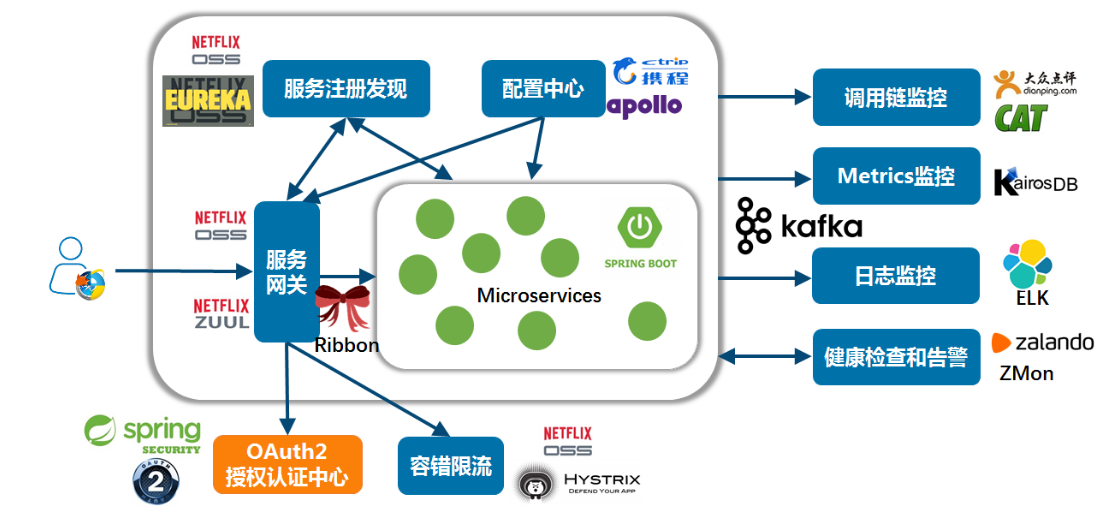
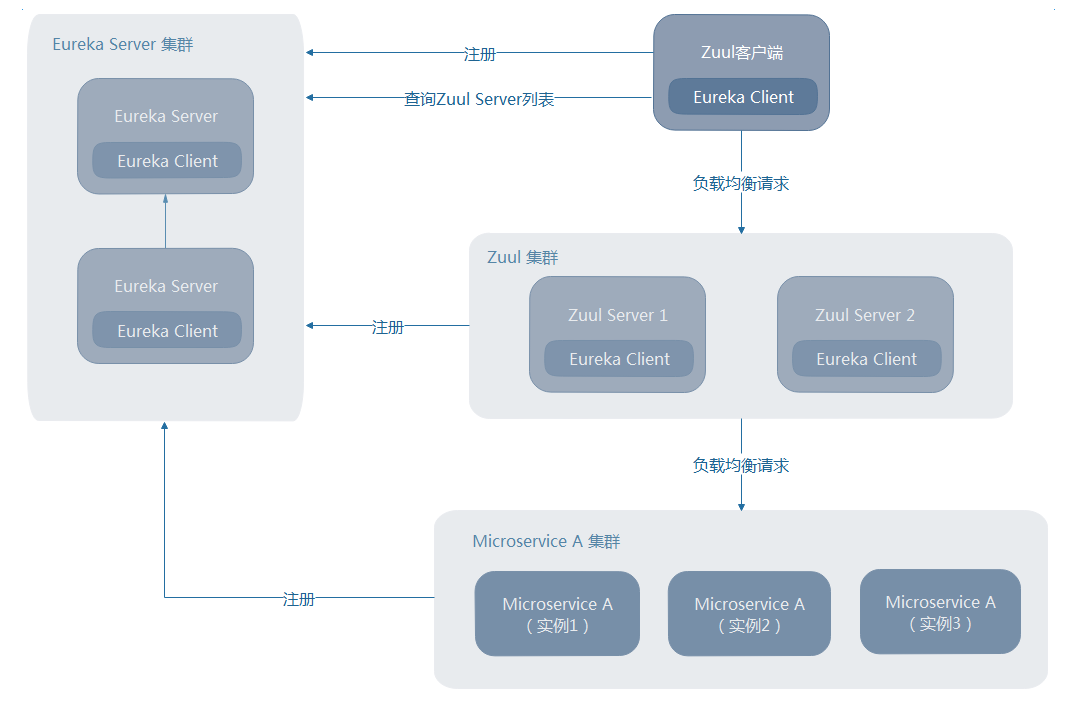
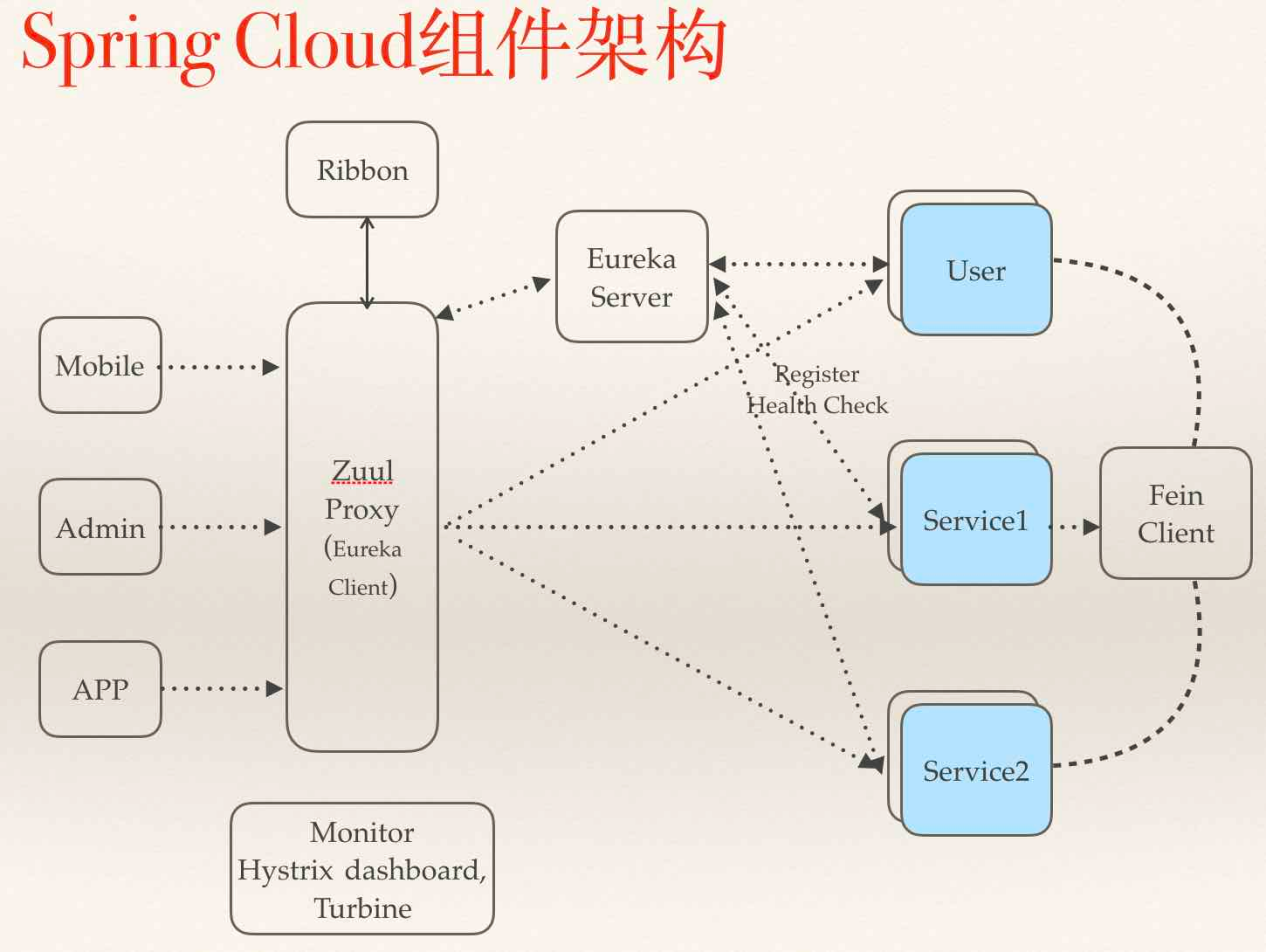
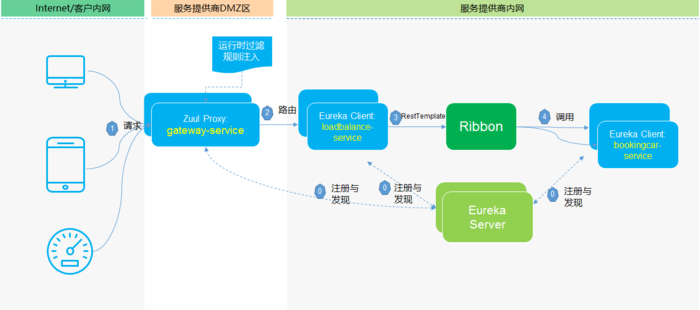
# 前言







<http://localhost:8763/hi?message=HelloSpringCloud> 对项目进行访问

<http://localhost:9411/zipkin/>

<http://localhost:8761/>

[http://localhost:8084/#/about](http://localhost:8084/" \l "/about)

#以 /api/a 开头的请求都转发给 hello-spring-cloud-web-admin-ribbon 服务

#以 /api/b 开头的请求都转发给 hello-spring-cloud-web-admin-feign 服务

[http://localhost:8769/api/b/hi](http://localhost:8769/api/b/hi?message=chenjiwei)

实践记录：

\* 为开启MyFallbackProvider.java的情况下

\* 访问（api/b/hi）web-admin-feign，跑到了service-admin的方法中去了

\* 访问（api/a/hi）web-admin-ribbon，内部设置除0异常，方法出错，不做任何反应

\* 访问（api/a/hi）web-admin-ribbon，内部正常，跳转到AdminService.java的fallbackMethod方法中，此方法是报错的。这没问题就跑报错方法，得探个究竟

hello-spring-cloud-service-admin

EurekaApplication > ServiceAdminApplication > WebAdminRibbonApplication > WebAdminFeignApplication > ZuulApplication 各服务

有了Eureka服务注册发现、Hystrix断路器、Ribbon服务调用负载均衡，以及spring cloud config 集群配置中心，似乎一个微服务框架已五脏俱全，last but not least，一个服务网关却不可或缺。

## Spring Cloud

引用（说明使用springcloud还是dubbo等其他案例）：

<https://blog.csdn.net/jlsdzhj/article/details/80661348?utm_medium=distribute.pc_aggpage_search_result.none-task-blog-2~all~sobaiduend~default-3-80661348.nonecase&utm_term=%E5%BE%AE%E6%9C%8D%E5%8A%A1%E6%9E%B6%E6%9E%84%E5%A6%82%E4%BD%95%E8%AE%A9%E5%89%8D%E7%AB%AF%E8%AE%BF%E9%97%AE>

云收藏一个参考项目：

<https://github.com/cloudfavorites/favorites-web>

参考项目：

<https://github.com/ityouknow/spring-cloud-examples>

### 是什么

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的云应用开发工具；

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。

微服务是可以独立部署、水平扩展、独立访问（或者有独立的数据库）的服务单元，Spring Cloud就是这些微服务的大管家，采用了微服务这种架构之后，项目的数量会非常多，Spring Cloud做为大管家就需要提供各种方案来维护整个生态。

Spring Cloud就是一套分布式服务治理的框架，既然它是一套服务治理的框架，那么它本身不会提供具体功能性的操作，更专注于服务之间的通讯、熔断、监控等。因此就需要很多的组件来支持一套功能，如果你对Spring Cloud组件不是特别了解的话，可以参考我的这篇文章：<https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247483752&idx=1&sn=d6f0c17e9028e74dafea617cda841d60&chksm=ebf6d917dc81500121d2cdc9fe7bfd2503339b8b6b39a1b626466589b80ca7d256ebee78c84d&scene=21#wechat_redirect>

### Spring Cloud的核心功能：

**\* 分布式/版本化配置**

**\* 服务注册和发现**

**\* 路由**

**\* 服务和服务之间的调用**

**\* 负载均衡**

**\* 断路器**

**\* 分布式消息传递**

### 组件配置使用运行流程：

1、请求统一通过API网关（Zuul）来访问内部服务.

2、网关接收到请求后，从注册中心（Eureka）获取可用服务

3、由Ribbon进行均衡负载后，分发到后端具体实例

4、微服务之间通过Feign进行通信处理业务

5、Hystrix负责处理服务超时熔断

6、Turbine监控服务间的调用和熔断相关指标

### 体系介绍

Spring Cloud 工具框架

1、Spring Cloud Config 配置中心，利用git集中管理程序的配置。

2、Spring Cloud Netflix 集成众多Netflix的开源软件

3、Spring Cloud Bus 消息总线，利用分布式消息将服务和服务实例连接在一起，用于在一个集群中传播状态的变化

4、Spring Cloud for Cloud Foundry 利用Pivotal Cloudfoundry集成你的应用程序

5、Spring Cloud Cloud Foundry Service Broker 为建立管理云托管服务的服务代理提供了一个起点。

6、Spring Cloud Cluster 基于Zookeeper, Redis, Hazelcast, Consul实现的领导选举和平民状态模式的抽象和实现。

7、Spring Cloud Consul 基于Hashicorp Consul实现的服务发现和配置管理。

8、Spring Cloud Security 在Zuul代理中为OAuth2 rest客户端和认证头转发提供负载均衡

9、Spring Cloud Sleuth SpringCloud应用的分布式追踪系统，和Zipkin，HTrace，ELK兼容。

10、Spring Cloud Data Flow 一个云本地程序和操作模型，组成数据微服务在一个结构化的平台上。

11、Spring Cloud Stream 基于Redis,Rabbit,Kafka实现的消息微服务，简单声明模型用以在Spring Cloud应用中收发消息。

12、Spring Cloud Stream App Starters 基于Spring Boot为外部系统提供spring的集成

13、Spring Cloud Task 短生命周期的微服务，为SpringBooot应用简单声明添加功能和非功能特性。

14、Spring Cloud Task App Starters

15、Spring Cloud Zookeeper 服务发现和配置管理基于Apache Zookeeper。

16、Spring Cloud for Amazon Web Services 快速和亚马逊网络服务集成。

17、Spring Cloud Connectors 便于PaaS应用在各种平台上连接到后端像数据库和消息经纪服务。

18、Spring Cloud Starters （项目已经终止并且在Angel.SR2后的版本和其他项目合并）

19、Spring Cloud CLI 插件用Groovy快速的创建Spring Cloud组件应用。

### 微服务、Spring Boot、Spring Cloud三者之间的关系

微服务是一种架构的理念，提出了微服务的设计原则，从理论为具体的技术落地提供了指导思想。

Spring Boot是一套快速配置脚手架，可以基于Spring Boot快速开发单个微服务；

Spring Cloud是一个基于Spring Boot实现的服务治理工具包；

Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架。

## Docker容器

## Dubbo

## EUREKA

引用：<https://blog.csdn.net/qwe86314/article/details/94552801>

引用：<https://blog.csdn.net/u012105931/article/details/104659073>

### 是什么

Eureka是Netflix组件的一个子模块，也是核心模块之一。云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

### 三元式

### 流程说明

\* Eureka客户端（以下简称客户端）启动后，定时向Eureka服务端（以下简称服务端）注册自己的服务信息（服务名、IP、端口等）；

\* 客户端启动后，定时拉取服务端以保存的服务注册信息；

\* 拉取服务端保存的服务注册信息后，就可调用和消费其他服务提供者提供的服务。

### 在以上流程中eureka做了什么

##### 客户端启动时如何注册到服务端？

源码分析看：<https://blog.csdn.net/u012105931/article/details/104659073>

Eureka客户端在启动时，首先会创建一个心跳的定时任务，定时向服务端发送心跳信息，服务端会对客户端心跳做出响应，如果响应状态码为404时，表示服务端没有该客户端的服务信息，那么客户端则会向服务端发送注册请求，注册信息包括服务名、ip、端口、唯一实例ID等信息。

##### 服务端如何保存客户端服务信息？

源码分析看：<https://blog.csdn.net/u012105931/article/details/104659073>

客户端通过Jersey框架（亚马逊的一个http框架）将服务实例信息发送到服务端，服务端将客户端信息放在一个ConcurrentHashMap对象中。

##### 客户端如何拉取服务端已保存的服务信息？

是需要使用的时候再去拉取，还是先拉取保存本地，使用的时候直接从本地获取？

客户端拉取服务端服务信息是通过一个定时任务定时拉取的，每次拉取后刷新本地已保存的信息，需要使用时直接从本地直接获取。

##### 如何构建高可用的Eureka集群？

首先，搭建一个高可用的Eureka集群，只需要在每个注册中心（服务端）

通过配置：

eureka.client.service-url.defaultZone指定其他服务端的地址，

多个使用逗号隔开，如：

eureka.client.service-url.defaultZone=http://localhost:10000/eureka/,http://localhost:10001/eureka/

在eureka的高可用状态下，这些注册中心是对等的，他们会互相将注册在自己的实例同步给其他的注册中心

注册中心收到注册信息后会判断是否是其他注册中心同步的信息还是客户端注册的信息，如果是客户端注册的信息，那么他将会将该客户端信息同步到其他注册中心去；否则收到信息后不作任何操作。通过此机制避免集群中信息同步的死循环。

##### 心跳和服务剔除机制是什么？

心跳机制：

客户端在启动时，首先会创建一个心跳的定时任务，定时向服务端发送心跳信息，告知服务端自己还活着，默认的心跳时间间隔是30秒。

服务剔除机制：

如果开启了自我保护机制，那么所有的服务，包括长时间没有收到心跳的服务（即已过期的服务）都不会被剔除；

如果未开启自我保护机制，那么将判断最后一分钟收到的心跳数与一分钟收到心跳数临界值比较，如果前者大于后者，且后者大于0的话，则启用服务剔除机制；

一旦服务剔除机制开启，则Eureka服务端并不会直接剔除所有已过期的服务，而是通过随机数的方式进行剔除，避免自我保护开启之前将所有的服务（包括正常的服务）给剔除。

##### Eureka自我保护机制是什么？

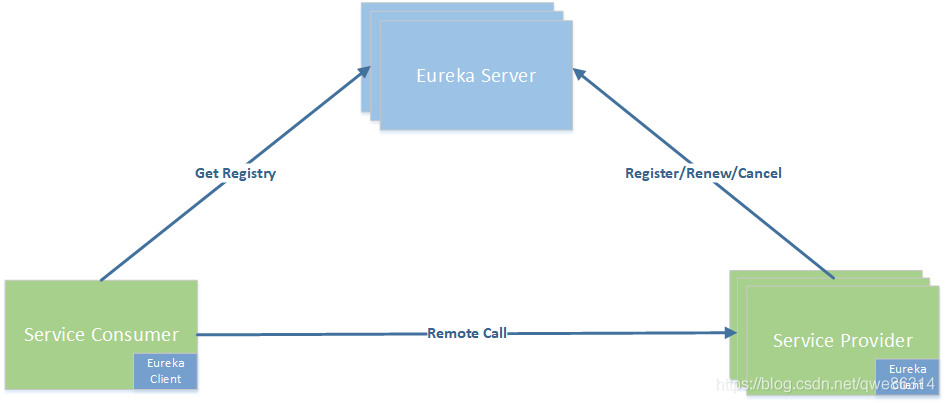
在分布式系统的CAP理论中，Eureka采用的AP，也就是Eureak保证了服务的可用性（A），而舍弃了数据的一致性（C）。当网络发生分区时，客户端和服务端的通讯将会终止，那么服务端在一定的时间内将收不到大部分的客户端的一个心跳，如果这个时候将这些收不到心跳的服务剔除，那可能会将可用的客户端剔除了，这就不符合AP理论。

### 相关概念

注册发现

Eureka 作为 Spring Cloud 体系中最核心、默认的注册中心组件

分为两个主体：Eureka Server 和 Eureka Client。



#### Eureka Server：注册中心服务端

提供了三个功能：

##### 服务注册

服务提供者启动时，会通过 Eureka Client 向 Eureka Server 注册信息，Eureka Server 会存储该服务的信息，Eureka Server 内部有二层缓存机制来维护整个注册表

##### 提供注册表

服务消费者在调用服务时，如果 Eureka Client 没有缓存注册表的话，会从 Eureka Server 获取最新的注册表

##### 同步状态

Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态。

#### Eureka Client：注册中心客户端

Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息。因此当所有的 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，但是当服务有更改的时候会出现信息不一致。

#### Register: 服务注册

服务的提供者，将自身注册到注册中心，服务提供者也是一个 Eureka Client。当 Eureka Client 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 Client 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 Client 运行正常，没有出现问题。 默认情况下，如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约，Server 端会将实例从其注册表中删除，此时间可配置，一般情况不建议更改。

服务续约的两个重要属性

服务续约任务的调用间隔时间，默认为30秒

eureka.instance.lease-renewal-interval-in-seconds=30

服务失效的时间，默认为90秒。

eureka.instance.lease-expiration-duration-in-seconds=90

Eviction 服务剔除

当 Eureka Client 和 Eureka Server 不再有心跳时，Eureka Server 会将该服务实例从服务注册列表中删除，即服务剔除。

#### Cancel: 服务下线

Eureka Client 在程序关闭时向 Eureka Server 发送取消请求。 发送请求后，该客户端实例信息将从 Eureka Server 的实例注册表中删除。

DiscoveryManager.getInstance().shutdownComponent()；

#### GetRegisty: 获取注册列表信息

Eureka Client 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

如果由于某种原因导致注册列表信息不能及时匹配，Eureka Client 则会重新获取整个注册表信息。 Eureka Server 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 Client 和 Eureka Server 可以使用 JSON/XML 格式进行通讯。在默认情况下 Eureka Client 使用压缩 JSON 格式来获取注册列表的信息。

#### Remote Call: 远程调用

当 Eureka Client 从注册中心获取到服务提供者信息后，就可以通过 Http 请求调用对应的服务；服务提供者有多个时，Eureka Client 客户端会通过 Ribbon 自动进行负载均衡。

#### 自我保护机制

#### 集群工作原理

集群的介绍暂时省略，看链接

#### Eurka 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

### 配置说明

| 配置参数 | 默认值 | 说明 |
| --- | --- | --- |
| ****eureka.instance**** | | |
| eureka.instance.prefer-ip-address | true | 不使用主机名来定义注册中心的地址，而使用IP地址的形式，如果设置了  eureka.instance.ip-address 属性，则使用该属性配置的IP，否则自动获取  除环路IP外的第一个IP地址 |
| eureka.instance.ip-address |  | IP地址 |
| eureka.instance.hostname |  | 设置当前实例的主机名称 |
| eureka.instance.appname |  | 服务名，默认取 spring.application.name 配置值，如果没有则为 unknown |
| eureka.instance.lease-renewal-interval-in-seconds | 30 | 定义服务续约任务（心跳）的调用间隔，单位：秒 |
| eureka.instance.lease-expiration-duration-in-seconds | 90 | 定义服务失效的时间，单位：秒 |
| eureka.instance.status-page-url-path | /info | 状态页面的URL，相对路径，默认使用 HTTP 访问，  如果需要使用 HTTPS则需要使用绝对路径配置 |
| eureka.instance.status-page-url |  | 状态页面的URL，绝对路径 |
| eureka.instance.health-check-url-path | /health | 健康检查页面的URL，相对路径，默认使用 HTTP 访问，  如果需要使用 HTTPS则需要使用绝对路径配置 |
| eureka.instance.health-check-url |  | 健康检查页面的URL，绝对路径 |
|  |  |  |
| ****eureka.server**** | | |
| eureka.server.enable-self-preservation | true | 是否开启自我保护，Eureka 会统计15分钟之内心跳失败的比例低于85%  将会触发保护机制，不剔除服务提供者，如果关闭服务注册中心将不可用的  实例正确剔除 |
| eureka.server.eviction-interval-timer-in-ms | 60 | server清理无效节点的时间间隔 |
| eureka.dashboard.enabled | true | 是否开启仪表盘 |
| eureka.dashboard.path | / | 仪表盘访问路径 |
|  |  |  |
| ****eureka.client**** | | |
| eureka.client.enabled | true | 是否开启client |
| eureka.client.service-url |  | 指定服务注册中心地址，类型为 HashMap，并设置有一组默认值，  默认的Key为 defaultZone；默认的Value为 http://localhost:8761/eureka  ，如果服务注册中心为高可用集群时，多个注册中心地址以逗号分隔。 如果服务注册中心加入了安全验证，这里配置的地址格式为： http://<username>:<password>@localhost:8761/eureka  其中 <username> 为安全校验的用户名；<password> 为该用户的密码 |
| eureka.client.fetch-registery | true | 是否检索服务 |
| eureka.client.registery-fetch-interval-seconds | 30 | 从Eureka服务器端获取注册信息的间隔时间，单位：秒 |
| eureka.client.register-with-eureka | true | 启动服务注册 |
| eureka.client.eureka-server-connect-timeout-seconds | 5 | 连接 Eureka Server 的超时时间，单位：秒 |
| eureka.client.eureka-server-read-timeout-seconds | 8 | 读取 Eureka Server 信息的超时时间，单位：秒 |
| eureka.client.filter-only-up-instances | true | 获取实例时是否过滤，只保留UP状态的实例 |
| eureka.client.eureka-connection-idle-timeout-seconds | 30 | Eureka 服务端连接空闲关闭时间，单位：秒 |
| eureka.client.eureka-server-total-connections | 200 | 从Eureka 客户端到所有Eureka服务端的连接总数 |
| eureka.client.eureka-server-total-connections-per-host | 50 | 从Eureka客户端到每个Eureka服务主机的连接总数 |

### 注册中心 Eureka 产品的使用

### 如何利用 Eureka 搭建单台和集群的注册中心

### 总结

讲了 Eureka 核心概念、Eureka 自我保护机制和 Eureka 集群原理。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，通过一些列的机制，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。

## 配置中心

看一下：<https://blog.csdn.net/wtdm_160604/article/details/83720391>

配置和搭建引用：<https://blog.csdn.net/pengjunlee/article/details/88061736>

概念和详细的使用过程引用：

<https://www.cnblogs.com/zhangjianbin/p/6262504.html>

不同配置中心产品的比较引用：

<https://baijiahao.baidu.com/s?id=1662923343412122321&wfr=spider&for=pc>

读不到远端配置文件解决：

<https://blog.csdn.net/u013100581/article/details/92759519>

<https://segmentfault.com/a/1190000018881380>

最优，配置中心服务端客户端的使用详细介绍以及借助spring clound bus的广播功能触发[http://localhost:8764/actuator/refresh](http://localhost:8764/refresh)更新：

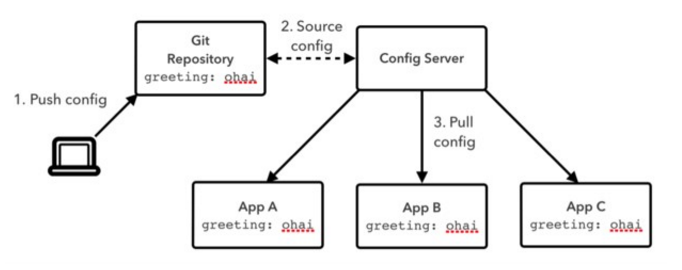
<https://www.cnblogs.com/fengzheng/p/11242128.html>

### 概念：

Spring Cloud Config为服务端和客户端提供了分布式系统的外部化配置支持。配置服务器为各应用的所有环境提供了一个中心化的外部配置。它实现了对服务端和客户端对Spring Environment和PropertySource抽象的映射，所以它除了适用于Spring构建的应用程序，也可以在任何其他语言运行的应用程序中使用。作为一个应用可以通过部署管道来进行测试或者投入生产，我们可以分别为这些环境创建配置，并且在需要迁移环境的时候获取对应环境的配置来运行。

置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。当然他也提供本地化文件系统的存储方式。

使用 spring Cloud 进行集中式配置管理，将以往的配置文件从项目中摘除后放到Git 或svn中集中管理，并在需要变更的时候，可以通知到各应用程序，应用程序刷新配置不需要重启。



### 基本原理：

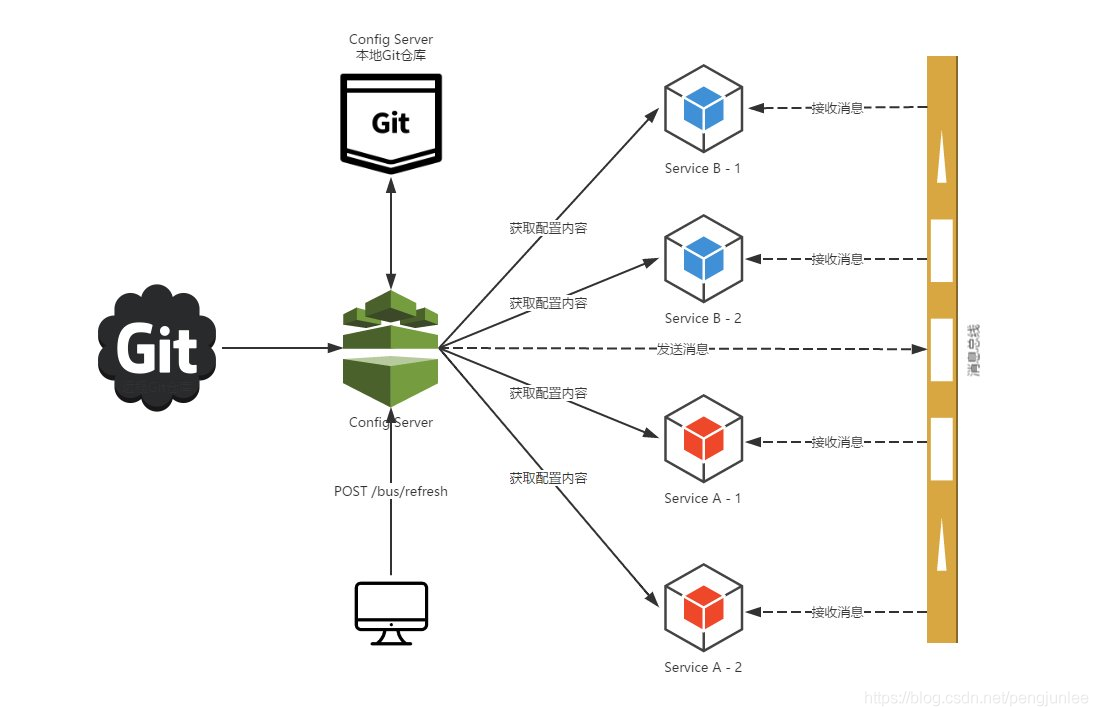
git 上存放我们的远程配置文件

config-server 连接到 git

config-client 连接到config-server

当我们启动config-client 服务的时候，client 会通过连接的 config-server 拿到远程git 上面的配置文件，然后通过 Spring 加载到对象中。

### 搭建配置中心：



引入依赖

<parent>

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

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

<version>2.0.6.RELEASE</version>

</parent>

<properties>

<spring-cloud.version>Finchley.SR2</spring-cloud.version>

</properties>

<dependencies>

<!-- Eureka-Client 依赖 -->

<dependency>

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

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

</dependency>

<!-- Config-Server 依赖 -->

<dependency>

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

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<!-- SpringCloud 版本控制依赖 -->

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

创建启动类

@SpringBootApplication

@EnableConfigServer

添加配置

server:

port: 9001

spring:

application:

name: config-center

eureka:

client:

serviceUrl:

defaultZone: <http://localhost:8761/eureka/>

spring:

cloud:

config:

server:

git:

uri: https://gitee.com/pengjunlee/config-cloud.git

username: 你的码云账号

password: 你的账号密码

创建Git仓库

客户端通过发送Http请求来从配置中心读取配置，这些Http请求的URI遵循以下规则：

/{name}-{profiles}.properties

/{name}-{profiles}.yml || /{name}-{profiles}.yaml

/{label}/{name}-{profiles}.properties

/{label}/{name}-{profiles}.json

/{name}/{profiles}/{label:.\*}

/{name}-{profiles}.json

/{label}/{name}-{profiles}.yml || /{label}/{name}-{profiles}.yaml

/{name}/{profiles:.\*[^-].\*}

/{name}/{profile}/{label}/\*\*

/{name}/{profile}/{label}/\*\*

/{name}/{profile}/\*\*

其中各个参数的含义如下：

name 服务的ID，即spring.application.name的值，本例中为 product-service；

profiles 激活的profile，通过spring.cloud.config.profile指定，本例中为 dev；

label 分枝的版本，通过spring.cloud.config.label指定，本例中为默认值 master；

在Git仓库的master分枝中创建product-service-dev.yml和product-service-test.yml两个文件，分别将服务的启动端口指定为8771和8772：

# product-service-dev.yml

server:

port: 8771

# product-service-test.yml

server:

port: 8772

### 搭建客户端：

依赖

<!-- Config-Client 依赖 -->

<dependency>

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

<artifactId>spring-cloud-config-client</artifactId>

</dependency>

或者

<!-- Starter-Config 依赖 -->

<dependency>

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

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

</dependency>

配置

# 设置服务(应用)名称

spring:

application:

name: product-service

# 指定用于获取配置的配置中心服务(应用)名称

cloud:

config:

discovery:

enabled: true

serviceId: config-center

profile: dev

# 指定分枝版本，默认为master

label: master

# 指定注册中心地址

eureka:

client:

serviceUrl:

defaultZone: http://localhost:8761/eureka/

### 如何在不重启 config-client 服务的情况下让项目中引用的配置信息更新呢？

config-client 工程中添加依赖监控模块，其中包含了/refresh刷新API

依赖

<dependency>

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

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

<optional>true</optional>

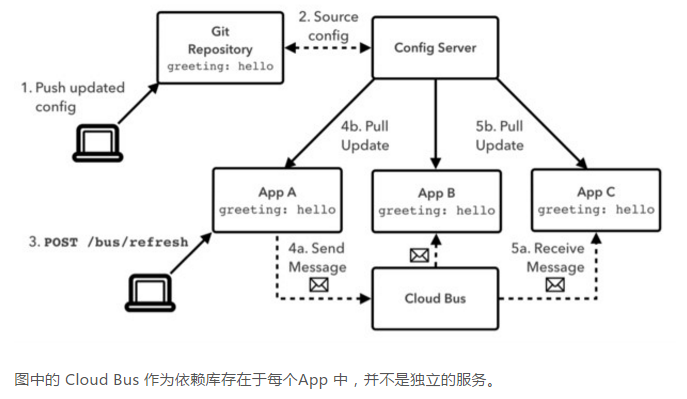
</dependency>

在需要自动更新配置变量的Java类上，使用注解 @RefreshScope 修饰

以后当我们再更新git上面的配置文件后，在 config-client 端执行POST 请求 http://localhost:8080/refresh 就可以更新刷新配置变量到内存中了。

如果我们 config-client 做的是分布式部署，岂不是要每一个机器都POST一下 /refresh 请求，这样显然不是最好的方法。

Spring Cloud Bus 为我们解决了这样的问题。



通过Spring Cloud Bus来实现以消息总线的方式进行通知配置信息的变化，完成集群上的自动化更新。

Cloud Bus 需要依赖AMQP、Redis、Kafka 这样的组件做为代理才可以使用。

### 不同产品的配置中心对比：

#### 第三方配置中心产品：

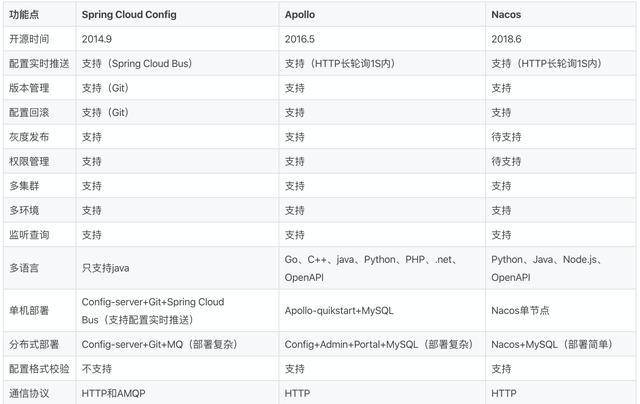
Spring Cloud Config： Spring Cloud 生态组件，可以和Spring Cloud体系无缝整合。

Apollo： 携程开源的配置管理中心，具备规范的权限、流程治理等特性。

Nacos： 阿里开源的配置中心，也可以做DNS和RPC的服务发现。

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

#### 产品功能特点比较：





总结：

压测环境：Nacos和Apollo使用同样的数据库（32C128G）部署Server服务的机器使用的8C16G配置的容器，磁盘是100G SSD。

Spring Cloud Config使用2.0.0.M9版本，Apollo使用1.2.0 release版本，Nacos使用0.5版本。

Spring Cloud Config 依赖git，使用局限性较大。

调研结果

### 应用

#### 配置客户端对配置中心云端git配置文件的获取

项目：

hello-spring-cloud-web-admin-feign

指向配置中心：

serviceId: hello-spring-cloud-config

配置文件：

配置客户端的配置需在bootstrap配置文件中配置

bootstrap.properties

spring.application.name=hello-spring-cloud-config

#spring.application.name=application

spring.cloud.config.profile=test

spring.cloud.config.label=master

spring.cloud.config.uri=http://localhost:8883/

application.yml

Spring:

cloud:

config:

#uri: http://localhost:8883

#name: web-admin-feign

discovery:

enabled: true

serviceId: hello-spring-cloud-config

label: master

profile: dev

依赖：

<!-- Config-Client 依赖 -->

<dependency>

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

<artifactId>spring-cloud-config-client</artifactId>

</dependency>

访问地址：

<http://localhost:8769/api/b/from?message=12345dfdsfds&a=123&b=123&token=45678>

问题：

@RefreshScope 控制类带有此注解会报错（异常找不到指定字段配置信息）

后来将文件前缀改为标准的文件名

‘application’解决

#### 配置客户端动态刷新本地bean与云端配置

项目

hello-spring-cloud-web-admin-feign

依赖

<!-- Config-Client 依赖 -->

<dependency>

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

<artifactId>spring-cloud-config-client</artifactId>

</dependency>

<!-- 应用监控 -->

<dependency>

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

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

</dependency>

云端对应配置文件

hello-spring-cloud-web-admin-feign.properties

hello-spring-cloud-web-admin-feign-dev.properties

hello-spring-cloud-web-admin-feign-prod.properties

hello-spring-cloud-web-admin-feign-test.properties

bootstrap.yml

启动test配置文件

spring:

profiles:

active: test

下方配置了三个环境的配置文件，分别是dev、test、prod

---

spring:

profiles: prod

application:

name: hello-spring-cloud-web-admin-feign

cloud:

config:

uri: http://localhost:8883

label: master

profile: prod

---

spring:

profiles: dev

application:

name: hello-spring-cloud-web-admin-feign

cloud:

config:

uri: http://localhost:8883

label: master

profile: dev

---

spring:

profiles: test

application:

name: hello-spring-cloud-web-admin-feign

cloud:

config:

#uri: http://localhost:8883

discovery:

enabled: true

serviceId: hello-spring-cloud-config

label: master

profile: test

application.yml配置文件

配置刷新接口权限[http://localhost:8764/actuator/refresh](http://localhost:8764/refresh)

management:

endpoint:

shutdown:

enabled: false

security:

Enabled: false

endpoints:

web:

exposure:

include: "\*"

实现刷新接口

在TestController.java 中Post请求[http://localhost:8764/actuator/refresh](http://localhost:8764/refresh)进行刷新

@RequestMapping("/refresh")

public String refresh() {

// 通过一个post方法调用 http://localhost:8764/actuator/refresh

// 去动态刷新bean容器的实体数据与云端配置

log.info("{} http://localhost:"+port+"/actuator/refresh", "发起请求");

new Refresh(port).start();

return "已刷新";

}

#### spring clound bus的广播功能触发[http://localhost:8764/actuator/refresh](http://localhost:8764/refresh)更新

## Zipkin

分布式跟踪系统，用于收集服务的定时数据，以解决微服务架构中的延迟问题。

包括数据的收集、存储、查找和展现

<http://localhost:9411> 此链接可以浏览服务追踪界面

## Zuul

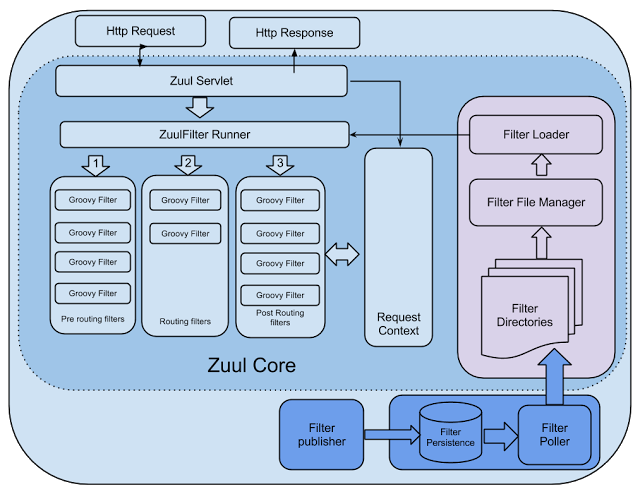
引用：<https://www.cnblogs.com/lexiaofei/p/7080257.html>

源码分析：<https://www.cnblogs.com/throwable/p/9653067.html>

如何去使用：<https://zhuanlan.zhihu.com/p/138943446>

多种使用方式，文件上传、回退等等：

<https://blog.csdn.net/u011063112/article/details/81428871>



### 是什么

是netflix开源的一个API Gateway 服务器, 本质上是一个web servlet应用

在云平台上提供动态路由，监控，弹性，安全等边缘服务的框架。Zuul 相当于是Web 网站后端所有请求的前门。

ribbon是对服务之间调用做负载，是服务之间的负载均衡，zuul是可以对外部请求做负载均衡。

Zuul包含了对请求的路由和过滤两个最主要的功能:

1. 其中路由功能负责将外部请求转发到具体的微服务实例上,是实现外部访问统一入口的基础而过滤器功能则负责对请求的处理过程进行干预,是实现请求校验、服务聚合等功能的基础.
2. Zuul和Eureka进行整合,将Zuul自身注册为Eureka服务治理下的应用,同时从Eureka中获得其他微服务的消息,也即以后的访问微服务都是通过Zuul跳转后获得.

1、过滤器机制

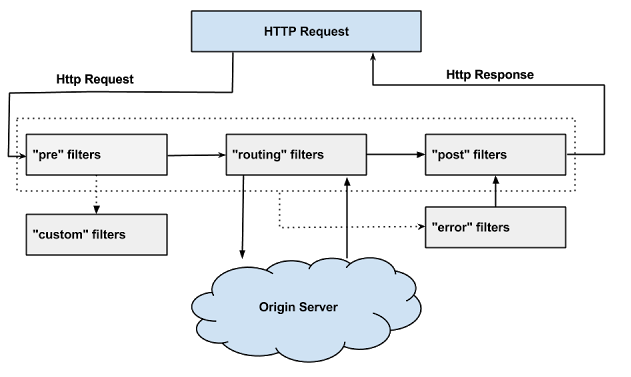
核心是一系列的filters, 其作用可以类比Servlet框架的Filter，或者AOP

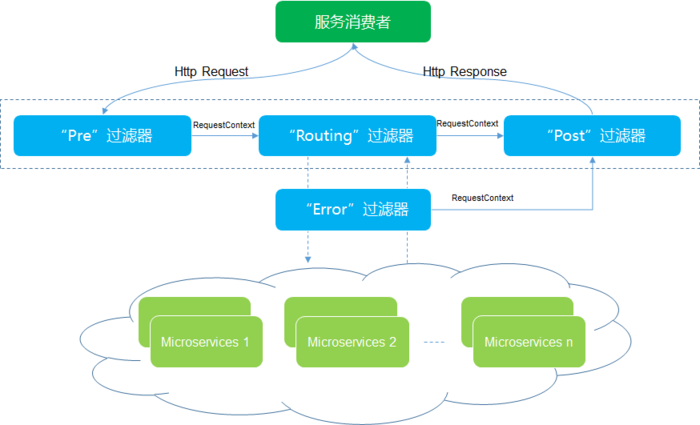
2、过滤器的生命周期

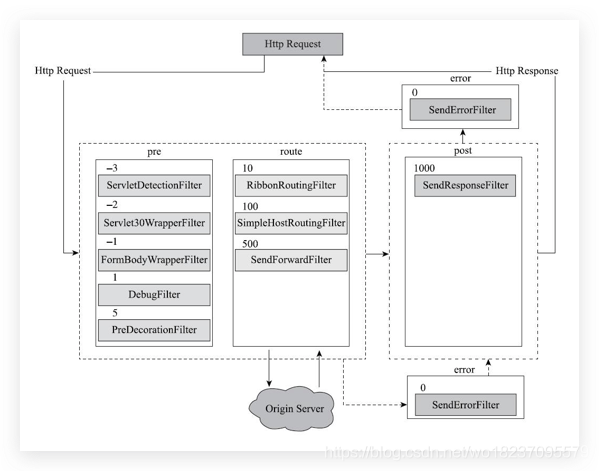
Zuul提供了一个框架，可以对过滤器进行动态的加载，编译，运行。

Zuul的过滤器之间没有直接的相互通信，他们之间通过一个RequestContext的静态类来进行数据传递的。RequestContext类中有ThreadLocal变量来记录每个Request所需要传递的数据。

Zuul的过滤器是由Groovy写成，这些过滤器文件被放在Zuul Server上的特定目录下面，Zuul会定期轮询这些目录，修改过的过滤器会动态的加载到Zuul Server中以便过滤请求使用









此图是对上图的说明，都是一些原生的过滤器，

不想使用可以通过配置去替代，

表达式：zuul.<SimpleClassName>.<flterType>.disable=true

实例：zuul.SendErrorFilter.error.disable=true

3、过滤器类型

(1) PRE：

在 Zuul 按照规则路由到下级服务之前执行。如果需要对请求进行预处理，比如鉴权、限流等

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。

1. ROUTING：

这类 Filter 是 Zuul 路由动作的执行者，是 Apache Http Client 或 Netflix Ribbon 构建和发送原始 HTTP 请求的地方，目前已支持 Okhttp

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。

1. POST：

这类 Filter 是在源服务返回结果或者异常信息发生后执行的，如果需要对返回信息做一些处理，则在此类 Filter 进行处理

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。

1. ERROR：

在整个生命周期内如果发生异常，则会进入 error Filter，可做全局异常处理。

在其他阶段发生错误时执行该过滤器。

（5）内置的特殊过滤器：

zuul还提供了一类特殊的过滤器，分别为：

StaticResponseFilter和SurgicalDebugFilter

StaticResponseFilter：此过滤器允许从Zuul本身生成响应，而不是将请求转发到源。

SurgicalDebugFilter：此过滤器允许将特定请求路由到分隔的调试集群或主机。

### Zuul的作用

Zuul可以通过加载动态过滤机制，从而实现以下各项功能：

\* 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理。

\* 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。

### Zuul和nginx的区别

相同点：Zuul和Nginx都可以实现负载均衡、反向代理（隐藏真实ip地址），过滤请求，实现网关的效果

不同点:

Nginx–c语言开发

Zuul–java语言开发

Zuul负载均衡实现：采用ribbon+eureka实现本地负载均衡

Nginx负载均衡实现：采用服务器实现负载均衡

Nginx相比zuul功能会更加强大，因为Nginx整合一些脚本语言（Nginx+lua）

Nginx适合于服务器端负载均衡

Zuul适合微服务中实现网关

### 限流

#### 为什么

1. 防止不需要频繁请求服务的请求恶意频繁请求服务，造成服务器资源浪费。
2. 防止不法分子恶意攻击系统，击穿系统盗取数据，防止数据安全隐患。

3、防止系统高峰时期，对系统频繁访问，给服务器带来巨大压力。

限流策略

#### 怎么做

流算法：

\* 漏桶: leakey bucket,原理：桶的下方的小孔会以一个相对恒定的速率漏水，而不管入桶的水流量，这样就达到了控制出水口的流量

\* 令牌桶: token bucket,原理：以相对恒定的速率向桶中加入令牌，请求来时于桶中取令牌，取到了就放行，没能取到令牌的请求则丢弃

限流粒度

粗粒度：

\* 网关限流

\* 单个服务

### 细粒度

user: 认证用户或者匿名，针对某个用户粒度进行限流

origin: 客户机的IP,针对请求客户机的IP进行限流

url: 特定url,针对请求的url粒度进行限流

serviceId: 特定服务，针对某个服务的id粒度进行限流

### 应用：

类似于Nginx在应用服务最前端添加一堵保护墙，zuul的负载均衡是针对将请求分发给集群中某台服务或者某个服务实例。而前面介绍过的ribbon也是主打服务负载功能，它所针对的是服务消费者将调用请求分发到某具体服务提供实例。两者均做负载均衡，实际是在系统不同的层级上进行。

#### 自定义网关Fileter

extends ZuulFilter 继承此类

@Override

public String filterType() {

return PRE\_TYPE;

}

@Override

public int filterOrder() {

return 0;

}

@Override

public boolean shouldFilter() {

return true;

}

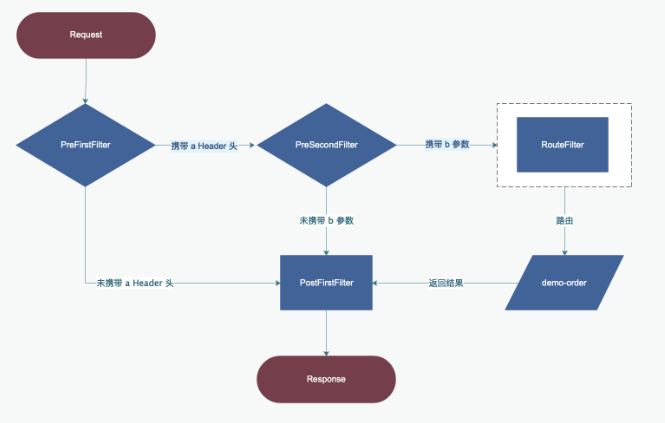
@Override

public Object run() throws ZuulException {

log.info("经过第一个 pre 过滤器");

return null;

}



可从上图看出post过滤器是必经之路

如上图是请求过程，定义了LoginFilter.java、PreSecondFilter.java、PostFirstFilter.java 三个类，当前排列顺序也是过滤顺序。每个类都设置了一些拦截情况，不符合过滤条件将会返回错误提示。

所有拦截器的交互都靠这个RequestContext进行交互

##### ZuulFilter接口

filterType()

四种不同生命周期的过滤器类型，详情看上文

filterOrder()

过滤顺序，越大越靠后过滤

shouldFilter() boolean

配置是否需要过滤

run()

具体过滤逻辑

##### 基本API

获取RequestContext

RequestContext ctx = RequestContext.getCurrentContext();

获取HttpServletRequest

ctx.getRequest()

传递数据

ctx.set(SessionContants.LOGIC\_IS\_SUCCESS, false);

更改响应状态

ctx.setResponseStatusCode(200);

设置返回信息

ctx.setResponseBody(String.format(SessionContants.ERROR\_RESPONSE\_BODY, "a Header头不足"));

设置返回编码

ctx.getResponse().setContentType("application/json; charset=utf-8");

ctx.getResponse().setCharacterEncoding("UTF-8");

禁止访问下游拦截器

ctx.setSendZuulResponse(false);

##### 配置

zuul:

routes:

api-a:

path: /api/a/\*\*

serviceId: hello-spring-cloud-web-admin-ribbon

api-b:

path: /api/b/\*\*

serviceId: hello-spring-cloud-web-admin-feign

##### 依赖

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-cloud-starter-netflix-zuul</artifactId>

</dependency>

#### zuul动态路由

通过导入数据库中的路由数据，

hello-spring-cloud-zuul 项目

DiscoveryRouteLocator.java

该类继承SimpleRouteLocator 重写locateRoutes()方法，重新定义路由的收集方式，即可通过配置文件配置路由，也可以读取数据库中的路由信息配置路由，实现动态路由的效果

ZuulRouteEntity.java

路由信息实体类

DiscoveryConfig.java

该类注解了一个@Configuration，对DiscoveryRouteLocator类创建bean

RefreshRouteService.java

对路由进行刷新

#### 配置文件中路由的控制

方式1：

通过实际的url去指定路由

zuul:

routes:

users:

path: /user/\*\*

url: <http://example.com/users_service>

方式2：

通过serviceId去指定路由

zuul:

routes:

users:

path: /user/\*\*

serviceId: users-service

#### 网关默认回退FallbackProvider无效和错误过滤的问题

Zuul默认已经整合了Hystrix, 而且如果启动了Dashborad，也可以知道Zuul对Hystrix监控的粒度是微服务，而不是某一个API; 同时也说明所有经过Zuul的请求都会被Hystrix保护起来。

实现回退需要实现ZuulFallbackProvider接口。

测试了一下，当服务没有启动，但是又被访问到，那么就会进行断路，返回预定好的内容

public class MyFallbackProvider implements FallbackProvider {

private Logger log = LoggerFactory.getLogger(MyFallbackProvider.class);

@Override

public String getRoute() {

// return "hello-spring-cloud-web-admin-feign"; // 可以指定某一个服务器

return "\*"; // 指定所有

}

@Override

public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {

if (cause instanceof HystrixTimeoutException) {

return response(HttpStatus.GATEWAY\_TIMEOUT);

} else {

return response(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

private ClientHttpResponse response(final HttpStatus status) {

return new ClientHttpResponse() {

@Override

public HttpStatus getStatusCode() throws IOException {

return status;

// return HttpStatus.BAD\_REQUEST;

}

@Override

public int getRawStatusCode() throws IOException {

return status.value();

// return HttpStatus.BAD\_REQUEST.value();

}

@Override

public String getStatusText() throws IOException {

return status.getReasonPhrase();

//return HttpStatus.BAD\_REQUEST.name();

// return HttpStatus.BAD\_REQUEST.getReasonPhrase();

}

@Override

public void close() {

}

@Override

public InputStream getBody() throws IOException {

log.info("getBody()");

ObjectMapper objectMapper = new ObjectMapper();

Map<String, Object> map = new HashMap<>();

map.put("status", status.value());

map.put("message", "无法连接，请检查您的网络");

map.put("fallback", MyFallbackProvider.this.getRoute());

return new ByteArrayInputStream(

objectMapper.writeValueAsString(map).getBytes("UTF-8"));

}

@Override

public HttpHeaders getHeaders() {

HttpHeaders headers = new HttpHeaders();

headers.setContentType(MediaType.APPLICATION\_JSON);

return headers;

}

};

}

}

回看：<https://blog.csdn.net/DearLC/article/details/102916468>

#### 遗留

WebAdminFeignFallbackProvider.java类的研究

## Hystrix

案例：<https://blog.csdn.net/qq_41402200/article/details/90447408>

使用的相关说明：<http://roc.havemail.cn/archives/156.html>

熔断的简单解说：<https://www.cnblogs.com/qianjinyan/p/10816952.html>

降级处理代码示例：<https://blog.csdn.net/varyall/article/details/99722594>

### 是什么

在分布式系统中，每个服务都可能会调用很多其他服务，被调用的那些服务就是依赖服务，有的时候某些依赖服务出现故障也是很常见的。

Hystrix 可以让我们在分布式系统中对服务间的调用进行控制，加入一些调用延迟或者依赖故障的容错机制。Hystrix 通过将依赖服务进行资源隔离，进而阻止某个依赖服务出现故障时在整个系统所有的依赖服务调用中进行蔓延；同时Hystrix 还提供故障时的 fallback 降级机制。

总而言之，Hystrix 通过这些方法帮助我们提升分布式系统的可用性和稳定性。

Hystrix是熔断器

在微服务架构中多层服务之间会相互调用，如果其中有一层服务故障了，可能会导致一层服务或者多层服务故障，从而导致整个系统故障。这种现象被称为服务雪崩效应。Hystrix 组件就可以解决此类问题，Hystrix 负责监控服务之间的调用情况，连续多次失败的情况进行熔断保护。保护的方法就是使用 Fallback，当调用的服务出现故障时，就可以使用 Fallback 方法的返回值；Hystrix 间隔时间会再次检查故障的服务，如果故障服务恢复，将继续使用服务

Hystrix是Netflix开源的一款容错系统，能帮助使用者码出具备强大的容错能力和鲁棒性的程序。

Hystrix 供分布式系统使用，提供延迟和容错功能，隔离远程系统、访问和第三方程序库的访问点，防止级联失败，保证复杂的分布系统在面临不可避免的失败时，仍能有其弹性。

在分布式环境中，许多服务依赖项中的一些必然会失败。Hystrix是一个库，通过添加延迟容忍和容错逻辑，帮助你控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点、停止级联失败和提供回退选项来实现这一点，所有这些都可以提高系统的整体弹性。

### 有什么用

\* 对通过第三方客户端库访问的依赖项（通常是通过网络）的延迟和故障进行保护和控制。

\* 在复杂的分布式系统中阻止级联故障。

\* 快速失败，快速恢复。

\* 回退，尽可能优雅地降级。

\* 启用近实时监控、警报和操作控制。

### 解决了什么问题

复杂分布式体系结构中的应用程序有许多依赖项，每个依赖项在某些时候都不可避免地会失败。如果主机应用程序没有与这些外部故障隔离，那么它有可能被他们拖垮。

### 设计原则是什么

\* 对依赖服务调用时出现的调用延迟和调用失败进行控制和容错保护。

\* 在复杂的分布式系统中，阻止某一个依赖服务的故障在整个系统中蔓延。比如某一个服务故障了，导致其它服务也跟着故障。

\* 提供 fail-fast（快速失败）和快速恢复的支持。

\* 提供 fallback 优雅降级的支持。

\* 支持近实时的监控、报警以及运维操作。

举个例子：

有一个分布式系统，服务A依赖于服务B，服务B依赖于服务C/D/E。在这样一个成熟的系统内，比如说最多可能只有100个线程资源。正常情况下,40个线程并发调用服务C，各30个线程并发调用 D/E。

调用服务 C，只需要 20ms，现在因为服务C故障了，比如延迟，或者挂了，此时线程会吊住2s左右。40个线程全部被卡住，由于请求不断涌入，其它的线程也用来调用服务 C，同样也会被卡住。这样导致服务B的线程资源被耗尽，无法接收新的请求，甚至可能因为大量线程不断的运转，导致自己宕机。服务A也挂了。

Hystrix可以对其进行资源隔离，比如限制服务B只有40个线程调用服务C。当此40个线程被hang住时，其它60个线程依然能正常调用工作。从而确保整个系统不会被拖垮。

-------

\* 防止任何单个依赖项耗尽所有容器（如Tomcat）用户线程。

\* 甩掉包袱，快速失败而不是排队。

\* 在任何可行的地方提供回退，以保护用户不受失败的影响。

\* 使用隔离技术（如隔离板、泳道和断路器模式）来限制任何一个依赖项的影响。

\* 通过近实时的度量、监视和警报来优化发现时间。

\* 通过配置的低延迟传播来优化恢复时间。

\* 支持对Hystrix的大多数方面的动态属性更改，允许使用低延迟反馈循环进行实时操作修改。

\* 避免在整个依赖客户端执行中出现故障，而不仅仅是在网络流量中。

### 限流

在微服务架构中服务端的请求承载量是有限的，希望在客户端请求时，限制请求的并发数，防止请求数量过大对服务端产生压力。

可以通过semaphore.maxConcurrentRequests,coreSize,maxQueueSize和

queueSizeRejectionThreshold设置信号量模式下的最大并发量、线程池大小、缓冲区大小和缓冲区降级阈值。

例如：

#不设置缓冲区，当请求数超过coreSize时直接降级

hystrix.threadpool.userThreadPool.maxQueueSize=-1

#超时时间大于我们的timeout接口返回时间

hystrix.command.userCommandKey.execution.isolation.thread.timeoutInMilliseconds=

### 如何实现它的目标的

\* 用一个HystrixCommand 或者 HystrixObservableCommand （这是命令模式的一个例子）包装所有的对外部系统（或者依赖）的调用，典型地它们在一个单独的线程中执行

\* 调用超时时间比你自己定义的阈值要长。有一个默认值，对于大多数的依赖项你是可以自定义超时时间的。

\* 为每个依赖项维护一个小的线程池(或信号量)；如果线程池满了，那么该依赖性将会立即拒绝请求，而不是排队。

\* 调用的结果有这么几种：成功、失败（客户端抛出异常）、超时、拒绝。

\* 在一段时间内，如果服务的错误百分比超过了一个阈值，就会触发一个断路器来停止对特定服务的所有请求，无论是手动的还是自动的。

\* 当请求失败、被拒绝、超时或短路时，执行回退逻辑。

\* 近实时监控指标和配置变化。

## Feign

案例解说如何使用feign：<https://blog.csdn.net/wufewu/article/details/84679962>

### 是什么

feign是声明式的web service客户端，它让微服务之间的调用变得更简单了，类似controller调用service。Spring Cloud集成了Ribbon和Eureka，可在使用Feign时提供负载均衡的http客户端。

### 意义

使用feign之后，我们调用eureka 注册的其他服务，在代码中就像各个service之间相互调用那么简单。

### 原理：

* 启动时，程序会进行包扫描，扫描所有包下所有@FeignClient注解的类，并将这些类注入到spring的IOC容器中。当定义的Feign中的接口被调用时，通过JDK的动态代理来生成RequestTemplate。
* RequestTemplate中包含请求的所有信息，如请求参数，请求URL等。
* RequestTemplate声场Request，然后将Request交给client处理，这个client默认是JDK的HTTPUrlConnection，也可以是OKhttp、Apache的HTTPClient等。
* 最后client封装成LoadBaLanceClient，结合ribbon负载均衡地发起调用。

### 注解：

@EnableFeignClients

在启动类上使用，意思是启用Feign

@FeignClient

| **属性名** | **默认值** | **作用** | **备注** |
| --- | --- | --- | --- |
| value | 空字符串 | 调用服务名称，和name属性相同 |  |
| serviceId | 空字符串 | 服务id，作用和name属性相同 | 已过期 |
| name | 空字符串 | 调用服务名称，和value属性相同 |  |
| url | 空字符串 | 全路径地址或hostname，http或https可选 |  |
| decode404 | false | 配置响应状态码为404时是否应该抛出FeignExceptions |  |
| configuration | {} | 自定义当前feign client的一些配置 | 参考FeignClientsConfiguration |
| fallback | void.class | 熔断机制，调用失败时，走的一些回退方法，可以用来抛出异常或给出默认返回数据。 | 底层依赖hystrix，启动类要加上@EnableHystrix |
| path | 空字符串 | 自动给所有方法的requestMapping前加上前缀，类似与controller类上的requestMapping |  |
| primary | true |  |  |

### 依赖

<!-- 导入Feign依赖 -->

<dependency>

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

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

</dependency>

### 应用

项目：

hello-spring-cloud-web-admin-feign

AdminController.java 请求位置

AdminService.java 此接口注解

@FeignClient(value = "hello-spring-cloud-service-admin",

fallback = AdminServiceHystrix.class) 指向hello-spring-cloud-service-admin服务

## Ribbon

引用：<https://www.jianshu.com/p/1bd66db5dc46>

### 是什么

主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层服务连接在一起。

ribbon是对服务之间调用做负载，是服务之间的负载均衡，zuul是可以对外部请求做负载均衡。

Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。

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网关那样需要独立部署，但是它几乎存在于每一个Spring Cloud构建的微服务和基础设施中。

因为微服务间的调用，API网关的请求转发等内容，实际上都是通过Ribbon来实现的，包括后续我们将要介绍的Feign，它也是基于Ribbon实现的工具。

所以，对Spring Cloud Ribbon的理解和使用，对于我们使用Spring Cloud来构建微服务非常重要。

### 负载算法

默认使用RoundRobinRule（轮循查询）

更改算法方式：

@Bean

public RandomRule randomRule() {

return new RandomRule();

}



## Spring Boot Admin

这是用于Spring Boot应用程序的管理UI。

要监视应用程序，必须在此服务器上注册它们。这可以通过包括Spring Boot Admin Client或使用Spring Cloud Discovery Client实现来完成。

通过此链接可访问界面<http://localhost:8084>

## 备忘

添加 @EnableEurekaClient 注解表示此工程可以向注册中心提供服务

添加 @EnableFeignClients 注解表示开启 Feign 功能进行远程调用

## 应用监控

### Actuator端点监控的使用

依赖

　　<dependency>

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

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

　　</dependency>

配置actuator权限配置

（否则访问一些暴露的监控信息会报401，很多博客里没有这一项，让很多人误解了）

（要在配置文件中开放访问端口，否则会报404,endpoints的配置）

management:

Security:

Enabled: false

endpoints:

web:

exposure:

include: '\*'

简单介绍：

| **ID** | **描述** | **敏感（Sensitive）** |
| --- | --- | --- |
| autoconfig | 显示一个auto-configuration的报告，该报告展示所有auto-configuration候选者及它们被应用或未被应用的原因 | true |
| beans | 显示一个应用中所有Spring Beans的完整列表 | true |
| configprops | 显示一个所有@ConfigurationProperties的整理列表 | true |
| dump | 执行一个线程转储 | true |
| env | 暴露来自Spring　ConfigurableEnvironment的属性 | true |
| health | 展示应用的健康信息（当使用一个未认证连接访问时显示一个简单的’status’，使用认证连接访问则显示全部信息详情） | false |
| info | 显示任意的应用信息 | false |
| metrics | 展示当前应用的’指标’信息 | true |
| mappings | 显示一个所有@RequestMapping路径的整理列表 | true |
| shutdown | 允许应用以优雅的方式关闭（默认情况下不启用） | true |
| trace | 显示trace信息（默认为最新的一些HTTP请求） | true |

接下来实际操作一下吧（简单举个栗子）：

　　beans（显示一个应用中所有Spring Beans的完整列表）

　　启动springBoot应用后，浏览器访问localhost:8080/项目名/beans，浏览器就会显示Bean列表

　　health

　　首先需配置

　  　endpoints.health.sensitive=false

浏览器访问localhost:8080/项目名/health，浏览器就会显示健康信息

查看所有路由

<http://localhost:8769/actuator/routes>

### 应用：

查看所有http访问踪迹

<http://localhost:8764/actuator/httptrace>

## REST API

是前后端分离最佳实践，是开发的一套标准或者说是一套规范，不是框架。

一个uri代表一种资源，通过get、post、put、delete对资源进行操作

好处：  
1、轻量，直接通过http，不需要额外的协议，通常有post/get/put/deletec操作。  
2、面向资源，一目了然，具有自解释性  
3、数据描述简单，一般通过json或者xml做数据通讯

## 二级标题

### 三级标题

#### 四级标题

#### xxx

# 一级标题

## 二级标题

### 三级标题

#### 四级标题

#### xxx