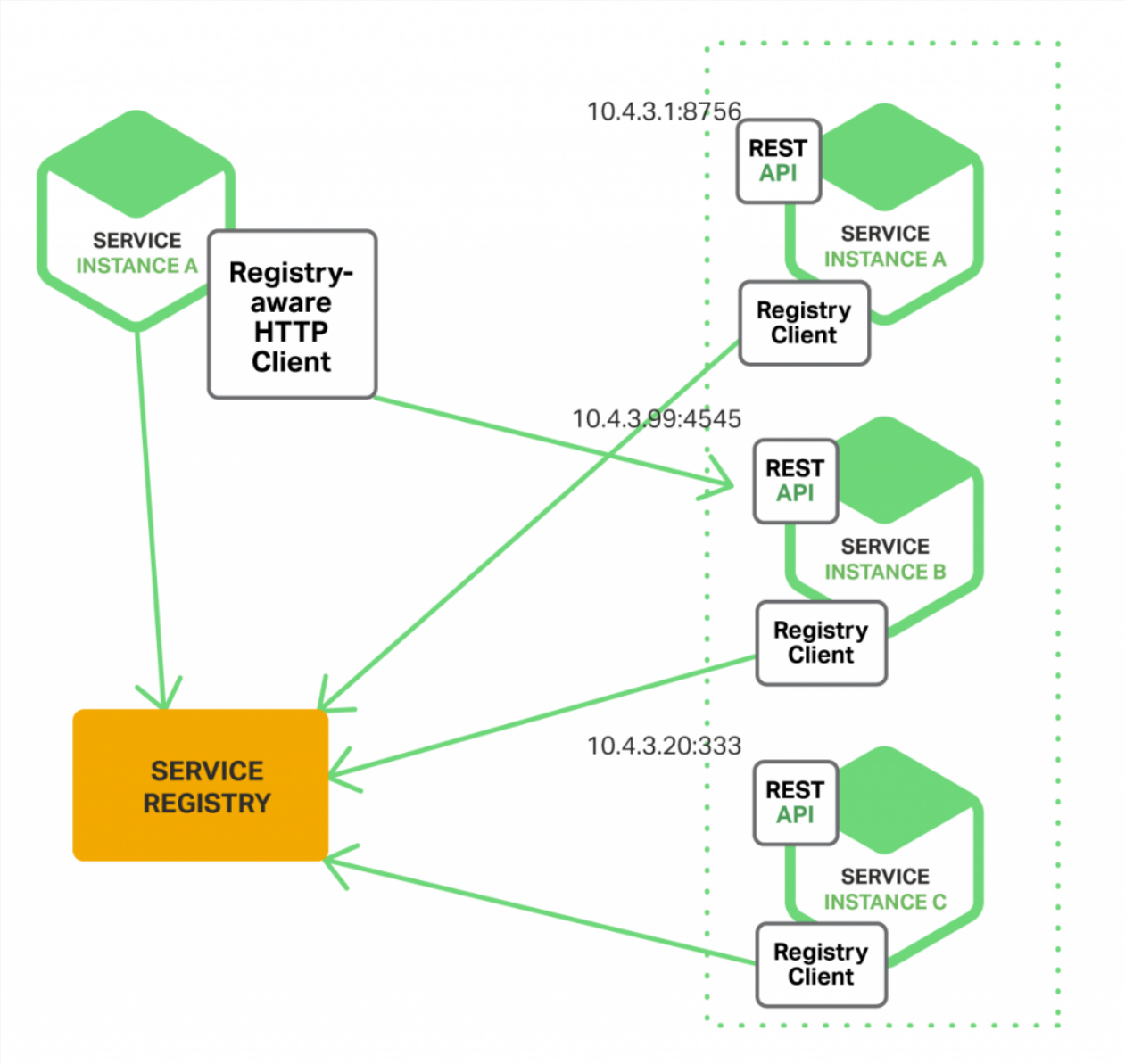


在基于云的微服务应用中，服务实例的网络地址（IP Address和Port）是动态分配的，并且由于系统的auto-scaling, failures 和 upgrades等因数，一些服务运行的实例数量也是动态变化的。因此，客户端代码需要使用一个非常精细和准确的服务发现机制。

有两种主要的服务发现方式：客户端发现（client-side discovery）和服务器端发现（server-side discovery）。

客户端发现方式

在使用客户端发现方式时，客户端通过查询服务注册中心，获取可用的服务的实际网络地址（IP Adress 和 端口号）。然后通过负载均衡算法来选择一个可用的服务实例，并将请求发送至该服务。下图显示了客户端发现方式的结构图：

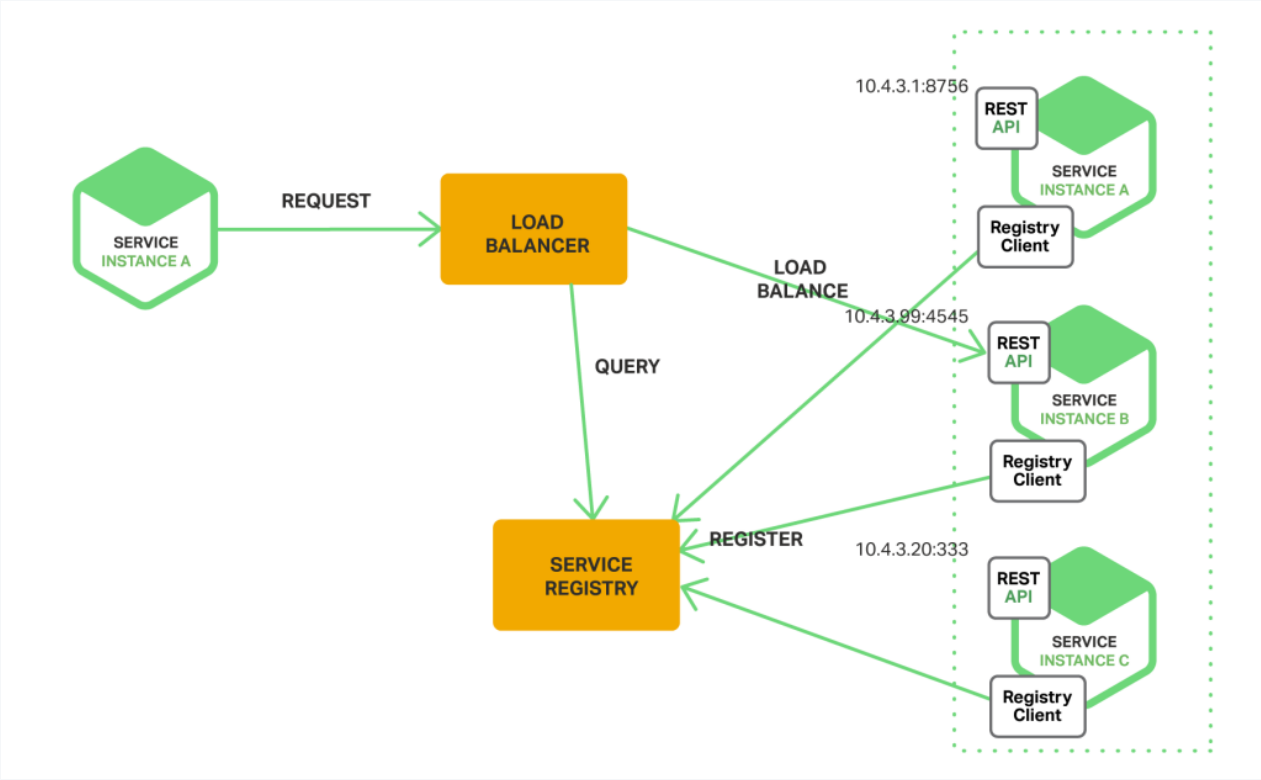


在服务启动的时候，向服务注册中心注册服务；在服务停止的时候，向服务注册中心注销服务。服务注册的一个典型的实现方式就是通过heartbeat机制定时刷新。Netflix OSS 就是使用客户端发现方式的一个很好的例子。 Netflix Eureka是一个服务注册中心。它提供了一个管理和查询可用服务的 REST API。 负载均衡功能是通过Netflix Ribbon（是一个IPC客户端）和Eureka一起共同实现的。在文章的后面将深入的介绍Eureka。

客户端发现方式的优缺点。由于客户端知道所有可用的服务的实际网络地址，所以可以非常方便的实现负载均衡功能（比如：一致性哈希）。但是这种方式有一个非常明显的缺点就是具有非常强的耦合性。针对不同的语言，每个服务的客户端都得实现一套服务发现的功能。

服务端发现方式

另外一种服务发现的方式就是Server-Side Discovery Pattern，下图展示了这种方式的架构示例图：



客户端向load balancer 发送请求。load balancer 查询服务注册中心找到可用的服务，然后转发请求到该服务上。和客户端发现一样，服务都要到注册中心进行服务注册和注销。AWS的弹性负载均衡（Elastic Load Balancer–ELB）就是服务端发现的一个例子。ELB通常是用于为外网服务提供负载平衡的。当然你也可以使用ELB为内部虚拟私有云（VPC）提供负载均衡服务。客户端通过使用DNS名称，发送HTTP或TCP请求到ELB。ELB为EC2或ECS集群提供负载均衡服务。AWS并没有提供单独的服务注册中心。而是通过ELB实现EC2实例和ECS容器的注册的。

NGINX不仅可以作为HTTP反向代理服务器和负载均衡器，也可以用来作为一个服务发现的负载均衡器。例如，这篇博客（Scalable Architecture DR CoN: Docker, Registrator, Consul, Consul Template and Nginx）介绍如何使用Consul Template 动态的配置NGINX功能。

Kubernetes 和 Marathon 是在通过集群中每个节点都运行一个代理来实现服务发现的功能的，代理的角色就是server-side discovery,客户端通过使用主机的IP Address和Port向Proxy发送请求，Proxy再将请求转发到集群中任何一个可用的服务上。

服务器端发现方式的优点是，服务的发现逻辑对客户端是透明的。客户只需简单的向load balancer发送请求就可以了。这就避免了为每种不同语言的客户端实现一套发现的逻辑。此外，许多软件都内置实现了这种功能。这种方式的一个最大的缺点是，你必须得关心该组件的高可用性。

服务注册中心

服务注册中心是服务发现的核心。它保存了各个可用服务实例的网络地址（IP Address和Port）。服务注册中心必须要有高可用性和实时更新功能。

上面提到的 Netflix Eureka 就是一个服务注册中心。它提供了服务注册和查询服务信息的REST API。服务通过使用POST请求注册自己的IP Address和Port。每30秒发送一个PUT请求刷新注册信息。通过DELETE请求注销服务。客户端通过GET请求获取可用的服务实例信息。

Netflix的高可用（Netflix achieves high availability ）是通过在Amazon EC2运行多个实例来实现的,每一个Eureka服务都有一个弹性IP Address。当Eureka服务启动时，有DNS服务器动态的分配。Eureka客户端通过查询 DNS来获取Eureka的网络地址（IP Address和Port）。一般情况下，都是返回和客户端在同一个可用区Eureka服务器地址。

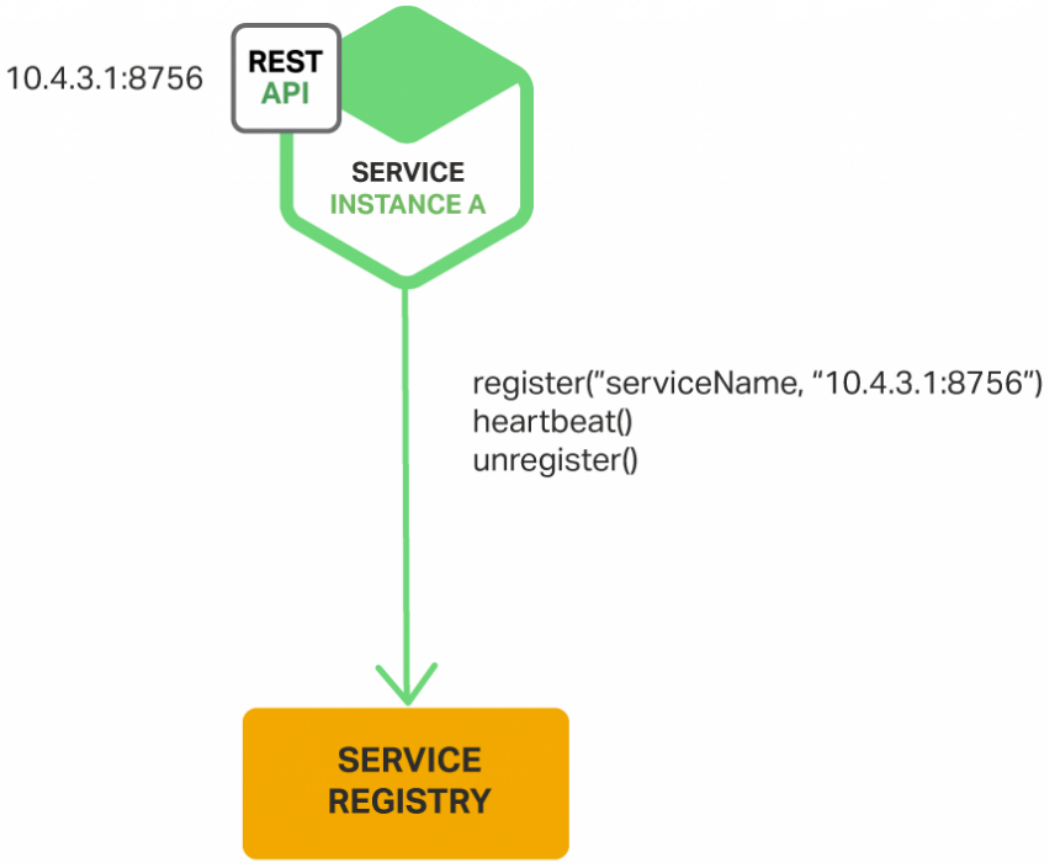
其他能够作为服务注册中心的有：

- etcd —— 高可用，分布式，强一致性的，key-value，Kubernetes和Cloud Foundry都是使用了etcd。
- consul ——一个用于discovering和 configuring的工具。它提供了允许客户端注册和发现服务的API。Consul可以进行服务健康检查，以确定服务的可用性。
- zookeeper —— 在分布式应用中被广泛使用，高性能的协调服务。 Apache Zookeeper 最初为Hadoop的一个子项目，但现在是一个顶级项目。

我们已经了解了服务注册中心的概念，接下来我们看看服务是如果注册到注册中心的。有两种不同的方式来处理服务的注册和注销。一种是服务自己主动注册-自己注册（self-registration）。另一种是通过其他组件来管理服务的注册-第三方注册（third-party registration）。

Self-Registration

使用Self-Registration的方式注册，服务实例必须自己主动的到注册中心注册和注销。比如可以使用heartbeat机制了实现。 下图为这种方式的示意图：

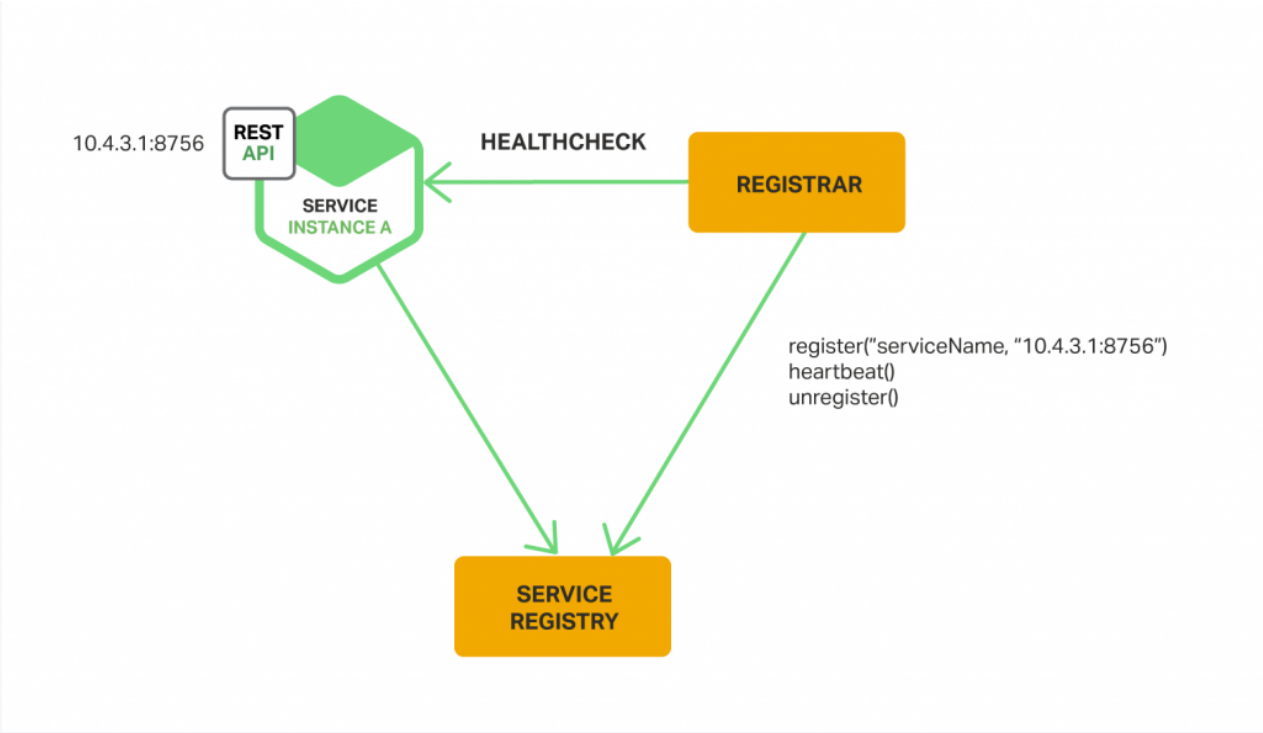


Netflix OSS Eureka client就是使用这种方式进行服务注册的。Eureka的客户端处理了服务注册和注销的所有工作。

Self-Registration方式的优缺点：一个明显的优点就是，非常简单，不需要任何其它辅助组件。而缺点也是比较明显的，使得各个服务和注册中心的耦合度比较高。服务的不同语言的客户端都得实现注册和注销的逻辑。另一种服务注册方式，可以达到解耦的功能，就是third-party registration方式。

Third-Party Registration

使用Third-Party方式进行服务注册时，服务本身不必关心注册和注销功能。而是通过其他组件（service registrarhandles）来实现服务注册功能。可以通过如事件订阅等方式来监控服务的状态，如果发现一个新的服务实例运行，就向注册中心注册该服务，如果监控到某一服务停止了，就向注册中心注销该服务。下图显示了这种方式的结构图示意图：



third-party Registration方式的优点是使服务和注册中心解耦，不用为每种语言实现服务注册的逻辑。这种方式的缺点就是必须得考虑该组件的高可用性。

总结

在微服务的应用系统中，服务实例的数量是动态变化。各服务实例动态分配网络地址（IP Address 和 Port）。因此，为了为客户端能够访问到服务，就必须要有一种服务的发现机制。

服务发现的核心是服务注册中心。服务注册中心保存了各个服务可用的实例的相关信息。服务注册中心提供了管理API和查询API。使用管理API进行服务注册、注销。系统的其他组件可以通过查询API来获得当前可用的服务实例信息。
有两种主要的服务发现方式：客户端发现（client-side service discovery）和服务端发现（ server-side discovery）。在使用客户端服务发现的方式中，客户通过查询服务注册中心，选择一个可用的服务实例。在使用服务器端发现系统中，客户访问Router/load balancer，通过Router/load balancer查询服务注册中心，并将请求转发到一个可用服务实例上。
服务注册和注销的方式也有两种。一种是服务自己主动的将自己注册到服务注册中心，称为self-registration。另一种是通过其他组件来处理服务的注册和注销，称为third-party registration。

在有些环境中，服务发现功能需要自己通过服务注册中心（比如：Netflix Eureka, etcd, Apache Zookeeper）实现，而有些环境中，服务发现功能是内置的。例如，Kubernetes和Marathon。
Nginx可以作为HTTP反向代理和负载平衡器，也可以用来作为一个服务发现的负载均衡器。通过向nginx推送routing information来修改nginx的配置，比如使用：Consul Template动态修改NGINX的配置. NGINX Plus 也支持动态修改配置功能。

- 在今后的文章中，我们将继续深入分析微服务的其他方面的内容。

原文作者发布的关于微服务系列文章的地址：

1. Introduction to Microservices
2. Building Microservices: Using an API Gateway
3. Building Microservices: Inter-Process Communication in a Microservices Architecture

4. Service Discovery in a Microservices Architecture
5. Event-Driven Data Management for Microservices
6. Choosing a Microservices Deployment Strategy
7. Refactoring a Monolith into Microservices

作者信息：

原文作者：Chris Richardson，他是早期基于Java的Amazonite EC2 PaaS平台CloudFoundry.com的创始人。现在他为企业提供如何开发和部署应用的咨询服务

原文链接：https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

翻译自MaxLeap团队_云服务研发成员：Lam Yu

分布式 微服务 服务发现 服务注册 注册中心