

注册中心原理和选型：Zookeeper、Eureka、Nacos、Consul 和 etcd

ImportNew 2022-03-28 11:30

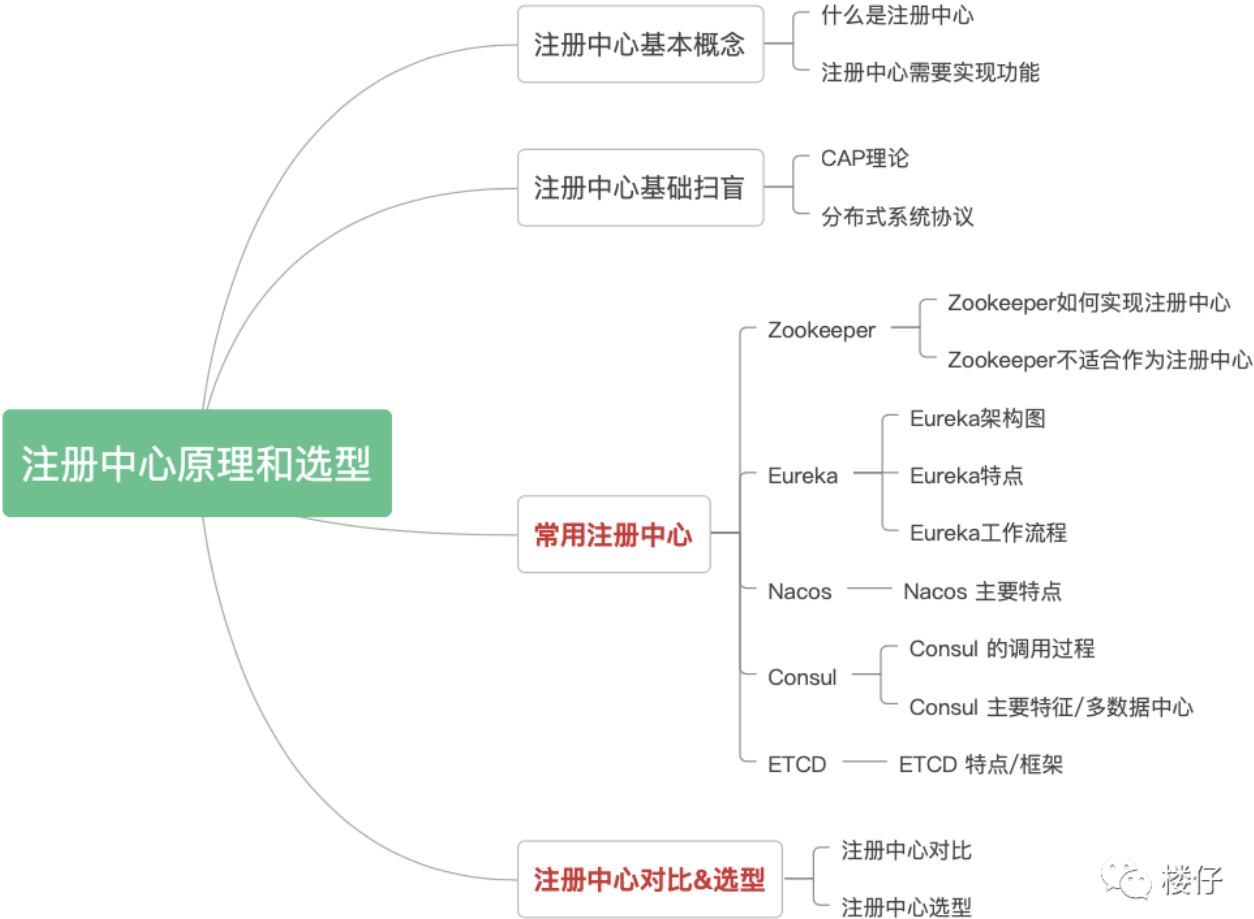
以下文章来源于楼仔，作者楼仔



楼仔

一枚小小的Go/Java代码搬运工，努力做一个懂技术、懂管理、懂业务、也懂生...

导读



楼仔

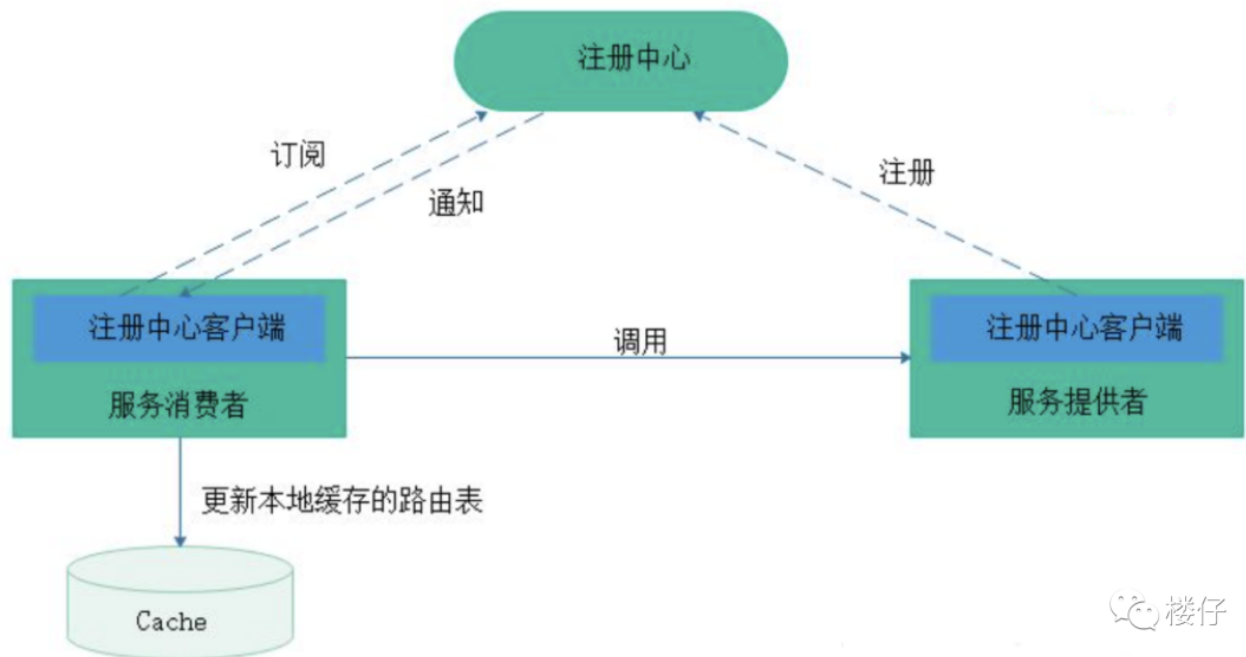
1. 注册中心基本概念

1.1 什么是注册中心？

注册中心主要有三种角色：

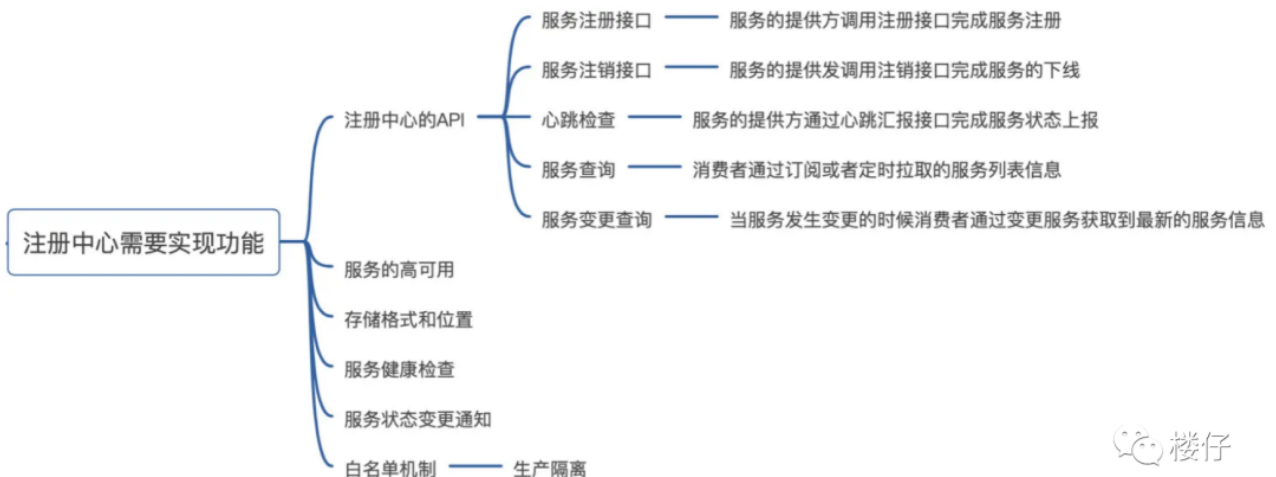
- **服务提供者 (RPC Server)**：在启动时，向 Registry 注册自身服务，并向 Registry 定期发送心跳汇报存活状态；
- **服务消费者 (RPC Client)**：在启动时，向 Registry 订阅服务，把 Registry 返回的服务节点列表缓存在本地内存中，并与 RPC Server 建立连接；
- **服务注册中心 (Registry)**：用于保存 RPC Server 的注册信息，当 RPC Server 节点发生变更时，Registry 会同步变更，RPC Client 感知后会刷新本地内存中缓存的服务节点列表。

最后，RPC Client 从本地缓存的服务节点列表中，基于负载均衡算法选择一台 RPC Server 发起调用。



1.2 注册中心需要实现功能

根据注册中心原理的描述，注册中心必须实现以下功能。偷个懒，直接贴幅图。



2. 注册中心基础扫盲

这块知识如果大家已经知道，可以直接跳过，主要是为了扫盲。

2.1 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，只能取其中两个。原因如下：

- 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，在此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。但是对于返回结果并不能保证。原因是，在分布式部署的时候，数据一致的过程不可能像切线路那么快；
- 再如果，同时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。

2.2 分布式系统协议

一致性协议算法主要有 Paxos、Raft、ZAB。

Paxos 算法是 Leslie Lamport 在 1990 年提出的一种基于消息传递的一致性算法，非常难以理解。基于 Paxos 协议的数据同步与传统主备方式最大的区别在于：Paxos 只需超过半数的副本在线且相互通信正常，就可以保证服务的持续可用，且数据不丢失。

Raft 是斯坦福大学的 Diego Ongaro、John Ousterhout 两个人以易理解为目标设计的一致性算法。已经有了十几种语言的 Raft 算法实现框架，较为出名的有 etcd。Google 的 Kubernetes 也是用了 etcd 作为它的服务发现框架。

Raft 是 Paxos 的简化版。与 Paxos 相比，Raft 强调的是易理解、易实现。Raft 和 Paxos 一样，只要保证超过半数的节点正常就能够提供服务。

ZooKeeper 原子消息广播协议（ZooKeeper Atomic Broadcast ZAB）是 ZooKeeper 实现分布式数据一致性的核心算法。ZAB 借鉴 Paxos 算法，但又不像 Paxos 算法那样是一种通用的分布式一致性算法。它是一种特别为 ZooKeeper 专门设计的支持崩溃恢复的原子广播协议。

3. 常用注册中心

这里主要介绍 5 种常用的注册中心，分别为 ZooKeeper、Eureka、Nacos、Consul 和 etcd。

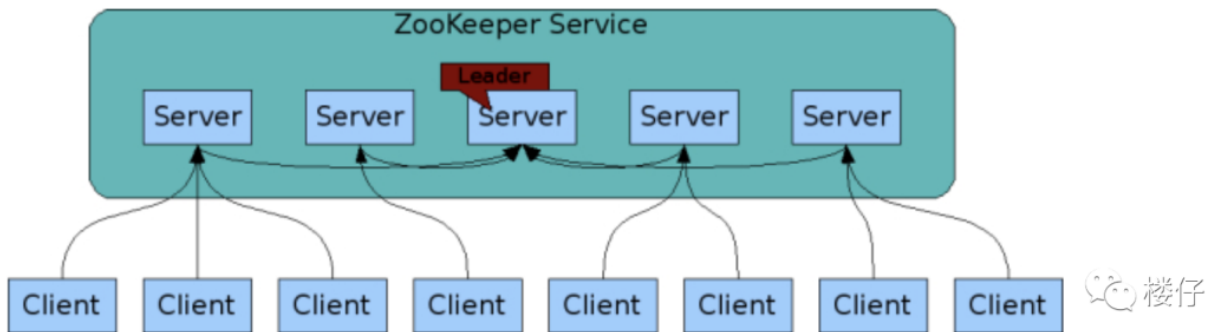
3.1 ZooKeeper

这个说起来有点意思的是，官方并没有说它是一个注册中心。但是国内 Dubbo 场景下很多都是使用 Zookeeper 来完成了注册中心的功能。

当然这有很多历史原因，这里我们就不追溯了。ZooKeeper 是非常经典的服务注册中心中间件。在国内环境下，由于受到 Dubbo 框架的影响，大部分情况下认为 Zookeeper 是 RPC 服务框架下注册中心最好选择。

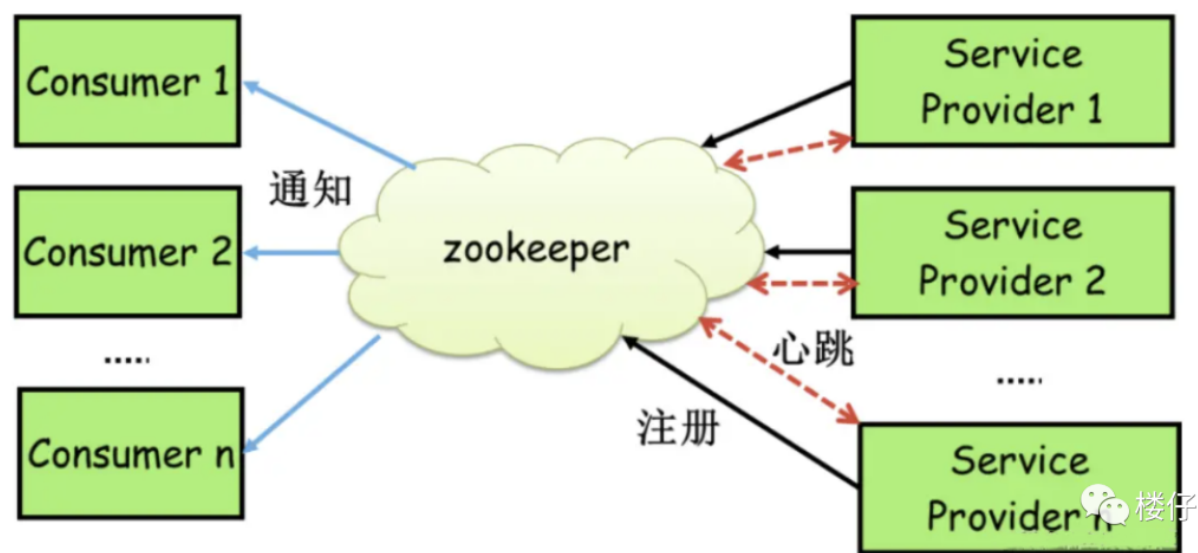
随着 Dubbo 框架的不断开发优化和各种注册中心组件的诞生，即使是 RPC 框架现在的注册中心也逐步放弃了 ZooKeeper。在常用的开发集群环境中，ZooKeeper 依然起到十分重要的作用。

Java 体系中，大部分的集群环境都是依赖 ZooKeeper 管理服务的各个节点。



ZooKeeper 如何实现注册中心

ZooKeeper 可以充当一个服务注册表（Service Registry），让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址（IP+端口）去访问具体的服务提供者。如下图所示：



每当一个服务提供者部署后都要将自己的服务注册到 ZooKeeper 的某一路径上：`/ {service}/ {version}/ {ip:port}`。

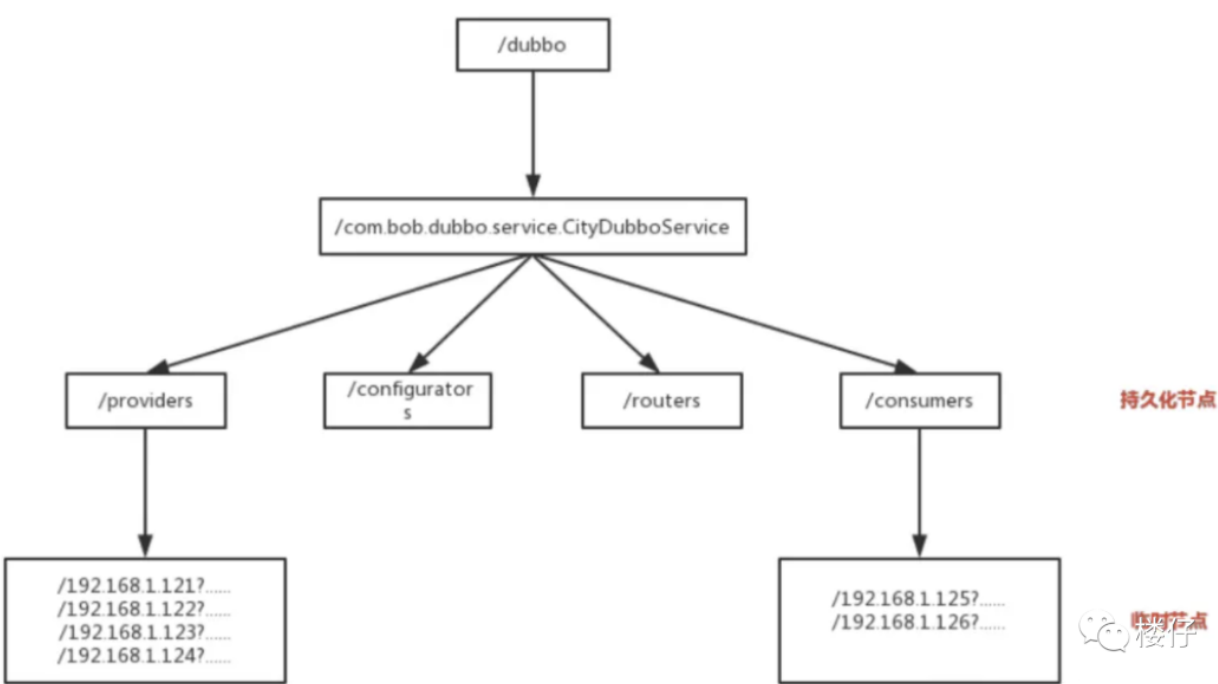
比如，我们的 HelloWorldService 部署到两台机器。那么 ZooKeeper 上就会创建两条目录：

- 1

/HelloWorldService/1.0.0/100.19.20.01:16888
- 2

/HelloWorldService/1.0.0/100.19.20.02:16888

这么描述有点不好理解，下图更直观：



在 ZooKeeper 中进行服务注册，实际上就是在 ZooKeeper 中创建了一个 znode 节点。该节点存储了该服务的 IP、端口、调用方式（协议、序列化方式）等。该节点承担着最重要的职责。它由服务提供者（发布服务时）创建，以供服务消费者获取节点中的信息，从而定位到服务提供者真正网络拓扑位置以及得知如何调用。

RPC 服务注册与发现过程简述如下：

1. 服务提供者启动时，会将其服务名称、IP 地址注册到配置中心；
2. 服务消费者在第一次调用服务时，会通过注册中心找到相应的服务的 IP 地址列表并缓存到本地，以供后续使用。当消费者调用服务时，不会再去请求注册中心，而是直接通过负载均衡算法从 IP 列表中获取一个服务提供者的服务器调用服务；
3. 当服务提供者的某台服务器宕机或下线时，相应的 IP 会从服务提供者 IP 列表中移除。同时，注册中心会将新的服务 IP 地址列表发送给服务消费者机器，缓存在消费者本机；
4. 当某个服务的所有服务器都下线了，那么这个服务也就下线了；
5. 同样，当服务提供者的某台服务器上线时，注册中心会将新的服务 IP 地址列表发送给服务消费者机器，缓存在消费者本机；
6. 服务提供方可以根据服务消费者的数量来作为服务下线的依据。

ZooKeeper 提供了“心跳检测”功能：它会定时向各个服务提供者发送一个请求（实际上建立的是一个 socket 长连接）。如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，并将其剔除。

比如 100.100.0.237 这台机器如果宕机了，那么 ZooKeeper 上的路径就会只剩 /HelloWorldService/1.0.0/100.100.0.238:16888。

ZooKeeper 的 Watch 机制其实就是一种推拉结合的模式：

- 服务消费者会去监听相应路径（/HelloWorldService/1.0.0），一旦路径上的数据有任务变化（增加或减少），ZooKeeper 只会发送一个事件类型和节点信息给关注的客户端，而不会包括具体的变更内容。所以事件本身是轻量级的，这就是推的部分；
- 收到变更通知的客户端需要自己去拉变更的数据，这就是拉的部分。

ZooKeeper 不适合作为注册中心

作为一个分布式协同服务 ZooKeeper 非常好，但是对于 Service 发现服务来说就不合适了。因为对于 Service 发现服务来说，就算是返回了包含不实的信息的结果也比什么都不返

回要好。所以当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接宕掉不可用。

但是 ZooKeeper 会出现这样一种情况，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，需要 30 ~ 120 秒，而且选举期间整个 ZooKeeper 集群都是不可用的。这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得 ZooKeeper 集群失去 master 节点是较大概率会发生的事。虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

所以说，作为注册中心，可用性的要求要高于一致性！

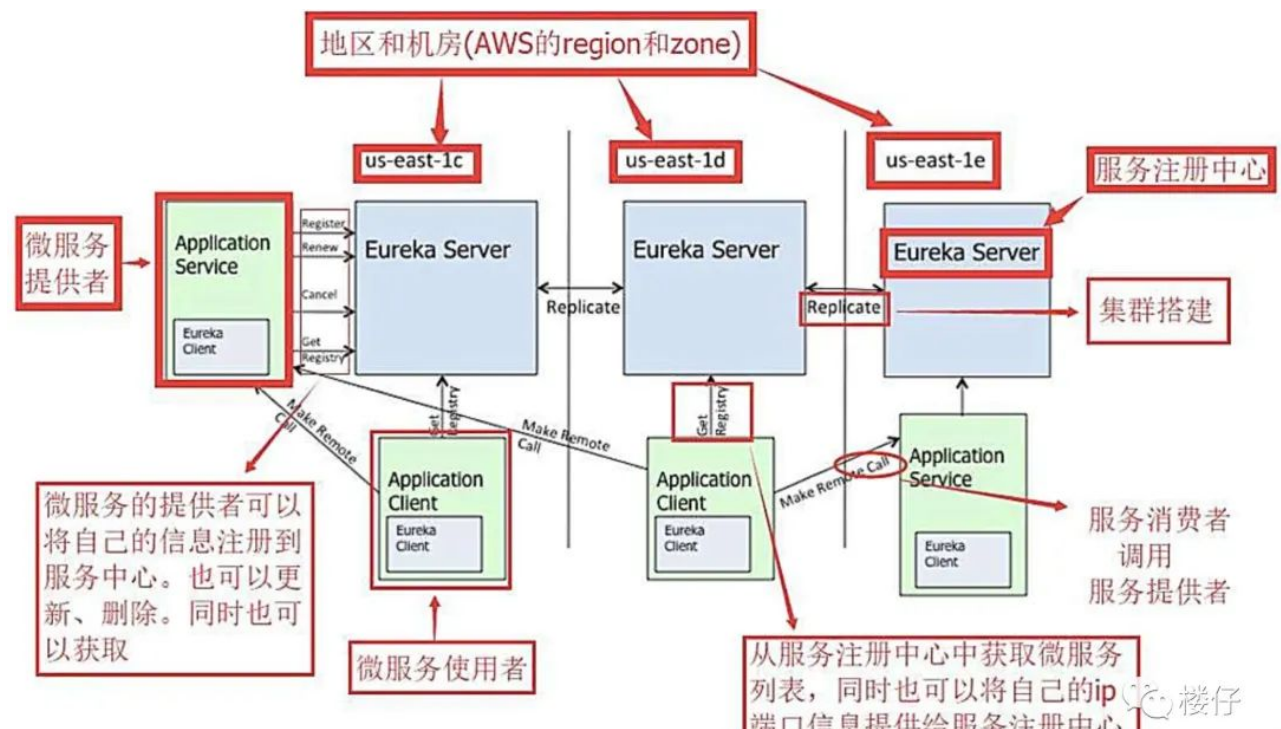
在 CAP 模型中，ZooKeeper 整体遵循一致性（CP）原则，即在任何时候对 Zookeeper 的访问请求能得到一致的数据结果。但是当机器下线或者宕机时，不能保证服务可用性。

那为什么 ZooKeeper 不使用最终一致性（AP）模型呢？

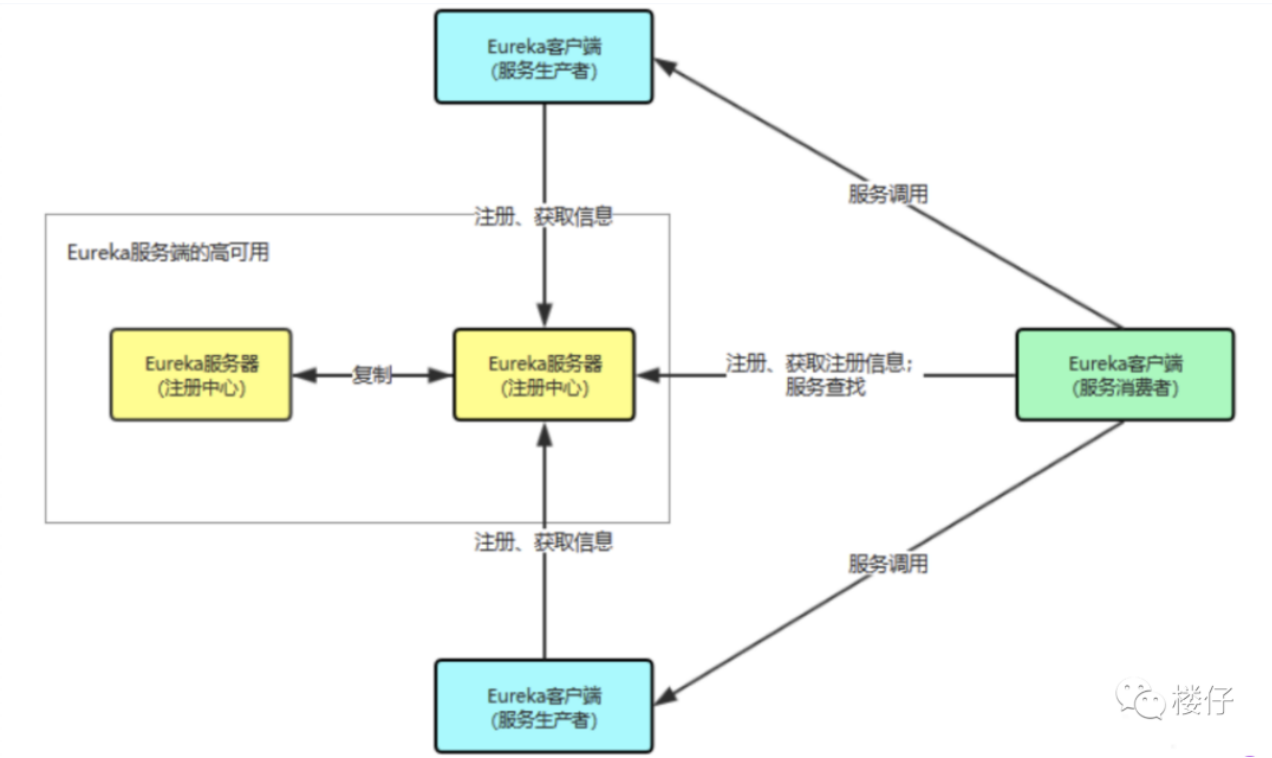
因为这个依赖 ZooKeeper 的核心算法是 ZAB，所有设计都是为了强一致性。这个对于分布式协调系统，完全没有毛病。但是你如果将 ZooKeeper 为分布式协调服务所做的一致性保障，用在注册中心或者说服务发现场景，这个其实就不合适。

3.2 Eureka

Eureka 架构图



什么？上面这幅图看起来很复杂？那我给你贴个简化版：



Eureka 特点

- **可用性 (AP 原则)**：Eureka 在设计时就遵循 AP 原则。Eureka 的集群中，只要有一台 Eureka 还在，就能保证注册服务可用。只不过查到的信息可能不是最新的（不保证强一致性）；
- **去中心化架构**：Eureka Server 可以运行多个实例来构建集群。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。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；
- **请求自动切换**：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；
- **节点间操作复制**：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操作，将请求复制到该 Eureka Server 当前所知的其它所有节点中；
- **自动注册与心跳**：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；
- **自动下线**：默认情况下，如果 Eureka Server 在一定时间内没有接收到某个服务实例的心跳（默认周期为 30 秒），Eureka Server 将会注销该实例（默认为 90

秒，`eureka.instance.lease-expiration-duration-in-seconds` 进行自定义配置)；

- **保护模式**：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。

除了上述特点，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：

- 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；
- 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）；
- 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。

Eureka 工作流程

了解完 Eureka 核心概念、自我保护机制以及集群内的工作原理后，我们来整体梳理一下 Eureka 的工作流程：

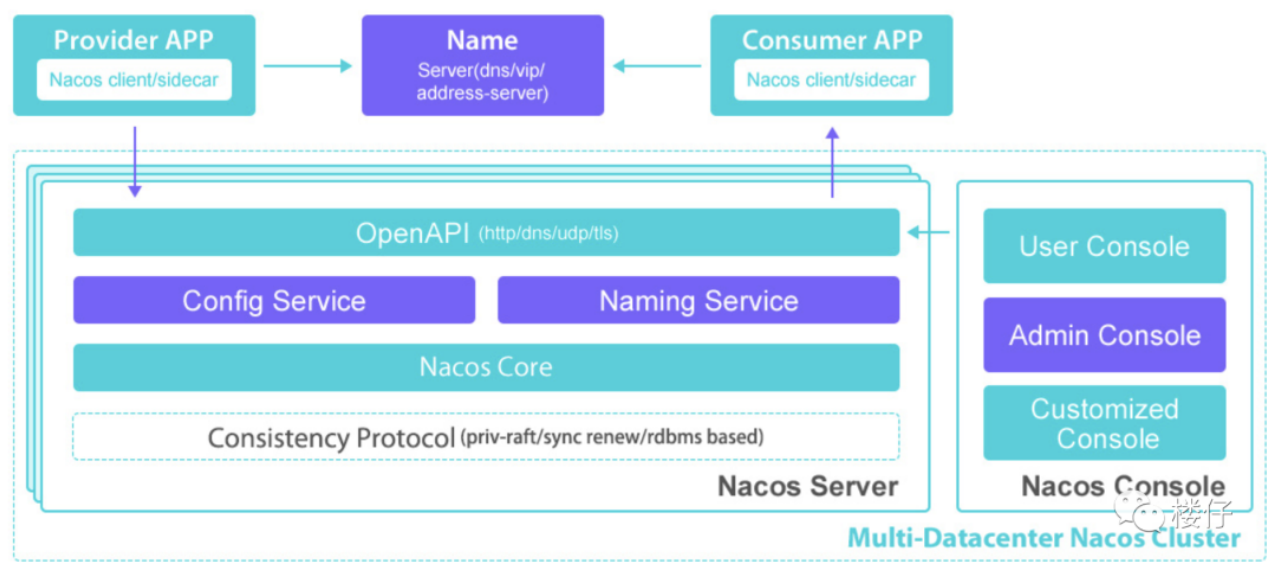
1. Eureka Server 启动成功，等待服务端注册。在启动过程中如果配置了集群，集群之间定时通过 Replicate 同步注册表，每个 Eureka Server 都存在独立完整的服务注册表信息；
2. Eureka Client 启动时根据配置的 Eureka Server 地址去注册中心注册服务；
3. Eureka Client 会每 30 秒向 Eureka Server 发送一次心跳请求，证明客户端服务正常；
4. 当 Eureka Server 90 秒内没有收到 Eureka Client 的心跳，注册中心则认为该节点失效，会注销该实例；
5. 单位时间内 Eureka Server 统计到有大量的 Eureka Client 没有上送心跳，则认为可能为网络异常，进入自我保护机制，不再剔除没有上送心跳的客户端；
6. 当 Eureka Client 心跳请求恢复正常之后，Eureka Server 自动退出自我保护模式；
7. Eureka Client 定时全量或者增量从注册中心获取服务注册表，并且将获取到的信息缓存到本地；
8. 服务调用时，Eureka Client 会先从本地缓存找寻调取的服务。如果获取不到，先从注册中心刷新注册表，再同步到本地缓存；
9. Eureka Client 获取到目标服务器信息，发起服务调用；
0. Eureka Client 程序关闭时向 Eureka Server 发送取消请求，Eureka Server 将实例从注册表中删除。

通过分析 Eureka 工作原理，我可以明显地感觉到 Eureka 的设计之巧妙，完美地解决了注册中心的稳定性和高可用性。

Eureka 为了保障注册中心的高可用性，容忍了数据的非强一致性，服务节点间的数据可能不一致，Client-Server 间的数据可能不一致。比较适合跨越多机房、对注册中心服务可用性要求较高的使用场景。

3.3 Nacos

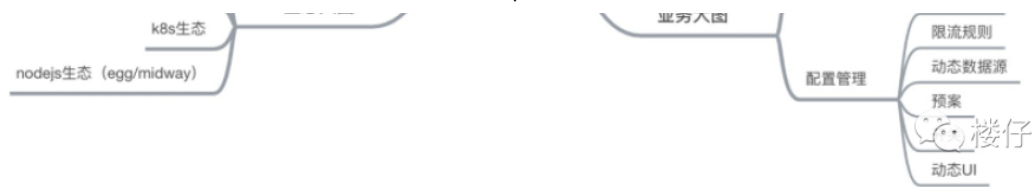
以下内容摘抄自 Nacos 官网：<https://nacos.io/zh-cn/docs/what-is-nacos.html>



Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。

Nacos 帮助您更敏捷和容易地构建、交付和管理微服务平台。Nacos 是构建以“服务”为中心的现代应用架构（例如微服务范式、云原生范式）的服务基础设施。





Nacos 主要特点

服务发现和服务健康监测：

- Nacos 支持基于 DNS 和基于 RPC 的服务发现。服务提供者使用原生 SDK、OpenAPI、或一个独立的 Agent 注册 Service 后，服务消费者可以使用 DNS TODO 或 HTTP&API 查找和发现服务；
- Nacos 提供对服务的实时的健康检查，阻止向不健康的主机或服务实例发送请求。Nacos 支持传输层（PING 或 TCP）和应用层（如 HTTP、MySQL、用户自定义）的健康检查。对于复杂的云环境和网络拓扑环境中（如 VPC、边缘网络等）服务的健康检查，Nacos 提供了 agent 上报模式和服务端主动检测两种健康检查模式。Nacos 还提供了统一的健康检查仪表盘，帮助您根据健康状态管理服务服务的可用性及流量。

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。
- Nacos 提供了一个简洁易用的 UI（控制台样例 Demo）帮助您管理所有的服务和应用的配置。Nacos 还提供包括配置版本跟踪、金丝雀发布、一键回滚配置以及客户端配置更新状态跟踪在内的一系列开箱即用的配置管理特性，帮助您更安全地在生产环境中管理配置变更和降低配置变更带来的风险。

动态 DNS 服务：

- 动态 DNS 服务支持权重路由，让您更容易地实现中间层负载均衡、更灵活的路由策略、流量控制以及数据中心内网的简单 DNS 解析服务。动态 DNS 服务还能让您更容易地实现以 DNS 协议为基础的服务发现，以帮助您消除耦合到厂商私有服务发现 API 上的风险。
- Nacos 提供了一些简单的 DNS API TODO 帮助您管理服务的关联域名和可用的 IP:PORT 列表。

小结一下：

- Nacos 是阿里开源的，支持基于 DNS 和基于 RPC 的服务发现；
- Nacos 的注册中心支持 CP 也支持 AP，对它来说只是一个命令的切换，随你玩。还支持各种注册中心迁移到 Nacos，反正一句话，只要你想要的它就有；
- Nacos 除了服务的注册发现之外，还支持动态配置服务。一句话概括就是 Nacos = Spring Cloud 注册中心 + Spring Cloud 配置中心。

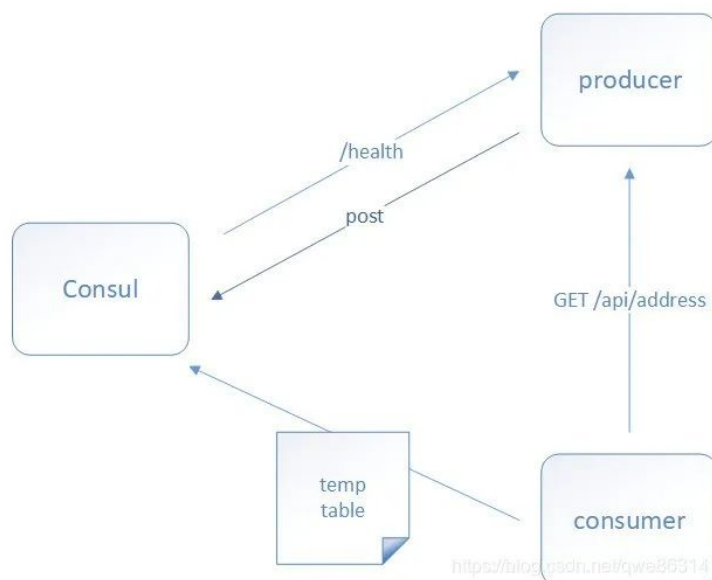
3.4 Consul

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。与其它分布式服务注册与发现的方案，Consul 的方案更“一站式”，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，不再需要依赖其它工具（比如 ZooKeeper 等）。

Consul 使用起来也较为简单，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）。安装包仅包含一个可执行文件，方便部署。与 Docker 等轻量级容器可无缝配合。

Consul 的调用过程

1. 当 Producer 启动的时候，会向 Consul 发送一个 POST 请求，告诉 Consul 自己的 IP 和 Port；
2. Consul 接收到 Producer 的注册后，每隔 10 秒（默认）会向 Producer 发送一个健康检查的请求，检验 Producer 是否健康；
3. 当 Consumer 发送 GET 方式请求 /api/address 到 Producer 时，会先从 Consul 中拿到一个存储服务 IP 和 Port 的临时表，从表中拿到 Producer 的 IP 和 Port 后再发送 GET 方式请求 /api/address；
4. 该临时表每隔 10s 会更新，只包含有通过了健康检查的 Producer。

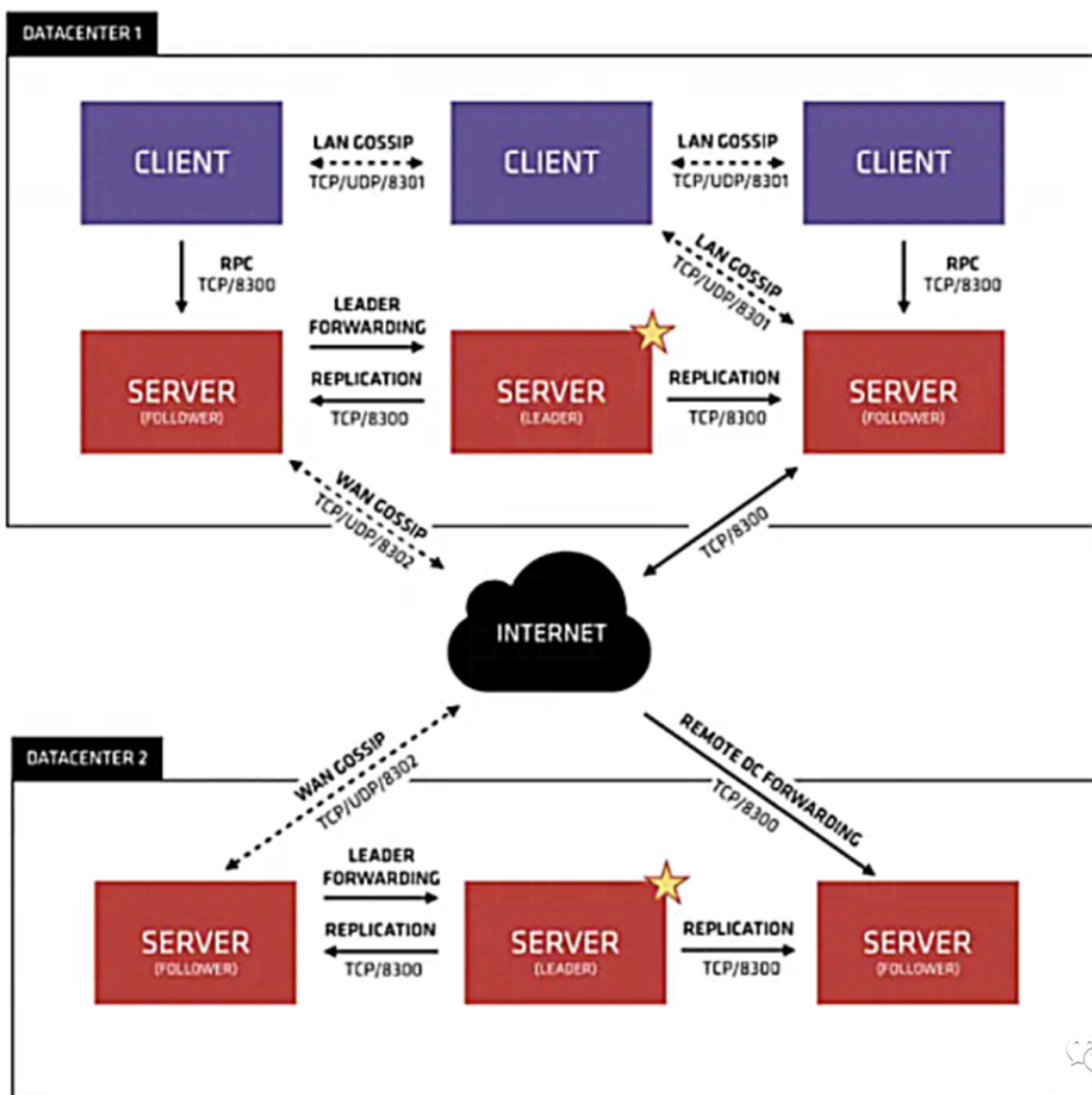


Consul 主要特征

- CP模型，使用 Raft 算法来保证强一致性，不保证可用性；
- 支持服务注册与发现、健康检查、KV Store 功能；
- 支持多数据中心，可以避免单数据中心的单点故障，而其部署则需要考虑网络延迟、分片等情况等；
- 支持安全服务通信，Consul 可以为服务生成和分发 TLS 证书，以建立相互的 TLS 连接；
- 支持 HTTP 和 DNS 协议接口；
- 官方提供 Web 管理界面。

多数据中心

Consul 支持开箱即用的多数据中心，这意味着用户不需要担心需要建立额外的抽象层让业务扩展到多个区域。



在上图中有两个 DataCenter，他们通过 Internet 互联。同时请注意，为了提高通信效率，只有 Server 节点才加入跨数据中心的通信。

在单个数据中心中，Consul 分为 Client 和 Server 两种节点（所有的节点也被称为 Agent）。Server 节点保存数据，Client 负责健康检查及转发数据请求到 Server；Server 节点有一个 Leader 和多个 Follower，Leader 节点会将数据同步到 Follower，Server 的数量推荐是 3 个或者 5 个，在 Leader 挂掉的时候会启动选举机制产生一个新的 Leader。

集群内的 Consul 节点通过流言协议（Gossip）维护成员关系。也就是说某个节点了解集群内现在还有哪些节点，这些节点是 Client 还是 Server。单个数据中心的流言协议同时使用 TCP 和 UDP 通信，并且都使用 8301 端口。跨数据中心的流言协议也同时使用 TCP 和 UDP 通信，端口使用 8302。

集群内数据的读写请求既可以直接发到 Server，也可以通过 Client 使用 RPC 转发到 Server，请求最终会到达 Leader 节点。在允许数据延时的情况下，读请求也可以在普通的 Server 节点完成。集群内数据的读写和复制都是通过 TCP 的 8300 端口完成。

3.5 etcd

etcd 是一个 Go 语言编写的分布式、高可用的一致性键值存储系统。用于提供可靠的分布式键值存储、配置共享和服务发现等功能。

etcd 特点

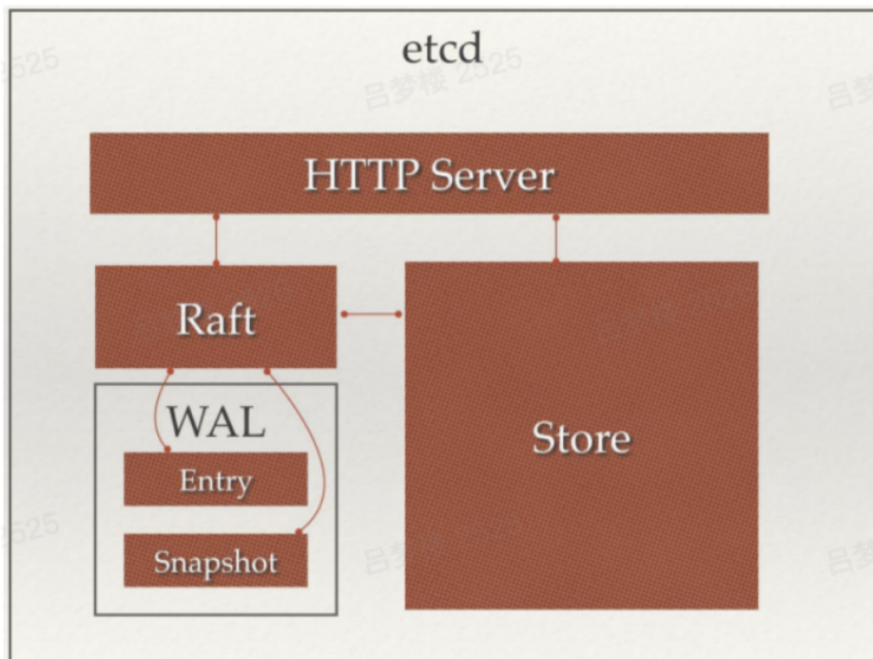
- 易使用：基于 HTTP+JSON 的 API 让你用 curl 就可以轻松使用；
- 易部署：使用 Go 语言编写，跨平台，部署和维护简单；
- 强一致：使用 Raft 算法充分保证了分布式系统数据的强一致性；
- 高可用：具有容错能力。假设集群有 n 个节点，当有 $(n-1)/2$ 节点发送故障，依然能提供服务；
- 持久化：数据更新后，会通过 WAL 格式数据持久化到磁盘，支持 Snapshot 快照；
- 快速：每个实例每秒支持一千次写操作，极限写性能可达 10K QPS；
- 安全：可选 SSL 客户认证机制；
- etcd 3.0：除了上述功能，还支持 gRPC 通信、watch 机制。

ETCD 框架

etcd 主要分为四个部分：

- HTTP Server：用于处理用户发送的 API 请求以及其它 etcd 节点的同步与心跳信息请求；

- Store：用于处理 etcd 支持的各类功能的事务，包括数据索引、节点状态变更、监控与反馈、事件处理与执行等等，是 etcd 对用户提供的绝大多数 API 功能的具体实现；
- Raft：Raft 强一致性算法的具体实现是 etcd 的核心；
- WAL：Write Ahead Log（预写式日志）是 etcd 的数据存储方式。除了在内存中存有所有数据的状态以及节点的索引以外，etcd 就通过 WAL 进行持久化存储。WAL 中，所有的数据提交前都会事先记录日志。Snapshot 是为了防止数据过多而进行的状态快照；Entry 表示存储的具体日志内容。


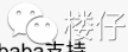


楼仔

通常，一个用户的请求发送过来，会经由 HTTP Server 转发给 Store 进行具体的事务处理，如果涉及到节点的修改，则交给 Raft 模块进行状态的变更、日志的记录，然后再同步给别的 etcd 节点以确认数据提交，最后进行数据的提交再次同步。

4. 注册中心对比与选型

4.1 注册中心对比

Feature	Consul	Zookeeper	Etcd	Eureka	Nacos
服务健康检查	服务状态, 内存, 硬盘等	(弱)长连接, keepalive	连接心跳	可配支持	传输层 (PING 或 TCP)和应用层 (如 HTTP、MySQL、用户自定义) 的健康检查
多数据中心	支持	—	—	—	支持
kv存储服务	支持	支持	支持	—	支持
一致性	Raft	Paxos	Raft	—	Raft
CAP定理	CP	CP	CP	AP	CP: 配置中心 AP: 注册中心
使用接口 (多语言能力)	支持http和dns	客户端	http/grpc	http (sidecar)	Nacos 支持基于 DNS 和基于 RPC 的服务发现。服务提供者使用 原生 SDK、OpenAPI、或一个独立的 Agent
watch支持	全量/支持long polling	支持	支持 long polling	支持 long polling/大部分增量	支持 long polling/大部分增量
自身监控	metrics	—	metrics	metrics	
安全	acl /https	acl	https支持 (弱)	—	acl 
Spring Cloud 集成	已支持	已支持	已支持	已支持	已支持
备注	可以作为eureka的替代使用			2.0不在更新	1. 支持dubbo  2. spring-cloud-alibaba支持

- **服务健康检查**: Eureka 使用时需要显式配置健康检查支持。Zookeeper、Etcd 则在失去了和服务进程的连接情况下任务不健康。而 Consul 相对更为详细点, 比如内存是否已使用了 90%, 文件系统的空间是不是快不足了;
- **多数据中心**: Consul 和 Nacos 都支持, 其他的产品则需要额外的开发工作来实现。
- **KV 存储服务**: 除了 Eureka, 其他几款都能够对外支持 k-v 的存储服务, 所以后面会讲到这几款产品追求高一致性的重要原因。而提供存储服务, 也能够较好地转化为动态配置服务哦;
- **CAP 理论的取舍**:
 - Eureka 是典型的 AP。Nacos 可以配置为 AP, 作为分布式场景下的服务发现的产品较为合适, 服务发现场景的可用性优先级较高, 一致性并不是特别致命;
 - 而 ZooKeeper、etcd、Consul 则是 CP 类型, 牺牲了可用性, 在服务发现场景并没太大优势;
- **Watch 支持**: ZooKeeper 支持服务器端推送变化, 其它都通过长轮询的方式来实现变化的感知;
- **自身集群的监控**: 除了 ZooKeeper 和 Nacos, 其它几款都默认支持 Metrics, 运维者可以搜集并报警这些度量信息达到监控目的。
- **Spring Cloud 集成**: 目前都有相对应的 Boot starter, 提供了集成能力。

4.2 注册中心选型

关于注册中心的对比和选型，其实上面已经讲的非常清楚了，我给出一些个人理解：

- **关于 CP 还是 AP 的选择：**选择 AP。因为可用性高于一致性，所以更倾向 Eureka 和 Nacos。关于 Eureka、Nacos 如何选择，哪个让我做的事少，我就选择哪个。显然 Nacos 帮我们做了更多的事；
- **技术体系：**etcd 和 Consul 都是 Go 开发的，Eureka、Nacos 和 Zookeeper 都是 Java 开发的，可能项目属于不同的技术栈，会偏向选择对应的技术体系；
- **高可用：**这几款开源产品都已经考虑如何搭建高可用集群，有些差别而已；
- **产品的活跃度：**这几款开源产品整体上都比较活跃。

- EOF -

推荐阅读 — 点击标题可跳转

- 1、[细说历经磨难的注册中心选型](#)
- 2、[使用 Eureka 实现服务注册与发现](#)
- 3、[一个线上问题的思考：Eureka注册中心集群如何实现客户端请求负载及故障转移？](#)

看完本文有收获？请转发分享给更多人

关注「ImportNew」，提升Java技能

ImportNew

分享 Java 相关技术干货 · 资讯 · 高薪职位 · 教程



微信号：ImportNew



长按识别二维码关注

伯乐在线 旗下微信公众号

商务合作QQ：2302462408

点赞和在看就是最大的支持 ❤️

喜欢此内容的人还喜欢

听说你想画个机制图？

YuLabSMU

十大 π 公式

原理

【长图】致敬！烟草“女神”

东方烟草报