一、微服务

1 什么是微服务

目前，对微服务业界并没有一个统一的、标准的定义，但通常而言，微服务架构是一种架构模式或者说是一种架构风格，他提倡将一个单一应用程序划分为一组小的服务，每个服务运行在其独立的进程中，服务之间相互协调、相互配合，为用户提供最终价值。服务之间采用轻量级的通信机制互相沟通（通常是基于Http的RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生成环境、类生产环境等。另外应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言，工具对其进行构建。可以有一个非常轻量级的集中式管理来协调这些服务，可以使用不同的语言来编写服务，也可以使用不同的数据存储。

微服务化的核心就是讲传统的一站式应用，根据业务拆分成一个一个的服务，彻底的去耦合，每一个微服务提供单个业务功能的服务，一个服务做一件事，从技术角度看就是一种小而独立的处理过程，类似进程概念，能够自行单独启动或销毁，拥有自己独立的数据库。

微服务与微服务架构？

微服务强调的是服务的大小，他关注的是某个点，是具体解决某个问题/提供落地对应服务的一个服务应用，而微服务架构是一种架构模式，面向的是多个服务。springboot与springcloud的区别

2 微服务的优缺点

优点：

1.每个服务足够内聚、足够小，代码容易理解这样能聚焦一个指定的业务功能或业务需求

2.开发简单、开发效率高，一个服务可能就是专一的只干一件事

3.微服务能够被小团队单独开发，2到5个人就能够开发

4.微服务是松耦合的，是有功能意义的服务，无论在开发阶段或部署阶段都是独立的

5.微服务能使用不同的语言开发

6.易于和第三方集成，微服务允许容易且灵活的方式集成自动部署

7.微服务易于被开发人员理解、修改和维护，

8.微服务允许利用融合最新的技术。对其他模块没有影响

9.微服务只是业务逻辑的代码，不会和Html、CSS或其他页面组件混合

10.每个微服务都可以有自己的存储能力，用自己的数据库，也可以有统一的数据库

缺点：

1.开发人员需要处理分布式系统的复杂性

2.多服务运维难度高、随着服务的增加，运维的压力也在增大

3.系统部署依赖

4.服务间的通信成本

5.数据一致性

6.系统集成测试

7.性能监控

等等.....

3 微服务技术栈有哪些

在微服务中包含多种技术：

服务开发 springboot、spring、springMVC等等

服务配置与管理 Netflix的Archaius、阿里的Diamond等等

服务注册发现 Eureka、zookeeper 、Consul等

服务调用 Rest、RPC、gRPC

服务熔断器 Hystrix、Envoy等

负载均衡 Ribbon、Nginx等

服务接口调用 Feign等

消息队列 kafka、RabbitMQ、ActiveMQ等

服务配置中心管理 SpringCloudConfig、Chef等

服务路由（API网关） Zuul等

服务监控 Zabbix、Nagios、Metrics、Spectator等

全链路追踪 Zipkin、Brave、Dapper等

服务部署 Docker、OpenStack、Kubernetes等

数据流操作开发包 SpringCloud Stream（封装与Redis，Rabbit，kafka等发送接收消息）

事件消息总线 SpringCloud Bus

.....

4 为什么选择SpringCloud

当前的一些微服务框架对比

功能点	SpringCloud	Motan	gRPC	Thrift	Dubbo
功能定位	完整的微服务框架	RPC框架，但整合了ZK或Consul，实现集群环境的基本的服务注册与发现	RPC框架	RPC框架	服务框架
支持Rest	是　Ribbon支持多种可插拔的序列化选择	否	否	否	否
支持RPC	否	是（Hession2）	是	是	是
支持多语言	是	否	是	是	否
服务注册发现	是（Eureka）　Eureka服务注册表，Karyon服务端框架支持服务自注册和健康检查	是（zookeeper/consul）	否	否	是
负载均衡	是（服务端Zuul+客户端Ribbon）Zuul服务，动态路由　云端负载均衡Eureka（针对中间层服务器）	是（客户端）	否	否	是（客户端）
	Netflix Archaius SpringCloud Config Server集中配置	是（Zookeeper提供）	否	否	否
服务调用链监控	是（zuul）Zuul提供边缘服务，API网关	否	否	否	否
高可用/容错	是（服务端Hystrix+客户端Ribbon）	是（客户端）	否	否	是（客户端）
典型应用案例	Netflix	Sina	Google	Facebook	
社区活跃程度	高	一般	高	一般	

功能点	SpringCloud	Motan	gRPC	Thrift	Dubbo
学习难度	中等	低	高	高	低
文档丰富度	高	一般	一般	一般	高
其他	SpringCloud Bus为我们的应用程序带来了更多管理端点	支持降级	Netflix内部在开发集成gRPC	IDL自定义	实践的公司比较多

二、SpringCloud概述

1 SpringCloud是什么

SpringCloud，基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现，配置中心，全链路监控、服务网关、负载均衡、熔断器等组件，除了基于Netflix的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。

SpringCloud利用SpringBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，SpringCloud为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、熔断器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等，他们都可以用SpringBoot的开发风格做到一键启动和部署。

SpringBoot并没有重复制造轮子，他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SpringBoot风格进行在封装屏蔽掉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。

SpringCloud是分布式微服务架构下的一站式解决方案，是各个微服务架构技术集合体，微服务的全家桶

2 SpringCloud与SpringBoot

SpringBoot专注于快速方便的开发单个个体微服务。

SpringCloud是关注全局的微服务协调治理框架，他将SpringBoot开发的一个个单体微服务整合起来管理。为各个微服务之间提供了配置管理、服务发现、熔断器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等集成服务。

SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot。属于依赖关系。

3 SpringCloud与Dubbo

SpringCloud与Dubbo比较

	Dubbo	SpringCloud
服务注册中心	Zookeeper	SpringCloud Netflix Eureka
	RPC	REST API
服务监控	Dubbo-monitor	SpringBoot Admin
	不完善	SpringCloud Netflix Hystrix
	无	SpringCloud Netflix Zuul
	无	SpringCloud Config
服务跟踪	无	SpringCloud Sleuth
	无	SpringCloud Bus
	无	SpringCloud Stream
	无	SpringCloud Task
……	……	……

最大的区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于Http的REST方式

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

品牌机与组装机的区别

SpringCloud功能比Dubbo更加强大，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、SpringBoot、SpringData、SpringBatch等其他Spring项目完美融合，这些对于微服务来说至关重要。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很可能因为一个环节就导致整个系统无法运行，除非你对所用技术足够熟悉，否则用起来总是会让人不怎么放心。而SpringCloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要使用非原装组件的东西，就需要对其有足够的了解。

社区支持和更新力度方面，SpringCloud也优于Dubbo

4 SpringCloud参考文档

1.官网文档

2.https://springcloud.cc/spring-cloud-dalston.html

3.https://springcloud.cc/spring-cloud-netflix.html

4.SpringCloud中国社区 springcloud.cn

三、Eureka

1 Eureka是什么

Eureka是Netflix的一个子模块，也是核心模块之一。Eureka是一个基于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。服务注册与发现对于微服务架构来说是非常重要的，有了服务发现与注册，只需要使用服务的标识符，就可以访问到服务，而不需要修改服务调用的配置文件了。功能类似于Dubbo的注册中心，比如Zookeeper。

2 Eureka的基本架构

SpringCloud封装了Netflix公司开发的Eureka模块来实现服务注册与发现。

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，他是服务注册中心。

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（如Zuul等）就可以通过Eureka Server来发现系统中的其他微服务，并执行相关逻辑。

Eureka包含两个组件：Eureka Server和Eureka Client

Eureka Server提供服务注册服务：各个节点启动后，会在Eureka Server中进行注册，这样Eureka Server中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以再界面中直观的看到。

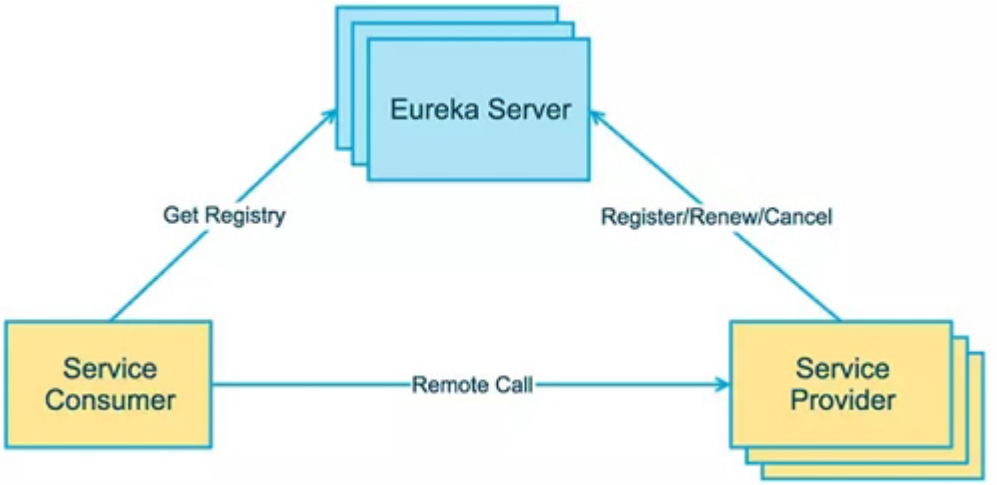
Eureka Client是一个Java客户端，用于简化与Eureka Server的交互。客户端同时也具备一个内置的、使用轮询（round-robin）负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳（默认周期为30秒）。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，Eureka Server将会从服务注册表中把这个服务节点移除（默认90秒）

3 Eureka三大角色

Eureka Server：提供服务注册与发现

Service Provider：服务提供方，将自身服务注册到Eureka，从而是服务消费方能够找到。

Service Consumer： 服务消费方，从Eureka获取注册服务列表，从而消费服务



4 构建步骤

4.1 服务注册中心EurekaServer的构建

1.创建一个SpringBoot项目

2.引入相关的依赖，EurekaServer需要在pom文件中引入以下依赖

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

    <groupId>com.example</groupId>
    <artifactId>eureka</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>eureka</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <!-- Springcloud版本 -->
        <spring-cloud.version>Finchley.SR1</spring-cloud.version>
    </properties>
```

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <!-- 添加SpringCloud相关依赖的版本管理 -->
    <dependencyManagement>
        <dependencies>
            <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>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

注意： 在Finchley版本过后EurekaServer可EurekaClient的依赖发生变化

以前

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

<!-- EurekaClient -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

升级Finchley版本过后

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

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

3.创建一个application.yml配置文件

```
server:
  port: 7001 #端口号


eureka:
  instance:
    hostname: localhost #eureka服务端实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: false #false表示自己就是注册中心,职责就是维护服务实例，不需要去检索服务
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  #设置EurekaServer
交互的地址  查询服务和注册服务都依赖这个地址
```

4.编写主配置类，添加@EnableEurekaServer注解

SpringCloud的其他组件，也是这样类似的注解@EnableXXXXX

```
@SpringBootApplication
//添加该注解，表示为EurekaServer启动类，接受其他服务的注册
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class, args);
    }
}
```

4.2 将微服务进行注册

1.创建一个需要注册的SpringBoot项目

2.添加EurekaClient相关依赖

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

    <groupId>com.example</groupId>
    <artifactId>micro-service-8081</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>micro-service-8081</name>
    <description>Demo project for Spring Boot</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <!-- Springcloud版本 -->
        <spring-cloud.version>Finchley.SR1</spring-cloud.version>
    </properties>

    <dependencies>

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

        </dependency>
```

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <!-- SpringCloud相关依赖版本管理 -->
    <dependencyManagement>
        <dependencies>
            <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>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

3.修改配置文件，添加注册中心相关配置

```
server:
  port: 8081

spring:
  application:
    name: micro-service-8081 #应用程序名称

eureka:
  client:
    service-url: #添加注册中心地址，将客户端注册进eureka服务列表内
      defaultZone: http://localhost:7001/eureka/
```

4.修改主配置类，添加@EnableEurekaClient注解

```
@SpringBootApplication
//标注为Eureka客户端，本服务启动后，会自动注册进Eureka服务中
@EnableEurekaClient
public class MicroService8081Application {
    public static void main(String[] args) {
        SpringApplication.run(MicroService8081Application.class, args);
    }
}
```

5.先启动EurekaServer，在启动EurekaClient完成服务注册

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICRO-SERVICE-8081	n/a (1)	(1)	UP (1) - 192.168.0.104:micro-service-8081:8081

5 完善配置

5.1 修改服务标识符

服务注册后，若没有设置则默认为 主机地址:服务名称:端口

可以通过修改配置文件完成设置 通过修改eureka.instance.instance-id

```
eureka:
  instance:
    instance-id: micro-service-8081 #服务标识符
    prefer-ip-address: true #鼠标悬停在标识符上显示IP
```

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
MICRO-SERVICE-8081	n/a (1)	(1)	UP (1) micro-service-8081 修改完成

5.2 info内容构建

若不对info内容进行构建，点击标识符跳转info页面报404

步骤：

1.添加pom依赖

```
<!-- 引入actuator监控信息依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2.修改配置文件，添加info信息

```
info:
  app.name: micro-service-8081
  company.name: www.demo.com
  artifactId: micro-service-8081
  version: 0.0.1-SNAPSHOT
```

6 自我保护机制

某个时刻某个微服务不可用，Eureka不会立即清理，依旧会保留该微服务的信息。

默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer会注销该实例（默认为90秒）。但是当网络发生故障时，微服务与EurekaServer之间的无法正常通信，EurekaServer无法接收到微服务的心跳，但微服务本身其实是健康的，这个微服务实例此时不应该被注销。Eureka通过"自我保护模式"来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能是发生网络故障），那么这个节点就会进入自我保护模式。一旦进入该模式，EurekaServer就会保护服务注册表中的信息。不再删除服务注册中心的数据（也就是不能注销任何微服务）。当网络故障恢复后，该EurekaServer节点会自动退出自我保护模式。

在自我保护模式中，EurekaServer会保护服务注册表中的信息，不再注销任何服务实例。当它收到的心跳数重新恢复到阈值以上时，该EurekaServer节点就会自动退出自我保护模式。他的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。

综上，自我保护模式是一种应对网络异常的安全措施。它的架构哲学是宁可同时保留所有微服务（健康的和不健康的都保留），也不盲目的删除任何健康的服务。使用自我保护机制，可以让Eureka集群更加健壮、稳定。

在SpringCloud中，可以使用eureka.server.enable-self-preservasion = false禁用自我保护模式。

7 服务发现

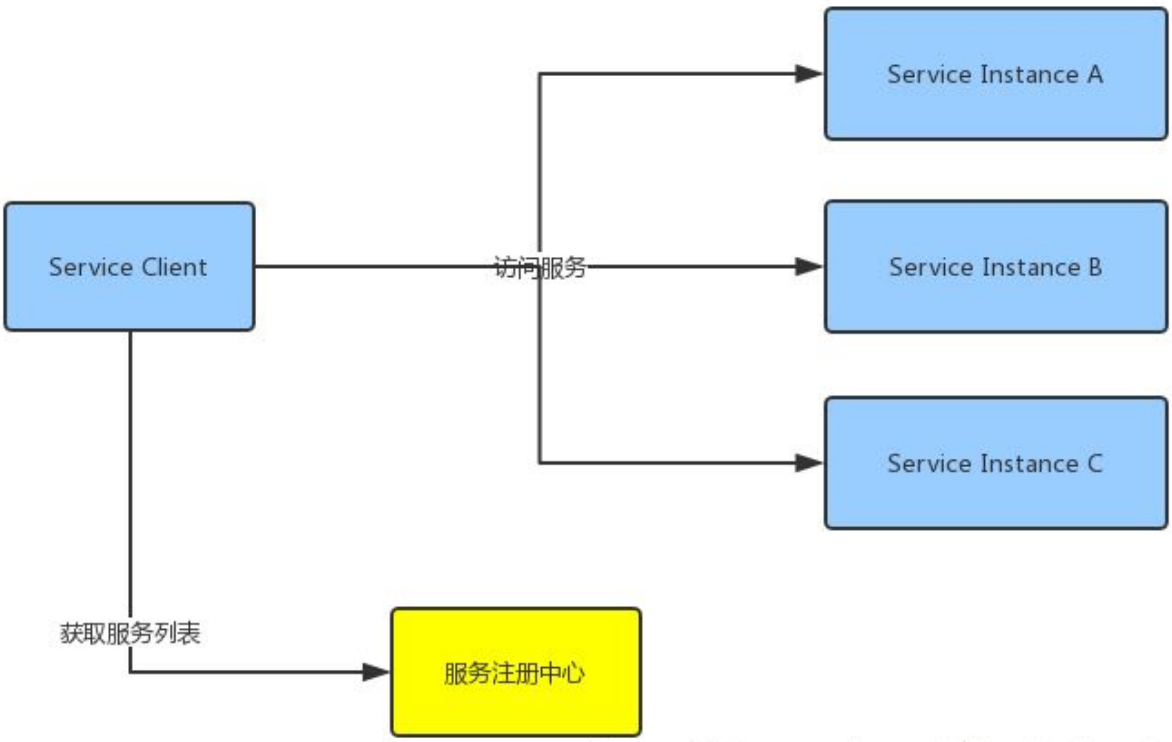
7.1 什么是服务发现

与访问DNS通过域名解析ip类似，服务发现就是通过一个标志来获取服务列表，并且这个服务列表是能够随着服务的状态而动态变更。

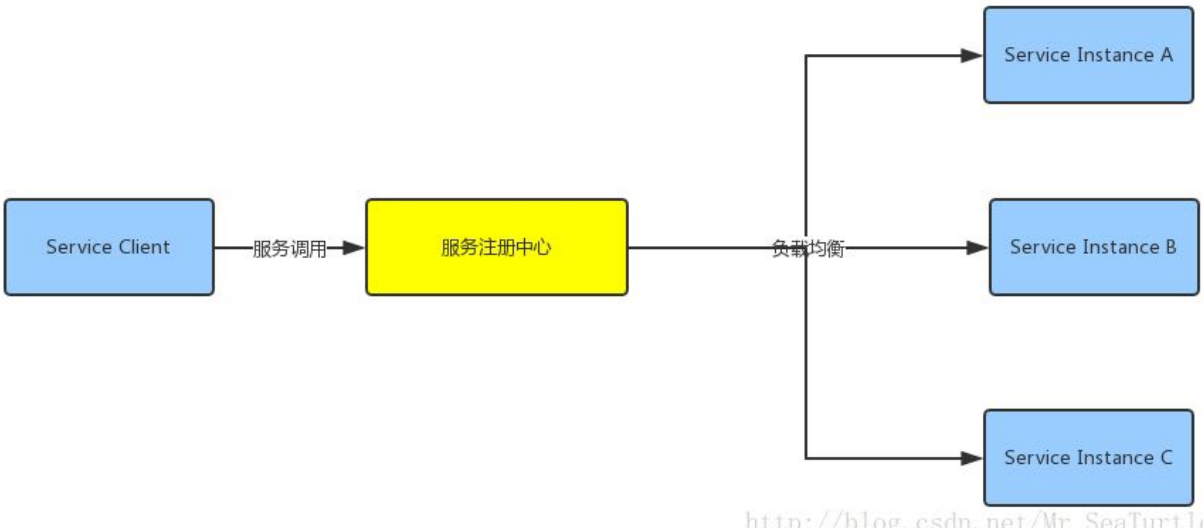
7.2 服务发现的模式

主要存在两种模式：客户端模式与服务端模式。两者本质的区别在于客户端是否保存服务列表信息。

客户端模式：要进行微服务调用，首先要进行的是到服务注册中心获取服务列表，然后再根据调用端本地的负载均衡策略，进行服务调用。



服务端模式：调用方直接向服务注册中心进行请求，服务注册中心再通过自身负载均衡策略，对微服务进行调用。在这个模式下，调用方不需要再自身节点维护服务发现逻辑以及服务注册信息，这个模式相对来说比较类似DNS模式。



对比：

客户端模式	服务端模式
只需要周期性获取列表，在调用服务时可以直接调用，少了一个RT。但需要在每个客户端维护获取列表的逻辑。	简单，不需要在客户端维护获取列表的逻辑。
可用性高，即使注册中心出现故障也能正常工作	可用性有路由器中间件决定，路由中间件故障则所有服务不可用，同时，由于所有调度以及存储都由中间件服务器完成，中间件服务器可能会面临过高的负载。
服务上下线对调用方有影响（会出现短暂调用失败）	服务上下线调用方无感知

目前来说，大部分服务发现的实现都采取了客户端模式。

7.3 Eureka的服务发现

通过DiscoveryClient实现

1.主配置类添加@EnableDiscoveryClient注解

```
@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
public class MicroService8081Application {

    public static void main(String[] args) {
        SpringApplication.run(MicroService8081Application.class, args);
    }
}
```

2.通过DiscoveryClient获取服务列表

```
@RestController
public class DiscoveryController {

    @Autowired
    private DiscoveryClient client;

    public void discovery(){
        //获取所有服务名列表
        List<String> servicesList = client.getServices();

        //根据服务名获取实例列表
        List<ServiceInstance> instances = client.getInstances("micro-service");

        for(ServiceInstance instance:instances){
            //获取服务ID
            String serviceId = instance.getServiceId();

            //IP地址
            String host = instance.getHost();
```

```
            //端口号
            int port = instance.getPort();

            //uri
            URI uri = instance.getUri();
        }
    }

}
```

8 集群配置

1.Eureka集群各个节点配置文件

7001：

```
server:
  port: 7001

eureka:
  instance:
    hostname: eureka7001.com #eureka服务端的实例名称
  client:
    register-with-eureka: false     #false表示不向注册中心注册自己。
    fetch-registry: false     #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
    service-url:
      #单机 defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/       #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址（单机）。
      defaultZone: http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

7002:

```
server:
  port: 7002

eureka:
  instance:
    hostname: eureka7002.com #eureka服务端的实例名称
  client:
    register-with-eureka: false     #false表示不向注册中心注册自己。
    fetch-registry: false     #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
    service-url:
      #defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/       #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7003.com:7003/eureka/
```

7003：

```yaml
server:
  port: 7003

eureka:
  instance:
    hostname: eureka7003.com #eureka服务端的实例名称
  client:
    register-with-eureka: false     #false表示不向注册中心注册自己。
    fetch-registry: false     #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
    service-url:
      #defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/       #设置与Eureka Server交互的地址查询服务和注册服务都需要依赖这个地址。
      defaultZone: http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/
```

客户端配置文件，需要些三个节点的地址：

```yaml
server:
  port: 80


eureka:
  client:
    register-with-eureka: false
    service-url:
      defaultZone:
http://eureka7001.com:7001/eureka/,http://eureka7002.com:7002/eureka/,http://eureka7003.com:7003/eureka/
```

9 Eureka与Zookeeper对比

CAP理论指出，一个分布式系统不可能同时满足C（一致性）、A（可用性）、P（分区容错性）。由于分区容错性P是在分布式系统中必须要保证的，因此我们只能在C与A之间权衡。

Zookeeper保证CP，Eureka保证AP。

9.1 Zookeeper保证CP

当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟之前的注册信息，但是不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求高于一致性。但是Zookeeper会出现这样的情况，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集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因为网络问题使得Zookeeper集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

9.2 Eureka保证AP

Eureka看明白了可用性大于一致性这一点，因此在设计就优先保证可用性。**Eureka各个节点都是平等的**，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册时如果发现连接失败，则会自动切换至其他节点，只要有一台Eureka还在，就能保证注册服务可用（保证可用性），只不过查到的信息可能不是最新的（不保证强一致性）。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：

1.Eureka不再从注册列表中移除因为长时间没有收到心跳而应该过期的服务

2.Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其他节点上（即保证当前节点依然可用）

3.当网络稳定时，当前实例新的注册信息会被同步到其他节点中

因此，Eureka可以很好地应对网络故障导致部分节点失去联系的情况，而不会像Zookeeper那样使整个注册服务瘫痪。

四、Ribbon

1 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。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。

LB，即负载均衡（Load Balancer），在微服务或分布式集群中经常用的一种应用。

负载均衡简单地说就是将用户的请求平摊分配到多个服务上，从而达到系统的HA（高可用）。常用的有软件Ngnix，LVS,硬件F5等。相应的在中间件，例如dubbo和SpringCloud中均给我们提供了负载均衡，SpringCloud的负载均衡算法可以自定义。

负载均衡分为集中式LB和进程内LB。

集中式LB：即在服务的消费方和提供方之间使用独立的LB设施（可以是硬件，如F5,也可以是软件，如ngnix），由该设施负责把访问请求通过某种策略转发至服务的提供方。

进程内LB：将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

2 Ribbon配置

1.给客户端添加依赖，注意客户端同样需要注册到eureka中

```
<!-- 添加Ribbon依赖 -->
<!-- 注意：老版本的SpringCloud依赖为spring-cloud-starter-ribbon -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>

<!-- Eureka客户端依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>

<!-- 使用注册中心还需要添加注册中心客户端所需依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>

...其他依赖省略
```

2.修改配置文件

```
server.port=80

eureka.client.register-with-eureka=false
eureka.client.service-url.defaultZone=http://localhost:7001/eureka/
```

3.修改主配置类，给RestTemplate加上@LoadBalanced注解

```
@SpringBootApplication
@EnableEurekaClient
public class ConsumerApplication {

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

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

Ribbon和Eureka整合后，客户端可以直接通过服务名调用服务，而不用再关注地址和端口号。

总结：Ribbon其实就是一个软负载均衡的客户端组件，他可以和其他所需请求的客户端结合使用，和eureka结合只是其中的一个实例。

3. 核心组件IRule

IRule：根据特定算法从服务列表中选取一个要访问的服务。

Ribbon在工作时分成两步：

1.先选择EurekaServer，它优先选择在同一区域内负载较少的server

2.在根据用户指定的策略，再从server取到的服务注册列表中选择一个地址，Ribbon提供了如轮询、随机和根据响应时间加权等策略。

Ribbon提供的7中负载均衡算法：

RoundRobinRule	轮询
RandomRule	随机
AvailabilityFilteringRule	会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，还有并发的连接数量超过阈值的服务，然后对剩余的服务列表按照轮询的策略进行访问
WeightedResponseTimeRule	根据平均响应时间计算所有服务的权重，响应时间越快服务权重越大被选中的概率越高，刚启动是如果统计信息不足，则使用RoundRobinRule策略，等统计信息足够，会切换到WeightedResponseTimeRule
RetryRule	先按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定时间内进行重试，获取可用的服务。
BestAvailableRule	会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量小的服务。
ZoneAvoidanceRule	默认规则，复合判断server所在区域的性能和server的可用性选择服务器

Ribbon默认采用轮询算法，要修改为其他算法，只需要在配置类中使用@Bean注解添加对应的IRule实现类即可。

```
@configuration
public class ConfigBean{

    //使用 @LoadBalanced注解开启负载均衡
    @Bean
    @LoadBalanced
    RestTemplate restTemplate(){
        return new RestTemplate();
    }

    //将默认的轮询算法修改为随机算法

    @Bean
```

```
    public IRule myRule(){
        return new RandomRule();
    }
}
```

4. 自定义Ribbon

1.在主配置文件添加@RibbonClient注解，给指定的服务添加指定的负载均衡配置类

```
@SpringBootApplication
@EnableEurekaClient
//使用该注解，在启动该微服务的时候就能去加载我们自定义的Ribbon配置类，从而使配置生效
//指定访问microserver微服务时，采用myselfRule这个配置类
@RibbonClient(name="microserver",configuration = myselfRule.class)
public class ConsumerApplication {

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

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

注意：官方文档明确的给出了警告，自定义配置类不能放在@ComponentScan所扫描的包及其子包下，否则我们自定义的配置类就会被所有的Ribbon客户端所共享，也就是说达不到我们特殊化定制的目的。

2.自定义配置类（在主启动类包外另建一个包）

```
@configuration
public class myselfRule{

    //将默认的轮询算法修改为随机算法
    @Bean
    public IRule myRule(){
        //使用自定义的MyRule算法
        return new MyRule();
    }
}
```

3.随机算法源码，自定义算法可以模仿源码修改

```
/*
 *
 * 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 loadbalacing 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

    }
}
```

五、Feign

1.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也支持可插拔式的编码器和解码器。SpringCloud对Feign进行了封装，使其支持了SpringMVC标准注解和HttpMessageConverters。Feign可以与Eureka、Ribbon组合使用以支持负载均衡。

Ribbon虽然功能强大，甚至可以自定义算法，但是通过微服务名调用服务不符合社区习惯的面向接口编程的规范，还是习惯WebService接口的调用模式。不是通过微服务名字获取服务地址而是通过接口+注解的形式来调用服务。Feign就是在Ribbon的基础上进行了进一步的封装，只需要通过创建一个接口，然后在上面添加注解即可调用服务，这符合社区统一面向接口编程的规范。

Feign能干什么：

Feign旨在使编写Java Http客户端变得更加容易。在使用Ribbon+RestTemplate时，利用RestTemplate对http请求进行封装处理，形成了一套模板化的调用方法。但是实际开发中，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的封装，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他（类似于Dao接口上标注Mapper注解，在微服务接口上标注Feign注解），即可完成对服务提供方的接口绑定，简化了使用SpringCloudRibbon时，封装服务调用客户端的开发量。

参考：https://github.com/OpenFeign/feign

2. 使用步骤

1.新建一个SpringBoot应用，作为服务调用客户端

2.修改pom文件，添加Feign相关依赖

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

Finchley版本为：

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

3.修改主程序类，添加@EnableFeignClients注解启用FeignClient

```
@SpringBootApplication
//所有组件都必须是Eureka客户端
@EnableEurekaClient
```

```
//添加注解启用FeignClient
@EnableFeignClients
public class ConsumerApplication {

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

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

4.创建微服务接口

```
//@FeignClient注解针对名为micro-service-8081微服务的调用客户端接口
@FeignClient(value = "micro-service-8081")
public interface microClientService {

    //该方法通过get方法调用micro-service-8081微服务/discovery接口
    @RequestMapping(value = "/discovery",method = RequestMethod.GET)
    public void discovery();
}
```

5.在其他地方直接调用微服务service即可

```
@RestController
public class MicroClientController {

    @Autowired
    private MicroClientService service;

    @RequestMapping(value = "/discovery")
    public void discovery(){
       this.service.discovery();
    }
}
```

6.若需要要传参使用@PathVariable注解或@RequestParam注解

（1）使用@RequestParam注解

FeignClientService:

```
@FeignClient(value = "micro-service-8081")
public interface MicroClientService {

    @RequestMapping(value = "/discovery",method = RequestMethod.GET)
    public List<String> discovery(@RequestParam(value = "serviceId") String serviceId);
}
```

controller：

```
@Autowired
private MicroClientService microClientService;

@RequestMapping(value = "/discovery")
public List<String> discovery(@RequestParam(value = "serviceId") String serviceId){
  List<String> list = this.microClientService.discovery(serviceId);
  return list;
}
```

（2）使用@PathVariable注解

```
@FeignClient(value = "micro-service-8081")
public interface MicroClientService {

    @RequestMapping(value = "/discovery/{serviceId}",method = RequestMethod.GET)
    public List<String> discovery(@PathVariable("serviceId") String serviceId);
}
```

```
@Autowired
private MicroClientService microClientService;

@RequestMapping(value = "/discovery/{serviceId}")
public List<String> discovery(@PathVariable("serviceId") String serviceId){
  List<String> list = this.microClientService.discovery(serviceId);
  return list;
}
```

3. 总结

Feign集成了Ribbon。Feign利用Ribbon维护列表信息，通过轮询实现了客户端的负载均衡。与Ribbon不同的是，通过Feign只需要定义服务绑定接口，以声明式的方法优雅而简单的实现了服务的调用。

Feign就是对Ribbon和RestTempelate的封装。

六、 Hystrix

1. 概述

1.1 分布式系统的问题

在复杂的分布式系统中，应用程序可能有数十个依赖，每个依赖关系在某个时间不可避免会出现异常。

服务雪崩：

多个微服务之间，如果微服务A调用微服务B和微服务C，而微服务B和微服务C又要调用其他的微服务，这就是所谓的”扇出“。如果扇出链路上某个微服务响应超时或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，出现”雪崩效应“。

对高流量的应用来说，单一的后端依赖可能会导致所有服务器上所有资源在几秒钟之内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些问题都需要对故障和延迟进行隔离和管理，防止单个依赖关系的失败，导致整个应用程序或系统崩溃。

1.2 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出现调用失败，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、不必要的占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩、

2. 服务熔断

2.1 什么是服务熔断

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回“错误”的响应信息。当检测到该节点微服务调用响应正常后恢复调用链路。在SpringCloud框架里熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定的阈值，缺省是5秒内20次调用失败就是启动熔断机制。熔断机制的注解是@HystrixCommand。

2.2 添加Hystrix

1.修改pom文件

根据springcloud版本选择添加

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

或

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

2.修改Controller

```
    @RequestMapping(value = "/discovery")
    //@HystrixCommand注解，当服务调用出现异常后，调用fallbackMethod进行处理
    @HystrixCommand(fallbackMethod = "descoveryHystrix")
    public List<String> discovery(String serviceIds){
        //获取所有服务名列表
        List<String> servicesList = client.getServices();

        //根据服务名获取实例列表
        List<ServiceInstance> instances = client.getInstances(serviceIds);

        List<String> list = new ArrayList<>();

        for(ServiceInstance instance:instances){
            //获取服务ID
            String serviceId = instance.getServiceId();

            //IP地址
            String host = instance.getHost();

            //端口号
            int port = instance.getPort();

            //uri
            URI uri = instance.getUri();

            String service = "serviceId:" + serviceId +";host:" + host +";port:" + port +";uri:"
+ uri;
            list.add(service);
        }
        if(list.size() == 0){
            throw new RuntimeException("没有serviceId为" + serviceIds + "的微服务");
        }
        return list;
    }

    //fallbackMethod，当discovery方法调用出现异常后，会调用该方法进行处理
    //fallbackMethod与原方法的参数列表以及返回值类型必须一致
    public List<String> descoveryHystrix(String serviceIds){

        List<String> list = new ArrayList<>();
```

```
        list.add("没有serviceId为" + serviceIds + "的微服务");
        return list;
    }
```

3.在主配置类添加@EnableCircuitBreaker注解，开启服务熔断机制

```
@SpringBootApplication
//标注为Eureka客户端，本服务启动后，会自动注册进Eureka服务中
@EnableEurekaClient
@EnableDiscoveryClient
@EnableCircuitBreaker
//开启熔断机制
public class MicroService8081Application {

    public static void main(String[] args) {
        SpringApplication.run(MicroService8081Application.class, args);
    }
}
```

3. 服务降级

3.1 什么是服务降级

当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心交易正常运作或高效运作。服务降级处理是在客户端实现的，与服务端没有关系。

3.2 服务降级实现

1.在客户端创建一个FallbackFactory接口的实现类，实现create方法

```
@Component
public class MyFallbackFactory implements FallbackFactory<MicroClientService> {

    @Override
    public MicroClientService create(Throwable cause) {
        return new MicroClientService(){
            @Override
            public List<String> discovery(String serviceId) {
                List<String> list = new ArrayList<>();
                list.add("HystrixClient:没有serviceId为" + serviceId + "的微服务");
                return list;
            }
        };
    }
}
```

2.对FeignClient接口的@FeignClient注解添加Fallback属性

```
@FeignClient(value = "micro-service-8081",fallbackFactory = MyFallbackFactory.class)
public interface MicroClientService {

    @RequestMapping(value = "/discovery",method = RequestMethod.GET)
    public List<String> discovery(@RequestParam(value = "serviceIds") String serviceId);
}
```

3.修改配置文件，添加feign.hystrix.enabled=true属性开启服务降级

```
#feign.hystrix.enabled
feign:
  hystrix:
    enabled: true
```

4. 服务监控

除了隔离依赖服务的调用以外，Hystrix还提供了准实时的调用监控（HystrixDashboard），Hystrix会持续地记录所有通过Hystrix发起的请求的执行信息，并以统计报表和图形的形式展示给用户，包括每秒执行多少请求，多少成功，多少失败等。Netflix通过hystrix-metrics-event-stream项目实现了对以上指标的监控。SpringCloud也提供了HystrixDashboard的整合，对监控内容转化成可视化界面。

4.1 构建

1.添加pom依赖

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

2.在主配置类添加@EnableHystrixDashboard注解

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboradApplication {

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

3.被监控应用必须添加监控相关的依赖

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

4.被监控应用启动类添加@EnableCircuitBreaker注解，也可以是@EnableHystrix注解，@EnableHystrix包括了@EnableCircuitBreaker注解，使用该注解开启熔断器功能

5.在springboot2.x以后，使用hystrix-dashboard进行监控会出现404。

在 spring boot 升为 2.0 后，为了安全，默认 Actuator 只暴露了2个端点，heath 和 info。

需要在被监控的微服务的application.yml文件中加入如下配置：

```
management:
  endpoints:
    web:
      exposure:
        include: ["hystrix-stream"]
```

这样就可以了，但是`actuator/health`就无法访问了，所以还可以选择全部放开。

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

注意：

1.Hystrix Dashboard 是个独立的服务，不用注册到 Eureka server

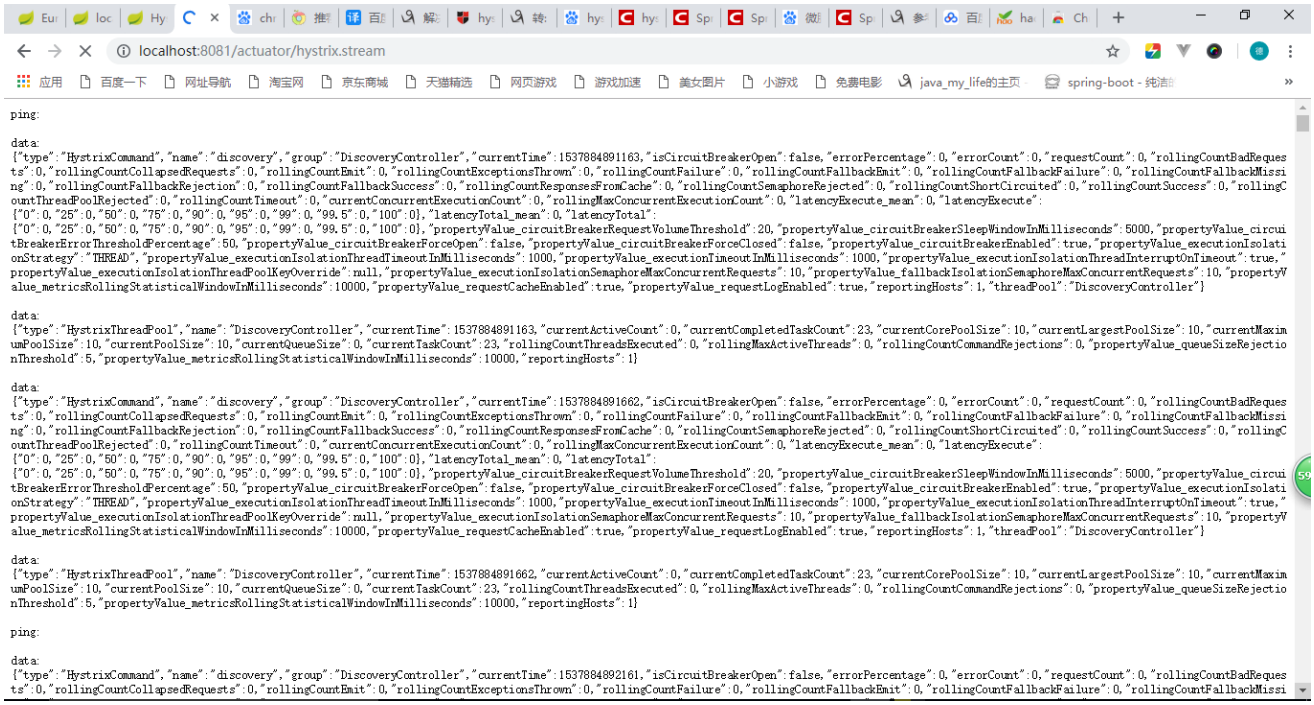
2.监控只对Hystrix接口有效，即用@HystrixCommand注解标注的接口

4.2 监控

1.监控地址：

Single Hystrix App: http://hystrix-app:port/actuator/hystrix.stream

直接访问监控地址会出现如下界面：

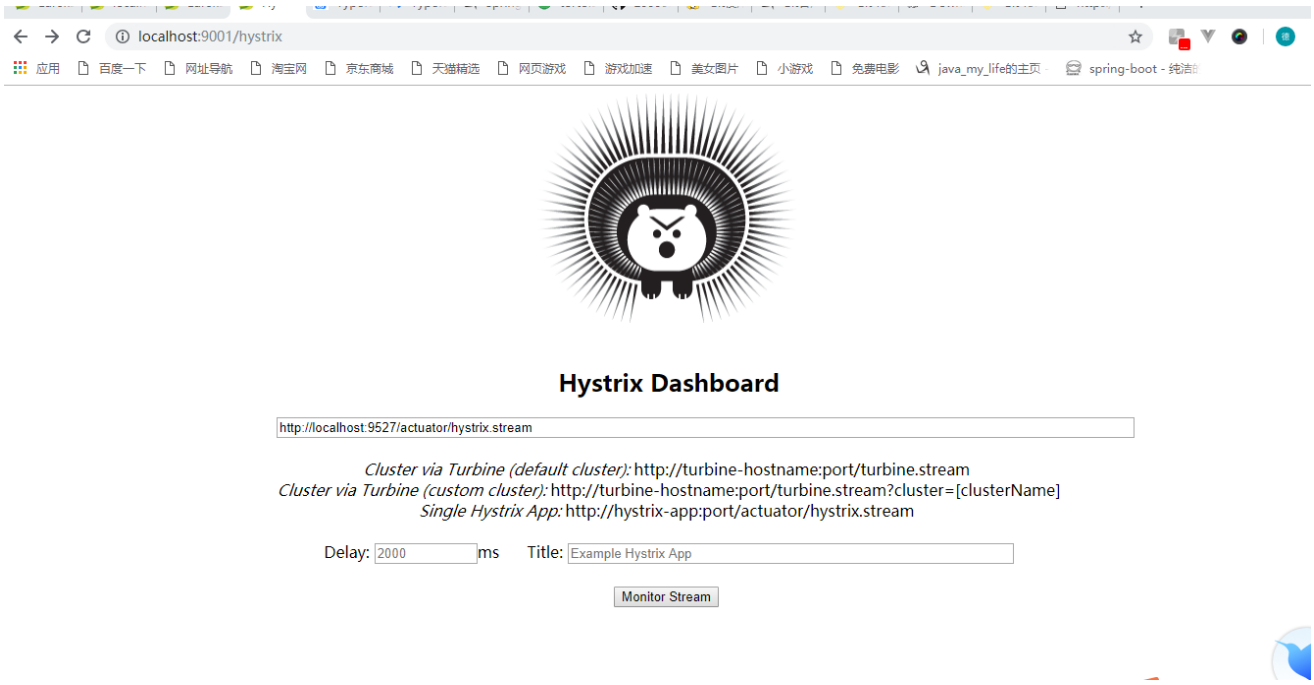


2.可视化页面

地址： http://{host}:{port}/hystrix

host、port：hystrix监控平台的地址。

出现如下页面：



填写监控地址和Delay（刷新时间）、Title（标题）。点击按钮进入监控页面

以上监控图表是每个接口对应一个图

4.3 集群监控

在实际应用中，我们要监控的应用往往是一个集群，这个时候我们就得采取Turbine集群监控了。Turbine有一个重要的功能就是汇聚监控信息，并将汇聚到的监控信息提供给Hystrix Dashboard来集中展示和监控。

4.3.1 搭建监控环境

1.创建一个名叫turbine的普通Spring Boot工程。

2.工程创建完成之后，我们需要添加一个依赖，如下：

```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Dalston.SR3</version>
    <relativePath/>
</parent>
<dependencies>
    <!-- 其他默认的依赖 -->
    <!-- 我们要添加的依赖 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-turbine</artifactId>
    </dependency>
</dependencies>
```

3.在启动类上添加@EnableTurbine注解表示开启Turbine

```
@SpringBootApplication
@EnableTurbine
public class TurbineApplication {

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

4.修改配置文件，加入eureka和turbine的相关配置

```
spring.application.name=turbine
server.port=2002
management.port=2003
eureka.client.service-url.defaultZone=http://localhost:1111/eureka/
turbine.app-config=ribbon-consumer
turbine.cluster-name-expression="default"
turbine.combine-host-port=true
```

关于这个配置文件:

```
1.turbine.app-config=ribbon-consumer指定了要监控的应用名字为ribbon-consumer，可为多个
2.turbine.cluster-name-expression="default",表示集群的名字为default
3.turbine.combine-host-port=true表示同一主机上的服务通过host和port的组合来进行区分，默认情况下是使用host来区分，这样会使本地调试有问题
```

5.被监控应用的相关配置与单应用监控一致

4.3.2 集群监控地址

集群监控地址：

（1）默认集群：

Cluster via Turbine (default cluster): http://turbine-hostname:port/turbine.stream

（2）自定义集群

Cluster via Turbine (custom cluster): http://turbine-hostname:port/turbine.stream?cluster=[clusterName]

七、zuul

1. 概述

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。过滤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，也即以后访问微服务都是通过Zuul跳转后获得的。

注意：Zuul服务最终还是会注册进Eureka。

Zuul提供的功能=代理+路由+过滤三大功能

2. 路由的基本配置

1.pom文件添加依赖

```
<!-- 由于zuul也需要注册进Eureka，所以需要添加EurekaClient相关的依赖 -->
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<!-- Zuul依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
```

2.修改application.yml文件

```
server:
  port: 9527

spring:
  application:
    name: zuul

eureka:
  client:
    service-url: #添加注册中心地址，将zuul注册进eureka服务列表内
      defaultZone: http://localhost:7001/eureka/
  instance:
     #服务标识符
    instance-id: zuul
    prefer-ip-address: true #鼠标悬停在标识符上显示IP
```

3.给启动类添加@EnableZuulProxy注解

```java
@SpringBootApplication
@EnableZuulProxy
public class ZuulApplication {

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

3. zuul访问映射规则

1.在配置文件下添加路由映射关系

```yaml
zuul:
  routes:
    microService.serviceId: micro-service
    microService.path: /service/**
```

添加完路由映射关系后：

原来通过zuul访问的url：http://localhost:9527/micro-service/demo

变成了http://localhost:9527/service/demo

这样做可以隐藏真实的服务名称，通过虚拟映射完成访问

2.虽然配置完成后可以通过虚拟路径访问，但是原来的真实路径依然有效，需要添加如下配置：

```yaml
zuul:
  routes:
    microService.serviceId: micro-service
    microService.path: /service/**
  ignored-services: micro-service
 #这样micro-service只能通过虚拟映射路径访问，用真实路径会出现404错误

#只需要屏蔽一部分服务的真实路径，可以直接写serviceId，如果要屏蔽所有就用"*"
zuul:
  routes:
    microService.serviceId: micro-service
    microService.path: /service/**
  ignored-services: "*"
```

3.使用zuul.prefix可以给所有路径添加统一前缀

```yaml
zuul:
  routes:
    microService.serviceId: micro-service
    microService.path: /service/**
  ignored-services: "*"
  prefix: /myservice
```

这样访问的url为：http://localhost:9527/myservice/service/**

八、SpringCloud Config

1.概述

1.1 微服务系统面临的配置问题

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务。如果有上百个微服务，那么就会对应上百个配置文件，由于每个服务都需要必要的配置信息才能运行，所以一套集中式的、动态的配置管理设施是必不可少的。SpringCloud提供了ConfigServer来解决这个问题。

1.2 是什么

SpringCloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

SpringCloud Config分为服务端和客户端两部分：

服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。配置服务器默认采用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。

1.3 能干什么

1.集中管理配置文件

2.不同环境不同配置，动态化的配置更新，分环境部署。比如dev/test/prod/beta/release

3.运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取自己的配置信息

4.当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置

5.将配置信息已REST接口的形式暴露

1.4 与GitHub整合配置

由于SpringCloud Config默认使用git来存储配置文件（也支持SVN或本地文件），但最推荐的还是git，而且使用的是http/https访问的形式。

2. 服务端配置

1.添加pom依赖：

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

2.修改配置文件

```
server:
  port: 3344

spring:
  application:
    name: micro-config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/z666666d/serviceConfig.git
         #git仓库地址
```

3.给主启动类添加@EnableConfigServer注解

```
@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {

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

4.配置好后，直接通过访问配置中心就可以看到配置文件内容

如：http://localhost:3344/application-dev.yml

没有设置profilename默认为dev，所以必须加-dev，否则会出现404错误

3. 配置读取规则

官方文档指出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
```

其中“应用程序”作为`SpringApplication`中的`spring.config.name`注入（即常规Spring Boot应用程序中通常为“应用程序”），“配置文件”是活动配置文件（或逗号分隔列表）的属性），“label”是可选的git标签（默认为“master”）。

4. 客户端配置

1.添加pom依赖

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

2.添加bootstrap.yml配置文件

application.yml是用户级的资源配置项。bootstrap.yml是系统级的，优先级最高。

SpringCloud会创建一个“Bootstrap Context”，作为spring应用的"Application Context"的父上下文。初始化的时候，"Bootstrap Context"负载从外部源加载配置属性并加载配置。这两个上下文共享一个从外部获取的”Environment“。”Bootstrap“属性有高优先级，默认情况下，他们不会被本地配置覆盖。”Bootstrap context“和”Application context“有着不同的约定，所以新增一个”bootstrap.yml“文件，保证”Bootstrap context“与”Application context“配置的分离。

```
spring:
  cloud:
    config:
      name: micro-service #需要从git上读取的资源名称，没有yml后缀
      profile: dev  #选择profile
      label: master
      uri: http://localhost:3344  #本微服务启动后先去找这个uri对应的springcloud config服务，获取github的服务地址
```