

微服务架构：Spring-Cloud

什么是微服务？

微服务就是把原本臃肿的一个项目的所有模块拆分开来并做到互相没有关联，甚至可以不使用同一个数据库。比如：项目里面有User模块和Power模块，但是User模块和Power模块并没有直接关系，仅仅只是一些数据需要交互，那么就可以把这2个模块单独拆分开来，当user需要调用power的时候，power是一个服务方，但是power需要调用user的时候，user又是服务方了，所以，他们并不在乎谁是服务方谁是调用方，他们都是2个独立的服务，这时候，微服务的概念就出来了。

经典问题:微服务和分布式的区别

谈到区别，我们先简单说一下分布式是什么，所谓分布式，就是将偌大的系统划分为多个模块（这一点和微服务很像）部署到不同机器上（因为一台机器可能承受不了这么大的压力或者说一台非常好的服务器的成本可能够好几台普通的了），各个模块通过接口进行数据交互，其实分布式也是一种微服务。因为都是把模块拆分开来变为独立的单元，提供接口来调用，那么他们本质的区别在哪呢？他们的区别主要体现在“目标”上，何为目标，就是你这样架构项目要做到的事情。分布式的目标是什么？我们刚刚也看见了，就是一台机器承受不了的，或者是成本问题，不得不使用多台机器来完成服务的部署，而微服务的目标只是让各个模块拆分开来，不会被互相影响，比如模块的升级亦或是出现BUG等等...

讲了这么多，可以用一句话来理解：分布式也是微服务的一种，而微服务他可以在一台机器上。

微服务与Spring-Cloud的关系（区别）

微服务只是一种项目的架构方式，或者说是一种概念，就如同我们的MVC架构一样，那么Spring-Cloud便是对这种技术的实现。

微服务一定要使用Spring-Cloud吗？

我们刚刚说过，微服务只是一种项目的架构方式，如果你足够了解微服务是什么概念你就会知道，其实微服务就算不借助任何技术也能实现，只是有很多问题需要我们解决罢了例如：负载均衡，服务的注册与发现，服务调用，路由。。。等等等一系列问题，所以, Spring-Cloud 就出来了，Spring-Cloud将处理这些问题的技术全部打包好了，就类似那种开袋即食的感觉。。

Spring-Cloud项目的搭建

因为spring-cloud是基于spring-boot项目来的，所以我们项目得是一个spring-boot项目，至于spring-boot项目，这节我们先不讨论，这里要注意的一个点是spring-cloud的版本与spring-boot的版本要对应下图：

Table 1. Release train Spring Boot compatibility

Release Train	Boot Version
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

Table 2. Release train content

在我这里我的版本是这样的

spring-boot:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.2.RELEASE</version>
</parent>
```

spring-cloud:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Finchley.SR2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

当你项目里面有这些依赖之后，你的spring cloud项目已经搭建好了(初次下载spring-cloud可能需要一点时间)

Spring-Cloud组件：

eureka：

eureka是什么？

eureka是Netflix的子模块之一，也是一个核心的模块，eureka里有2个组件，一个是EurekaServer(一个独立的项目)这个用于定位服务以实现中间层服务器的负载平衡和故障转移，另一个便是EurekaClient（我们的微服务）它是用于与Server交互的，可以使得交互变得非常简单:只需要通过服务标识符即可拿到服务。

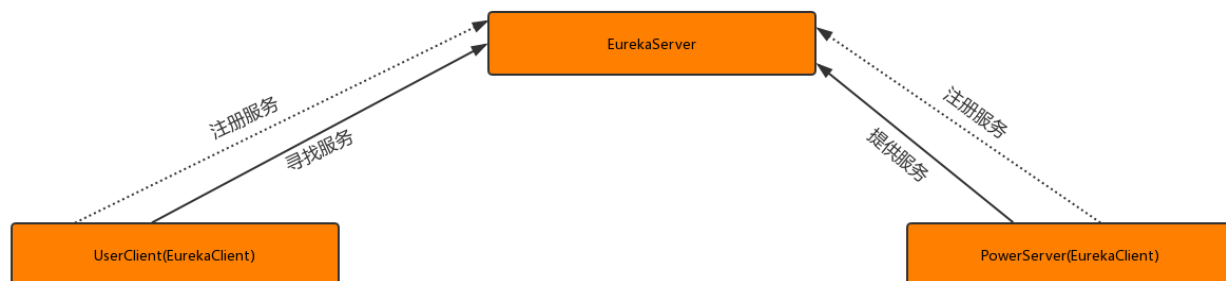
与spring-cloud的关系：

Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现(可以对比Zookeeper)。

Eureka 采用了 C-S 的设计架构。Eureka Server 作为服务注册功能的服务器，它是服务注册中心。

而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 并维持心跳连接。这样系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。SpringCloud 的一些其他模块（比如Zuul）就可以通过 Eureka Server 来发现系统中的其他微服务，并执行相关的逻辑。

角色关系图：



如何使用？

在spring-cloud项目里面加入依赖：

eureka客户端：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

eureka服务端：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

eureka服务端项目里面加入以下配置：

```
server:
  port: 3000
eureka:
  server:
    enable-self-preservation: false #关闭自我保护机制
    eviction-interval-timer-in-ms: 4000 #设置清理间隔（单位：毫秒 默认是60*1000）
  instance:
    hostname: localhost

client:
  registerWithEureka: false #不把自己作为一个客户端注册到自己身上
```

```
fetchRegistry: false #不需要从服务端获取注册信息（因为在这里自己就是服务端，而且已经禁用自己注册了）
serviceUrl:
  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka
```

当然，不是全部必要的，这里只是把我这里的配置copy过来了

然后在spring-boot启动项目上 加入注解:@EnableEurekaServer 就可以启动项目了

```
/**
 * 想要咨询vip课程相关的同学加一下木兰老师QQ: 2746251334
 * 想要往期视频的同学加一下安其拉老师QQ: 3164703201
 * author: 鲁班学院-商鞅老师
 */
@EnableEurekaServer
@SpringBootApplication
public class AppEureka {

    public static void main(String[] args) {
        SpringApplication.run(AppEureka.class);
    }
}
```

如果看见这个图片，那么说明你就搭建好了：

The screenshot shows the Spring Eureka web interface. The top navigation bar includes 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details.

Environment	test
Data center	default
- Current time:** 2019-01-06T20:47:43 +0800
- Uptime:** 00:00
- Lease expiration enabled:** true
- Renews threshold:** 1
- Renews (last min):** 0

A red warning message is displayed: "THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS."

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	485mb
environment	test

这个警告只是说你把他的自我保护机制关闭了

eureka客户端配置:

```
server:
  port: 6000
eureka:
```

```

client:
  serviceUrl:
    defaultZone: http://localhost:3000/eureka/ #eureka服务端提供的注册地址 参考服务端配
置的这个路径
  instance:

    instance-id: power-1 #此实例注册到eureka服务端的唯一的实例ID
    prefer-ip-address: true #是否显示IP地址
    leaseRenewalIntervalInSeconds: 10 #eureka客户需要多长时间发送心跳给eureka服务器，表明它仍
    然活着，默认为30 秒（与下面配置的单位都是秒）
    leaseExpirationDurationInSeconds: 30 #Eureka服务器在接收到实例的最后一次发出的心跳后，需要
    等待多久才可以将此实例删除，默认为90秒

spring:
  application:
    name: server-power #此实例注册到eureka服务端的name

```

然后在客户端的spring-boot启动项目上 加入注解:@EnableEurekaClient 就可以启动项目了 这里就不截图了我们直接来看效果图：

The screenshot shows the Spring Eureka web interface. The top navigation bar includes 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into several sections:

- System Status:** A table showing environment details (test, default) and system metrics (Current time, Uptime, Lease expiration enabled, Renewals threshold, Renewals (last min)).
- RENEWALS ARE LESSER THAN THE THRESHOLD. THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.** A red warning message.
- DS Replicas:** A section for distributed system replicas.
- Instances currently registered with Eureka:** A table showing the application 'SERVER-POWER' with 1 instance in the 'UP' state.
- General Info:** A table showing system information like total available memory (485mb) and environment (test).

这里我们能看见 名字叫server-power的（图中将其大写了） id为 power-1的服务 注册到我们的Eureka上面来了 至此，一个简单的eureka已经搭建好了。

eureka集群:

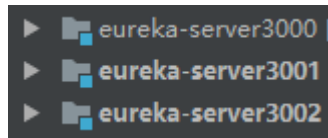
eureka集群原理

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

eureka集群配置

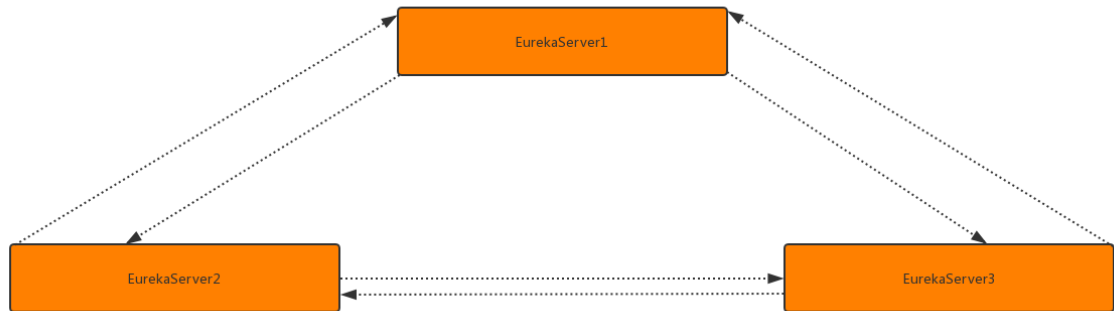
刚刚我们了解到 Eureka Server会将注册信息向其他Eureka Server进行同步 那么我们得声明有哪些server呢?

这里 假设我们有3个Eureka Server 如图:



现在怎么声明集群环境的server呢? 我们看一张图:

注:虚线代表注册关系



可能看着有点抽象, 我们来看看具体配置

```
server:
  port: 3000
eureka:
  server:
    enable-self-preservation: false
    eviction-interval-timer-in-ms: 4000
  instance:
    hostname: eureka3000.com

client:
  registerWithEureka: false
  fetchRegistry: false
  serviceUrl:
    defaultZone: http://eureka3001.com:3001/eureka,http://eureka3002.com:3002/eureka
```

这里 方便理解集群 我们做了一个域名的映射(条件不是特别支持我使用三台笔记本来测试。。。)至于域名怎么映射的话 这里简单提一下吧 修改你的hosts文件 (win10的目录在C:\Windows\System32\drivers\etc 其他系统的话自行百度一下把) 附上我的hosts文件:

```
127.0.0.1 eureka3000.com
127.0.0.1 eureka3001.com
127.0.0.1 eureka3002.com
```

我们回到主题，我们发现 集群配置与单体不同的点在于 原来是把服务注册到自己身上，而现在是注册到其它服务身上

至于为什么不注册自己了呢？，回到最上面我们说过，eureka的server会把自己的注册信息与其他的server同步，所以这里我们不需要注册到自己身上，因为另外两台服务器会配置本台服务器。(这里可能有点绕，可以参考一下刚刚那张集群环境的图，或者自己动手配置一下，另外两台eureka的配置与这个是差不多的，就不发出来了，只要注意是注册到其他的服务上面就好了)

当三台eureka配置好之后，全部启动一下就可以看见效果了：

The screenshot shows the Spring Eureka web interface. At the top, there's a navigation bar with 'HOME' and 'LAST 1000 SINCE STARTUP'. Below this, the 'System Status' section displays a table with system information:

System Status	
Environment	test
Data center	default
Current time	2019-01-06T21:59:21 +0800
Uptime	00:00
Lease expiration enabled	true
Renews threshold	1
Renews (last min)	0

Below the system status, a red warning message states: 'THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.'

The 'DS Replicas' section shows two replicas: 'eureka3002.com' and 'eureka3001.com'.

The 'Instances currently registered with Eureka' section shows a table with columns: Application, AMIs, Availability Zones, and Status. The table is currently empty, displaying 'No instances available'.

The 'General Info' section shows a table with columns: Name and Value. This section is also empty.

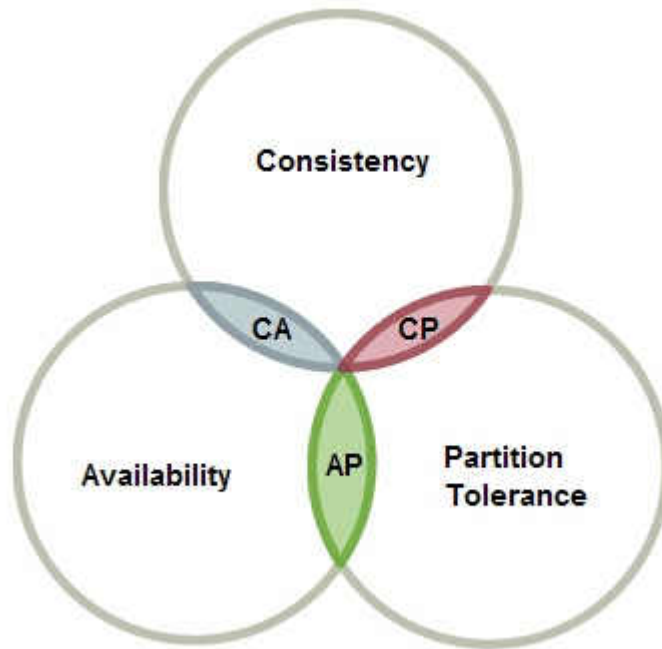
当然，我们这里仅仅是把服务端配置好了，那客户端怎么配置呢？话不多说，上代码：

```
client:
  serviceUrl:
    defaultZone:
      http://localhost:3000/eureka/,http://eureka3001.com:3001/eureka,http://eureka3002.com:3002/eureka
```

我们这里只截取了要改动的那一部分。就是 原来是注册到那一个地址上面，现在是要写三个eureka注册地址，但是不是代表他会注册三次，因为我们eureka server的注册信息是同步的，这里只需要注册一次就可以了，但是为什么要写三个地址呢。因为这样就可以做到高可用的配置：打个比方有3台服务器。但是突然宕机了一台，但是其他2台还健在，依然可以注册我们的服务，换句话来讲，只要有一台服务还建在，那么就可以注册服务，这里需要理解一下。

这里效果图就不发了，和之前单机的没什么两样，只是你服务随便注册到哪个eureka server上其他的eureka server上都有该服务的注册信息。

CAP定理的含义：



1998年，加州大学的计算机科学家 Eric Brewer 提出，分布式系统有三个指标。

Consistency --- 一致性
Availability --- 可用性
Partition tolerance --- 分区容错性

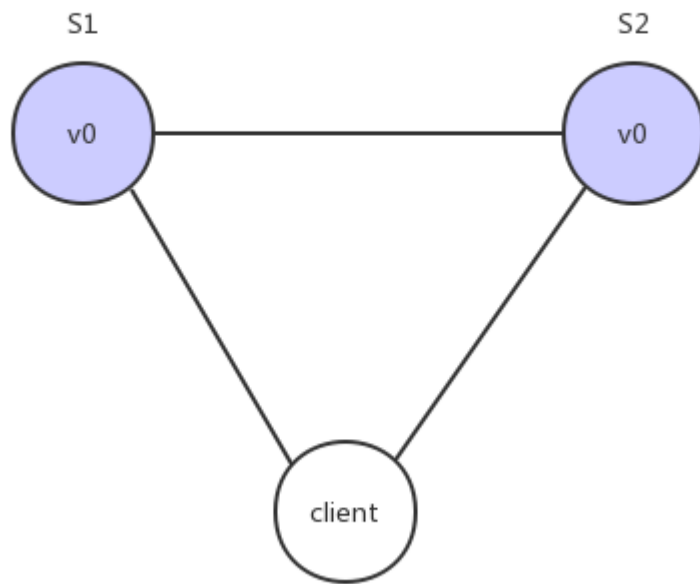
他们第一个字母分别是C,A,P

Eric Brewer 说，这三个指标不可能同时做到。这个结论就叫做 CAP 定理。

Partition tolerance

中文叫做"分区容错"。

大多数分布式系统都分布在多个子网络。每个子网络就叫做一个区 (partition)。分区容错的意思是，区间通信可能失败。比如，一台服务器放在本地，另一台服务器放在外地（可能是外省，甚至是外国），这就是两个区，它们之间可能无法通信。

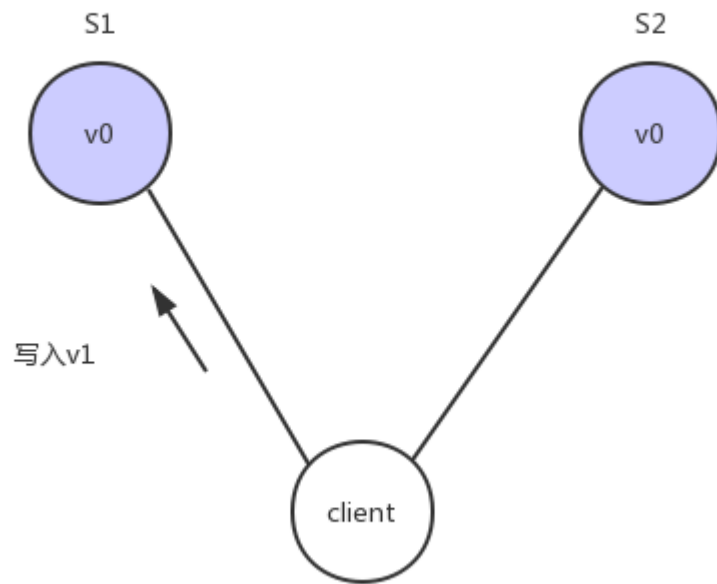


上图中，S1 和 S2 是两台跨区的服务器。S1 向 S2 发送一条消息，S2 可能无法收到。系统设计的时候，必须考虑到这种情况。

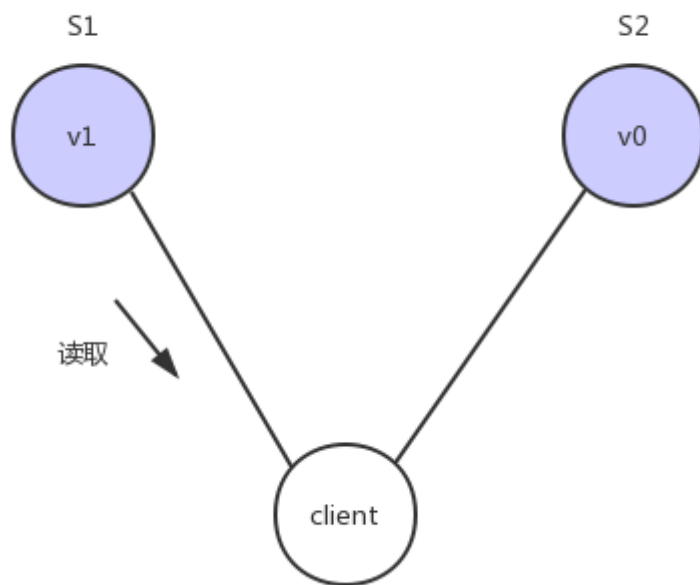
一般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。

Consistency

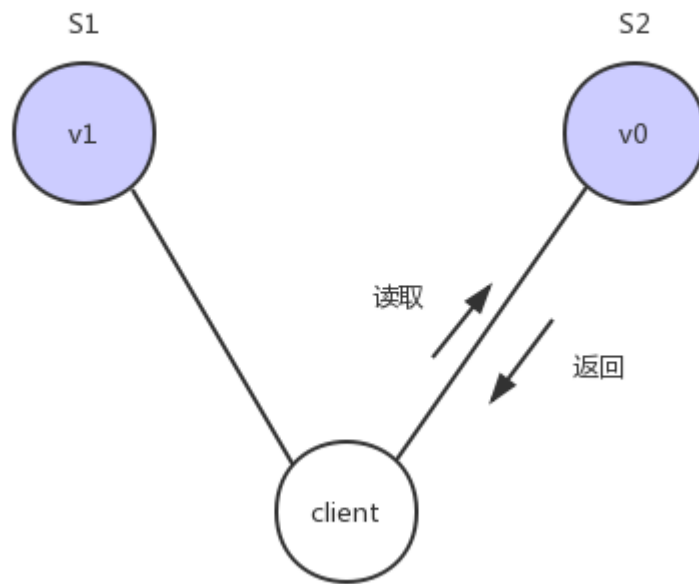
Consistency 中文叫做"一致性"。意思是，写操作之后的读操作，必须返回该值。举例来说，某条记录是 v0，用户向 S1 发起一个写操作，将其改为 v1。



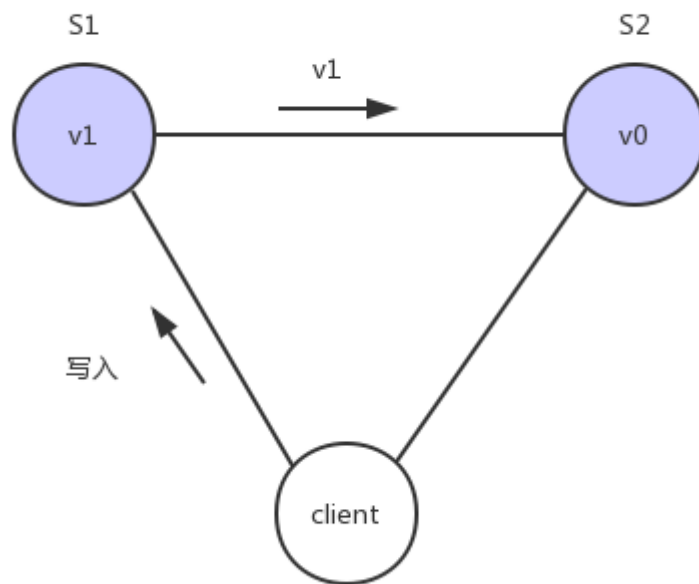
接下来用户读操作就会得到v1。这就叫一致性。



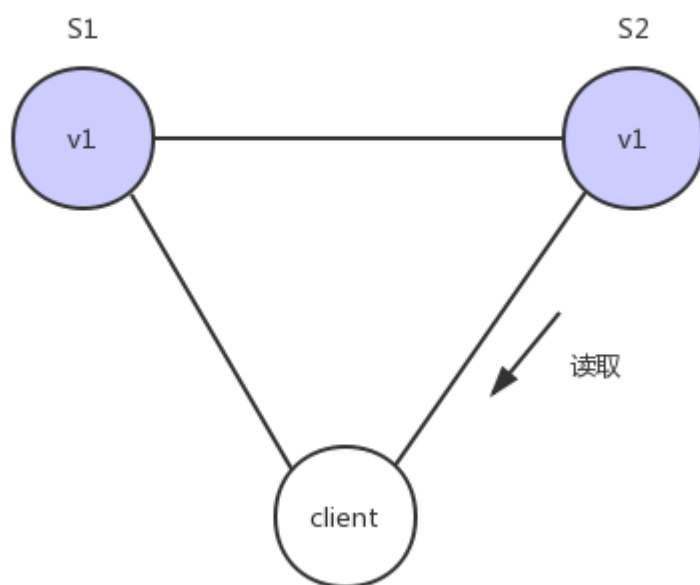
问题是，用户有可能会向S2发起读取操作，由于G2的值没有发生变化，因此返回的是v0，所以S1和S2的读操作不一致，这就不满足一致性了



为了让S2的返回值与S1一致，所以我们需要在往S1执行写操作的时候，让S1给S2也发送一条消息，要求S2也变成v1



这样子用户向S2发起读操作，就能得到v1



Availability

Availability 中文叫做"可用性", 意思是只要收到用户的请求, 服务器就必须给出回应。

用户可以选择向 S1 或 S2 发起读操作。不管是哪台服务器, 只要收到请求, 就必须告诉用户, 到底是 v0 还是 v1, 否则就不满足可用性。

Consistency 和 Availability 的矛盾

一致性和可用性, 为什么不可能同时成立? 答案很简单, 因为可能通信失败 (即出现分区容错)。

如果保证 S2 的一致性, 那么 S1 必须在写操作时, 锁定 S2 的读操作和写操作。只有数据同步后, 才能重新开放读写。锁定期间, S2 不能读写, 没有可用性不。

如果保证 S2 的可用性, 那么势必不能锁定 S2, 所以一致性不成立。

综上所述, S2 无法同时做到一致性和可用性。系统设计时只能选择一个目标。如果追求一致性, 那么无法保证所有节点的可用性; 如果追求所有节点的可用性, 那就没法做到一致性。

eureka对比Zookeeper:

Zookeeper在设计的时候遵循的是CP原则, 即一致性,Zookeeper会出现这样一种情况, 当master节点因为网络故障与其他节点失去联系时剩余节点会重新进行leader选举, 问题在于, 选举leader的时间太长: 30~120s, 且选举期间整个Zookeeper集群是不可用的, 这就导致在选举期间注册服务处于瘫痪状态, 在云部署的环境下, 因网络环境使Zookeeper集群失去master节点是较大概率发生的事情, 虽然服务能够最终恢复, 但是漫长的选举时间导致长期的服务注册不可用是不能容忍的。

Eureka在设计的时候遵循的是AP原则，即可用性。Eureka各个节点（服务）是平等的，没有主从之分，几个节点down掉不会影响正常工作，剩余的节点（服务）依然可以提供注册与查询服务，而Eureka的客户端在向某个Eureka注册或发现连接失败，则会自动切换到其他节点，也就是说，只要有一台Eureka还在，就能注册可用（保证可用性），只不过查询到的信息不是最新的（不保证强一致），除此之外，Eureka还有自我保护机制，如果在15分钟内超过85%节点都没有正常心跳，那么eureka就认为客户端与注册中心出现了网络故障，此时会出现一下情况：

- 1:Eureka 不再从注册列表中移除因为长时间没有收到心跳而过期的服务。
- 2: Eureka 仍然能够接收新服务的注册和查询请求，但是不会被同步到其它节点上（即保证当前节点可用）
- 3: 当网络稳定后，当前实例新的注册信息会被同步到其它节点中

ribbon:

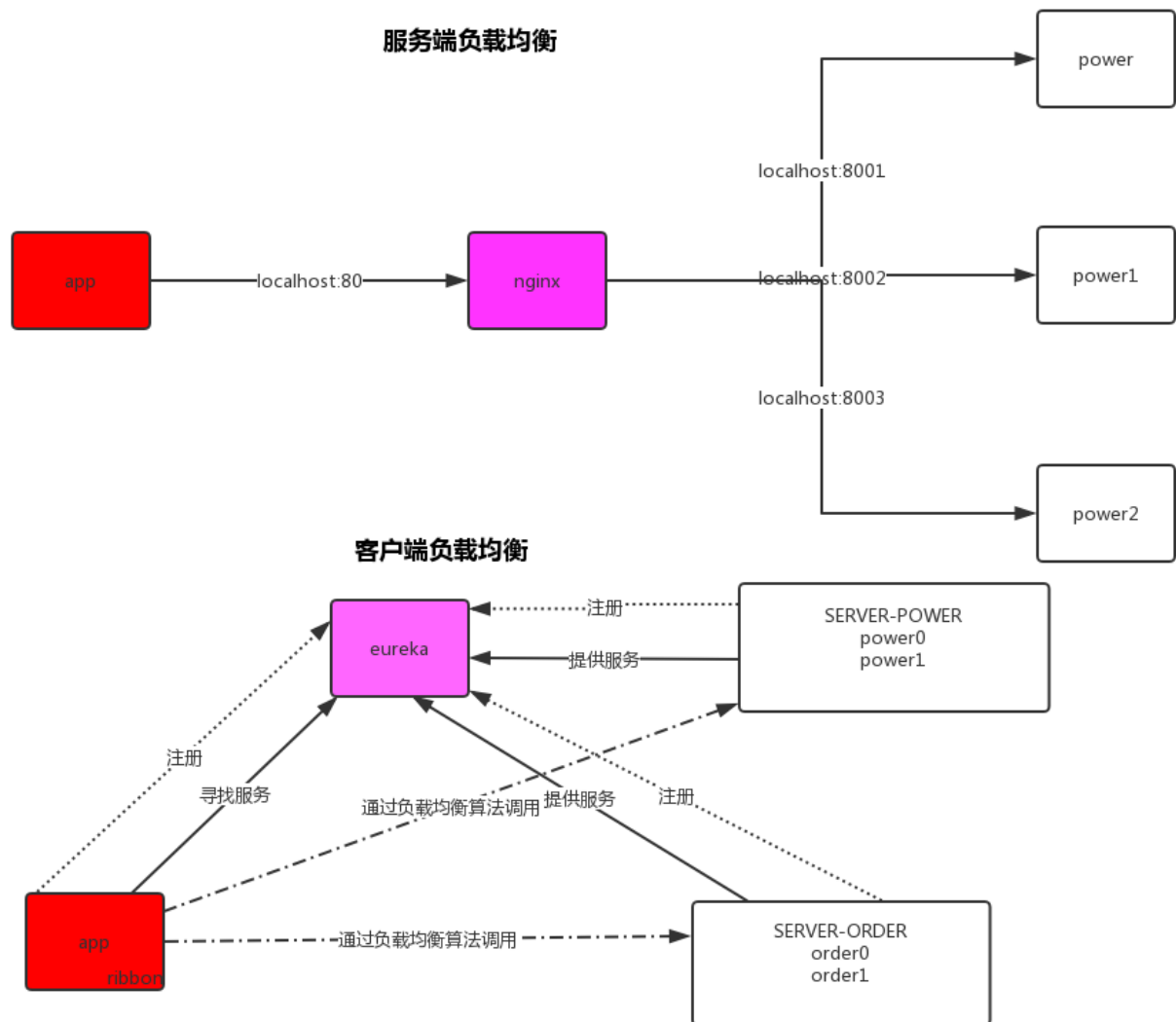
ribbon是什么？

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端 负载均衡的工具。

简单的说，Ribbon是Netflix发布的开源项目，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客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer（简称LB）后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。

客户端负载均衡？？ 服务端负载均衡??

我们用一张图来描述一下这两者的区别



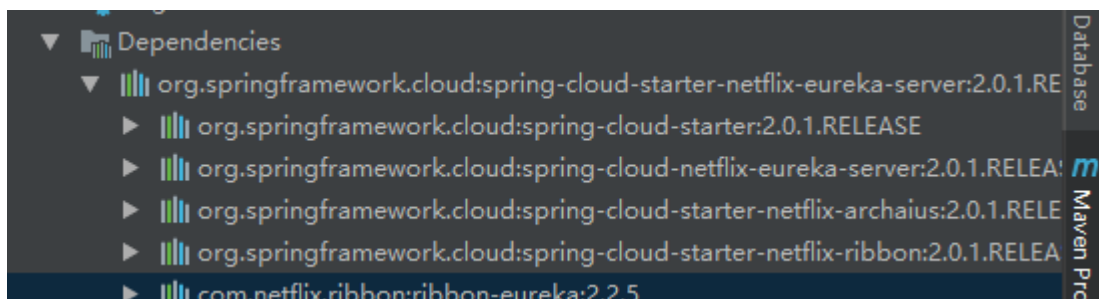
这篇文章里面不会去解释nginx，如果不知道是什么的话，可以先忽略，先看看这张图

服务端的负载均衡是一个url先经过一个代理服务器（这里是nginx），然后通过这个代理服务器通过算法（轮询，随机，权重等等..）反向代理你的服务，来完成负载均衡

而客户端的负载均衡则是一个请求在客户端的时候已经声明了要调用哪个服务，然后通过具体的负载均衡算法来完成负载均衡

如何使用:

首先，我们还是要引入依赖，但是，eureka已经把ribbon集成到他的依赖里面去了，所以这里不需要再引用ribbon的依赖，如图：

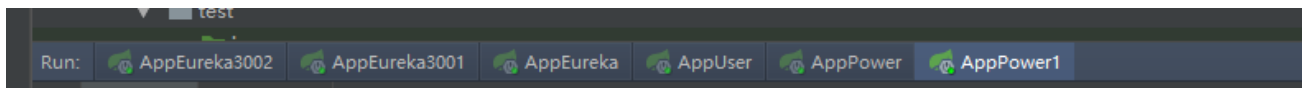


要使用ribbon，只需要一个注解：

```
@Bean
@LoadBalanced
public RestTemplate restTemplate(){
    RestTemplate restTemplate = new RestTemplate();
    return restTemplate;
}
```

在RestTemplate上面加入@LoadBalanced注解，这样子就已经有了负载均衡，怎么来证明？

我们这里现在启动了eureka集群（3个eureka）和Power集群（2个power）和一个服务调用者（User）



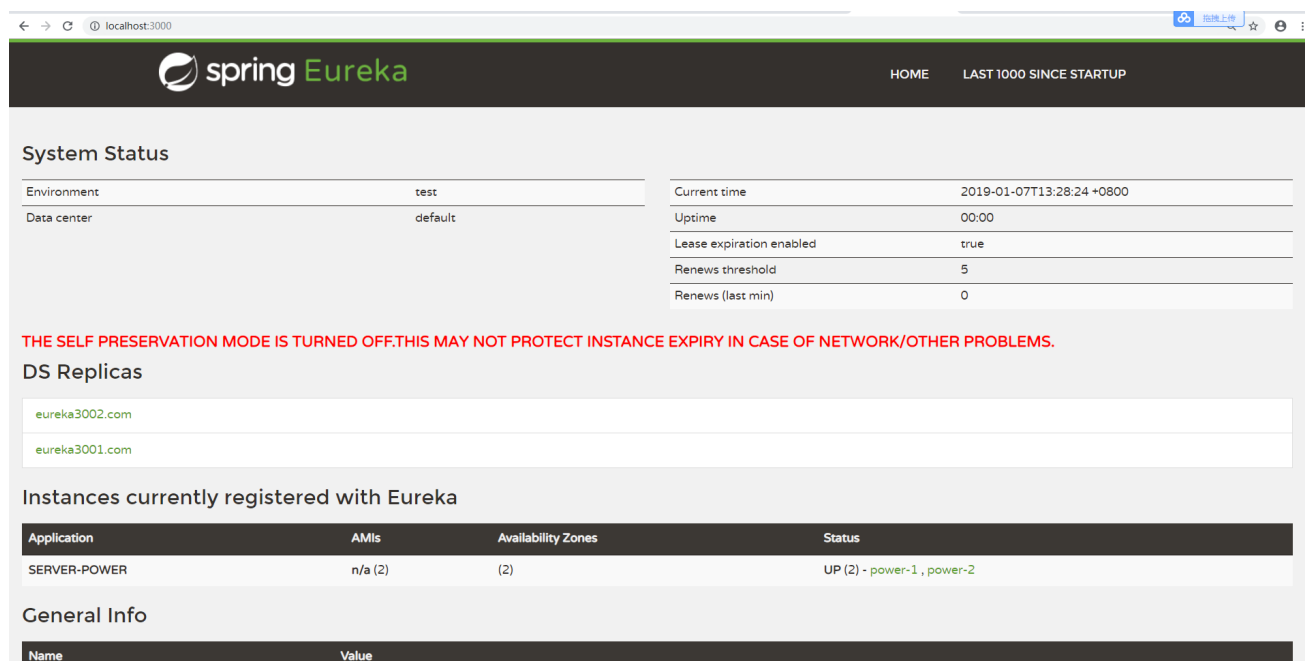
但是我们的User仅仅只需要调用服务，不需要注册服务信息，所以需要改一下配置文件：

配置什么意思就不做过多解释了，上面讲eureka的时候有讲到过

```
server:
  port: 5000
eureka:

  client:
    registerWithEureka: false
    serviceUrl:
      defaultZone:
http://localhost:3000/eureka/,http://eureka3001.com:3001/eureka,http://eureka3002.com:3002/eureka
```

然后启动起来的页面是这样子的



The screenshot shows the Spring Eureka web interface. The header includes the Spring Eureka logo and navigation links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into several sections:

- System Status:** A table showing environment (test), data center (default), current time (2019-01-07T13:28:24+0800), uptime (00:00), lease expiration enabled (true), renew threshold (5), and renewals (last min) (0).
- DS Replicas:** A list of replicas: eureka3002.com and eureka3001.com.
- Instances currently registered with Eureka:** A table showing application (SERVER-POWER), AMIs (n/a (2)), availability zones ((2)), and status (UP (2) - power-1, power-2).
- General Info:** A table with columns Name and Value.

A red warning message is displayed: "THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS."

我们能看见 微服务名:SERVER-POWER 下面有2个微服务 (power-1,power2) , 现在我们来通过微服务名调用这个服务

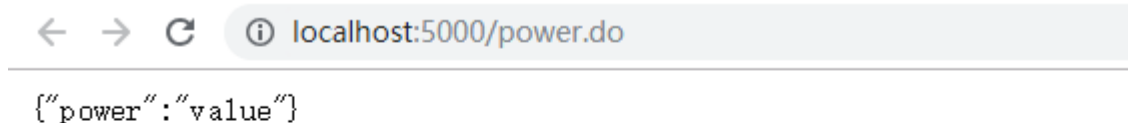
这是我们的user项目的调用代码：

```
private static final String URL="http://SERVER-POWER";

@Autowired
private RestTemplate restTemplate;

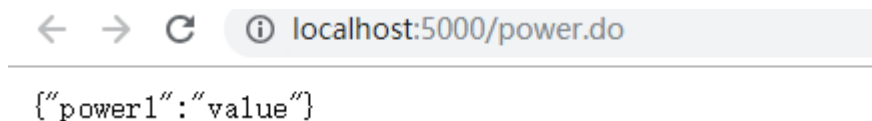
@RequestMapping("/power.do")
public Object power(){
    return restTemplate.getForObject(URL+"/power.do",Object.class);
}
```

我们来看看效果:



localhost:5000/power.do

{ "power": "value" }



localhost:5000/power.do

{ "power1": "value" }

这里可能有点抽象，需要你们自己去写才能体会到，但是我们已经完成了负载均衡，他默认的负载均衡是轮询策略，也就是一人一次，下一节我们来讲一下他还有哪些策略。

核心组件：IRule

IRule是什么? 它是Ribbon对于负载均衡策略实现的接口，怎么理解这句话？说白了就是你实现这个接口，就能自定义负载均衡策略，自定义我们待会儿来讲，我们先来看看他有哪些默认的实现



这里是ribbon负载均衡默认的实现，由于是笔记的关系，这里不好测试，只能你们自己去测试一下了，具体如何使用呢？

看代码：

```
@Bean
public IRule iRule(){
    return new RoundRobinRule();
}
```

在Spring 的配置类里面把对应的实现作为一个Bean返回出去就行了。

自定义负载均衡策略：

我们刚刚讲过，只要实现了IRule就可以完成自定义负载均衡，至于具体怎么来，我们先看看他默认的实现

```
/*
 *
 * Copyright 2013 Netflix, Inc.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
```

```

    * limitations under the License.
    *
    */
package com.netflix.loadbalancer;

import java.util.List;
import java.util.Random;

import com.netflix.client.config.IClientConfig;

/**
 * A loadbalancing strategy that randomly distributes traffic amongst existing
 * servers.
 *
 * @author stonse
 *
 */
public class RandomRule extends AbstractLoadBalancerRule {

    Random rand;

    public RandomRule() {
        rand = new Random();
    }

    /**
     * Randomly choose from all living servers
     */
    @edu.umd.cs.findbugs.annotations.SuppressWarnings(value =
"RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE")
    public Server choose(ILoadBalancer lb, Object key) {
        if (lb == null) {
            return null;
        }
        Server server = null;

        while (server == null) {

            if (Thread.interrupted()) {
                return null;
            }

            List<Server> upList = lb.getReachableServers();

            List<Server> allList = lb.getAllServers();

            int serverCount = allList.size();
            if (serverCount == 0) {
                /*
                 * No servers. End regardless of pass, because subsequent passes
                 * only get more restrictive.
                 */
                return null;
            }

```

```

    }

    int index = rand.nextInt(serverCount);
    server = upList.get(index);

    if (server == null) {
        /*
         * The only time this should happen is if the server list were
         * somehow trimmed. This is a transient condition. Retry after
         * yielding.
         */
        Thread.yield();
        continue;
    }

    if (server.isAlive()) {
        return (server);
    }

    // Shouldn't actually happen.. but must be transient or a bug.
    server = null;
    Thread.yield();
}

return server;

}

@Override
public Server choose(Object key) {
    return choose(getLoadBalancer(), key);
}

@Override
public void initWithNiwsConfig(IClientConfig clientConfig) {
    // TODO Auto-generated method stub
}
}

```

我们来看看这个类AbstractLoadBalancerRule

```

public abstract class AbstractLoadBalancerRule implements IRule, IClientConfigAware {

    private ILoadBalancer lb;

    @Override
    public void setLoadBalancer(ILoadBalancer lb){
        this.lb = lb;
    }

    @Override
    public ILoadBalancer getLoadBalancer(){

```

```

        return lb;
    }
}

```

这里我们能发现，还是我们上面所说的 实现了IRule就能够自定义负载均衡即使是他默认的策略也实现了IRule
我们可以直接把代码copy过来改动一点:

```

package com.luban.rule;

import com.netflix.client.config.IClientConfig;
import com.netflix.loadbalancer.AbstractLoadBalancerRule;
import com.netflix.loadbalancer.ILoadBalancer;
import com.netflix.loadbalancer.Server;

import java.util.List;
import java.util.Random;

/**
 * 想要咨询vip课程相关的同学加一下木兰老师QQ: 2746251334
 * 想要往期视频的同学加一下安其拉老师QQ: 3164703201
 * author: 鲁班学院-商鞅老师
 */
public class LubanRule extends AbstractLoadBalancerRule {

    //原来是纯随机策略 我们现在改为。 如果一个下标已经被随机到了2次了，第三次还是同样的下标的话，那就再
    //随机一次
    Random rand;

    private int lastIndex = -1;
    private int nowIndex = -1;
    private int skipIndex = -1;

    public LubanRule() {
        rand = new Random();
    }

    /**
     * Randomly choose from all living servers
     */
    public Server choose(ILoadBalancer lb, Object key) {
        if (lb == null) {
            return null;
        }
        Server server = null;

        while (server == null) {
            if (Thread.interrupted()) {
                return null;
            }

```

```

List<Server> upList = lb.getReachableServers();
List<Server> allList = lb.getAllServers();

int serverCount = allList.size();
if (serverCount == 0) {
    /*
     * No servers. End regardless of pass, because subsequent passes
     * only get more restrictive.
     */
    return null;
}

int index = rand.nextInt(serverCount);
System.out.println("当前下标为:"+index);
if (skipIndex>=0&&index == skipIndex) {
    System.out.println("跳过");
    index = rand.nextInt(serverCount);
    System.out.println("跳过后下标:"+index);
}
skipIndex=-1;

nowIndex = index;
if (nowIndex == lastIndex) {
    System.out.println("下一次需要跳过的下标"+nowIndex);
    skipIndex = nowIndex;
}
lastIndex = nowIndex;
server = upList.get(index);

if (server == null) {
    /*
     * The only time this should happen is if the server list were
     * somehow trimmed. This is a transient condition. Retry after
     * yielding.
     */
    Thread.yield();
    continue;
}

if (server.isAlive()) {
    return (server);
}

// Shouldn't actually happen.. but must be transient or a bug.
server = null;
Thread.yield();
}

return server;
}

```

```

@Override
public Server choose(Object key) {
    return choose(getLoadBalancer(), key);
}

@Override
public void initWithNiwsConfig(IClientConfig clientConfig) {
    // TODO Auto-generated method stub
}

}

```

这里我们就把自己写的Rule给new出来交给spring 就好了

```

@Bean
public IRule iRule(){
    return new LubanRule();
}

```

具体测试的话就不测试了，那个效果放在笔记上不太明显，可以自己把代码copy过去测试一下

feign负载均衡：

feign是什么：

Feign是一个声明式WebService客户端。使用Feign能让编写Web Service客户端更加简单, 它的使用方法是定义一个接口，然后在上面添加注解，同时也支持JAX-RS标准的注解。Feign也支持可拔插式的编码器和解码器。Spring Cloud对Feign进行了封装，使其支持了Spring MVC标准注解和HttpMessageConverters。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

feign 能干什么：

Feign旨在使编写Java Http客户端变得更容易。前面在使用Ribbon+RestTemplate时，利用RestTemplate对http请求的封装处理，形成了一套模版化的调用方法。但是在实际开发中，由于对服务依赖的调用可能不止一处，往往一个接口会被多处调用，所以通常都会针对每个微服务自行封装一些客户端类来包装这些依赖服务的调用。所以，Feign在此基础上做了进一步封装，由他来帮助我们定义和实现依赖服务接口的定义。在Feign的实现下，我们只需创建一个接口并使用注解的方式来配置它(以前是Dao接口上面标注Mapper注解,现在是一个微服务接口上面标注一个Feign注解即可)，即可完成对服务提供方的接口绑定，简化了使用Spring cloud Ribbon时，自动封装服务调用客户端的开发量。

如何使用？

在客户端(User)引入依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

在启动类上面加上注解:@EnableFeignClients

然后编写一个service类加上@FeignClient()注解 参数就是你的微服务名字

```
@FeignClient("SERVER-POWER")
public interface PowerServiceClient {

    @RequestMapping("/power.do")
    public Object power();

}
```

下面是调用代码:

```
package com.luban.controller;

import com.luban.service.OrderServiceClient;
import com.luban.service.PowerServiceClient;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

/**
 * 想要咨询vip课程相关的同学加一下木兰老师QQ: 2746251334
 * 想要往期视频的同学加一下安其拉老师QQ: 3164703201
 * author: 鲁班学院-商鞅老师
 */
@RestController
public class UserController {

    private static final String URL="http://SERVER-POWER";

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    PowerServiceClient powerServiceClient;

    @RequestMapping("/power.do")
    public Object power(){
        return restTemplate.getForObject(URL+"/power.do",Object.class);
    }

    @RequestMapping("/feignPower.do")
    public Object feignPower(){
```

```
        return powerServiceClient.power();  
    }  
}
```

这里拿了RestTemplate做对比 可以看看2者区别

Feign集成了Ribbon

利用Ribbon维护了服务列表信息，并且融合了Ribbon的负载均衡配置，也就是说之前自定义的负载均衡也有效，这里需要你们自己跑一遍理解一下。而与Ribbon不同的是，通过feign只需要定义服务绑定接口且以声明式的方法，优雅而简单的实现了服务调用

hystrix断路器:

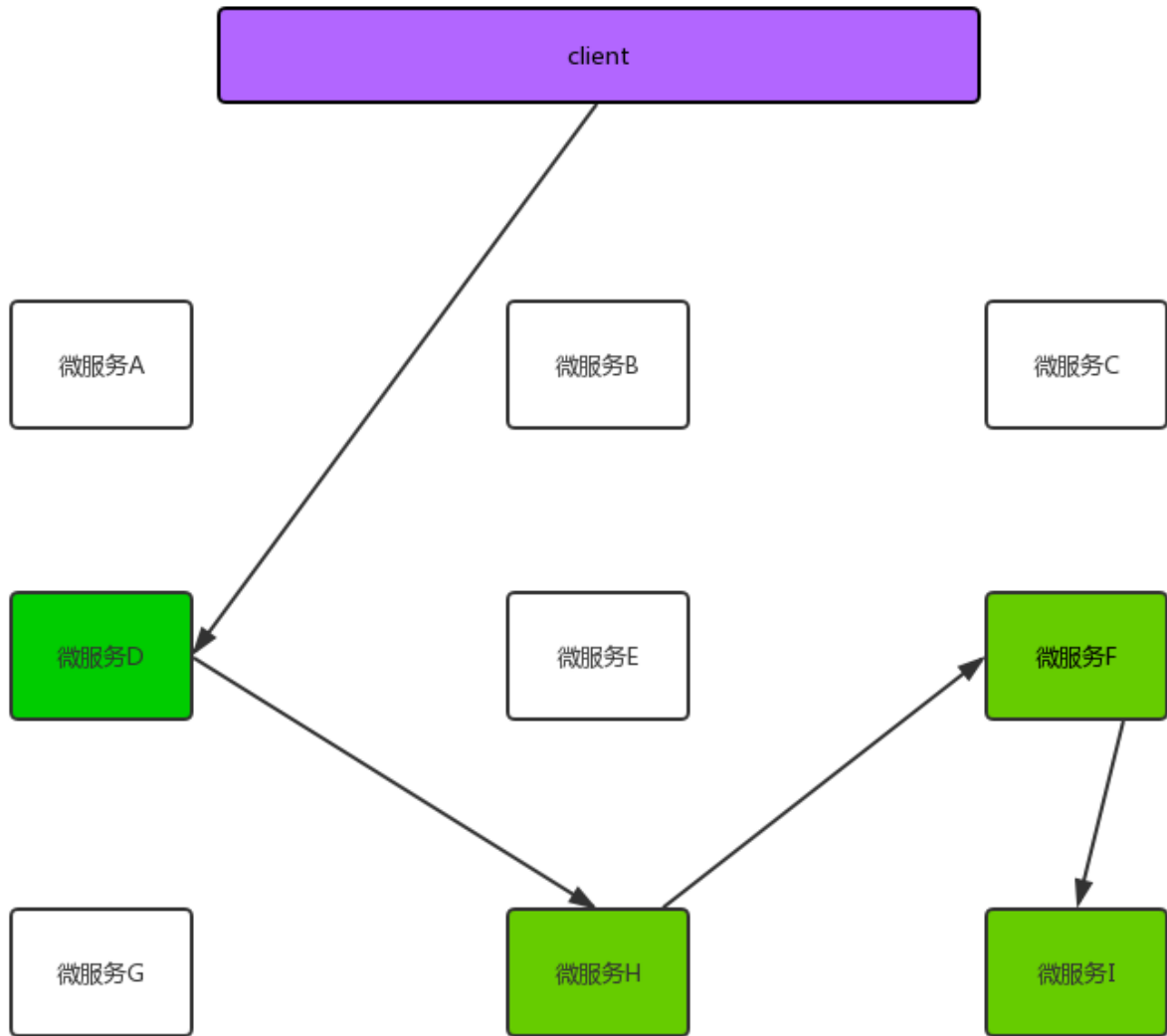
hystrix是什么?

Hystrix是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

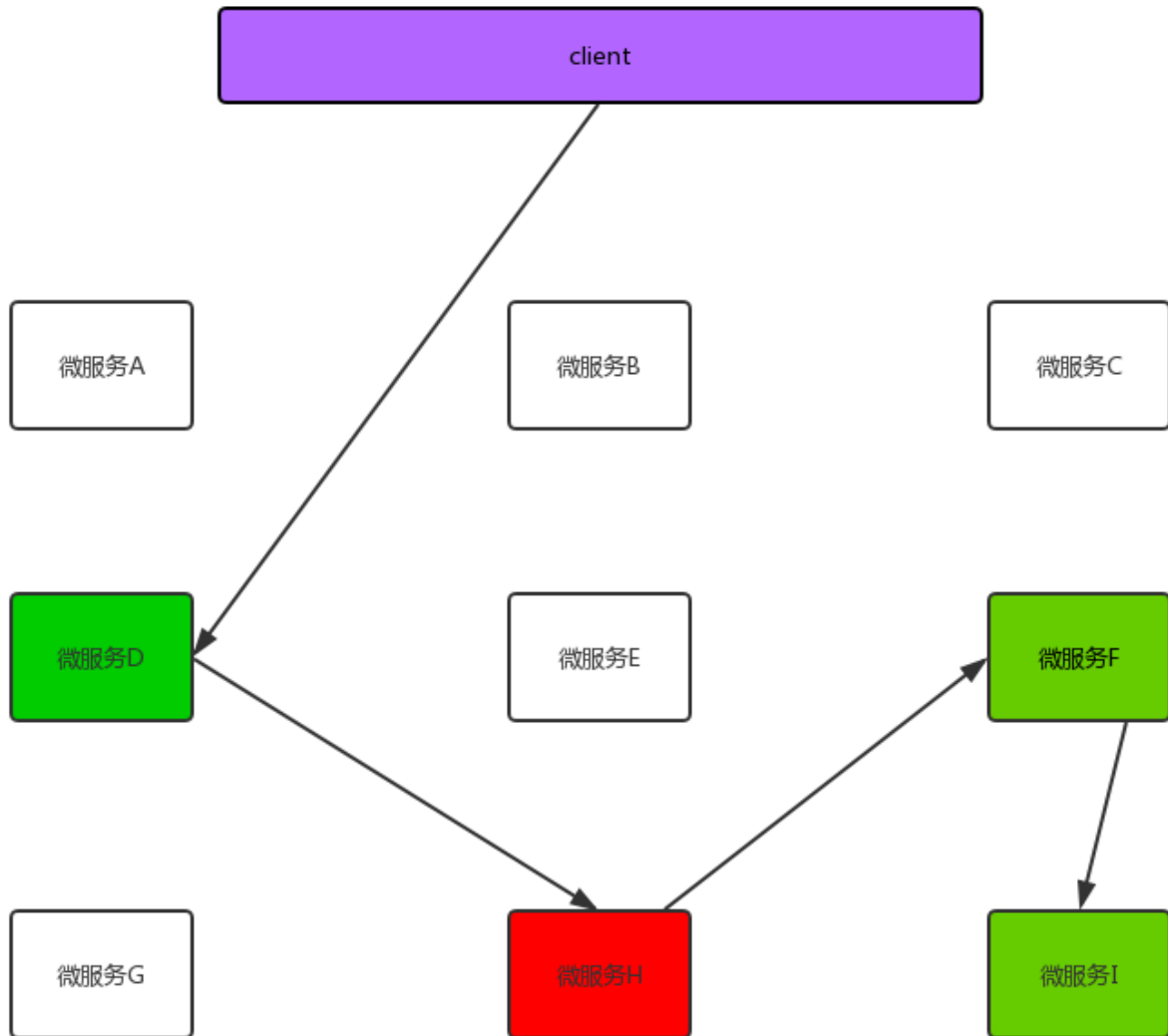
“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

大型项目中会出现的一些问题:

典型的一个案例就是服务血崩效应 我们来看一张图:



上图是一条微服务调用链，正常的情况我们就不必在讨论了，我们来说一下非正常情况，假设现在 微服务H 响应时间过长,或者微服务H直接down机了如图:



来看下上图，我们联想一下上图，如果发生这种情况，也就是说所有发给微服务D的请求 都会被卡在微服务H那，就会导致线程一直累计在这里，那么其他的微服务（比如A，B，C...）就没有可用线程了，导致整个服务器崩溃，这就是服务血崩。

导致服务雪崩的情况我们来总结一下，再看看怎么解决：

程序BUG，数据不匹配，响应时间过长，服务不可用等等.....

针对上面的问题，我们来看看有哪些解决方案：

服务限流

超时监控

服务熔断

服务降级

降级，超时：

我们先来解释一下降级,降级是当我们的某个微服务响应时间过长, 或者不可用了, 讲白了也就是那个微服务调用不了了, 我们不能吧错误信息返回出来, 或者让他一直卡在那里, 所以要在准备一个对应的策略(一个方法) 当发生这种问题的时候我们直接调用这个方法快速返回这个请求, 不让他一直卡在那。

讲了这么多,我们来看看具体怎么操作:

我们刚刚说了某个微服务调用不了了要做降级, 也就是说, 要在调用方做降级(不然那个微服务都down掉了再做降级也没什么意义了) 比如说我们 user 调用power 那么就在user 做降级

先把hystrix的依赖加入:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

启动类加入注解@EnableHystrix 或者@EnableCircuitBreaker(他们之间是一个继承关系, 2个注解所描述的内容是完全一样的, 可能看大家之前都是EnableXXX (比如eureka) 这里专门再写一个EnableHystrix方便大家记吧)

然后在我们的controller上面加入注解@HystrixCommand(fallbackMethod就是我们刚刚说的方法的名字)

```
@RequestMapping("/feignPower.do")
@HystrixCommand(fallbackMethod = "fallbackMethod")
public Object feignPower(String name){
    return powerServiceClient.power();
}
```

fallbackMethod:

这个R不重要, 你看做一个Map就好了, 是我封装的一个返回值的类。

```
public Object fallbackMethod(String name){
    System.out.println(name);
    return R.error("降级信息");
}
```

这里的这个降级信息具体内容得根据业务需求来, 比如说返回一个默认的查询信息, 亦或是系统维护(因为有可能要暂时关闭某个微服务而吧资源让给其他服务) 等等...

我们在power代码里面模拟一个异常

```
@RequestMapping("/power.do")
public Object power(String name) throws Exception{
    Map<String,Object> map = new HashMap<String, Object>();
    map.put("power1","value");
    if (name==null)
        throw new Exception();

    return map;
}
```

然后启动服务调用一下看看结果:

← → ↻ ⓘ localhost:5000/power.do

{"msg":"降级信息","code":"500"}

我们来测试一下超时:

我们改动一下power的代码 让他故意等待一回儿 (模拟响应超时)

```
@RequestMapping("/power.do")
public Object power(String name) throws Exception{
    Map<String,Object> map = new HashMap<String, Object>();
    map.put("power1","value");
    Thread.sleep(2000);

    return map;
}
```

这里我就把返回值内容改一下方便大家看:

← → ↻ ⓘ localhost:5000/power.do

{"msg":"超时降级信息","code":"500"}

可能有些同学有疑问, 我这里什么都没干, 就让他休眠了一下, 怎么就知道我这里超时了呢?

因为hystrix他有默认的超时监听, 当你这个请求默认超过了1秒钟就会超时 当然, 这个可以配置的, 至于怎么配置, 待会儿我会把一些配置统一列出来

讲了这么多, 这个降级到底有什么用呢?

第一, 他可以监听你的请求有没有超时, 第二, 报错了他这里直接截断了没有让请求一直卡在这里

其实降级还有一个好处, 就是当你的系统马上迎来大量的并发(双十一秒杀这种 或者促销活动) 这时候如果发现系统马上承载不了这么大的并发时, 可以考虑先关闭一些不重要的微服务(在降级方法里面返回一个比较友好的信息), 吧资源让给主微服务,总结一下就是

整体资源快不够了, 忍痛将某些服务先关掉, 待渡过难关, 再开启回来。

熔断,限流:

讲完降级,我们来讲讲熔断,其实熔断,就好像我们生活中的跳闸一样,比如说你的电路出故障了,为了防止出现大型事故 这里直接切断了你的电源以免意外继续发生,把这个概念放在我们程序上也是如此, 当一个微服务调用多次出现问题时(默认是10秒内20次当然 这个也能配置), hystrix就会采取熔断机制,不再继续调用你的方法(会在默认5秒钟内和电器短路一样, 5秒钟后会试探性的先关闭熔断机制,但是如果这时候再失败一次{之前是20次}那么又会重新进行熔断) 而是直接调用降级方法,这样就一定程度上避免了服务雪崩的问题

这个东西光笔记不太好测试,只能你们自己去测试了

限流

限流, 顾名思义, 就是限制你某个微服务的使用量(可用线程)

hystrix通过线程池的方式来管理你的微服务调用,他默认是一个线程池(10大小) 管理你的所有微服务,你可以给某个微服务开辟新的线程池:

```
@RequestMapping("/feignOrder.do")
@HystrixCommand(fallbackMethod = "fallbackOrderMethod" ,
    threadPoolKey = "order",
    threadPoolProperties = {@HystrixProperty(name = "coreSize",value = "2")
        ,@HystrixProperty(name = "maxQueueSize",value = "1"}})
public Object feignOrder(String name){
    System.out.println(1);
    return restTemplate.getForObject(ORDERURL+"/order.do",Object.class);
}
```

threadPoolKey 就是在线程池唯一标识, hystrix 会拿你这个标识去计数,看线程占用是否超过了,超过了就会直接降级该次调用

比如, 这里coreSize给他值为2 那么假设你这个方法调用时间是3s执行完, 那么在3s内如果有超过2个请求进来的话, 剩下的请求则全部降级

这一章节概念比较多, 例子比较少, 需要你们自己多去测试一下

feign整合hystrix:

feign 默认是支持hystrix的, 但是在Spring - cloud Dalston 版本之后就默认关闭了, 因为不一定业务需求要用的到,

所以现在要使用首先得打开他, 在yaml文件加上如下配置:

```
feign:
  hystrix:
    enabled: true
```

加上配置之后降级方法怎么写呢？

```
@FeignClient(value = "SERVER-POWER", fallback = PowerServiceFallback.class)
public interface PowerServiceClient {

    @RequestMapping("/power.do")
    public Object power(@RequestParam("name") String name);

}
```

在feign客户端的注解上 有个属性叫fallback 然后指向一个类

PowerServiceFallback 类：

```
@Component
public class PowerServiceFallback implements PowerServiceClient {
    @Override
    public Object power(String name) {
        return R.error("测试降级");
    }
}
```

这样子，方法降级就写好了

当然 可能你有这种需求，需要拿到具体的错误信息，那么可以这样写：

```
@Component
public class PowerServiceClientFallbackFactory implements
FallbackFactory<PowerServiceClient> {
    @Override
    public PowerServiceClient create(Throwable throwable) {
        return new PowerServiceClient() {
            @Override
            public Object power(String name) {
                String message = throwable.getMessage();
                return R.error("feign降级");
            }
        };
    }
}
```

客户端指定一个fallbackFactory就好了

```

@FeignClient(value = "SERVER-POWER", fallbackFactory =
PowerServiceClientFallbackFactory.class)
public interface PowerServiceClient {

    @RequestMapping("/power.do")
    public Object power(@RequestParam("name") String name);

}

```

这个message 就是拿到的错误信息

至此，就完成了feign与hystrix的整合

hystrix相关配置:

Execution相关的属性的配置

`hystrix.command.default.execution.isolation.strategy` 隔离策略，默认是Thread，可选Thread | Semaphore

`hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds` 命令执行超时时间，默认1000ms

`hystrix.command.default.execution.timeout.enabled` 执行是否启用超时，默认启用true

`hystrix.command.default.execution.isolation.thread.interruptOnTimeout` 发生超时是否中断，默认true

`hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests` 最大并发请求数，默认10，该参数当使用ExecutionIsolationStrategy.SEMAPHORE策略时才有效。如果达到最大并发请求数，请求会被拒绝。理论上选择semaphore size的原则和选择thread size一致，但选用semaphore时每次执行的单元要比较小且执行速度快（ms级别），否则的话应该用thread。semaphore应该占整个容器（tomcat）的线程池的一小部分。Fallback相关的属性 这些参数可以应用于Hystrix的THREAD和SEMAPHORE策略

`hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests` 如果并发数达到该设置值，请求会被拒绝和抛出异常并且fallback不会被调用。默认10

`hystrix.command.default.fallback.enabled` 当执行失败或者请求被拒绝，是否会尝试调用

`hystrixCommand.getFallback()` 。默认true

Circuit Breaker相关的属性

`hystrix.command.default.circuitBreaker.enabled` 用来跟踪circuit的健康性，如果未达标则让request短路。默认true

`hystrix.command.default.circuitBreaker.requestVolumeThreshold` 一个rolling window内最小的请求数。如果设为20，那么当一个rolling window的时间内（比如说1个rolling window是10秒）收到19个请求，即使19个请求都失败，也不会触发circuit break。默认20

`hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds` 触发短路的时间值，当该值设为5000时，则当触发circuit break后的5000毫秒内都会拒绝request，也就是5000毫秒后才会关闭circuit。默认5000

`hystrix.command.default.circuitBreaker.errorThresholdPercentage` 错误比率阈值，如果错误率 \geq 该值，circuit会被打开，并短路所有请求触发fallback。默认50

`hystrix.command.default.circuitBreaker.forceOpen` 强制打开熔断器，如果打开这个开关，那么拒绝所有 request，默认false

`hystrix.command.default.circuitBreaker.forceClosed` 强制关闭熔断器 如果这个开关打开，circuit将一直关闭且忽略`circuitBreaker.errorThresholdPercentage`

Metrics相关参数

`hystrix.command.default.metrics.rollingStats.timeInMilliseconds` 设置统计的时间窗口值的，毫秒值，circuit break 的打开会根据1个rolling window的统计来计算。若rolling window被设为10000毫秒，则rolling window会被分成n个buckets，每个bucket包含success, failure, timeout, rejection的次数 的统计信息。默认10000

`hystrix.command.default.metrics.rollingStats.numBuckets` 设置一个rolling window被划分的数量，若numBuckets=10，rolling window=10000，那么一个bucket的时间即1秒。必须符合rolling window % numberBuckets == 0。默认10

`hystrix.command.default.metrics.rollingPercentile.enabled` 执行时是否enable指标的计算和跟踪，默认true

`hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds` 设置rolling percentile window的时间，默认60000

`hystrix.command.default.metrics.rollingPercentile.numBuckets` 设置rolling percentile window的numberBuckets。逻辑同上。默认6

`hystrix.command.default.metrics.rollingPercentile.bucketSize` 如果bucket size=100，window=10s，若这10s里有500次执行，只有最后100次执行会被统计到bucket里去。增加该值会增加内存开销以及排序 的开销。默认100

`hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds` 记录health 快照（用来统计成功和错误绿）的间隔，默认500ms

Request Context 相关参数

`hystrix.command.default.requestCache.enabled` 默认true，需要重载`getCacheKey()`，返回null时不缓存

`hystrix.command.default.requestLog.enabled` 记录日志到HystrixRequestLog，默认true

Collapser Properties 相关参数

`hystrix.collapse.default.maxRequestsInBatch` 单次批处理的最大请求数，达到该数量触发批处理，默认Integer.MAX_VALUE

`hystrix.collapse.default.timerDelayInMilliseconds` 触发批处理的延迟，也可以为创建批处理的时间+该值，默认10

`hystrix.collapse.default.requestCache.enabled` 是否对HystrixCollapse.execute() and HystrixCollapse.queue()的cache, 默认true

ThreadPool 相关参数

线程数默认值10适用于大部分情况 (有时可以设置得更小), 如果需要设置得更大, 那有个基本得公式可以 follow: requests per second at peak when healthy \times 99th percentile latency in seconds + some breathing room 每秒最大支撑的请求数 (99%平均响应时间 + 缓存值) 比如: 每秒能处理1000个请求, 99%的请求响应时间是60ms, 那么公式是: $1000 \times (0.060 + 0.012)$

基本得原则时保持线程池尽可能小, 他主要是为了释放压力, 防止资源被阻塞。 当一切都是正常的时候, 线程池一般仅会有1到2个线程激活来提供服务

`hystrix.threadpool.default.coreSize` 并发执行的最大线程数, 默认10

`hystrix.threadpool.default.maxQueueSize` BlockingQueue的最大队列数, 当设为 - 1, 会使用

SynchronousQueue, 值为正时使用LinkedBlockingQueue。该设置只会在初始化时有效, 之后不能修改 threadpool的queue size, 除非reinitialising thread executor。默认 - 1。

`hystrix.threadpool.default.queueSizeRejectionThreshold` 即使maxQueueSize没有达到, 达到 queueSizeRejectionThreshold该值后, 请求也会被拒绝。因为maxQueueSize不能被动态修改, 这个参数将允许我们动态设置该值。if maxQueueSize == 1, 该字段将不起作用

`hystrix.threadpool.default.keepAliveTimeMinutes` 如果corePoolSize和maxPoolSize设成一样 (默认实现) 该设置无效。如果通过plugin (<https://github.com/Netflix/Hystrix/wiki/Plugins>) 使用自定义 实现, 该设置才有用, 默认1。

`hystrix.threadpool.default.metrics.rollingStats.timeInMilliseconds` 线程池统计指标的时间, 默认10000

`hystrix.threadpool.default.metrics.rollingStats.numBuckets` 将rolling window划分为n个 buckets, 默认10

zuul:

zuul是什么?

Zuul包含了对请求的路由和过滤两个最主要的功能:

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。

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。

注意: Zuul服务最终还是会注册进Eureka

路由:

项目加入依赖:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

因为上文说过Zuul最终会注册进eureka 所以我们这里也依赖了eureka

yml文件:

```
server:
  port: 9000

eureka:

  client:
    serviceUrl:
      defaultZone: http://localhost:3000/eureka/
  instance:
    instance-id: zuul-1
    prefer-ip-address: true

spring:
  application:
    name: zuul
```

启动类:

```
/**
 * 想要咨询vip课程相关的同学加一下安其拉老师QQ: 3164703201
 * 想要往期视频的同学加一下妮可老师QQ: 2860884084
 * author: 鲁班学院-商鞅老师
 */
@SpringBootApplication
@EnableZuulProxy
public class AppZuul {

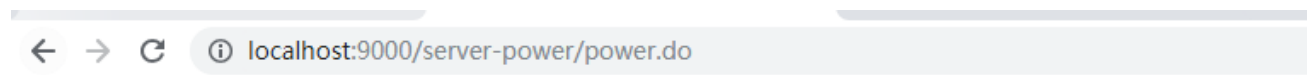
    public static void main(String[] args) {
        SpringApplication.run(AppZuul.class);
    }
}
```

这样 简单的zuul就搭建好了， 启动项目 我们就可以通过zuul然后加上对应的微服务名字访问微服务:

看看eureka上面的微服务名称

Application	AMIs	Availability Zones	Status
SERVER-ORDER	n/a (1)	(1)	UP (1) - order-2
SERVER-POWER	n/a (1)	(1)	UP (1) - power-1
ZUUL	n/a (1)	(1)	UP (1) - zuul-1

调用:



```
{"power": "value"}
```

到这里 一个简单的zuul已经搭建好了

在实际开发当中我们肯定不会是这样通过微服务调用，比如我要调用power 可能只要一个/power就好了 而不是/server-power

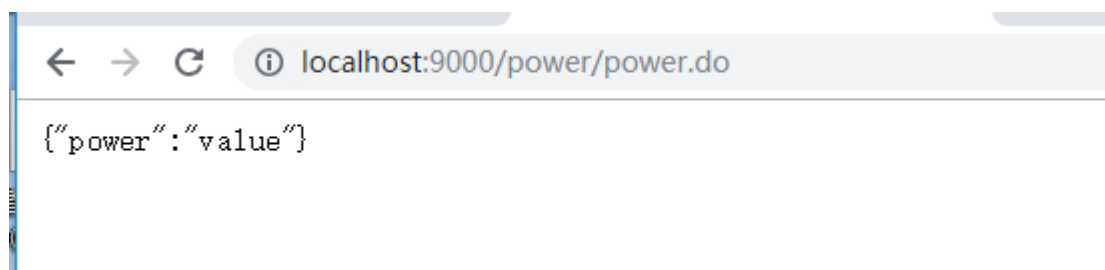
在yml加入以下配置即可:

```
zuul:
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

讲道理看意思都看得出来把，my***是自己制定的名字 这个就不解释了

注意/ **代表是所有层级 / * 是代表一层。 如果是/ * 的话 /power/admin/getUser.do 就不会被路由。

来看效果:



这时候我们能通过我们自定义的规则来访问了，但是还有一个问题，就是我们现在依然能用之前的微服务名调用，这样子是不合理的，第一是有多重地址了，第二，一般微服务名这种最好不要暴露在外。所以我们一般会禁用微服务名方式调用。

加入配置：

```
ignored-services: server-power
```

这里咱们先禁用power的看看：

← → ↻ ⓘ localhost:9000/server-power/power.do

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Jan 11 16:50:53 CST 2019

There was an unexpected error (type=Not Found, status=404).

No message available

这里能发现我们不能通过微服务名来调用了，不过这个配置

如果一个一个通过微服务名来配置难免有点复杂，所以一般这样配置来禁用所有：

```
ignored-services: "**"
```

可能有时候我们的接口调用需要一定的规范，譬如调用微服务的API URL前缀需要加上/api 对于这种情况，zuul也考虑到了并给出了解决方案：

```
zuul:
  prefix: /api
  ignored-services: "**"
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

加上一个prefix 即定义好了一个前缀，那么我们每次需要路由的时候需要加上一个/api的前缀

但是 这样有一个问题，就是这个/api前缀 会不会出现在我们路由后的IP地址中呢？因为有可能我们微服务提供的接口也是含有/api前缀的

答案是不会的。但是可以进行配置

```
zuul:
  prefix: /api
  strip-prefix: false
  ignored-services: "*"
  routes:
    mypower:
      serviceId: server-power
      path: /power/**
    myorder:
      serviceId: server-order
      path: /order/**
```

过滤器:

过滤器(filter)是zuul的核心组件 zuul大部分功能都是通过过滤器来实现的。 zuul中定义了4种标准过滤器类型, 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。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。

如果要编写一个过滤器, 则需继承ZuulFilter类 实现其中方法:

```
@Component
public class LogFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return FilterConstants.ROUTE_TYPE;
    }

    @Override
    public int filterOrder() {
        return FilterConstants.PRE_DECORATION_FILTER_ORDER;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() throws ZuulException {
        RequestContext currentContext = RequestContext.getCurrentContext();
        HttpServletRequest request = currentContext.getRequest();
        String remoteAddr = request.getRemoteAddr();
        System.out.println("访问者IP: "+remoteAddr+"访问地址:"+request.getRequestURI());
        return null;
    }
}
```

由代码可知，自定义的 zuul Filter需实现以下几个方法。

filterType:返回过滤器的类型。有 pre、route、post、error等几种取值，分别对应上文的几种过滤器。

详细可以参考 com.netflix.zuul.ZuulFilter.filterType()中的注释。

filterOrder:返回一个 int值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。

shouldFilter: 返回一个 boolean值来判断该过滤器是否要执行， true表示执行， false表示不执行。

run: 过滤器的具体逻辑。

禁用zuul过滤器 Spring Cloud默认为Zuul编写并启用了一些过滤器，例如DebugFilter、FormBodyWrapperFilter等，这些过滤器都存放在spring-cloud-netflix-core这个jar包 里，一些场景下，想要禁用掉部分过滤器，该怎么办呢？只需在application.yml里设置zuul...disable=true 例如，要禁用上面我们写的过滤器，这样配置就行了：
zuul.LogFilter.pre.disable=true

zuul容错与回退

zuul默认是整合了hystrix和ribbon的，提供降级回退，那如何来使用hystrix呢？

我们自行写一个类，继承FallbackProvider 类 然后重写里面的方法

```
@Override
public String getRoute() {
    return null;
}

@Override
public ClientHttpResponse fallbackResponse(String route, Throwable cause) {
    return null;
}
```

这里 会发现有这2个方法需要重写，那么如何来写呢？我们可以查阅官方文档：

If you would like to provide a default fallback for all routes, you can create a bean of type `FallbackProvider` and have the `getRoute` method return `*` or `null`, as shown in the following example:

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        return "*";
    }

    @Override
    public ClientHttpResponse fallbackResponse(String route, Throwable throwable) {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {
            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("Fallback".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }
}
```

这是官方提供的demo

代码：

```
class MyFallbackProvider implements FallbackProvider {
    @Override
    public String getRoute() {
        //制定为哪个微服务提供回退（这里写微服务名 写*代表所有微服务）
        return "*";
    }

    //此方法需要返回一个ClientHttpResponse对象 ClientHttpResponse是一个接口，具体的回退逻辑要实现此接口
    //route: 出错的微服务名      cause: 出错的异常对象
    @Override
    public ClientHttpResponse fallbackResponse(String route, final Throwable cause) {
        //这里可以判断根据不同的异常来做不同的处理，也可以不判断
        //完了之后调用response方法并根据异常类型传入HttpStatus
        if (cause instanceof HystrixTimeoutException) {
            return response(HttpStatus.GATEWAY_TIMEOUT);
        } else {
            return response(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    private ClientHttpResponse response(final HttpStatus status) {
        //这里返回一个ClientHttpResponse对象 并实现其中的方法，关于回退逻辑的详细，便在下面的方法中
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
```

```

        //返回一个HttpStatus对象 这个对象是个枚举对象， 里面包含了一个status code 和
        reasonPhrase信息
        return status;
    }

    @Override
    public int getRawStatusCode() throws IOException {
        //返回status的代码 比如 404, 500等
        return status.value();
    }

    @Override
    public String getStatusText() throws IOException {
        //返回一个HttpStatus对象的reasonPhrase信息
        return status.getReasonPhrase();
    }

    @Override
    public void close() {
        //close的时候调用的方法， 讲白了就是当降级信息全部响应完了之后调用的方法
    }

    @Override
    public InputStream getBody() throws IOException {
        //吧降级信息响应回前端
        return new ByteArrayInputStream("降级信息".getBytes());
    }

    @Override
    public HttpHeaders getHeaders() {
        //需要对响应报头设置的话可以在此设置
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        return headers;
    }
};
}
}

```

HystrixDashbord

Hystrix（注意 是单纯的Hystrix） 提供了对于微服务调用状态的监控（信息）， 但是， 需要结合 spring-boot-actuator 模块一起使用.

在包含了 hystrix的项目中， 引入依赖：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

```


(需要注意的是, 在Spring Finchley 版本以前访问路径是/hystrix.stream, 如果是Finchley 的话 还得在yml里面加入配置:

因为spring Boot 2.0.x以后的Actuator 只暴露了info 和health 2个端点,这里我们把所有端点开放。

这里会发现没有任何信息，因为我刚启动项目，我们来调用几个接口看看：

[illegible]

这些密密麻麻的，就是我们的微服务监控的信息，但是，这种json格式的字符串，难免会让人不太好阅读，所以，这时候需要我们的主角登场了：

HystrixDashbord

什么是HystrixDashbord/如何使用？

Dashbord 翻译一下的意思是 仪表盘，顾名思义，hystrix监控信息的仪表盘，那这个仪表盘到底是什么样子呢？以及 怎么来使用呢？

我们新建一个项目 加入依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

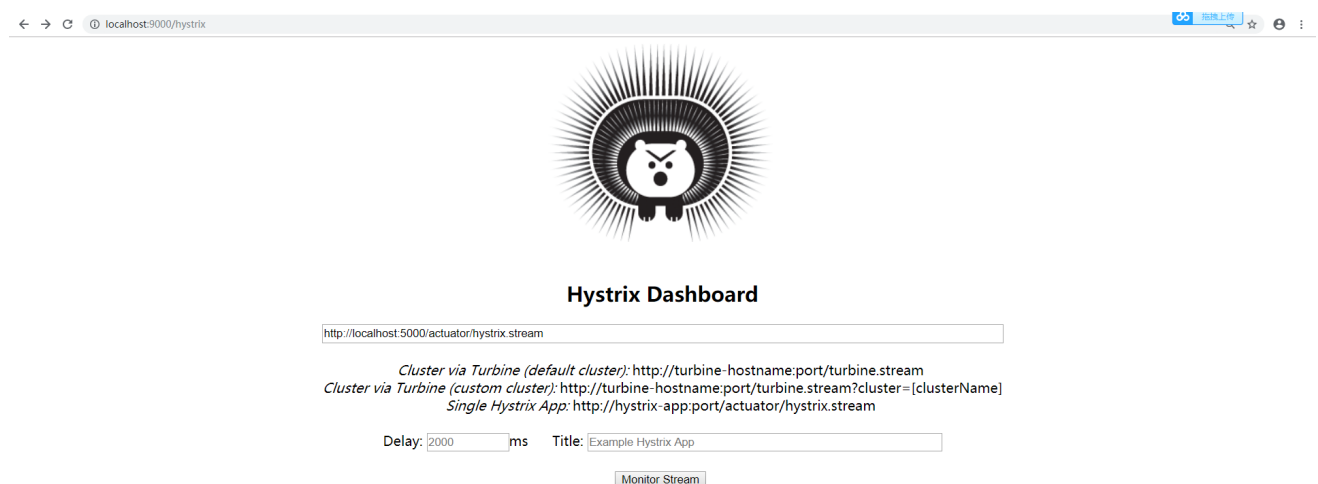
在spring boot启动类上面加入注解EnableHystrixDashboard

```
@SpringBootApplication
@EnableHystrixDashboard
public class AppHystrixDashbord {

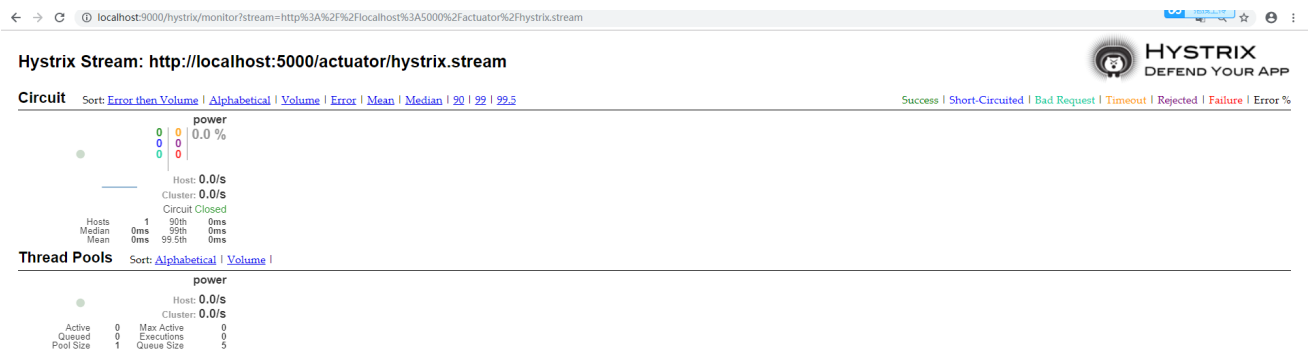
    public static void main(String[] args) {
        SpringApplication.run(AppHystrixDashbord.class);
    }

}
```

启动项目后访问/hystrix能看见一个类似tomcat的首页：



在中间这个输入框中，填入需要监控的微服务的监控地址 也就是/actuator/hystrix.stream点击按钮，就会跳转到仪表盘页面：



当然，如果你微服务没有发生过调用，那么这个页面就会一直显示加载中，我这里是调用后的效果。

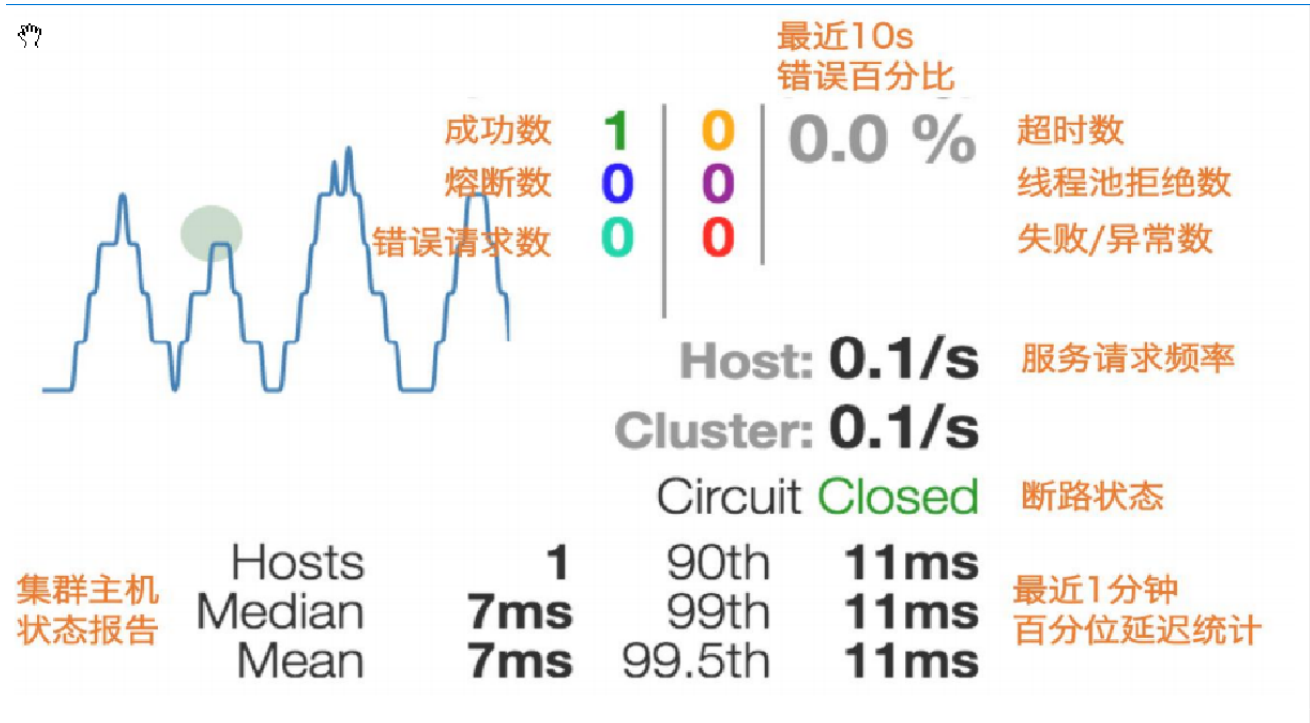
Hystrix仪表盘解释：

实心圆：共有两种含义。它通过颜色的变化代表了实例的健康程度，它的健康度从绿色

该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，就可以在大量的实例中快速的发现故障实例和高压力实例。

曲线：用来记录2分钟内流量的相对变化，可以通过它来观察到流量的上升和下降趋势。

整图解释：



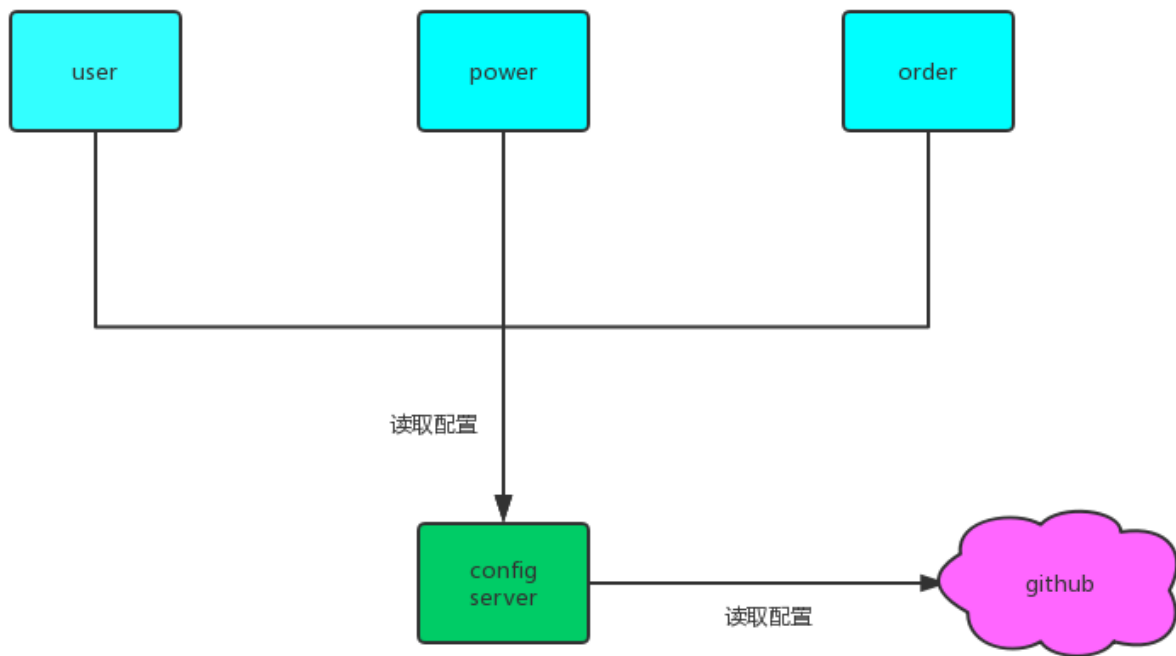
Spring-Cloud-Config

config是什么?

我们既然要做项目，那么就少不了配置，传统的项目还好，但是我们微服务项目，每个微服务就要做独立的配置，这样难免有点复杂，所以，config项目出来了，它就是为了解决这个问题：把你所有的微服务配置通过某个平台：

比如 github，gitlib 或者其他的git仓库 进行集中化管理（当然，也可以放在本地）。

可能这样讲有点抽象，我们来看一张图：



大概是这样一个关系

如何使用config?

刚刚讲完理论，那么我们来实践一下，怎么配置这个config呢？我们刚刚说过 由一个config server 来管理所有的配置文件，那么我们现在新建一个config server 项目 然后引入依赖：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

spring-cloud 的依赖我们就不提了

然后启动类上面加入注解@EnableConfigServer:

```
@SpringBootApplication
@EnableConfigServer
public class AppConfig {

    public static void main(String[] args) {
        SpringApplication.run(AppConfig.class);
    }
}
```

yaml配置:

```




server:
  port: 8080
spring:
  application:
    name: test

cloud:
  config:
    server:
      git:
        uri: https://github.com/513667225/my-spring-cloud-config.git #配置文件在github
上的地址
        #          search-paths: foo,bar* #Configserver会在 Git仓库根目录、 foo子目录, 以及所有以
        bar开始的子目录中查找配置文件。
        #          clone-on-start: true #启动时就clone仓库到本地, 默认是在配置被首次请求时, config
        server才会clone git仓库
        #native:
        #search-locations: classpath:/config #若配置中心在本地, 本地的地址

```

配置好以后, 我们先试试通过config server来读取配置

这里我在github上有一些配置文件:




 test-config.yml	测试提交	21 days ago
 test-config1.yml	测试提交	21 days ago
 test-config2.yml	测试提交	21 days ago

Help people interested in this repository understand your project by adding a README.
 Add a README

我们来看看test-config的内容:

1 contributor

22 lines (15 sloc) | 210 Bytes

Raw
 Blame
 History
 



```

1  spring:
2    profiles:
3      active: test
4
5
6  ---
7
8  server:
9    port: 8201
10
11 spring:
12   profiles: dev
13   application:
14     name: test-cloud-dev-2.0
15
16
17 ---
18
19 spring:
20   profiles: test
21   application:
22     name: test-cloud-test-2.0

```

那么如何通过config server来访问呢?

启动项目后，我们可以通过名字来读取里面的配置信息：

localhost:8080/test-config.yml

```
spring:
  profiles:
    active: test
```

那我们要获取dev环境/或者test环境下的配置呢？通过-隔开即可。

我们现在来访问 test-config-dev：

localhost:8080/test-config-dev.yml

```
server:
  port: 8201
spring:
  application:
    name: test-cloud-dev-2.0
  profiles:
    active: test
```

同理 如果要访问test环境下的配置，改为test即可

其实，config访问配置文件，是需要一个具体的访问规则的，那么这个访问规则到底是什么呢？我们可以在官网找到：

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

application就是配置文件的名字， profile就是对应的环境 label就是不同的分支 由这个规则可见，我们使用的是第二种规则，剩下的规则，同学们可以自己试试，对于yml 和properties类型config可以完美转换，也就是说你存的是yml 但是可以读取为properties类型的反过来也是如此：

客户端从config上获取配置

刚刚给大家简单演示了一下config 以及怎么读取配置，不过实际开发中，更多的不是我们人为去获取，而是由微服务从config上加载配置，那么，怎么来加载呢？

首先，我们需要在我们的微服务加入一个依赖声明他是config的客户端：

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>
```

需要注意的是，这个依赖不包括spring -boot依赖，也就是说，假设你这个项目要当作spring boot来启动的话，还得依赖spring boot

启动类不需要做改动，标准的spring boot启动类即可

需要注意的是yaml文件

以前我们对于spring boot的配置 是在application.yml里面配置的,现在从config上读取配置的话，还得需要一个bootstrap.yml配置文件

解释一下这个bootstrap.yml:

spring cloud有一个“引导上下文”的概念，这是主应用程序的父上下文。引导上下文负责从配置服务器加载配置属性，以及解密外部配置文件中的属性。和主应用程序加载application.(yml或properties)中的属性不同，引导上下文加载(bootstrap.)中的属性。配置在 bootstrap.*中的属性有更高的优先级，因此默认情况下它们不能被本地配置

那么我们application.yml配置文件里面 只需要做一些简单的配置就可以了：

```
spring:  
  application:  
    name: test-config
```

重点在于bootstrap.yml:

```
spring:  
  cloud:  
    config:  
      name: test-config #这是我们要读取的配置文件名 对应获取规则的{application}  
      profile: dev      #这个是要获取的环境 对应的便是{profile}  
      label: master     #这个就是获取的节点 对应的是{label}  
      uri: http://localhost:8080/ #这就是我们config server的一个地址
```


那么 他就会获取到我们刚刚看到的那个配置:

```
server:
  port: 8201

spring:
  profiles: dev
  application:
    name: test-cloud-dev-2.0
```

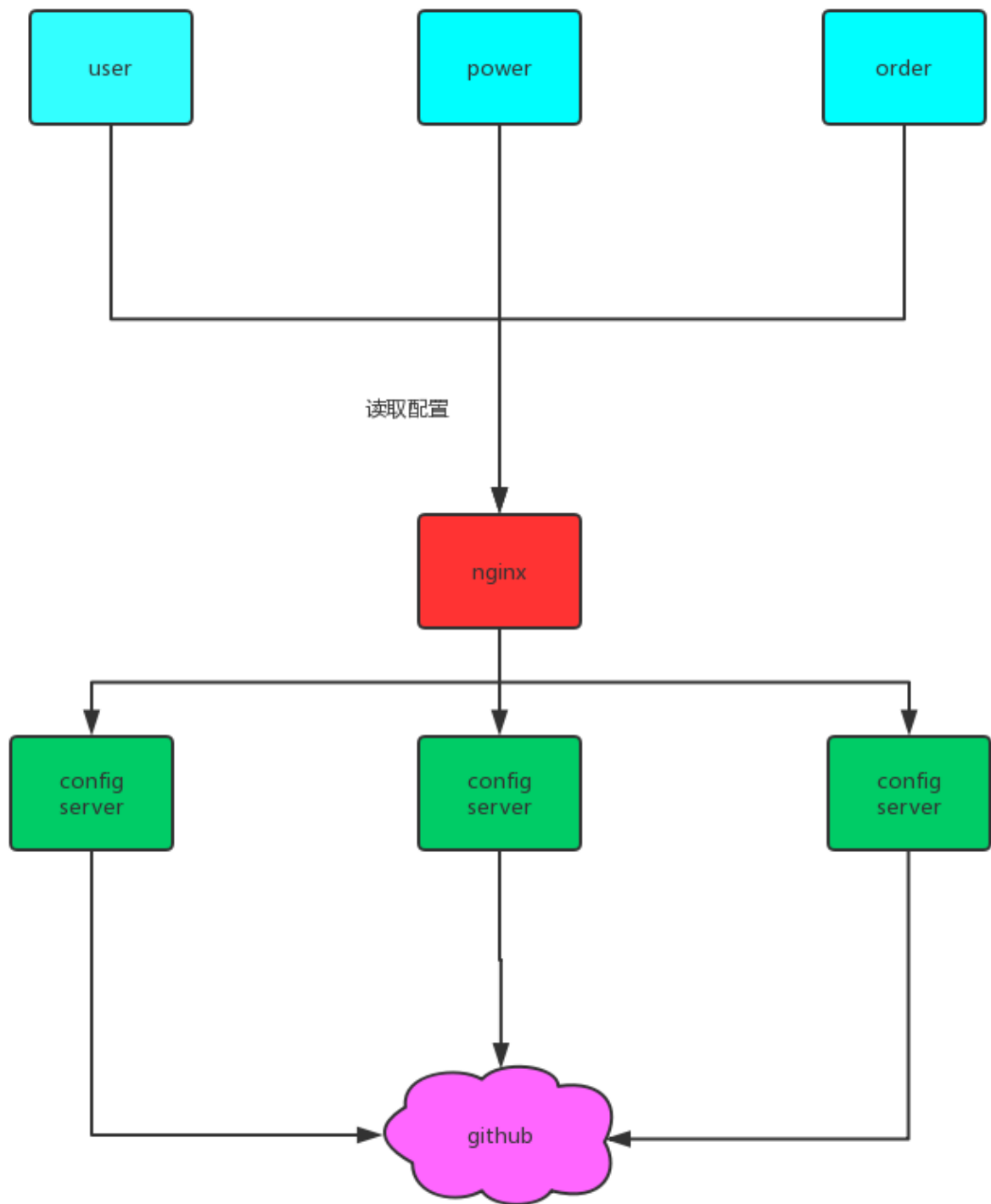
我们来测试一下 看看他会不会使用这个8201端口启动

```
2019-02-12 20:36:03.312 INFO 18300 --- [main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8201
2019-02-12 20:36:03.315 INFO 18300 --- [main] com.AppConfigClient : Started AppConfigClient in 12.812 seconds (JVM running for 13.3)
2019-02-12 20:36:05.288 ERROR 18300 --- [nfoReplicator-0] c.n.d.s.t.d.RedirectingEurekaHttpClient : Request execution error
```

这里 我们查看启动信息，能发现他现在使用的是我们从config server上读取到的配置。

spring cloud config 高可用

config 高可用可以通过很多种方式，比如说搭建一个nginx:



或者config server注册到eureka上，client端也注册到eureka上，则已经实现高可用

如何注册就不提了，需要注意的点就是当config server都注册完之后 client的配置文件进行以下改动：

```
spring:
  cloud:
    config:
      name: test-config
      profile: dev
      label: master
      discovery:
        enabled: true
        service-id: test-config
  eureka:
    client:
      serviceUrl:
        defaultZone: http://localhost:3000/eureka/
```

spring -cloud -sleuth

分布式链路跟踪

分布式链路跟踪介绍

本节课来讨论微服务“跟踪”大家先看几个问题，对于一个大型的微服务架构系统，会有哪些常见问题？

如何串联调用链，快速定位问题

如何厘清微服务之间的依赖关系

如何进行各个服务接口的性能分析

如何跟踪业务流的处理

sleuth介绍：

spring Cloud Sleuth为 spring Cloud提供了分布式跟踪的解决方案，它大量借用了Google Dapper、Twitter Zipkin和 Apache HTrace的设计先来了解一下 Sleuth的术语，Sleuth借用了 Dapper的术语。

span（跨度）：基本工作单元。span用一个64位的id唯一标识。除ID外，span还包含其他数据，例如描述、时间戳事件、键值对的注解（标签），spanID、span父ID等。span被启动和停止时，记录了时间信息。初始化span被称为"rootspan"，该span的id和trace的ID相等。

trace（跟踪）：一组共享"rootspan"的span组成的树状结构称为trace。trace也用一个64位的ID唯一标识，trace中的所有span都共享该trace的ID

annotation（标注）：annotation用来记录事件的存在，其中，核心annotation用来定义请求的开始和结束。

CS（Client sent客户端发送）：客户端发起一个请求，该annotation描述了span的开始。

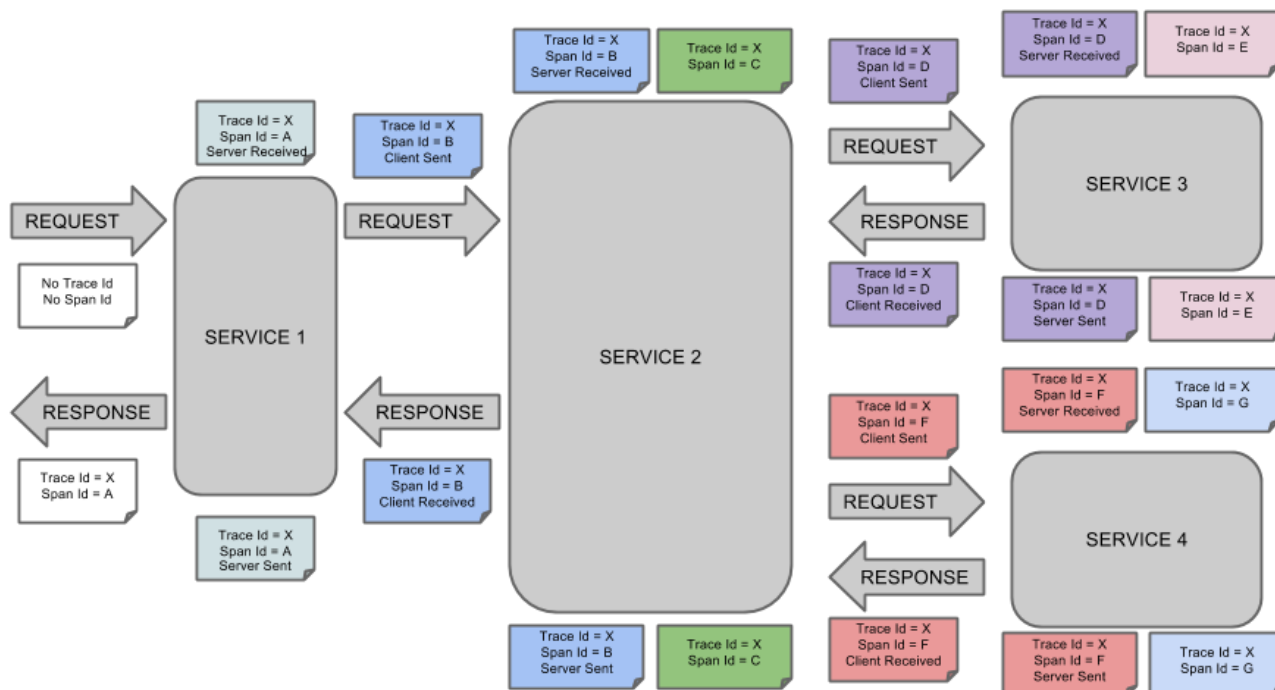
SR（server Received服务器端接收）：服务器端获得请求并准备处理它。如果用SR减去CS时间戳，就能得到网络延迟。

SS（server sent服务器端发送）：该annotation表明完成请求处理（当响应发回客户端时）。如果用SS减去SR时间戳，就能得到服务器端处理请求所需的时间。

CR (Client Received客户端接收)：span结束的标识。客户端成功接收到服务器端的响应。如果 CR减去 CS时间戳，就能得到从客户端发送请求到服务器响应的所需的时间

Spring Cloud Sleuth可以追踪10种类型的组件：async、Hystrix，messaging，websocket，rxjava，scheduling，web（Spring MVC Controller，Servlet），webclient（Spring RestTemplate）、Feign、Zuul

下面我们通过一张图来看一下一个简单的微服务调用链：



这张图是spring cloud 官方给出的示例图

图片详细讲了我们上文所说的概念在调用链中 处于什么状态以及改变

sleuth整合Zipkin实现分布式链路跟踪

Zipkin简介:

Zipkin是 Twitter开源的分布式跟踪系统，基于 Dapper的论文设计而来。它的主要功能是收集系统的时序数据，从而追踪微服务架构的系统延时等问题。Zipkin还提供了一个非常友好的界面，来帮助分析追踪数据。官网地址：<http://zipkin>.

为什么要Zipkin

因为sleuth对于分布式链路的跟踪仅仅是一些数据的记录，这些数据我们人为来读取和处理难免会太麻烦了，所以我们一般吧这种数据上交给Zipkin Server 来统一处理。

编写一个Zipkin Server

我们新建一个项目，然后引入依赖:

```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-ui</artifactId>
  <version>2.8.4</version>
</dependency>
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-server</artifactId>
  <version>2.8.4</version>
</dependency>
```

在启动类上加入注解:@EnableZipkinServer:

```
@EnableZipkinServer
@SpringBootApplication
public class AppSleuth {
    public static void main(String[] args) {
        SpringApplication.run(AppSleuth.class);
    }
}
```

yml文件加上如下配置:

```
management:
  metrics:
    web:
      server:
        autoTimeRequests: false
```

这个配置解释一下: 在zipkin2.7.x以后便不支持自定义服务器需要使用官方的版本或者Docker 但是如果还是要使用的话就得加上这个配置。

完成上面的步骤之后, 我们启动项目, 你会发现Zipkin 的专属图标, 而且细心的同学会发现 他是基于spring boot 来的,

然后打开浏览器 访问: /zipkin/

The screenshot shows the Zipkin web interface at localhost:9000/zipkin/. The interface has a dark header with navigation links: "Zipkin", "Investigate system behavior", "Find a trace", "View Saved Trace", and "Dependencies". A "Go to trace" input field is on the right. Below the header is a search form with the following fields:

- Service Name:** A dropdown menu with "all" selected.
- Span Name:** A dropdown menu with "Span Name" selected.
- Lookback:** A dropdown menu with "1 hour" selected.
- Annotations Query:** A text input field containing "e.g. 'http.path=/foo/bar/' and cluster=foo and cache.miss".
- Duration (µs) >=:** An empty text input field.
- Limit:** A text input field with "10".
- Sort:** A dropdown menu with "Longest First" selected.

Below the form is a blue button labeled "Find Traces" and a help icon. At the bottom, a light blue message box says "Please select the criteria for your trace lookup."

看到这个页面，基本上你的zipkin server搭建完毕了

这是用来查询分布式链路数据的页面，这里列出了查询条件，从第一行开始从左到右分别是：

微服务名称（就是你配置文件里面的application name），span（即上文所解释的）名称，时间段，自定义查询条件，一次调用链的持续时间，一页数量，排序规则

目前来讲，我们肯定是查询不到数据的，我们把我们自己的微服务和 sleuth整合 并把数据上传到zipkin server

sleuth微服务整合Zipkin

首先 我们需要依赖sleuth 和 sleuth与zipkin的整合依赖:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>

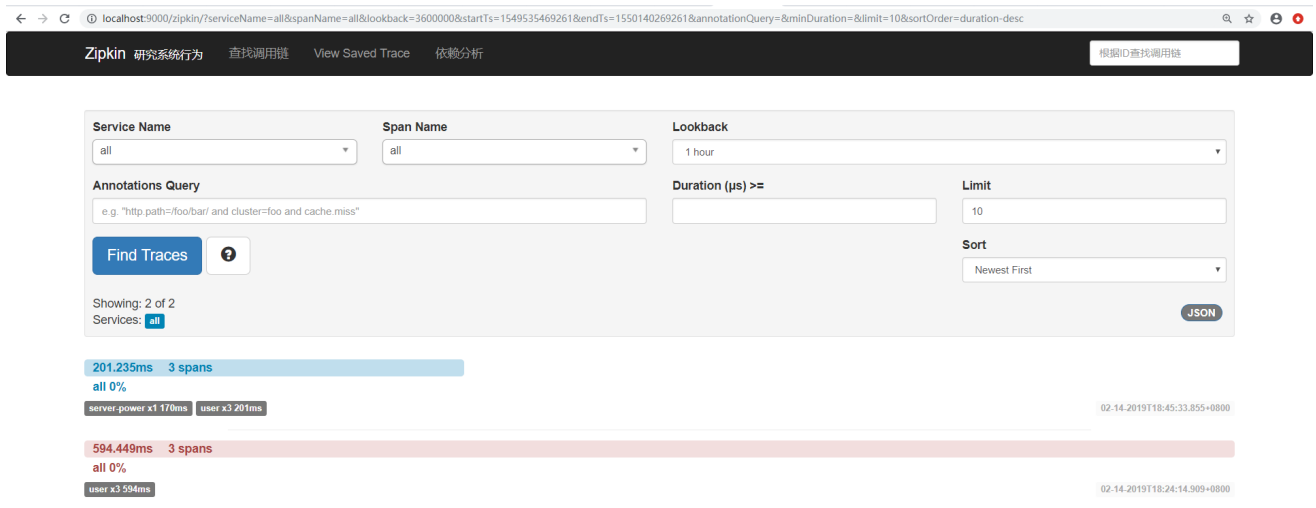
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

然后在微服务中加入以下配置:

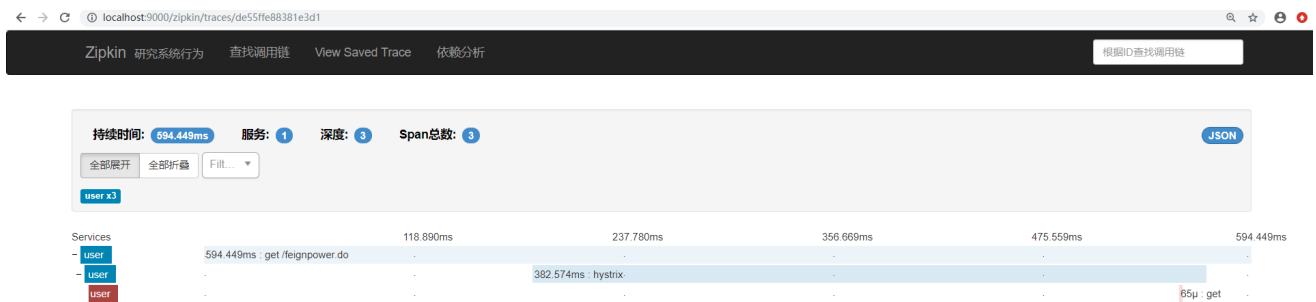
```
spring:
  zipkin:
    base-url: http://localhost:9000 #指定zipkin server地址
  sleuth:
    sampler:
      probability: 1.0 #request采样的数量 默认是0.1 也即是10% 顾名思义 采取10%的请求数据
      因为在分布式系统中，数据量可能会非常大，因此采样非常重要。我们示例数据少最好配置为1全采样
```

然后启动微服务并模拟一次调用链 我这里是用户微服务调用了power微服务（注意，每个微服务都需要和zipkin整合）

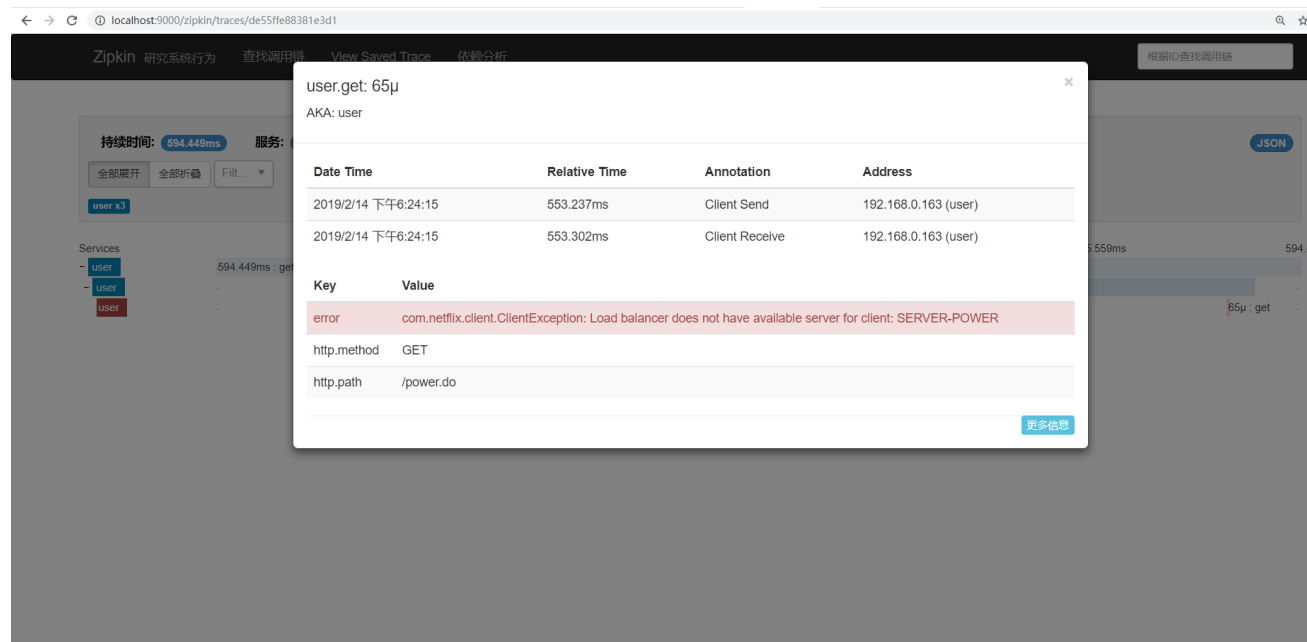
调用完成之后 我们去zipkin server 页面去看看：



这里我模拟了2条请求 一个是正常的 一个是不正常的 正常的就不看了 我们看看不正常的是什么样子的



他会显示你的微服务调用耗时并且在哪个阶段出了错误 还会把微服务名给显示出来（因为我这里就是在user这里出错的 所以这里显示的是user 如果是power微服务出错了 那么这个微服务名就会变成power） 而且可以点击进去查看详情：



他会把具体的错误信息给你展示出来 方便错误的定位。

其他的你们可以自己去测试看看

zipkin server 数据持久化问题

刚刚我们介绍了如何把分布式链路调用信息上传到 zipkin server 但是 有一个问题：

当zipkin重启后我们的分布式链路数据全部清空了。

因为zipkin server 默认数据是存储在内存当中， 所以当你服务重启之后内存自然而然也就清空了。

使用Elasticsearch 做数据持久化

我们这里借用ES来做数据持久化， 当然 还可以用ELK来做， 我们这里演示ES

关于Elasticsearch 具体的介绍 我们本文暂时不讨论， 没学习过的同学可以把他当作mysql来看待

Elasticsearch 下载地址： <https://www.elastic.co/cn/downloads/elasticsearch>

下载完是个压缩包 解压出来 打开bin目录 找到elasticsearch.bat文件启动

等他启动一会儿然后在页面上输入localhost:9000看见如下信息说明Elasticsearch 启动好了：


```
{
  "name" : "B2IhzVK",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "SEt3CfnASDW4aLTAC7g26w",
  "version" : {
    "number" : "6.6.0",
    "build_flavor" : "default",
    "build_type" : "zip",
    "build_hash" : "a9861f4",
    "build_date" : "2019-01-24T11:27:09.439740Z",
    "build_snapshot" : false,
    "lucene_version" : "7.6.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum_index_compatibility_version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

zipkin 与 Elasticsearch整合：首先 我们在我们的zipkin server里面引入依赖:

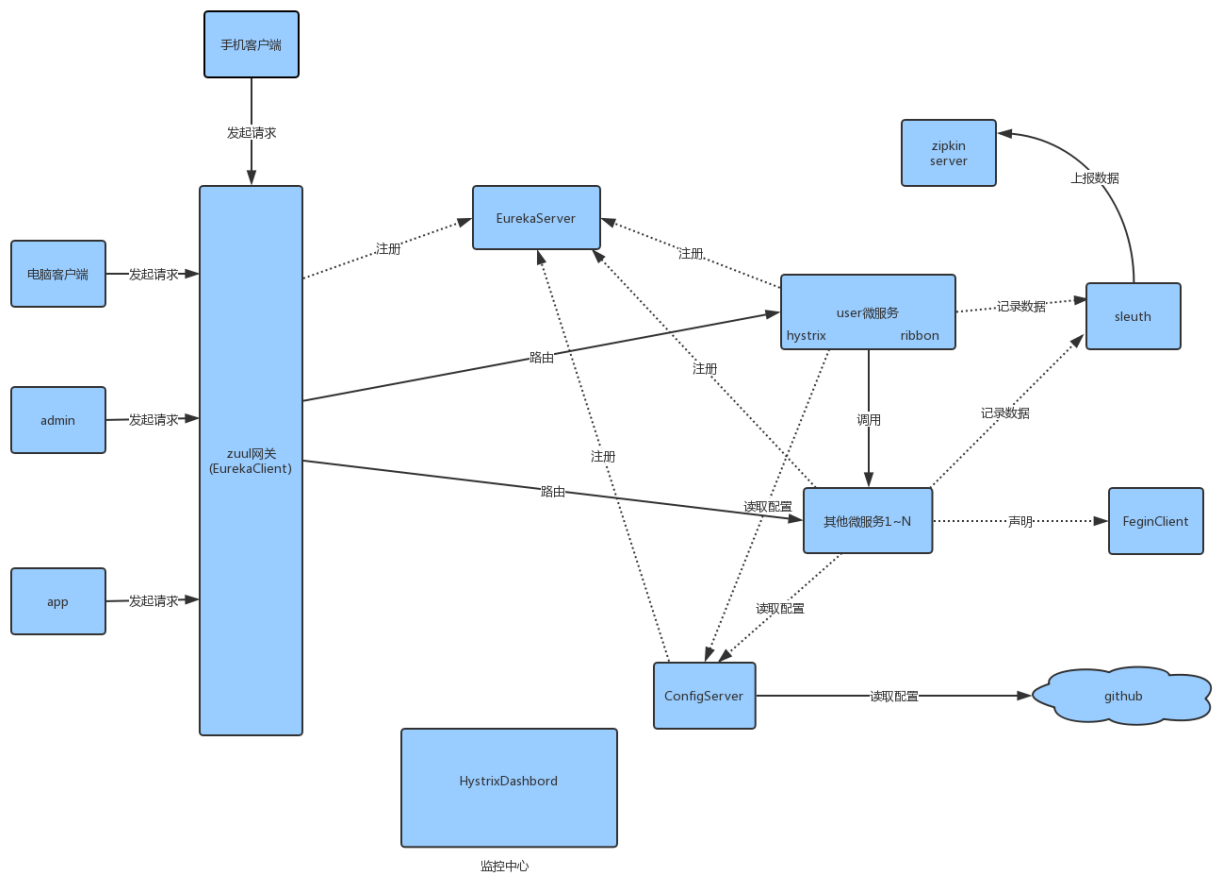
```
<dependency>
  <groupId>io.zipkin.java</groupId>
  <artifactId>zipkin-autoconfigure-storage-elasticsearch-http</artifactId>
  <version>2.3.1</version>
</dependency>
```

然后在yml加入配置:

```
zipkin:
  storage:
    type: elasticsearch
    elasticsearch:
      cluster: elasticsearch
      hosts: http://localhost:9200 #使用Elasticsearch的连接地址
      index: zipkin
```

至此 zipkin的数据便和Elasticsearch整合起来了，现在再启动zipkin server 并且存储几条数据，就算重启，数据还会在上面。

微服务总结



这里我画了张图简单总结了以下我们的spring cloud 的学习

由上图可以发现， spring cloud 把各个组件相互配合起来， 整合成一套成熟的微服务架构体系

其中， 由eureka做服务注册与发现， 很好的把各个服务链接起来

ribbon+fegin提供了微服务的调用和负载均衡解决方案

hystrix 负责监控微服务之间的调用情况， 以及降级和熔断保护

Hystrix dashboard监控Hystrix的熔断情况以及监控信息以图形化界面展示

spring cloud config 提供了统一的配置中心服务

所有外来的请求由zuul统一进行路由和转发， 起到了API网关的作用

Sleuth+Zipkin把我们微服务的追踪数据记录下来并展示方便我们进行后续分析