架构选型

楼仔 著

目录

😊 前言

大家好，我是楼仔！

为了方便大家学习，我会把所有的系列文章整理成手册，今天给大家整理的是「架构选型手册」。

这个系列有多肝？

注册中心，转载了 15 次，转载阅读量 3W +，消息队列转载 13 次，转载阅读量 2.5W +，微服务网关转载 8 次，转载阅读量 2W +。

所以这个系列，是我转载最多的系列，通过这个系列的学习，让你对常用的开源软件有个基础、全局的认识，无论是面试，还是技术选型，都非常有帮助。

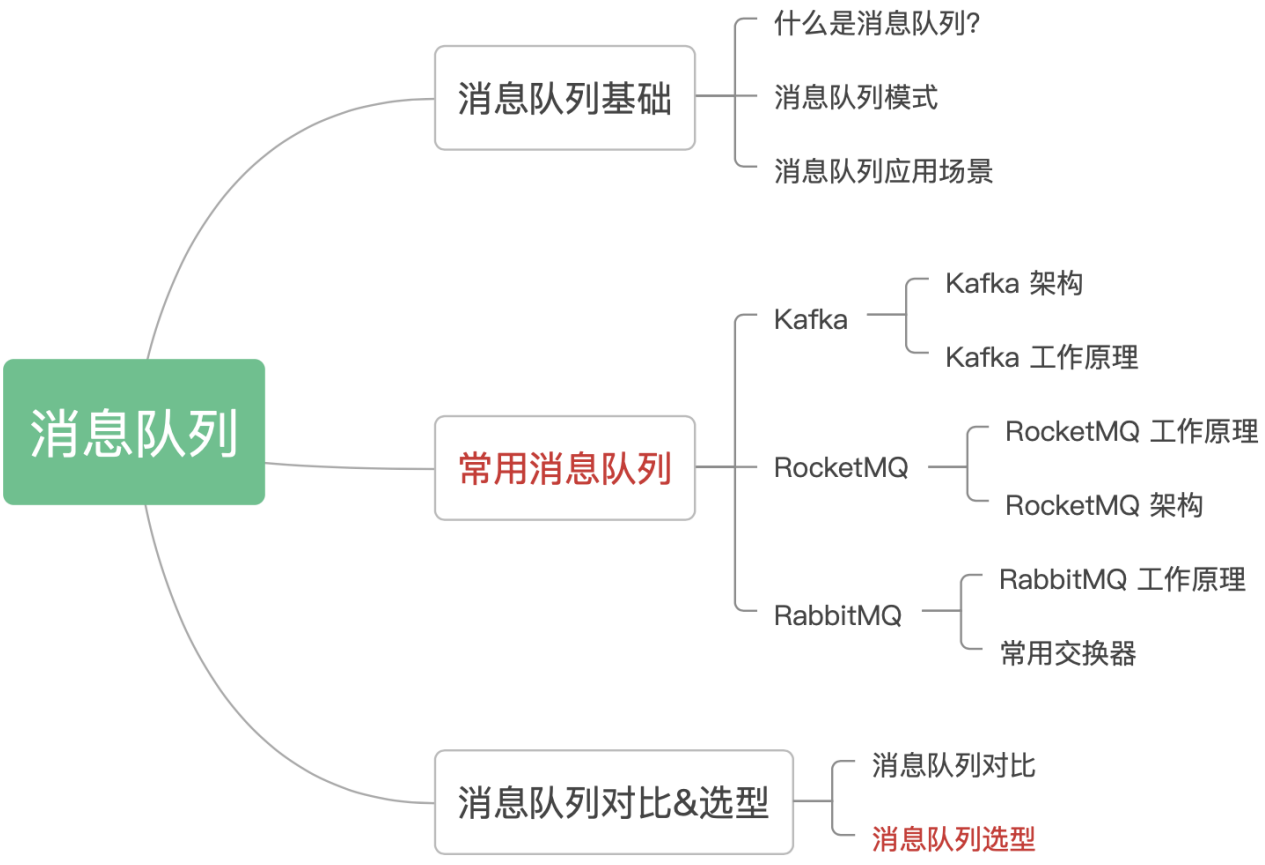
这个系列有 5 篇是原创，1 篇转载，《监控系统》是引用大佬骆俊武的文章，特此说明！

📚 第 1 章：消息队列

消息队列中间件重要吗？面试必问问题之一，你说重不重要。我有时会问同事，为啥你用 RabbitMQ，不用 Kafka，或者 RocketMQ 呢，他给我的回答 “因为公司用的就是这个，大家都这么用”，如果你去面试，直接就被 Pass，今天这篇文章，告诉你如何回答。

这篇文章，我重点突出消息队列选型，弱化每种队列内部的实现细节，精华提炼，可读性更强！

常用的消息队列主要这 4 种，分别为 Kafka、RabbitMQ、RocketMQ 和 ActiveMQ，主要介绍前三，不BB，上思维导图！

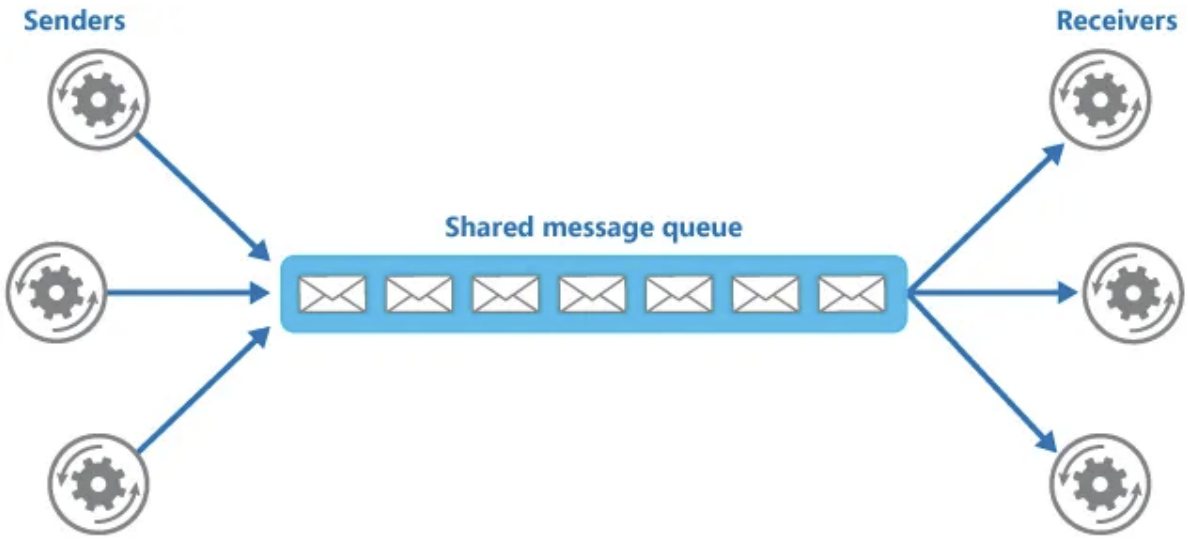


消息队列基础

什么是消息队列?

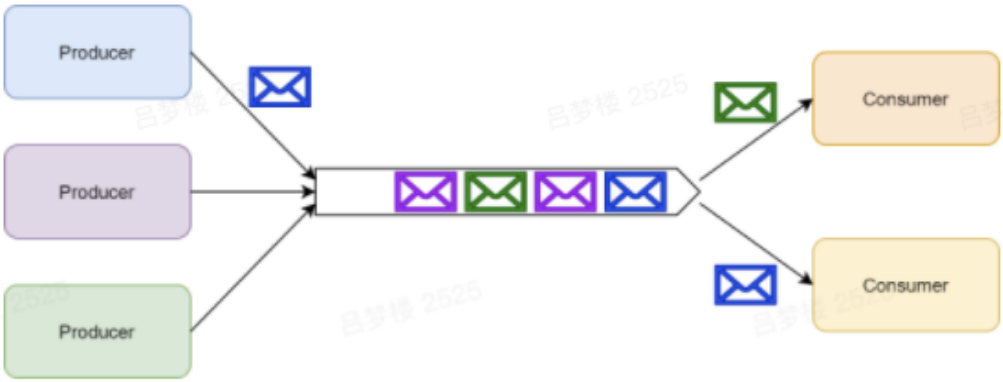
消息队列是在消息的传输过程中保存消息的容器，用于接收消息并以文件的方式存储，一个消息队列可以被一个也可以被多个消费者消费，包含以下 3 元素：

- Producer：消息生产者，负责产生和发送消息到 Broker；
- Broker：消息处理中心，负责消息存储、确认、重试等，一般其中会包含多个 Queue；
- Consumer：消息消费者，负责从 Broker 中获取消息，并进行相应处理。

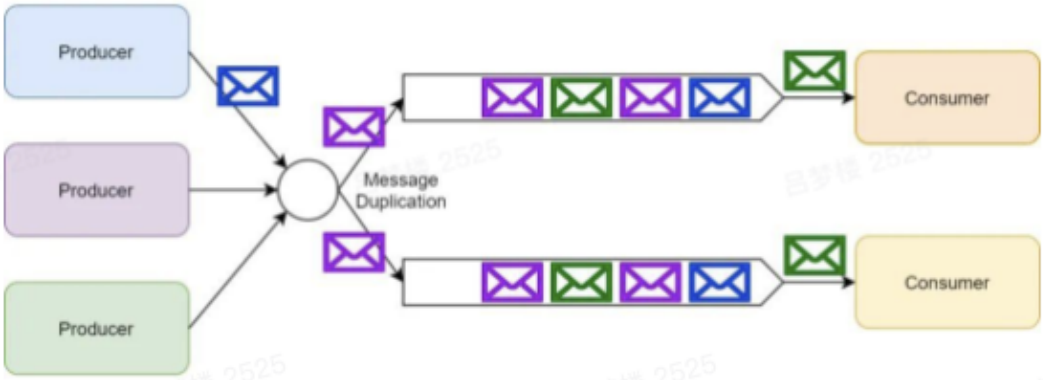


消息队列模式

- 点对点模式：多个生产者可以向同一个消息队列发送消息，一个具体的消息只能由一个消费者消费。



- 发布/订阅模式：单个消息可以被多个订阅者并发的获取和处理。



消息队列应用场景

- **应用解耦**：消息队列减少了服务之间的耦合性，不同的服务可以通过消息队列进行通信，而不用关心彼此的实现细节。
- **异步处理**：消息队列本身是异步的，它允许接收者在消息发送很长时间后再取回消息。
- **流量削锋**：当上下游系统处理能力存在差距的时候，利用消息队列做一个通用的”载体”，在下游有能力处理的时候，再进行分发与处理。
- **日志处理**：日志处理是指将消息队列用在日志处理中，比如 Kafka 的应用，解决大量日志传输的问题。
- **消息通讯**：消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯，比如实现点对点消息队列，或者聊天室等。
- **消息广播**：如果没有消息队列，每当一个新的业务方接入，我们都要接入一次新接口。有了消息队列，我们只需要关心消息是否送达了队列，至于谁希望订阅，是下游的事情，无疑极大地减少了开发和联调的工作量。

常用消息队列

由于官方社区现在对 ActiveMQ 5.x 维护越来越少，较少在大规模吞吐的场景中使用，所以我们主要讲解 Kafka、RabbitMQ 和 RocketMQ。

Kafka

Apache Kafka 最初由 LinkedIn 公司基于独特的设计实现为一个分布式的提交日志系统，之后成为 Apache 项目的一部分，**号称大数据的杀手锏，在数据采集、传输、存储的过程中发挥着举足轻重的作用。**

它是一个分布式的，支持多分区、多副本，基于 Zookeeper 的分布式消息流平台，它同时也是一款开源的基于发布订阅模式的消息引擎系统。

重要概念

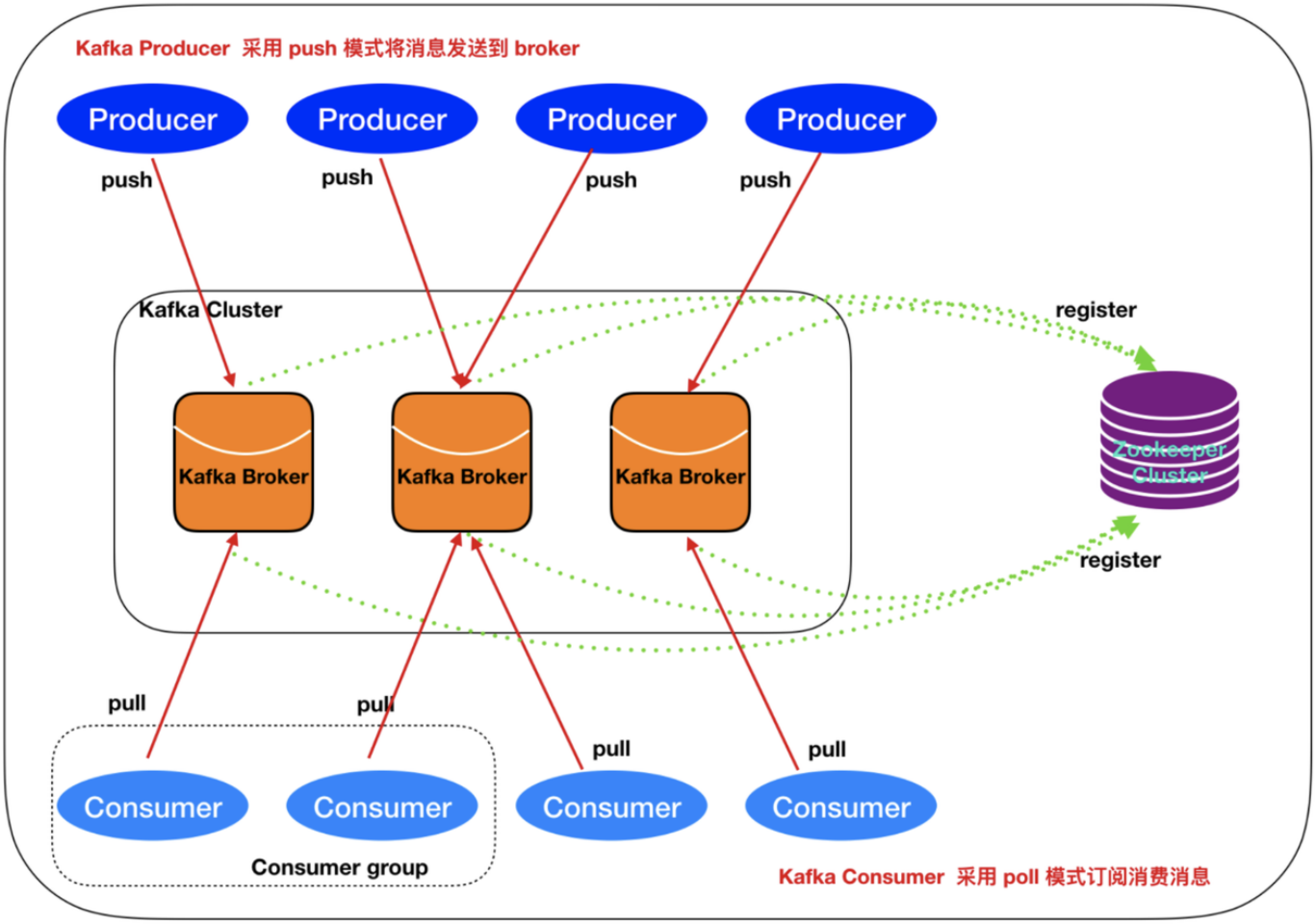
- **主题（Topic）**：消息的种类称为主题，可以说一个主题代表了一类消息，相当于是对消息进行分类，主题就像是数据库中的表。
- **分区（partition）**：主题可以被分为若干个分区，同一个主题中的分区可以不在一个机器上，有可能会部署在多个机器上，由此来实现 kafka 的伸缩性。
- **批次**：为了提高效率， 消息会分批次写入 Kafka，批次就代指的是一组消息。
- **消费者群组（Consumer Group）**：消费者群组指的就是由一个或多个消费者组成的群体。
- **Broker**: 一个独立的 Kafka 服务器就被称为 broker，broker 接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。
- **Broker 集群**：broker 集群由一个或多个 broker 组成。

- **重平衡（Rebalance）**：消费者组内某个消费者实例挂掉后，其他消费者实例自动重新分配订阅主题分区的过程。

Kafka 架构

一个典型的 Kafka 集群中包含 Producer、broker、Consumer Group、Zookeeper 集群。

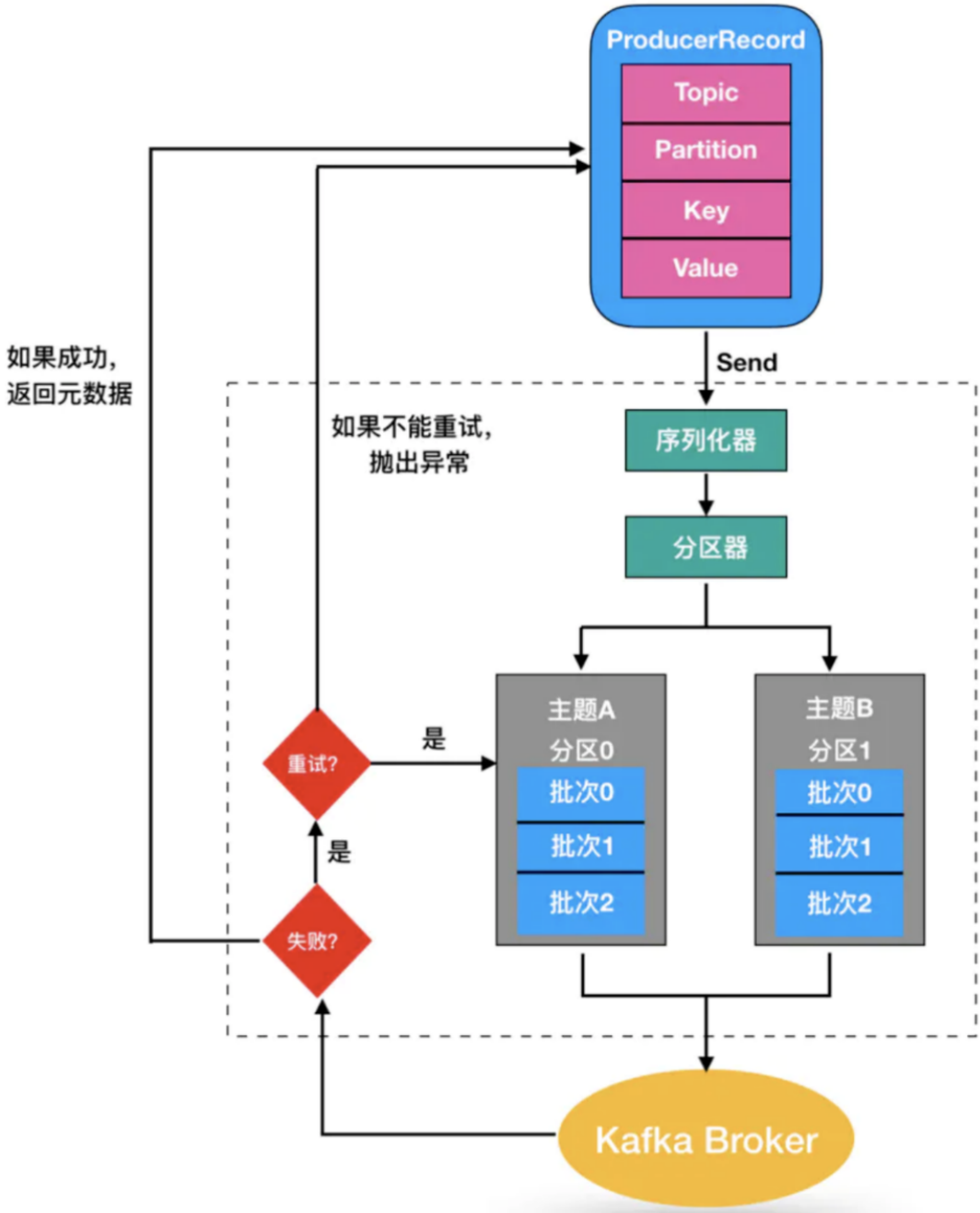
Kafka 通过 Zookeeper 管理集群配置，选举 leader，以及在 Consumer Group 发生变化时进行 rebalance。Producer 使用 push 模式将消息发布到 broker，Consumer 使用 pull 模式从 broker 订阅并消费消息。



Kafka 工作原理

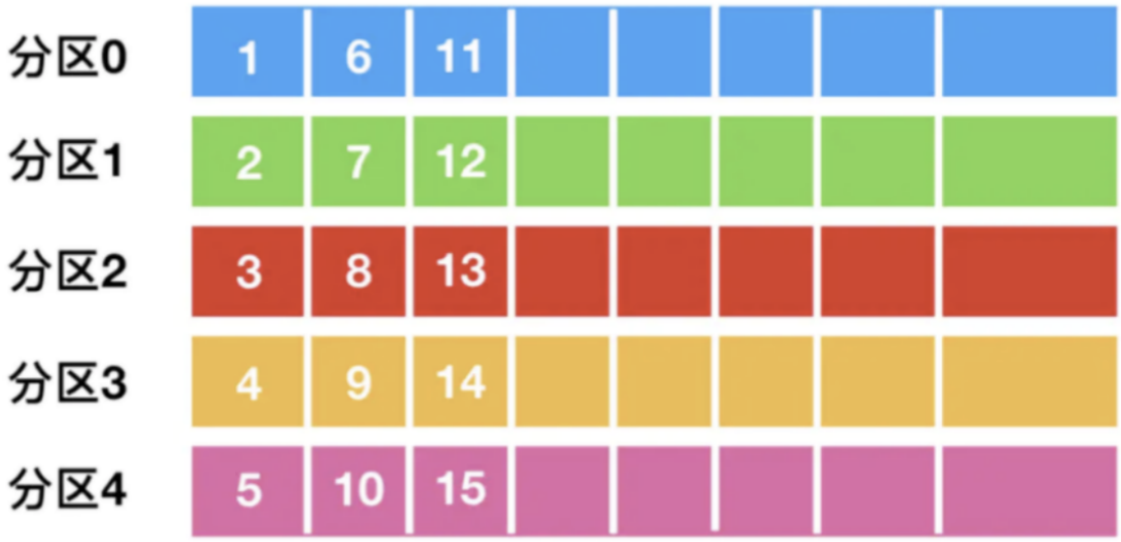
消息经过序列化后，通过不同的分区策略，找到对应的分区。

相同主题和分区的消息，会被存放在同一个批次里，然后由一个独立的线程负责把它们发到 Kafka Broker 上。



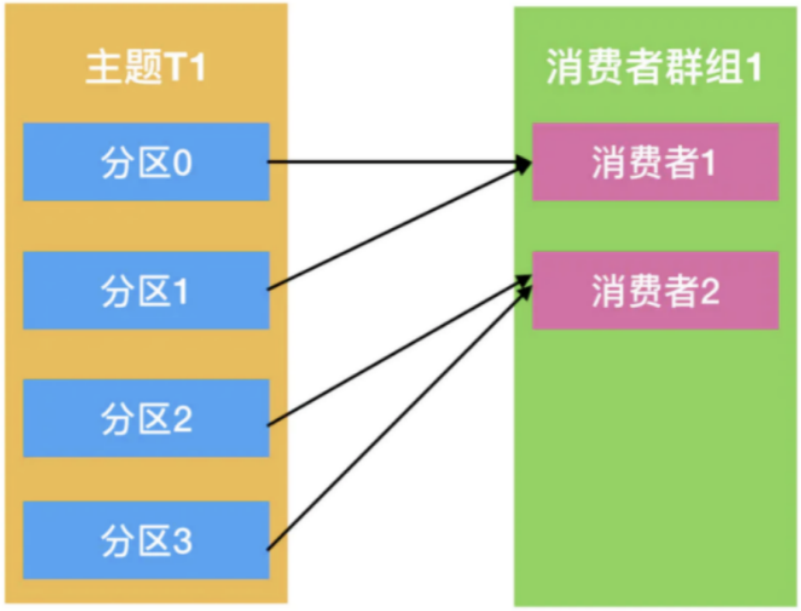
分区的策略包括顺序轮询、随机轮询和 key hash 这 3 种方式，那什么是分区呢?

分区是 Kafka 读写数据的最小粒度，比如主题 A 有 15 条消息，有 5 个分区，如果采用顺序轮询的方式，15 条消息会顺序分配给这 5 个分区，后续消费的时候，也是按照分区粒度消费。



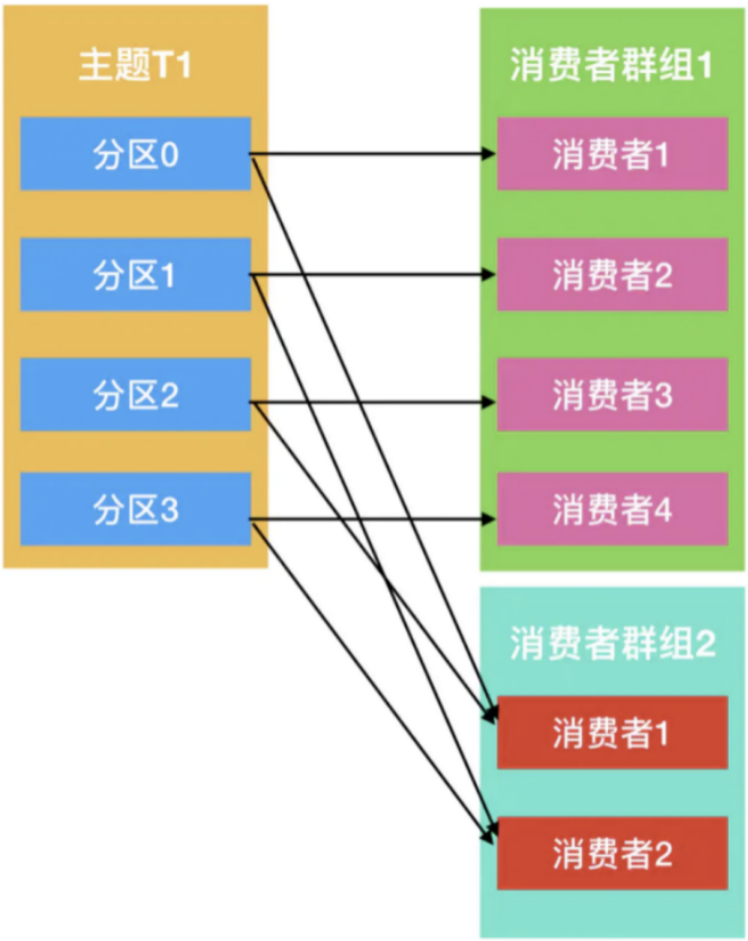
由于分区可以部署在多个不同的机器上，所以可以通过分区实现 Kafka 的伸缩性，比如主题 A 的 5 个分区，分别部署在 5 台机器上，如果下线一台，分区就变为 4。

Kafka 消费是通过消费群组完成，同一个消费者群组，**一个消费者可以消费多个分区，但是一个分区，只能被一个消费者消费。**



如果消费者增加，会触发 Rebalance，也就是分区和消费者需要重新配对。

不同的消费群组互不干涉，比如下图的 2 个消费群组，可以分别消费这 4 个分区的消息，互不影响。



> 更多知识，详见 《原理初探之 Kafka》

RocketMQ

RocketMQ 是阿里开源的消息中间件，它是纯 Java 开发，**具有高性能、高可靠、高实时、适合大规模分布式系统应用的特点。**

RocketMQ 思路起源于 Kafka，但并不是 Kafka 的一个 Copy，**它对消息的可靠传输及事务性做了优化**，目前在阿里集团被广泛应用于交易、充值、流计算、消息推送、日志流式处理、binglog 分发等场景。

重要概念

- **Name 服务器（NameServer）**：充当注册中心，类似 Kafka 中的 Zookeeper。
- **Broker**: 一个独立的 RocketMQ 服务器就被称为 broker，broker 接收来自生产者的消息，为消息设置偏移量。
- **主题（Topic）**：消息的第一级类型，一条消息必须有一个 Topic。
- **子主题（Tag）**：消息的第二级类型，同一业务模块不同目的的消息就可以用相同 Topic 和不同的 Tag 来标识。
- **分组（Group）**：一个组可以订阅多个 Topic，包括生产者组（Producer Group）和消费者组（Consumer Group）。
- **队列（Queue）**：可以类比 Kafka 的分区 Partition。

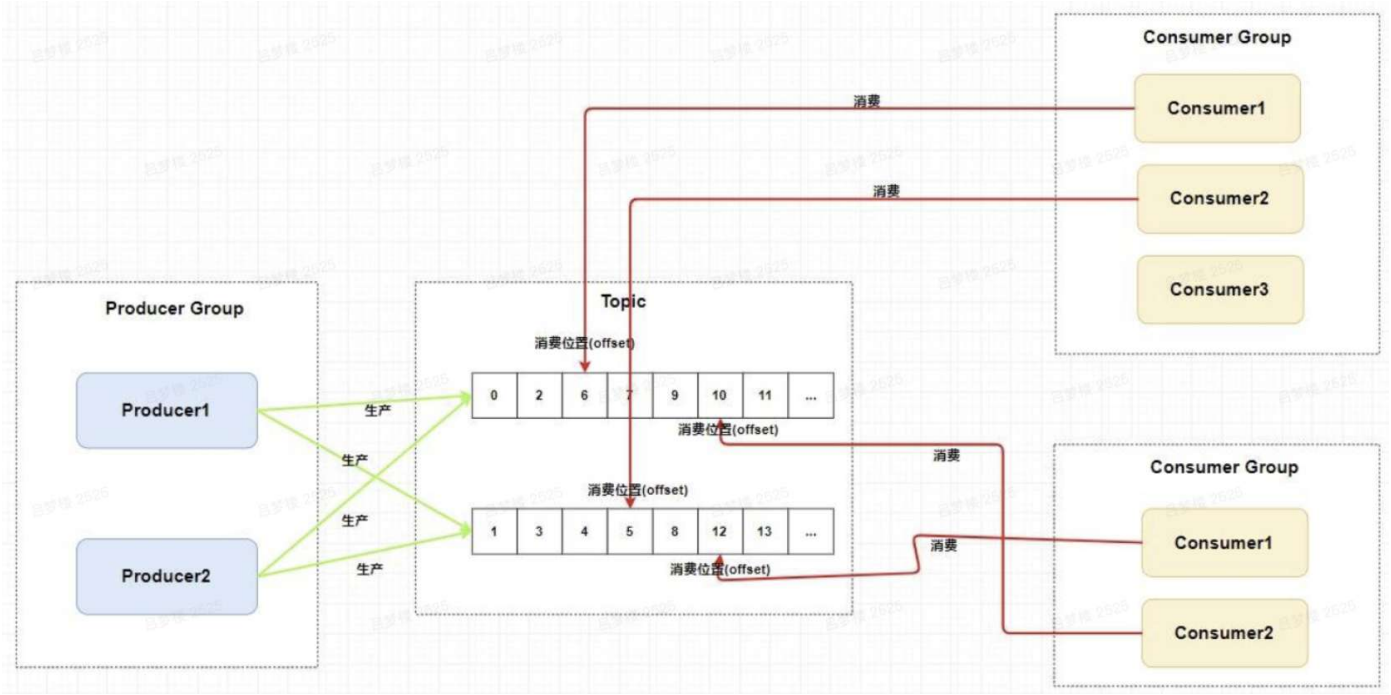
RocketMQ 工作原理

RockerMQ 中的消息模型就是按照主题模型所实现的，包括 Producer Group、Topic、Consumer Group 三个角色。

为了提高并发能力，一个 Topic 包含多个 Queue，生产者组根据主题将消息放入对应的 Topic，下图是采用轮询的方式找到里面的 Queue。

RockerMQ 中的消费群组和 Queue，可以类比 Kafka 中的消费群组和 Partition：**不同的消费者组互不干扰，一个 Queue 只能被一个消费者消费，一个消费者可以消费多个 Queue。**

消费 Queue 的过程中，通过偏移量记录消费的位置。

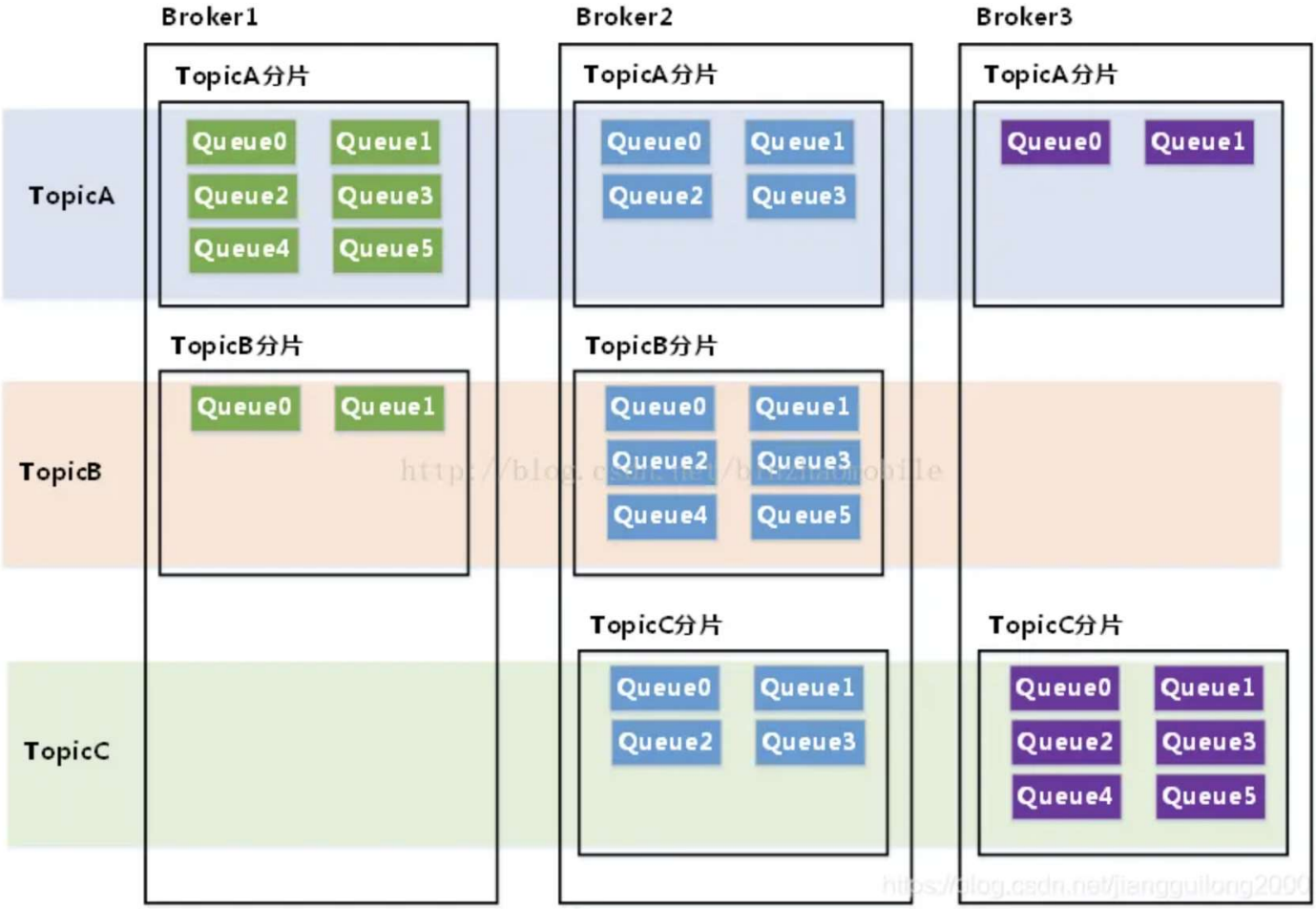


RocketMQ 架构

RocketMQ 技术架构中有四大角色 NameServer、Broker、Producer 和 Consumer，下面主要介绍 Broker。

Broker 用于存放 Queue，一个 Broker 可以配置多个 Topic，一个 Topic 中存在多个 Queue。

如果某个 Topic 消息量很大，应该给它多配置几个 Queue，并且尽量多分布在不同 broker 上，以减轻某个 broker 的压力。Topic 消息量都比较均匀的情况下，如果某个 broker 上的队列越多，则该 broker 压力越大。



简单提一下，Broker 通过集群部署，并且提供了 master/slave 的结构，salve 定时从 master 同步数据（同步刷盘或者异步刷盘），如果 master 宕机，则 slave 提供消费服务，但是不能写入消息。

看到这里，大家应该可以发现，RocketMQ 的设计和 Kafka 真的很像！

> 更多知识，详见《原理初探之 RocketMQ》

RabbitMQ

RabbitMQ 2007 年发布，是使用 Erlang 语言开发的开源消息队列系统，基于 AMQP 协议来实现。

AMQP 的主要特征是面向消息、队列、路由、可靠性、安全。**AMQP 协议更多用在企业系统内，对数据一致性、稳定性和可靠性要求很高的场景，对性能和吞吐量的要求还在其次。**

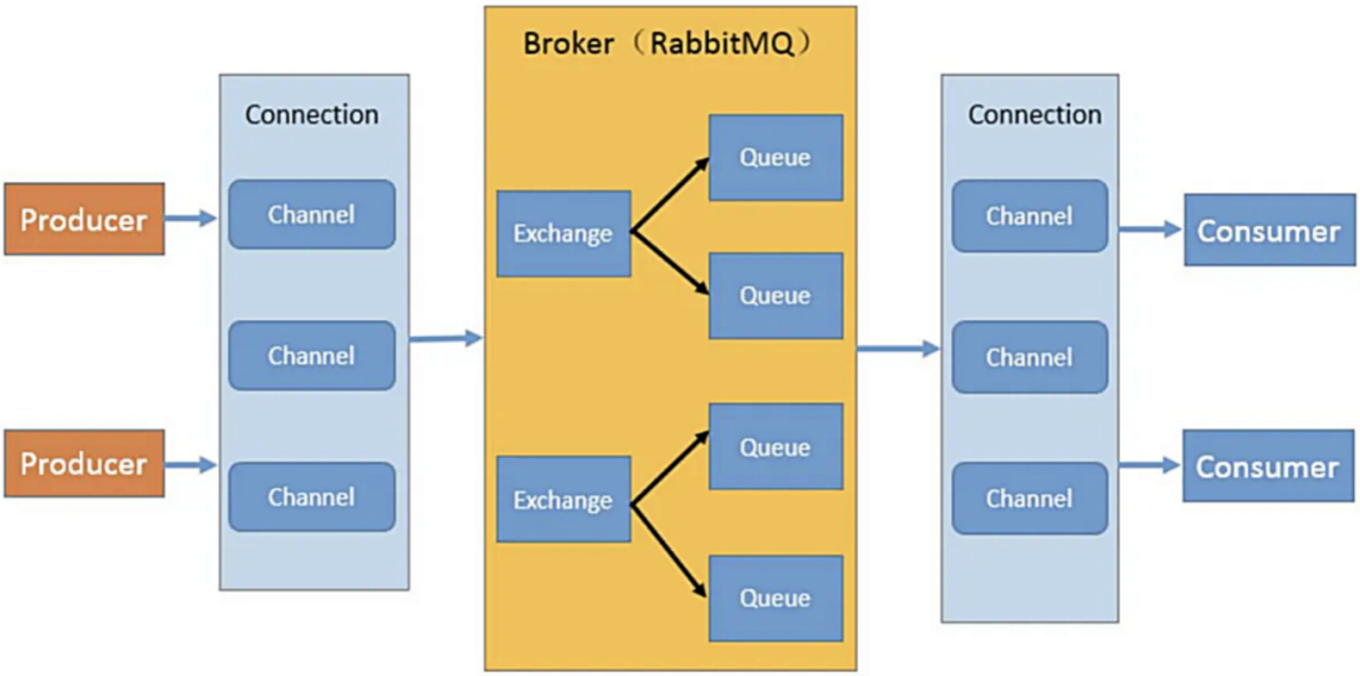
重要概念

- **信道（Channel）**：消息读写等操作在信道中进行，客户端可以建立多个信道，每个信道代表一个会话任务。
- **交换器（Exchange）**：接收消息，按照路由规则将消息路由到一个或者多个队列；如果路由不到，或者返回给生产者，或者直接丢弃。
- **路由键（RoutingKey）**：生产者将消息发送给交换器的时候，会发送一个 RoutingKey，用来指定路由规则，这样交换器就知道把消息发送到哪个队列。
- **绑定（Binding）**：交换器和消息队列之间的虚拟连接，绑定中可以包含一个或者多个 RoutingKey。

RabbitMQ 工作原理

AMQP 协议模型由三部分组成：生产者、消费者和服务端，执行流程如下：

1. 生产者是连接到 Server，建立一个连接，开启一个信道。
2. 生产者声明交换器和队列，设置相关属性，并通过路由键将交换器和队列进行绑定。
3. 消费者也需要进行建立连接，开启信道等操作，便于接收消息。
4. 生产者发送消息，发送到服务端中的虚拟主机。
5. 虚拟主机中的交换器根据路由键选择路由规则，发送到不同的消息队列中。
6. 订阅了消息队列的消费者就可以获取到消息，进行消费。



常用交换器

RabbitMQ 常用的交换器类型有 direct、topic、fanout、headers 四种，每种方法的详细介绍看这篇《入门 RabbitMQ，这一篇绝对够！》。

具体的使用方法，可以参考官网：

- 官网入口：https://www.rabbitmq.com/getstarted.html

1 "Hello World!"

The simplest thing that does *something*

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

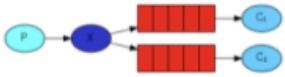
2 Work queues

Distributing tasks among workers (the competing consumers pattern)

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

3 Publish/Subscribe

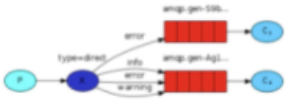
Sending messages to many consumers at once



- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift
- Spring AMQP

4 Routing

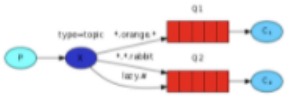
Receiving messages selectively



- Python

5 Topics

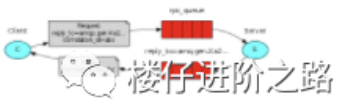
Receiving messages based on a pattern (topics)



- Python

6 RPC

Request/reply pattern example



- Python

> 更多知识，详见《入门RabbitMQ，这一篇绝对够！》

消息队列对比&选型

A	B	C	D	E
特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
PRODUCER-COMSUMER	支持	支持	支持	支持
PUBLISH-SUBSCRIBE	支持	支持	支持	支持
REQUEST-REPLY	支持	支持		
API完备性	高	高	高	高
多语言支持	支持，JAVA优先	语言无关	只支持JAVA	支持，java优先
单机吞吐量	万级	万级	万级	十万级
消息延迟		微秒级	毫秒级	毫秒级
可用性	高（主从）	高（主从）	非常高（分布式）	非常高（分布式）
消息丢失	低	低	理论上不会丢失	理论上不会丢失
消息重复		可控制		理论上会有重复
文档的完备性	高	高	高	高
提供快速入门	有	有	有	有
首次部署难度		低		中
社区活跃度	高	高	中	高
商业支持	无	无	阿里云	无
成熟度	成熟	成熟	比较成熟	成熟日志领域
特点	功能齐全，被大 量开源项目使用	由于Erlang 语言的并发能 力，性能很好	各个环节分 布式扩展设计，主从 HA；支持上万个队列；多 种消费模式；性能很好	
支持协议	OpenWire、 STOMP、 REST、XMPP、AMQP	AMQP	自己定义的一 套(社区提供 JMS--不成熟)	
持久化	内存、文件、数据库	内存、文件	磁盘文件	
事务	支持	不支持	支持	
负载均衡	支持	支持	支持	
管理界面	一般	好	有web console实现	
部署方式	独立、嵌入	独立	独立	
评价	优点：成熟的产品，已经在很多公 司得到应用（非大规模场景）。有 较多的文档。各种协议支持较好， 有多重语言的成熟的客户端； 缺点：根据其他用户反馈，会出莫 名其妙的问题，切会丢失消息。其 重心放到 activemq6.0 产品—apollo 上去了，目前社区不活跃，且对5.x 维护较少； Activemq 不适合用于上 千个队列的应用场景。	优点：由于erlang语言的特 性，mq性能较好；管理界面 较丰富，在互联网公司也有 较大规模的应用；支持amqp 系诶，有多中语言且支持 amqp的客户端可用； 缺点：erlang语言难度较 大。集群不支持动态扩展。	优点：模型简单，接口易用（JMS的接口很多 场合并不太实用）。在阿里大规模应用。目前 支付宝中的余额宝等新兴产品均使用 rocketmq 。集群规模大概在50台左右，单日处理消息上 百亿；性能非常好，可以大量堆积消息在 broker中；支持多种消费，包括集群消费、广 播消费等。开发度较活跃，版本更新很快。 缺点：产品较新文档比较缺乏。没有在mq核心 中去实现JMS等接口，对已有系统而言不能兼 容。阿里内部还有一套未开源的MQAPI，这一 层API可以将上层应用和下层MQ的实现解耦 （阿里内部有多个mq的实现，如notify、 metaq1.x，metaq2.x，rocketmq等），使得下 面mq可以很方便的进行切换和升级而对应用无 任何影响，目前这一套东西未开源。	

消息队列对比

Kafka

优点：

- **高吞吐、低延迟**：Kafka 最大的特点就是收发消息非常快，Kafka 每秒可以处理几十万条消息，它的最低延迟只有几毫秒；
- **高伸缩性**：每个主题（topic）包含多个分区（partition），主题中的分区可以分布在不同的主机（broker）中；
- **高稳定性**：Kafka 是分布式的，一个数据多个副本，某个节点宕机，Kafka 集群能够正常工作；
- **持久性、可靠性、可回溯**： Kafka 能够允许数据的持久化存储，消息被持久化到磁盘，并支持数据备份防止数据丢失，支持消息回溯；
- 消息有序：通过控制能够保证所有消息被消费且仅被消费一次；
- 有优秀的第三方 Kafka Web 管理界面 Kafka-Manager，在日志领域比较成熟，被多家公司和多个开源项目使用。

缺点：

- Kafka 单机超过 64 个队列/分区，Load 会发生明显的飙高现象，队列越多，load 越高，发送消息响应时间变长；
- **不支持消息路由，不支持延迟发送，不支持消息重试；**
- **社区更新较慢。**

RocketMQ

优点：

- **高吞吐**：借鉴 Kafka 的设计，单一队列百万消息的堆积能力；
- **高伸缩性**：灵活的分布式横向扩展部署架构，整体架构其实和 kafka 很像；
- **高容错性**：通过ACK机制，保证消息一定能正常消费；
- **持久化、可回溯**：消息可以持久化到磁盘中，支持消息回溯；
- 消息有序：在一个队列中可靠的先进先出（FIFO）和严格的顺序传递；
- 支持发布/订阅和点对点消息模型，支持拉、推两种消息模式；
- 提供 docker 镜像用于隔离测试和云集群部署，提供配置、指标和监控等功能丰富的 Dashboard。

缺点：

- 不支持消息路由，**支持的客户端语言不多，目前是 java 及 c++，其中 c++ 不成熟**；
- 部分支持消息有序：需要将同一类的消息 hash 到同一个队列 Queue 中，才能支持消息的顺序，如果同一类消息散落到不同的 Queue中，就不能支持消息的顺序。
- **社区活跃度一般。**

RabbitMQ

优点：

- **支持几乎所有最受欢迎的编程语言**：Java，C，C ++，C # ，Ruby，Perl，Python，PHP等等；
- **支持消息路由**：RabbitMQ 可以通过不同的交换器支持不同种类的消息路由；
- **消息时序**：通过延时队列，可以指定消息的延时时间，过期时间TTL等；
- 支持容错处理：通过交付重试和死信交换器（DLX）来处理消息处理故障；
- 提供了一个易用的用户界面，使得用户可以监控和管理消息 Broker；
- **社区活跃度高。**

缺点：

- **Erlang 开发，很难去看懂源码，不利于做二次开发和维护**，基本职能依赖于开源社区的快速维护和修复 bug；
- **RabbitMQ 吞吐量会低一些**，这是因为他做的实现机制比较重；
- 不支持消息有序、持久化不好、不支持消息回溯、伸缩性一般。

消息队列选型

- Kafka：追求高吞吐量，一开始的目的就是用于日志收集和传输，**适合产生大量数据的互联网服务的数据收集业务**，大型公司建议可以选用，**如果有日志采集功能，肯定是首选 kafka。**
- RocketMQ：**天生为金融互联网领域而生，对于可靠性要求很高的场景**，尤其是电商里面的订单扣款，以及业务削峰，在大量交易涌入时，后端可能无法及时处理的情况。RoketMQ 在稳定性上可能更值得信赖，这些业务场景在阿里双 11 已经经历了多次考验，**如果你的业务有上述并发场景，建议可以选择 RocketMQ。**

- RabbitMQ：结合 erlang 语言本身的并发优势，性能较好，社区活跃度也比较高，但是不利于做二次开发和维护，不过 RabbitMQ 的社区十分活跃，可以解决开发过程中遇到的 bug。**如果你的数据量没有那么大，小公司优先选择功能比较完备的 RabbitMQ。**
- ActiveMQ：官方社区现在对 ActiveMQ 5.x 维护越来越少，**较少在大规模吞吐的场景中使用。**

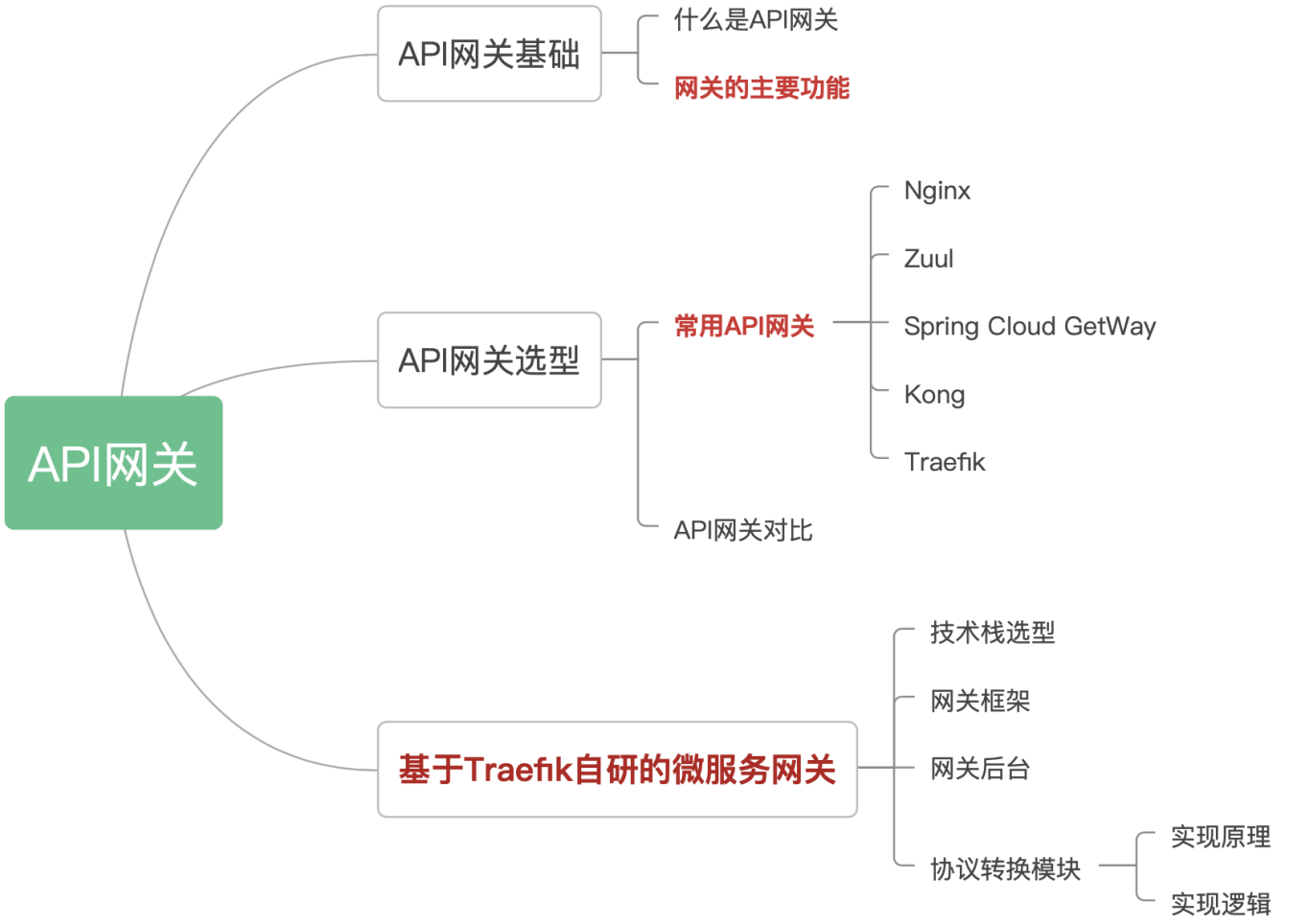
📚 第 2 章：微服务网关

> 常用API网关的对比和选型，并讲解我司自研的微服务网关，干货满满！

大家好，我是楼仔！微服务近几年非常火，围绕微服务的技术生态也比较多，比如微服务网关、Docker、Kubernetes等。

我是于2019年开始接触微服务网关，当时和公司的一位同事一起开发，由于技术能力有限，我只负责网关后台，后续微服务网关的迭代，我其实没有参与，不过后来抽空看了微服务网关前台的代码，所以对这套微服务网关的实现原理算是基本掌握。

最近在写技术栈相关的文章，刚好写到微服务网关，就把之前学习的知识进行简单总结，同时也把市面上常用的微服务网关进行梳理，一方面便于后续技术选型，另一方面也算是给自己一个交代。下面是文章目录：



API网关基础

什么是API网关

API网关是一个服务器，是系统的唯一入口。 从面向对象设计的角度看，它与外观模式类似。

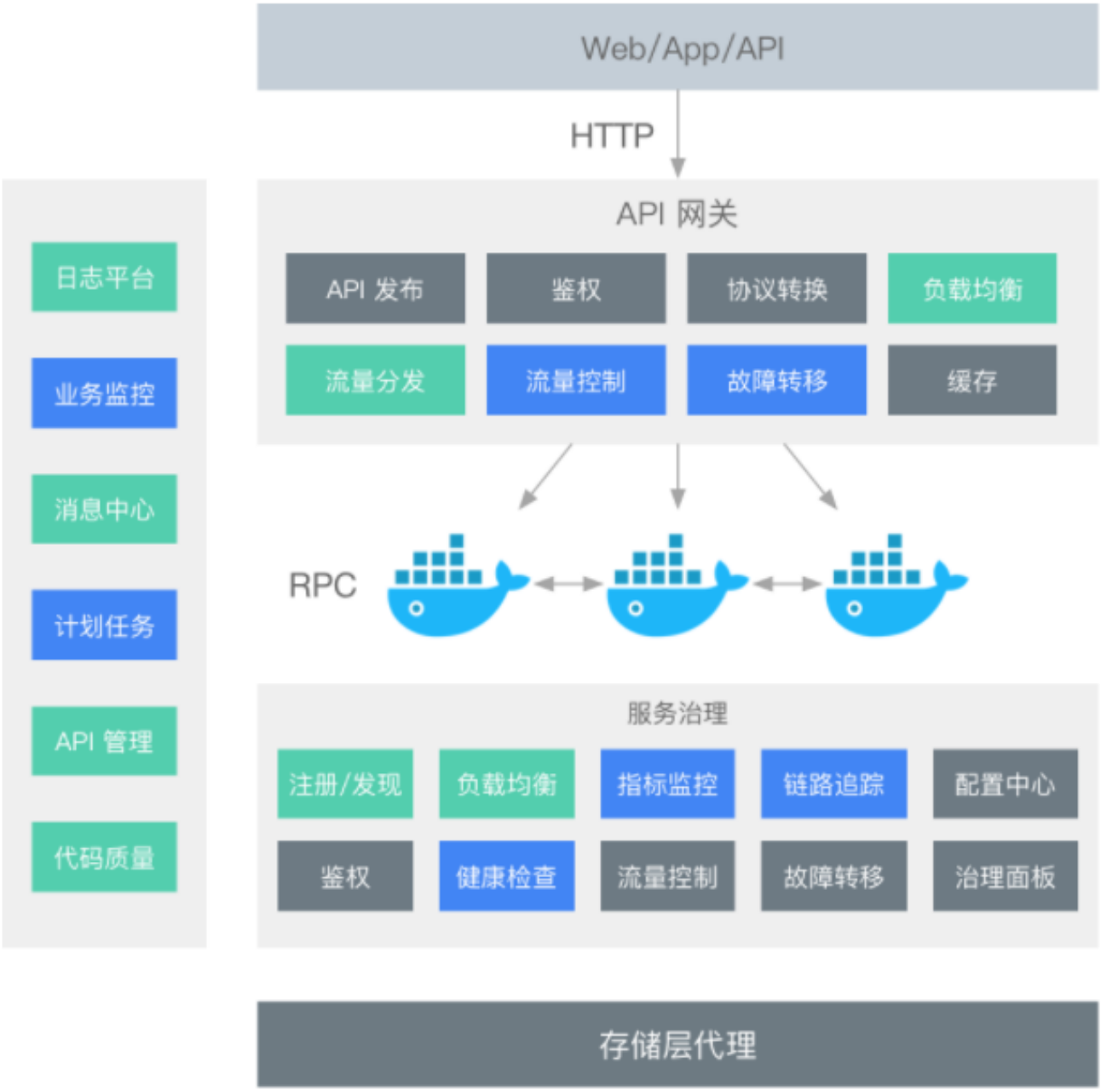
API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、协议转换、限流熔断、静态响应处理。

API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。通常，网关也是提供REST/HTTP的访问API。

网关的主要功能

微服务网关作为微服务后端服务的统一入口，它可以统筹管理后端服务，主要分为数据平面和控制平面：

- 数据平面主要功能是接入用户的HTTP请求和微服务被拆分后的聚合。使用微服务网关统一对外暴露后端服务的API和契约，路由和过滤功能正是网关的核心能力模块。另外，微服务网关可以实现拦截机制和专注跨横切面的功能，包括协议转换、安全认证、熔断限流、灰度发布、日志管理、流量监控等。
- 控制平面主要功能是对后端服务做统一的管控和配置管理。例如，可以控制网关的弹性伸缩；可以统一下发配置；可以对网关服务添加标签；可以在微服务网关上通过配置Swagger功能统一将后端服务的API契约暴露给使用方，完成文档服务，提高工作效率和降低沟通成本。



- **路由功能**：路由是微服务网关的核心能力。通过路由功能微服务网关可以将请求转发到目标微服务。在微服务架构中，网关可以结合注册中心的动态服务发现，实现对后端服务的发现，调用方只需要知道网关对外暴露的服务API就可以透明地访问后端微服务。
- **负载均衡**：API网关结合负载均衡技术，利用Eureka或者Consul等服务发现工具，通过轮询、指定权重、IP地址哈希等机制实现下游服务的负载均衡。
- **统一鉴权**：一般而言，无论对内网还是外网的接口都需要做用户身份认证，而用户认证在一些规模较大的系统中都会采用统一的单点登录（Single Sign On）系统，如果每个微服务都要对接单点登录系统，那么显然比较浪费资源且开发效率低。API网关是统一管理安全性的绝佳场所，可以将认证的部分抽取到网关层，微服务系统无须关注认证的逻辑，只关注自身业务即可。
- **协议转换**：API网关的一大作用在于构建异构系统，API网关作为单一入口，通过协议转换整合后台基于REST、AMQP、Dubbo等不同风格和实现技术的微服务，面向Web Mobile、开放平台等特定客户端提供统一服务。
- **指标监控**：网关可以统计后端服务的请求次数，并且可以实时地更新当前的流量健康状态，可以对URL粒度的服务进行延迟统计，也可以使用Hystrix Dashboard查看后端服务的流量状态及是否有熔断发生。
- **限流熔断**：在某些场景下需要控制客户端的访问次数和访问频率，一些高并发系统有时还会有限流的需求。在

网关上可以配置一个阈值，当请求数超过阈值时就直接返回错误而不继续访问后台服务。当出现流量洪峰或者后端服务出现延迟或故障时，网关能够主动进行熔断，保护后端服务，并保持前端用户体验良好。

- **黑白名单**：微服务网关可以使用系统黑名单，过滤HTTP请求特征，拦截异常客户端的请求，例如DDoS攻击等侵蚀带宽或资源迫使服务中断等行为，可以在网关层面进行拦截过滤。比较常见的拦截策略是根据IP地址增加黑名单。在存在鉴权管理的路由服务中可以通过设置白名单跳过鉴权管理而直接访问后端服务资源。
- **灰度发布**：微服务网关可以根据HTTP请求中的特殊标记和后端服务列表元数据标识进行流量控制，实现在用户无感知的情况下完成灰度发布。
- **流量染色**：和灰度发布的原理相似，网关可以根据HTTP请求的Host、Head、Agent等标识对请求进行染色，有了网关的流量染色功能，我们可以对服务后续的调用链路进行跟踪，对服务延迟及服务运行状况进行进一步的链路分析。
- **文档中心**：网关结合Swagger，可以将后端的微服务暴露给网关，网关作为统一的入口给接口的使用方提供查看后端服务的API规范，不需要知道每一个后端微服务的Swagger地址，这样网关起到了对后端API聚合的效果。
- **日志审计**：微服务网关可以作为统一的日志记录和收集器，对服务URL粒度的日志请求信息和响应信息进行拦截。

API网关选型

常用API网关

先简单看一下市面上常用的API网关：

Nginx(2004)	Nginx Inc	C/Lua	高性能，成熟稳固	门槛高,偏运维,可编程弱
Zuul1(2012)	Netflix/Pivotal	Java	成熟,简略门槛低	性能个别，可编程个别
Spring Cloud Gateway(2016)	Pivotal	Java	异步,配置灵便	晚期产品
Envoy(2016)	Lyft	C++	高性能,可编程API/ServiceMesh集成	门槛较高
Kong(2014)	Kong Inc	OpenResty/Lua	高性能,可编程API	门槛较高
Traefik(2015)	Containous	Golang	云原生,可编程API/对接各种服务发现	生产案例不太多

Nginx

Nginx是一个高性能的HTTP和反向代理服务器。**Nginx一方面可以做反向代理，另外一方面可以做静态资源服务器，接口使用Lua动态语言可以完成灵活的定制功能。**

Nginx 在启动后，会有一个 Master 进程和多个 Worker 进程，Master 进程和 Worker 进程之间是通过进程间通信进行交互的，如图所示。Worker 工作进程的阻塞点是在像 select()、epoll_wait() 等这样的 I/O 多路复用函数调用处，以等待发生数据可读 / 写事件。Nginx 采用了异步非阻塞的方式来处理请求，也就是说，Nginx 是可以同时处理成千上万个请求的。

Zuul

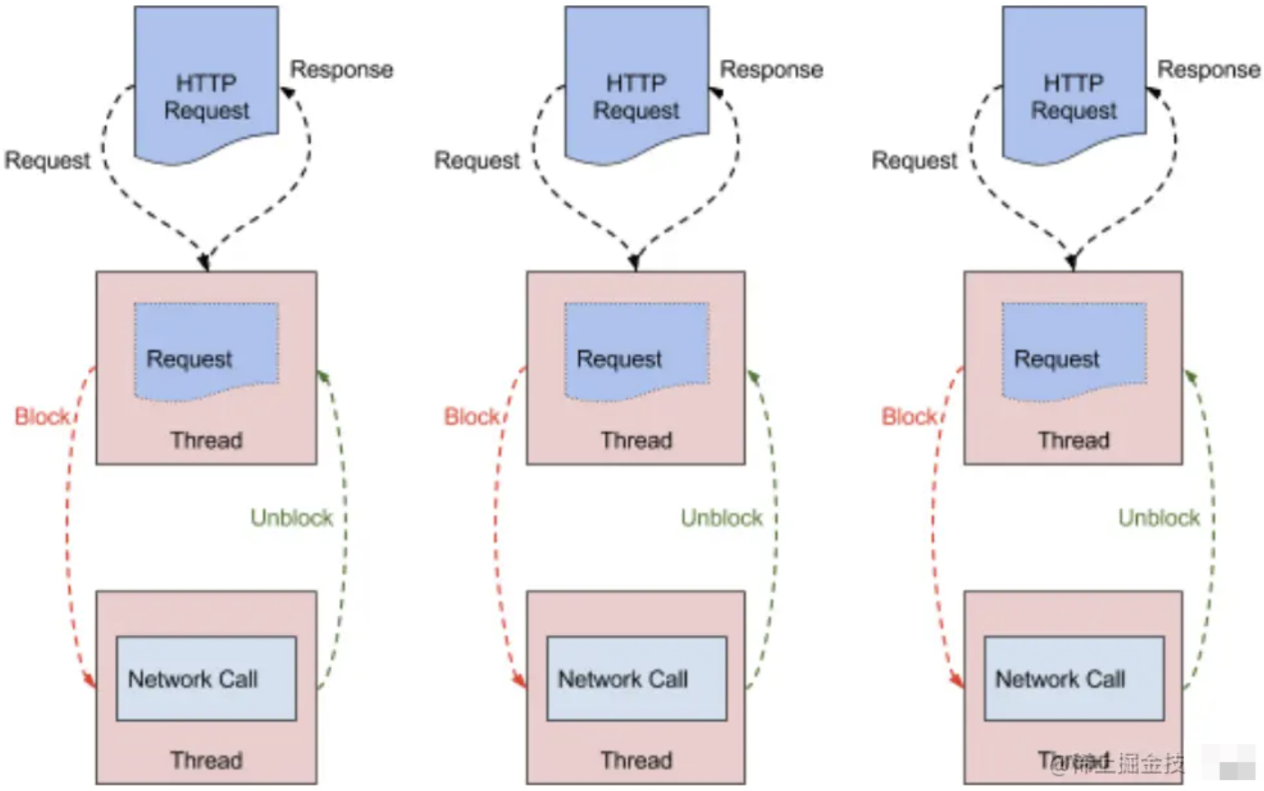
Zuul 是 Netflix 开源的一个API网关组件，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。社区活跃，融合于 SpringCloud 完整生态，是构建微服务体系前置网关服务的最佳选型之一。

Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能：

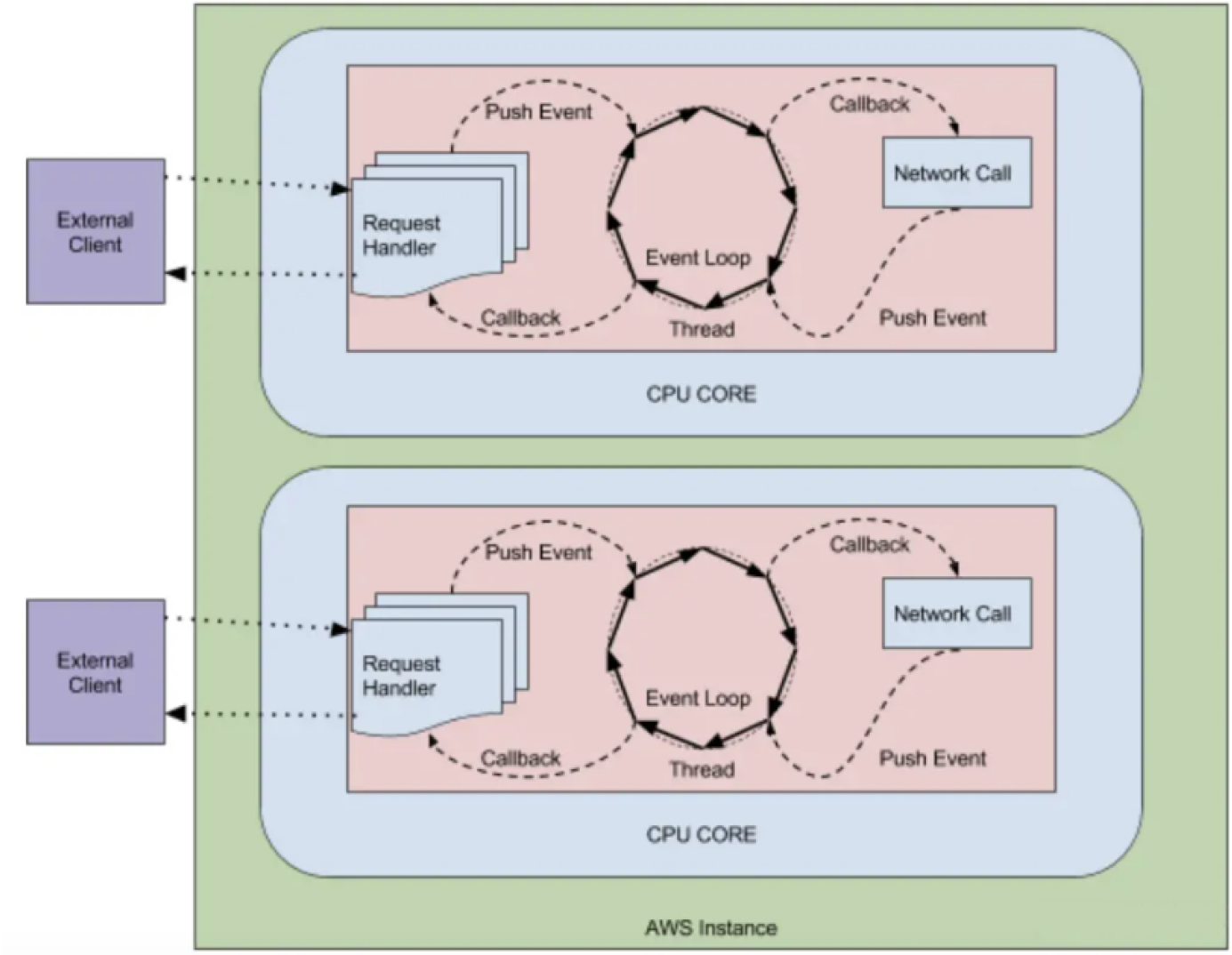
- **统一鉴权 + 动态路由 + 负载均衡 + 压力测试**
- **审查与监控**：与边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
- **多区域弹性**：跨越 AWS Region 进行请求路由，旨在实现 ELB（Elastic Load Balancing，弹性负载均衡）使用的多样化，以及让系统的边缘更贴近系统的使用者。

Zuul 目前有两个大的版本：**Zuul1 和 Zuul2**

Zuul1 是基于 Servlet 框架构建，如图所示，采用的是阻塞和多线程方式，即一个线程处理一次连接请求，这种方式在内部延迟严重、设备故障较多情况下会引起存活的连接增多和线程增加的情况发生。



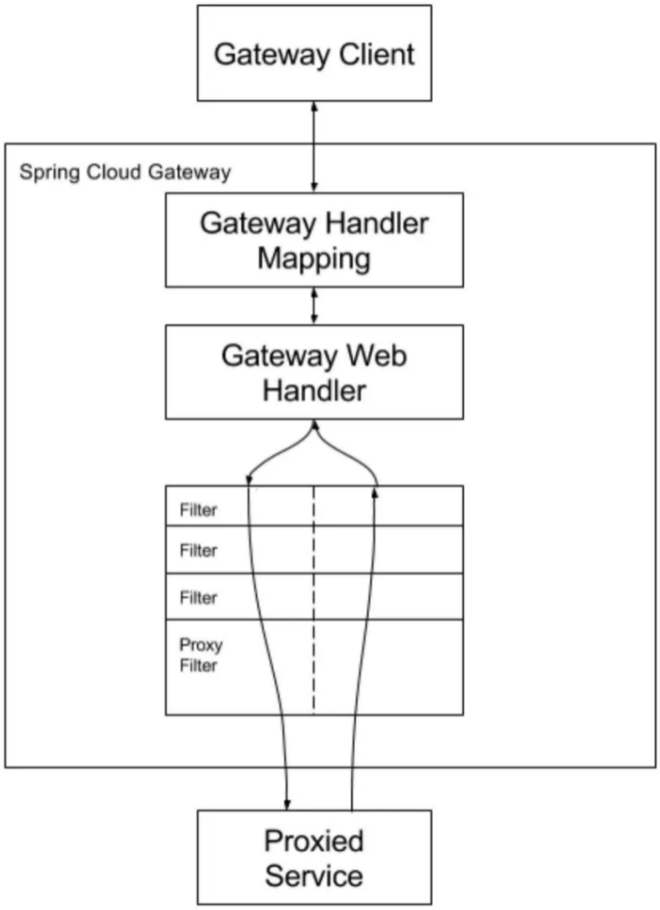
Netflix 发布的 Zuul2 有重大的更新，它运行在异步和无阻塞框架上，每个 CPU 核一个线程，处理所有的请求和响应，请求和响应的生命周期是通过事件和回调来处理的，这种方式减少了线程数量，因此开销较小。



Spring Cloud GetWay

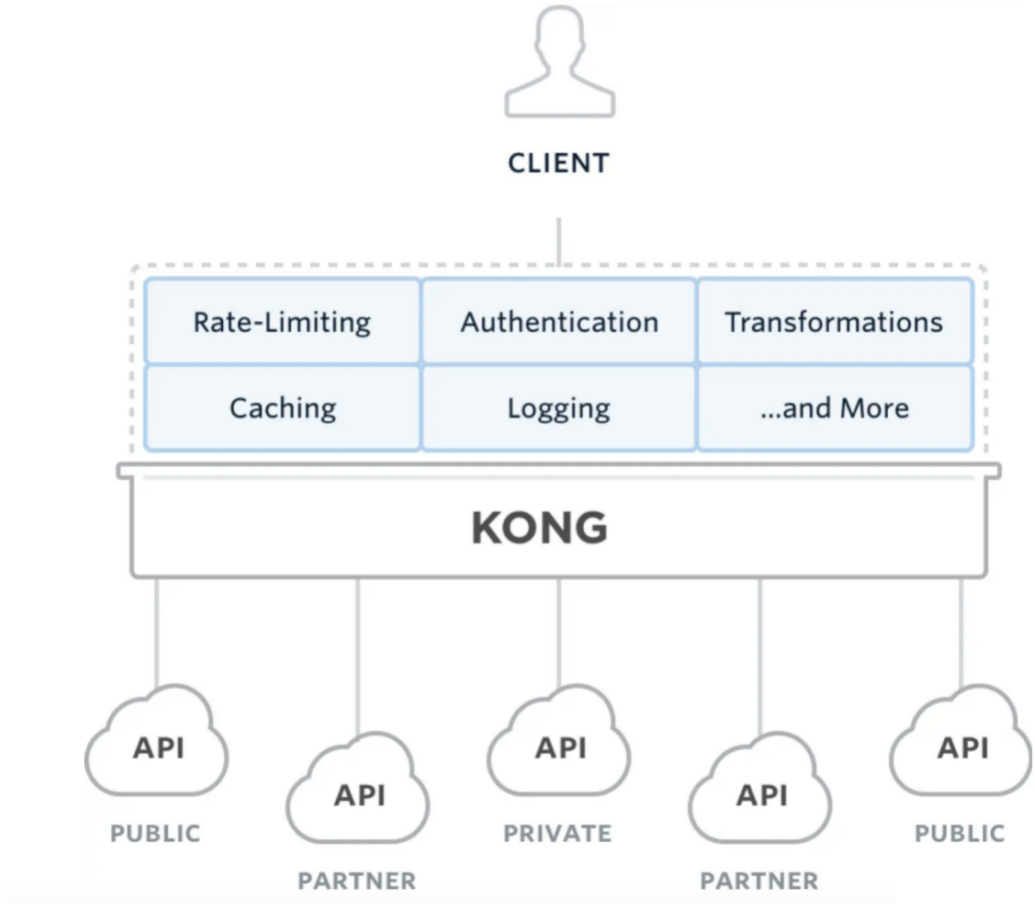
Spring Cloud Gateway 是Spring Cloud的一个全新的API网关项目，目的是为了替换掉Zuul1，它基于Spring5.0 + SpringBoot2.0 + WebFlux（基于高性能的Reactor模式响应式通信框架Netty，异步非阻塞模型）等技术开发，性能高于Zuul，官方测试，**Spring Cloud GateWay是Zuul的1.6倍**，旨在为微服务架构提供一种简单有效的统一的API路由管理方式。

Spring Cloud Gateway可以与Spring Cloud Discovery Client（如Eureka）、Ribbon、Hystrix等组件配合使用，**实现路由转发、负载均衡、熔断、鉴权、路径重写、日志监控等，并且Gateway还内置了限流过滤器，实现了限流的功能。**



Kong

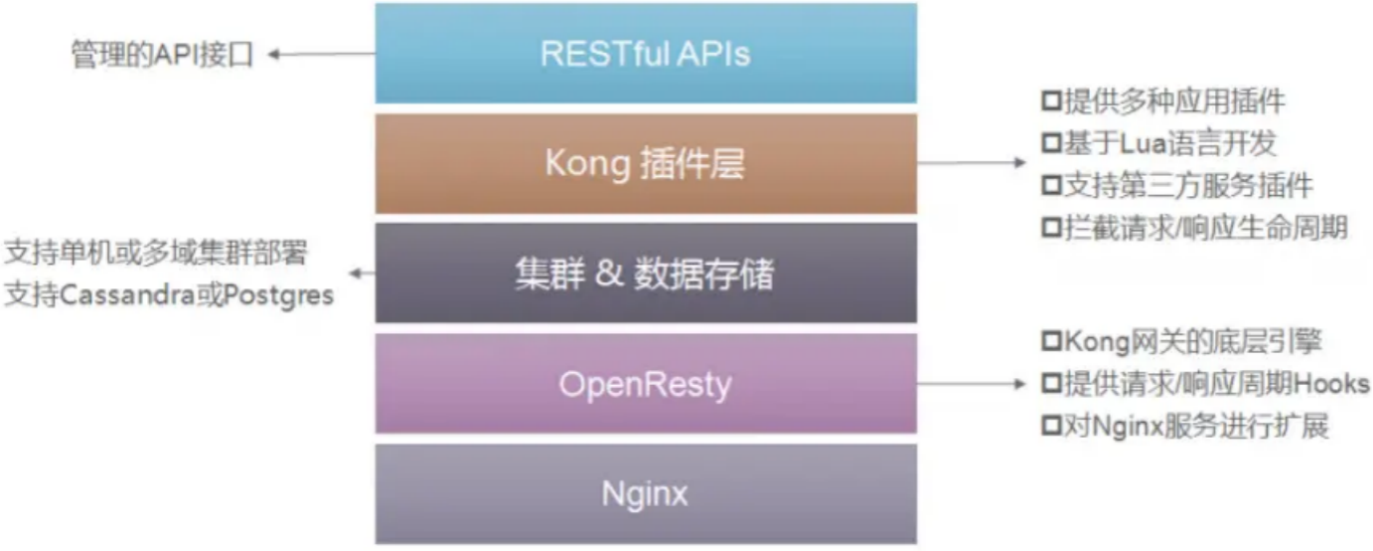
Kong是一款基于OpenResty（Nginx + Lua模块）编写的高可用、易扩展的，由Mashape公司开源的API Gateway项目。**Kong是基于NGINX和Apache Cassandra或PostgreSQL构建的**，能提供易于使用的RESTful API来操作和配置API管理系统，所以它可以水平扩展多个Kong服务器，通过前置的负载均衡配置把请求均匀地分发到各个Server，来应对大批量的网络请求。



Kong主要有三个组件：

- Kong Server ： 基于Nginx的服务器，用来接收API请求。
- Apache Cassandra/PostgreSQL ： 用来存储操作数据。
- Kong dashboard：官方推荐UI管理工具，也可以使用 restfull 方式管理admin api。

Kong采用插件机制进行功能定制，插件集（可以是0或N个）在API请求响应循环的生命周期中被执行。插件使用Lua编写，目前已有几个基础功能：**HTTP基本认证、密钥认证、CORS（Cross-Origin Resource Sharing，跨域资源共享）、TCP、UDP、文件日志、API请求限流、请求转发以及Nginx监控。**



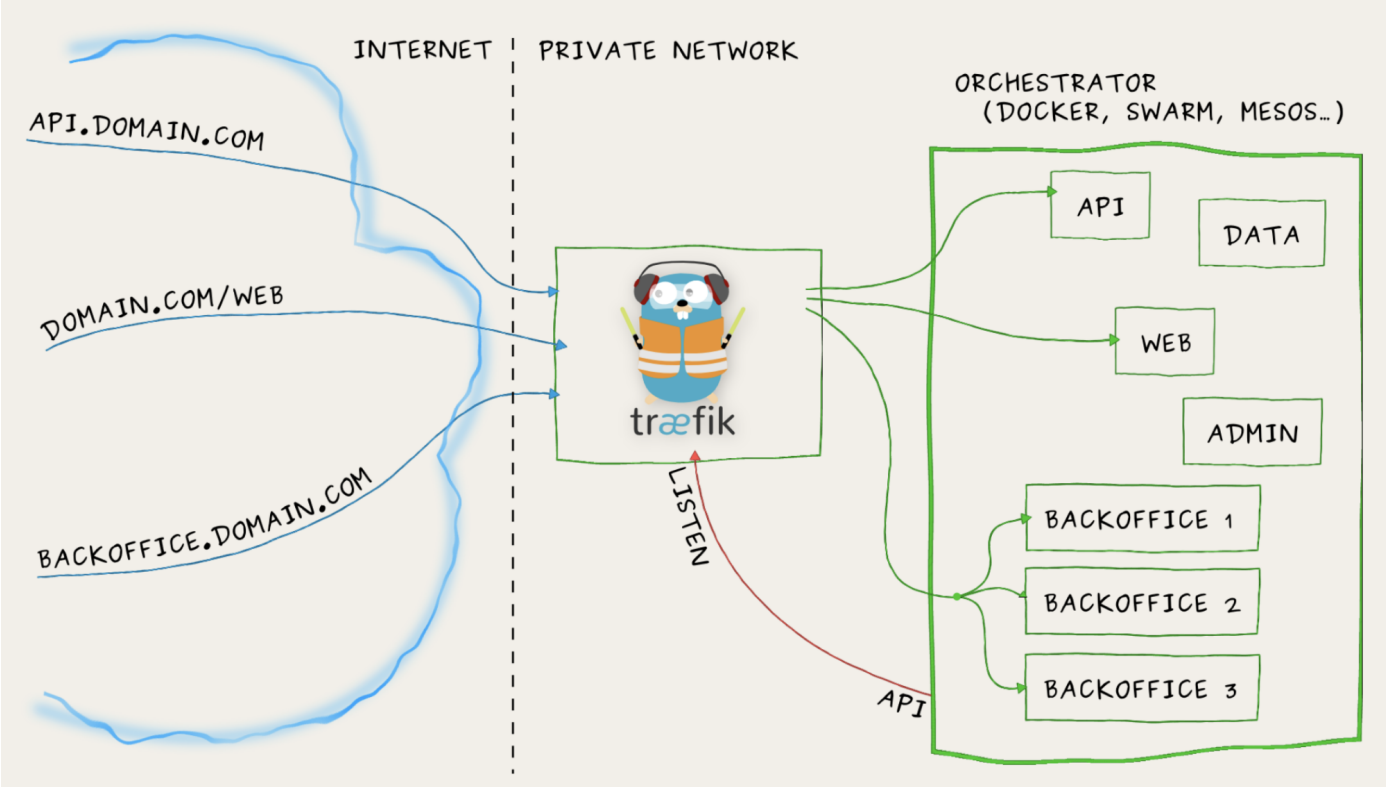
Kong网关具有以下的特性：

- 可扩展性: 通过简单地添加更多的服务器，可以轻松地进行横向扩展，这意味着您的平台可以在一个较低负载

的情况下处理任何请求；
- 模块化: 可以通过添加新的插件进行扩展，这些插件可以通过RESTful Admin API轻松配置；
- 在任何基础架构上运行: Kong网关可以在任何地方都能运行。您可以在云或内部网络环境中部署Kong，包括单个或多个数据中心设置，以及public，private 或invite-only APIs。

Traefik

Træfık 是一个为了让部署微服务更加便捷而诞生的现代HTTP反向代理、负载均衡工具。 它支持多种后台 (Docker, Swarm, Kubernetes, Marathon, Mesos, Consul, Etcd, Zookeeper, BoltDB, Rest API, file...) 来自动化、动态的应用它的配置文件设置。



重要特性：

- 它非常快，无需安装其他依赖，通过Go语言编写的单一可执行文件；
- **多种后台支持：Docker, Swarm, Kubernetes, Marathon, Mesos, Consul, Etcd；**
- 支持支持Rest API、Websocket、HTTP/2、Docker镜像；
- 监听后台变化进而自动化应用新的配置文件设置；
- 配置文件热更新，无需重启进程；
- 后端断路器、负载均衡、容错机制；
- 清爽的前端页面，可监控服务指标。

> 关于Traefik的更多内容，可以查看官网：https://traefik.cn/

API网关对比

	Kong	Traefik	Ambassador	Tyk	Zuul
基本					
主要用途	企业级API管理	微服务网关	微服务网关	微服务网关	微服务网关
学习曲线	适中	simple	simple	适中	simple
成本	开源/企业版	开源	开源/pro	开源/企业版	开源
社区star	20742	21194	1719	4299	7186
配置					
配置语言	Admin Rest api, Text file(nginx.conf 等)	TOML	YAML(kubernetes annotation)	Tyk REST API	REST API，YAML静态配置
配置端点类型	命令式	声明式	声明式	命令式	命令式
拖拽配置	yes	no	no	no	no
管理模式	configurable	decentralised, self-service	decentralised, self-service	decentralised, self-service	decentralised, self-service
部署					
kubernetes	适中(k8s yaml,helm chart)	easy	easy	适中(k8s yaml,helm chart)	适中(k8s yaml,helm chart)
Cloud IAAS	high	easy	N/A	easy	easy
Private Data Center	high	easy	N/A	easy	easy
部署模式	金丝雀(企业版)	金丝雀	金丝雀，shadow	金丝雀	金丝雀
state	postgres,cassandra	kubernetes	kubernetes	redis	内存文件
可扩展性					
扩展功能	插件	自己实现	插件	插件	自己实现
扩展方法	水平	水平	水平	水平	水平
功能					
服务发现	动态	动态	动态	动态	动态
协议	http,https,websocket	http,https,grpc,websocket	http,https,grpc,websocket	http,https,grpc,websocket	http,https
基于	kong+nginx	traefik	envoy	tyk	zuul
ssl 终止	yes	yes	yes	yes	no
websocket	yes	yes	yes	yes	no
routing	host,path,method	host,path	host,path,header	host,path	
限流	yes	no	yes	yes	需要开发
熔断	yes	yes	no	yes	需要其他组件
重试	yes	yes	no	yes	yes
健康检查	yes	no	no	yes	yes
负载均衡算					轮询，随机，

法	轮询，哈希	轮询，加权轮询	加权轮询	轮询	加权轮询，自定义
权限	Basic Auth, HMAC, JWT, Key, LDAP, OAuth 2.0, PASETO, plus paid Kong Enterprise options like OpenID Connect	basic	yes	HMAC，JWT，Mutual TLS，OpenID Connect，基本身份验证，LDAP，社交OAuth（例如GPlus，Twitter，Github）和传统基本身份验证提供程序	开发实现
tracing	yes	yes	yes	yes	需要其他组件
istio集成	no	no	yes	no	no
dashboard	yes	yes	grafana,Prometheus	yes	no

上面是网关对比截图，偷个懒，大家主要关注Kong、Traefik和Zuul即可：

- 从**开源社区活跃度**来看，无疑是Kong和Traefik较好；
- 从**成熟度**来看，较好的是Kong、Tyk、Traefik；
- 从**性能**来看，Kong要比其他几个领先一些；
- 从**架构优势**的扩展性来看，Kong、Tyk有丰富的插件，Ambassador也有插件但不多，而Zuul是完全需要自研，但Zuul由于与Spring Cloud深度集成，使用度也很高，近年来Istio服务网格的流行，Ambassador因为能够和Istio无缝集成也是相当大的优势。

下面是其它网友的思考结论，可供参考：

- **性能**：Nginx+Lua形式必然是高于Java语言实现的网关的，Java技术栈里面Zuul1.0是基于Servlet实现的，剩下都是基于webflux实现，性能是高于基于Servlet实现的。**在性能方面我觉得选择网关可能不算那么重要，多加几台机器就可以搞定。**
- **可维护性和扩展性**：Nginx+Lua这个组合掌握的人不算多，如果团队有大神，大佬们就随意了，当没看到这段话，对于一般团队来说的话，选择自己团队擅长的语言更重要。Java技术栈下的3种网关，对于Zuul和Spring Cloud Getway需要或多或少要搞一些集成和配置页面来维护，但是对于Soul我就无脑看看文章，需要哪个搬哪个好了，尤其是可以无脑对接Dubbo美滋滋，此外Soul2.0以后版本可以摆脱ZK，在我心里再无诟病，我就喜欢无脑操作。
- **高可用**：对于网关高可用基本都是统一的策略都是采用多机器部署的方式，前面挂一个负载，对于而外需要用的一些组件大家注意一下。

基于Traefik自研的微服务网关

这个是我司自研的微服务网关，基于Traefik进行开发，下面从技术选型、网关框架、网关后台、协议转换进行讲解，绝对干货！

技术栈选型

- **Traefik**：一款开源的反向代理与负载均衡工具，它最大的优点是能够与常见的微服务系统直接整合，可以实现自动化动态配置。traefik较为轻量，非常易于使用和设置，性能比较好，已在全球范围内用于生产环境。
- **Etcd**：一个Go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。（更多内容可以查看文章 肝了一个月的ETCD，从Raft原理到实践 ）
- **Go**：并发能力强，性能媲美C，处理能力是PHP的4倍，效率高，语法简单，易上手，开发效率接近PHP。

技术栈选型：GO ＋ ETCD

GO

原子操作，基于CPU指令实现，避免锁竞争
GMP调度模型，天然支持高并发

Traefik

一款开源的反向代理与负载均衡工具
适合与微服务系统结合，实现自动化动态配置

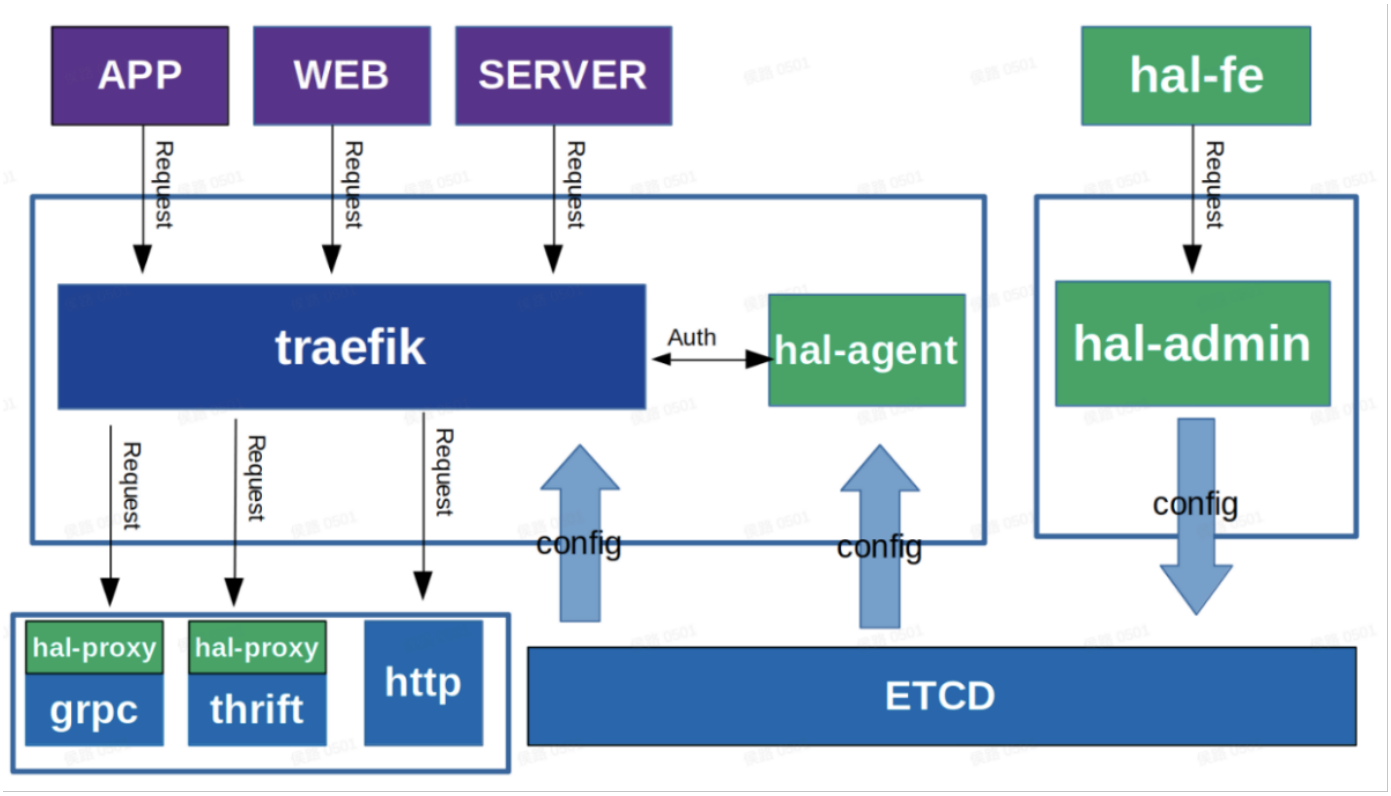
ETCD

分布式、高可用的一致性键值存储系统
提供可靠的配置共享和服务发现功能

网关框架

整个网关框架分为3块：

- **网关后台**（hal-fe和hal-admin）：用于应用、服务和插件的配置，然后将配置信息发布到ETCD；
- **Traefik**：读取ETCD配置，根据配置信息对请求进行路由分发，如果需要鉴权，会直接通过hal-agent模块进行统一鉴权。鉴权完毕后，如果是Http请求，直接打到下游服务，如果是Grpc和Thrift协议，会通过hal-proxy模块进行协议转换。
- **协议转换模块**：读取ETCD配置，对Traefik分发过来的请求，进行Grpc和Thrift协议转换（更多内容可以查看文章 RPC框架：从原理到选型，一文带你搞懂RPC），并通过服务发现机制，获取服务下游机器，并通过负载均衡，将转换后的数据打到下游服务机器。

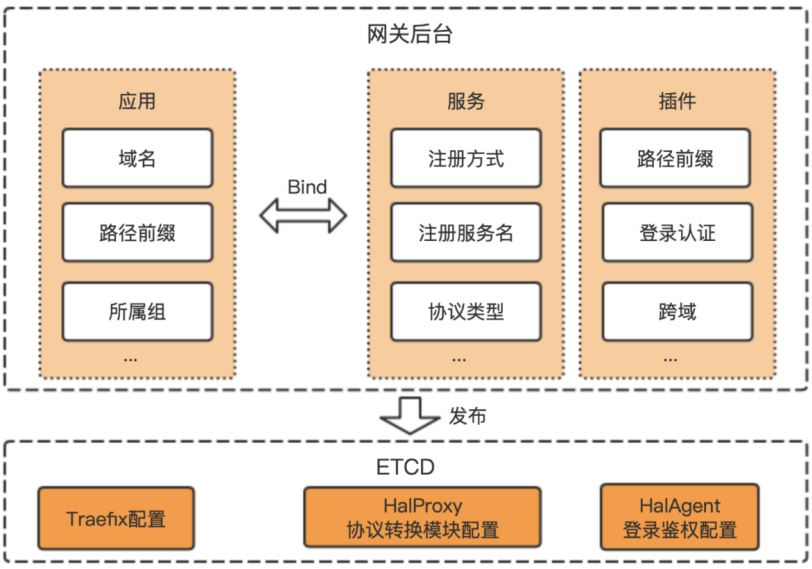


网关后台

主要由3大模块组成：

- **应用**：主要包括应用名、域名、路径前缀、所属组、状态等，比如印度海外商城、印度社区；
- **服务**：主要包括服务名、注册方式、协议类型、所属组、状态等，比如评论服务、地址服务、搜索服务。
- **插件**：主要包括插件名称、插件类型、插件属性配置等，比如路径前缀替换插件、鉴权插件。

一个应用只能绑定一个服务，但是可以绑定多个插件。 通过后台完成网关配置后，将这些配置信息生成Config文件，发布到ETCD中，Config文件需要遵循严格的数据格式，比如Traefix配置需要遵循官方的文件配置格式，才能被Traefik识别。



Traefxi配置文件

```
[http.middlewares]
  [http.middlewares.address_strip_prefix]
    [http.middlewares.address_strip_prefix.stripPrefix]
      prefixes = ["/wms/address"]
[http.routers]
  [http.routers.wms_service_mi_address_service_268]
    entryPoints = ["http"]
    middlewares =
["wms_service_mi_address_service_268_proxy_info","address_ai
    rule = "Host(`xxx_url`) && PathPrefix(`/wms/address`)"
    service = "mi_address_service"
[http.services]
  [http.services.mi_address_service]
    [http.services.mi_address_service.loadbalancer]
      [[http.services.mi_address_service.loadbalancer.servers]]
        url = "http://xxx_IP:xxx_Port"
```

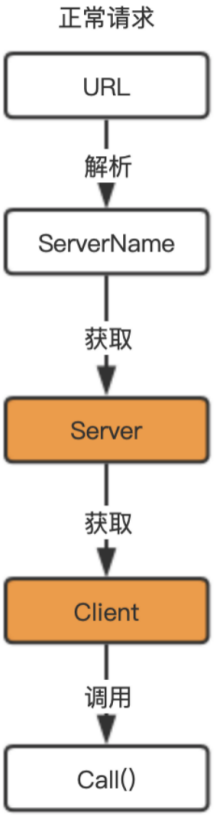
协议转换模块

hal-proxy模块是整个微服务网关最复杂，也是技术含量最高的模块，所以给大家详细讲解一下。

问题引入

在讲这个模块前，我们先看下面几个问题：

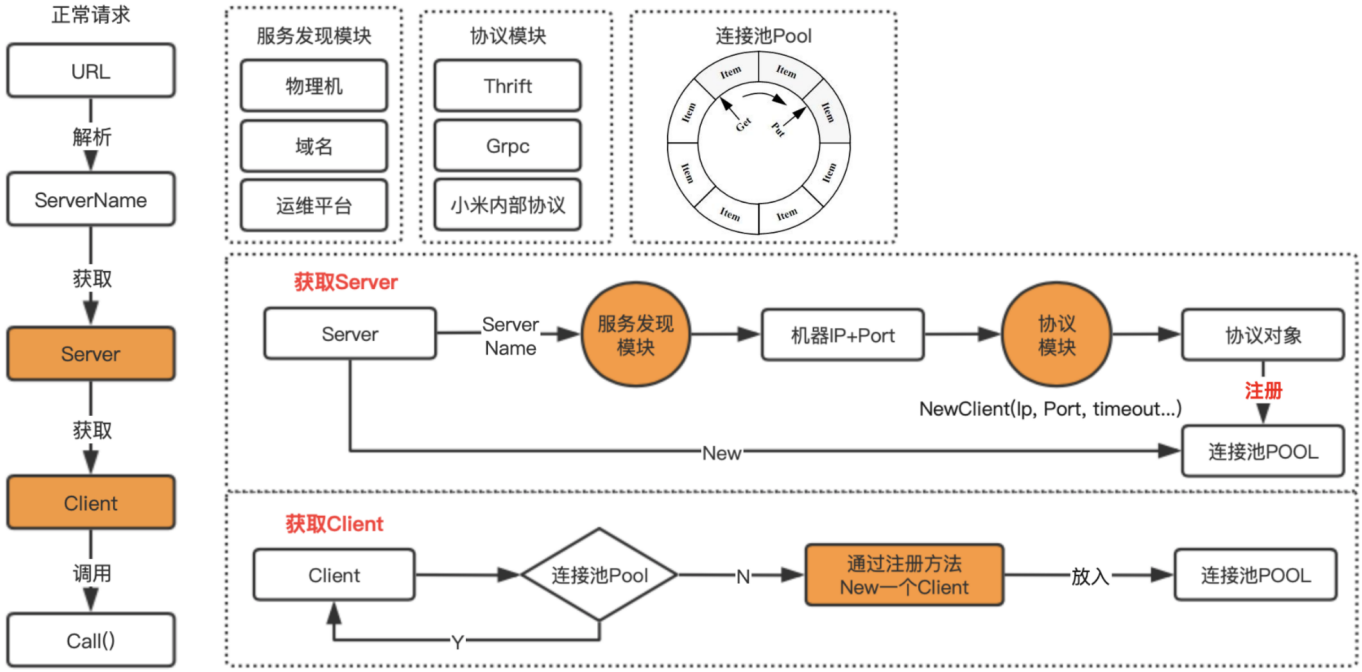
- 当请求从上游的trafik过来时，需要知道访问下游的机器IP和端口，才能将请求发送给下游，这些机器如何获取呢？
- 有了机器后，我们需要和下游机器建立连接，如果连接用一次就直接释放，肯定对服务会造成很大的压力，这就需要引入Client缓存池，那这个Client缓存池我们又该如何实现呢？
- 最后就是需要对协议进行转换，因为不同的下游服务，支持的协议类型是不一样的，这个网关又是如何动态支持的呢？



hal—proxy协议转换模块需要解决的问题：

1. 如何通过服务发现，获取服务机器IP和端口信息？
2. Client缓存池如何实现才能高效？
3. 如何动态指定服务的协议类型？
4. 如何实现Thrift、Grpc协议转换？

实现原理



我们还是先看一下hal-proxy内部有哪些模块，首先是Resolver模块，这个模块的是什么作用呢？这里我简单介绍一下，目前公司内部通过服务获取到机器列表的方式有多种，比如MIS平台、服务树等，也就是有的是通过平台配置的，有的是直接挂在服务树下，无论哪种方式，我们都通过服务名，通过一定的方式，找到该服务下面所有的主机。

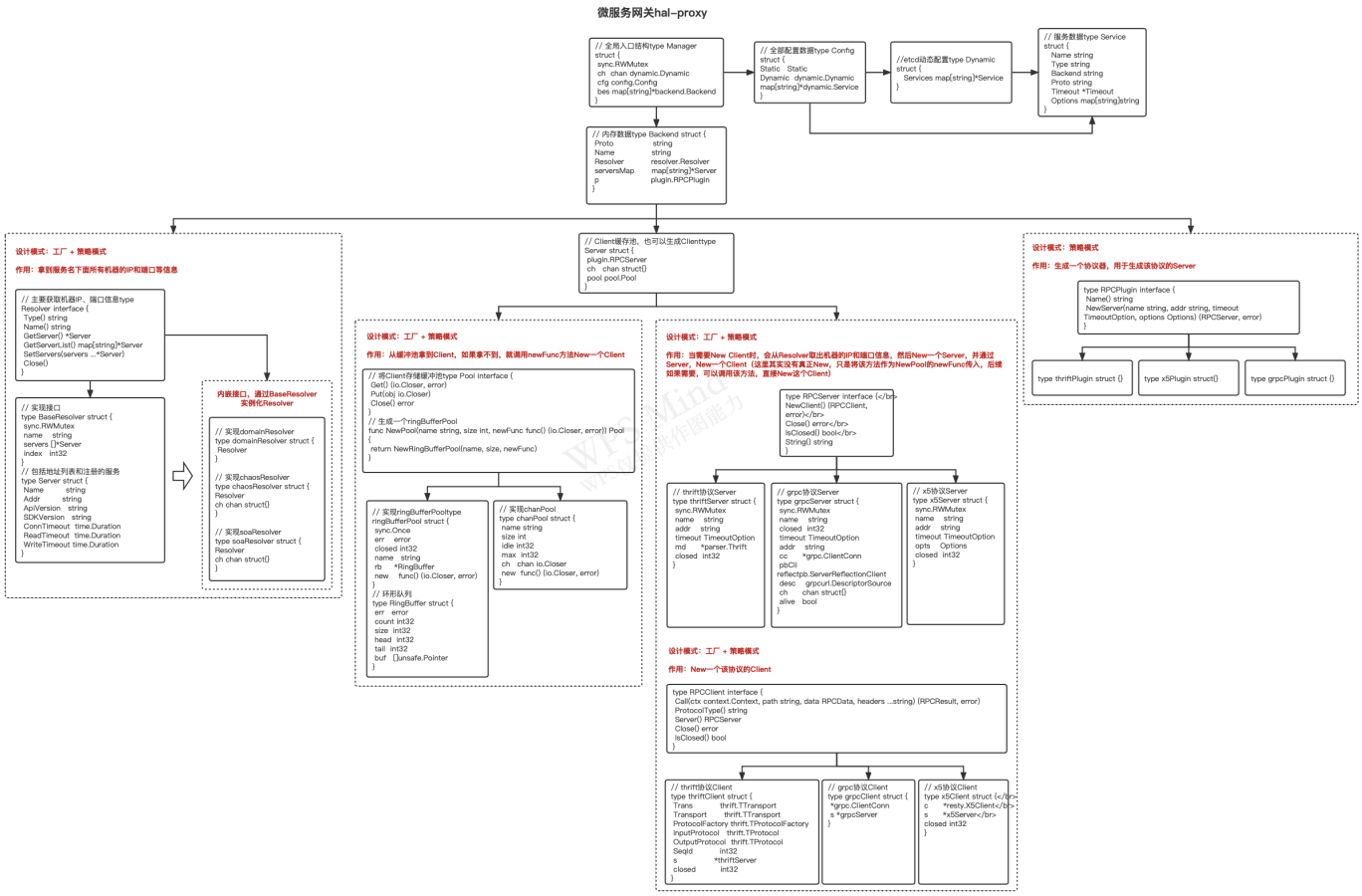
所以Resolver模块的作用，其实就是通过服务名，找到该服务下的所有机器的IP和服务端口，然后持久化到内存中，并定时更新。

协议模块就是支持不同的协议转换，每个协议类型的转换，都需要单独实现，**这些协议转换，无非就是先通过机器IP和端口初始化Client，然后再将数据进行转换后，直接发送到下游的机器。**

最后就是连接池，之前我们其实也用到go自带的pool来做，但是当对pool数据进行更新时，需要加锁，所以性能一直起不来，后来**改成了环形队列，然后对数据的操作全部通过原子操作方式，就实现了无锁操作，大大提高的并发性能。** 环形队列的代码，也给你安排上，可以直接看这篇文章 Go语言核心手册-10.原子操作。

实现逻辑

这个是hal-proxy的逻辑实现图，画了2天，包含所有核心对象的交互方式，这里就不去细讲，能掌握多少，靠大家自己领悟，如果有任何疑问(或者看不清图片)，可以关注我公众号，加我微信沟通。



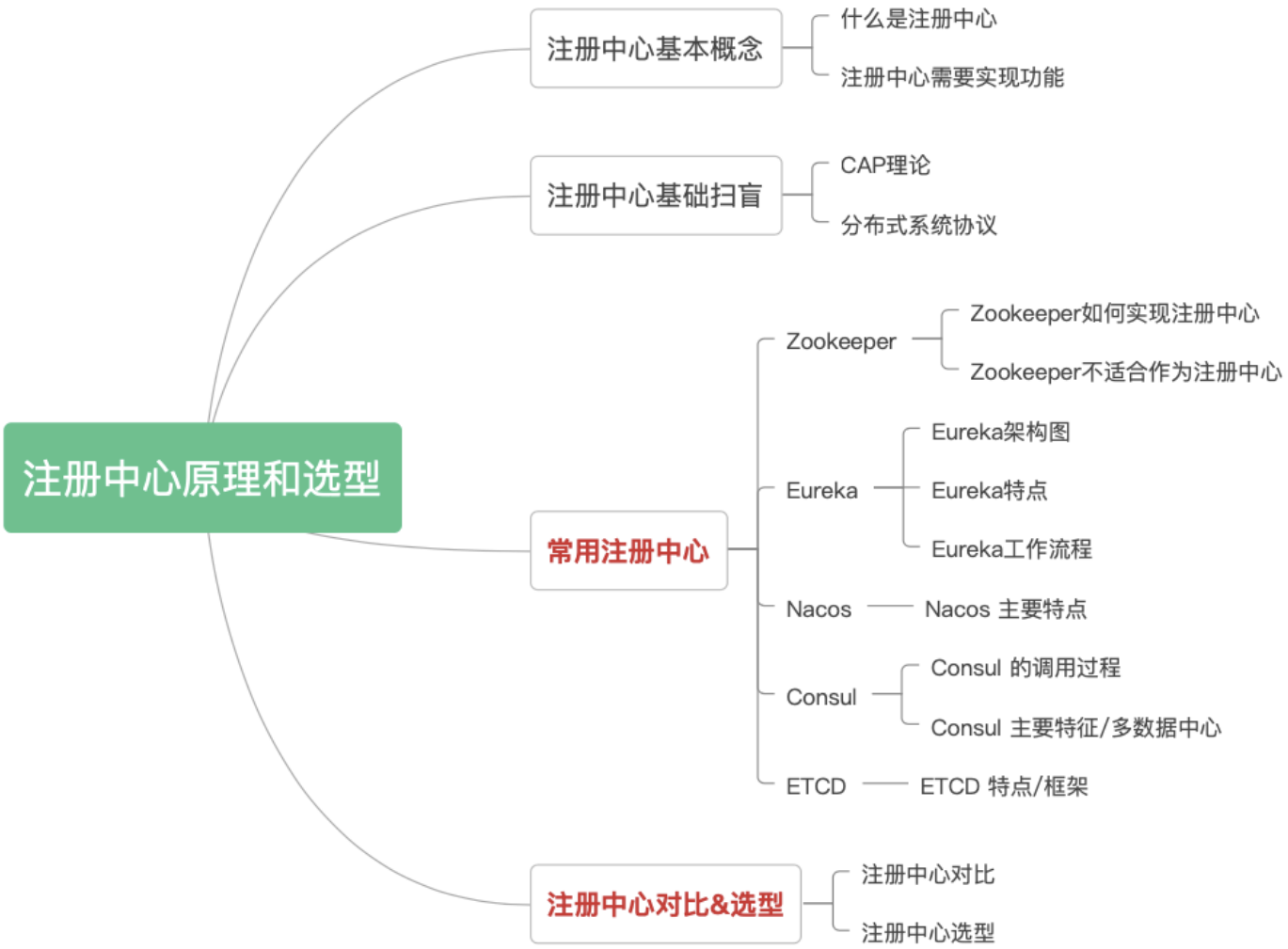
📚 第 3 章：注册中心

> 讲解5种常用的注册中心，对比其流程和原理，无论是面试还是技术选型，都非常有帮助。

大家好，我是楼仔！对于注册中心，在写这篇文章前，我其实只对ETCD有比较深入的了解，但是对于Zookeeper和其它的注册中心了解甚少，甚至都没有考虑过ETCD和Zookeeper是否适合作为注册中心。

经过近2周的学习，原来注册中心除了ETCD和Zookeeper，常用的还有Eureka、Nacos、Consul，下面我们就对这些常用的注册中心，初探它们的异同，便于后续技术选型。

全文接近 **8千字**，有点长，**建议先收藏，再慢慢看**，下面是文章目录：



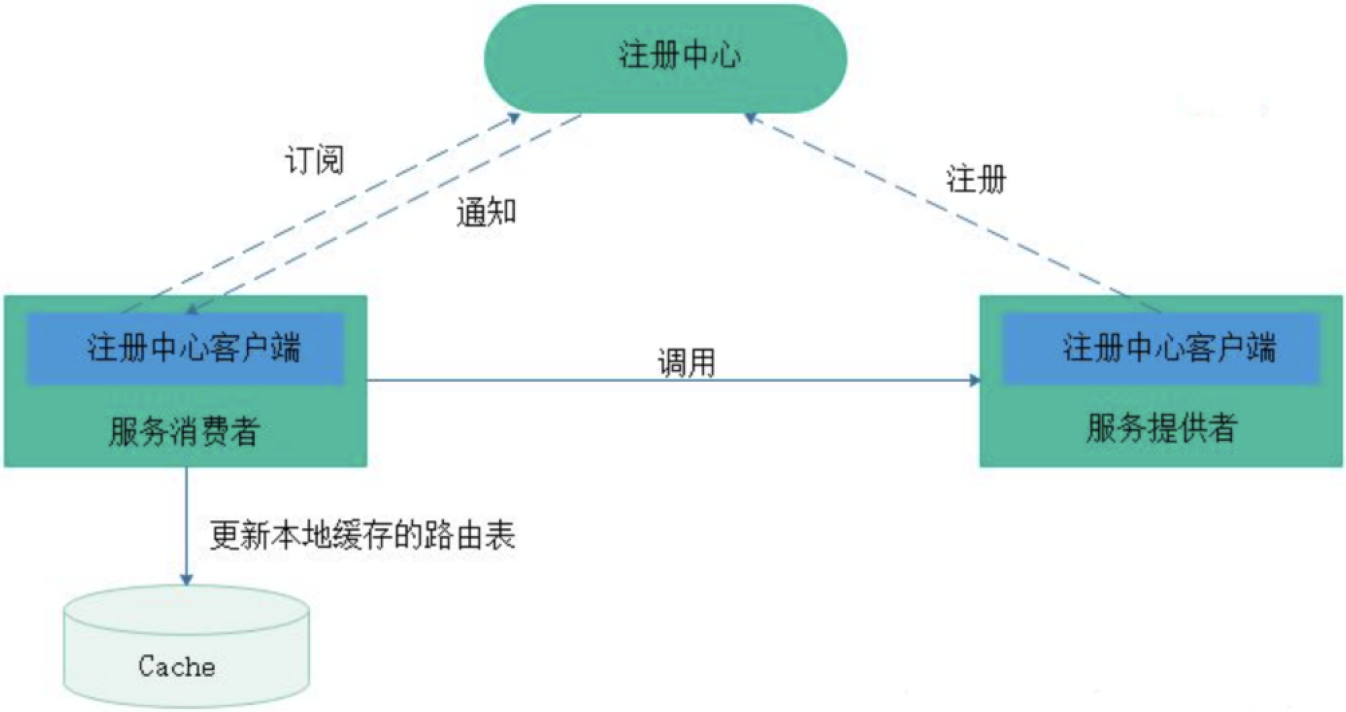
注册中心基本概念

什么是注册中心?

注册中心主要有三种角色：

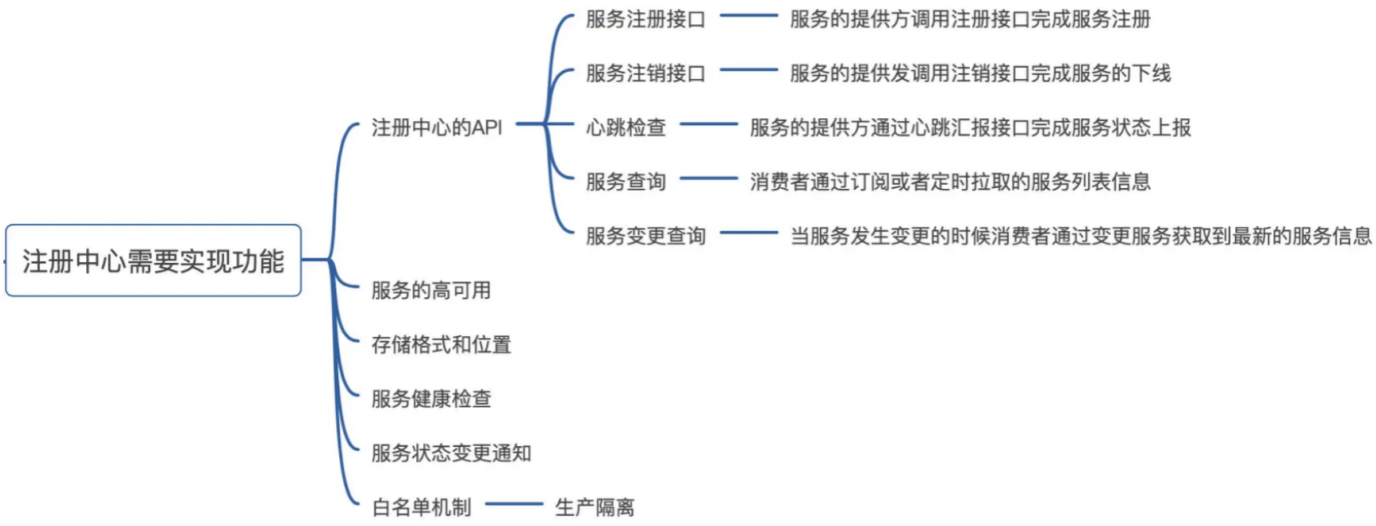
- **服务提供者（RPC Server）**：在启动时，向 Registry 注册自身服务，并向 Registry 定期发送心跳汇报存活状态。
- **服务消费者（RPC Client）**：在启动时，向 Registry 订阅服务，把 Registry 返回的服务节点列表缓存在本地内存中，并与 RPC Sever 建立连接。
- **服务注册中心（Registry）**：用于保存 RPC Server 的注册信息，当 RPC Server 节点发生变更时，Registry 会同步变更，RPC Client 感知后会刷新本地 内存中缓存的服务节点列表。

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 Sever 发起调用。



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：



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。

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个的原因如下：

- 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，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。

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作为他的服务发现框架。

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。这篇文章 《ETCD教程-2.Raft协议》 详细讲解了Raft原理，非常有意思，感兴趣的同学可以看看。

ZooKeeper Atomic Broadcast (ZAB, ZooKeeper原子消息广播协议)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。

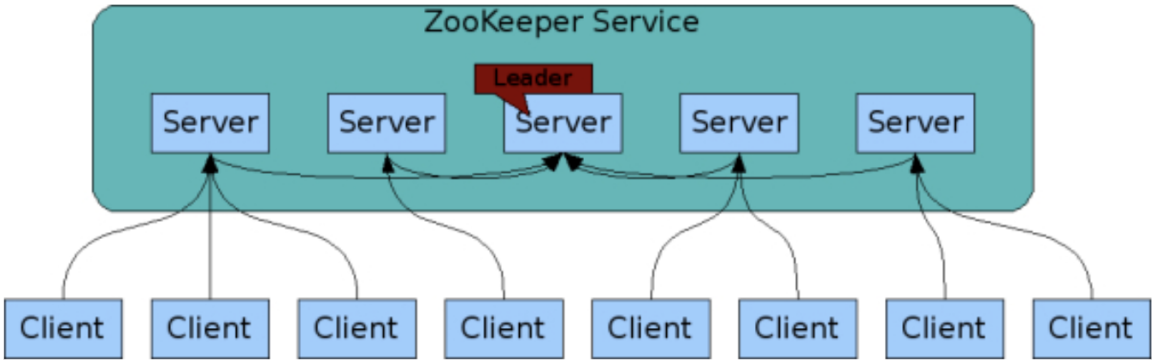
常用注册中心

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

Zookeeper

这个说起来有点意思的是官方并没有说他是一个注册中心，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。

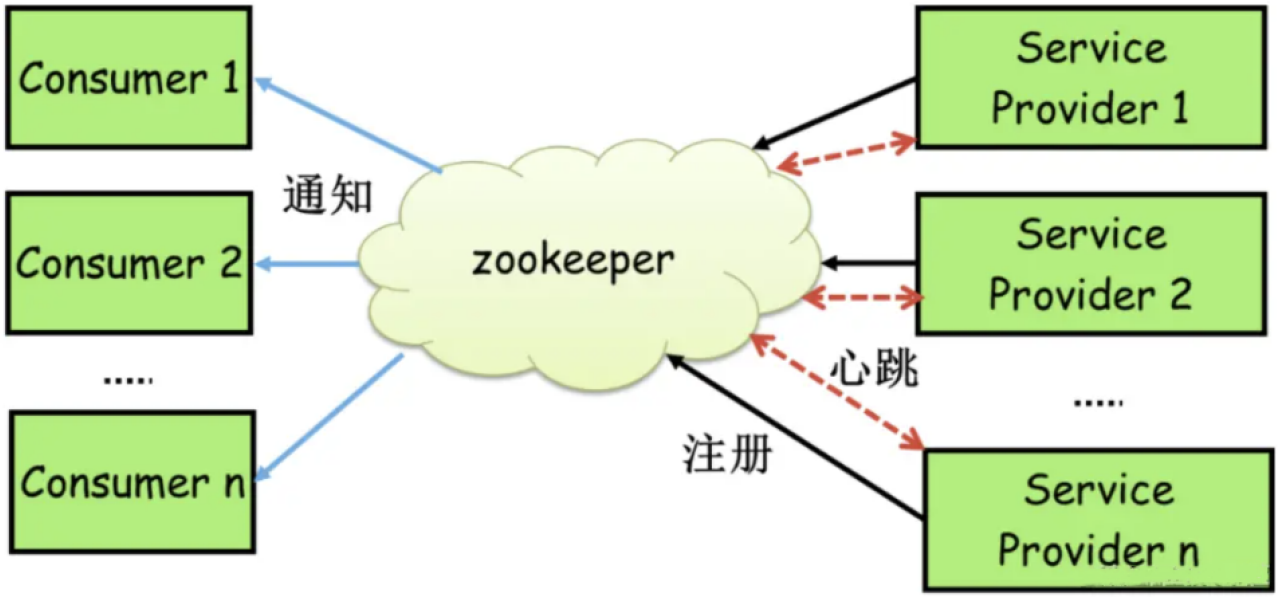
当然这有很多历史原因，这里我们就不追溯了。ZooKeeper是非常经典的服务注册中心中间件，在国内环境下，由于受到Dubbo框架的影响，大部分情况下认为Zookeeper是RPC服务框架下注册中心最好选择，随着Dubbo框架的不断开发优化，和各种注册中心组件的诞生，即使是RPC框架，现在的注册中心也逐步放弃了ZooKeeper。在常用的开发集群环境中，ZooKeeper依然起到十分重要的作用，Java体系中，大部分的集群环境都是依赖ZooKeeper管理服务的各个节点。



Zookeeper如何实现注册中心

> 具体可参考这篇文章 《Zookeeper用作注册中心的原理》，下面的内容都出自该文章。

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：

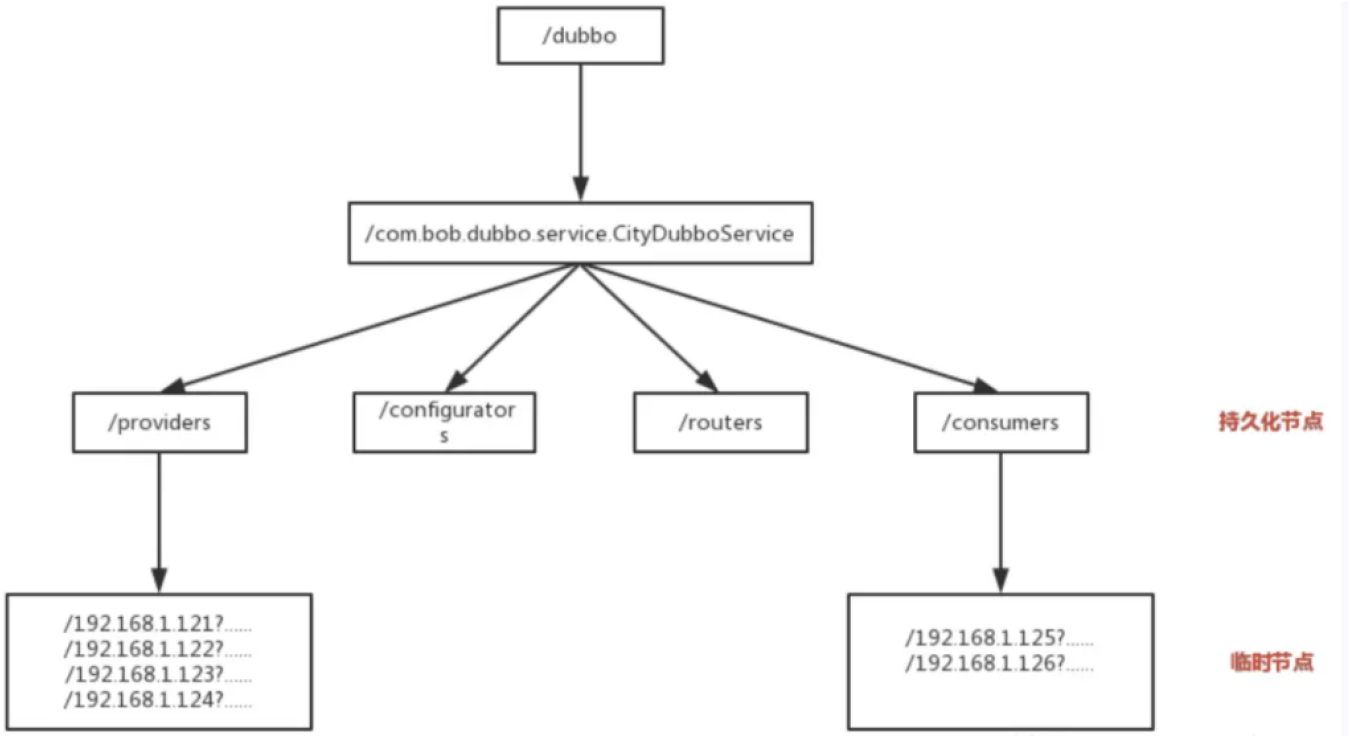


每当一个服务提供者部署后都要将自己的服务注册到zookeeper的某一路径上: /{service}/{version}/{ip:port} 。

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：

- /HelloWorldService/1.0.0/100.19.20.01:16888
- /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/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down掉不可用。**

但是zk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，30 ~ 120s, 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。在云部署的环境下，因网络问题使得zk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。

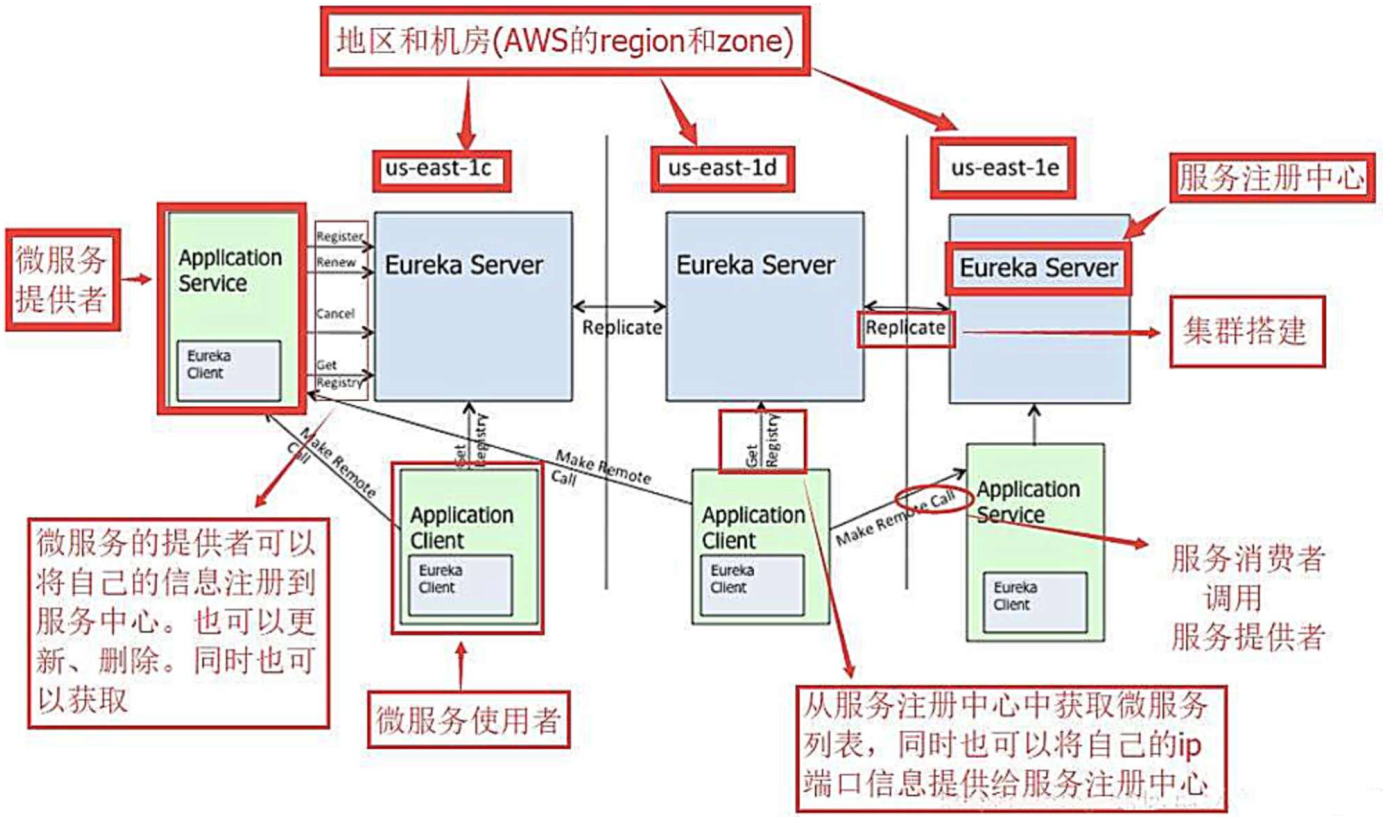
所以说，**作为注册中心，可用性的要求要高于一致性！**

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。**

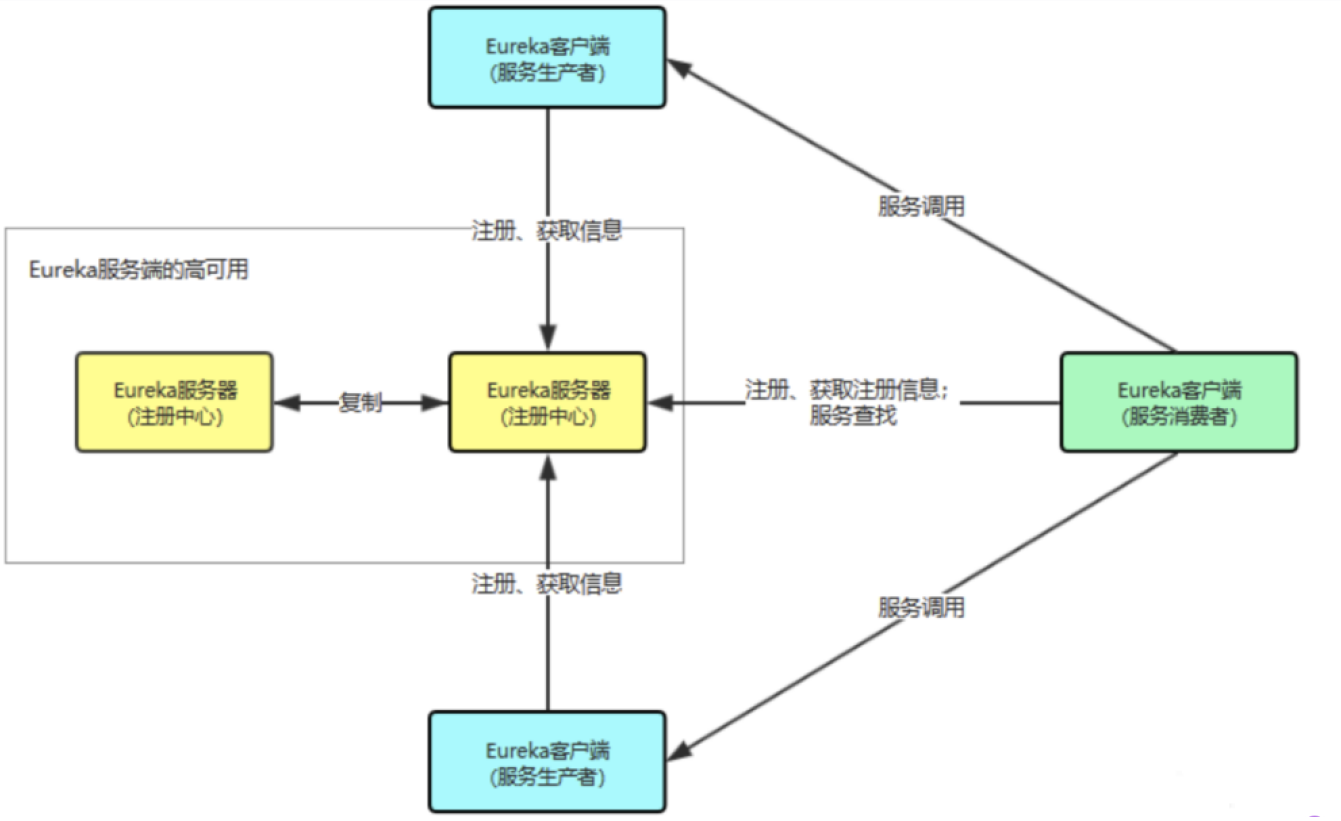
那为什么Zookeeper不使用最终一致性（AP）模型呢？因为这个依赖**Zookeeper的核心算法是ZAB，所有设计都是为了强一致性**。这个对于分布式协调系统，完全没没有毛病，但是**你如果将Zookeeper为分布式协调服务所做的一致性保障，用在注册中心，或者说服务发现场景，这个其实就不合适。**

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 在设计时就紧遵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&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

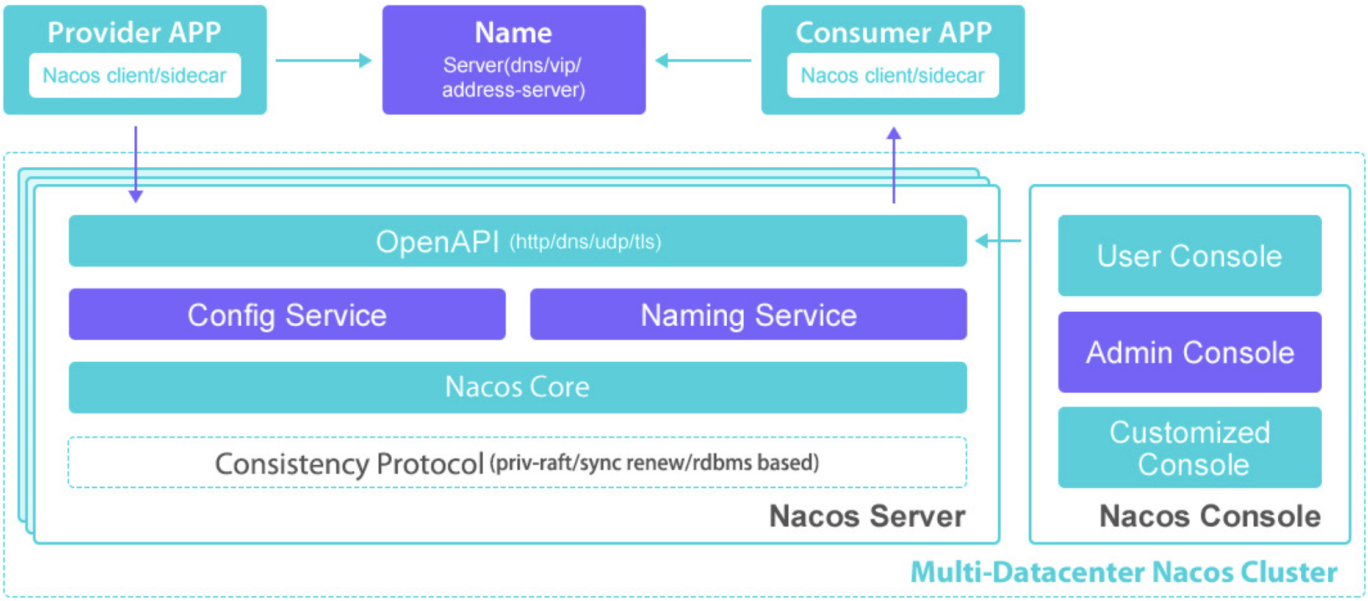
了解完 Eureka 核心概念，自我保护机制，以及集群内的工作原理后，我们来整体梳理一下 Eureka 的工作流程：

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 30s 向 Eureka Server 发送一次心跳请求，证明客户端服务正常。
4. 当 Eureka Server 90s 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。
10. 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。**

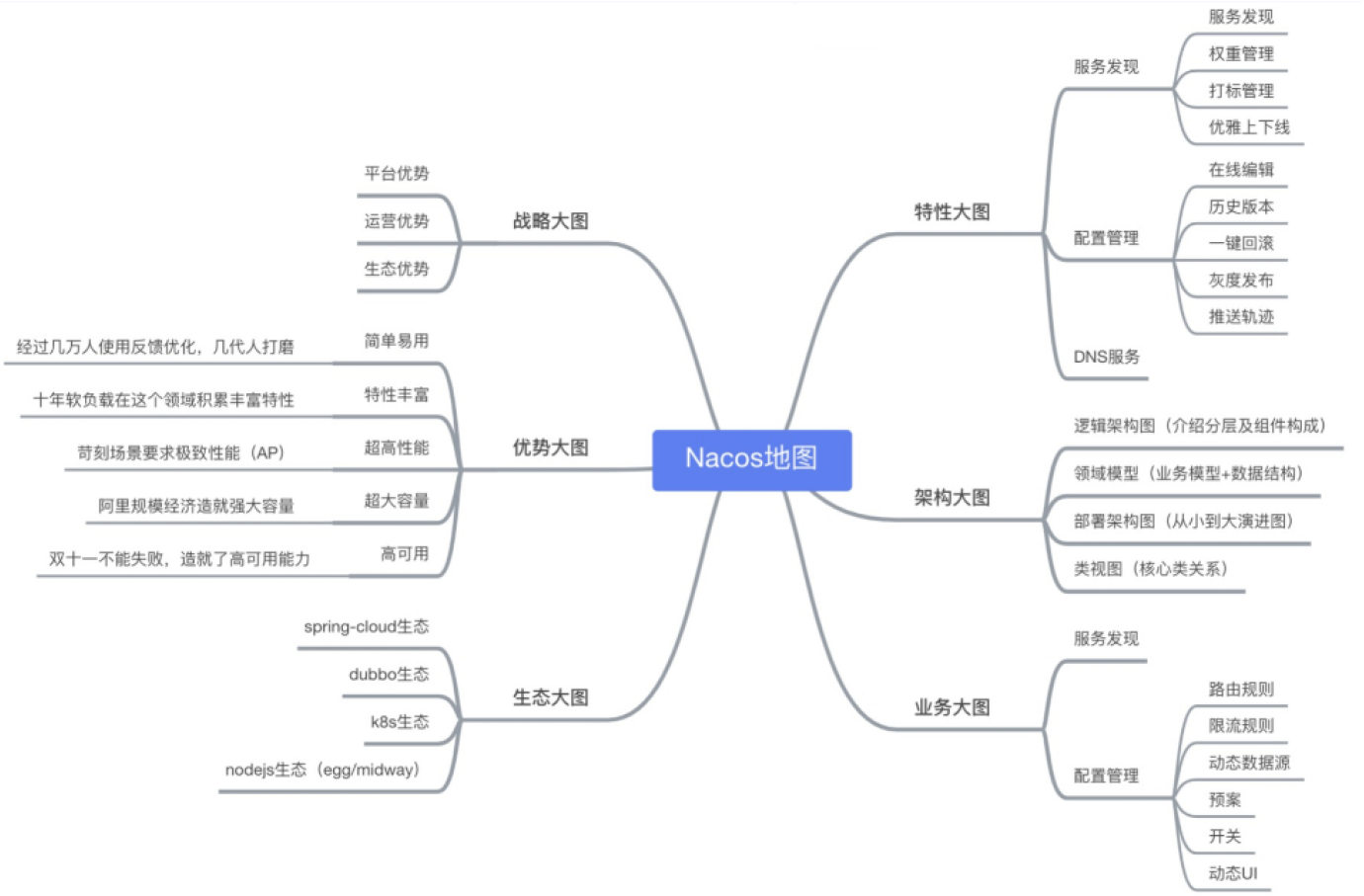
Nacos

> 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 TODO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2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 APIs TODO 帮助您管理服务的关联域名和可用的 IP:PORT 列表。

> 小节一下：

> - 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**，对他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，只要你想要的他就有。
> - **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**。

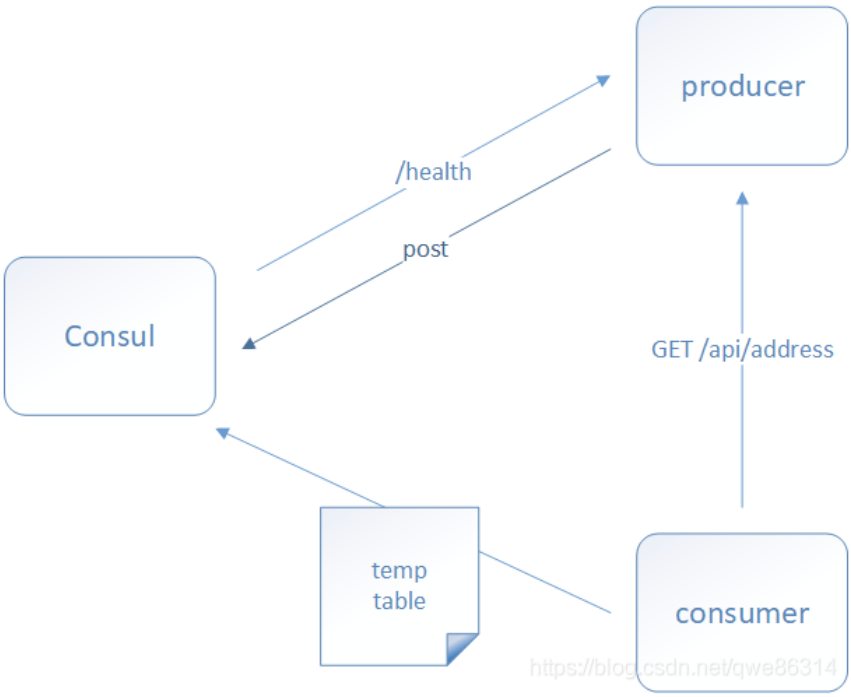
Consul

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 10s（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

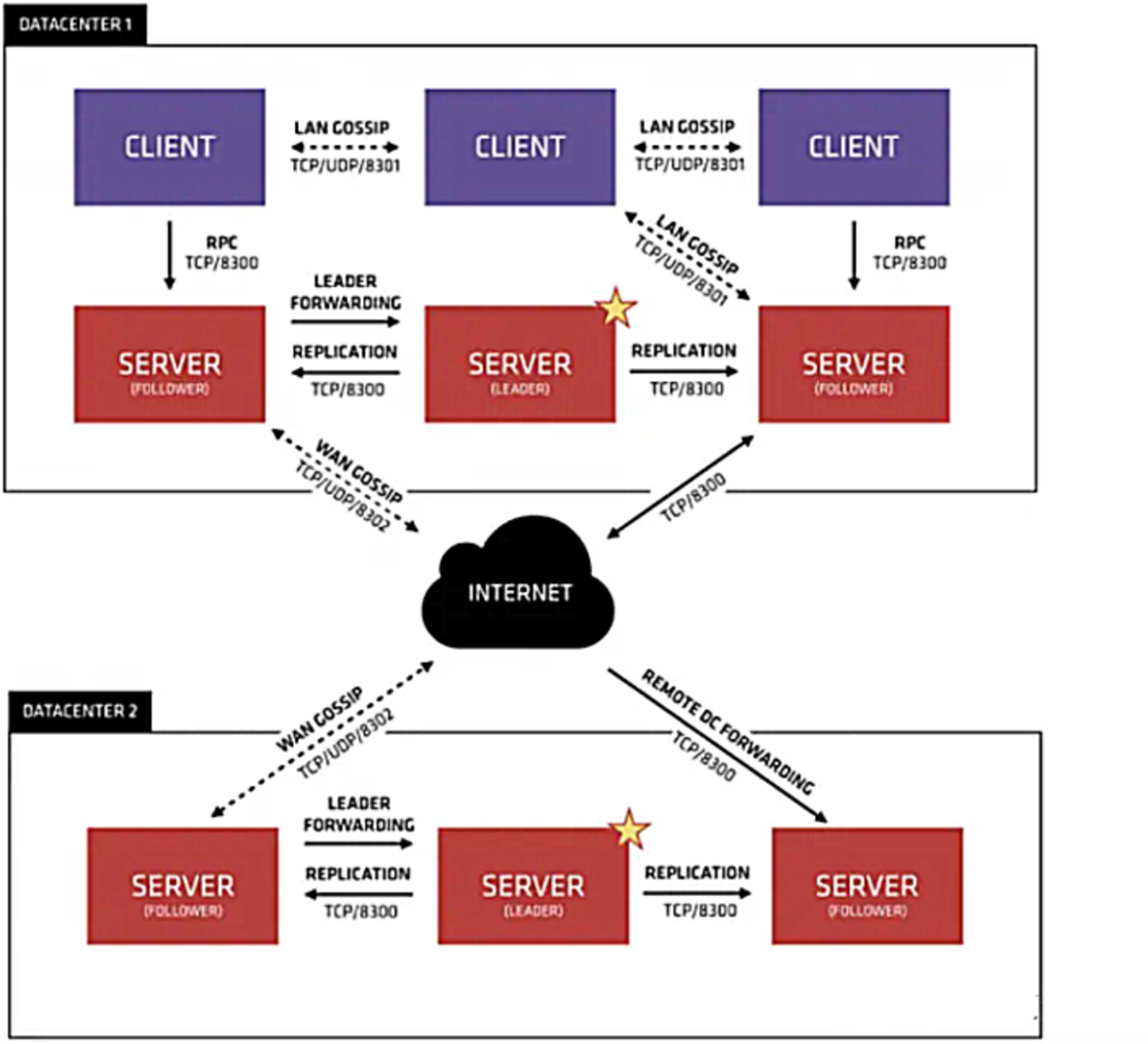
- CP模型，使用 Raft 算法来保证强一致性，不保证可用性；
- 支持服务注册与发现、健康检查、KV Store功能。
- 支持多数据中心，可以避免单数据中心的单点故障，而其部署则需要考虑网络延迟, 分片等情况等。
- 支持安全服务通信，Consul可以为服务生成和分发TLS证书，以建立相互的TLS连接。
- 支持 http 和 dns 协议接口；

- 官方提供 web 管理界面。

多数据中心

> 这里纯属了解，学习一下 Consul 的多数据中心是如何实现的。

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gossip协议（流言协议）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。

ETCD

etcd是一个Go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。

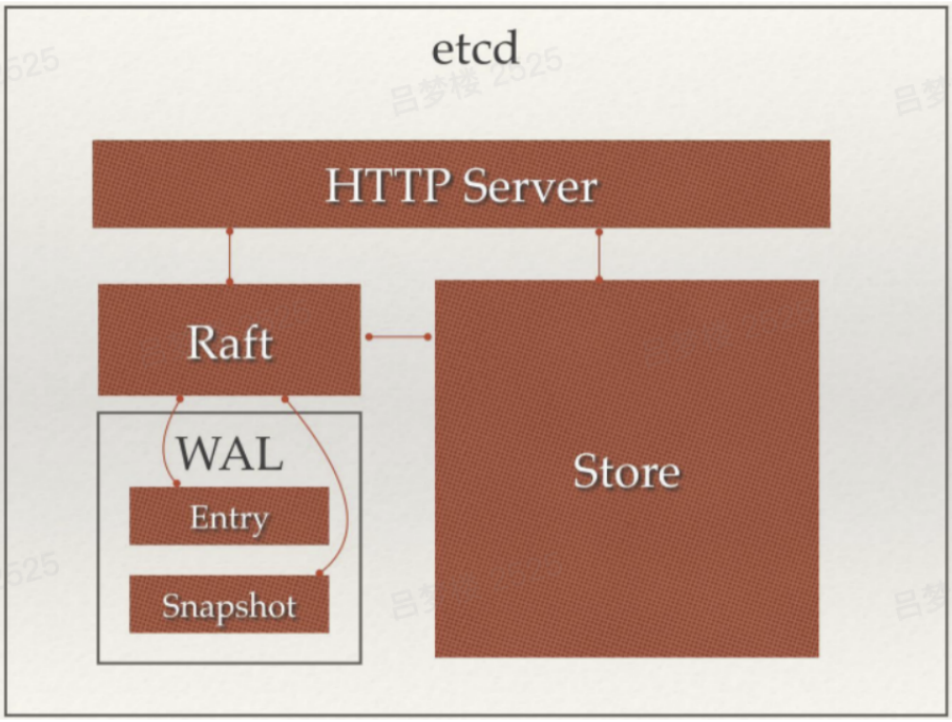
ETCD 特点

- 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。



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。

更多关于ETCD相关知识，可以查看该文章 《肝了一个月的ETCD，从Raft原理到实践》

注册中心对比&选型

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**：Euraka 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，也能够较好的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 类型牺牲可用性，在服务发现场景并没太大优势；
- **Watch的支持**：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的集成**：目前都有相对应的 boot starter，提供了集成能力。

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：

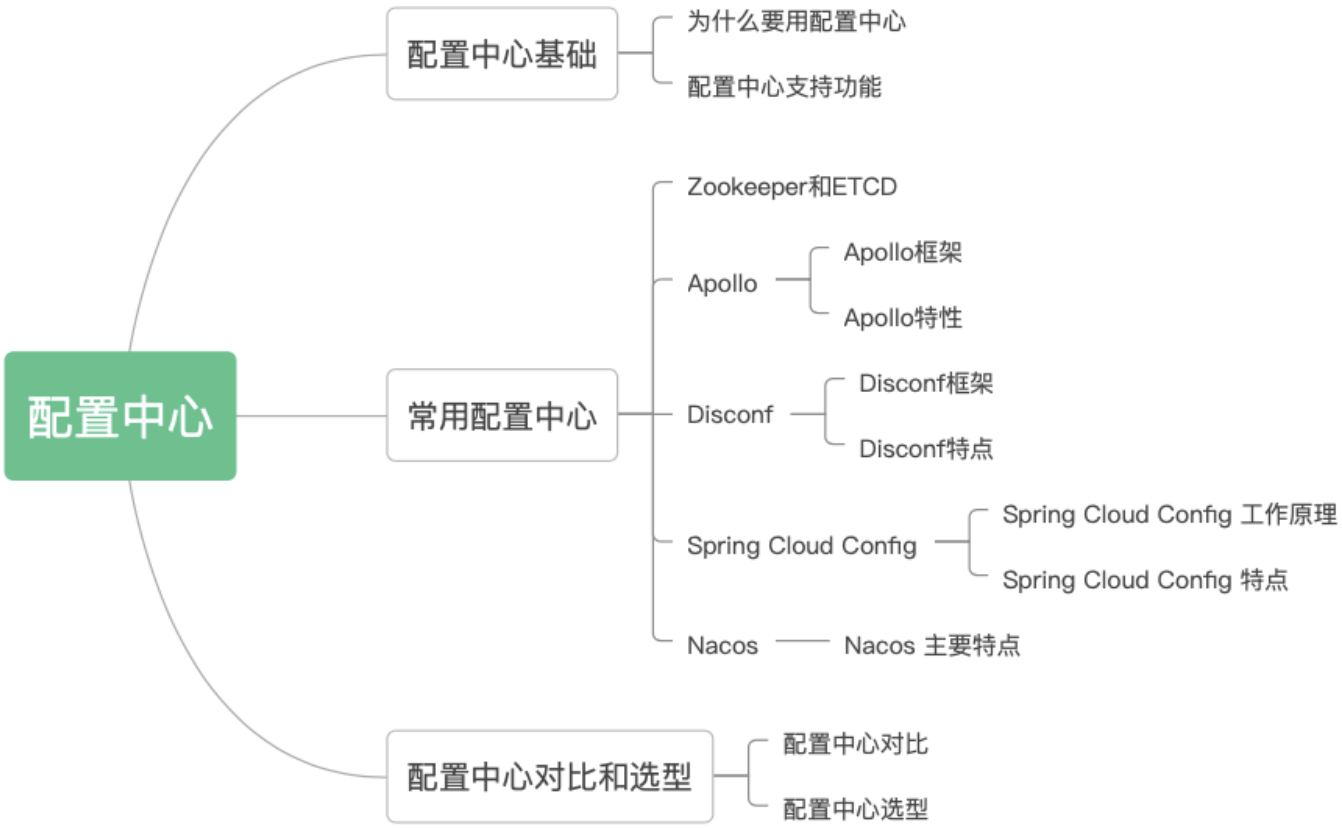
- **关于CP还是AP的选择**：选择 AP，因为可用性高于一致性，所以更倾向 Eureka 和 Nacos；关于Eureka、Nacos如何选择，哪个让我做的事少，我就选择哪个，显然 Nacos 帮我们做了更多的事。
- **技术体系**：Etcd 和 Consul 都是Go开发的，Eureka、Nacos、Zookeeper 和 Zookeeper 都是Java开发的，可能项目属于不同的技术栈，会偏向选择对应的技术体系。
- **高可用**：这几款开源产品都已经考虑如何搭建高可用集群，有些差别而已；
- **产品的活跃度**：这几款开源产品整体上都比较活跃。

📚 第 4 章：配置中心

> 讲解4种常用的配置中心，对比其流程和原理，无论是面试还是技术选型，都非常有帮助。

大家好，我是楼仔！这是我写的第5篇关于"技术选型"相关的文章，前4篇分别为**消息队列、注册中心、微服务网关、RPC框架**，这篇配置中心应该是该系列的最后一篇，前后跨度近一年，应该是我跨度时间最长的一个系列了。

学完注册中心，再看配置中心这块，感觉简单很多，因为很多知识原理是相辅相成的，下面是文章目录：



配置中心基础

为什么要用配置中心?

- **配置实时生效**：传统的静态配置方式要想修改某个配置只能修改之后重新发布应用，要实现动态性，可以选择使用数据库，通过定时轮询访问数据库来感知配置的变化。轮询频率低感知配置变化的延时就长，轮询频率高，感知配置变化的延时就短，但比较损耗性能，需要在实时性和性能之间做折中。**配置中心专门针对这个业务场景，兼顾实时性和一致性来管理动态配置**；
- **配置管理流程**：配置的权限管控、灰度发布、版本管理、格式检验和安全配置等一系列的配置管理相关的特性，也是配置中心不可获取的一部分；
- **分布式场景**：随着采用分布式的开发模式，项目之间的相互引用随着服务的不断增多，相互之间的调用复杂度成指数升高，每次投产或者上线新的项目时苦不堪言，需要引用配置中心治理。

配置中心支持功能

- **灰度发布**：配置的灰度发布是配置中心比较重要的功能，当配置的变更影响比较大的时候，需要先在部分应用实例中验证配置的变更是否符合预期，然后再推送到所有应用实例。
- **权限管理**：配置的变更和代码变更都是对应用运行逻辑的改变，重要的配置变更常常会带来核弹的效果，对于配置变更的权限管控和审计能力同样是配置中心重要的功能。
- **版本管理&回滚**：当配置变更不符合预期的时候，需要根据配置的发布版本进行回滚。
- **配置格式校验**：应用的配置数据存储在配置中心一般都会以一种配置格式存储，比如Properties、Json、Yaml等，如果配置格式错误，会导致客户端解析配置失败引起生产故障，配置中心对配置的格式校验能够有效防止人为错误操作的发生，是配置中心核心功能中的刚需。
- **监听查询**：当排查问题或者进行统计的时候，需要知道一个配置被哪些应用实例使用到，以及一个实例使用到了哪些配置。

- **多环境**：在实际生产中，配置中心常常需要涉及多环境或者多集群，业务在开发的时候可以将开发环境和生产环境分开，或者根据不同的业务线存在多个生产环境。如果各个环境之间的相互影响比较小（开发环境影响到生产环境稳定性），配置中心可以通过逻辑隔离的方式支持多环境。
- **多集群**：当对稳定性要求比较高，不允许各个环境相互影响的时候，需要将多个环境通过多集群的方式进行物理隔离。

常用配置中心

写在前面

如果只要能作为分布式存储的服务都作为配置中心，那选择途径就太多了， 比如Zookeeper和ETCD，所以需要提前说明一下。

在文章 《注册中心原理和选型：Zookeeper、Eureka、Nacos、Consul和ETCD》 已经提到过，Zookeeper和ETCD可以存储数据，作为配置中心使用，比如我司的微服务网关，将配置发布到ETCD，供网关各模块调用，具体可以参考文章 《微服务网关：从对比到选型，由理论到实践》 。

但是我们选择配置中心时，为什么不优先考虑Zookeeper和ETCD，因为以下两点原因：

- 没有方便的UI管理工具，且缺乏权限、审核、灰度发布、审核机制等；
- 最重要的是，Zookeeper和ETCD通常定义为服务注册中心，统一配置中心的事情交给专业的工具去解决。

大白话总结一下，就是**专业的人干专业的事，他两很多功能没法支持**。可能你会问，那你们公司为啥用ETCD作为配置中心呢？因为我们自己写了个后台，支持权限、灰度发布、版本控制等功能。

所以给大家介绍的配置中心，主要是以下4种，分别为 **Disconf、Spring Cloud Config、Apollo 和 Nacos**。

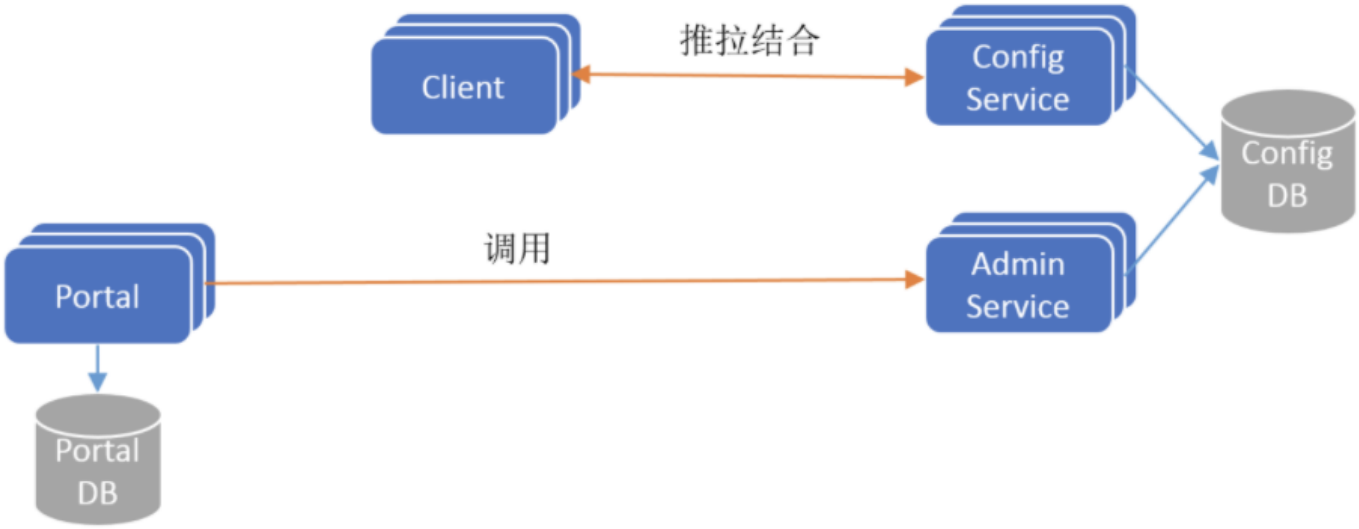
Apollo

> GitHub：https://github.com/apolloconfig/apollo

Apollo（阿波罗）是携程框架部门研发的开源配置管理中心，具备规范的权限、流程治理等特性。

Apollo框架

Apollo的框架有点复杂，如果不考虑分布式微服务架构中的服务发现问题，Apollo的最简架构如下图所示：



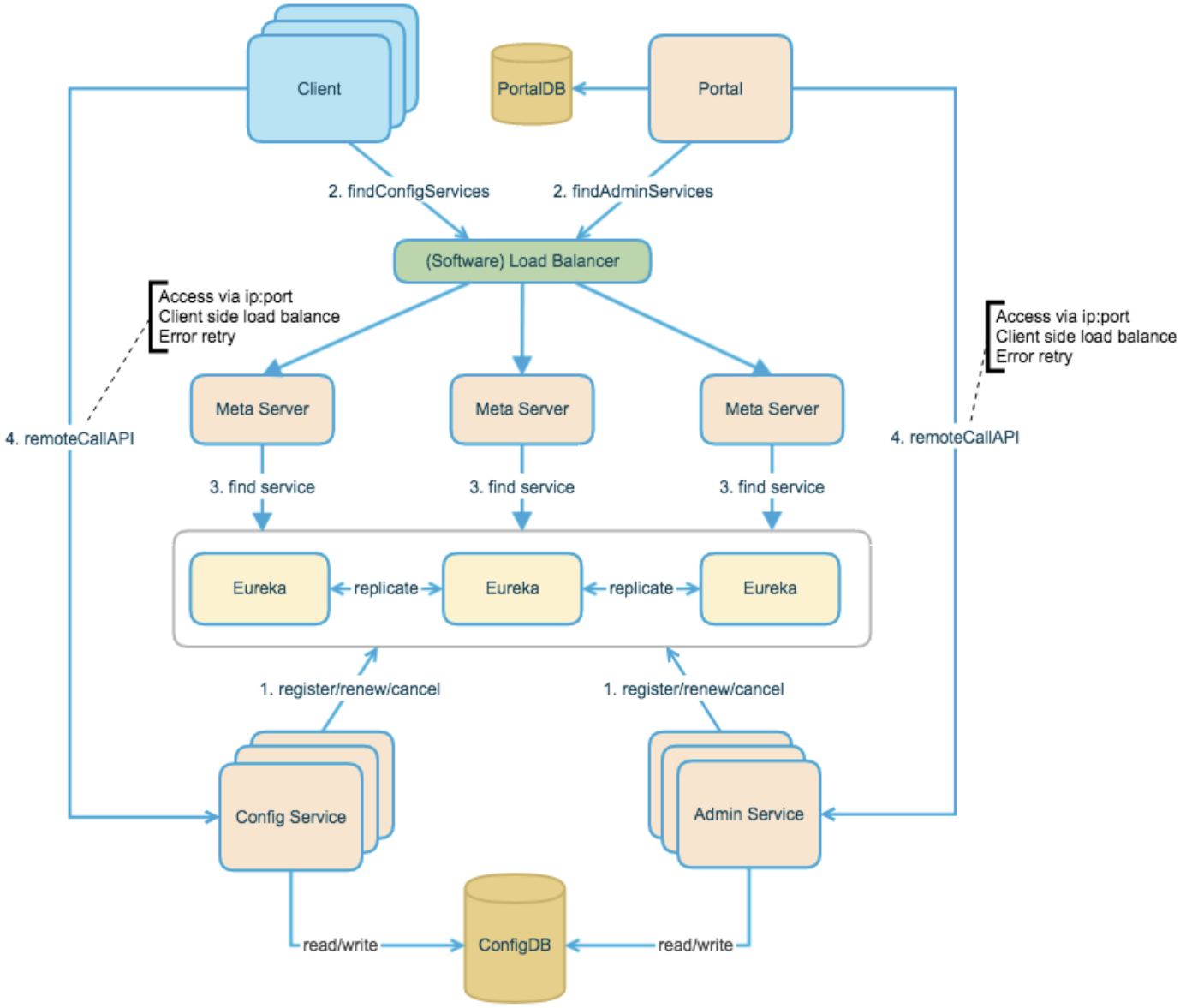
这里面包含Apollo框架的**4个核心模块**：

- **ConfigService**：提供配置获取接口，提供配置推送接口，服务于Apollo客户端。
- **AdminService**：提供配置管理接口，提供配置修改发布接口，服务于管理界面Portal。
- **Client**：为应用获取配置，支持实时更新，通过MetaServer获取ConfigService的服务列表，使用客户端软负载SLB方式调用ConfigService。
- **Portal**：配置管理界面，通过MetaServer获取AdminService的服务列表，使用客户端软负载SLB方式调用AdminService。

调用流程：

1. ConfigService是一个独立的微服务，服务于Client进行配置获取。
2. Client和ConfigService保持长连接，通过一种拖拉结合(push & pull)的模式，实现配置实时更新的同时，保证配置更新不丢失。
3. AdminService是一个独立的微服务，服务于Portal进行配置管理。Portal通过调用AdminService进行配置管理和发布。
4. ConfigService和AdminService共享ConfigDB，ConfigDB中存放项目在某个环境的配置信息。ConfigService/AdminService/ConfigDB三者在每个环境(DEV/FAT/UAT/PRO)中都要部署一份。
5. Protal有一个独立的PortalDB，存放用户权限、项目和配置的元数据信息。Protal只需部署一份，它可以管理多套环境。

加上分布式微服务架构中的服务发现，真正的Apollo框架如下：



如果你了解RPC和注册中心，这幅图其实不难理解：

- Eureka用于注册中心，AP原则，所以Config Service和Admin Service的机器列表注册到Eureka中；
- Client和Portal需要获取注册中心的机器列表，但是由于Eureka仅支持Java客户端，所以搞个Meta Server，将Eureka的服务发现接口以HTTP接口的形式暴露出来；
- 由于Meta Server是集群部署，需要搞个NginxLB去找Meta Server机器。

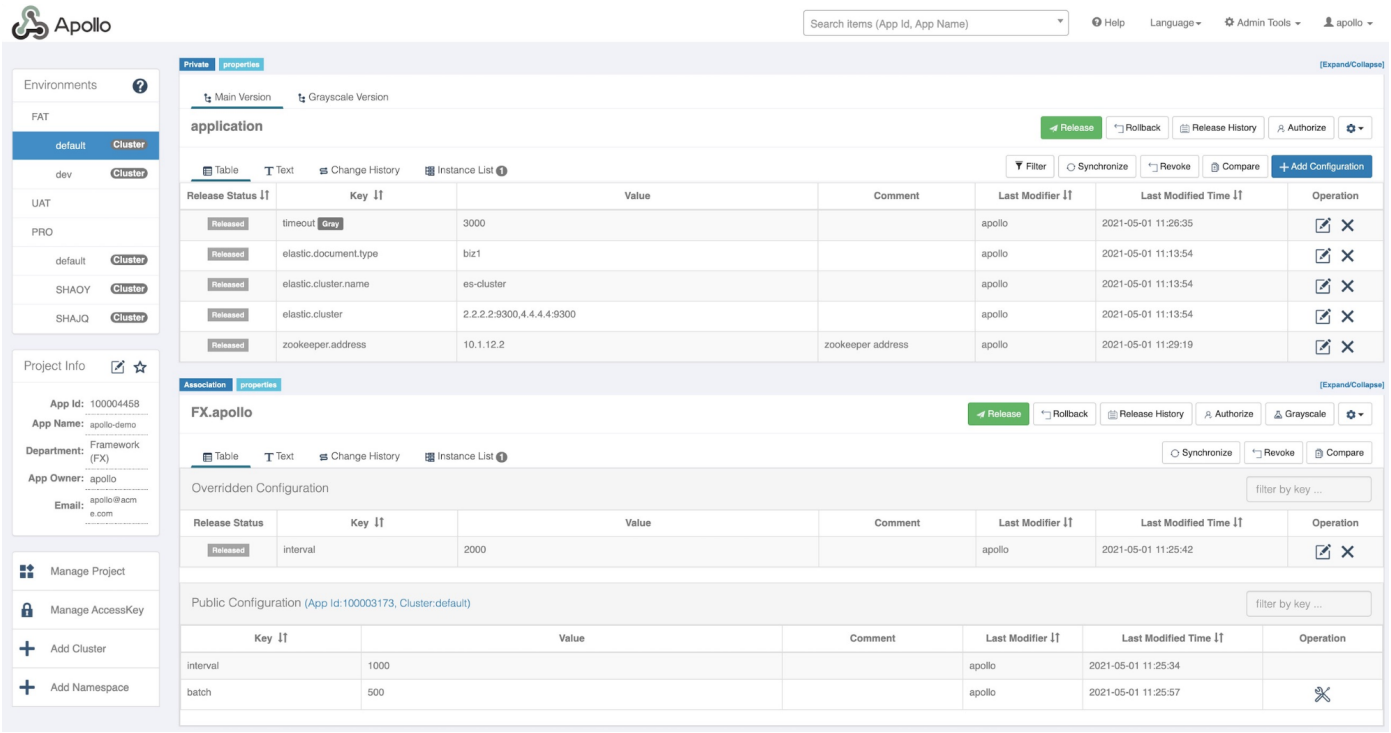
所以搞NginxLB + Meta Server，其实就是为了找Eureka中的机器列表配置，Client和Portal拿到这些机器配置，就可以发起调用了，最后就回到我们前面的简图，是不是So Easy!

我讲的已经够清楚了，如果还是不懂，就看这篇文章 《微服务架构~携程Apollo配置中心架构剖析》

Apollo特性

- **统一管理不同环境、不同集群的配置**：
  - Apollo提供了一个统一界面集中式管理不同环境（environment）、不同集群（cluster）、不同命名空间（namespace）的配置。
  - 同一份代码部署在不同的集群，可以有不同的配置，比如zk的地址等。
  - 通过命名空间（namespace）可以很方便的支持多个不同应用共享同一份配置，同时还允许应用对共享的配置进行覆盖。
- **配置修改实时生效（热发布）**：用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最新的配置，并通知到应用程序。
- **版本发布管理 + 灰度发布**
- **权限管理、发布审核、操作审计**：应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。所有的操作都有审计日志，可以方便的追踪问题。
- **客户端配置信息监控**：可以在界面上方便地看到配置在被哪些实例使用。
- **提供Java和.Net原生客户端**：
  - 提供了Java和.Net的原生客户端，方便应用集成。
  - 支持Spring Placeholder、Annotation和Spring Boot的ConfigurationProperties，方便应用使用。
  - 提供了Http接口，非Java和.Net应用也可以方便的使用。
- **提供开放平台API**：
  - Apollo自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。
  - Apollo出于通用性考虑，对配置的修改不会做过多限制，只要符合基本的格式就能够保存。
  - 对于有些使用方，它们的配置可能会有比较复杂的格式，而且对输入的值也需要进行校验后方可保存，如检查数据库、用户名和密码是否匹配。对于这类应用，Apollo支持应用方通过开放接口在Apollo进行配置的修改和发布，并且具备完善的授权和权限控制。

最后通过后台界面，直观感受一下：



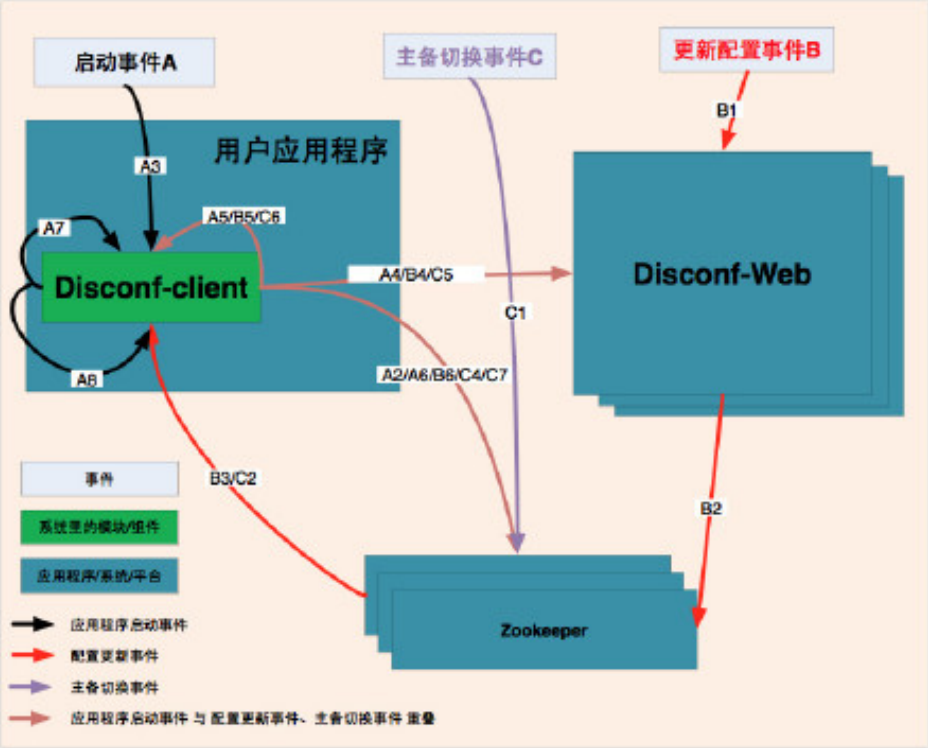
Disconf

> GitHub：https://github.com/knightliao/disconf

2014年7月百度开源的配置管理中心，同样具备配置的管理能力，不过目前已经不维护，最近的一次提交是两年前了。

Disconf框架

Disconf是一套完整的基于zookeeper的分布式配置统一解决方案，它通过disconf-web管理配置信息，然后将配置的key在Zookeeper上建立节点，disconf-client启动后拉取自身需要的配置信息并监听Zookeeper的节点。在web上更新配置信息会触发zk节点状态的变动，client可以实时感知到变化，然后从web上拉取最新配置信息。



> 这里想吐槽一下，Disconf官方文档的图画真的丑啊，关键是很不清晰，就不能好好维护一下么？

Disconf特点

- 支持配置（配置项+配置文件）的分布式化管理：
  - 配置发布统一化
  - 配置发布、更新统一化（云端存储、发布）：配置存储在云端系统，用户统一在平台上进行发布、更新配置。
  - 配置更新自动化：用户在平台更新配置，使用该配置的系统会自动发现该情况，并应用新配置。特殊地，如果用户为此配置定义了回调函数类，则此函数类会被自动调用。
- 配置异构系统管理：
  - 异构包部署统一化：这里的异构系统是指一个系统部署多个实例时，由于配置不同，从而需要多个部署包（jar或war）的情况（下同）。使用Disconf后，异构系统的部署只需要一个部署包，不同实例的配置会自动分配。特别地，在业界大量使用部署虚拟化（如JPAAS系统，SAE，BAE）的情况下，同一个系统使用同一个部署包的情景会越来越多，Disconf可以很自然地与他天然契合。 异构主备自动切换：如果一个异构系统存在主备机，主机发生挂机时，备机可以自动获取主机配置从而变成主机。
  - 异构主备机Context共享工具：异构系统下，主备机切换时可能需要共享Context。可以使用Context共享工具来共享主备的Context。
- 注解式编程，极简的使用方式：我们追求的是极简的、用户编程体验良好的编程方式。通过简单的标注+极简单的代码撰写，即可完成复杂的配置分布式化。
- 需要Spring编程环境。
- 可以托管任何类型的配置文件。
- 提供界面良好Web管理功能，可以非常方便的查看配置被哪些实例使用了。

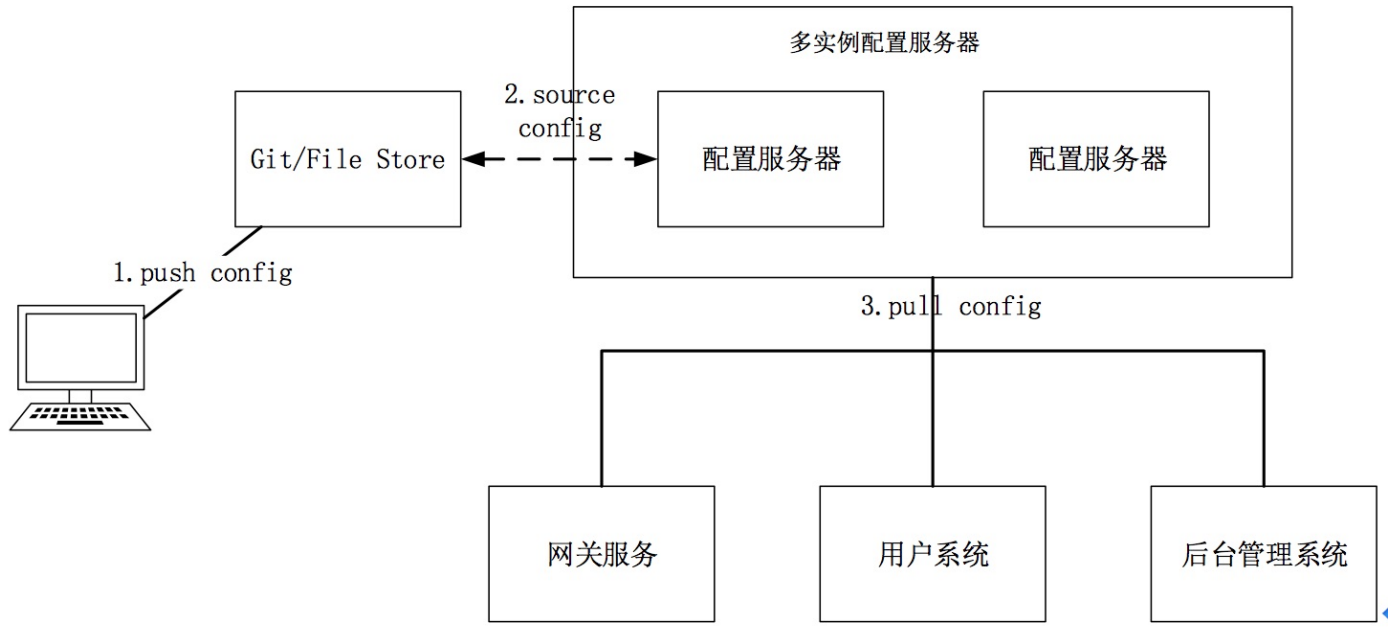
Spring Cloud Config

> GitHub：https://github.com/spring-cloud/spring-cloud-config

2014年9月开源，Spring Cloud 生态组件，可以和Spring Cloud体系无缝整合。

Spring Cloud Config 工作原理

应用架构图：



工作流程：

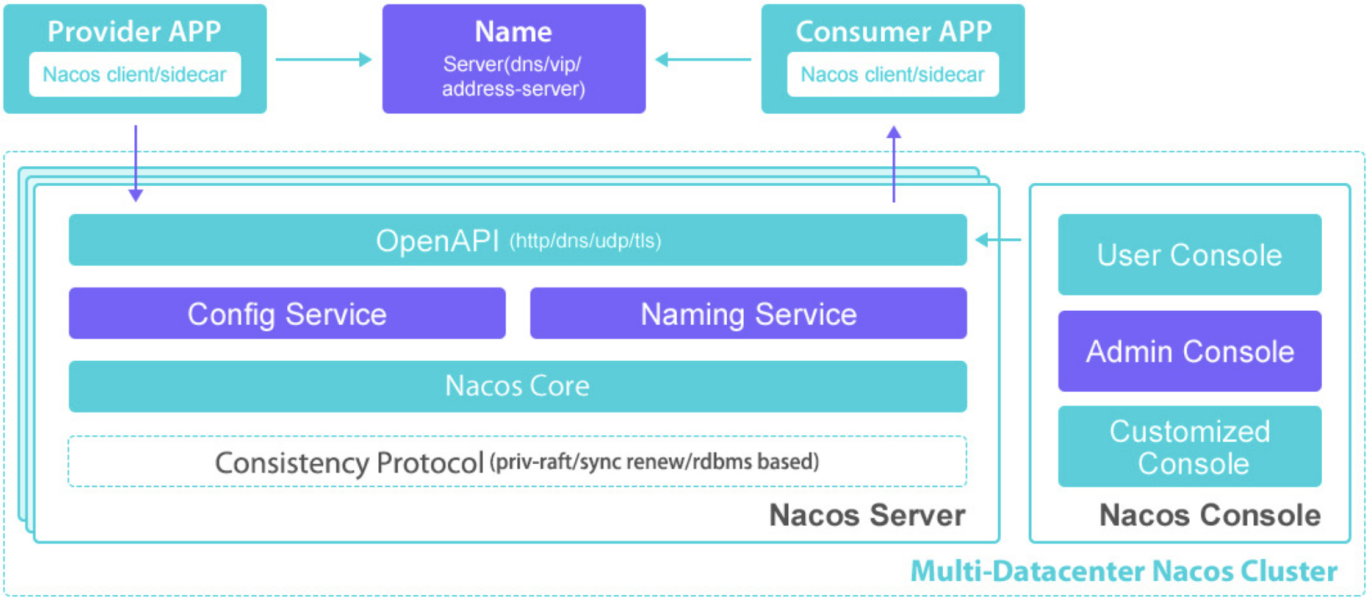
- 在部署环境之前，需要将相应的配置信息推送到配置仓库；
- 配置服务器启动之后，将配置信息拉取并同步至本地仓库；
- 配置服务器对外提供REST接口，其他所有的配置客户端启动时根据spring.cloud.config配置的{application}/{profile}/{label}信息去配置服务器拉取相应的配置。配置仓库支持多样的源，如Git、SVN、jdbc数据库和本地文件系统等。
- 其他应用启动，从配置服务器拉取配置。（配置中心还支持动态刷新配置信息，不需要重启应用，通过spring-cloud-config-monitor监控模块，其中包含了/monitor刷新API，webhook调用该端点API，达到动态刷新的效果。）

Spring Cloud Config 特点

- 提供配置的服务端和客户端支持；
- 集中式管理分布式环境下的应用配置；
- 基于 Spring 环境，可以无缝与Spring应用集成；
- 可用于任何语言开发的程序，为其管理与提供配置信息；
- 默认实现基于git仓库，可以进行版本管理。

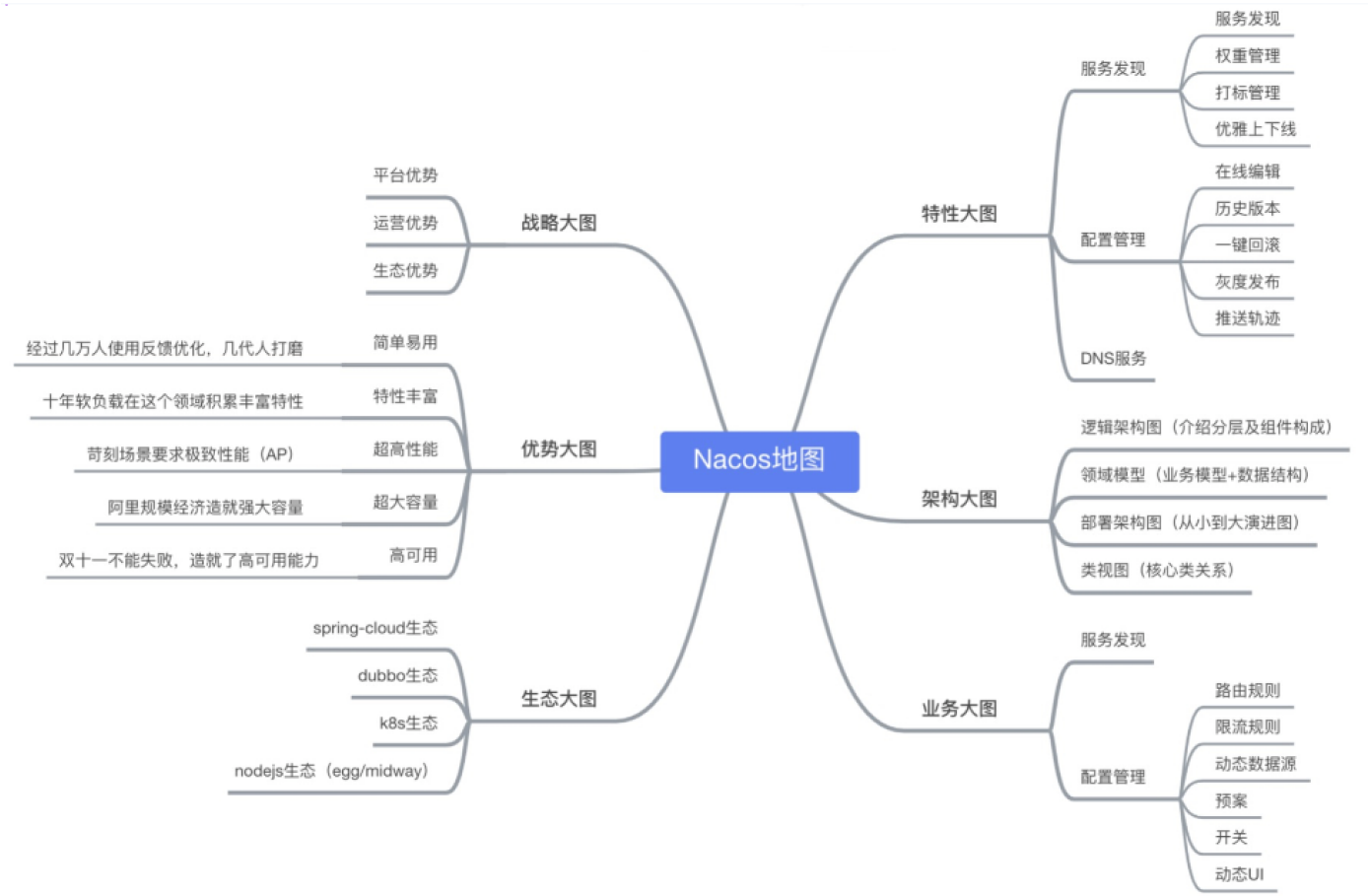
Nacos

> 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 TODO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2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 APIs TODO 帮助您管理服务的关联域名和可用的 IP:PORT 列表。

> 小节一下：
>
> - 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**，对他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，只要你想要的他就有。
> - **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**。

配置中心对比和选型

由于 Disconf 不再维护，下面对比一下 Spring Cloud Config、Apollo 和 Nacos。

配置中心对比

功能点	Spring Cloud Config	Apollo	Nacos
开源时间	2014.9	2016.5	2018.6
配置界面	不支持	支持	支持
配置实时推送	支持（Spring Cloud Bus）	支持（HTTP 长轮询 1S内）	支持（HTTP 长轮询 1S内）
版本管理	支持（Git）	支持	支持
配置回滚	支持（Git）	支持	支持
灰度发布	支持	支持	待支持
权限管理	支持	支持	待支持
多机群	支持	支持	支持
多环境	支持	支持	支持
监听查询	支持	支持	支持
多语言	只支持Java	Go,C++,Python,Java,.net,OpenAPI	Python,Java,Nodejs,OpenAPI
配置格式校验	不支持	支持	支持
分布式部署	复杂	复杂	简单
数据一致性	Git保证数据一致性，Config-Server从Git读取数据	数据库模拟消息队列，Apollo定时读消息	HTTP异步通知
通信协议	HTTP 和 AMQP	HTTP	HTTP
单机读	7（限流所致）	9000	15000
单机写	5（限流所致）	1100	1800

- **灰度发布**：
  - Spring Cloud Config支持通过/bus/refresh端点的destination参数来指定要更新配置的机器，不过整个流程不够自动化和体系化。
  - Apollo可以直接在控制台上点灰度发布指定发布机器的IP，接着再全量发布，做得比较体系化。Nacos目前发布到0.9版本，还不支持灰度发布。
- **权限管理**：
  - Spring Cloud Config依赖Git的权限管理能力，开源的GitHub权限控制可以分为Admin、Write和Read权限，权限管理比较完善。
  - Apollo通过项目的维度来对配置进行权限管理，一个项目的owner可以授权给其他用户配置的修改发布权限。
  - Nacos目前看还不具备权限管理能力。

- **版本管理&回滚**：
  - Spring Cloud Config、Apollo和Nacos都具备配置的版本管理和回滚能力，可以在控制台上查看配置的变更情况或进行回滚操作。
  - Spring Cloud Config通过Git来做版本管理，更方便些。
- **配置格式校验**：
  - Spring Cloud Config使用Git，目前还不支持格式检验，格式的正确性依赖研发人员自己。
  - Apollo和Nacos都会对配置格式的正确性进行检验，可以有效防止人为错误。
- **监听查询**：
  - Spring Cloud Config使用Spring Cloud Bus推送配置变更，Spring Cloud Bus兼容 RabbitMQ、Kafka等，支持查询订阅Topic和Consumer的订阅关系。
  - Apollo可以通过灰度实例列表查看监听配置的实例列表，但实例监听的配置(Apollo称为命名空间)目前还没有展示出来。
  - Nacos可以查看监听配置的实例，也可以查看实例监听的配置情况。
  - 基本上，这三个产品都具备监听查询能力，在我们自己的使用过程中，Nacos使用起来相对简单，易用性相对更好些。
- **多环境**：
  - Spring Cloud Config支持Profile的方式隔离多个环境，通过在Git上配置多个Profile的配置文件，客户端启动时指定Profile就可以访问对应的配置文件。
  - Apollo也支持多环境，在控制台创建配置的时候就要指定配置所在的环境，客户端在启动的时候指定JVM参数ENV来访问对应环境的配置文件。
  - Nacos通过命名空间来支持多环境，每个命名空间的配置相互隔离，客户端指定想要访问的命名空间就可以达到逻辑隔离的作用。
- **多集群**：
  - Spring Cloud Config可以通过搭建多套Config Server，Git使用同一个Git的多个仓库，来实现物理隔离。
  - Apollo可以搭建多套集群，Apollo的控制台和数据更新推送服务分开部署，控制台部署一套就可以管控多个集群。
  - Nacos控制台和后端配置服务是部署在一起的，可以通过不同的域名切换来支持多集群。
- **配置实时推送**：
  - Nacos和Apollo配置推送都是基于HTTP长轮询，客户端和配置中心建立HTTP长联接，当配置变更的的时候，配置中心把配置推送到客户端。
  - Spring Cloud Config原生不支持配置的实时推送，需要依赖Git的WebHook、Spring Cloud Bus和客户端/bus/refresh端点。
  - Nacos和Apollo在配置实时推送链路上是比较简单高效的，Spring Cloud Config的配置推送引入Spring Cloud Bus，链路较长，比较复杂。
- **多语言支持**：
  - Spring Cloud服务于Java生态，一开始只是针对Java微服务应用，对于非Java应用的微服务调用，可以使用Sidecar提供了HTTP API，但动态配置方面还不能很好的支持。
  - Apollo已经支持了多种语言，并且提供了open API。其他不支持的语言，Apollo的接入成本相对较低。
  - Nacos支持主流的语言，例如Java、Go、Python、Nodejs、PHP等，也提供了open API。
- **性能对比**：
  - Nacos的读写性能最高，Apollo次之，Spring Cloud Config的依赖Git场景不适合开放的大规模自动化运维API。

配置中心选型

总的来说：

- Apollo和Nacos相对于Spring Cloud Config的生态支持更广，在配置管理流程上做的更好。
- Apollo相对于Nacos在配置管理做的更加全面，不过使用起来也要麻烦一些。
- Apollo容器化较困难，Nacos有官网的镜像可以直接部署，总体来说，Nacos比Apollo更符合KISS原则。
- Nacos使用起来相对比较简洁，在对性能要求比较高的大规模场景更适合。

此外，Nacos除了提供配置中心的功能，还提供了动态服务发现、服务共享与管理的功能，降低了服务化改造过程中的难度。

但对于一个开源项目的选型，除了以上这几个方面，项目上的人力投入（迭代进度、文档的完整性）、社区的活跃度（issue的数量和解决速度、Contributor数量、社群的交流频次等）、社区的规范程度（免责说明、安全性说明等），这些可能才是用户更关注的内容。

更多对比选型内容，请参考文章 《阿里面试这样问：Nacos、Apollo、Config配置中心如何选型？这10个维度告诉你！》，文中很多知识点也是来自该文章。

📚 第 5 章：监控系统

> 本篇文章为转载文章，支持原创！
>
> 原作者：骆俊武
>
> 原文链接：监控系统选型，这篇不可不读！

之前，我写过几篇有关「线上问题排查」的文章，文中附带了一些监控图，有些读者对此很感兴趣，问我监控系统选型上有没有好的建议？

目前我所经历的几家公司，监控系统都是自研的。其实业界有很多优秀的开源产品可供选择，能满足绝大部分的监控需求，如果能从中选择一款满足企业当下的诉求，显然最省时省力。

这篇文章，我将对监控体系的基础知识、原理和架构做一次系统性整理，同时还会对几款最常用的开源监控产品做下介绍，以便大家选型时参考。内容包括3部分：

- 必知必会的监控基础知识
- 主流监控系统介绍
- 监控系统的选型建议

01 必知必会的监控基础知识

监控系统俗称「第三只眼」，几乎是我们每天都会打交道的系统，下面 4 项基础知识我认为是必须要了解的。

1. 监控系统的7大作用

正所谓「无监控，不运维」，监控系统的地位不言而喻。不管你是监控系统的开发者还是使用者，首先肯定要清楚：监控系统的目标是什么？它能发挥什么作用？

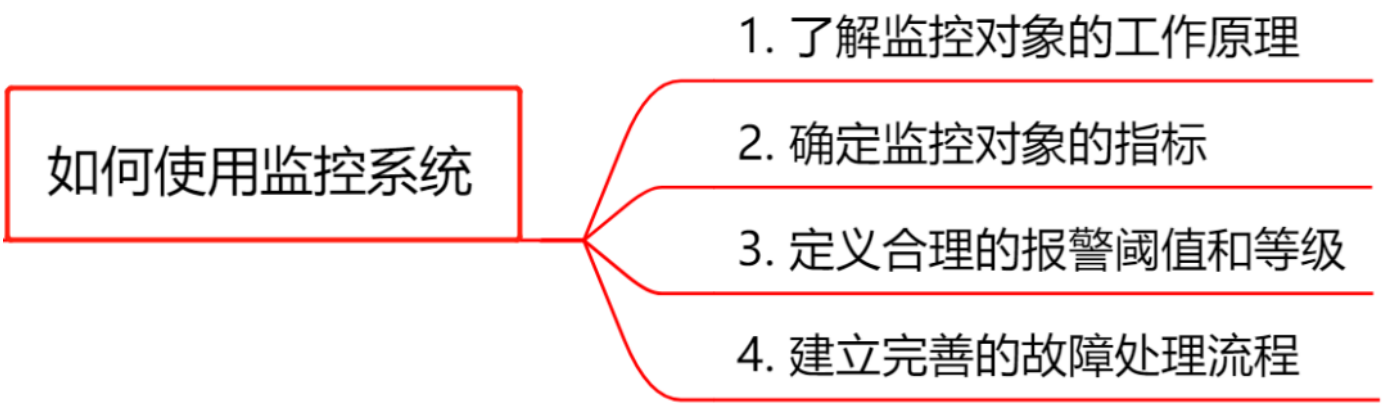
- **实时采集监控数据**：包括硬件、操作系统、中间件、应用程序等各个维度的数据。
- **实时反馈监控状态**：通过对采集的数据进行多维度统计和可视化展示，能实时体现监控对象的状态是正常还是异常。
- **预知故障和告警**：能够提前预知故障风险，并及时发出告警信息。
- **辅助定位故障**：提供故障发生时的各项指标数据，辅助故障分析和定位。
- **辅助性能调优**：为性能调优提供数据支持，比如慢SQL，接口响应时间等。
- **辅助容量规划**：为服务器、中间件以及应用集群的容量规划提供数据支撑。
- **辅助自动化运维**：为自动扩容或者根据配置的SLA进行服务降级等智能运维提供数据支撑。

2. 使用监控系统的正确姿势

> “
>
> 出任何线上事故，先不说其他地方有问题，监控部分一定是有问题的。

听着很甩锅的一句话，仔细思考好像有一定道理。我们在事故复盘时，通常会思考这3个和监控有关的问题：有没有做监控？监控是否及时？监控信息是否有助于快速定位问题？

可见光有一套好的监控系统还不够，还必须知道**「如何 用好它」**。一个成熟的研发团队通常会定一个监控规范，用来统一监控系统的使用方法。



- **了解监控对象的工作原理：**要做到对监控对象有基本的了解，清楚它的工作原理。比如想对JVM进行监控，你必须清楚JVM的堆内存结构和垃圾回收机制。
- **确定监控对象的指标：**清楚使用哪些指标来刻画监控对象的状态？比如想对某个接口进行监控，可以采用请求量、耗时、超时量、异常量等指标来衡量。
- **定义合理的报警阈值和等级：**达到什么阈值需要告警？对应的故障等级是多少？不需要处理的告警不是好告警，可见定义合理的阈值有多重要，否则只会降低运维效率或者让监控系统失去它的作用。
- **建立完善的故障处理流程：**收到故障告警后，一定要有相应的处理流程和oncall机制，让故障及时被跟进处理。

3. 监控的对象和指标都有哪些?

监控已然成为了整个产品生命周期非常重要的一环，运维关注硬件和基础监控，研发关注各类中间件和应用层的监控，产品关注核心业务指标的监控。可见，监控的对象已经越来越立体化。

这里，我对常用的监控对象以及监控指标做了分类整理，供大家参考。

3.1 硬件监控

包括：电源状态、CPU状态、机器温度、风扇状态、物理磁盘、raid状态、内存状态、网卡状态

3.2 服务器基础监控

- CPU：单个CPU以及整体的使用情况
- 内存：已用内存、可用内存
- 磁盘：磁盘使用率、磁盘读写的吞吐量
- 网络：出口流量、入口流量、TCP连接状态

3.3 数据库监控

包括：数据库连接数、QPS、TPS、并行处理的会话数、缓存命中率、主从延时、锁状态、慢查询

3.4 中间件监控

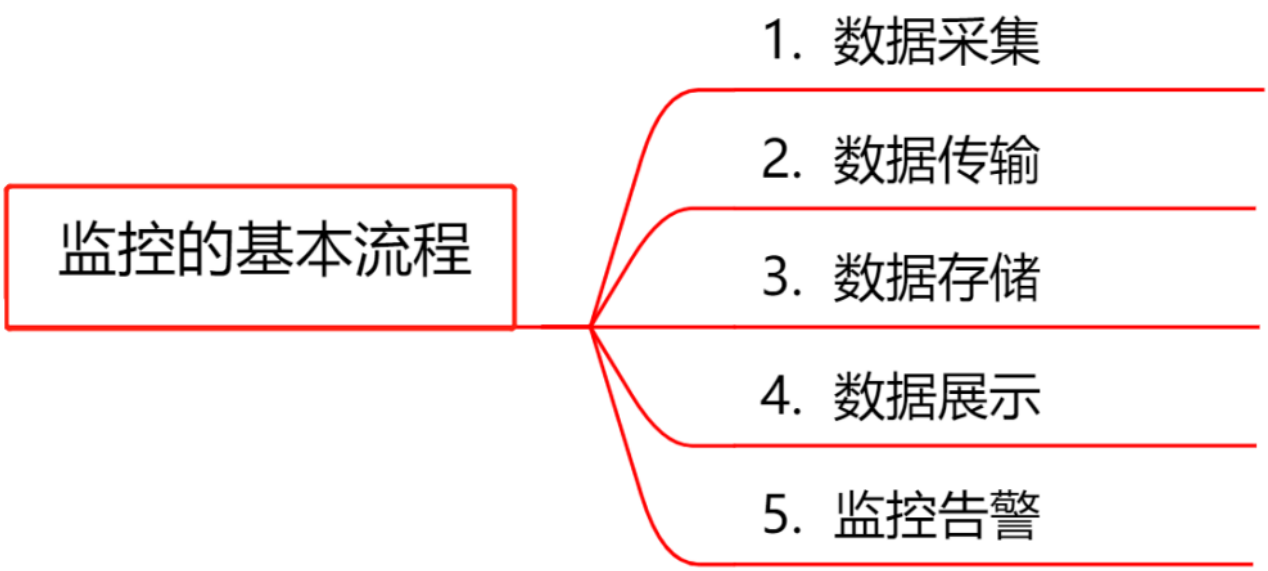
- Nginx：活跃连接数、等待连接数、丢弃连接数、请求量、耗时、5XX错误率
- Tomcat：最大线程数、当前线程数、请求量、耗时、错误量、堆内存使用情况、GC次数和耗时
- 缓存 ： 成功连接数、阻塞连接数、已使用内存、内存碎片率、请求量、耗时、缓存命中率
- 消息队列：连接数、队列数、生产速率、消费速率、消息堆积量

3.5 应用监控

- HTTP接口：URL存活、请求量、耗时、异常量
- RPC接口：请求量、耗时、超时量、拒绝量
- JVM ： GC次数、GC耗时、各个内存区域的大小、当前线程数、死锁线程数
- 线程池：活跃线程数、任务队列大小、任务执行耗时、拒绝任务数
- 连接池：总连接数、活跃连接数
- 日志监控：访问日志、错误日志
- 业务指标：视业务来定，比如PV、订单量等

4. 监控系统的基本流程

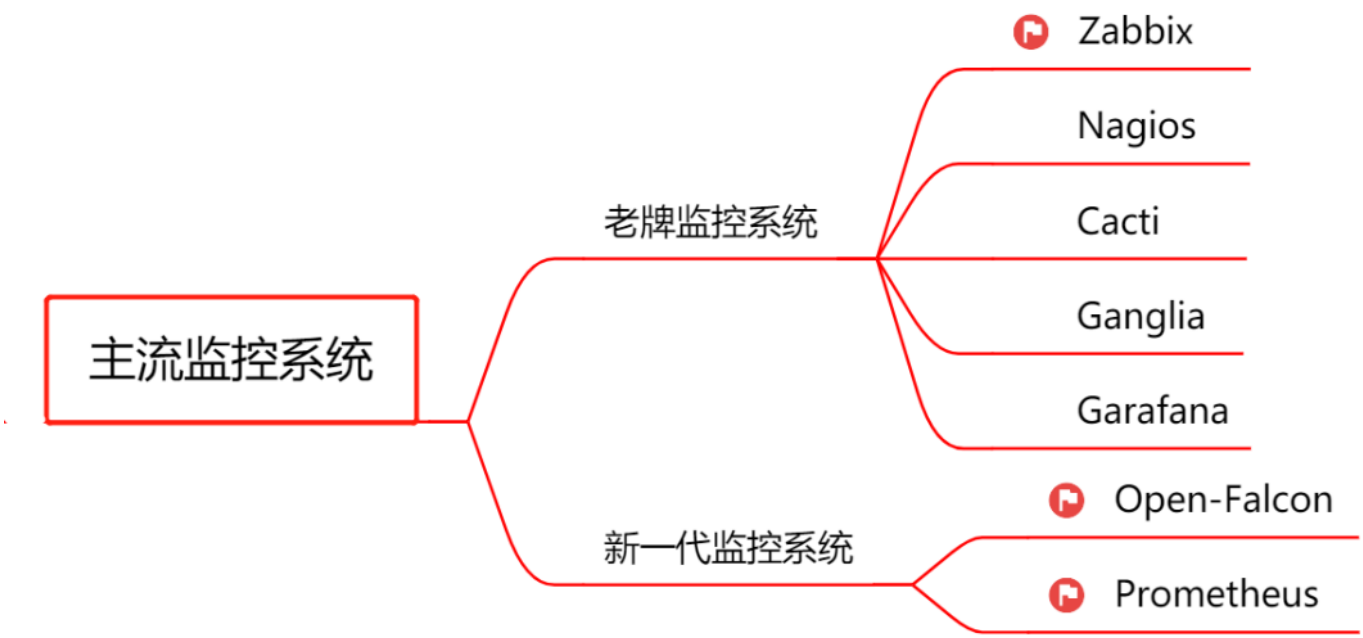
无论是开源的监控系统还是自研的监控系统，监控的整个流程大同小异，一般都包括以下模块：



- **数据采集**：采集的方式有很多种，包括日志埋点进行采集（通过Logstash、Filebeat等进行上报和解析），JMX标准接口输出监控指标，被监控对象提供REST API进行数据采集（如Hadoop、ES），系统命令行，统一的SDK进行侵入式的埋点和上报等。
- **数据传输**：将采集的数据以TCP、UDP或者HTTP协议的形式上报给监控系统，有主动Push模式，也有被动Pull模式。
- **数据存储**：有使用MySQL、Oracle等RDBMS存储的，也有使用时序数据库RRDTool、OpentTSDB、InfluxDB存储的，还有使用HBase存储的。
- **数据展示**：数据指标的图形化展示。
- **监控告警**：灵活的告警设置，以及支持邮件、短信、IM等多种通知通道。

02 主流监控系统介绍

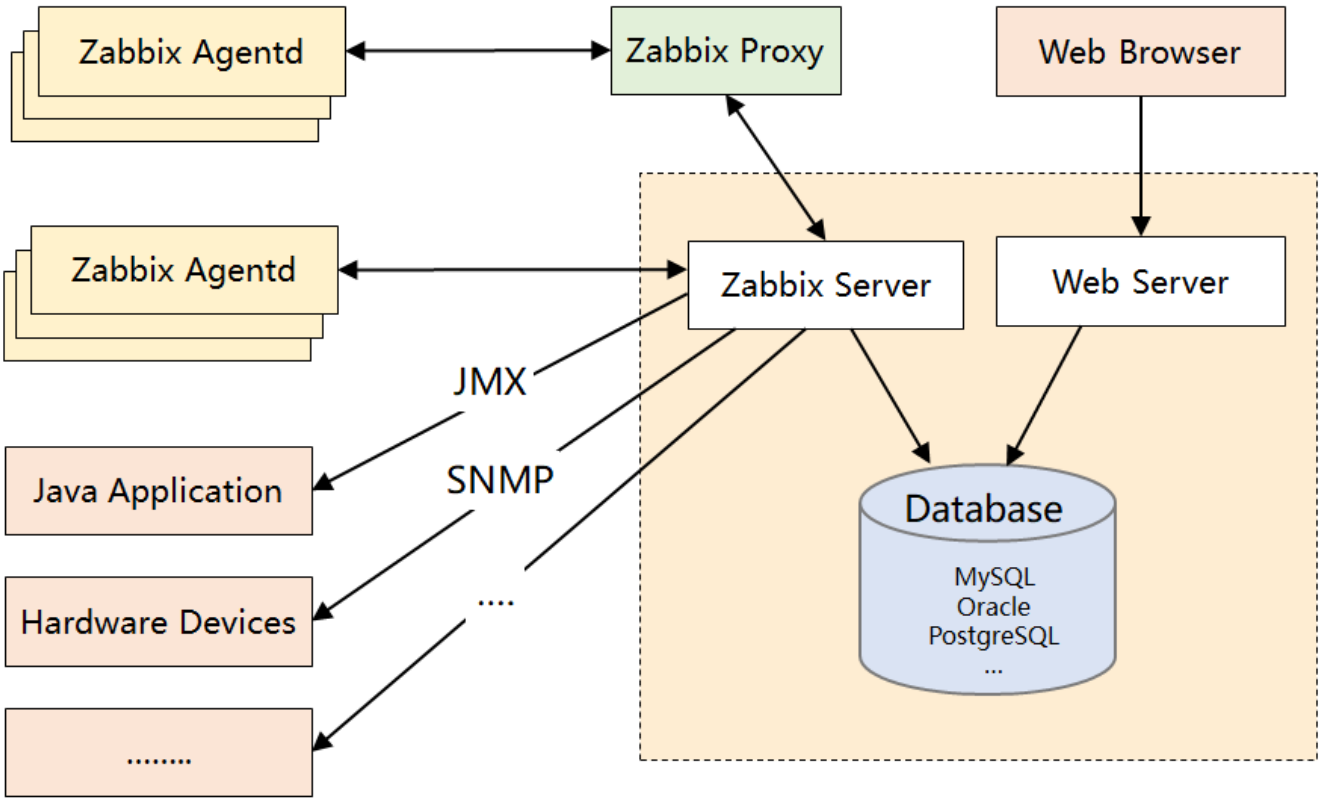
下面再来认识下主流的开源监控系统，由于篇幅有限，我挑选了3款使用最广泛的监控系统：Zabbix、Open-Falcon、Prometheus，会对它们的架构进行介绍，同时总结下各自的优劣势。



1. Zabbix（老牌监控的优秀代表）

Zabbix 1998年诞生，核心组件采用C语言开发，Web端采用PHP开发。它属于老牌监控系统中的优秀代表，监控功能很全面，使用也很广泛，差不多有70%左右的互联网公司都曾使用过 Zabbix 作为监控解决方案。

先来了解下Zabbix的架构设计：



Zabbix架构图

- **Zabbix Server**：核心组件，C语言编写，负责接收Agent、Proxy发送的监控数据，也支持JMX、SNMP等多种协议直接采集数据。同时，它还负责数据的汇总存储以及告警触发等。
- **Zabbix Proxy**：可选组件，对于被监控机器较多的情况下，可使用Proxy进行分布式监控，它能代理Server收集部分监控数据，以减轻Server的压力。
- **Zabbix Agentd**：部署在被监控主机上，用于采集本机的数据并发送给Proxy或者Server，它的插件机制支持用户自定义数据采集脚本。Agent可在Server端手动配置，也可以通过自动发现机制被识别。数据收集方式同时支持主动Push和被动Pull 两种模式。
- **Database**：用于存储配置信息以及采集到的数据，支持MySQL、Oracle等关系型数据库。同时，最新版本的Zabbix已经开始支持时序数据库，不过成熟度还不高。
- **Web Server**：Zabbix的GUI组件，PHP编写，提供监控数据的展现和告警配置。

下面是 Zabbix 的优势：

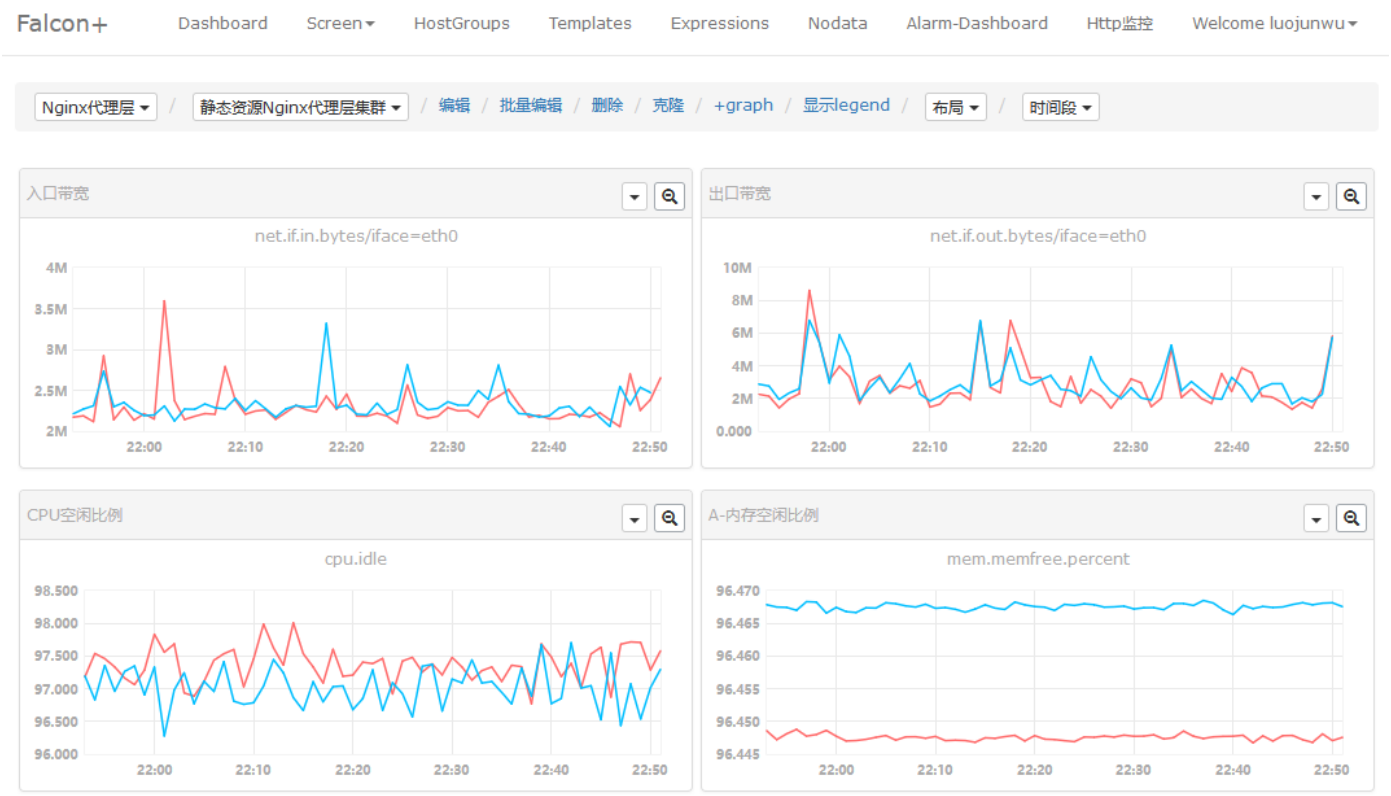
- **产品成熟**：由于诞生时间长且使用广泛，拥有丰富的文档资料以及各种开源的数据采集插件，能覆盖绝大部分监控场景。
- **采集方式丰富**：支持Agent、SNMP、JMX、SSH等多种采集方式，以及主动和被动的数据传输方式。
- **较强的扩展性**：支持Proxy分布式监控，有agent自动发现功能，插件式架构支持用户自定义数据采集脚本。
- **配置管理方便**：能通过Web界面进行监控和告警配置，操作方便，上手简单。

下面是 Zabbix 的劣势：

- **性能瓶颈**：机器量或者业务量大了后，关系型数据库的写入一定是瓶颈，官方给出的单机上限是5000台，个人感觉达不到，尤其现在应用层的指标越来越多。虽然最新版已经开始支持时序数据库，不过成熟度还不高。
- **应用层监控支持有限**：如果想对应用程序做侵入式的埋点和采集（比如监控线程池或者接口性能），zabbix没有提供对应的sdk，通过插件式的脚本也能曲线实现此功能，个人感觉zabbix就不是做这个事的。

- **数据模型不强大**：不支持tag，因此没法按多维度进行聚合统计和告警配置，使用起来不灵活。
- **方便二次开发难度大**：Zabbix采用的是C语言，二次开发往往需要熟悉它的数据表结构，基于它提供的API更多只能做展示层的定制。

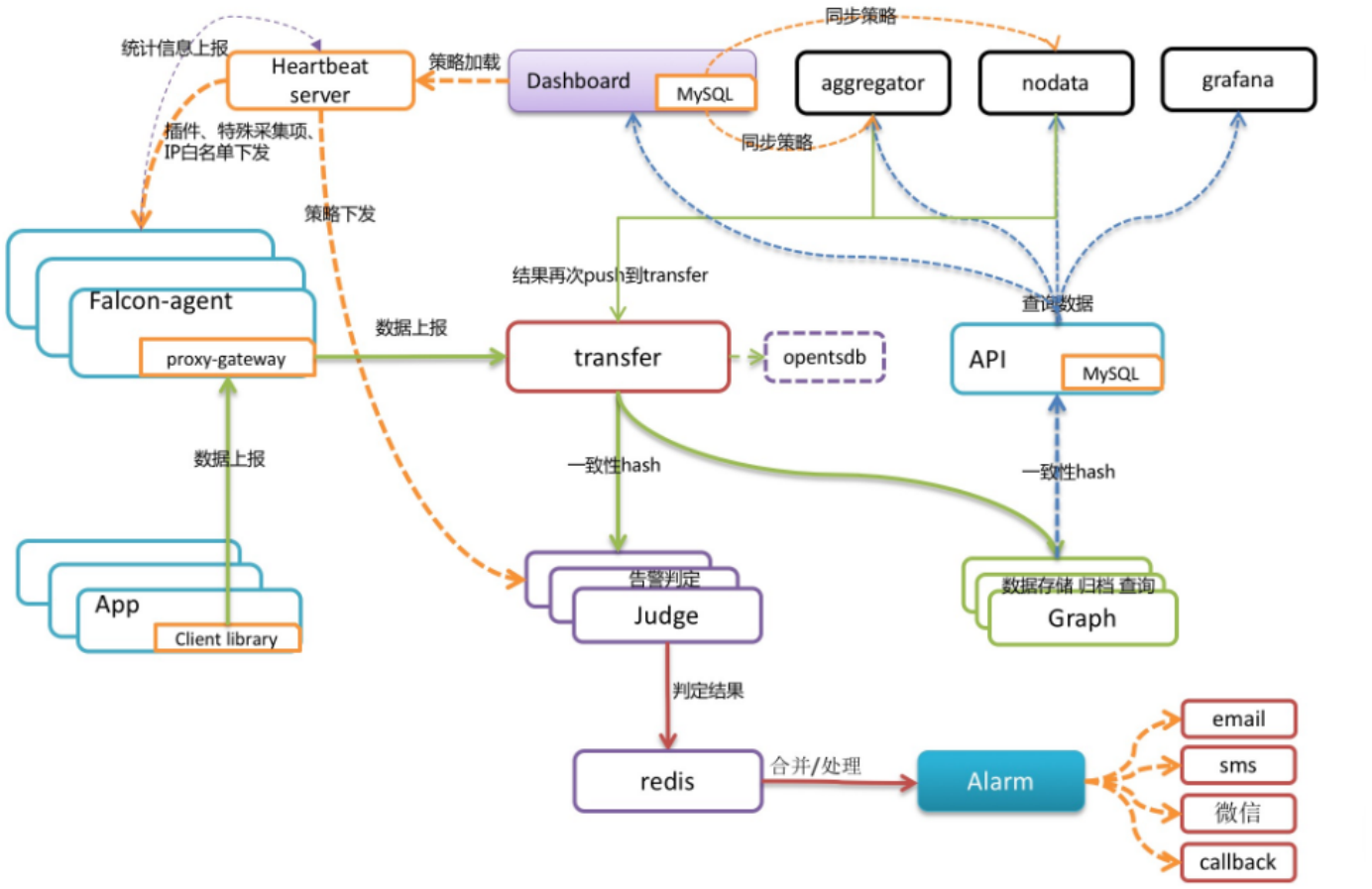
2. Open-Falcon（小米出品，国内流行）



Open-falcon 是小米2015年开源的企业级监控工具，采用Go和Python语言开发，这是一款灵活、高性能且易扩展的新一代监控方案，目前小米、美团、滴滴等超过200家公司在使用它。

小米初期也使用的Zabbix进行监控，但是机器量和业务量上来后，Zabbix就有些力不从心了。因此，后来自主研发了Open-Falcon，在架构设计上吸取了Zabbix的经验，同时很好地解决了Zabbix的诸多痛点。

先来了解下Open-Falcon的架构设计：



Open-Falcon架构图，来自网络

- **Falcon-agent**：数据采集器和收集器，Go开发，部署在被监控的机器上，支持3种数据采集方式。首先它能自动采集单机200多个基础监控指标，无需做任何配置；同时支持用户自定义的plugin获取监控数据；此外，用户可通过http接口，自主push数据到本机的proxy-gateway，由gateway转发到server.
- **Transfer**：数据分发组件，接收客户端发送的数据，分别发送给数据存储组件Graph和告警判定组件Judge，Graph和Judge均采用一致性hash做数据分片，以提高横向扩展能力。同时Transfer还支持将数据分发到OpenTSDB，用于历史归档。
- **Graph**：数据存储组件，底层使用RRDTool（时序数据库）做单个指标的存储，并通过缓存、分批写入磁盘等方式进行了优化。据说一个graph实例能够处理8W+每秒的写入速率。
- **Judge和Alarm**：告警组件，Judge对Transfer组件上报的数据进行实时计算，判断是否要产生告警事件，Alarm组件对告警事件进行收敛处理后，将告警消息推送给各个消息通道。
- **API**：面向终端用户，收到查询请求后会去Graph中查询指标数据，汇总结果后统一返回给用户，屏蔽了存储集群的分片细节。

下面是Open-Falcon的优势：

- **自动采集能力**：Falcon-agent 能自动采集服务器的200多个基础指标（比如CPU、内存等），无需在server上做任何配置，这一点可以秒杀Zabbix.
- **强大的存储能力**：底层采用RRDTool，并且通过一致性hash进行数据分片，构建了一个分布式的时序数据存储系统，可扩展性强。
- **灵活的数据模型**：借鉴OpenTSDB，数据模型中引入了tag，这样能支持多维度的聚合统计以及告警规则设置，大大提高了使用效率。
- **插件统一管理**：Open-Falcon的插件机制实现了对用户自定义脚本的统一化管理，可通过HeartBeat Server分发给agent，减轻了使用者自主维护脚本的成本。
- **个性化监控支持**：基于Proxy-gateway，很容易通过自主埋点实现应用层的监控（比如监控接口的访问量和耗

时）和其他个性化监控需求，集成方便。

下面是Open-Falcon的劣势：

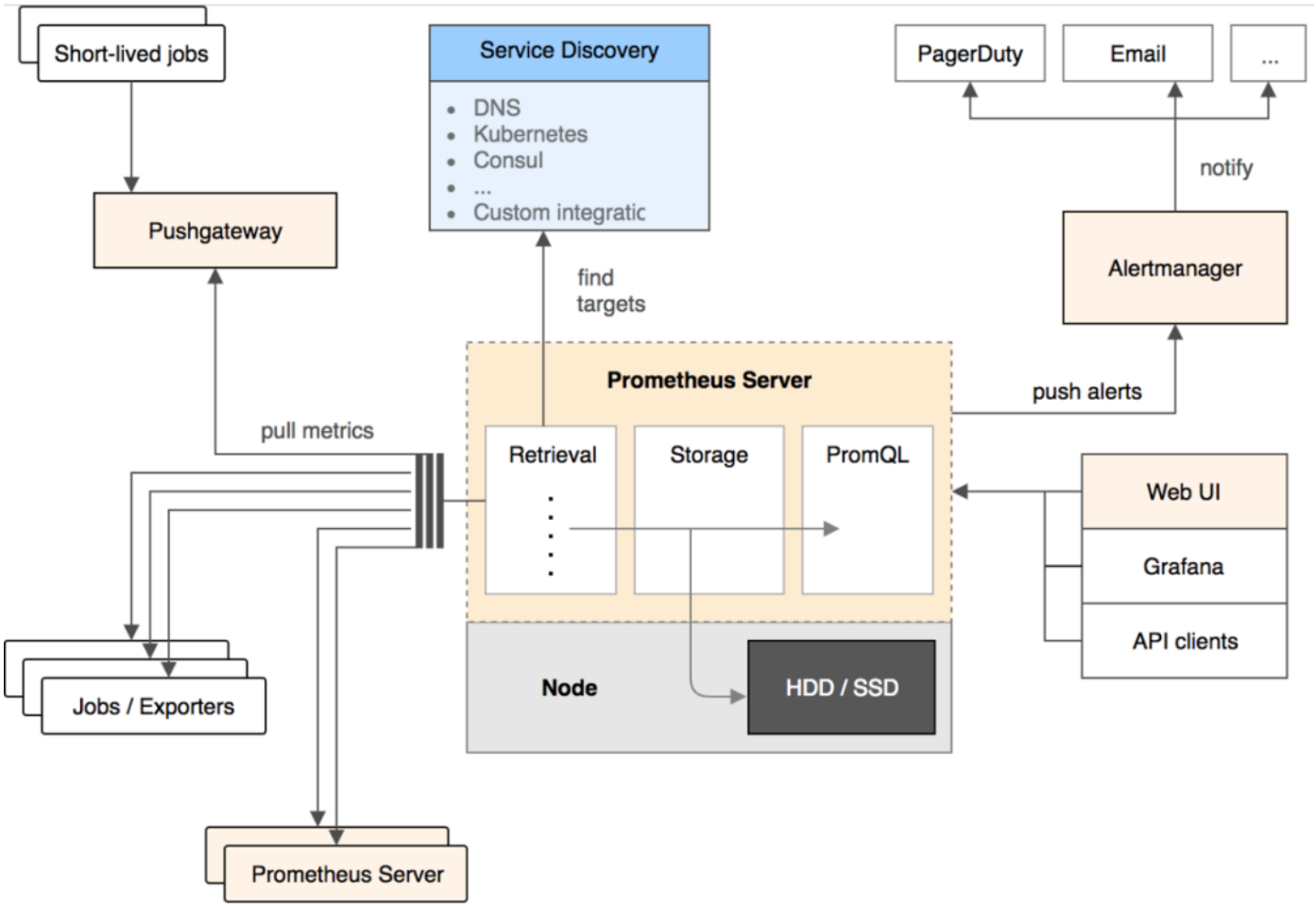
- **整体发展一般：** 社区活跃度不算高，同时版本更新慢，有些大厂是基于它的稳定版本直接做二次开发的，关于以后的前景其实有点担忧。
- **UI不够友好**：对于业务线的研发来说，可能只想便捷地完成告警配置和业务监控，但是它把机器分组、策略模板、模板继承等概念全部暴露在UI上，感觉在围绕这几个概念设计UI，理解有点费劲。
- **安装比较复杂：**个人的亲身感受，由于它是从小米内部衍生出来的，虽然去掉了对小米内部系统的依赖，但是组件还是比较多，如果对整个架构不熟悉，安装很难一蹴而就。

3. Prometheus（号称下一代监控系统）

Prometheus（普罗米修斯）是由前google员工2015年正式发布的开源监控系统，采用Go语言开发。它不仅有一个很酷的名字，同时它有Google与k8s的强力支持，开源社区异常火爆。

Prometheus 2016年加入云原生基金会，是继k8s后托管的第二个项目，未来前景被相当看好。它和Open-Falcon最大不同在于：数据采集是基于Pull模式的，而不是Push模式，并且架构非常简单。

先来了解下Prometheus的架构设计：



Prometheus架构图，来自网络

- **Prometheus Server**：核心组件，用于收集、存储监控数据。它同时支持静态配置和通过Service Discovery动态发现来管理监控目标，并从监控目标中获取数据。此外，Prometheus Server 也是一个时序数据库，它将监控数据保存在本地磁盘中，并对外提供自定义的 PromQL 语言实现对数据的查询和分析。
- **Exporter**：用来采集数据，作用类似于agent，区别在于Prometheus是基于Pull方式拉取采集数据的，因此，Exporter通过HTTP服务的形式将监控数据按照标准格式暴露给Prometheus Server，社区中已经有大量现成的Exporter可以直接使用，用户也可以使用各种语言的client library自定义实现。
- **Push gateway**：主要用于瞬时任务的场景，防止Prometheus Server来pull数据之前此类Short-lived jobs就已经执行完毕了，因此job可以采用push的方式将监控数据主动汇报给Push gateway缓存起来进行中转。
- **Alert Manager**：当告警产生时，Prometheus Server将告警信息推送给Alert Manager，由它发送告警信息给接收方。
- **Web UI**：Prometheus内置了一个简单的web控制台，可以查询配置信息和指标等，而实际应用中我们通常会将Prometheus作为Grafana的数据源，创建仪表盘以及查看指标。

下面是Prometheus的优势：

- **轻量管理**：架构简单，不依赖外部存储，单个服务器节点可直接工作，二进制文件启动即可，属于轻量级的Server，便于迁移和维护。
- **较强的处理能力**：监控数据直接存储在Prometheus Server本地的时序数据库中，单个实例可以处理数百万的metrics。
- **灵活的数据模型**：同Open-Falcon，引入了tag，属于多维数据模型，聚合统计更方便。
- **强大的查询语句**：PromQL允许在同一个查询语句中，对多个metrics进行加法、连接和取分位值等操作。
- **很好地支持云环境**：能自动发现容器，同时k8s和etcd等项目都提供了对Prometheus的原生支持，是目前容器监控最流行的方案。

下面是Prometheus的劣势：

- **功能不够完善**：Prometheus从一开始的架构设计就是要做到简单，不提供集群化方案，长期的持久化存储和用户管理，而这些是企业变大后所必须的特性，目前要做到这些只能在Prometheus之上进行扩展。
- **网络规划变复杂**：由于Prometheus采用的是Pull模型拉取数据，意味着所有被监控的endpoint必须是可达的，需要合理规划网络的安全配置。

03 监控系统的选型建议

通过上面的介绍，大家对主流的监控系统应该有了一定的认识。面对选型问题，我的建议是：

1、先明确清楚你的监控需求：要监控的对象有哪些？机器数量和监控指标有多少？需要具备什么样的告警功能?

2、监控是一项长期建设的事情，一开始就想做一个 All In One 的监控解决方案，我觉得没有必要。从成本角度考虑，在初期直接使用开源的监控方案即可，先解决有无问题。

3、从系统成熟度上看，Zabbix属于老牌的监控系统，资料多，功能全面且稳定，如果机器数量在几百台以内，不用太担心性能问题，另外，采用数据库分区、SSD硬盘、Proxy架构、Push采集模式都可以提高监控性能。

4、Zabbix在服务器监控方面占绝对优势，可以满足90%以上的监控场景，但是应用层的监控似乎并不擅长，比如要监控线程池的状态、某个内部接口的执行时间等，这种通常都要做侵入式埋点。相反，新一代的监控系统Open-Falcon和Prometheus在这一点做得很好。

5、从整体表现上来看，新一代监控系统也有明显的优势，比如：灵活的数据模型、更成熟的时序数据库、强大的告警功能，如果之前对zabbix这种传统监控没有技术积累，建议使用Open-Falcon或者Prometheus.

6、Open-Falcon的核心优势在于数据分片功能，能支撑更多的机器和监控项；Prometheus则是容器监控方面的标配，有Google和k8s加持。

7、Zabbix、Open-Falcon和Prometheus都支持和Grafana做快速集成，想要美观且强大的可视化体验，可以和Grafana进行组合。

8、用合适的监控系统解决相应的问题即可，可以多套监控同时使用，这种在企业初期很常见。

9、到中后期，随着机器数据增加和个性化需求增多（比如希望统一监控平台、打通公司的CMDB和组织架构关系），往往需要二次开发或者通过监控系统提供的API做集成，从这点来看，Open-Falcon或者Prometheus更合适。

10、如果非要自研，可以多研究下主流监控系统的架构方案，借鉴它们的优势。

04 最后的话

本文对监控体系的基础知识、原理和主流架构做了详细梳理，希望有助于大家对监控系统的认识，以及在技术选型时做出更合适的选择。

由于篇幅问题，本文的内容并未涉及到全链路监控、日志监控、以及Web前端和客户端的监控，可见监控真的是一个庞大且复杂的体系，如果想理解透彻，必须理论结合实践再做深入。

对于运维监控体系，如果你们也有自己的经验和体会，欢迎留言讨论。

微信搜 **楼仔** 或扫描下方二维码关注楼仔的原创公众号，回复 **110** 即可免费领取。

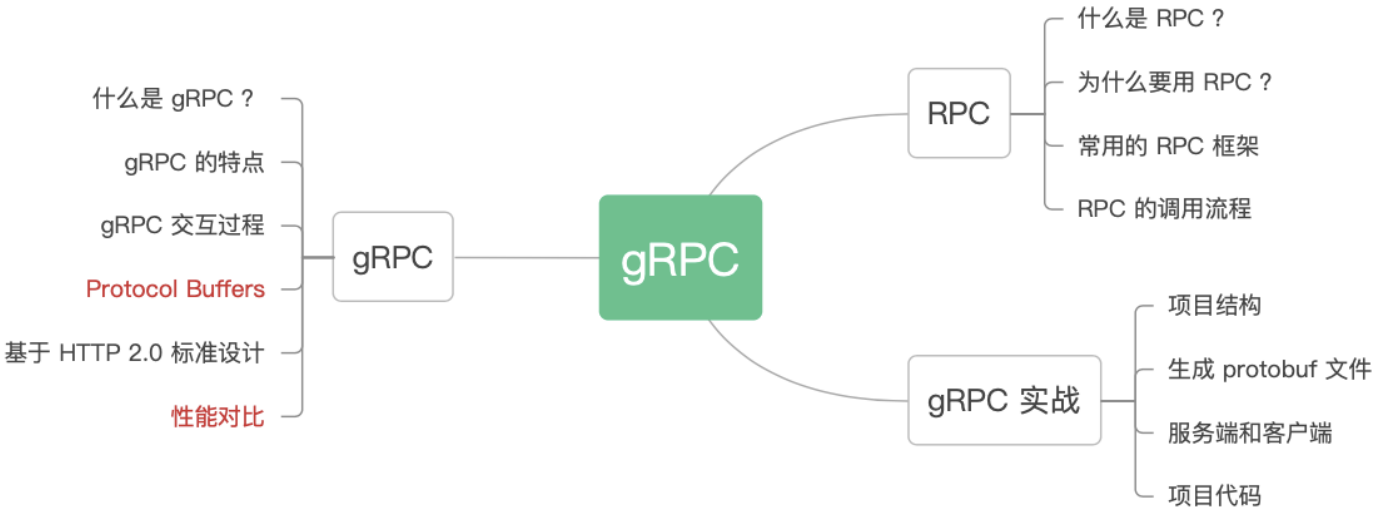
📚 第 6 章：RPC 框架

> 文章阅读 25 分钟，建议收藏。

RPC、gRPC、Thrift、HTTP，大家知道它们之前的联系和区别么？这些都是面试常考的问题，今天我们带大家先搞懂 RPC 和 gRPC。

在讲述 gPRC 之前，我们需要先搞懂什么是 RPC。

不 BB，直接上文章目录：

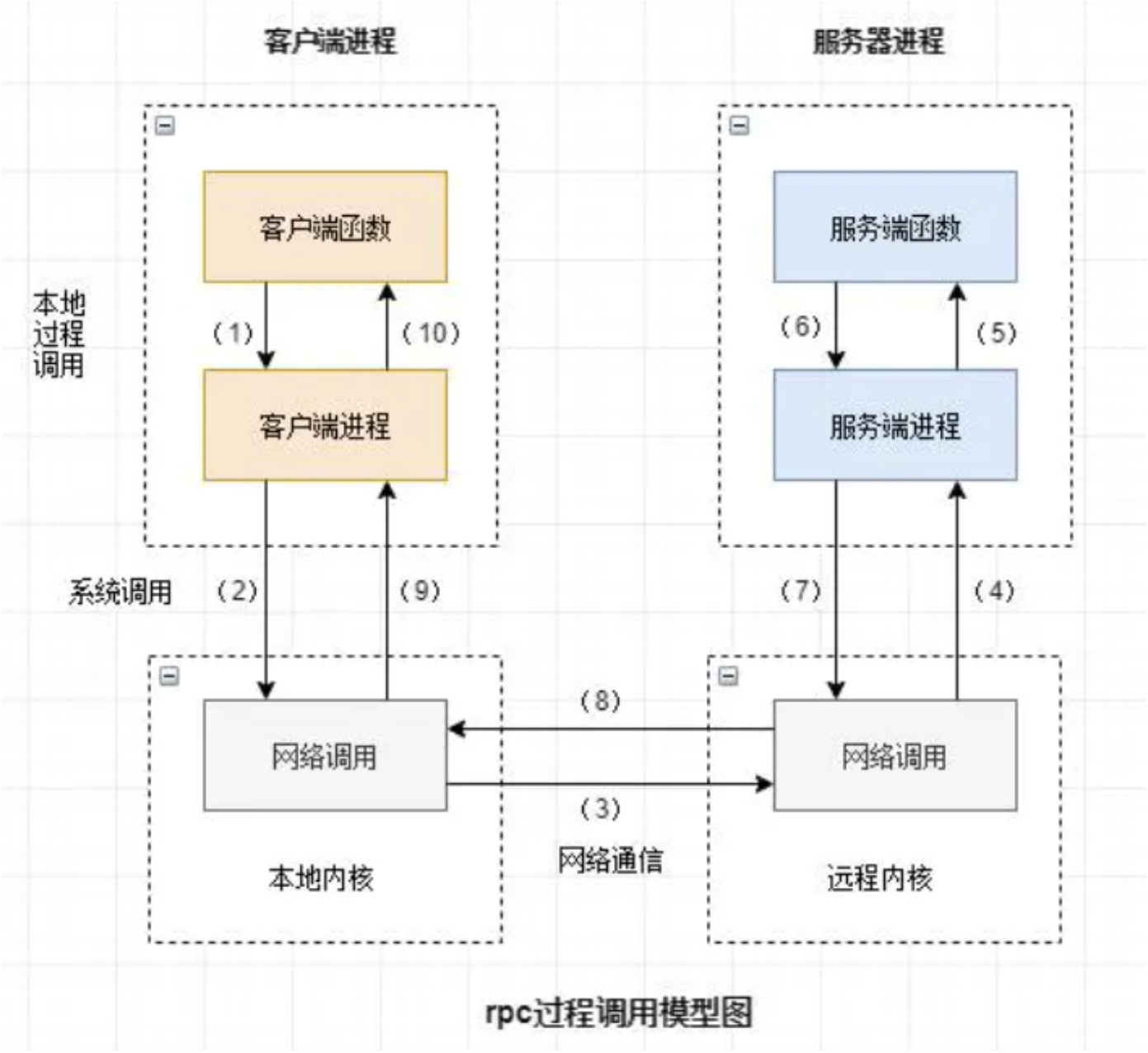


1. RPC

1.1 什么是 RPC？

RPC（Remote Procedure Call Protocol）远程过程调用协议，目标就是让远程服务调用更加简单、透明。

RPC 框架负责屏蔽底层的传输方式（TCP 或者 UDP）、序列化方式（XML/Json/ 二进制）和通信细节，服务调用者可以像调用本地接口一样调用远程的服务提供者，而不需要关心底层通信细节和调用过程。



1.2 为什么要用 RPC？

当我们的业务越来越多、应用也越来越多时，自然的，我们会发现有些功能已经不能简单划分开来或者划分不出来。

此时可以将公共业务逻辑抽离出来，将之组成独立的服务 Service 应用，而原有的、新增的应用都可以与那些独立的 Service 应用 交互，以此来完成完整的业务功能。

所以我们急需一种高效的应用程序之间的通讯手段来完成这种需求，RPC 大显身手的时候来了！

1.3 常用的 RPC 框架

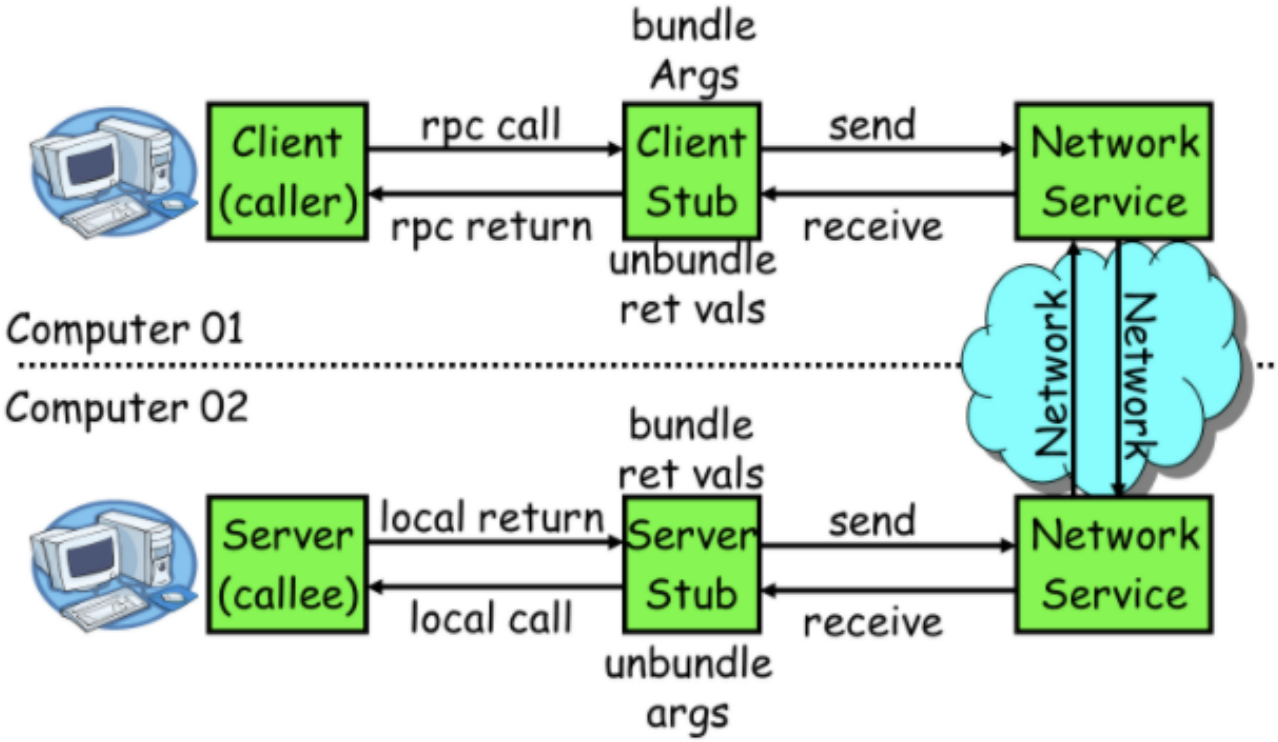
- **gRPC**：一开始由 google 开发，是一款语言中立、平台中立、开源的远程过程调用(RPC)系统。
- **Thrift**：thrift 是一个软件框架，用来进行可扩展且跨语言的服务的开发。它结合了功能强大的软件堆栈和代码生成引擎，以构建在 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml 这些编程语言间无缝结合的、高效的服务。
- **Dubbo**：Dubbo 是一个分布式服务框架，以及 SOA 治理方案，Dubbo自2011年开源后，已被许多非阿里系

公司使用。

- **Spring Cloud**：Spring Cloud 由众多子项目组成，如 Spring Cloud Config、Spring Cloud Netflix、Spring Cloud Consul 等，提供了搭建分布式系统及微服务常用的工具。

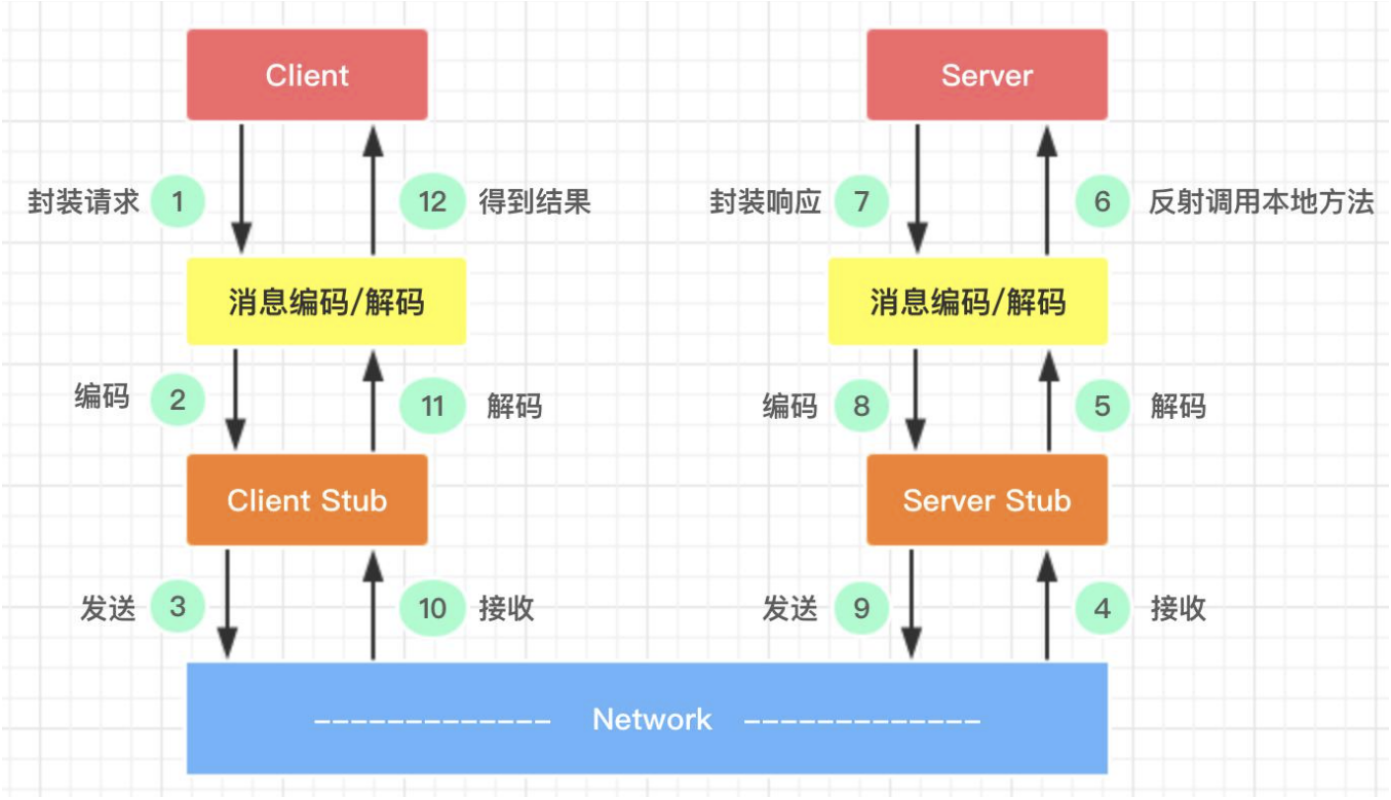
1.4 RPC 的调用流程

要让网络通信细节对使用者透明，我们需要对通信细节进行封装，我们先看下一个 RPC 调用的流程涉及到哪些通信细节：



1. 服务消费方（client）调用以本地调用方式调用服务；
2. client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
3. client stub找到服务地址，并将消息发送到服务端；
4. server stub收到消息后进行解码；
5. server stub根据解码结果调用本地的服务；
6. 本地服务执行并将结果返回给 server stub；
7. server stub将返回结果打包成消息并发送至消费方；
8. client stub接收到消息，并进行解码；
9. 服务消费方得到最终结果。

RPC 的目标就是要 2~8 这些步骤都封装起来，让用户对这些细节透明，下面是网上的另外一幅图，感觉一目了然：



2. gRPC

2.1 什么是 gRPC？

gRPC 是一个高性能、通用的开源 RPC 框架，其由 Google
2015 年主要面向移动应用开发并基于 HTTP/2 协议标准而设计，基于 ProtoBuf 序列化协议开发，且支持众多开发语言。

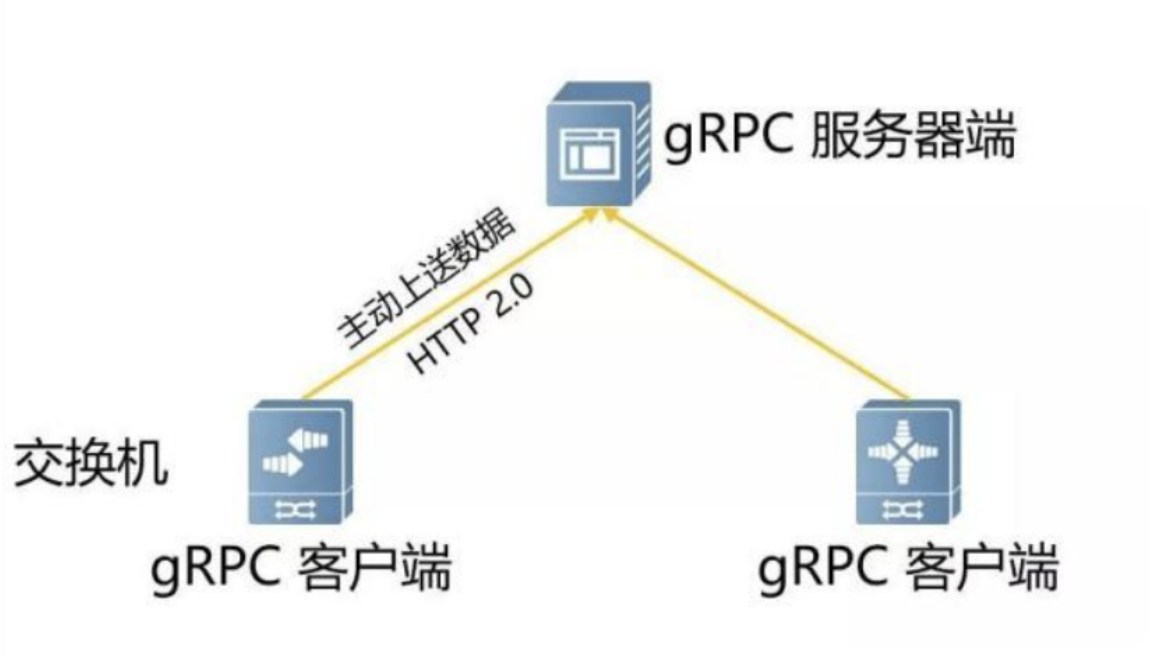
由于是开源框架，通信的双方可以进行二次开发，所以客户端和服务器端之间的通信会更加专注于业务层面的内容，减少了对由 gRPC 框架实现的底层通信的关注。

如下图，DATA 部分即业务层面内容，下面所有的信息都由 gRPC 进行封装。

2.2 gRPC 的特点

- 跨语言使用，支持 C++、Java、Go、Python、Ruby、C#、Node.js、Android Java、Objective-C、PHP 等编程语言；
- 基于 IDL 文件定义服务，通过 proto3 工具生成指定语言的数据结构、服务端接口以及客户端 Stub；
- 通信协议基于标准的 HTTP/2 设计，支持双向流、消息头压缩、单 TCP 的多路复用、服务端推送等特性，这些特性使得 gRPC 在移动端设备上更加省电和节省网络流量；
- 序列化支持 PB（Protocol Buffer）和 JSON，PB 是一种语言无关的高性能序列化框架，基于 HTTP/2 + PB, 保障了 RPC 调用的高性能；
- 安装简单，扩展方便（用该框架每秒可达到百万个RPC）。

2.3 gRPC 交互过程



- 交换机在开启 gRPC 功能后充当 gRPC 客户端的角色，采集服务器充当 gRPC 服务器角色；
- 交换机会根据订阅的事件构建对应数据的格式（GPB/JSON），通过 Protocol Buffers 进行编写 proto 文件，交换机与服务器建立 gRPC 通道，通过 gRPC 协议向服务器发送请求消息；
- 服务器收到请求消息后，服务器会通过 Protocol Buffers 解译 proto 文件，还原出最先定义好格式的数据结构，进行业务处理；
- 数据处理完后，服务器需要使用 Protocol Buffers 重编译应答数据，通过 gRPC 协议向交换机发送应答消息；
- 交换机收到应答消息后，结束本次的 gRPC 交互。

> 简单地说，gRPC 就是在客户端和服务器端开启 gRPC 功能后建立连接，将设备上配置的订阅数据推送给服务器端。
>
> 我们可以看到整个过程是需要用到 Protocol Buffers 将所需要处理数据的结构化数据在 proto 文件中进行定义。

2.4 Protocol Buffers

你可以理解 **ProtoBuf 是一种更加灵活、高效的数据格式**，与 XML、JSON 类似，在一些高性能且对响应速度有要求的数据传输场景非常适用。

ProtoBuf 在 gRPC 的框架中主要有三个作用：定义数据结构、定义服务接口，通过序列化和反序列化方式提升传输效率。

为什么 ProtoBuf 会**提高传输效率**呢?

我们知道使用 XML、JSON 进行数据编译时，数据文本格式更容易阅读，但进行数据交换时，设备就需要耗费大量的 CPU 在 I/O 动作上，自然会影响整个传输速率。

Protocol Buffers 不像前者，它会将字符串进行序列化后再进行传输，即**二进制数据**。

Protocol Buffers 编码	对应的 JSON 编码
{	{
1: “Ruijie”	"producerName": " Ruijie ",
2: “Ruijie”	"deviceName": " Ruijie ",
}	}

可以看到其实两者内容相差不大，并且内容非常直观，但是 Protocol Buffers 编码的内容只是提供给操作者阅读的，实际上传输的并不会以这种文本形式，而是序列化后的二进制数据，字节数会比 JSON、XML 的字节数少很多，速率更快。

gPRC 如何**支撑跨平台，多语言**呢？

Protocol Buffers 自带一个编译器也是一个优势点，前面提到的 proto 文件就是通过编译器进行编译的，proto 文件需要编译生成一个类似库文件，基于库文件才能真正开发数据应用。

具体用什么编程语言编译生成这个库文件呢？由于现网中负责网络设备和服务器设备的运维人员往往不是同一组人，运维人员可能会习惯使用不同的编程语言进行运维开发，那么 Protocol Buffers 其中一个优势就能发挥出来——跨语言。

从上面的介绍，我们得出在编码方面 Protocol Buffers 对比 JSON、XML 的优点：

- 标准的 IDL 和 IDL 编译器，这使得其对工程师非常友好；
- 序列化数据非常简洁，紧凑，与 XML 相比，其序**列化之后的数据量约为 1/3 到 1/10；**
- 解析速度非常快，**比对应的 XML 快约 20-100 倍；**
- 提供了非常友好的动态库，使用非常简单，反序列化只需要一行代码。

Protobuf 也有其局限性：

- 由于 Protobuf 产生于 Google，所以目前其**仅支持 Java、C++、Python 三种语言**；
- Protobuf 支持的数据类型相对较少，不支持常量类型；
- 由于其设计的理念是纯粹的展现层协议（Presentation Layer），目前并没有一个专门支持 Protobuf 的 RPC 框架。

Protobuf 适用场景：

- Protobuf 具有广泛的用户基础，空间开销小以及高解析性能是其亮点，**非常适合于公司内部的对性能要求高的 RPC 调用**；
- 由于 Protobuf 提供了标准的 IDL 以及对应的编译器，其 IDL 文件是参与各方的非常强的业务约束；
- Protobuf 与传输层无关，采用 HTTP 具有良好的跨防火墙的访问属性，所以 Protobuf 也适用于公司间对性能要求比较高的场景；

- 由于其解析性能高，序列化后数据量相对少，**非常适合应用层对象的持久化场景；**
- 主要问题在于其所**支持的语言相对较少**，另外由于没有绑定的标准底层传输层协议，**在公司间进行传输层协议的调试工作相对麻烦。**

2.5 基于 HTTP 2.0 标准设计

除了 Protocol Buffers 之外，从交互图中和分层框架可以看到， gRPC 还有另外一个优势——它是基于 HTTP 2.0 协议的。

由于 gRPC 基于 HTTP 2.0 标准设计，带来了更多强大功能，**如多路复用、二进制帧、头部压缩、推送机制。**

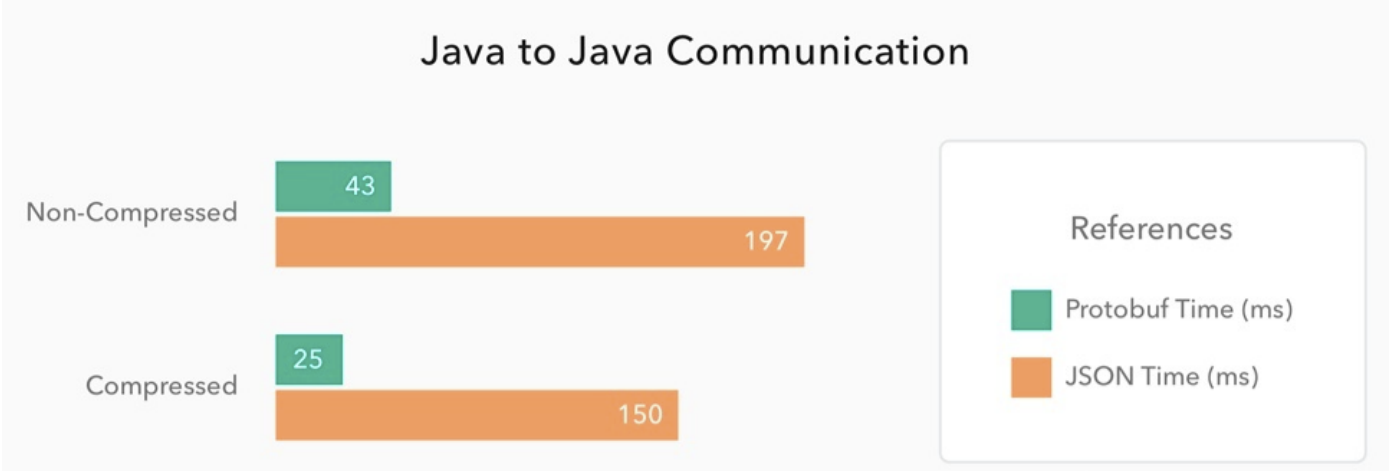
这些功能给设备带来重大益处，如节省带宽、降低 TCP 连接次数、节省 CPU 使用等，gRPC 既能够在客户端应用，也能够在服务器端应用，从而以透明的方式实现两端的通信和简化通信系统的构建。

HTTP 1.X 定义了四种与服务器交互的方式，分别为 GET、POST、PUT、DELETE，这些在 HTTP 2.0 中均保留，我们看看 HTTP 2.0 的**新特性**：双向流、多路复用、二进制帧、头部压缩。

2.6 性能对比

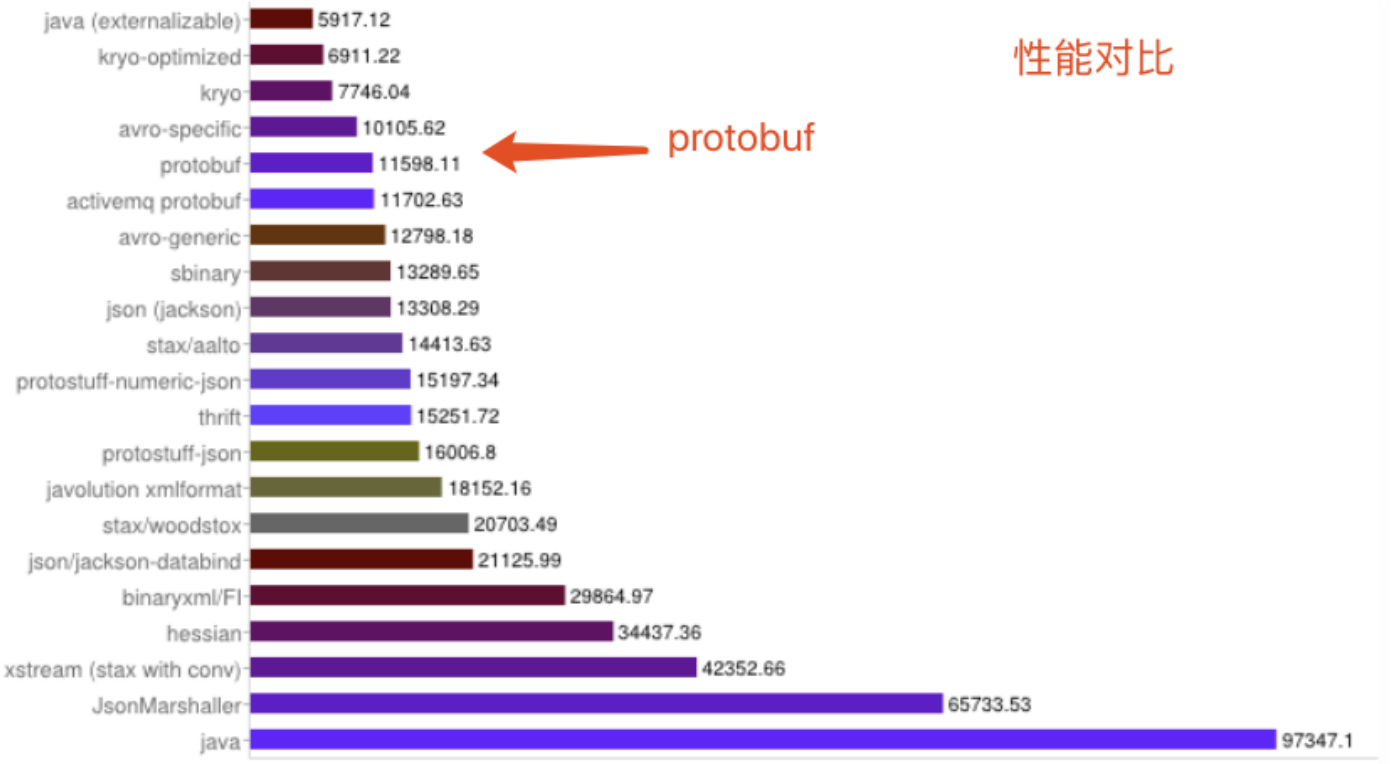
与采用文本格式的 JSON 相比，采用二进制格式的 protobuf 在速度上可以达到前者的 5 倍！

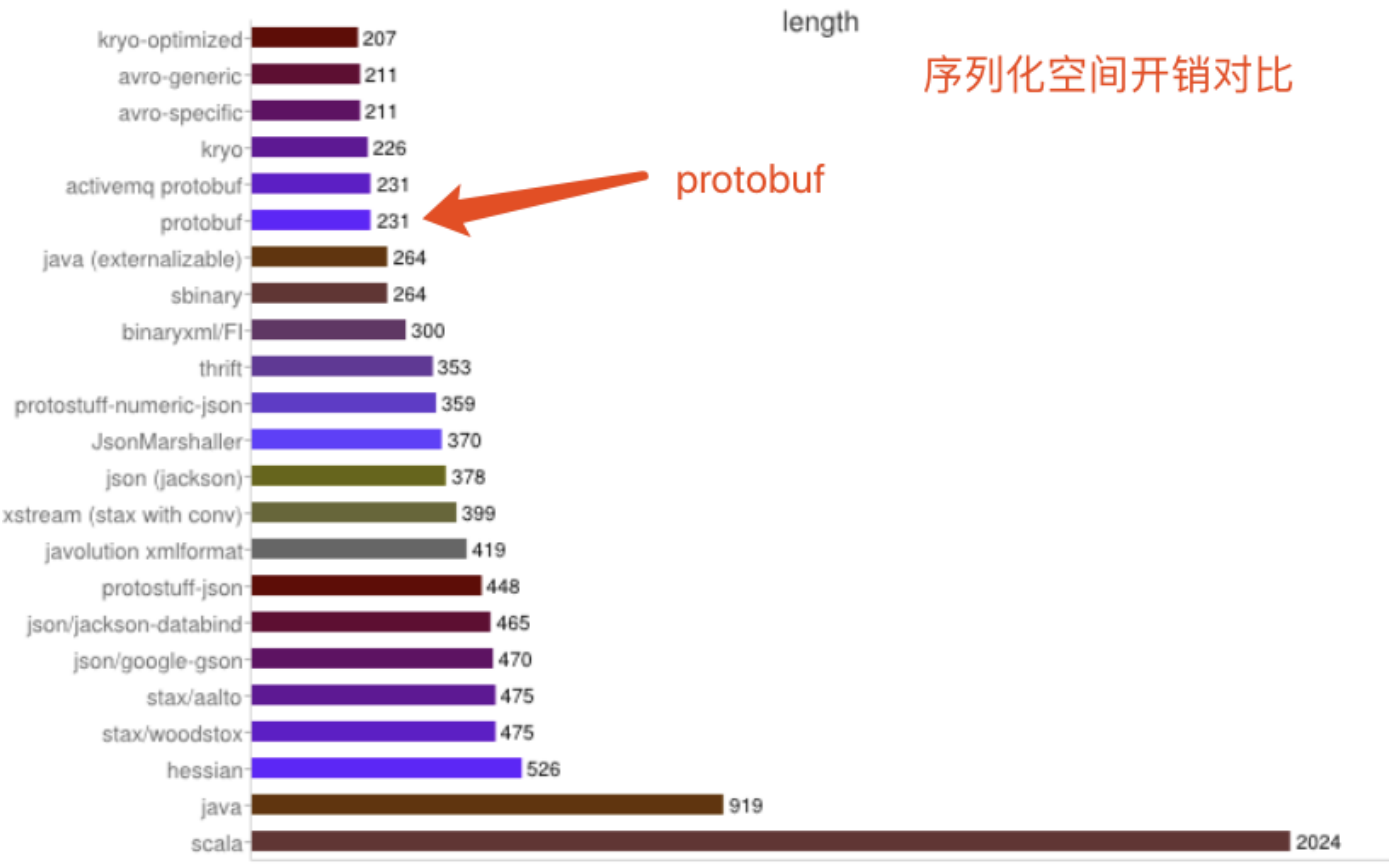
Auth0 网站所做的性能测试结果显示，protobuf 和 JSON 的优势差异在 Java、Python 等环境中尤为明显，下图是 Auth0 在两个 Spring Boot 应用程序间所做的对比测试结果。



结果显示，**protobuf 所需的请求时间最多只有 JSON 的 20% 左右，即速度是其 5 倍!**

下面看一下性能和空间开销对比。





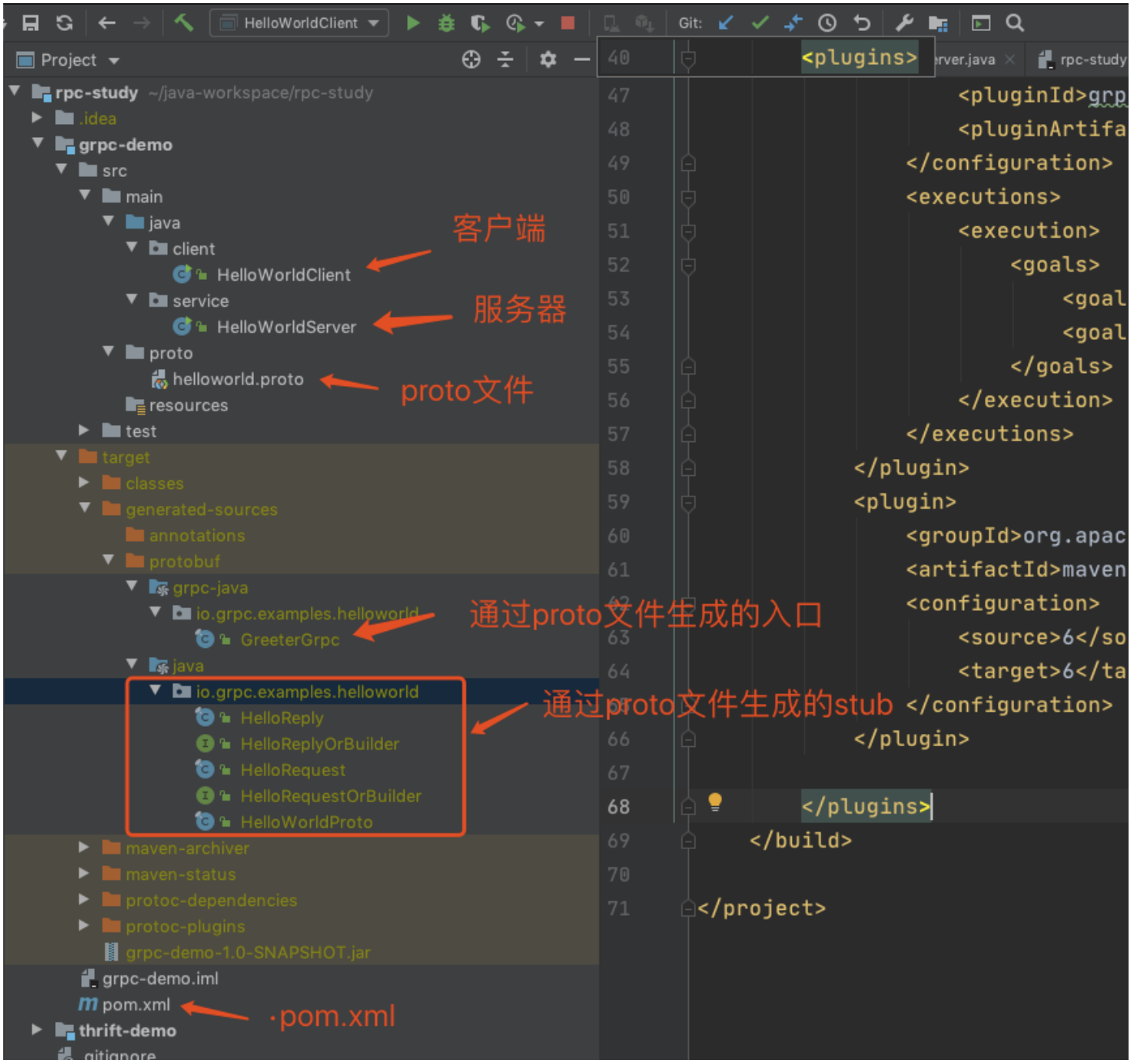
从上图可得出如下结论：

- XML序列化（Xstream）无论在性能和简洁性上比较差。
- Thrift 与 Protobuf 相比在时空开销方面都有一定的劣势。
- Protobuf 和 Avro 在两方面表现都非常优越。

3. gRPC 实战

3.1 项目结构

我们先看一下项目结构：



3.2 生成 protobuf 文件

文件 helloworld.proto：

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "io.grpc.examples.helloworld";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;
```

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

这里提供了一个 SayHello() 方法，然后入参为 HelloRequest，返回值为 HelloReply，可以看到 proto 文件只定义了入参和返回值的格式，以及调用的接口，至于接口内部的实现，该文件完全不用关心。

文件 pom.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>rpc-study</artifactId>
        <groupId>org.example</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>grpc-demo</artifactId>

    <dependencies>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-netty-shaded</artifactId>
            <version>1.14.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
            <artifactId>grpc-protobuf</artifactId>
            <version>1.14.0</version>
        </dependency>
        <dependency>
            <groupId>io.grpc</groupId>
```

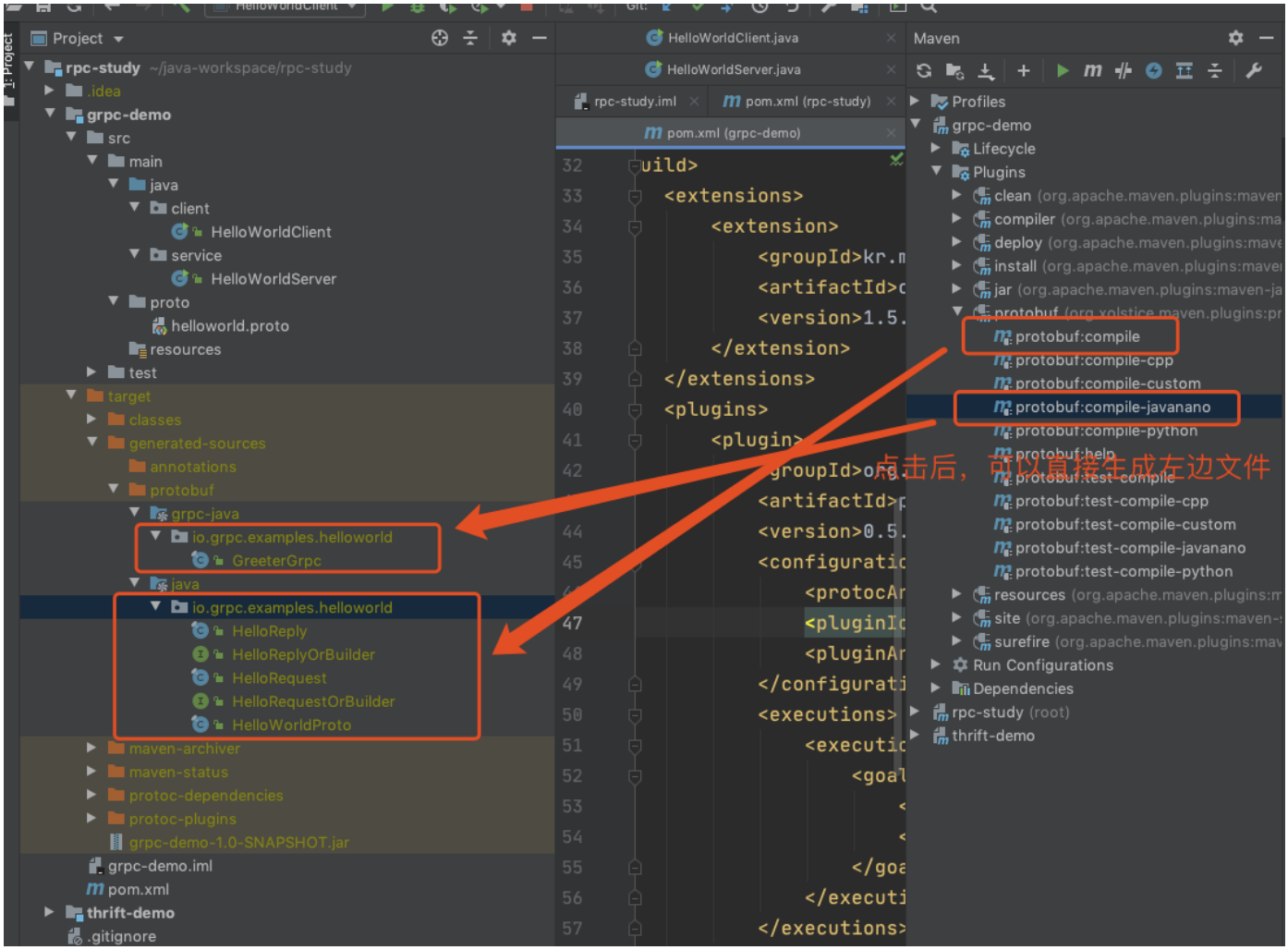
```
            <artifactId>grpc-stub</artifactId>
            <version>1.14.0</version>
        </dependency>
    </dependencies>

    <build>
        <extensions>
            <extension>
                <groupId>kr.motd.maven</groupId>
                <artifactId>os-maven-plugin</artifactId>
                <version>1.5.0.Final</version>
            </extension>
        </extensions>
        <plugins>
            <plugin>
                <groupId>org.xolstice.maven.plugins</groupId>
                <artifactId>protobuf-maven-plugin</artifactId>
                <version>0.5.1</version>
                <configuration>
                    <protocArtifact>com.google.protobuf:protoc:3.5.1-
1:exe:${os.detected.classifier}</protocArtifact>
                    <pluginId>grpc-java</pluginId>
                    <pluginArtifact>io.grpc:protoc-gen-grpc-
java:1.14.0:exe:${os.detected.classifier}</pluginArtifact>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>compile</goal>
                            <goal>compile-custom</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>6</source>
                    <target>6</target>
                </configuration>
            </plugin>

        </plugins>
    </build>
</project>
```

这里面的 build 其实是为了安装 protobuf 插件，里面其实有 2 个插件我们需要用到，分别为 protobuf:compile 和 protobuf:compile-javanano，当我们直接执行时，会生成左侧文件，其中 GreeterGrpc 提供调用接口，Hello 开头的文件功能主要是对数据进行序列化，然后处理入参和返回值。

> 可能有同学会问，你把文件生成到 target 中，我想放到 main.src 中，你可以把这些文件 copy 出来，或者也可以通过工具生成：
>
> - 下载 protoc.exe 工具 ， 下载地址：https://github.com/protocolbuffers/protobuf/releases
> - 下载 protoc-gen-grpc 插件, 下载地址: http://jcenter.bintray.com/io/grpc/protoc-gen-grpc-java/



3.3 服务端和客户端

文件 HelloWorldClient.java：

```
public class HelloWorldClient {
    private final ManagedChannel channel;
    private final GreeterGrpc.GreeterBlockingStub blockingStub;
    private static final Logger logger =
Logger.getLogger(HelloWorldClient.class.getName());

    public HelloWorldClient(String host,int port){
        channel = ManagedChannelBuilder.forAddress(host,port)
                .usePlaintext(true)
                .build();
```

```
        blockingStub = GreeterGrpc.newBlockingStub(channel);
    }


    public void shutdown() throws InterruptedException {
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
    }

    public  void greet(String name){
        HelloRequest request = HelloRequest.newBuilder().setName(name).build();
        HelloReply response;
        try{
            response = blockingStub.sayHello(request);
        } catch (StatusRuntimeException e)
        {
            logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
            return;
        }
        logger.info("Message from gRPC-Server: "+response.getMessage());
    }

    public static void main(String[] args) throws InterruptedException {
        HelloWorldClient client = new HelloWorldClient("127.0.0.1",50051);
        try{
            String user = "world";
            if (args.length > 0){
                user = args[0];
            }
            client.greet(user);
        }finally {
            client.shutdown();
        }
    }
}
```

这个太简单了，就是连接服务端口，调用 sayHello() 方法。

文件 HelloWorldServer.java：

```
public class HelloWorldServer {
    private static final Logger logger =
Logger.getLogger(HelloWorldServer.class.getName());

    private int port = 50051;
    private Server server;

    private void start() throws IOException {
        server = ServerBuilder.forPort(port)
```

```
                .addService(new GreeterImpl())
                .build()
                .start();
        logger.info("Server started, listening on " + port);

        Runtime.getRuntime().addShutdownHook(new Thread() {

            @Override
            public void run() {

                System.err.println("*** shutting down gRPC server since JVM is shutting
down");
                HelloWorldServer.this.stop();
                System.err.println("*** server shut down");
            }
        });
    }

    private void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    // block 一直到退出程序
    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException {
        final HelloWorldServer server = new HelloWorldServer();
        server.start();
        server.blockUntilShutdown();
    }

    // 实现 定义一个实现服务接口的类
    private class GreeterImpl extends GreeterGrpc.GreeterImplBase {
        @Override
        public void sayHello(HelloRequest req, StreamObserver<HelloReply>
responseObserver) {
            HelloReply reply = HelloReply.newBuilder().setMessage(("Hello " +
req.getName())).build();
            responseObserver.onNext(reply);
            responseObserver.onCompleted();
            System.out.println("Message from gRPC-Client:" + req.getName());
            System.out.println("Message Response:" + reply.getMessage());
        }
```

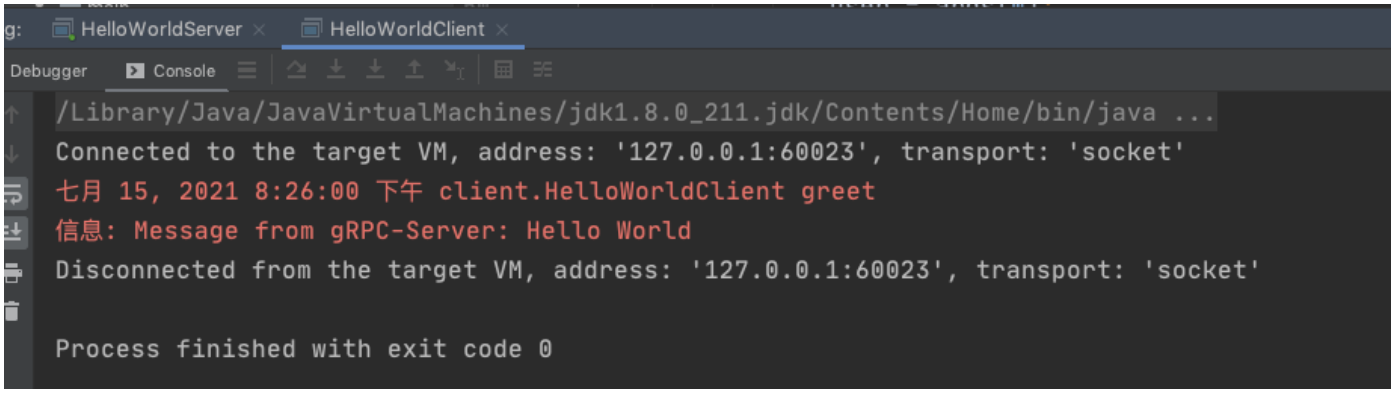
```
    }
}
```

主要是实现 sayHello() 方法，里面对数据进行了简单处理，入参为 “W orld”，返回的是 “Hello World”。

3.4 启动服务

先启动 Server，返回如下：

再启动 Client，返回如下：



同时 Server返回如下：

3.5 项目代码

Git 地址：https://github.com/lml200701158/rpc-study

4. 写在最后

这篇文章其实是我去年写的，这次是重新整理，**文章详细讲解了 RPC 和 gRPC，以及 gRPC 的应用示例**，非常全面，后面会再把 Thrift 整理出来。

这个 Demo 看起来很简单，我 TM 居然搞了大半天，一开始是因为不知道需要执行 2 个不同的插件来生成 protobuf，以为只需要点击 protobuf:compile 就可以，结果发现 protobuf:compile-javanano 也需要点一下。

还有就是我自己喜欢作，感觉通过插件生成 protobuf 不完美，我想通过自己下载的插件，手动生成 protobuf 文件，结果手动生成的没有搞定，自动生成的方式也不可用，搞了半天才发现是缓存的问题，最后直接执行 “Invalidate Caches / Restart” 才搞定。

应征了一句话“no zuo no die”，不过这个过程还是需要经历的。
