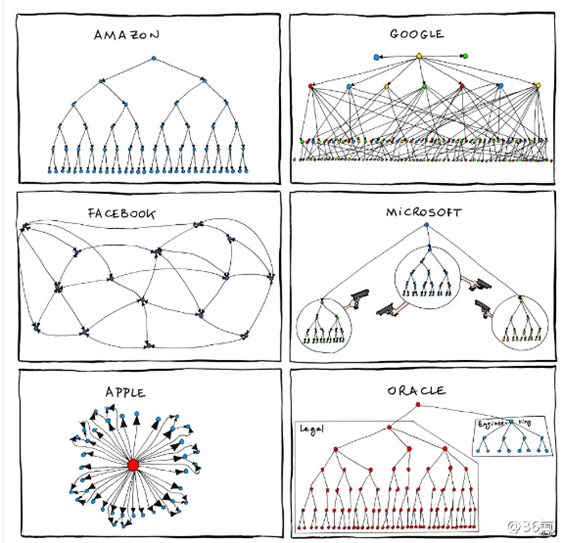
# Cloud Native

Cloud Native的五个层面：

（1） **康威定律**：业务云化推行，从某种意义上讲也是一种变革。既然是变革，必然会涉及组织的各个层面，开发、质量、运维等等都会涉及。康威定律则准确的描述了系统架构和组织的关系：组织决定系统架构！   
  
一个云系统最终长成什么样子，则完全是企业的组织结构决定的，是组织内部、组织之间的沟通结构。   
  
如果要想得到一个合理的Cloud架构，仅从技术入手是不够的，还需要从组织架构入手，才真正有效。

（2） **DevOps**：（英文Development和Operations的组合）是一组过程、方法与系统的统称，用 于促进开发（应用程序/软件工程）、运维和质量保障（QA）部门之间的沟通、协作与整合。它的出现是由于软件行业日益清晰地认识到：为了按时交付软件产品 和服务，开发和运维必须紧密合作。   
  
（3） **持续交付（Continuous Delivery）**：是一系列的开发实践方法，用来确保让代码能够快 速、安全的部署到产品环境中，它通过将每一次改动都提交到一个模拟产品环境中，使用严格的自动化测试，确保业务应用和服务能符合预期。因为使用完全的自动 化过程来把每个变更自动的提交到测试环境中，所以当业务开发完成时，你有信心只需要按一次按钮就能将应用安全的部署到产品环境中。持续交付可以采 用：CI（持续集成）、代码检查、UT（单元测试），持续部署等方式，打通开发、测试、生产的各个环节，持续的增量的交付产品。   
  
（4） **微服务（MicroServices）**：微服务首先是一个服务，其次该服务的颗粒比较小。微服务可以采用Docker、LXC等技术手段实现。   
  
（5） **敏捷基础设施（Agile Infrastructure）**：提供弹性、按需的计算、存储、网络资源能力。可以通过Openstack、KVM、Ceph、OVS等技术手段实现。

Cloud Native可以解决上面的诉求：

* 持续交付、DevOps、微服务解决-->更快的上线速度
* 微服务解决-->细致的故障探测和发现
* 微服务解决-->故障时能自动隔离
* 敏捷基础设施、微服务解决-->故障时能够自动恢复
* 敏捷基础设施、微服务解决-->方便的水平扩容

如何推行Cloud Native

推行Cloud Native可以从如下几方面入手： 

* 组织变革：根据康威定律，如果要达到比较理想的云化效果，必须进行组织变革。一个合理的组织架构，将会极大提高云化的推行。相信很多公司，在决定搞云计算后，都大大小小进行了部门合并和组织结构调整。这块水比较深，相信很多人有变革的切肤之痛。
* 推行DevOps文化：在公司层面推行DevOps文化，倡导开放、合作的组织文化，打破“部门墙”。
* 推行持续交付：联合开发、质量、运维各个环节，打通代码提交、代码检查、UT、开发环境部署、staging环境部署、线上环境部署等流水线。
* 建设敏捷基础设施：这部分是整个Cloud Native的根基。这部分可以采纳的技术非常多，开源的、商用的都比较多。
* 采用微服务架构：微服务架构是Cloud Native的一个核心要素。微服务包含几方面的内容： （1）有支撑微服务的平台。  
  （2）有符合微服务平台规范的APP。  
  （3）如何引入微服务。  
  （4）微服务核心技术点。

### 微服务的4个方面

#### （1）有支撑微服务的平台

支撑微服务的可选技术框架比较多，每个公司都可以根据自身情况选择合适的技术框架。比如： 

* Kubernetes
* Mesos+Docker
* OpenShift V3
* Machine + Swarm + Compose
* OpenStack + Docker
* Cloud Foundry Lattice 其他技术等等

这方面的资料非常多，不再细讲。 

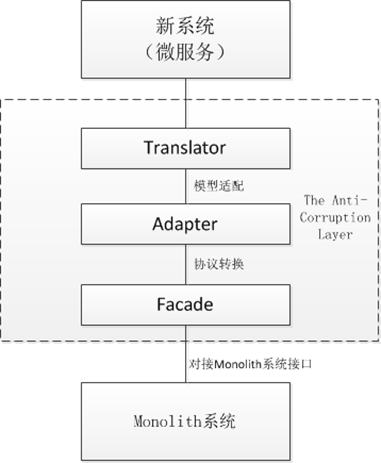
#### （2）有符合微服务平台规范的APP

APP要符合12因子（Twelve-Factor）的规范：   
http://www.infoq.com/cn/news/2012/09/12-factor-app

* 基准代码（Codebase）：代码必须纳入配置库统一管理。
* 依赖（Dependencies）：显式的声明对其他服务的依赖，比如通过Maven、Bundler、NPM等。
* 配置（Config）：对于不同环境（开发/staging/生产等）的参数配置，是通过环境变量的方式进行注入。
* 后台服务（Backing services）：对于DB、缓存等后台服务，是作为附加资源，可以独立的Bind/Unbind。
* 编译/发布/运行（Build、Release、Run）：Build、Release、Run这三个阶段要清晰的定义和分开。
* 无状态进程（Processes）：App的进程是无状态的，任何状态信息都存储到Backing services（DB，缓存等）。
* 端口绑定（Port binding）：App是自包含的，所有对外服务通过Port Binding暴露，比如通过Http。
* 并发（Concurrency）：App可以水平的Scaling。
* 快速启动终止（Disposability）：App进程可以被安全的、快速的关闭和重启。
* 环境一致性（Dev/prod parity）：尽可能的保持开发、staging、线上环境的一致性。
* 日志（Logs）：把日志作为事件流，不管理日志文件，通过一个集中的服务，由执行环境去收集、聚合、索引、分析日志事件。

#### （3）如何引入微服务

直接对原有系统进行微服务改造，是比较困难的，几乎是不现实的。比较合理的方法是新系统采用微服务开发，对老系统进行服务封装，系统架构如下所示：



Monolith系统：对应企业老的系统。   
  
The Anti-Corruption Layer：反腐层，这层完成对老系统的桥接，并阻止老系统的腐烂蔓延。它包含三部分：

1. Facade：简化对老系统接口的对接。
2. Adapter：Request，Response 请求协议适配
3. Translator：领域模型适配，转换微服务模型和老系统模型。

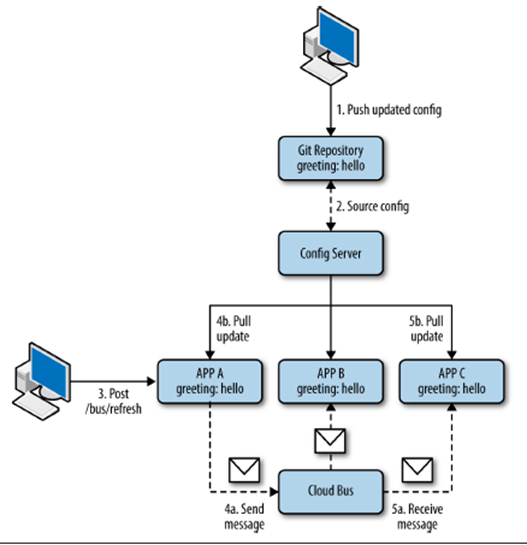
新系统（微服务）：新系统开发采用微服务架构。 

#### （4）微服务核心技术点

主要包含如下几个方面：版本控制的分布式配置中心、服务注册和发现、路由和LB(Load Balancer)、容错、API网关/边缘服务。 

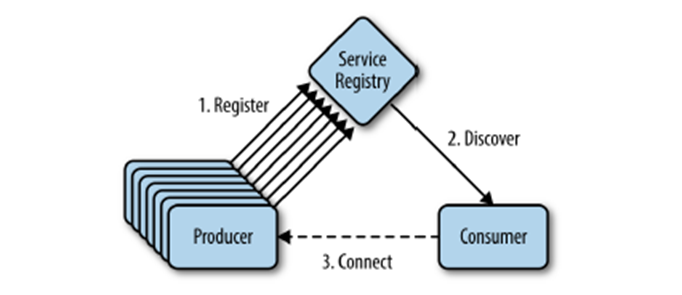
##### （4.1）版本控制的分布式配置中心

支持配置信息版本化控制，可审计，安全，配置更新不需要重启，分布式。这部分可以参考Spring Cloud Config Server、Spring Cloud Bus。



1：用户提交配置更新   
2：配置server更新   
3：对APP A发起Refresh更新配置操作   
4a：发送消息到Bus   
4b：Pull 更新的配置   
5a：APP C接收消息   
5b：Pull更新的配置   
6：其他节点同步更新

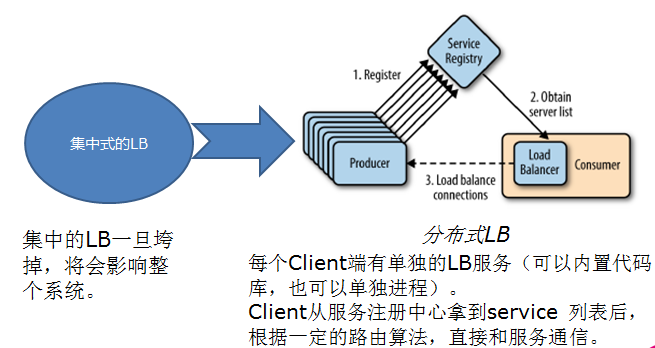
##### （4.2）服务注册和发现



这个是传统的服务注册和发现架构图。服务注册方式，常见的包括DNS，基于ZooKeeper的服务注册方案等等。

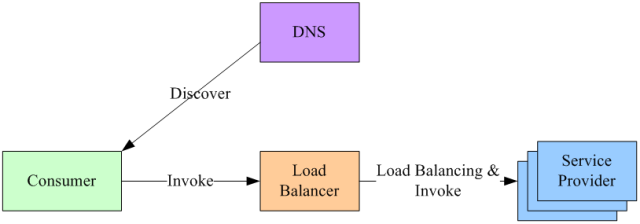
备注：Consumer和Producer之间一般还有个LB。

##### （4.3）路由和LB



##### LB的三种方式

**集中式LB**



集中式LB方案实现简单，在LB上也容易做集中式的访问控制，这一方案目前还是业界主流。集中式LB的主要问题是单点问题，所有服务调用流量都经过LB，当服务数量和调用量大的时候，LB容易成为瓶颈，且一旦LB发生故障对整个系统的影响是灾难性的。另外，LB在服务消费方和服务提供方之间增加了一跳(hop)，有一定性能开销。

**进程内LB**

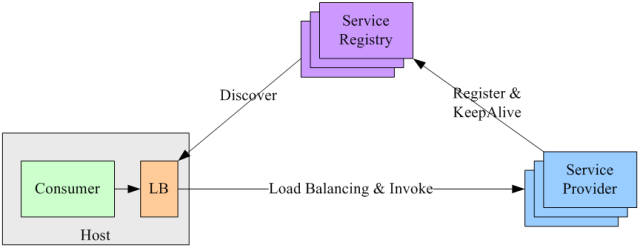
这一方案对服务注册表的可用性(Availability)要求很高，一般采用能满足高可用分布式一致的组件（例如Zookeeper, Consul, Etcd等）来实现。

进程内LB方案是一种分布式方案，LB和服务发现能力被分散到每一个服务消费者的进程内部，同时服务消费方和服务提供方之间是直接调用，没有额外开销，性能比较好。但是，该方案以客户库(Client Library)的方式集成到服务调用方进程里头，如果企业内有多种不同的语言栈，就要配合开发多种不同的客户端，有一定的研发和维护成本。另外，一旦客户端跟随服务调用方发布到生产环境中，后续如果要对客户库进行升级，势必要求服务调用方修改代码并重新发布，所以该方案的升级推广有不小的阻力。

**进程内LB的案例**是Netflix的开源服务框架，对应的组件分别是：Eureka服务注册表，Karyon服务端框架支持服务自注册和健康检查，Ribbon客户端框架支持服务自发现和软路由。另外，阿里开源的服务框架Dubbo也是采用类似机制。

**主机独立LB进程方案**

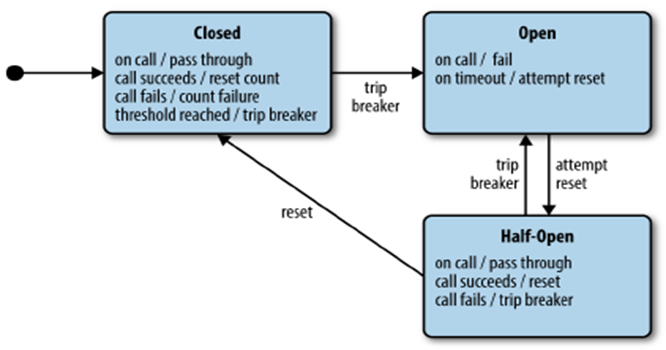
该方案是针对第二种方案的不足而提出的一种折中方案，原理和第二种方案基本类似，不同之处是，他将LB和服务发现功能从进程内移出来，变成主机上的一个独立进程，主机上的一个或者多个服务要访问目标服务时，他们都通过同一主机上的独立LB进程做服务发现和负载均衡。



该方案的典型案例是Airbnb的SmartStack服务发现框架，对应组件分别是：Zookeeper作为服务注册表，Nerve独立进程负责服务注册和健康检查，Synapse/HAproxy独立进程负责服务发现和负载均衡。Google最新推出的基于容器的PaaS平台Kubernetes，其内部服务发现采用类似的机制。

##### （4.4）容错

介绍两种模式：   
**（A）电路熔断器（Circuit Breaker）**： 该模式的原理类似于电路熔断器，如果电路发生短路，熔断器能够主动熔断电路，以避免灾难性损失



1. 正常状态下，电路处于关闭状态（Closed），调用是直接传递给依赖服务的；
2. 如果调用出错，则进入失败计数状态；
3. 失败计数达到一定阈值后，进入熔断状态（Open），这时的调用总是返回失败；
4. 累计一段时间以后，保护器会尝试进入半熔断状态（Half-Open）；
5. 处于Harf-Open状态时，调用先被传递给依赖的服务，如果成功，则重置电路状态为“Closed”，否则把电路状态置为“Open”；

**（B）舱壁（Bulkheads）**：该模式像舱壁一样对资源或失败单元进行隔离，如果一个船舱破了进水，只损失一个船舱，其它船舱可以不受影响 。   
  
这种模式比较常见的思路为： 

1. 采用微服务是首选，比如Docker。Docker是进程隔离的，单个Docker失效不会影响其他Docker容器。
2. 把大的并行处理工作，由多个线程池来负荷分担。

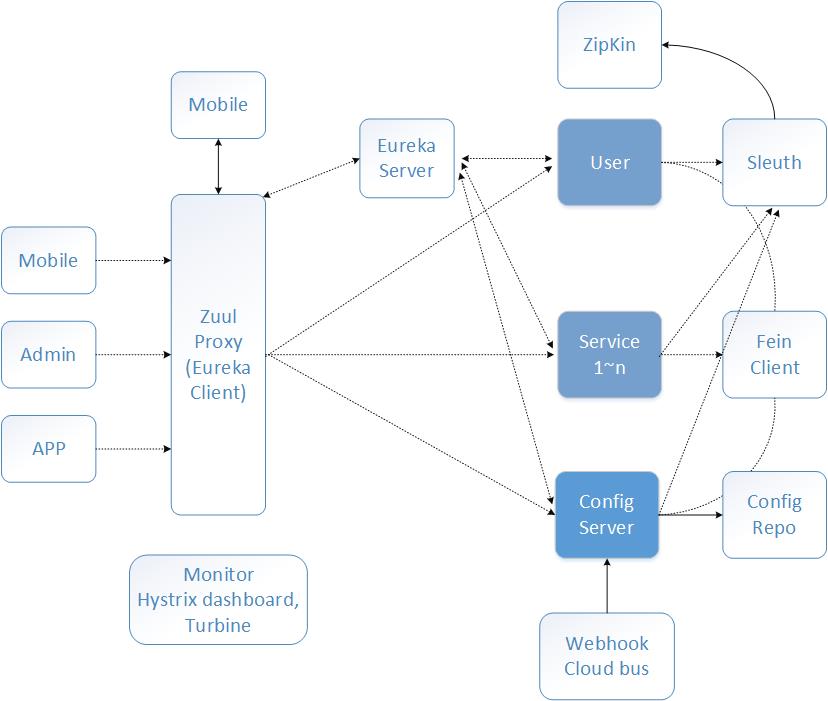
**API Gateway：**

1. 对设备侧（PC,Mobile等）提供简化的单一服务接口；
2. 它内部聚合后台几十甚至上百微服务。

价值：它的主要作用是简化设备侧开发的复杂度，减少微服务网络调用数量和网络延迟问题。

# Quick Start

http://start.spring.io/



## 服务注册发现

### 基于Eureka实现

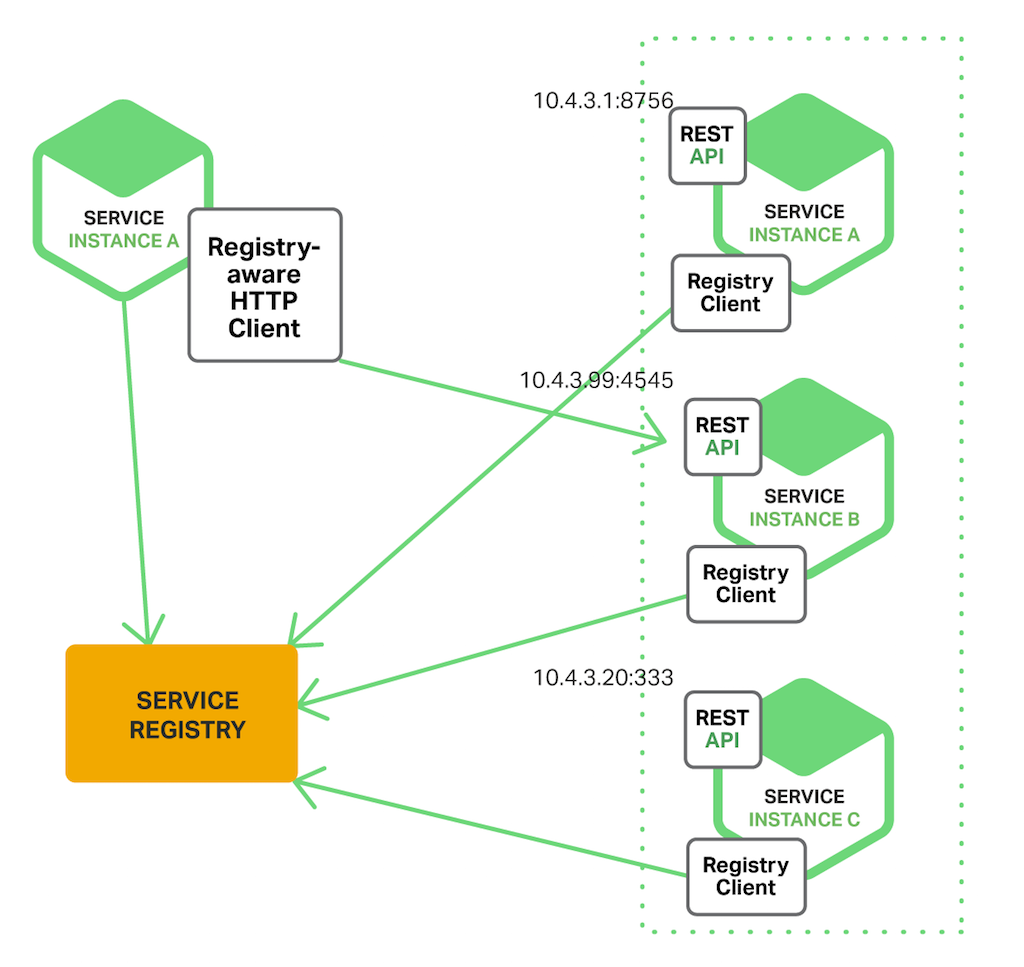
Eureka是微服务中的服务注册发现组件，负责服务的注册，以及提供服务。

在一个完整的系统架构中，任何单点的服务都不能保证不会中断，因此我们需要服务发现机制，在某个节点中断后，其它的节点能够继续提供服务，从而保证整个系统是高可用的。   
服务发现有两种模式：一种是客户端发现模式(Eureka,ZK)，一种是服务端发现模式(Consule)。

服务发现解释：http://blog.csdn.net/mr\_seaturtle\_/article/details/77618403

Erueka采用的是客户端发现模式。

客户端服务发现的架构如图：



Eureka Server会提供服务注册服务，各个服务节点启动后，会在Eureka Server中进行注册，这样Eureka Server中就有了所有服务节点的信息，并且Eureka有监控页面，可以在页面中直观的看到所有注册的服务的情况。同时Eureka有心跳机制，当某个节点服务在规定时间内没有发送心跳信号时，Eureka会从服务注册表中把这个服务节点移除。Eureka还提供了客户端缓存的机制，即使所有的Eureka Server都挂掉，客户端仍可以利用缓存中的信息调用服务节点的服务。Eureka一般配合Ribbon进行使用，Ribbon提供了客户端负载均衡的功能，Ribbon利用从Eureka中读取到的服务信息，在调用服务节点提供的服务时，会合理的进行负载。   
Eureka通过心跳检测、健康检查、客户端缓存等机制，保证了系统具有高可用和灵活性。

### Eureka服务端（服务中心）

在<http://start.spring.io/>中，输入eureka server快速生成eureka服务中心项目。

添加@EnableEurekaServer注解，再启动springboot项目，此时注册中心就已经启动完毕。

### Eureka客户端（服务生产者）

在<http://start.spring.io/>中，输入eureka discovery快速生成eureka服务客户端项目。

添加@EnableDiscoveryClient注解，启动springboot项目(此项目与其他Restful风格的springboot项目一致,只需要添加@EnableDiscoveryClien即可注册至服务中心)

## Eureka服务调用（服务消费者）

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，feign默认集成了ribbon。

### ribbon

ribbon是一个负载均衡客户端，可以很好的控制http和tcp的一些行为。

Ribbon工作时分为两步：第一步先选择Eureka Server，它优先选择在同一个Zone且负载较少的Server；第二步再根据用户指定的策略，在从Server取到的服务注册列表中选择一个地址。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。

**Simple**

Ribbon需要注册至eureka服务中心，通过eureka服务发现获取服务列表，ribbion负载均衡通过rule选择要调用的服务，所以工程需要引入eureka discovery与ribbon。

在<http://start.spring.io/>中，输入eureka discovery与ribbon快速生成eureka服务调用项目。

添加@EnableDiscoveryClient注解，启动springboot项目(此项目与其他Restful风格的springboot项目一致,只需要添加@EnableDiscoveryClien即可注册至服务中心)

### Feign

Feign采用接口注解的形式，简化了Http客户端，使客户端可以基于接口进行开发。Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

**Simple**

在<http://start.spring.io/>中，输入eureka discovery与feign快速生成eureka服务调用项目。

添加@EnableDiscoveryClient注解与@EnableFeignClients，启动springboot项目(此项目与其他Restful风格的springboot项目一致,只需要添加@EnableDiscoveryClien即可注册至服务中心)

声明@EnableFeignClients，启用feign客户端。

定义接口与controller



## 服务网关（Zuul）

Zuul主要功能是路由转发和过滤器，为API服务提供网关功能，你可以理解为他是一个nginx（路由转发）+lua（自定义代码实现过滤器）的java实现。它可以用来提供动态路由、监控、授权、安全、调度等等的边缘服务(edge service)，Zuul默认和ribbon实现的负载均衡的功能。

Zuul基于groovy语言可以实现动态编译。

### 路由配置

#### 配置文件中配置

配置/api-a-url请求都转发至localhost:8080(ribbon+restTempalte) /api-b-url/请求都转发至localhost:8081(feign)

zuul.routes.api-a-url.path=/api-a-url/\*\*

zuul.routes.api-a-url.url=http://localhost:8080/

zuul.routes.api-b-url.path=/api-b-url/\*\*

zuul.routes.api-b-url.url=http://localhost:8081/

此种方式相当于在配置文件中写死了配置，无法扩展。所以引出了配置2

#### 基于服务中心的服务映射（serviceId）

eureka中已经有我们的配置信息，那我们可以借助他来完成地址映射。

zuul.routes.api-a.path=/api-a/\*\*

#ribbon+restTempalte的实现方式

zuul.routes.api-a.serviceId= eureka-consumer

zuul.routes.api-b.path=/api-b/\*\*

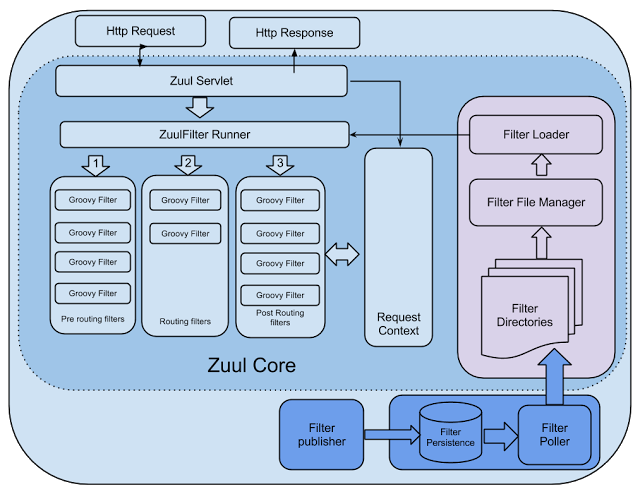
#feign实现方式的消费者

zuul.routes.api-b.serviceId= eureka-consumer-feign

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

### 过滤器

#### 过滤机制

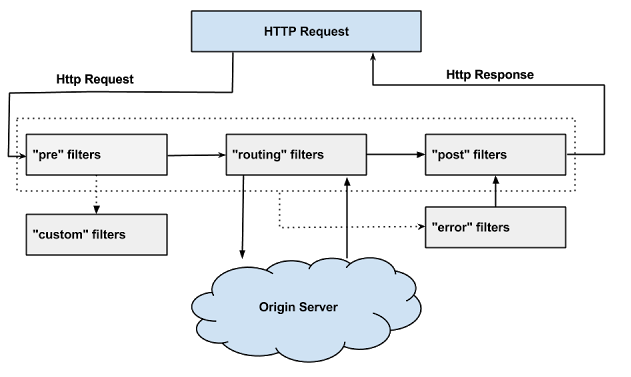


zuul把Request route到 用户处理逻辑 的过程中，这些filter参与一些过滤处理，比如

* 验证与安全保障: 识别面向各类资源的验证要求并拒绝那些与要求不符的请求。
* 审查与监控: 在边缘位置追踪有意义数据及统计结果，从而为我们带来准确的生产状态结论。
* 动态路由: 以动态方式根据需要将请求路由至不同后端集群处。
* 压力测试: 逐渐增加指向集群的负载流量，从而计算性能水平。
* 负载分配: 为每一种负载类型分配对应容量，并弃用超出限定值的请求。
* 静态响应处理: 在边缘位置直接建立部分响应，从而避免其流入内部集群。
* 多区域弹性: 跨越AWS区域进行请求路由，旨在实现ELB使用多样化并保证边缘位置与使用者尽可能接近。

#### 生命周期

过滤器的声明周期，和类似aop。



#### Simple

在<http://start.spring.io/>中，输入eureka discovery与zuul快速生成zuul服务网关项目。

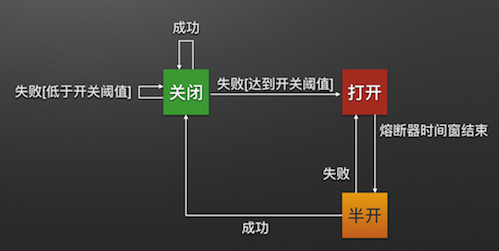
添加@EnableDiscoveryClient注解与@EnableZuulProxy，启动springboot项目(此项目与其他Restful风格的springboot项目一致,只需要添加@EnableDiscoveryClien即可注册至服务中心)

声明@ EnableZuulProxy，启用zuul路由之后基于服务中心的serviceId配置路由，编写filter进行具体过滤的操作。

## 熔断器 （hystrix）

断路器很好理解, 当Hystrix Command请求后端服务失败数量超过一定比例(默认50%), 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. 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). Hystrix的断路器就像我们家庭电路中的保险丝, 一旦后端服务不可用, 断路器会直接切断请求链, 避免发送大量无效请求影响系统吞吐量, 并且断路器有自我检测并恢复的能力.

熔断器开关相互转换的逻辑如下图：



### 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. 但是带来的代价就是维护多个线程池会对系统带来额外的性能开销. 如果是对性能有严格要求而且确信自己调用服务的客户端代码不会出问题的话, 可以使用Hystrix的信号模式(Semaphores)来隔离资源.

### HystrixCommand

HystrixCommand 表明该方法为hystrix包裹，可以对依赖服务进行隔离、降级、快速失败、快速重试等等hystrix相关功能

该注解属性较多，下面讲解其中几个

fallbackMethod 降级方法

commandProperties 普通配置属性，可以配置HystrixCommand对应属性，例如采用线程池还是信号量隔离、熔断器熔断规则等等

ignoreExceptions 忽略的异常，默认HystrixBadRequestException不计入失败

groupKey() 组名称，默认使用类名称

commandKey 命令名称，默认使用方法名

### Dashboard

开启dashboard需要引入

<!-- 监控页面 -->

<dependency>

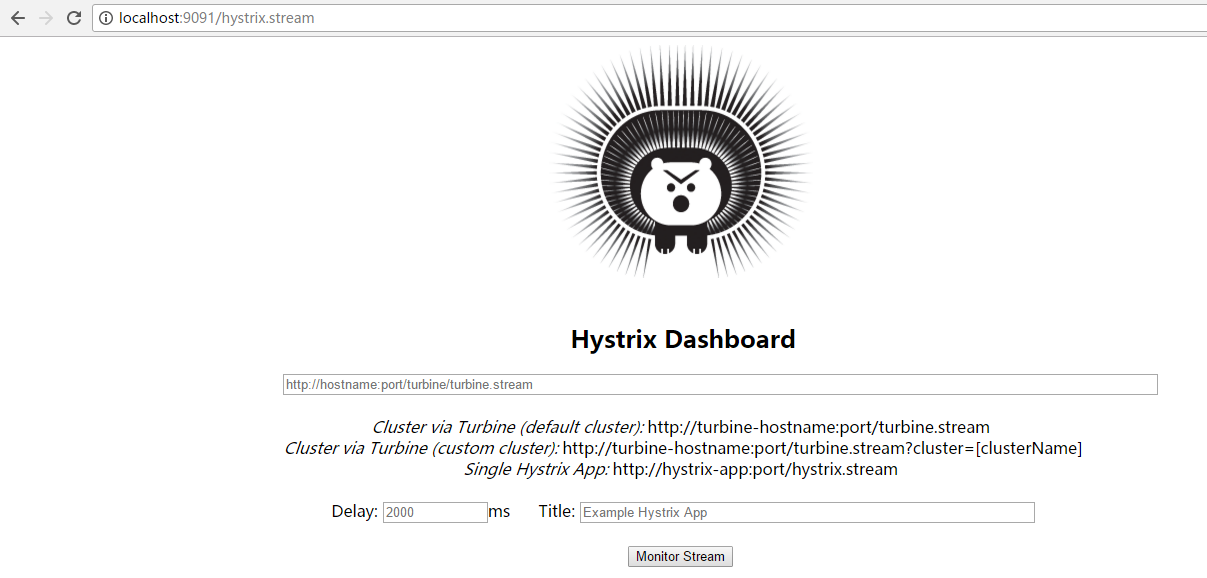
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

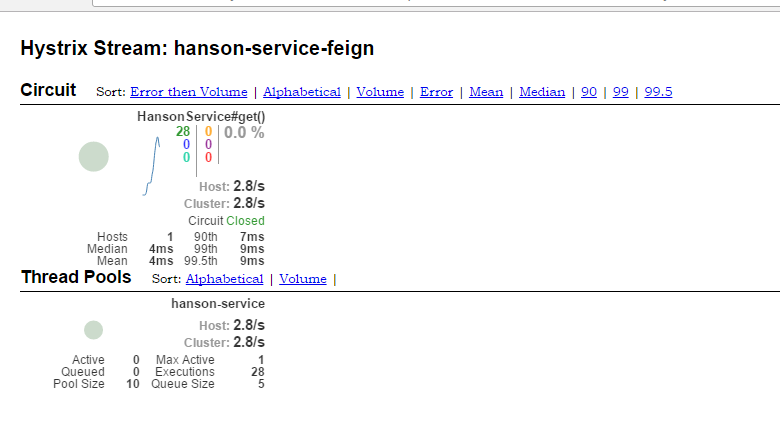
</dependency>

运行工程，可以访问 http://localhost:9090/hystrix.stream 获取dashboard信息，默认最大打开5个终端获取监控信息，可以增加delay参数指定获取监控数据间隔时间

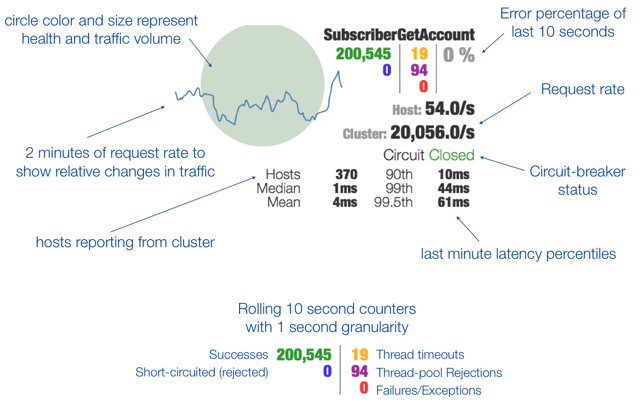
直接访问hystrix.stream肯定是不明智的，官方提供监控.war包,下载后放入tomcat中，得到如下界面



输入<http://localhost:9090/hystrix.stream>，以及监控间隔以及名称，之后会看到如下页面



页面详解



### Simple

#### Ribbon实现

在<http://start.spring.io/>中，输入eureka discovery、ribbon、hystrix、hystrix dashboard快速生成ribbon-hystrix服务调用项目。

与正常的ribbon实现一致，只是需要在请求的方法前添加

@HystrixCommand(fallbackMethod = "fireError")注解，声明当熔断触发时，回调的方法。

#### Feign实现

在<http://start.spring.io/>中，输入eureka discovery、feign、hystrix、hystrix dashboard快速生成feign-hystrix服务调用项目。

Feign是自带断路器的，在D版本的Spring Cloud中，它没有默认打开。需要在配置文件中配置打开它，在配置文件加以下代码：

feign.hystrix.enabled=true

改造之前的代码，在HansonService 接口的feignClient注解上添加fallback指定一个此接口的熔断处理类。

@FeignClient(value = "hanson-service",fallback = HansonServiceHystrix.class )

HansonServiceHystrix，实现HansonService接口，添加熔断发生后处理逻辑。

## 配置中心(Spring Cloud Config)

在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是config server，二是config client。

#### 服务端(Config Server)

#### Simple

在<http://start.spring.io/>中，输入config server

添加@EnableConfigServer，配置文件指定git地址与路径即可。

在git上上传配置文件，其中有配置

如果为gitlab需要在server.uri结尾处添加.git

init.failed.exit: false

访问

http://localhost:8888/var-dev.properties

返回

{"name":"ureka.server.renewalPercentThreshold","profiles":["dev"],"label":null,"version":"a68876a6211369bae723348d5f8c3defe4a55e04","state":null,"propertySources":[ init.failed.exit:0.85

]}

HTTP服务具有以下格式的资源：

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

证明已经可以从远端git获取配置。

#### 客户端(Config Client)

#### Simple

在<http://start.spring.io/>中，输入config client

@Value("${init.failed.exit}") // 读取gitlab配置文件中的属性，如果我们读取到了值，说明客户端是OK的

@RefreshScope //注解@RefreshScope指示Config客户端在服务器配置改变时，也刷新注入的属性值

spring.application.name需要与git上的{application}一致，如果spring.application.name和git上的属性文件名对应不上，需要配置spring.cloud.config.name: 文件名不包括profile

刷新：

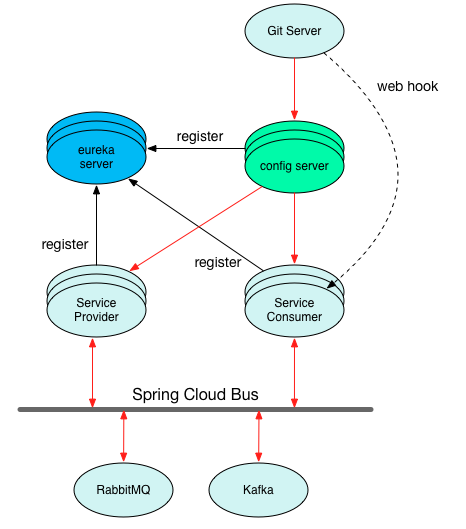
http://localhost:8080/refresh 手动向这个地址，发一个post请求（可以用postman或 curl -d '' http://localhost:8080/refresh)，可以看到

## 配置中心高可用（Spring Cloud Config+Eureka+Spring Cloud Bus）

配置中心也可像其他springcloud中的服务一样，注册在服务中心，基于eureka可实现配置中心的高可用。

基于git的web hook与Springcloud bus可实现配置变更后推送至相应服务。

整体架构如图，



#### 配置中心修改

基于[配置中心](#_配置中心(Spring_Cloud_Config))进行改造，首先将Config Server引入eureka注册至服务中心。

添加eureka客户端依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

修改Springcloud.version为<spring-cloud.version>Edgware.SR2</spring-cloud.version>

代码添加@EnableDiscoveryClient，使其注册至服务中心。

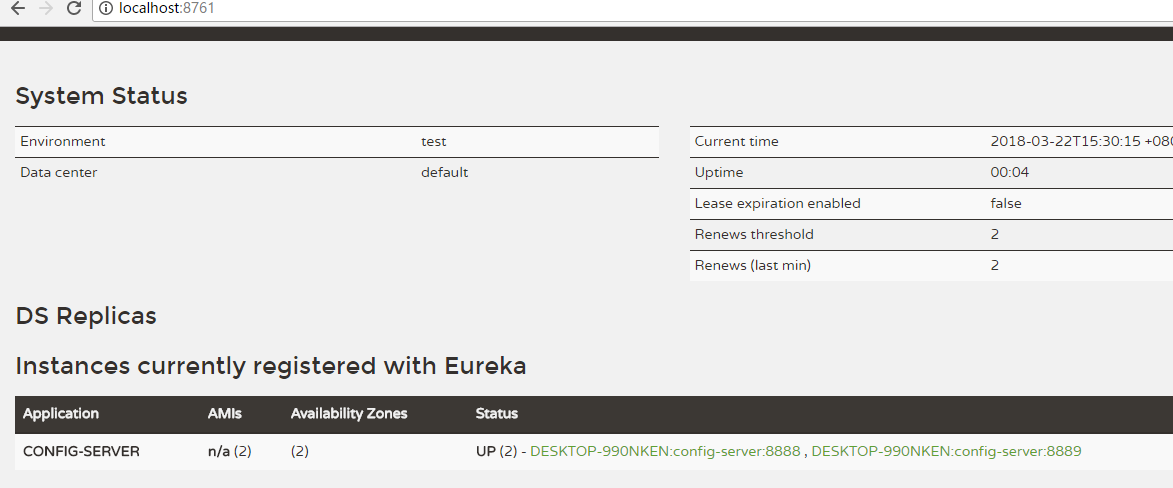
添加eurekaserver的相关配置

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

启动多服务时可在vm中添加参数，配置不同端口。

--server.port=8889

此时服务中心已经



#### 配置中心客户端修改

基于[配置中心](#_配置中心(Spring_Cloud_Config))进行客户端改造，首先将Config Client引入eureka注册至服务中心。

添加eureka客户端依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

修改

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.10.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

以及

Springcloud.version为<spring-cloud.version>Edgware.SR2</spring-cloud.version>

添加eurekaserver的相关配置

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/

#指定从配置中心读取文件

spring.cloud.config.discovery.enabled=true  
#配置中心的servieId，即服务名

spring.cloud.config.discovery.serviceId=config-server

访问<http://localhost:8080/> ，此时已经通过配置中心返回了配置。

#### 配置更新自动发布 （SpringCloud Bus+git webHook）

Spring Cloud Bus 将分布式的节点用轻量的消息代理连接起来。它可以用于广播配置文件的更改或者服务之间的通讯，也可以用于监控。本文要讲述的是用Spring Cloud Bus实现通知微服务架构的配置文件的更改。SringCloud Bus可以依赖kafka和rabbitmq通讯，本例我们使用rabbitmq进行通讯。