cos对比

原创 琦彦 于 2019-08-22 21:11:09 发布 205158 收藏 1111 版权

分类专栏： 微服务-SpringCloud和Istio 文章标签： 注册中心 ZooKeeper Eureka Consul Nacos

前言

服务注册中心本质上是为了解耦服务提供者和服务消费者。对于任何一个微服务 ，原则上都应存在或者支持多个提供者，这是由微服务的分布式属性决定的。更进一步，为了支持弹性扩缩容特性，一个微服务的提供者的数量和分布往往是动态变化的，也是无法预先确定的。因此，原本在单体应用阶段常用的静态LB机制就不再适用了，需要引入额外的组件来管理微服务提供者的注册与发现，而这个组件就是服务注册中心。

CAP理论

CAP理论是 分布式 架构中重要理论

> - 一致性(Consistency) (所有节点在同一时间具有相同的数据)
> - 可用性(Availability) (保证每个请求不管成功或者失败都有响应)
> - 分隔容忍(Partition tolerance) (系统中任意信息的丢失或失败不会影响系统的继续运作)

关于

P的理解，我觉得是在整个系统中某个部分，挂掉了，或者宕机了，并不影响整个系统的运作或者说使用，

而可用性是，某个系统的某个节点挂了，但是并不影响系统的接受或者发出请求，CAP 不可能都取，只能取其中2个

原因是

如果C是第一需求的话，那么会影响A的性能，因为要数据同步，不然请求结果会有差异，但是数据同步会消耗时间，期间可用性就会降低。

如果A是第一需求，那么只要有一个服务在，就能正常接受请求，但是对与返回结果变不能保证，原因是，在分布式部署的时候，数据一致的过程不可能想切线路那么快。

再如果，同事满足一致性和可用性，那么分区容错就很难保证了，也就是单点，也是分布式的基本核心，好了，明白这些理论，就可以在相应的场景选取服务注册与发现了

服务注册中心解决方案

设计或者选型一个服务注册中心，首先要考虑的就是服务注册与发现机制。纵观当下各种主流的服务注册中心解决方案，大致可归为三类：

- 应用内：直接集成到应用中，依赖于应用自身完成服务的注册与发现，最典型的是Netflix提供的Eureka
- 应用外：把应用当成黑盒，通过应用外的某种机制将服务注册到注册中心，最小化对应用的侵入性，比如Airbnb的SmartStack，HashiCorp的Consul
- DNS：将服务注册为DNS的SRV记录，严格来说 是一种特殊的应用外注册方式 SkyDNS 是其中的代表

注2：由于DNS固有的缓存缺陷，本文不对第三类注册方式作深入探讨。

除了基本的服务注册与发现机制，从开发和运维角度，至少还要考虑如下五个方面：

- 测活：服务注册之后，如何对服务进行测活以保证服务的可用性?
- 负载均衡：当存在多个服务提供者时，如何均衡各个提供者的负载?
- 集成：在服务提供端或者调用端，如何集成注册中心?
- 运行时依赖：引入注册中心之后，对应用的运行时环境有何影响?
- 可用性：如何保证注册中心本身的可用性，特别是消除单点故障?

主流注册中心产品

> 软件产品特性并非一成不变，如果发现功能特性有变更，欢迎评论指正

	Nacos	Eureka	Consul	CoreDNS	Zookeeper
一致性协议	CP+AP	AP	CP	—	CP
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	—	Keep Alive
负载均衡策略	权重/ metadata/Selector	Ribbon	Fabio	RoundRobin	—
雪崩保护	有	有	无	无	无
自动注销实例	支持	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心同步	支持	不支持	支持	不支持	不支持
SpringCloud集成	支持	支持	支持	不支持	支持
Dubbo集成	支持	不支持	支持	不支持	支持
K8S集成	支持	不支持	支持	支持	不支持

> - Consul是支持自动注销服务实例， 请见文档： https://www.consul.io/api-docs/agent/service，在check的 DeregisterCriticalServiceAfter 这个参数-- 感谢@超帅的菜鸟博主提供最新信息
> - 新版本的Dubbo也扩展了对 Consul 的支持。 参考: https://github.com/apache/dubbo/tree/master/dubbo-registry

Apache Zookeeper –> CP

与 Eureka 有所不同，Apache Zookeeper 在设计时就紧遵CP原则，即任何时候对 Zookeeper 的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性，但是 Zookeeper 不能保证每次服务请求都是可达的。

半数以上服务器节点不可用（例如有三个节点，如果节点一检测到节点三挂了，节点二也检测到节点三挂了，那这个节点才算是真的挂了），那么将无法处理该请求。所以说，Zookeeper 不能保证服务可用性。

当然，在大多数分布式环境中，尤其是涉及到数据存储的场景，数据一致性应该是首先被保证的，这也是 Zookeeper 设计紧遵CP原则的另一个原因。

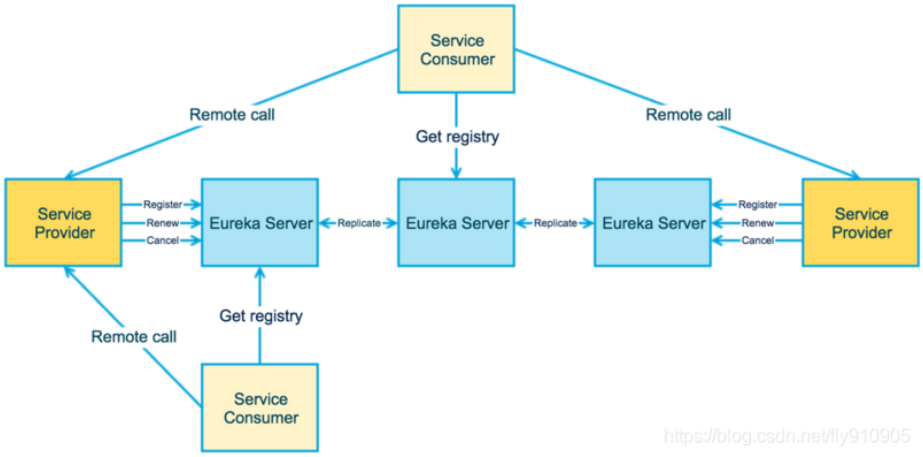
但是对于服务发现来说，情况就不太一样了，针对同一个服务，即使注册中心的不同节点保存的服务提供者信息不尽相同，也并不会造成灾难性的后果。

因为对于服务消费者来说，能消费才是最重要的，消费者虽然拿到可能不正确的服务实例信息后尝试消费一下，也要胜过因为无法获取实例信息而不去消费，导致系统异常要好（淘宝的双十一，京东的618就是紧遵AP的最好参照）。

当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30~120s，而且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。

在云部署环境下， 因为网络问题使得zk集群失去master节点是大概率事件，虽然服务能最终恢复，但是漫长的选举事件导致注册长期不可用是不能容忍的。

Spring Cloud Eureka –> AP



Spring Cloud Netflix 在设计 Eureka 时就紧遵AP原则（尽管现在2.0发布了，但是由于其闭源的原因，但是目前 Ereka 1.x 任然是比较活跃的）。

Eureka Server 也可以运行多个实例来构建集群，解决单点问题，但不同于 ZooKeeper 的选举 leader 的过程，Eureka Server 采用的是Peer to Peer 对等通信。这是一种去中心化的架构，无 master/slave 之分，每一个 Peer 都是对等的。在这种架构风格中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的 serviceUrl 指向其他节点。每个节点都可被视为其他节点的副本。

在集群环境中如果某台 Eureka Server 宕机，Eureka Client 的请求会自动切换到新的 Eureka Server 节点上，当宕机的服务器重新恢复后，Eureka 会再次将其纳入到服务器集群管理之中。当节点开始接受客户端请求时，所有的操作都会在节点间进行复制（replicate To Peer）操作，将请求复制到该 Eureka Server 当前所知的其它所有节点中。

当一个新的 Eureka Server 节点启动后，会首先尝试从邻近节点获取所有注册列表信息，并完成初始化。Eureka Server 通过 getEurekaServiceUrls() 方法获取所有的节点，并且会通过心跳契约的方式定期更新。

in-seconds 进行自定义配置）。

当 Eureka Server 节点在短时间内丢失过多的心跳时，那么这个节点就会进入自我保护模式。

Eureka的集群中，只要有一台Eureka还在，就能保证注册服务可用（保证可用性），只不过查到的信息可能不是最新的（不保证强一致性）。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1. Eureka不再从注册表中移除因为长时间没有收到心跳而过期的服务；
2. Eureka仍然能够接受新服务注册和查询请求，但是不会被同步到其它节点上（即保证当前节点依然可用）；
3. 当网络稳定时，当前实例新注册的信息会被同步到其它节点中；

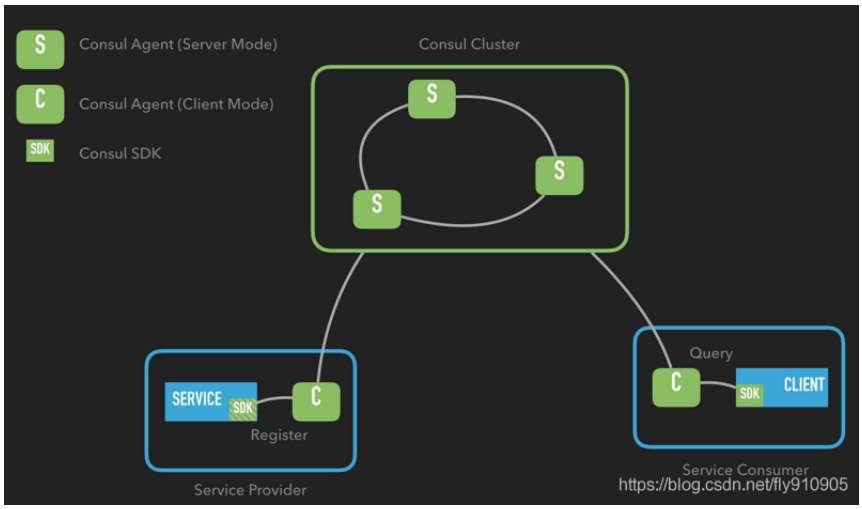
因此，Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使得整个注册服务瘫痪。

Consul ：

Consul 是 HashiCorp 公司推出的开源工具，用于实现分布式系统的服务发现与配置。Consul 使用 Go 语言编写，因此具有天然可移植性（支持Linux、windows和Mac OS X）。

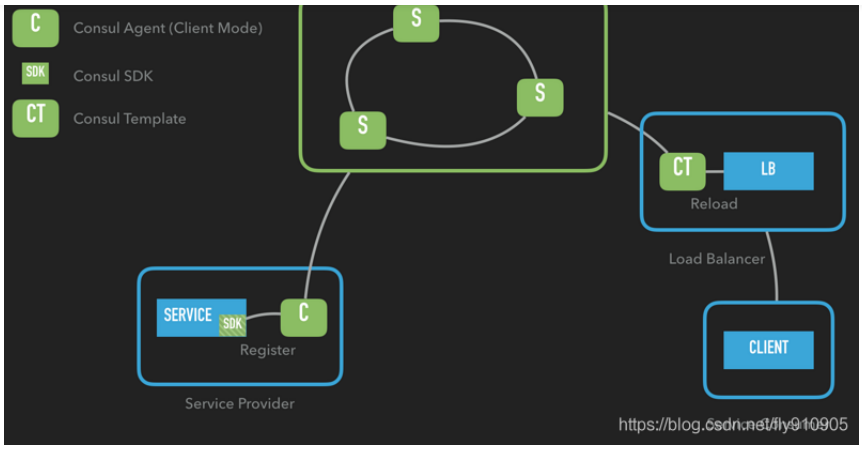
Consul 内置了服务注册与发现框架、分布一致性协议实现、健康检查、Key/Value 存储、多数据中心方案，不再需要依赖其他工具（比如 ZooKeeper 等），使用起来也较为简单。

Consul 遵循CAP原理中的CP原则，保证了强一致性和分区容错性，且使用的是Raft算法，比zookeeper使用的Paxos算法更加简单。虽然保证了强一致性，但是可用性就相应下降了，例如服务注册的时间会稍长一些，因为 Consul 的 raft 协议要求必须过半数的节点都写入成功才认为注册成功；在leader挂掉了之后，重新选举出leader之前会导致Consul 服务不可用。



默认依赖于SDK

Consul本质上属于应用外的注册方式，但可以通过SDK简化注册流程。而服务发现恰好相反，默认依赖于SDK，但可以通过Consul Template（下文会提到）去除SDK依赖。



Consul Template

Consul Template

Consul，默认服务调用者需要依赖Consul SDK来发现服务，这就无法保证对应用的零侵入性。

所幸通过Consul Template，可以定时从Consul集群获取最新的服务提供者列表并刷新LB配置（比如nginx的upstream），这样对于服务调用者而言，只需要配置一个统一的服务调用地址即可。

Consul强一致性(C)带来的是：

1. **服务注册相比Eureka会稍慢一些。因为Consul的raft协议要求必须过半数的节点都写入成功才认为注册成功**
2. **Leader挂掉时，重新选举期间整个consul不可用。保证了强一致性但牺牲了可用性。**

Eureka保证高可用(A)和最终一致性：

1. **服务注册相对要快，因为不需要等注册信息replicate到其他节点，也不保证注册信息是否replicate成功**
2. **当数据出现不一致时，虽然A, B上的注册信息不完全相同，但每个Eureka节点依然能够正常对外提供服务，这会出现查询服务信息时如果请求A查不到，但请求B就能查到。如此保证了可用性但牺牲了一致性。**

其他方面，eureka就是个servlet程序，跑在servlet容器中; Consul则是go编写而成。

Nacos ：

Nacos是阿里开源的，Nacos 支持基于 DNS 和基于 RPC 的服务发现。在Spring Cloud中使用Nacos，只需要先下载 Nacos 并启动 Nacos server，Nacos只需要简单的配置就可以完成服务的注册发现。

Nacos除了服务的注册发现之外，还支持动态配置服务。动态配置服务可以让您以中心化、外部化和动态化的方式管理所有环境的应用配置和服务配置。动态配置消除了配置变更时重新部署应用和服务的需要，让配置管理变得更加高效和敏捷。配置中心化管理让实现无状态服务变得更简单，让服务按需弹性扩展变得更容易。

一句话概括就是Nacos = Spring Cloud注册中心 + Spring Cloud配置中心。

参考链接：

https://yq.aliyun.com/articles/698930

https://nacos.io