**微服务架构方案与技术介绍**

版本修订说明：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 时间 | 修改人 | 修改内容简述 | 审阅 |
| 1.0 | 2019-01-03 | 晁星宇 | 添加新内容 |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 目录 2](#_Toc18907)

[第一章 技术列表 4](#_Toc20940)

[第二章 spring cloud 框架 4](#_Toc16939)

[第三章 服务注册（netflix eureka） 5](#_Toc10283)

[第四章 配置中心（config） 6](#_Toc8637)

[第五章 服务治理（ribbon histrix） 7](#_Toc32642)

[第六章 服务网关（zuul） 8](#_Toc24371)

[第七章 微服务基础框架（spring boot） 9](#_Toc3031)

[第八章 日志采集（log4j2+kafka+elk） 9](#_Toc16055)

[第九章 日志分析（Logstash+Kibana+Elasticsearch） 10](#_Toc26591)

[第十章 链路跟踪（zipkin） 11](#_Toc8832)

[第十一章 分布式缓存（redis） 12](#_Toc7685)

[第十二章 分库分表（sharding-jdbc） 13](#_Toc44)

[第十三章 数据库管理系统（mariaDB） 14](#_Toc5832)

[第十四章 批量服务（spring batch） 14](#_Toc10801)

[第十五章 调度服务（elastic-job） 16](#_Toc32492)

[第十六章 监控告警（brave+zipkin/prometheus+grafana） 17](#_Toc21955)

[第十七章 实时计算引擎（JStorm /Spark Streaming） 20](#_Toc3156)

[第十八章 rpc网关（netty4） 21](#_Toc18643)

[第十九章 持续集成（Gitlab-CI+Kubernetes） 22](#_Toc32376)

## 微服务

### 微服务定义

微服务的概念源于 Martin Fowler 于 2014 年 3 月 25 日写的一篇文章 Microservices（

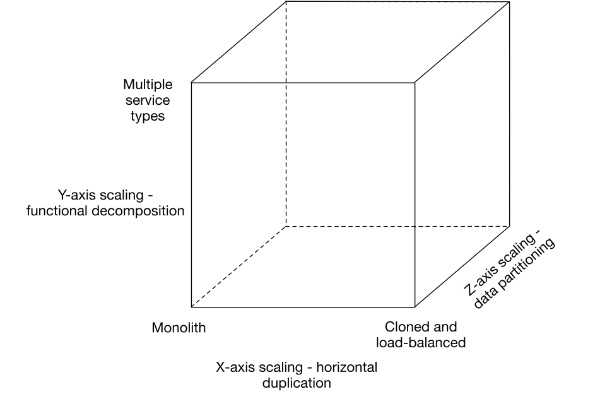
原文地址：<https://martinfowler.com/articles/microservices.html>

中文版翻译地址：<http://www.cnblogs.com/liuning8023/p/4493156.html>

）。从某种程度是可以简单的理解为，微服务是一个概念、一个项目开发的架构思想。

微服务架构模式（Microservices Architecture Pattern）的目的是将大型的、复杂的、长期运行的应用程序构建为一组相互配合的服务，每个服务都可以很容易得局部改良。 Micro 这个词意味着每个服务都应该足够小，但是，这里的小不能用代码量来比较，而应该是从业务逻辑上比较——符合 SRP 原则的才叫微服务。

关于微服务的一个形象表达：



X 轴：水平扩展，即在负载均衡服务器后增加多个运行实例

Z 轴：数据库的扩展，即分库分表

Y 轴：功能分解，即将不同职能的模块分成不同的服务

### 微服务架构优势

**复杂度可控：**在将应用分解的同时，规避了原本复杂度无止境的积累。每一个微服务专注于单一功能，并通过定义良好的接口清晰表述服务边界。由于体积小、复杂度低，每个微服务可由一个小规模开发团队完全掌控，易于保持高可维护性和开发效率。

**独立部署：**由于微服务具备独立的运行进程，所以每个微服务也可以独立部署。当某个微服务发生变更时无需编译、部署整个应用。由微服务组成的应用相当于具备一系列可并行的发布流程，使得发布更加高效，同时降低对生产环境所造成的风险，最终缩短应用交付周期。

**技术选型灵活：**微服务架构下，技术选型是去中心化的。每个团队可以根据自身服务的需求和行业发展的现状，自由选择最适合的技术栈。由于每个微服务相对简单，故需要对技术栈进行升级时所面临的风险就较低，甚至完全重构一个微服务也是可行的。

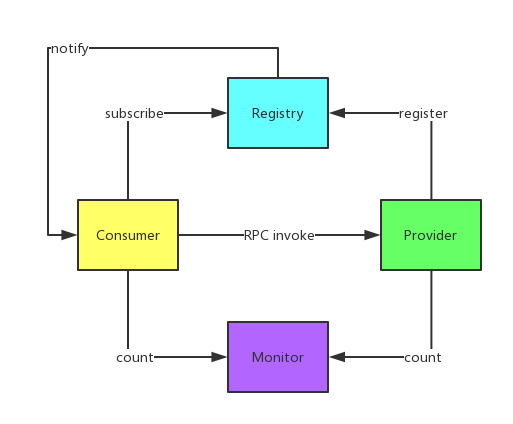
**容错（fault isolation）：**当某一组建发生故障时，在单一进程的传统架构下，故障很有可能在进程内扩散，形成应用全局性的不可用。在微服务架构下，故障会被隔离在单个服务中。若设计良好，其他服务可通过重试、平稳退化等机制实现应用层面的容错。

**扩展：**每个服务可以各自进行 x 扩展和 z 扩展，而且，每个服务可以根据自己的需要部署到合适的硬件服务器上。当应用的不同组件在扩展需求上存在差异时，微服务架构便体现出其灵活性，因为每个服务可以根据实际需求独立进行扩展。

## 基于 Dubbo 实现微服务

Dubbo 是阿里开源的一个 SOA 服务治理解决方案，文档丰富，在国内的使用度非常高。

Dubbo 服务调用流程：



* 调用中间层变成了可选组件，消费者可以直接访问服务提供者。
* 服务信息被集中到 Registry 中，形成了服务治理的中心组件。
* 通过 Monitor 监控系统，可以直观地展示服务调用的统计信息。
* Consumer 可以进行负载均衡、服务降级的选择。

但是对于微服务架构而言，Dubbo 也并不是十全十美的：

* Registry 严重依赖第三方组件（zookeeper 或者 redis），当这些组件出现问题时，服务调用很快就会中断。
* Dubbo 只支持 RPC 调用。使得服务提供方与调用方在代码上产生了强依赖，服务提供者需要不断将包含公共代码的 jar 包打包出来供消费者使用。一旦打包出现问题，就会导致服务调用出错。

## Spring Cloud 与 Dubbo 对比

作为新一代的服务框架，Spring Cloud 提出的口号是开发 “面向云环境的应用程序”，它为微服务架构提供了更加全面的技术支持。

根据微服务架构在各方面的要素，我们把 Spring Cloud 与 Dubbo 进行一番对比。

其实把 Spring Cloud 和 Dubbo 放一起对比有点不公平，Dubbo 只是实现了服务治理，而 Spring Cloud 下面有 23 个子项目（截止到 2018.4）分别覆盖了微服务架构下的方方面面，服务治理只是其中的一个方面，一定程度来说，Dubbo 只是 Spring Cloud Netflix 中的一个子集。但是在选择框架上，方案完整度恰恰是一个需要重点关注的内容。

|  |  |  |
| --- | --- | --- |
|  | **Dubbo** | **Spring Cloud** |
| 服务注册中心 | Zookeeper | Spring Cloud Netflix Eureka  Spring Cloud Consul |
| 服务调用方式 | RPC | REST API |
| 服务监控 | Dubbo-monitor | Spring Boot Admin |
| 断路器 | 不完善 | Spring Cloud Netflix Hystrix  Alibaba Sentinel  Resilience4J |
| 服务网关 | 无 | Spring Cloud Netflix Zuul  Spring Cloud Gateway  Linkerd |
| 分布式配置 | 无 | Spring Cloud Config |
| 服务跟踪 | 无 | Spring Cloud Sleuth |
| 消息总线 | 无 | Spring Cloud Bus |
| 数据流 | 无 | Spring Cloud Stream |
| 批量任务 | 无 | Spring Cloud Task |
| …… | …… | …… |

服务调用方式：Spring Cloud 抛弃了 Dubbo 的 RPC 通信，采用的是基于 HTTP 的 REST 方式。严格来说，这两种方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生 RPC 带来的问题。而且 REST 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

服务注册和发现：Eureka 相比于 zookeeper，更加适合于服务发现的场景，这点会在下一篇会详细展开。

Dubbo 的功能只是 Spring Cloud 体系的一部分。Spring Cloud 的功能更加强大，涵盖面更广，而且作为 Spring 的拳头项目，它也能够与 Spring Framework、Spring Boot、Spring Data、Spring Batch 等其他 Spring 项目完美融合，这些对于微服务而言是至关重要的。前面提到，微服务背后一个重要的理念就是持续集成、快速交付，而在服务内部使用一个统一的技术框架，显然比把分散的技术组合到一起更有效率。

## Spring Cloud 框架

**说明：**Spring Cloud 是一系列框架的有序集合，它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用 Spring Boot 的开发风格做到一键启动和部署。Spring 并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过 Spring Boot 风格进行再封装、屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

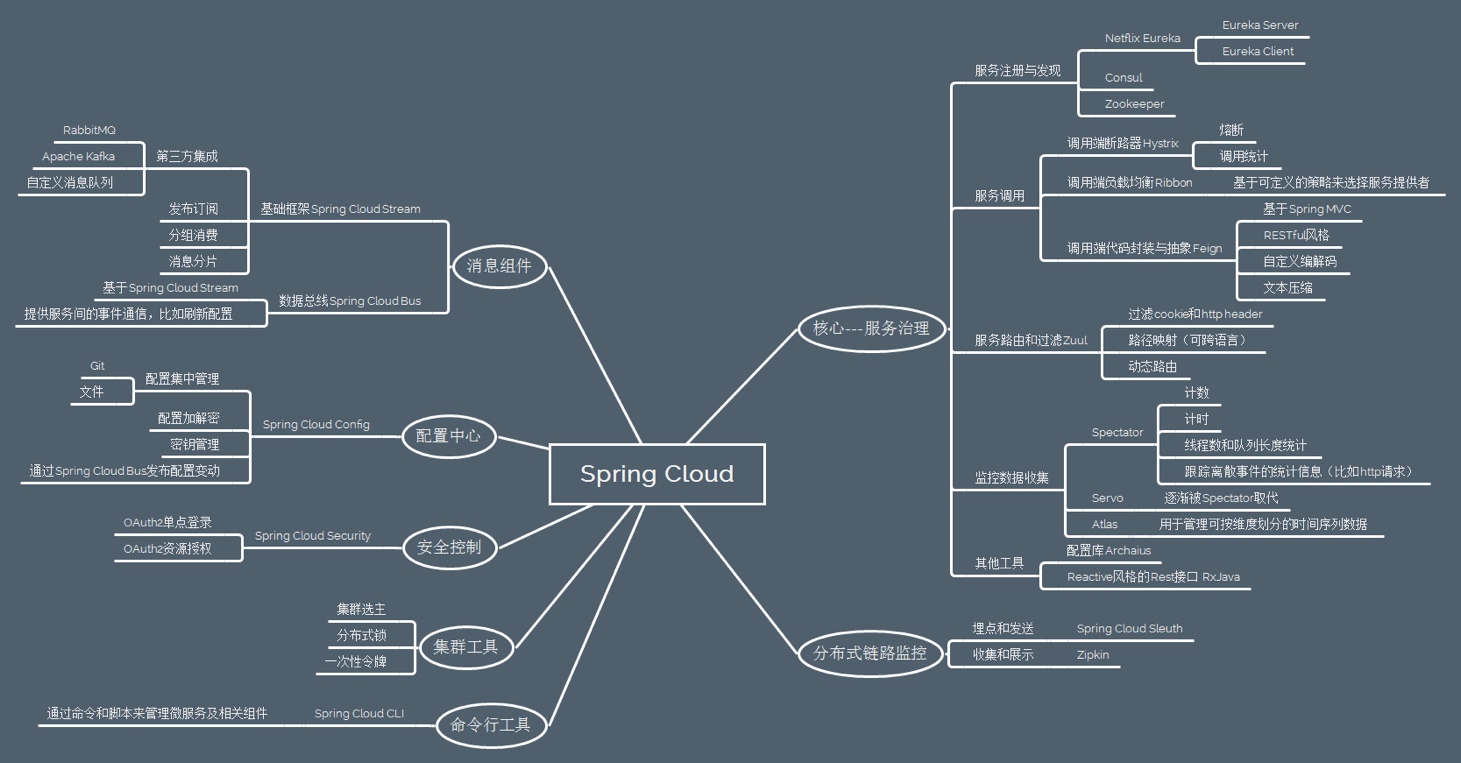
Spring Cloud/Spring Boot兼容性（2019-01-03）：

|  |  |
| --- | --- |
| **Spring Cloud版本** | **Spring Boot版本** |
| Greenwich | 2.1.x |
| Finchley | 2.0.x |
| Edgware | 1.5.x |
| Dalston | 1.5.x |

Spring Cloud 建议使用Finchley版本。

### Spring Cloud 技术概览

下图展示了 Spring Cloud 的完整技术组成：



**服务治理：**这是 Spring Cloud 的核心。目前 Spring Cloud 主要通过整合 Netflix 的相关产品来实现这方面的功能（Spring Cloud Netflix），包括用于服务注册和发现的 Eureka，调用断路器 Hystrix，调用端负载均衡 Ribbon，Rest 客户端 Feign，智能服务路由 Zuul，用于监控数据收集和展示的 Spectator、Servo、Atlas，用于配置读取的 Archaius 和提供 Controller 层 Reactive 封装的 RxJava。除此之外，针对于服务的注册和发现，除了 Eureka，Spring Cloud 也整合了 Consul 和 Zookeeper 作为备选，但是因为这两个方案在 CAP 理论上都遵循 CP 而不是 AP（下一篇会详细介绍这点），所以官方并没有推荐使用。Feign 和 RxJava 并不是 Netiflix 的产品，但是被整合到了 Spring Cloud Netflix 中。

**分布式链路监控：**Spring Cloud Sleuth 提供了全自动、可配置的数据埋点，以收集微服务调用链路上的性能数据，并发送给 Zipkin 进行存储、统计和展示。

**消息组件：**Spring Cloud Stream 对于分布式消息的各种需求进行了抽象，包括发布订阅、分组消费、消息分片等功能，实现了微服务之间的异步通信。Spring Cloud Stream 也集成了第三方的 RabbitMQ 和 Apache Kafka 作为消息队列的实现。而 Spring Cloud Bus 基于 Spring Cloud Stream，主要提供了服务间的事件通信（比如刷新配置）。

**配置中心：**基于 Spring Cloud Netflix 和 Spring Cloud Bus，Spring 又提供了 Spring Cloud Config，实现了配置集中管理、动态刷新的配置中心概念。配置通过 Git 或者简单文件来存储，支持加解密。

**安全控制：**Spring Cloud Security 基于 OAuth2 这个开放网络的安全标准，提供了微服务环境下的单点登录、资源授权、令牌管理等功能。

**命令行工具：**Spring Cloud Cli 提供了以命令行和脚本的方式来管理微服务及 Spring Cloud 组件的方式。

**集群工具：**Spring Cloud Cluster 提供了集群选主、分布式锁（暂未实现）、一次性令牌（暂未实现）等分布式集群需要的技术组件。

### 服务治理

#### 服务中心

**说明：**服务中心将所有的可以提供的服务都注册到它这里来管理，其它各调用者需要的时候去注册中心获取，然后再进行调用，避免了服务之间的直接调用，方便后续的水平扩展、故障转移等。

下面是 Spring Cloud 支持的服务发现软件以及特性对比：

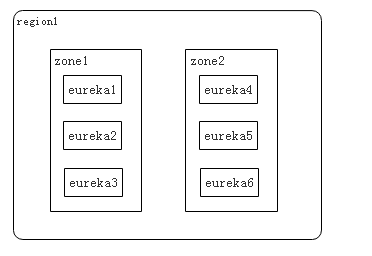
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | Euerka | **Consul** | **zookeeper** | **etcd** |
| 服务健康检查 | 可配支持 | 服务状态，  内存，硬盘等 | (弱)长连接、  keepalive | 连接心跳 |
| 多数据中心 | — | 支持 | — | — |
| kv 存储服务 | — | 支持 | 支持 | 支持 |
| 一致性 | — | raft | paxos | raft |
| cap | ap | ca | cp | cp |
| 使用接口  (多语言能力) | http（sidecar） | 支持 http、  dns | 客户端 | http/  grpc |
| watch 支持 | 支持 long polling  /大部分增量 | 全量/  支持long polling | 支持 | 支持long polling |
| 自身监控 | metrics | metrics | — | metrics |
| 安全 | — | acl /https | acl | https支持(弱) |
| spring cloud集成 | 已支持 | 已支持 | 已支持 | 已支持 |

在以上服务发现的软件中，Euerka 和 Consul 使用最为广泛。

##### Netflix Eureka

Eureka 是 Netflix 开发的服务发现组件，采用了 C-S 的设计架构，本身是一个基于REST的服务。Spring Cloud 将它集成在其子项目 spring-cloud-netflix 中，以实现 Spring Cloud 的服务发现功能。

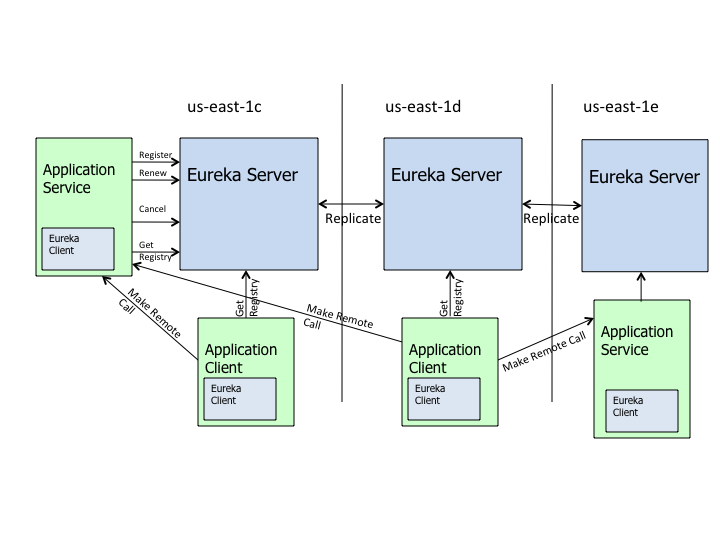
在分析 Eureka 原理之前，我们先来了解一下region、zone、Eureka集群三者的关系，如下图：



区域（region）和可用区（zone）（或者Availability Zone，AZ）均是AWS（亚马逊AWS）的概念。在非AWS环境下，我们可以简单地将region理解为Eureka集群，zone理解成机房。这样就很好理解了——一个Eureka集群被部署在了zone1机房和zone2机房中。Spring Cloud中默认的region是us-east-1 。**Eureka的一些概念**

* Register：服务注册 当Eureka客户端向Eureka Server注册时，它提供自身的元数据，比如IP地址、端口，运行状况指示符URL，主页等。
* Renew：服务续约 Eureka客户会每隔30秒发送一次心跳来续约。 通过续约来告知Eureka Server该Eureka客户仍然存在，没有出现问题。 正常情况下，如果Eureka Server在90秒没有收到Eureka客户的续约，它会将实例从其注册表中删除。 建议不要更改续约间隔。
* Fetch Registries：获取注册列表信息 Eureka客户端从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与Eureka客户端的缓存信息不同， Eureka客户端自动处理。如果由于某种原因导致注册列表信息不能及时匹配，Eureka客户端则会重新获取整个注册表信息。 Eureka服务器缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。Eureka客户端和Eureka 服务器可以使用JSON / XML格式进行通讯。在默认的情况下Eureka客户端使用压缩JSON格式来获取注册列表的信息。
* Cancel：服务下线 Eureka客户端在程序关闭时向Eureka服务器发送取消请求。 发送请求后，该客户端实例信息将从服务器的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：DiscoveryManager.getInstance().shutdownComponent()；
* Eviction 服务剔除 在默认的情况下，当Eureka客户端连续90秒没有向Eureka服务器发送服务续约，即心跳，Eureka服务器会将该服务实例从服务注册列表删除，即服务剔除。

**Eureka 官方架构图：**



Eureka官方的架构图，大致描述了Eureka集群的工作过程。图中包含的组件非常多，可能比较难以理解，我们用通俗易懂的语言解释一下：

* Application Service 相当于本书中的服务提供者，Application Client相当于本书中的服务消费者；
* Make Remote Call，可以简单理解为调用RESTful API；
* us-east-1c、us-east-1d等都是zone，它们都属于us-east-1这个region；

由图可知，Eureka包含两个组件：Eureka Server 和 Eureka Client，它们的作用如下：

* Eureka Client是一个Java客户端，用于简化与Eureka Server的交互；
* Eureka Server提供服务发现的能力，各个微服务启动时，会通过Eureka Client向Eureka Server进行注册自己的信息（例如网络信息），Eureka Server会存储该服务的信息；
* 微服务启动后，会周期性地向Eureka Server发送心跳（默认周期为30秒）以续约自己的信息。如果Eureka Server在一定时间内没有接收到某个微服务节点的心跳，Eureka Server将会注销该微服务节点（默认90秒）；
* 每个Eureka Server同时也是Eureka Client，多个Eureka Server之间通过复制的方式完成服务注册表的同步；
* Eureka Client会缓存Eureka Server中的信息。即使所有的Eureka Server节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者。

综上，Eureka通过心跳检测、健康检查和客户端缓存等机制，提高了系统的灵活性、可伸缩性和可用性。

##### 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）。使用起来也较 为简单。Consul 使用 Go 语言编写，因此具有天然可移植性(支持Linux、windows和Mac OS X)；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 的优势：**

* 使用 Raft 算法来保证一致性, 比复杂的 Paxos 算法更直接. 相比较而言, zookeeper 采用的是 Paxos, 而 etcd 使用的则是 Raft。
* 支持多数据中心，内外网的服务采用不同的端口进行监听。 多数据中心集群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。 zookeeper 和 etcd 均不提供多数据中心功能的支持。
* 支持健康检查。 etcd 不提供此功能。
* 支持 http 和 dns 协议接口。 zookeeper 的集成较为复杂, etcd 只支持 http 协议。
* 官方提供 web 管理界面, etcd 无此功能。
* 综合比较, Consul 作为服务注册和配置管理的新星, 比较值得关注和研究。

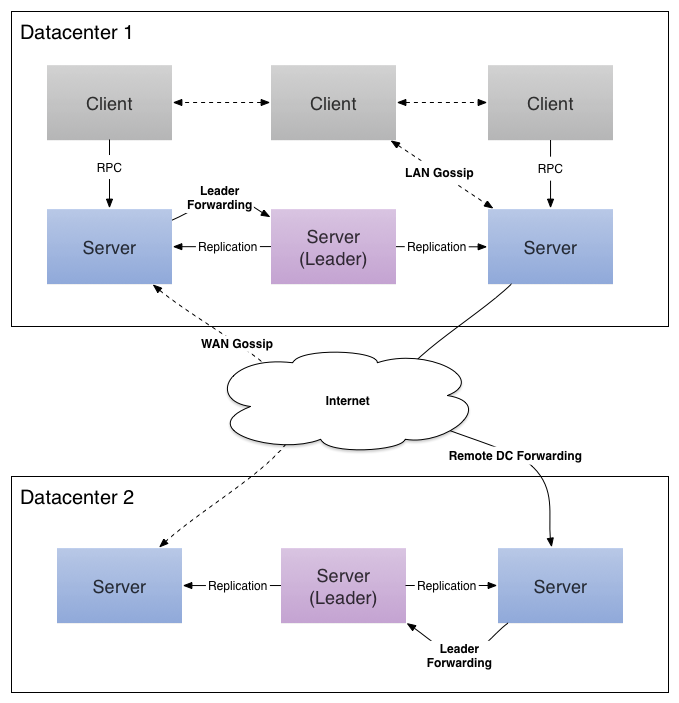
**特性：**

* 服务发现
* 健康检查
* Key/Value 存储
* 多数据中心

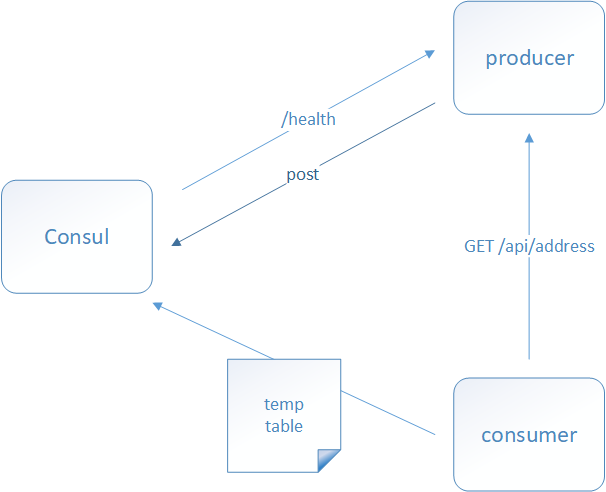
**Consul 角色**

* client: 客户端, 无状态, 将 HTTP 和 DNS 接口请求转发给局域网内的服务端集群。
* server: 服务端, 保存配置信息, 高可用集群, 在局域网内与本地客户端通讯, 通过广域网与其它数据中心通讯。 每个数据中心的 server 数量推荐为 3 个或是 5 个。

Consul 客户端、服务端还支持夸中心的使用，更加提高了它的高可用性。



**Consul 工作原理：**



a、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

b、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

c、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

d、该临时表每隔10s会更新，只包含有通过了健康检查的 Producer

Spring Cloud Consul 项目是针对 Consul 的服务治理实现。Consul 是一个分布式高可用的系统，它包含多个组件，但是作为一个整体，在微服务架构中为我们的基础设施提供服务发现和服务配置的工具。

**Consul 安装:**

Consul 不同于 Eureka 需要单独安装，访问Consul 官网（https://www.consul.io/downloads.html）下载 Consul 的最新版本，根据不同的系统类型选择不同的安装包。

以 Windows 为例，下载下来是一个 consul\_1.4.0\_windows\_amd64.zip 的压缩包，解压是是一个 consul.exe 的执行文件。

cd 到解压后consul.exe的目录下，使用 cmd 启动 Consul

cd D:\Common Files\consul

#cmd启动

.\consul agent -dev # -dev表示开发模式运行，另外还有-server表示服务模式运行

可以在同级目录下创建一个 run.bat 脚本来启动，脚本内容如下：

.\consul agent -dev

pause

启动成功之后访问：http://localhost:8500，可以看到 Consul 的管理界面

##### Eureka VS Consul

Eureka 是一个服务发现工具。该体系结构主要是客户端/服务器，每个数据中心有一组 Eureka 服务器，通常每个可用区域一个。通常 Eureka 的客户使用嵌入式 SDK 来注册和发现服务。对于非本地集成的客户，官方提供的 Eureka 一些 REST 操作 API，其它语言可以使用这些 API 来实现对 Eureka Server 的操作从而实现一个非 jvm 语言的 Eureka Client。

Eureka 提供了一个弱一致的服务视图，尽可能的提供服务可用性。当客户端向服务器注册时，该服务器将尝试复制到其它服务器，但不提供保证复制完成。服务注册的生存时间（TTL）较短，要求客户端对服务器心跳检测。不健康的服务或节点停止心跳，导致它们超时并从注册表中删除。服务发现可以路由到注册的任何服务，由于心跳检测机制有时间间隔，可能会导致部分服务不可用。这个简化的模型允许简单的群集管理和高可扩展性。

Consul 提供了一些列特性，包括更丰富的健康检查，键值对存储以及多数据中心。Consul 需要每个数据中心都有一套服务，以及每个客户端的 agent，类似于使用像 Ribbon 这样的服务。Consul agent 允许大多数应用程序成为 Consul 不知情者，通过配置文件执行服务注册并通过 DNS 或负载平衡器 sidecars 发现。

Consul 提供强大的一致性保证，因为服务器使用 Raft 协议复制状态 。Consul 支持丰富的健康检查，包括 TCP，HTTP，Nagios / Sensu 兼容脚本或基于 Eureka 的 TTL。客户端节点参与基于 Gossip 协议的健康检查，该检查分发健康检查工作，而不像集中式心跳检测那样成为可扩展性挑战。发现请求被路由到选举出来的 leader，这使他们默认情况下强一致性。允许客户端过时读取取使任何服务器处理他们的请求，从而实现像 Eureka 这样的线性可伸缩性。

Consul 强烈的一致性意味着它可以作为领导选举和集群协调的锁定服务。Eureka 不提供类似的保证，并且通常需要为需要执行协调或具有更强一致性需求的服务运行 ZooKeeper。

Consul 提供了支持面向服务的体系结构所需的一系列功能。这包括服务发现，还包括丰富的运行状况检查，锁定，密钥/值，多数据中心联合，事件系统和 ACL。Consul 和 consul-template 和 envconsul 等工具生态系统都试图尽量减少集成所需的应用程序更改，以避免需要通过 SDK 进行本地集成。Eureka 是一个更大的 Netflix OSS 套件的一部分，该套件预计应用程序相对均匀且紧密集成。因此 Eureka 只解决了一小部分问题，可以和 ZooKeeper 等其它工具可以一起使用。

**Consul 强一致性(C)带来的是：**

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

**Eureka 保证高可用(A)和最终一致性：**

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

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

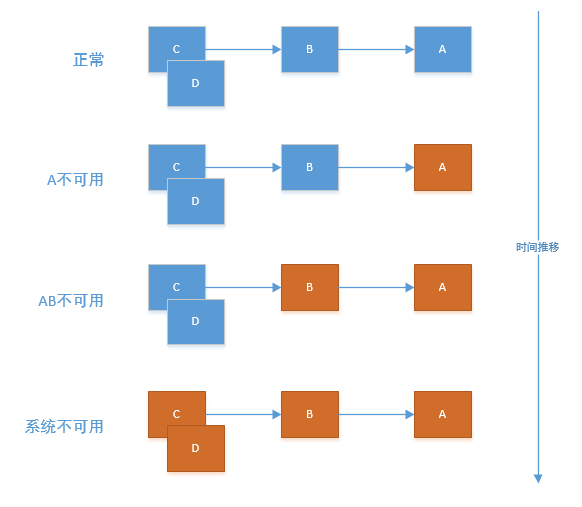
#### 服务调用

**说明：**断路器可选择的有 Hystrix、Alibaba Sentinel（阿里巴巴开源的一款断路器实现）与 Resilience4J（ Hystrix 官方推荐的替代产品）。

##### 断路器（Hystrix）

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障，进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



服务雪崩效应产生原因：

* 服务提供者不可用

**硬件故障** 可能为硬件损坏造成的服务器主机宕机，网络硬件故障造成的服务提供者的不可访问。

**程序 Bug**

**缓存击穿** 一般发生在缓存应用重启，所有缓存被清空时，以及短时间内大量缓存失效时。大量的缓存不命中，使请求直击后端，造成服务提供者超负荷运行，引起服务不可用。

**用户大量请求** 秒杀和大促开始前，如果准备不充分，用户发起大量请求也会造成服务提供者的不可用。

* 重试加大流量

**用户重试** 在服务提供者不可用后，用户由于忍受不了界面上长时间的等待，而不断刷新页面甚至提交表单。

**代码逻辑重试** 服务调用端的会存在大量服务异常后的重试逻辑。

* 服务调用者不可用

**同步等待造成的资源耗尽**  当服务调用者使用 **同步调用** 时，会产生大量的等待线程占用系统资源。一旦线程资源被耗尽，服务调用者提供的服务也将处于不可用状态。

**应对策略**

针对造成服务雪崩的不同原因，可以使用不同的应对策略:

* 流量控制

**网关限流** 因为 Nginx 的高性能，目前一线互联网公司大量采用 Nginx+Lua 的网关进行流量 控制，由此而来的 OpenResty 也越来越热门。

**用户交互限流** 采用加载动画，提高用户的忍耐等待时间；提交按钮添加强制等待时间机制。

**关闭重试**

* 改进缓存模式

**缓存预加载**

**同步改为异步刷新**

* 服务自动扩容

**AWS 的 auto scaling**

* 服务调用者降级服务

**资源隔离** 对调用服务的线程池进行隔离。

**对依赖服务进行分类** 根据具体业务，将依赖服务分为: 强依赖和若依赖。强依赖服务不可用

会导致当前业务中止，而弱依赖服务的不可用不会导致当前业务的中止。

**不可用服务的调用快速失败** 通过 超时机制, 熔断器 和熔断后的 降级方法 来实现。

为了应对服务雪崩，一种常见的做法是手动服务降级。而 Hystrix 的出现，给我们提供了另一种选择。

Hystrix [hɪst’rɪks] 的中文含义是 “豪猪”，豪猪周身长满了刺，能保护自己不受天敌的伤害，代表了一种防御机制，这与 Hystrix 本身的功能不谋而合，因此 Netflix 团队将该框架命名为 Hystrix，并使用了对应的卡通形象做作为 logo。

**Hystrix特性**

* 服务降级（Fallback）

Fallback相当于是降级操作. 对于查询操作, 我们可以实现一个fallback方法, 当请求后端服务出现

异常的时候, 可以使用fallback方法返回的值. fallback方法的返回值一般是设置的默认值或者来自缓存.

* 资源隔离

在 Hystrix 中，主要通过线程池来实现资源隔离。通常在使用的时候我们会根据调用的远程服务划

分出多个线程池。例如调用产品服务的 Command 放入 A 线程池，调用账户服务的 Command 放入

B 线程池。这样做的主要优点是运行环境被隔离开了。这样就算调用服务的代码存在 bug 或者由于其

他原因导致自己所在线程池被耗尽时，不会对系统的其他服务造成影响。

通过对依赖服务的线程池隔离实现，可以带来如下优势：

应用自身得到完全的保护，不会受不可控的依赖服务影响。即便给依赖服务分配的线程池被填满，

也不会影响应用自身的额其余部分；

可以有效的降低接入新服务的风险。如果新服务接入后运行不稳定或存在问题，完全不会影响到应

用其他的请求；

当依赖的服务从失效恢复正常后，它的线程池会被清理并且能够马上恢复健康的服务，相比之下容

器级别的清理恢复速度要慢得多；

当依赖的服务出现配置错误的时候，线程池会快速的反应出此问题（通过失败次数、延迟、超时、

拒绝等指标的增加情况）。同时，我们可以在不影响应用功能的情况下通过实时的动态属性刷新（后续

会通过 Spring Cloud Config 与 Spring Cloud Bus 的联合使用来介绍）来处理它；

当依赖的服务因实现机制调整等原因造成其性能出现很大变化的时候，此时线程池的监控指标信息

会反映出这样的变化。同时，我们也可以通过实时动态刷新自身应用对依赖服务的阈值进行调整以适应

依赖方的改变；

除了上面通过线程池隔离服务发挥的优点之外，每个专有线程池都提供了内置的并发实现，可以利

用它为同步的依赖服务构建异步的访问。

总之，通过对依赖服务实现线程池隔离，让我们的应用更加健壮，不会因为个别依赖服务出现问题而引

起非相关服务的异常。同时，也使得我们的应用变得更加灵活，可以在不停止服务的情况下，配合动态配置

刷新实现性能配置上的调整。

Hystrix 中除了使用线程池之外，还可以使用信号量来控制单个依赖服务的并发度，信号量的开销要远

比线程池的开销小得多，但是它不能设置超时和实现异步访问。所以，只有在依赖服务是足够可靠的情况下

才使用信号量。在 HystrixCommand 和 HystrixObservableCommand 中 2 处支持信号量的使用：

**命令执行：**如果隔离策略参数 execution.isolation.strategy 设置为 SEMAPHORE，Hystrix 会使用

信号量替代线程池来控制依赖服务的并发控制。

**降级逻辑：**当 Hystrix 尝试降级逻辑时候，它会在调用线程中使用信号量。

信号量的默认值为 10，我们也可以通过动态刷新配置的方式来控制并发线程的数量。对于信号量大小

的估算方法与线程池并发度的估算类似。仅访问内存数据的请求一般耗时在 1ms 以内，性能可以达到

5000rps，这样级别的请求我们可以将信号量设置为 1 或者 2，我们可以按此标准并根据实际请求耗时来设

置信号量。

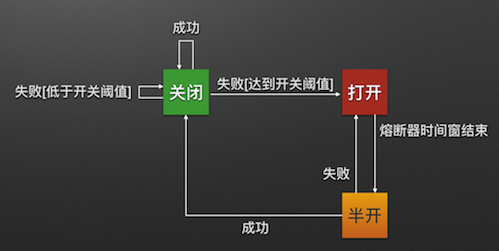
* 断路器模式

在分布式架构中，断路器模式的作用，当某个服务单元发生故障（类似用电器发生短路）之后，通

过断路器的故障监控（类似熔断保险丝），直接切断原来的主逻辑调用。但是，在 Hystrix 中的断路器

除了切断主逻辑的功能之外，还有更复杂的逻辑，下面我们来看看它更为深层次的处理逻辑。

断路器开关相互转换的逻辑如下图：



当 Hystrix Command 请求后端服务失败数量超过一定阈值，断路器会切换到开路状态 (Open)。这

时所有请求会直接失败而不会发送到后端服务。

这个阈值涉及到三个重要参数：快照时间窗、请求总数下限、错误百分比下限。参数作用分别是：

**快照时间窗：**断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间

窗，默认为最近的 10 秒。

**请求总数下限：**在快照时间窗内，必须满足请求总数下限才有资格进行熔断。默认为 20，意味着在

10 秒内，如果该 Hystrix Command 的调用此时不足 20 次，即时所有的请求都超时或其他原因失败，

断路器都不会打开。

**错误百分比下限：**当请求总数在快照时间窗内超过了下限，比如发生了 30 次调用，如果在这 30

次调用中，有 16 次发生了超时异常，也就是超过 50% 的错误百分比，在默认设定 50% 下限情况

下，这时候就会将断路器打开。

断路器保持在开路状态一段时间后 (默认 5 秒)，自动切换到半开路状态 (HALF-OPEN)。这时会判

断下一次请求的返回情况，如果请求成功，断路器切回闭路状态 (CLOSED)，否则重新切换到开路状态

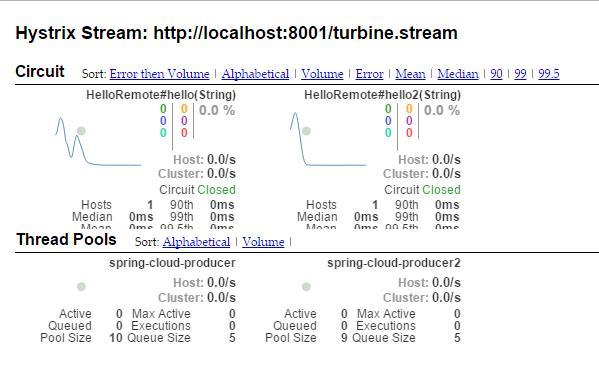
(OPEN)。

##### 熔断监控Hystrix Dashboard和Turbine

当熔断发生的时候需要迅速的响应来解决问题，避免故障进一步扩散，那么对熔断的监控就变得非常重要。熔断的监控现在有两款工具：Hystrix-dashboard 和 Turbine

Hystrix-dashboard 是一款针对 Hystrix 进行实时监控的工具，通过 Hystrix Dashboard 我们可以直观地看到各 Hystrix Command 的请求响应时间, 请求成功率等数据。但是只使用 Hystrix Dashboard 的话, 你只能看到单个应用内的服务信息, 这明显不够. 我们需要一个工具能让我们汇总系统内多个服务的数据并显示到 Hystrix Dashboard 上, 这个工具就是 Turbine.

监控的效果图如下：



##### 负载均衡（ribbon）

在微服务架构中，业务都会被拆分成一个独立的服务，服务与服务的通讯是基于http restful的。Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign。

Ribbon 是 Netflix 开源的基于 HTTP 和 TCP 的客户端负载均衡器框架，目前也已被 Spring Cloud 团队集成在 spring-cloud-netflix 子项目下，主要用于客户端软负载功能，内部已实现了 随机、轮训、权重、减压（选取压力最小的） 等常见的负载算法，同时也提供了 ILoadBalance 与 IRule 两个接口方便我们自己编写适合自己的负载算法。

**Ribbon特性：**

* 负载均衡
* 容错
* 多协议（HTTP，TCP，UDP）支持异步和反应模型
* 缓存和批处理

Ribbon 交互图：

注册中心

restTemplate

服务提供者1

服务消费者

Ribbon

服务提供者2

##### 服务消费者（Feign）

注解式的 Feign 使得 Java HTTP 客户端编写更方便。Feign 灵感来源于安卓网络编程框架 Retrofit、JAXRS-2.0 和 WebSocket，支持可插拔编码器和解码器，降低 HTTP API 的复杂度，通过最少的资源和代码来实现和 HTTP API 的连接。通过可定制的解码器和错误处理，可以编写任意的HTTP API。Spring Cloud Feign 封装了 Ribbon 这一组件，所以在使用 Feign 同时还能提供负载均衡的功能，这一切只需要一个 @FeignClient 即可完成。

早期版本的 Feign 被 Spring Cloud 团队集成在 spring-cloud-netflix 子项目下，但如今 Spring Cloud 团队将 Spring Cloud Feign 独立成一个单独的 spring-cloud-openfeign 项目。

##### 服务路由（网关）

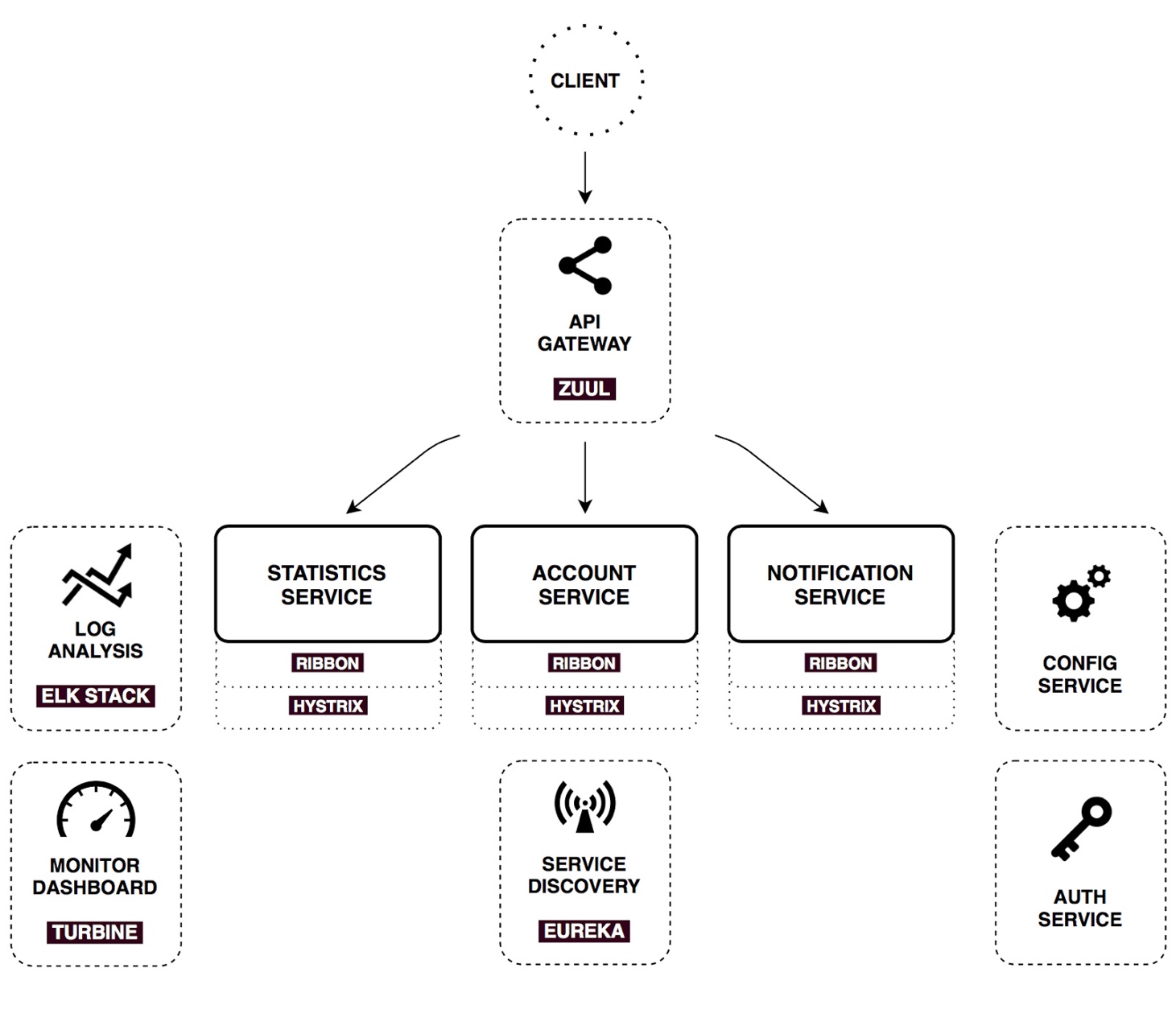
传统路由配置方式就是在不依赖于服务发现机制的情况下，通过在配置文件中具体指定每个路由表达式与服务实例的映射关系来实现API网关对外部请求的路由。

在微服务架构中，后端服务往往不直接开放给调用端，而是通过一个 API 网关根据请求的 URL，路由到相应的服务。API 网关是介于客户端和服务器端之间的中间层，所有的外部请求都会先经过 API 网关这一层。也就是说，API 的实现方面更多的考虑业务逻辑，而安全、性能、监控可以交由 API 网关来做，这样既提高业务灵活性又不缺安全性。

Spring Cloud 的网关选择可有 Spring Cloud Zuul（Zuul 1.x，是一个基于阻塞io的API Gateway） 、Spring Cloud Gateway（基于Netty，是非阻塞的）、Linkerd （基于Scala实现的）三种选择。

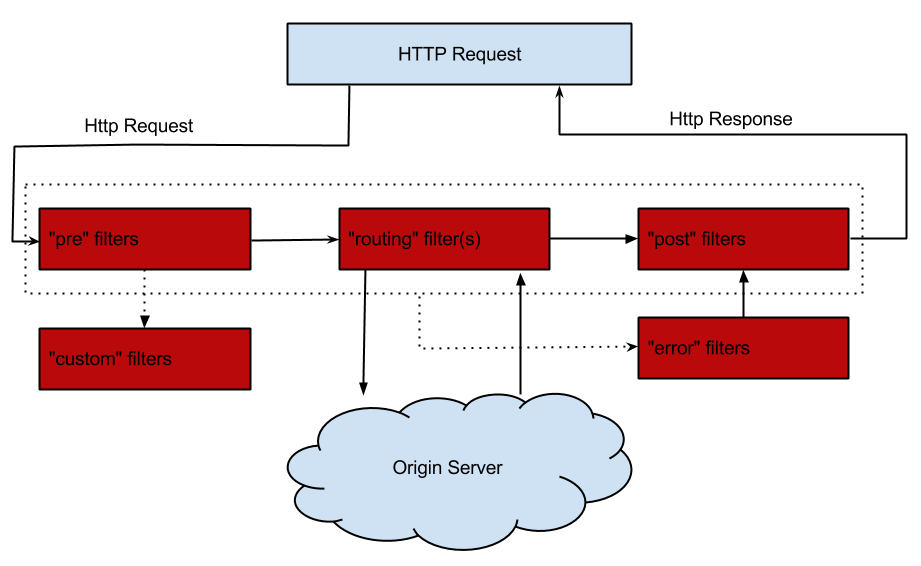
##### Spring Cloud Zuul

Spring Cloud Zuul 路由是微服务架构的不可或缺的一部分，提供动态路由、监控、弹性、安全等的边缘服务。Zuul 是 Netflix 出品的一个基于 JVM 路由和服务端的负载均衡器。Zuul（1.x） 基于 Servlet，使用阻塞 API，它不支持任何长连接，如 WebSockets。



**Zuul的核心**

Filter 是Zuul的核心，用来实现对外服务的控制。Filter的生命周期有4个，分别是“PRE”、“ROUTING”、“POST”、“ERROR”，整个生命周期可以用下图来表示。



Zuul大部分功能都是通过过滤器来实现的，这些过滤器类型对应于请求的典型生命周期。

* PRE： 这种过滤器在请求被路由之前调用。我们可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。
* ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用Apache HttpClient或Netfilx Ribbon请求微服务。
* POST：这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。
* ERROR：在其他阶段发生错误时执行该过滤器。 除了默认的过滤器类型，Zuul还允许我们创建自定义的过滤器类型。例如，我们可以定制一种STATIC类型的过滤器，直接在Zuul中生成响应，而不将请求转发到后端的微服务。

Zuul中默认实现的Filter：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **顺序** | **过滤器** | **功能** |
| pre | -3 | ServletDetectionFilter | 标记处理Servlet的类型 |
| pre | -2 | Servlet30WrapperFilter | 包装HttpServletRequest请求 |
| pre | -1 | FormBodyWrapperFilter | 包装请求体 |
| route | 1 | DebugFilter | 标记调试标志 |
| route | 5 | PreDecorationFilter | 处理请求上下文供后续使用 |
| route | 10 | RibbonRoutingFilter | serviceId请求转发 |
| route | 100 | SimpleHostRoutingFilter | url请求转发 |
| route | 500 | SendForwardFilter | forward请求转发 |
| post | 0 | SendErrorFilter | 处理有错误的请求响应 |
| post | 1000 | SendResponseFilter | 处理正常的请求响应 |

**禁用指定的Filter**

可以在application.yml中配置需要禁用的filter，格式：

zuul:

FormBodyWrapperFilter:

pre:

disable: true

##### Spring Cloud Gateway

Spring Cloud Gateway 是 Spring Cloud 的一个全新项目，该项目是基于 Spring 5.0，Spring Boot 2.0 和 Project Reactor 等技术开发的网关，它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 作为 Spring Cloud 生态系统中的网关，目标是替代 Netflix Zuul，其不仅提供统一的路由方式，并且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控/指标，和限流。

**Spring Cloud Gateway 的特征：**

基于 Spring Framework 5，Project Reactor 和 Spring Boot 2.0

动态路由

Predicates 和 Filters 作用于特定路由

集成 Hystrix 断路器

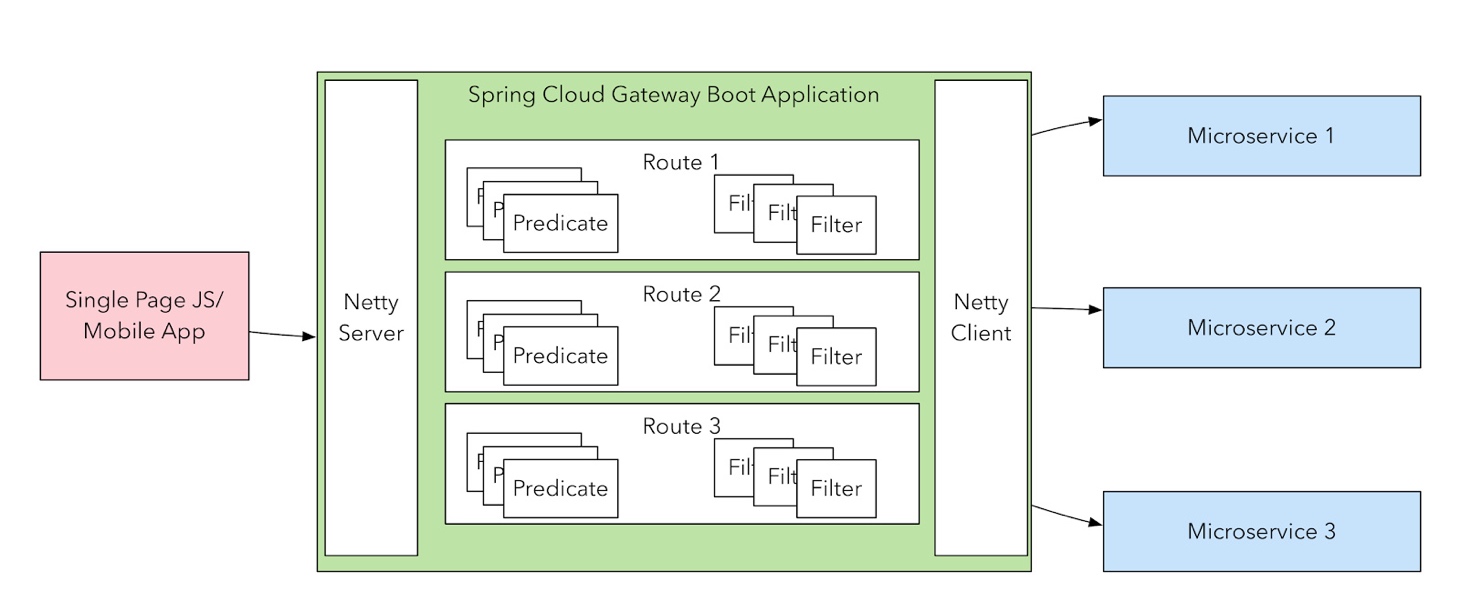
集成 Spring Cloud DiscoveryClient

易于编写的 Predicates 和 Filters

限流

路径重写

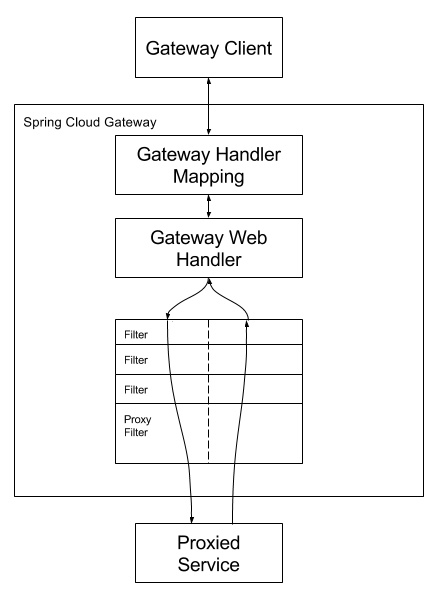
**Spring Cloud Gateway基本代理的路由转发配置原理图：**



**术语**

* Route（路由）：这是网关的基本构建块。它由一个 ID，一个目标 URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配。
* Predicate（断言）：这是一个 Java 8 的 Predicate。输入类型是一个 ServerWebExchange。我们可以使用它来匹配来自 HTTP 请求的任何内容，例如 headers 或参数。
* Filter（过滤器）：这是org.springframework.cloud.gateway.filter.GatewayFilter的实例，我们可以使用它修改请求和响应。

**工作流程：**



客户端向 Spring Cloud Gateway 发出请求。如果 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler。Handler 再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。 过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前（“pre”）或之后（“post”）执行业务逻辑。

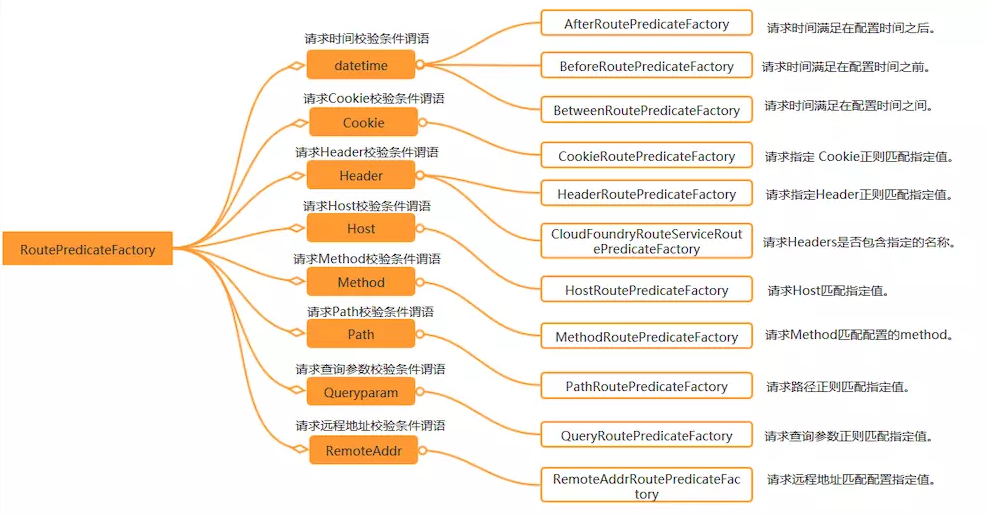
**路由规则**

Spring Cloud Gateway 是通过 Spring WebFlux 的 HandlerMapping 做为底层支持来匹配到转发路由，Spring Cloud Gateway 内置了很多 Predicates 工厂，这些 Predicates 工厂通过不同的 HTTP 请求参数来匹配，多个 Predicates 工厂可以组合使用。

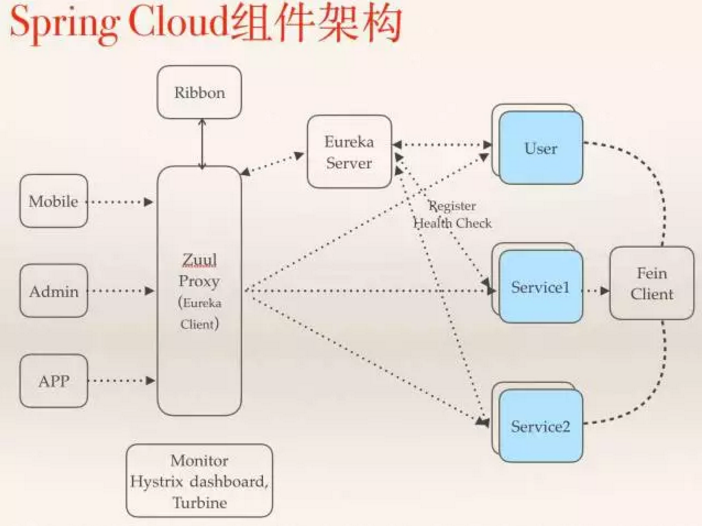
**Predicate 介绍**

Predicate 来源于 Java 8，是 Java 8 中引入的一个函数，Predicate 接受一个输入参数，返回一个布尔值结果。该接口包含多种默认方法来将 Predicate 组合成其他复杂的逻辑（比如：与，或，非）。可以用于接口请求参数校验、判断新老数据是否有变化需要进行更新操作。

在 Spring Cloud Gateway 中 Spring 利用 Predicate 的特性实现了各种路由匹配规则，有通过 Header、请求参数等不同的条件来进行作为条件匹配到对应的路由。网上有一张图总结了 Spring Cloud 内置的几种 Predicate 的实现。



|  |  |  |
| --- | --- | --- |
| **序号** | **技术名称** | **备注** |
| 1 | spring cloud | 微服务框架 |
| 2 | netflix eureka | 服务注册 |
| 3 | config | 配置中心 |
| 4 | ribbon histrix | 服务治理 |
| 5 | zuul | 服务网关 |
| 6 | spring boot | 微服务框架 |
| 7 | log4j2+kafka+elk | 日志采集 |
| 8 | logstash+kibana+elasticsearch | 日志分析 |
| 9 | zipkin | 链路跟踪 |
| 10 | redis | 分布式缓存 |
| 11 | sharding-jdbc，tomcat | 分库分表 |
| 12 | mariaDB | 数据库 |
| 13 | spring batch | 批量服务 |
| 14 | elastic-job | 调度服务 |
| 15 | brave+zipkin/prometheus+grafana+con | 监控告警 |
| 16 | jstorm spark-streaming | 实时计算引擎 |
| 17 | netty4 | rpc网关 |
| 18 | gitlab-ci+kubernetes | 持续集成 |



**解释一下这张图中各组件的运行流程：**

**所有请求都统一通过 API 网关（Zuul）来访问内部服务。**

**网关接收到请求后，从注册中心（Eureka）获取可用服务。**

**由 Ribbon 进行均衡负载后，分发到后端的具体实例。**

**微服务之间通过 Feign 进行通信处理业务。**

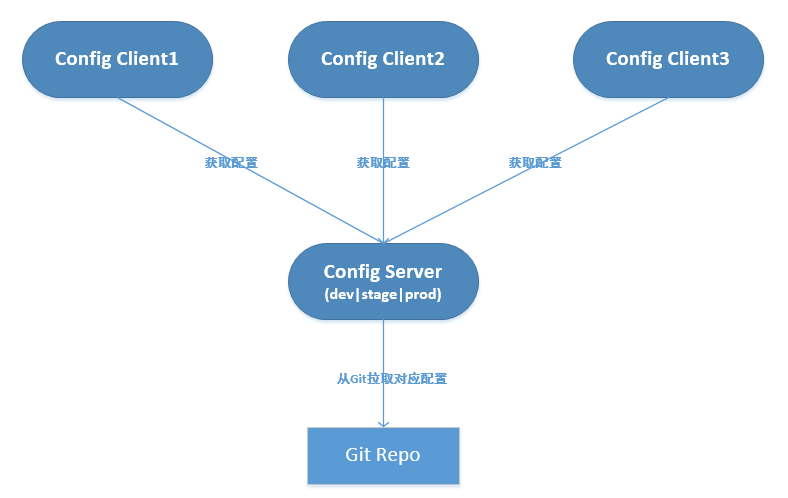
**Hystrix 负责处理服务超时熔断。**

**Turbine 监控服务间的调用和熔断相关指标。**

## 配置中心（config）

Spring Cloud Config是Spring Cloud团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，**它分为服务端与客户端两个部分**。其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring Cloud Config实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于Spring构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于Spring Cloud Config实现的配置中心默认采用Git来存储配置信息，所以使用Spring Cloud Config构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过Git客户端工具来方便地管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如SVN仓库、本地化文件系统。

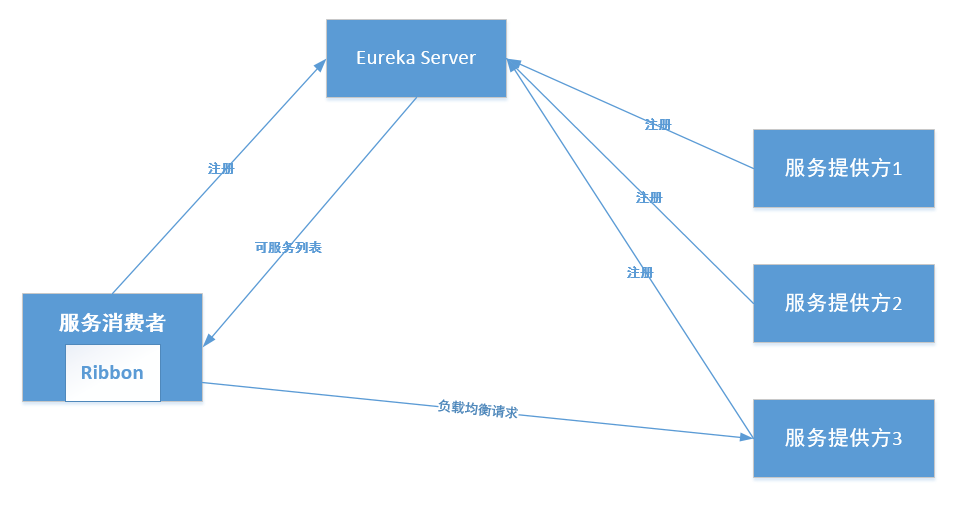
**结构图：**



## 服务治理（ribbon histrix）

1. Spring Cloud Ribbon **是一个基于Http和TCP的客服端负载均衡工具**，它是基于Netflix Ribbon实现的。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。Spring Cloud Ribbon虽然只是一个工具类框架，它不像服务注册中心、配置中心、API网关那样独立部署，但是它几乎存在于每个Spring Cloud构建的微服务和基础设施中。因为服务间的调用，API网关的请求转发等内容实际上都是通过Ribboin来实现的。Ribbon对于使用Spring Cloud来讲非常的重要，因为负载均衡是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一。

Eureka使用Ribboin时候的架构图：



**解释一下这张图中各组件的运行流程：**

**第一步：服务提供方将服务注册到Eureka Server。**

**第二步：服务消费方从服务注册中心获取服务列表，此处使用Ribbon来消费服务，其中Ribbon提供了多种策略，例如轮询round robin、随机Random、根据响应时间加权等。**

2. Spring Cloud Hystrix （服务容错保护）在微服务架构中，我们将系统拆分为很多个服务，各个服务之间通过注册与订阅的方式相互依赖，由于各个服务都是在各自的进程中运行，就有可能由于网络原因或者服务自身的问题导致调用故障或延迟，**随着服务的积压，可能会导致服务崩溃**。为了解决这一系列的问题，断路器等一系列服务保护机制出现了。

断路器本身是一种开关保护机制，用于在电路上保护线路过载，当线路中有电器发生短路时，断路器能够及时切断故障电路，防止发生过载、发热甚至起火等严重后果。在分布式架构中，断路器模式的作用也是类似的。

针对上述问题，Spring Cloud Hystrix 实现了断路器、线路隔离等一系列服务保护功能。它也是基于 Netflix 的开源框架 Hystrix 实现的，该框架的目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix 具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能。

## 服务网关（zuul）

**Zuul的主要功能是路由转发和过滤器**。路由功能是微服务的一部分，比如/api/user转发到到user服务，/api/shop转发到到shop服务。zuul默认和Ribbon结合实现了负载均衡的功能。

其核心功能有两个：请求路由和请求过滤。

这里对路由的配置用了3种格式，体现了API Gateway的路由分发功能：

第一种格式（path-url）：如果请求/163这个地址，将会转发到http://www.163.com上去。

第二种(path-serviceId)：如果请求/api-consumer开头的地址，将会转发到eureka上serviceId为“demo-feign-freeconsumer”这个服务上去。

第三种(给微服务名指定path)：给demo-feign-freeservice这个微服务指定了它的请求地址是/api-service/\*\*。

API网关作为系统的统一入口，屏蔽了系统内部各个微服务的细节；可以与服务治理框架结合，实现自动化的服务实例维护以及负载均衡的路由转发。可以实现接口权限校验与微服务业务逻辑的解耦；通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。

## 微服务基础框架（spring boot）

SpringBoot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Boot致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

Spring boot的特点

1. 创建独立的Spring应用程序

2. 嵌入的Tomcat，无需部署WAR文件

3. 简化Maven配置

4. 自动配置Spring

5. 提供生产就绪型功能，如指标，健康检查和外部配置

6. 绝对没有代码生成和对XML没有要求配置

**Spring boot的优点**

spring boot 可以支持你快速的开发出 restful 风格的微服务架构，自动化确实方便，做微服务再合适不过了，单一jar包部署和管理都非常方便。只要系统架构设计合理，大型项目也能用，加上nginx负载均衡，轻松实现横向扩展。

**Spring boot的宗旨**

希望通过设计大量的自动化配置等方式来简化Spring原有样板化的配置，使得开发者可以快速构建应用。除了解决配置问题外，Spring Boot还通过一系列Starter POMs的定义，让我们整合各项功能的时候，不需要在Maven的pom.xml中维护哪些错综复杂的依赖关系，而是通过类似模块化的Starter模块定义来引用，使得依赖管理工作变得更为简单。

## 日志采集（log4j2+kafka+elk）

**Kafka简介：**

Kafka是一种分布式的，基于发布/订阅的消息系统。主要设计目标如下：

以时间复杂度为O(1)的方式提供消息持久化能力，即使对TB级以上数据也能保证常数时间复杂度的访问性能。

高吞吐率。即使在非常廉价的商用机器上也能做到单机支持每秒100K条以上消息的传输。

支持Kafka Server间的消息分区，及分布式消费，同时保证每个Partition内的消息顺序传输。

同时支持离线数据处理和实时数据处理。

Scale out：支持在线水平扩展。

**ELK（Elastic Stack）简介：**

开源的实时日志分析平台。ELK由ElasticSearch、Logstash和Kiabana三个开源工具组成。

Elasticsearch是个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，restful风格接口，多数据源，自动搜索负载等。

Logstash是一个完全开源的工具，他可以对你的日志进行收集、过滤，并将其存储供以后使用（如，搜索）。

Kibana 也是一个开源和免费的工具，它Kibana可以为 Logstash 和 ElasticSearch 提供的日志分析友好的 Web 界面，可以帮助您汇总、分析和搜索重要数据日志。

## 日志分析（Logstash+Kibana+Elasticsearch）

Elasticsearch所涉及到的每一项技术都不是创新或者革命性的，全文搜索，分析系统以及分布式数据库这些早就已经存在了。它的革命性在于将这些独立且有用的技术整合成一个一体化的、实时的应用。Elasticsearch是面向文档(document oriented)的，这意味着它可以存储整个对象或文档(document)。然而它不仅仅是存储，还会索引(index)每个文档的内容使之可以被搜索。在Elasticsearch中，你可以对文档（而非成行成列的数据）进行索引、搜索、排序、过滤。这种理解数据的方式与以往完全不同，这也Elasticsearch能够执行复杂的全文搜索的原因之一。

应用程序的日志大部分都是输出在服务器的日志文件中，这些日志大多数都是开发人员来看，然后开发却没有登陆服务器的权限，如果开发人员需要查看日志就需要到服务器来拿日志，然后交给开发；试想下，一个公司有10个开发，一个开发每天找运维拿一次日志，对运维人员来说就是一个不小的工作量，这样大大影响了运维的工作效率，部署ELKstack之后，开发任意就可以直接登陆到Kibana中进行日志的查看，就不需要通过运维查看日志，这样就减轻了运维的工作。

日志种类多，且分散在不同的位置难以查找：如LAMP/LNMP网站出现访问故障，这个时候可能就需要通过查询日志来进行分析故障原因，如果需要查看apache的错误日志，就需要登陆到Apache服务器查看，如果查看数据库错误日志就需要登陆到数据库查询，试想一下，如果是一个集群环境几十台主机呢？这时如果部署了ELKstack就可以登陆到Kibana页面进行查看日志，查看不同类型的日志只需要电动鼠标切换一下索引即可。

**ELK职能分工**

Logstash做日志对接，接受应用系统的log，然后将其写入到elasticsearch中，logstash可以支持N种log渠道，kafka渠道写进来的、和log目录对接的方式、也可以对reids中的log数据进行监控读取，等等。

Elasticsearch存储日志数据，方便的扩展特效，可以存储足够多的日志数据。

Kibana则是对存放在elasticsearch中的log数据进行：数据展现、报表展现，并且是实时的。

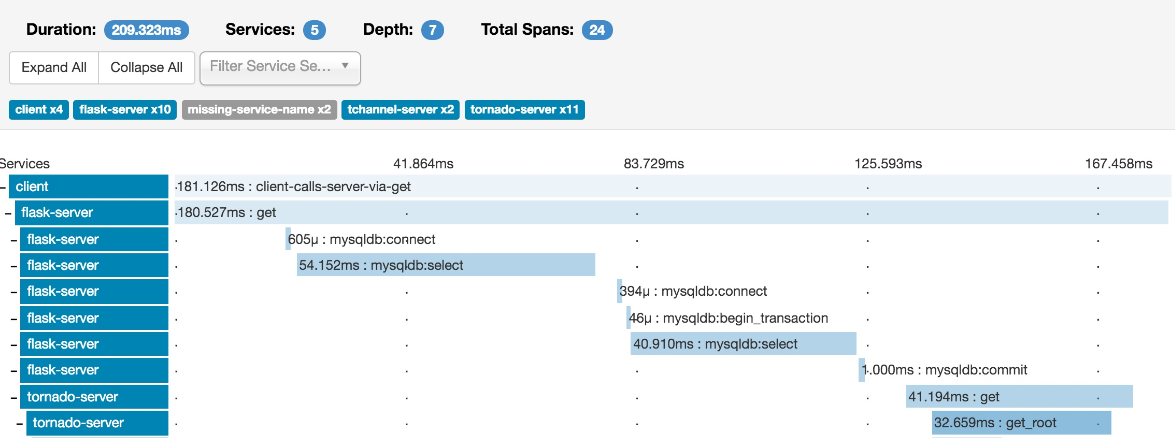
**集成**

使用Elasticsearch来存储日志信息，对一般系统来说可以理解为可以存储无限条数据，因elasticsearch有良好的扩展性，然后是有一个logstash，可以把理解为数据接口，为elasticsearch对接外面过来的log数据，它对接的渠道，有kafka，有log文件，有redis等等，足够兼容N多log形式，最后还有一个部分就是kibana，它主要用来做数据展现，log那么多数据都存放在elasticsearch中，我们得看看log是什么样子的吧，这个kibana就是为了让我们看log数据的，但还有一个更重要的功能是，可以编辑N种图表形式，什么柱状图，折线图等等，来对log数据进行直观的展现。

## 链路跟踪（zipkin）

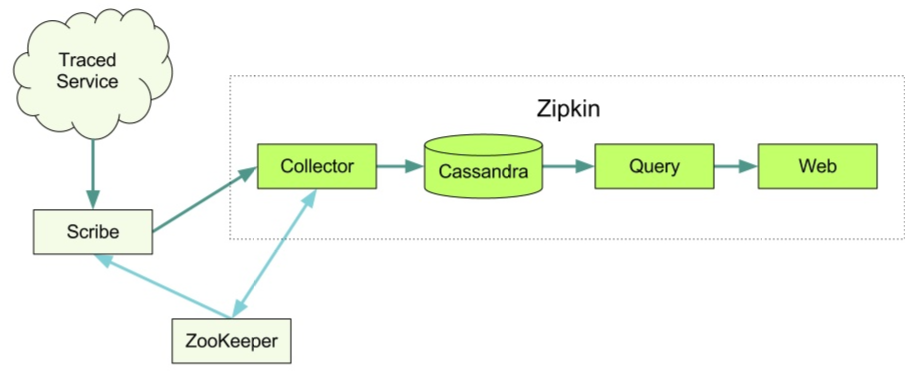
Zipkin 是一款开源的分布式实时数据追踪系统（Distributed Tracking System），基于 Google Dapper 的论文设计而来，由 Twitter 公司开发贡献。其主要功能是聚集来自各个异构系统的实时监控数据，用来追踪微服务架构下的系统延时问题。

应用系统需要进行装备（instrument）以向 Zipkin 报告数据。Zipkin 的用户界面可以呈现一幅关联图表，以显示有多少被追踪的请求通过了每一层应用。



Zipkin 以 Trace 结构表示对一次请求的追踪，又把每个 Trace 拆分为若干个有依赖关系的 Span。在微服务架构中，一次用户请求可能会由后台若干个服务负责处理，那么每个处理请求的服务就可以理解为一个 Span（可以包括 API 服务，缓存服务，数据库服务以及报表服务等）。当然这个服务也可能继续请求其他的服务，因此 Span 是一个树形结构，以体现服务之间的调用关系。

Zipkin 的用户界面除了可以查看 Span 的依赖关系之外，还以瀑布图的形式显示了每个 Span 的耗时情况，可以一目了然的看到各个服务的性能状况。打开每个 Span，还有更详细的数据以键值对的形式呈现，而且这些数据可以在装备应用的时候自行添加。



**解释一下这张图中各组件：**

Zipkin 主要由四部分构成：收集器、数据存储、查询以及 Web 界面。Zipkin 的收集器负责将各系统报告过来的追踪数据进行接收；而数据存储默认使用 Cassandra，也可以替换为 MySQL；查询服务用来向其他服务提供数据查询的能力，而 Web 服务是官方默认提供的一个图形用户界面。

## 分布式缓存（redis）

Redis是基于内存、可持久化的日志型、Key-Value数据库 高性能存储系统，并提供多种语言的API。其基本原理，redis持久化(aof) append online file： 写log(aof), 到一定程度再和内存合并. 追加再追加, 顺序写磁盘, 对性能影响非常小。由于redis是单点，项目中需要使用，必须自己实现分布式，通过key做一致性哈希，实现key对应redis结点的分布。

1. 支持5种数据结构

支持strings, hashes, lists, sets, sorted sets

string是很好的存储方式，用来做计数存储。sets用于建立索引库非常棒；

2. K-V 存储 vs K-V 缓存

Redis中持久化的应用和非持久化的方式不会差别很大：

非持久化的为8-9万tps，那么持久化在7-8万tps左右；

当使用持久化时，需要考虑到持久化和写性能的配比，也就是要考虑redis使用的内存大小和硬盘写的速率的比例计算；

3. 社区活跃

Redis目前有3万多行代码, 代码写的精简，有很多巧妙的实现，作者有技术洁癖

Redis的社区活跃度很高，这是衡量开源软件质量的重要指标，开源软件的初期一般都没有商业技术服务支持，如果没有活跃社区做支撑，一旦发生问题都无处求救；

## 分库分表（sharding-jdbc）

**Sharding-JDBC是当当应用框架ddframe中**，从关系型数据库模块dd-rdb中分离出来的数据库水平分片框架，实现透明化数据库分库分表访问。Sharding-JDBC是继dubbox和elastic-job之后，ddframe系列开源的第3个项目。

Sharding-JDBC直接封装JDBC API，可以理解为增强版的JDBC驱动，旧代码迁移成本几乎为零：

可适用于任何基于Java的ORM框架，如JPA、Hibernate、Mybatis、Spring JDBC Template或直接使用JDBC。

可基于任何第三方的数据库连接池，如DBCP、C3P0、 BoneCP、Druid等。

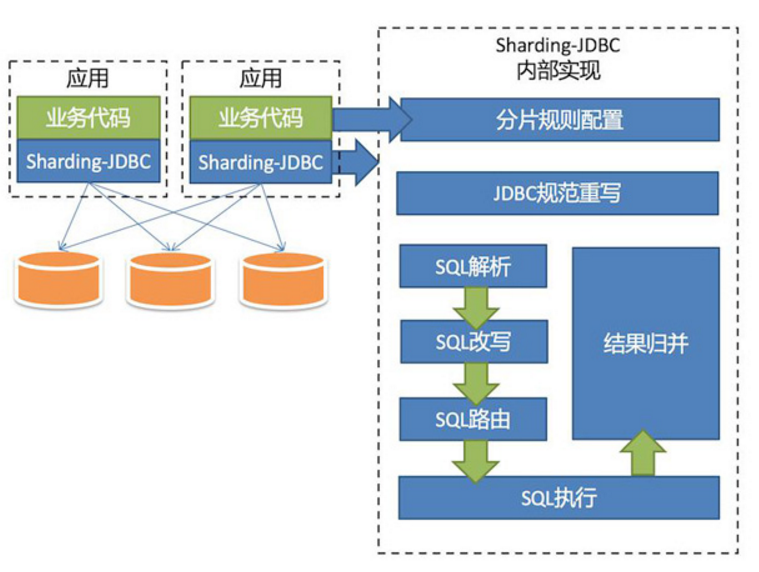
理论上可支持任意实现JDBC规范的数据库。虽然目前仅支持MySQL，但已有支持Oracle、SQLServer等数据库的计划。

Sharding-JDBC定位为轻量Java框架，使用客户端直连数据库，以jar包形式提供服务，无proxy代理层，无需额外部署，无其他依赖，DBA也无需改变原有的运维方式。

Sharding-JDBC分片策略灵活，可支持等号、between、in等多维度分片，也可支持多分片键。

SQL解析功能完善，支持聚合、分组、排序、limit、or等查询，并支持Binding Table以及笛卡尔积表查询。

**Sharding-JDBC的整体架构图**



## 数据库管理系统（mariaDB）

MariaDB数据库管理系统是MySQL的一个分支，主要由开源社区在维护，采用GPL授权许可 MariaDB的目的是完全兼容MySQL，包括API和命令行，使之能轻松成为MySQL的代替品。在存储引擎方面，使用XtraDB（英语：XtraDB）来代替MySQL的InnoDB。 MariaDB由MySQL的创始人Michael Widenius（英语：Michael Widenius）主导开发，他早前曾以10亿美元的价格，将自己创建的公司MySQL AB卖给了SUN，此后，随着SUN被甲骨文收购，MySQL的所有权也落入Oracle的手中。MariaDB名称来自Michael Widenius的女儿Maria的名字。

MariaDB基于事务的Maria存储引擎，替换了MySQL的MyISAM存储引擎，它使用了Percona的 XtraDB，InnoDB的变体，分支的开发者希望提供访问即将到来的MySQL 5.4 InnoDB性能。这个版本还包括了 PrimeBase XT (PBXT) 和 FederatedX存储引擎。

MariaDB虽然被视为MySQL数据库的替代品，**但它在扩展功能、存储引擎以及一些新的功能改进方面都强过MySQL**。而且从MySQL迁移到MariaDB也是非常简单的：

1、数据和表定义文件（.frm）是二进制兼容的

2、所有客户端API、协议和结构都是完全一致的

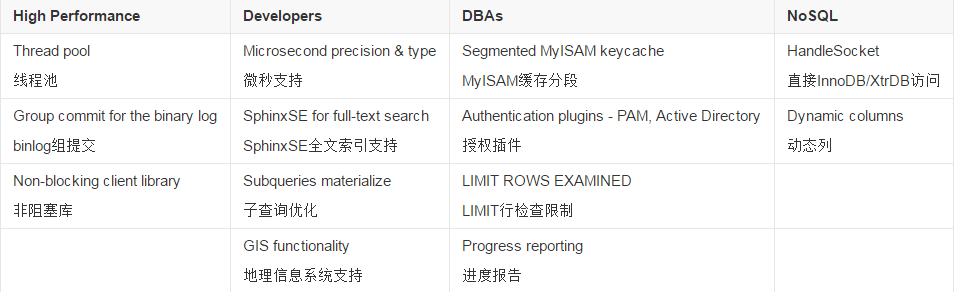
3、所有文件名、二进制、路径、端口等都是一致的

4、所有的MySQL连接器，比如PHP、Perl、Python、Java、.NET、MyODBC、Ruby以及MySQL C connector等在MariaDB中都保持不变

5、mysql-client包在MariaDB服务器中也能够正常运行

6、共享的客户端库与MySQL也是二进制兼容的

相对于MySQL最新的版本5.6来说，在性能、功能、管理、NoSQL扩展方面，MariaDB包含了更丰富的特性。比如微秒的支持、线程池、子查询优化、组提交、进度报告等。详情见列表。



## 批量服务（spring batch）

Spring Batch是一个轻量级、完善的批处理框架，旨在帮助企业建立健壮、高效的批处理应用。SpringBatch是Spring的一个子项目，使用java语言并基于Spring框架为基础开发，使得已经使用Spring框架的开发者或者企业更容易访问和利用企业原有服务。

**SpringBatch提供了大量可重用的组件，包括日志、事务、任务作业统计、任务重启、跳过、重复、资源管理**。对于大数据量和高性能的批处理任务，SpringBacth同样提供了高级功能和特性来支持，比如分区功能、远程功能。总之，通过SpringBatch能够支持简单的、复杂的和大数据量的批处理作业。

**SpringBatch是一个批处理应用框架，不是调度框架，但需要和调度框架合作来构建完成批处理任务**。它只关注批处理任务相关的问题，如事务、并发、监控、执行等，并不提供相应的调度功能。

SpringBatch批处理框架支撑的业务场景：

1、定期提交批处理任务

2、并行批处理，即并行处理任务

3、企业消息驱动处理

4、大规模的并行处理

5、手动或定时重启

6、按顺序处理依赖的任务(可扩展为工作流驱动的批处理)

7、部分处理，如在回滚时忽略记录

8、完整的批处理事物

SpringBatch批处理框架支撑的技术目标：

1、利用Spring编程模型，使程序员专注于业务处理，让Spring框架管理流程

2、明确分离批处理的执行环境和应用

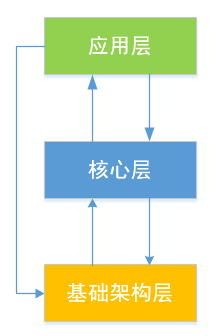
3、将通用核心的服务以接口形式提供

4、提供“开箱即用”的简单的默认的核心执行接口

5、提供Spring框架中配置、自定义和扩展服务

6、所有默认实现的核心服务能够容易的被扩展与替换，不会影响基础层

7、提供一个简单的部署形式，使用Maven进行编译



**解释一下这张图中各个分层：**

这种分层结构有三个重要的组成部分：应用层、核心层、基础架构层。应用层包含所有的批处理作业，通过Spring框架管理程序员自定义的代码。核心层包含了Batch启动和控制所需要的核心类，如：JobLauncher、Job和step等。应用层和核心层建立在基础构架层之上，基础构架层提供共通的读（ItemReader）、写（ItemWriter）、和服务（如RetryTemplate：重试模块。可以被应用层和核心层使用）。

## 调度服务（elastic-job）

elastic-job是当当开源的一款非常好用的作业框架，在这之前，我们开发定时任务一般都是使用quartz或者spring-task（ScheduledExecutorService），无论是使用quartz还是spring-task，我们都会至少遇到两个痛点：

1.不敢轻易跟着应用服务多节点部署，可能会重复多次执行而引发系统逻辑的错误。

2.quartz的集群仅仅只是用来HA，节点数量的增加并不能给我们的每次执行效率带来提升，即不能实现水平扩展。

**主要功能**

**分布式：**  重写Quartz基于数据库的分布式功能，改用Zookeeper实现注册中心。

**并行调度：** **采用任务分片方式实现。将一个任务拆分为多个独立的任务项，由分布式的服务器并行执行各自分配到的分片项。**

弹性扩容缩容： 将任务拆分为多个子任务项后，各个服务器分别执行各自分配到的任务项。一旦有新的服务器加入集群，或现有服务器下线，elastic-job将在保留本次任务执行不变的情况下，下次任务开始前触发任务重分片。

集中管理： 采用基于Zookeeper的注册中心，集中管理和协调分布式作业的状态，分配和监听。外部系统可直接根据Zookeeper的数据管理和监控elastic-job。

定制化流程型任务： 作业可分为简单和数据流处理两种模式，数据流又分为高吞吐处理模式和顺序性处理模式，其中高吞吐处理模式可以开启足够多的线程快速的处理数据，而顺序性处理模式将每个分片项分配到一个独立线程，用于保证同一分片的顺序性。

**其他功能**

失效转移：运行中的作业服务器崩溃不会导致重新分片，只会在下次作业启动时分片。启用失效转移功能可以在本次作业执行过程中，监测其他作业服务器空闲，抓取未完成的孤儿分片项执行。

**非功能需求**

稳定性： 在服务器无波动的情况下，并不会重新分片；即使服务器有波动，下次分片的结果也会根据服务器IP和作业名称哈希值算出稳定的分片顺序，尽量不做大的变动。

高性能： 同一服务器的批量数据处理采用自动切割并多线程并行处理。

一致性： elastic-job可牺牲部分性能用以保证同一分片项不会同时在两个服务器上运行。

容错性： 任务挂掉，elastic-job会找空闲的作业服务器（可能是未分配任务的，也可能是完成执行本次任务执行的）执行。 如果当时没有空闲服务器，则将在某服务器完成分配的任务时抓取未分配的分片项。

运维平台： 提供web控制台用于管理作业。主要有

1> 登录安全控制

2> 注册中心管理

3> 作业维度状态查看

4> 服务器维度状态查看

5> 快捷修改作业设置

6> 控制作业暂停和恢复运行

## 监控告警（brave+zipkin/prometheus+grafana）

使用 **Brave+Zipkin**可以实现分布式系统追踪，Brave 是用来装备 Java 程序的类库，提供了面向 Standard Servlet、Spring MVC、Http Client、JAX RS、Jersey、Resteasy 和 MySQL 等接口的装备能力，可以通过编写简单的配置和代码，让基于这些框架构建的应用可以向 Zipkin 报告数据。同时 Brave 也提供了非常简单且标准化的接口，在以上封装无法满足要求的时候可以方便扩展与定制。

1、装备标准的 Servlet 应用

Brave 提供了 brave-web-servlet-filter 模块，可以为标准的 Servlet 应用添加向 Zipkin 服务器报告数据的能力，需要做的就是在 web.xml 文件增加一个 BraveServletFilter。

不过这个 Filter 在初始化的时候需要传入几个参数，这些参数可以通过 brave 对象的对应方法获得，但是注入这些构造参数，最简单的办法还是使用 Spring 提供的 DelegatingFilterProxy。

2、装备 Spring MVC 应用

Brave 自带了 brave-spring-web-servlet-interceptor 模块，因此装备 Spring MVC 项目变得非常容易，只需要在配置文件中添加一些 interceptor 就好了。

3、装备 MySQL 服务

brave-mysql 模块在 JDBC 驱动层面添加了一些拦截器，可以对 MySQL 的查询进行监控。在使用之前也需要通过 Spring 进行一下配置。

**Prometheus**是一个开源监控系统，它前身是SoundCloud的警告工具包。从2012年开始，许多公司和组织开始使用Prometheus。该项目的开发人员和用户社区非常活跃，越来越多的开发人员和用户参与到该项目中。目前它是一个独立的开源项目，且不依赖与任何公司。为了强调这点和明确该项目治理结构，Prometheus在2016年继Kurberntes之后，加入了Cloud Native Computing Foundation。主要具有如下功能：

多维 数据模型（时序由 metric 名字和 k/v 的 labels 构成）。

灵活的查询语句（PromQL）。

无依赖存储，支持 local 和 remote 不同模型。

采用 http 协议，使用 pull 模式，拉取数据，简单易懂。

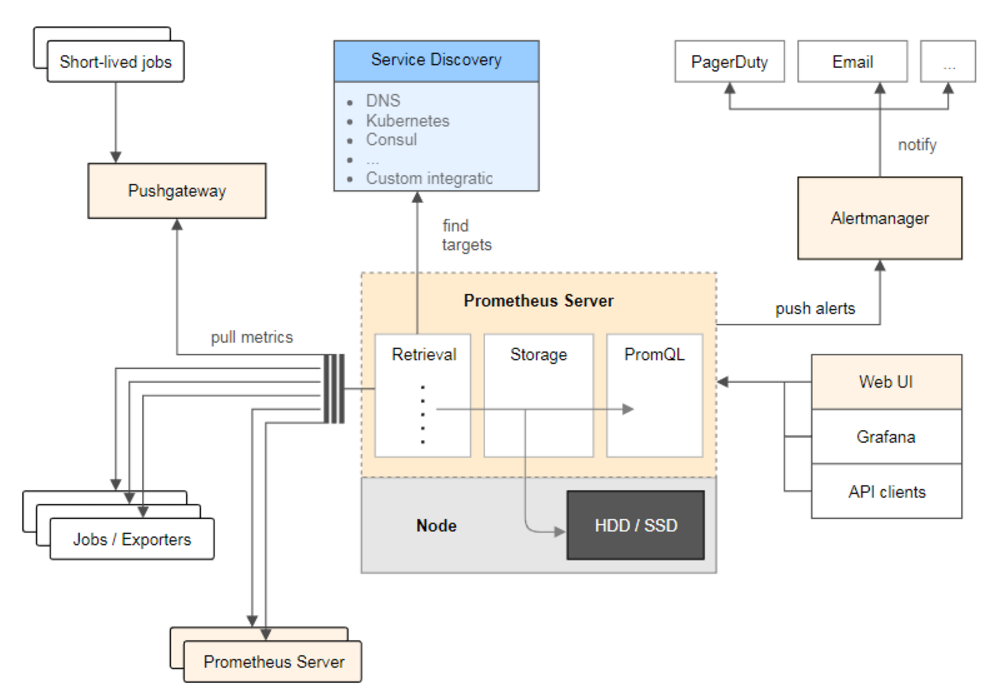
监控目标，可以采用服务发现或静态配置的方式。

支持多种统计数据模型，图形化友好。

**适用场景：**

Prometheus在记录纯数字时间序列方面表现非常好。它既适用于面向服务器等硬件指标的监控，也适用于高动态的面向服务架构的监控。对于现在流行的微服务，Prometheus的多维度数据收集和数据筛选查询语言也是非常的强大。Prometheus是为服务的可靠性而设计的，当服务出现故障时，它可以使你快速定位和诊断问题。它的搭建过程对硬件和服务没有很强的依赖关系。

Prometheus，它的价值在于可靠性，甚至在很恶劣的环境下，你都可以随时访问它和查看系统服务各种指标的统计信息。 如果你对统计数据需要100%的精确，它并不适用，例如：它不适用于实时计费系统。



**解释一下这张图中各个组件功能：**

从这个架构图，也可以看出 Prometheus 的主要模块包含， prometheus server, exporters, pushgateway, PromQL, Alertmanager, WebUI 等。下面简单介绍各个组件实现的功能：

1. prometheus server： 定期从静态配置的 targets 或者服务发现（主要是DNS、consul、k8s、mesos等）的 targets 拉取数据。

2. exporters：负责向prometheus server做数据汇报的程序统。而不同的数据汇报由不同的exporters实现，比如监控主机有node-exporters，mysql有MySQL server exporter。

3. pushgateway：主要使用场景为：Prometheus 采用 pull 模式，可能由于不在一个子网或者防火墙原因，导致 Prometheus 无法直接拉取各个 target 数据。在监控业务数据的时候，需要将不同数据汇总, 由 Prometheus 统一收集。

4. Alertmanager：实现prometheus的告警功能。

5. webui：主要通过grafana来实现webui展示。

**Grafana**

Grafana 是一个开源仪表盘工具，它可用于Graphite、InfluxDB与 OpenTSDB 一起使用。最新的版本还可以用于其他的数据源，比如Elasticsearch。

从本质上说，它是一个功能丰富的Graphite-web 替代品，能帮助用户更简单地创建和编辑仪表盘。它包含一个独一无二的Graphite目标解析器，从而可以简化度量和函数的编辑。Grafana快速的客户端渲染默认使用的是 Flot ，即使很长的时间范围也可应对，这样用户就可以创建具有智能轴格式（比如线和点）的复杂图表了。其功能有：

1. 日志与度量

这方面的主要的不同在于，Grafana专注于根据CPU和IO利用率之类的特定指标提供时间序列图表。Kibana则专注于另一方面，它运行于Elasticsearch的上层，能创建一个复杂的日志分析仪表盘。举个例子，Grafana无法进行数据的检索和浏览。

2. 基于角色的访问

默认情况下，Kibana的仪表盘是公开的，没有进行基于角色的访问控制。如果你需要针对多个用户设置不同的权限级别，就得增加额外的配置预算采购 Shield 了。比较而言，Grafana内置的RBA允许你维护用户和团队访问仪表盘的权限。另外，Grafana的富API可能用于保存特定仪表表、创建用户用户和更新数据源的任务。你还可以创建特定的API关键字并赋予它们新的职责。

3. 仪表盘灵活性

虽然Kibana有大量内置的图表类型，但它们之上的控制仍是最初的限制，Grafana包括更多的选择，可以更灵活地浏览和使用图表，如果要选择一个指定的时间序列，可以使用Y-Bar之类的工具。然而，随着 Kibana 4中展示的增强特性，比如可以保存元素并拖曳到新的仪表盘中，我们可以预期Kibana能快速地弥补这一差距。

4. 数据源的集成

Grafana支持许多不同的存储后端。Grafana针对每个数据源都有一个特定的查询编辑器，它是针对数据源所具备的特性和能力特别定制的。然而，正如上文所说，Kibana原生集成进了ELK栈，这使安装极为简单，对用户非常友好。

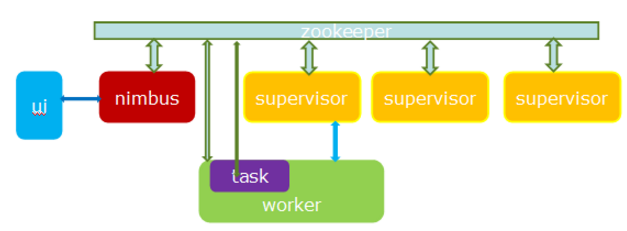
5. 开源社区

这两个开源工具都很强大。如果密切观察一下Github上的Grafana，你将发现大概有7000次代码提交，而Kibana更有12000次。而如果要了解Kibana开源社区真正的优势，可以看看谷歌趋势，它把ELK作为一个整体进行了相关信息的揭示：使把Graphite也算进来，ELK仍保持着快速的增长，并有潜力在不久的将来保持领先。

## 实时计算引擎（JStorm /Spark Streaming）

JStorm 是一个类似于 Hadoop 的MapReduce的计算系统，它是由Alibaba开源的实时计算模型，它使用Java重写了原生的Storm模型（Clojure和Java混合编写的），并且再原来的基础上做了许多改进。用户只需按照指定的接口实现一个任务，然后将这个任务提交给JStorm系统，JStorm在接受了任务指令后，会无间断运行任务，一旦出现异常导致某个Worker发送故障，调度器立刻会分配一个新的Worker去顶替异常的Worker。

从架构上看，其本质是一个基于zk的分布式调度系统。



**解释一下这张图中各个组件功能：**

Nimbus 调度器

Supervisor Worker的代理角色，负责Kill掉Worker和运行Worker

Worker Task的容器

Task 任务的执行者

ZooKeeper 系统的协调者

**应用场景：**

从应用的角度来说，JStorm它是一种分布式的应用；从系统层面来说，它又类似于MapReduce这样的调度系统；而从数据方面来说，它又是一种基于流水数据的实时处理解决方案。如今，DT时代的当下，用户和企业也不仅仅只满足于离线数据，对于数据的实时性要求也越来越高了。

在早期，Storm和JStorm未问世之前，业界有很多实时计算系统，可谓百家争鸣，自Storm和JStorm出世之后，基本这两者占据主要地位，原因如下：

易开发：接口简单，上手容易，只需要按照Spout，Bolt以及Topology的编程规范即可开发一个扩展性良好的应用，底层的细节我们可以不用去深究其原因。

扩展性：可线性扩展性能。

容错：当Worker异常或挂起，会自动分配新的Worker去工作。

数据精准：其包含Ack机制，规避了数据丢失的风险。使用事物机制，提高数据精度。

日志分析：从收集的日志当中，统计出特定的数据结果，并将统计后的结果持久化到外界存储介质中，如：DB。当下，实时统计主流使用JStorm和Storm。

消息转移：将接受的消息进行Filter后，定向的存储到另外的消息中间件中。

**Spark**是一个类似于MapReduce的分布式计算框架，其核心是弹性分布式数据集，提供了比MapReduce更丰富的模型，可以在快速在内存中对数据集进行多次迭代，以支持复杂的数据挖掘算法和图形计算算法。Spark Streaming是一种构建在Spark上的实时计算框架，它扩展了Spark处理大规模流式数据的能力。

Spark Streaming 类似于 Apache Storm，用于流式数据的处理。根据其官方文档介绍，Spark Streaming 有高吞吐量和容错能力强这两个特点。Spark Streaming 支持的数据输入源很多，例如：Kafka、Flume、Twitter、ZeroMQ 和简单的 TCP 套接字等等。数据输入后可以用 Spark 的高度抽象原语如：map、reduce、join、window 等进行运算。而结果也能保存在很多地方，如 HDFS，数据库等。另外 Spark Streaming 也能和 MLlib（机器学习）以及 Graphx 完美融合。

在 Spark Streaming 中，处理数据的单位是一批而不是单条，而数据采集却是逐条进行的，因此 Spark Streaming 系统需要设置间隔使得数据汇总到一定的量后再一并操作，这个间隔就是批处理间隔。批处理间隔是 Spark Streaming 的核心概念和关键参数，它决定了 Spark Streaming 提交作业的频率和数据处理的延迟，同时也影响着数据处理的吞吐量和性能。

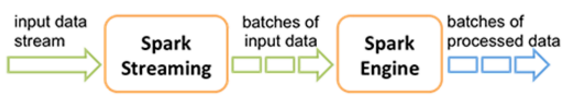
Spark Streaming的优势在于：

能运行在100+的结点上，并达到秒级延迟。

使用基于内存的Spark作为执行引擎，具有高效和容错的特性。

能集成Spark的批处理和交互查询。

为实现复杂的算法提供和批处理类似的简单接口。



**解释一下这张图的处理流程：**

spark streaming接收实时数据流输入的数据流后，再将其划分为一个个batch（小批次数据流）供后续Spark engine处理，所以实际上，Spark Streaming是按一个个batch（小批次）来处理数据流的。

## rpc网关（netty4）

Netty是一个NIO的编程框架，Netty是非常容易和快速开发出网络应用程序的，它提供了一种全新的形式来方便你编写网络应用：它提供了对一些对复杂问题的抽象，提供了一套非常容易使用的api来把我们的业务逻辑和纯粹处理网络的代码分离开来。因为Betty是基于NIO，因此它的整个API都是异步的。

Netty简化了基于TCP和UDP的编程，但是你仍可以用它的底层的API做一些底层处理，因为Netty提供了一系列高抽象的API。作为当前最流行的NIO框架，Netty在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，一些业界著名的开源组件也基于Netty的NIO框架构建。

     Netty有一系列丰富的特性，让我们来看一下它的强大之处：

有一套统一的API来处理异步和同步编程模式

使用非常灵活

简单但却强大的线程机制

业务组件分离方便重用

极小的缩减不必要的Memory Copy

**无锁化的串行设计理念**

在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能的避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。

为了尽可能提升性能，Netty采用了串行无锁化设计，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。

**高效的Reactor线程模型**

在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极特殊应用场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如百万客户端并发连接，或者服务端需要对客户端的握手消息进行安全认证，认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型-主从Reactor多线程模型。

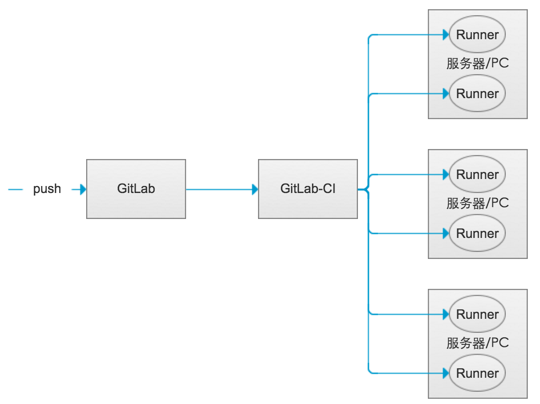
主从Reactor线程模型的特点是：服务端用于接收客户端连接的不再是个1个单独的NIO线程，而是一个独立的NIO线程池。Acceptor接收到客户端TCP连接请求处理完成后（可能包含接入认证等），将新创建的SocketChannel注册到IO线程池（sub reactor线程池）的某个IO线程上，由它负责SocketChannel的读写和编解码工作。Acceptor线程池仅仅只用于客户端的登陆、握手和安全认证，一旦链路建立成功，就将链路注册到后端subReactor线程池的IO线程上，由IO线程负责后续的IO操作。

## 持续集成（[Gitlab-CI](https://docs.gitlab.com/ce/ci/quick_start/README.html)+Kubernetes）

[**Gitlab-CI**](https://docs.gitlab.com/ce/ci/quick_start/README.html)是GitLab Continuous Integration（Gitlab持续集成）的简称。从Gitlab的8.0版本开始，gitlab就全面集成了Gitlab-CI,并且对所有项目默认开启。只要在项目仓库的根目录添加.gitlab-ci.yml文件，并且配置了Runner（运行器），那么每一次合并请求（MR）或者push都会触发CI [pipeline](https://docs.gitlab.com/ce/ci/pipelines.html)。

GitLab-Runner是配合GitLab-CI进行使用的。一般地，GitLab里面的每一个工程都会定义一个属于这个工程的软件集成脚本，用来自动化地完成一些软件集成工作。当这个工程的仓库代码发生变动时，比如有人push了代码，GitLab就会将这个变动通知GitLab-CI。这时GitLab-CI会找出与这个工程相关联的Runner，并通知这些Runner把代码更新到本地并执行预定义好的执行脚本。

所以，GitLab-Runner就是一个用来执行软件集成脚本的东西。你可以想象一下：Runner就像一个个的工人，而GitLab-CI就是这些工人的一个管理中心，所有工人都要在GitLab-CI里面登记注册，并且表明自己是为哪个工程服务的。当相应的工程发生变化时，GitLab-CI就会通知相应的工人执行软件集成脚本。如下图所示：



**Kubernetes**

他是一个全新的基于容器技术的分布式架构领先方案。Kubernetes(k8s)是Google开源的容器集群管理系统（谷歌内部:Borg）。在Docker技术的基础上，为容器化的应用提供部署运行、资源调度、服务发现和动态伸缩等一系列完整功能，提高了大规模容器集群管理的便捷性。

Kubernetes是一个完备的分布式系统支撑平台，具有完备的集群管理能力，多扩多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和发现机制、內建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制以及多粒度的资源配额管理能力。同时Kubernetes提供完善的管理工具，涵盖了包括开发、部署测试、运维监控在内的各个环节。

Kubernetes中，Service是分布式集群架构的核心，一个Service对象拥有如下关键特征：

拥有一个唯一指定的名字

拥有一个虚拟IP（Cluster IP、Service IP、或VIP）和端口号

能够体统某种远程服务能力

被映射到了提供这种服务能力的一组容器应用上

**Kubernetes的核心概念**

**1.Master**

k8s集群的管理节点，负责管理集群，提供集群的资源数据访问入口。拥有Etcd存储服务（可选），运行Api Server进程，Controller Manager服务进程及Scheduler服务进程，关联工作节点Node。Kubernetes API server提供HTTP Rest接口的关键服务进程，是Kubernetes里所有资源的增、删、改、查等操作的唯一入口。也是集群控制的入口进程；Kubernetes Controller Manager是Kubernetes所有资源对象的自动化控制中心；Kubernetes Schedule是负责资源调度（Pod调度）的进程。

**2.Node**

Node是Kubernetes集群架构中运行Pod的服务节点（亦叫agent或minion）。Node是Kubernetes集群操作的单元，用来承载被分配Pod的运行，是Pod运行的宿主机。关联Master管理节点，拥有名称和IP、系统资源信息。运行docker eninge服务，守护进程kunelet及负载均衡器kube-proxy。

**3.Service**

Service定义了Pod的逻辑集合和访问该集合的策略，是真实服务的抽象。Service提供了一个统一的服务访问入口以及服务代理和发现机制，关联多个相同Label的Pod，用户不需要了解后台Pod是如何运行。

**4.Pod**

运行于Node节点上，若干相关容器的组合。Pod内包含的容器运行在同一宿主机上，使用相同的网络命名空间、IP地址和端口，能够通过localhost进行通。Pod是Kurbernetes进行创建、调度和管理的最小单位，它提供了比容器更高层次的抽象，使得部署和管理更加灵活。一个Pod可以包含一个容器或者多个相关容器。