

SpringCloud

微服务框架对比

服务框架	Netflix/SpringCloud	Motan	gRPC	Thrift	Dubbo/Dubbox
功能定位	完整的微服务架构	RPC框架，但整合了ZK或Consul	RPC框架	RPC框架	服务框架
支持Rest	支持	No	No	No	No
支持RPC	SpringCloud不支持RPC，但是SpringCloud可以和Dubbo兼容。	No	No	No	No
支持多语言	是	No	是	是	No
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）	No	No	是（客户端）
配置服务	Netfix Archaius，Spring Cloud Config Server集中配置	是（zookeeper提供）	No	No	No
服务调用链监控	是（zuul），zuul提供边缘服务，API网关	No	No	No	No
高可用/容错	是（服务端Hystrix+客户端Ribbon）	是（客户端）	No	No	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	2017年后重新开始维护，之前中段了5年
学习难度	中	低	高	高	低
文档丰富程度	高	一般	一般	一般	高
其他	Spring Cloud Bus为我们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定义	实践的公司比较多

微服务的技术栈有哪些

微服务条目	落地技术
服务开发	SpringBoot, Spring, SpringMVC
服务配置与管理	Netflix公司的Archaius、阿里的Diamond等
服务注册与发现	Eureka、Consul、Zookeeper等
服务调用	Rest、RPC、gRPC（谷歌）
服务熔断器	Hystrix、Envoy等
负载均衡	Ribbon、Nginx等
服务接口调用（客户端调用服务的简化工具）	Feign等
消息队列	Kafka、RabbitMQ、ActiveMQ等
服务配置中心管理	SpringCloudConfig（GitHub远程配置）、Chef等
服务路由(API网关)	Zuul等
服务监控	Zabbix、Nagios、Metrics、Specatator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

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，断路器，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的并发风格做到一键启动和部署。

SpringBoot并没有重复造轮子，它只是将目前各家公司开发的比较成熟，经得起实际考验的服务框架组合起来，通过SpringBoot风格进行再封装，屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂，易部署和易维护的分布式系统开发工具包

SpringCloud是分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务全家桶。

SpringCloud和SpringBoot关系

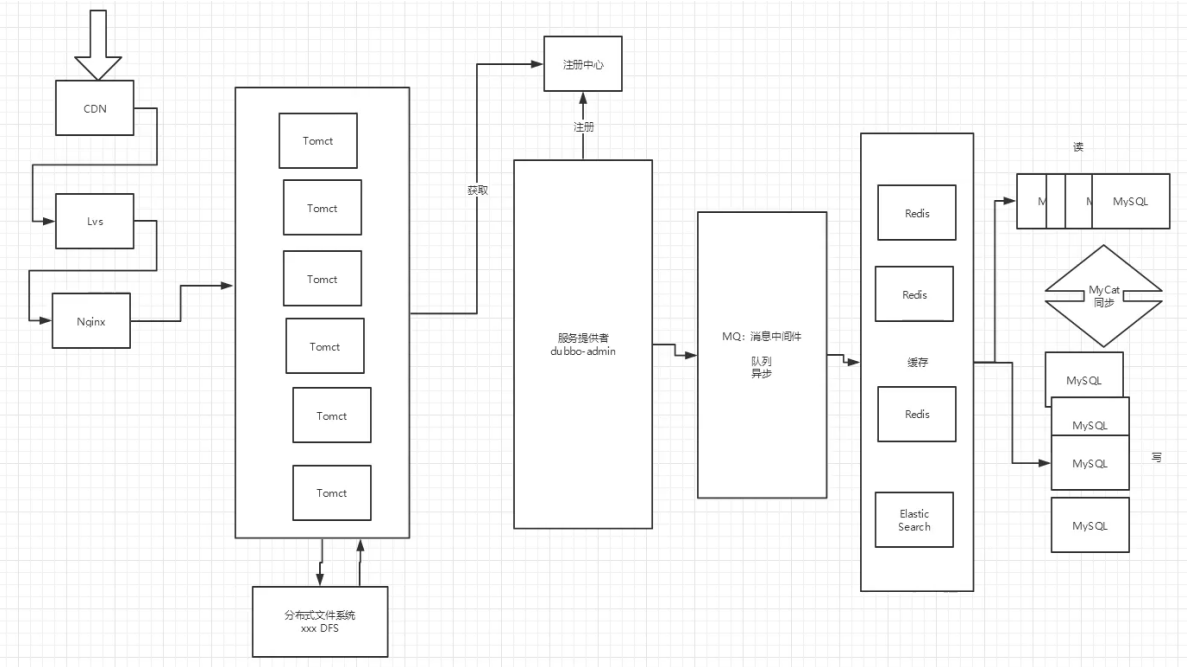
- SpringBoot专注于快速方便的开发单个体微服务。
- SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合管理起来，为各个微服务之间提供:配置管理，服务发现，断路器，路由，微代理，事件总线，全局锁，决策竞选,分布式会话等等集成服务。

- 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
- SpringBoot专注于快速、方便的开发单个个体微服务，SpringCloud关注全局的服务治理框架

Dubbo和SpringCloud技术选型

1、分布式+服务治理Dubbo

目前成熟的互联网架构：应用服务化拆分+消息中间件



2、Dubbo和SpringCloud对比

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

最大区别: 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，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖。这在强调快速演化的微服务环境下，显得更加合适。

品牌机与组装机区别

很明显，Spring Cloud的功能比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SpringFramework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。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，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题;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，但是如果要在非原装组件外的东西，就需要对其基础有足够的了解。

社区支持与更新力度

最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

总结:

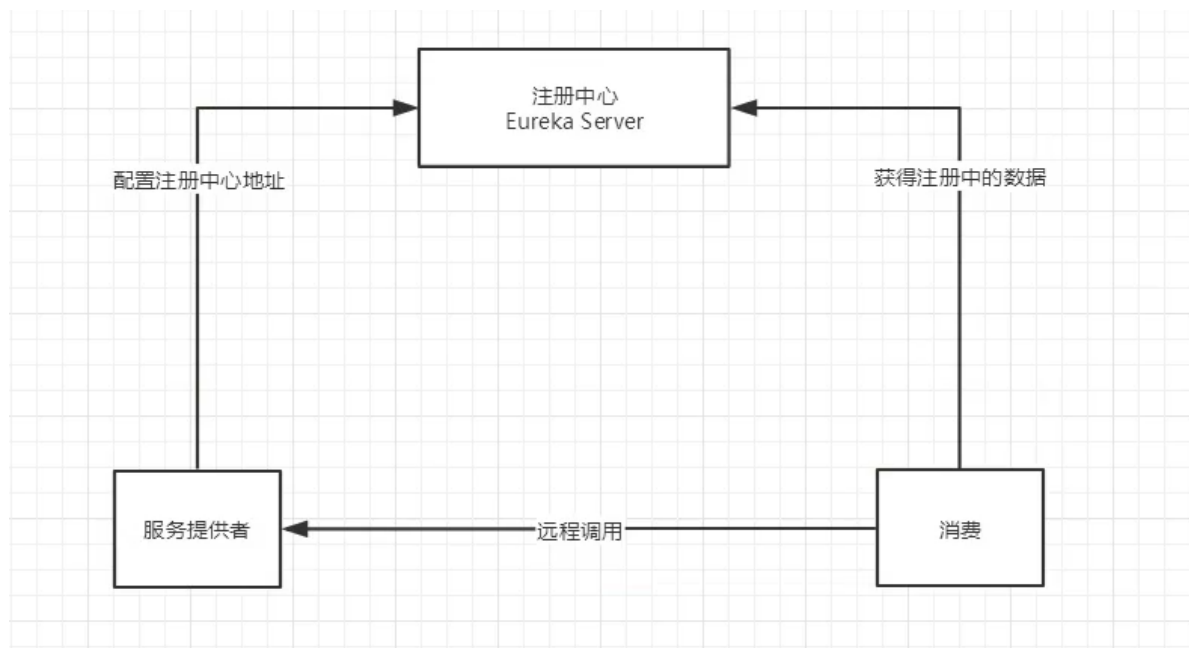
曾风靡国内的开源RPC服务框架Dubbo在重启维护后，令许多用户为之雀跃，但同时，也迎来了一些质疑的声音。互联网技术发展迅速，Dubbo是否还能跟上时代? Dubbo与Spring Cloud相比又有何优势和差异?是否会有相关举措保证 Dubbo的后续更新频率?

解决的问题域不一样：Dubbo的定位是一款RPC框架，Spring Cloud的目标是微服务框架的下一站式解决方案

Eureka服务注册与发现

什么是Eureka

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;



原理

- SpringCloud封装了Netflix公司开发的Eureka模块来实现服务注册和发现（对比Zookeeper）o Eureka采用了C-S的架构设计，EurekaServer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EurekaServer并维持心跳连接。这样系统的维护人员就可以通过EurekaServer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EurekaServer来发现系统中的其他微服务，并执行相关的逻辑;
- 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EurekaServer中进行注册，这样Eureka Server中的服务注册表中将会村粗所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。
- Eureka Client是一个Java客户端，用于简化EurekaServer的交互，客户端同时也具备一个内置的，使用轮询负载算法的负载均衡器。在应用启动后，将会向EurekaServer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，EurekaServer将会从服务注册表中把这个服务节点移除掉（默认周期为90秒）

三大角色

- Eureka Server：提供服务的注册与发现。
- Service Provider：将自身服务注册到Eureka中，从而使消费方能够找到。
- Service Consumer：服务消费方从Eureka中获取注册服务列表，从而找到消费服务。

Eureka服务：

导入eureka服务依赖：

```
<!-- 导入eureka服务器依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    <version>3.0.4</version>
</dependency>
```

编写配置文件：

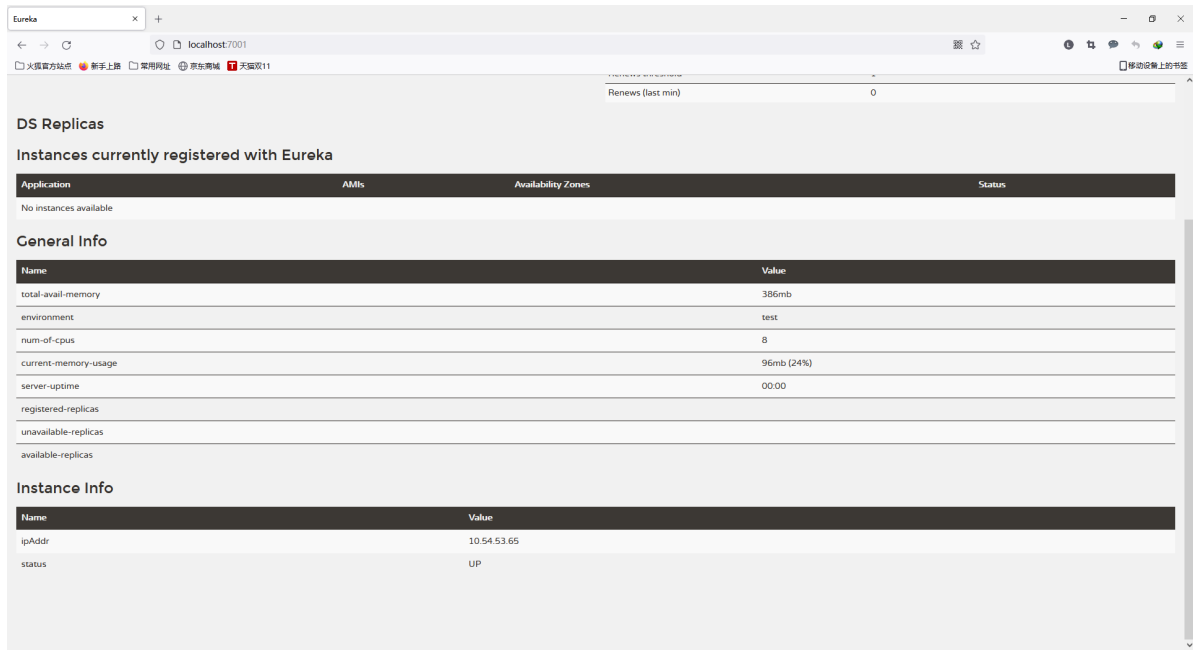
```
server:
    port: 7001

# Eureka配置
eureka:
    instance:
        hostname: localhost #eureka服务端实例
    client:
        fetch-registry: false #fetch-registry为false, 则表示自己是注册中心
        register-with-eureka: false #表示是否想eureka注册中心发现自己
        service-url: #重新设置默认监控页面
            defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

编写启动类：

```
@SpringBootApplication
@EnableEurekaServer //服务端的启动类，可以接收别人注册进来
public class EurekaServer_7001 {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServer_7001.class, args);
    }
}
```

启动: <http://localhost:7001/eureka/>



添加Eureka服务提供者:

在8001下导入依赖:

```
<!-- eureka依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>3.0.4</version>
</dependency>
```

编写配置:

```
# eureka配置，表示服务需要注册到哪里
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/
  instance:
    instance-id: sprincloud-eureka-provider-dept8001 # 修改默认描述
```

添加启动项:

```

@SpringBootApplication
@EnableEurekaClient //在服务启动后自动注册到eureka中!
public class DeptProvider_8001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptProvider_8001.class, args);
    }
}

```

The screenshot shows an IDE on the left with the `application.yml` file open. The `eureka:` section is highlighted, showing the `instance-id` set to `springcloud-eureka-provider-dept8001`. A red box highlights this line with the comment `# 修改默认描述`. The IDE's console shows the application starting successfully.

On the right, a web browser displays the Eureka dashboard at `localhost:7001`. The 'System Status' section shows the environment as 'test' and the data center as 'default'. A red warning message is visible: `RENEWALS ARE LESSER THAN THE THRESHOLD. THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.`

The 'Instances currently registered with Eureka' table shows one instance:

Application	ADMs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (2)	(2)	UP (2) - springcloud-eureka-provider-dept8001, DESKTOP-188H5G8:springcloud-provider-dept.8001

The 'General Info' section shows details like total-avail-memory (471mb), environment (test), num-of-cpus (8), and current-memory-usage (96mb (20%)).

Eureka自我保护机制：

- 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将会注销该实例（默认90秒）。但是当网络分区故障发生时，微服务与Eureka之间无法正常通行，以上行为可能变得非常危险了--因为微服务本身其实是健康的，此时本不应该注销这个服务。Eureka通过自我保护机制来解决这个问题--当EurekaServer节点在短时间内丢失过多客户端时(可能发生了网络分区故障)，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，不再删除服务注册表中的数据（也就是不会注销任何微服务）。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。它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。一句话:好死不如赖活着
- 综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮和稳定
- 在SpringCloud中，可以使用`eureka.server.enable-self-preservation = false`禁用自我保护模式【不推荐关闭自我保护机制】

服务发现，在多个联调下使用：

```

//注册进来的微服务，获取一些信息
@GetMapping("/dept/discovery")
public Object discovery() {
    //获取微服务列表的清单
    List<String> services = discoveryClient.getServices();
    System.out.println("discovery=>services:" + services);

    //得到一个具体的微服务信息，通过具体的微服务id:application-name
}

```

```

        List<ServiceInstance> instances =
discoveryClient.getInstances("SPRINGCLOUD-PROVIDER-DEPT");
        for (ServiceInstance instance : instances) {
            System.out.println(
                instance.getHost() + "\t" +
                instance.getPort() + "\t" +
                instance.getUri() + "\t" +
                instance.getServiceId() + "\t" +
                instance.getMetadata()
            );
        }
        return discoveryClient;
    }
}

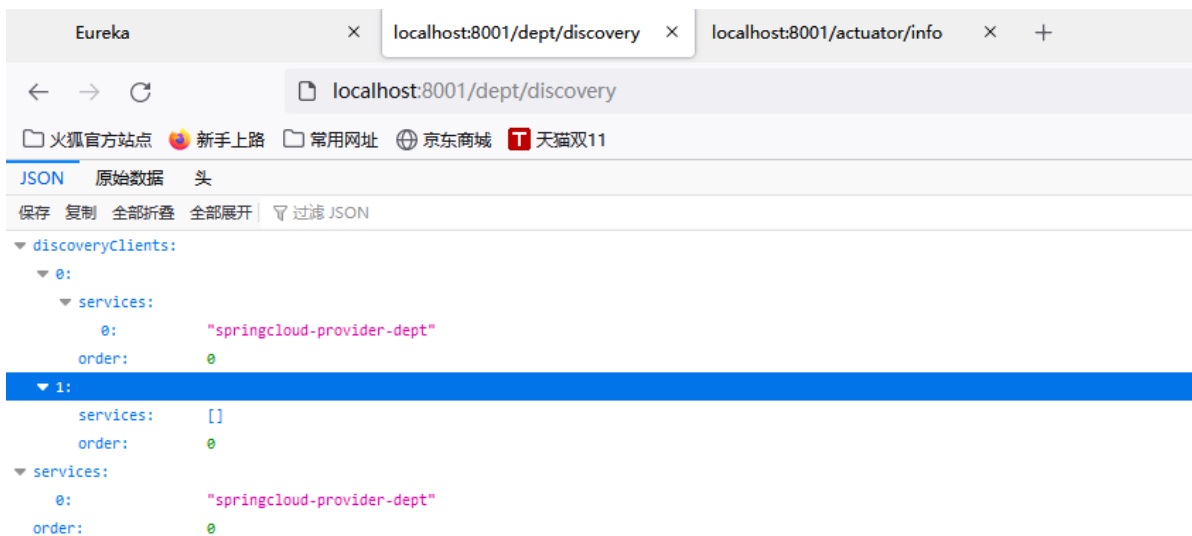
```

打开:

```

@SpringBootApplication
@EnableEurekaClient //在服务启动后自动注册到eureka中!
@EnableDiscoveryClient //注册服务发现
public class DeptProvider_8001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptProvider_8001.class,args);
    }
}

```



控制台:

```

discovery=>services:[springcloud-provider-dept]
DESKTOP-188H5G8 8001 http://DESKTOP-188H5G8:8001 SPRINGCLOUD-PROVIDER-DEPT {management.port=8001, jmx.port=65450}

```

简单集群搭建:

在项目基础上多创建两个端口分别为7002和7003的服务注册中心:

且配置中分别向另外两个注册中心配置: 如:

```

server:
    port: 7001

```



```
# Eureka配置
eureka:
  instance:
    hostname: localhost #eureka服务端实例
  client:
    fetch-registry: false #fetch-registry为false, 则表示自己是注册中心
    register-with-eureka: false #表示是否想eureka注册中心发现自己
    service-url: #重新设置默认监控页面
      # 单击配置 :
#    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
#    集群配置: 向另外两个服务注册中心中配置
    defaultZone: http://localhost:7002/eureka/,http://localhost:7003/eureka/
```

服务提供方需要向三个注册中心同时注册服务:

```
# eureka配置, 表示服务需要注册到哪里
eureka:
  client:
    service-url:
#    单机:
#    defaultZone: http://localhost:7001/eureka/
#    集群: 需要向每个集群中注册进去
    defaultZone:
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
  instance:
    instance-id: sprincloud-eureka-provider-dept8001 # 修改eureka默认描述信息
```

即可搭建简单的集群注册中心

CAP原则

- C(Consistency)强一致性
- A(Availability)可用性
- P(Partition tolerance)分区容错性

一个分布式系统不可能同时很好的满足一致性, 可用性和分区容错性这三个需求

根据CAP原理, 将NoSQL数据库分成了满足CA原则, 满足CP原则和满足AP原则三大类:

- CA:单点集群, 满足一致性, 可用性的系统, 通常可扩展性较差
- CP:满足一致性, 分区容错性的系统, 通常性能不是特别高
- AP:满足可用性, 分区容错性的系统, 通常可能对一致性要求低一些

作为服务注册中心, Eureka比zookeeper好在哪里?

著名的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(容错性)。由于分区容错性P在分布式系统中是必须要保证的, 因此我们只能在A和C之间进行权衡。

- Zookeeper保证的是CP;
- Eureka保证的是AP;

Ribbon

- 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的客户端组件提供一系列完整的配置项如:连接超时、重

试等等。简单的说，就是在配置文件中列出LoadBalancer(简称LB:负载均衡) 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!

负载均衡简单的说就是将用户的请求平摊的分配到多个服务上，从而达到系统的HA(高可用)。

常见的负载均衡软件有Nginx, Lvs等

服务消费者导入 `spring-cloud-starter-netflix-eureka-client` 即可不用导入ribbon依赖

配置eureka:

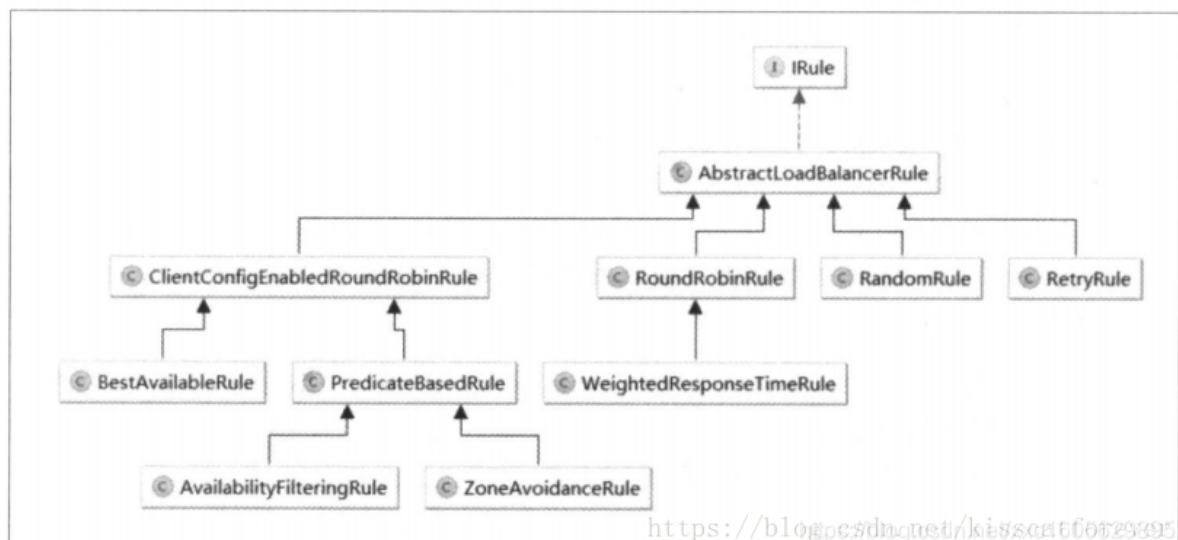
```
# eureka配置
eureka:
  client:
    register-with-eureka: false #表示不向eureka注册中心发现自己
    service-url:
      # 向集群中获取服务
      defaultZone:
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
```

controller:

```
//ribbon: 这里的地址应该是一个变量，通过服务名来访问，即SPRINGCLOUD-PROVIDER-DEPT
public static final String URL_PREFIX = "http://SPRINGCLOUD-PROVIDER-DEPT/dept";
```

@EnableEurekaClient //开启eureka客户端

自定义负载均衡策略



首先上面的这张图是Ribbon选择策略，我们使用的策略重点是最下面的6个子类：

策略名	策略声明	策略描述	实现说明
BestAvailableRule	public class BestAvailableRule extends ClientConfigEnabledRoundRobinRule	选择一个最小的并发请求的server	逐个考察Server，如果Server被tripped了，则忽略，在选择其中ActiveRequestsCount最小的server
AvailabilityFilteringRule	public class AvailabilityFilteringRule extends PredicateBasedRule	过滤掉那些因为一直连接失败的被标记为circuit	tripped的后端server，并过滤掉那些高并发的后端server（active connections 超过配置的阈值）使用一个AvailabilityPredicate来包含过滤server的逻辑，其实也就是检查status里记录的各个server的运行状态
WeightedResponseTimeRule	public class WeightedResponseTimeRule extends RoundRobinRule	根据响应时间分配一个weight，响应时间越长，weight越小，被选中的可能性越低。	一个后台线程定期的从status里面读取评价响应时间，为每个server计算一个weight。Weight的计算也比较简单responsetime 减去每个server自己平均的responsetime是server的权重。当刚开始运行，没有形成status时，使用roundRobin策略选择server。
RetryRule	public class RetryRule extends AbstractLoadBalancerRule	对选定的负载均衡策略机上重试机制。	在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server
RoundRobinRule	public class RoundRobinRule extends AbstractLoadBalancerRule	roundRobin方式轮询选择server	轮询index，选择index对应位置的server
RandomRule	public class RandomRule extends AbstractLoadBalancerRule	随机选择一个server	在index上随机，选择index对应位置的server
ZoneAvoidanceRule	public class ZoneAvoidanceRule extends PredicateBasedRule	复合判断server所在区域的性能和server的可用性选择server	使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。

注意： `FooConfiguration` 必须是 `@Configuration`，但请注意，它不在主应用程序上下文的 `@ComponentScan` 中，否则将由所有 `@RibbonClients` 共享。如果您使用 `@ComponentScan`（或 `@SpringBootApplication`），则需要采取措施避免包含（例如将其放在一个单独的，不重叠的包中，或者指定要在 `@ComponentScan`）。

编写自定义Ribbon规则配置类：

```
package com.youxin.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.concurrent.ThreadLocalRandom;

/**
 * @author youxin
 * @program springcloud
 * @description
 * @date 2021-11-11 16:25
 */

public class MyRandomRule extends AbstractLoadBalancerRule {

    /**
     * 每个服务访问5次则换下一个服务(总共3个服务)
     * <p>
     * total=0,默认=0,如果=5,指向下一个服务节点
     * index=0,默认=0,如果total=5,index+1
     */

    private int total = 0; //被调用的次数
    private int currentIndex = 0; //当前是谁在提供服务
    //@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();//获得当前活着的服务
    List<Server> allList = lb.getAllServers();//获取所有的服务
    int serverCount = allList.size();
    if (serverCount == 0) {

        /*
         * No servers. End regardless of pass, because subsequent passes
         * only get more restrictive.
         */

        return null;
    }
    //int index = chooseRandomInt(serverCount);//生成区间随机数
    //server = upList.get(index);//从或活着的服务中,随机获取一个
    //=====自定义代码=====
    if (total < 5) {
        server = upList.get(currentIndex);
        total++;
    } else {
        total = 0;
        currentIndex++;
        if (currentIndex >= upList.size()) {
            currentIndex = 0;
        }
        server = upList.get(currentIndex);//从活着的服务中,获取指定的服务来进行操作
    }
    //=====
    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;
}

protected int chooseRandomInt(int serverCount) {
    return ThreadLocalRandom.current().nextInt(serverCount);
}

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

```

```

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

```

注入spring容器中：

```

@Configuration
public class MyLoadBalancerRule {

    /**
     * IRule:
     * RoundRobinRule 轮询策略
     * RandomRule 随机策略
     * AvailabilityFilteringRule : 会先过滤掉，跳闸，访问故障的服务~，对剩下的进行轮询~
     * RetryRule : 会先按照轮询获取服务~，如果服务获取失败，则会在指定的时间内进行，重试
     */
    @Bean
    public IRule myRule() {
        return new MyRandomRule();
    }

}

```

启动类中添加Ribbon注解指定自定义配置类：

```

@SpringBootApplication
@EnableEurekaClient //开启eureka客户端
// 微服务在启动时就去加载自定义的Ribbon配置类
@RibbonClient(name = "springcloud-provider-dept",configuration =
MyLoadBalancerRule.class)
public class DeptConsumer_80 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumer_80.class, args);
    }
}

```

如果在使用自定义负载均衡时报错，大概率是因为版本问题。springcloud在2020.0.0之后，移除掉了netflix-ribbon 使用eureka-client中的loadbalancer，使用自定义负载均衡不使用IRule接口。官方文档上有写，可以写一个配置类，可以与启动类同级

```

public class CustomerLoadBalancerConfiguration {
    [ @Bean ] (https://space.bilibili.com/427090)
    ReactorLoadBalancerServiceInstance randomLoadBalancer(Environment
environment,

    LoadBalancerClientFactory loadBalancerClientFactory) {
        String name =
environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);

        return new RandomLoadBalancer(loadBalancerClientFactory
        .getLazyProvider(name, ServiceInstanceListSupplier.class),
        name);
    }
}

```

然后在注入resttemplate的类中，使用注解@LoadBalancerClient
@LoadBalancerClient(name = 这里写提供者名称, configuration =
CustomerLoadBalancerConfiguration.class)

Feign：负载均衡（基于服务端）

Feign简介

Feign是声明式Web Service客户端，它让微服务之间的调用变得更简单，类似controller调用service。SpringCloud集成了Ribbon和Eureka，可以使用Feign提供负载均衡的http客户端

只需要创建一个接口，然后添加注解即可~

Feign，主要是社区版，大家都习惯面向接口编程。这个是很多开发人员的规范。调用微服务访问两种方法

1. 微服务名字 【ribbon】
2. 接口和注解 【feign】

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，形成了一套模板化的调用方法。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个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，我们只需要创建一个接口并使用注解的方式来配置它（类似以前Dao接口上标注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。

Feign默认集成了Ribbon

- 利用Ribbon维护了MicroServiceCloud-Dept的服务列表信息，并且通过轮询实现了客户端的负载均衡，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。

使用：

1、创建springcloud-consumer-fdept-feign模块并拷贝springcloud-consumer-dept-80模块下的pom.xml, resource, 以及java代码到springcloud-consumer-feign模块，并添加feign依赖。

```

<!-- feign依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.2.5.RELEASE</version>
</dependency>

```

2、在springcloud-api中创建服务接口且在api中也引入feign依赖:

```

package com.youxin.springcloud.service;

import com.youxin.springcloud.pojo.Dept;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;

import java.util.List;

@Service
// @FeignClient:微服务客户端注解,value:指定微服务的名字,这样就可以使Feign客户端直接找到对应的微服务
@FeignClient(value = "springcloud-provider-dept")
public interface DeptClientService {

    //这里的GetMapping要跟服务提供者一样
    @RequestMapping("/dept/add")
    public boolean addDept(Dept dept);

    @GetMapping("/dept/findAll")
    public List<Dept> queryAll();

    @GetMapping("/dept/find/{id}")
    public Dept queryById(@PathVariable("id") Long dept_no);
}

```

3、修改FeignDept下的controller使其不再用rest风格访问,而是用接口形式访问,当然归根结底还是以rest风格访问:

```

package com.youxin.springcloud.controller;

import com.youxin.springcloud.pojo.Dept;
import com.youxin.springcloud.service.DeptClientService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.client.RestTemplate;

import java.util.List;

/**
 * @author youxin
 * @program springcloud
 * @description
 * @date 2021-11-09 15:51

```

```

*/
@RestController
public class DeptConsumerController {

    @Autowired
    private DeptClientService deptClientService;

    //url : http://localhost:8001/dept/find/1
    @GetMapping("/consumer/dept/get/{id}")
    public Dept findById(@PathVariable("id") Long dept_no) {
        return deptClientService.queryById(dept_no);
    }

    @GetMapping("/consumer/dept/getAll")
    public List<Dept> getAll() {
        return deptClientService.queryAll();
    }

    @RequestMapping("/consumer/dept/add")
    public boolean add(@RequestBody Dept dept) {
        return deptClientService.addDept(dept);
    }

}

```

4、不用修改配置，编写启动类：

```

package com.youxin.springcloud;

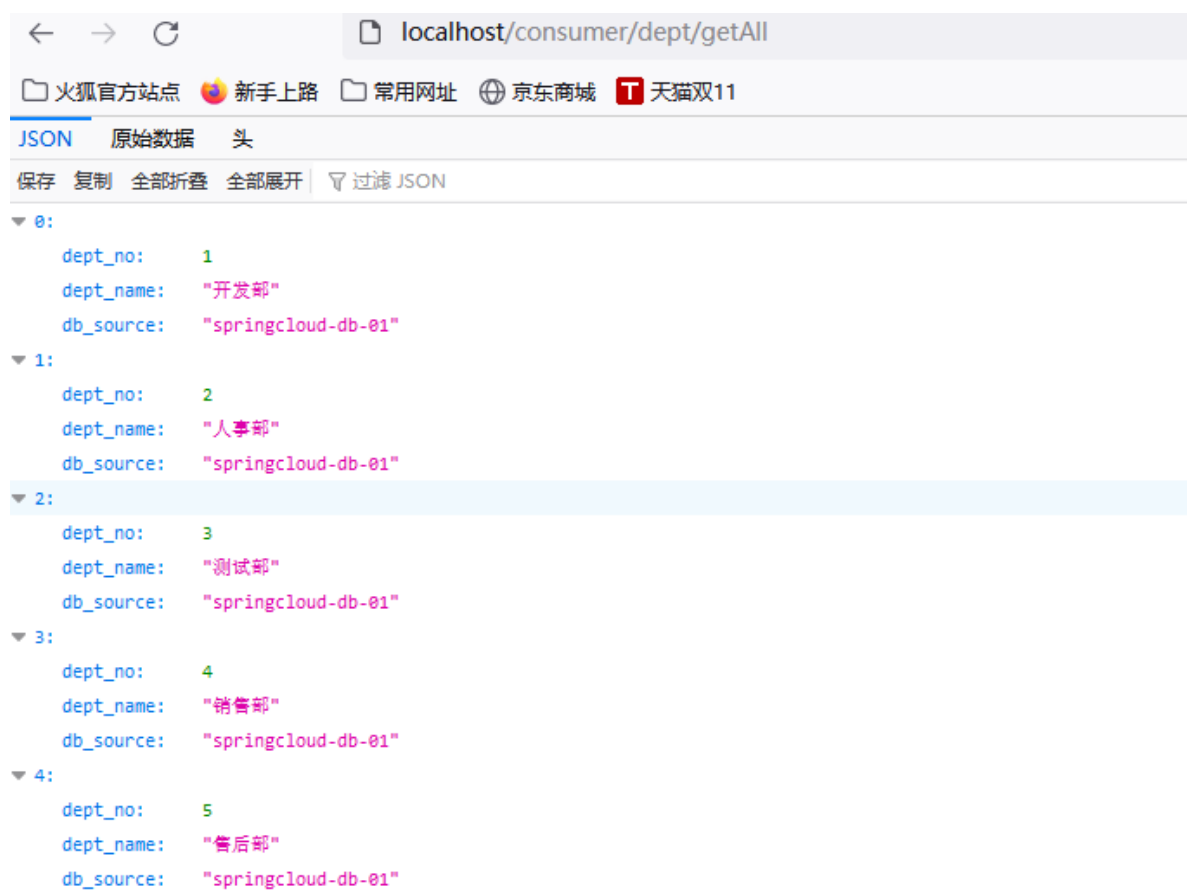
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.context.annotation.ComponentScan;

/**
 * @author youxin
 * @program springcloud
 * @description
 * @date 2021-11-09 16:13
 */
@SpringBootApplication
@EnableEurekaClient //开启eureka客户端
// feign客户端注解,并指定要扫描的包以及配置接口DeptClientService
@EnableFeignClients(basePackages = {"com.youxin.springcloud.service"})
// 切记不要加这个注解，不然会出现404访问不到
//@ComponentScan("com.youxin.springcloud")
public class FeignDeptConsumer_80 {
    public static void main(String[] args) {
        SpringApplication.run(FeignDeptConsumer_80.class, args);
    }
}

```


Feign和Ribbon二者对比，前者显现出面向接口编程特点，代码看起来更清爽，而且Feign调用方式更符合我们之前在做SSM或者SpringBoot项目时，Controller层调用Service层的编程习惯！

feign做负载均衡结果和ribbon一样（默认还是轮询算法调用服务提供者）：



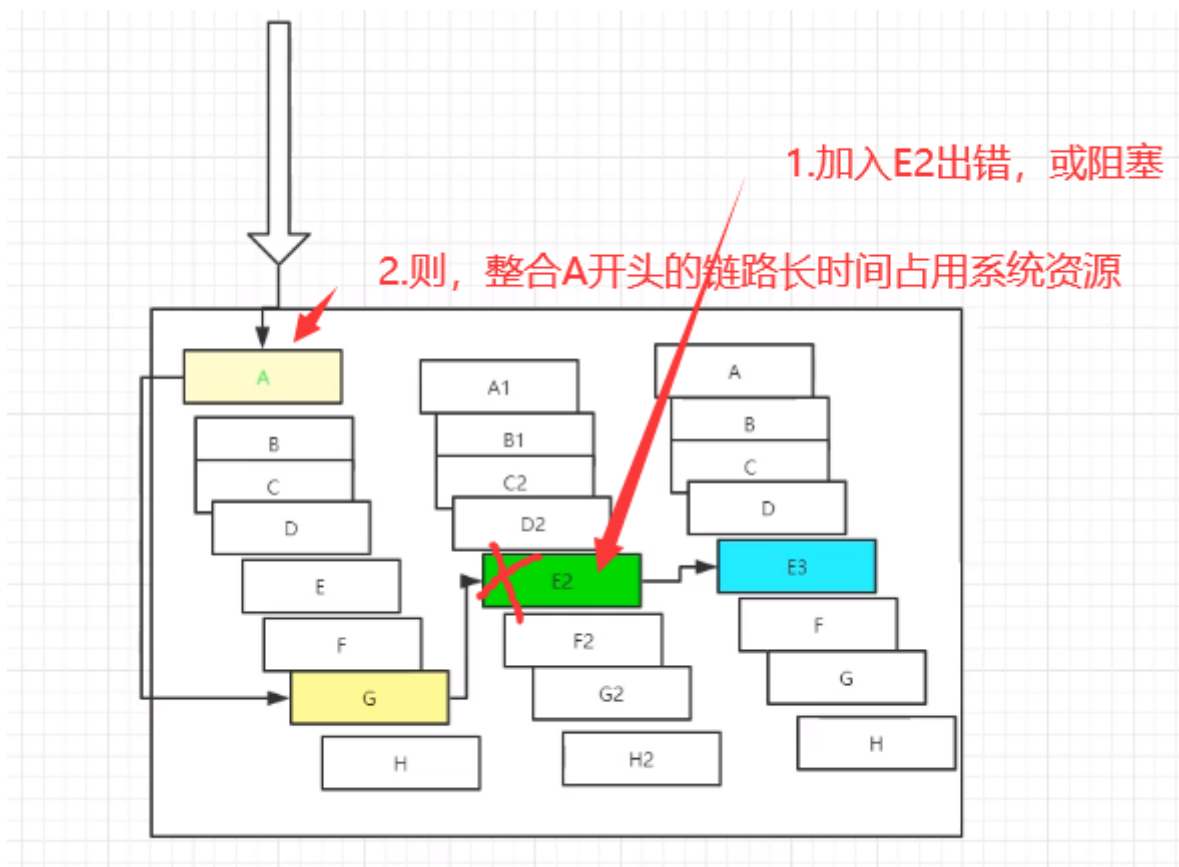
Hystrix（服务熔断）

分布式系统面临的问题

复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免失败！

服务雪崩

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其他的微服务，这就是所谓的“扇出”，如果扇出的链路上某个微服务的调用响应时间过长，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，所谓的“雪崩效应”。



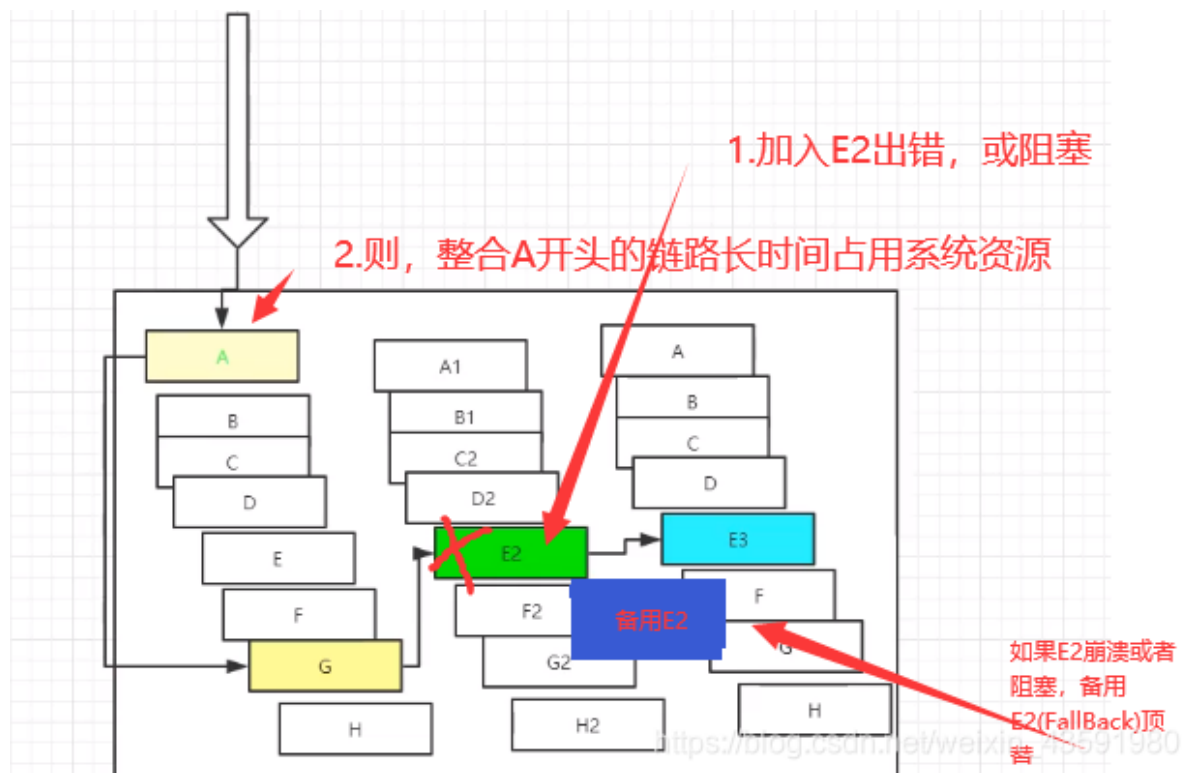
对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几十秒内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障，**这些都表示需要对故障和延迟进行隔离和管理，以达到单个依赖关系的失败而不影响整个应用程序或系统运行。**

我们需要，弃车保帅！

什么是Hystrix?

Hystrix是一个应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，不会导致整个体系服务失败，避免级联故障，以提高分布式系统的弹性。

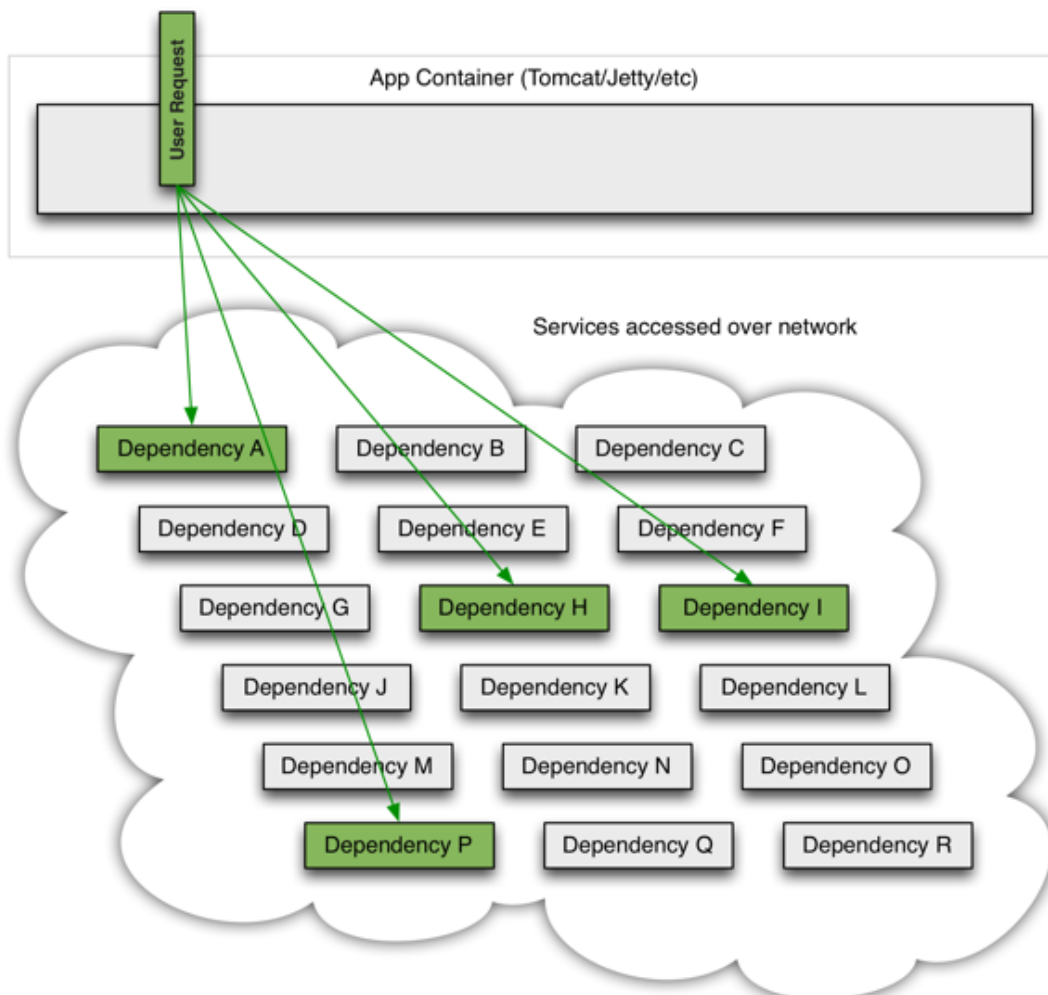
“**断路器**”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控 (类似熔断保险丝)，向调用方返回一个服务预期的，可处理的备选响应 (FallBack)，而不是长时间的等待或者抛出调用方法无法处理的异常，这样就可以保证了服务调用方的线程不会被长时间，不必要的占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。



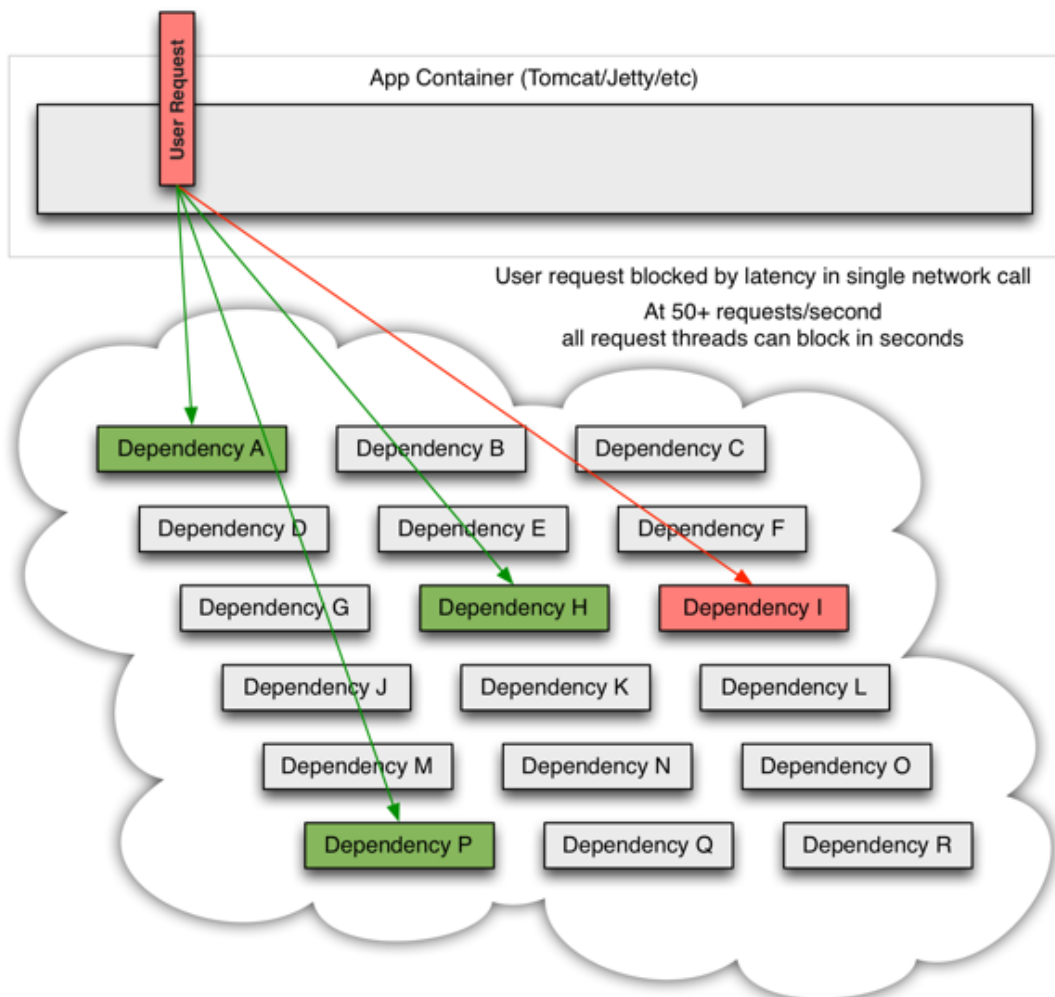
Hystrix能干嘛?

- 服务降级
- 服务熔断
- 服务限流
- 接近实时的监控
- ...

当一切正常时, 请求流可以如下所示:

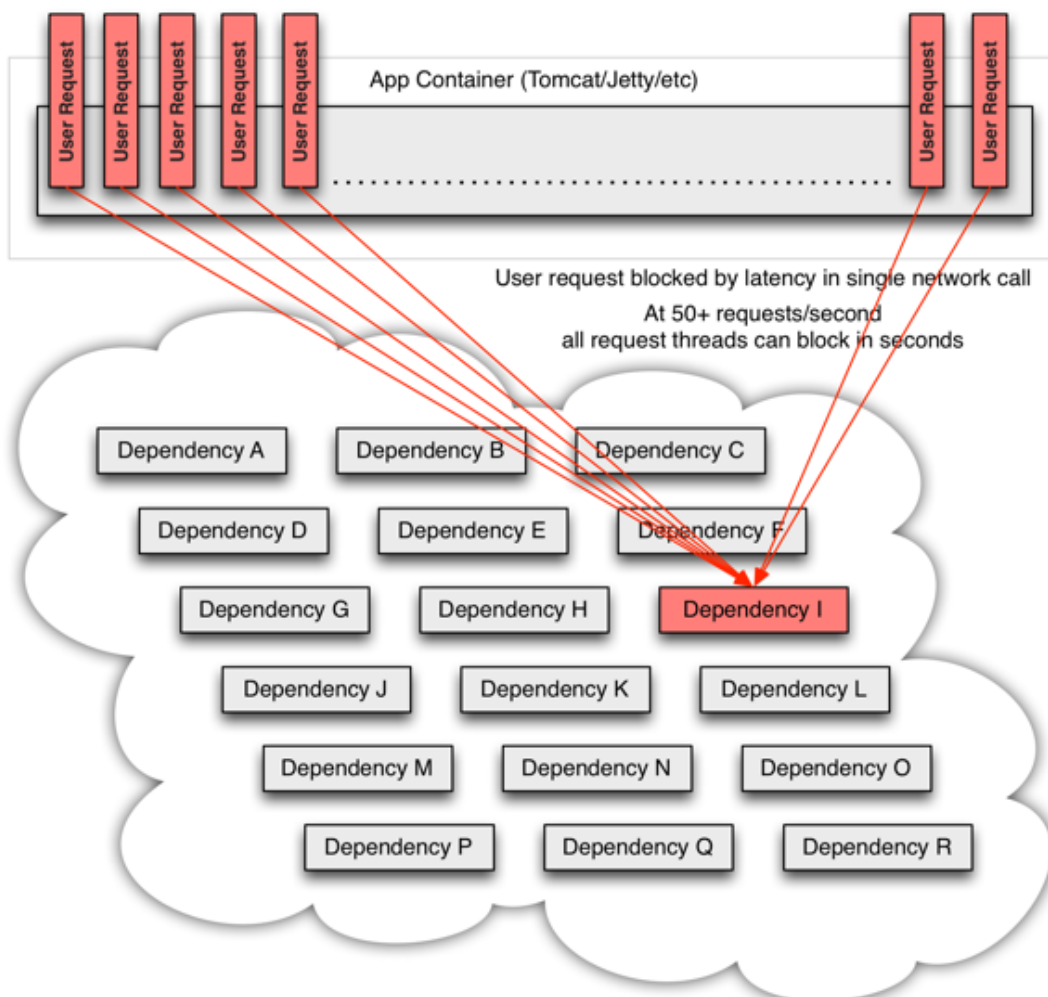


当许多后端系统中有一个潜在阻塞服务时，它可以阻止整个用户请求：

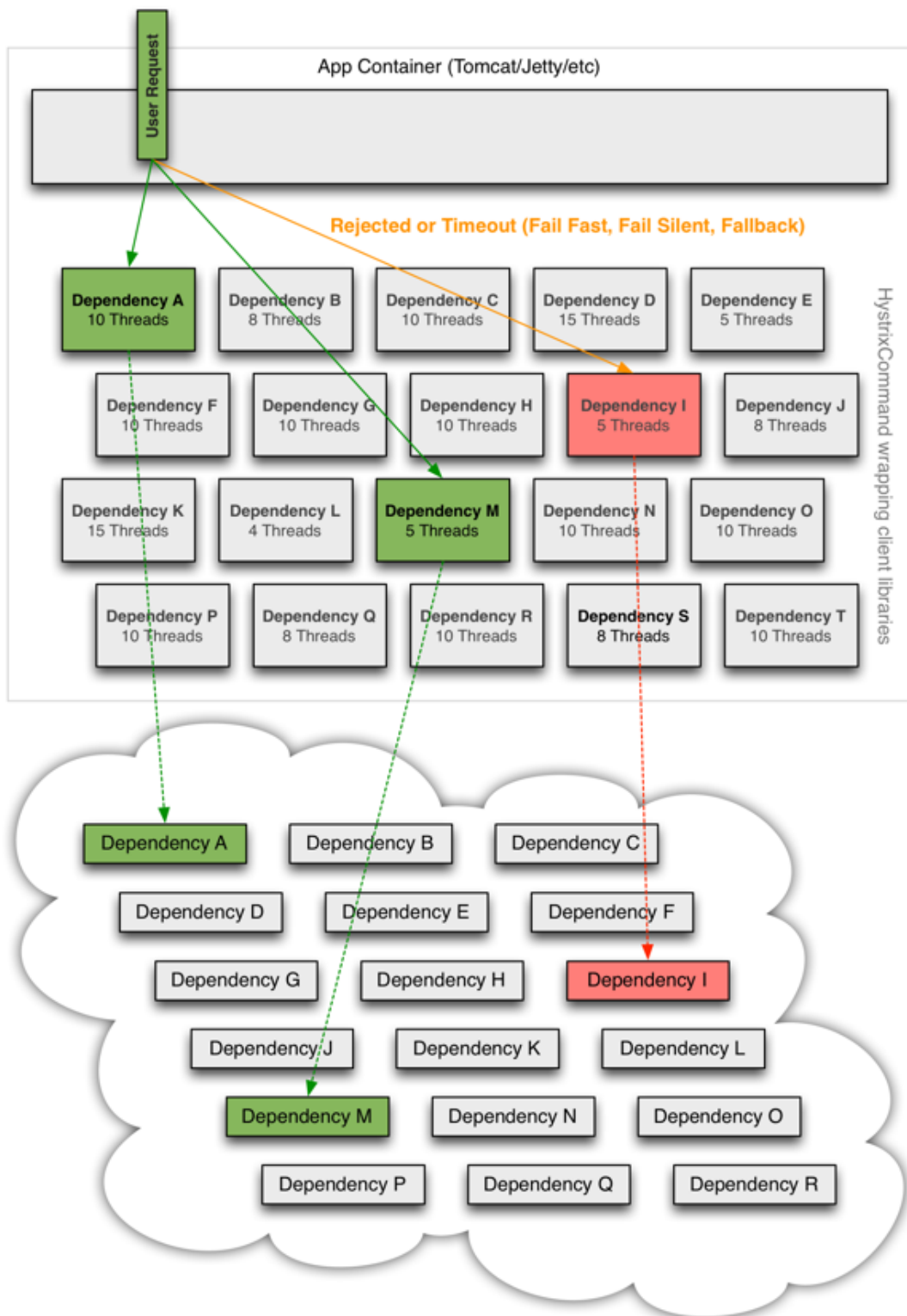


随着大容量通信量的增加，单个后端依赖项的潜在性会导致所有服务器上的所有资源在几秒钟内饱和。

应用程序中通过网络或客户端库可能导致网络请求的每个点都是潜在故障的来源。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，从而备份队列、线程和其他系统资源，从而导致更多跨系统的级联故障。



当使用**Hystrix**包装每个基础依赖项时，上面的图表中所示的体系结构会发生类似于以下关系图的变化。**每个依赖项是相互隔离的**，限制在延迟发生时它可以填充的资源中，并包含在回退逻辑中，该逻辑决定在依赖项中发生任何类型的故障时要做出什么样的响应：



https://blog.csdn.net/weixin_43591980

官网资料: <https://github.com/Netflix/Hystrix/wiki>

服务熔断

什么是服务熔断?

熔断机制是赌赢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。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阈值缺省是5秒内20次调用失败, 就会启动熔断机制。熔断机制的注解是: `@HystrixCommand`。

服务熔断解决如下问题：

- 当所依赖的对象不稳定时，能够起到快速失败的目的；
- 快速失败后，能够根据一定的算法动态试探所依赖对象是否恢复。

案例：

导入hystrix依赖：

```
<!-- 熔断机制hystrix依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>2.2.9.RELEASE</version>
</dependency>
```

编写接口：

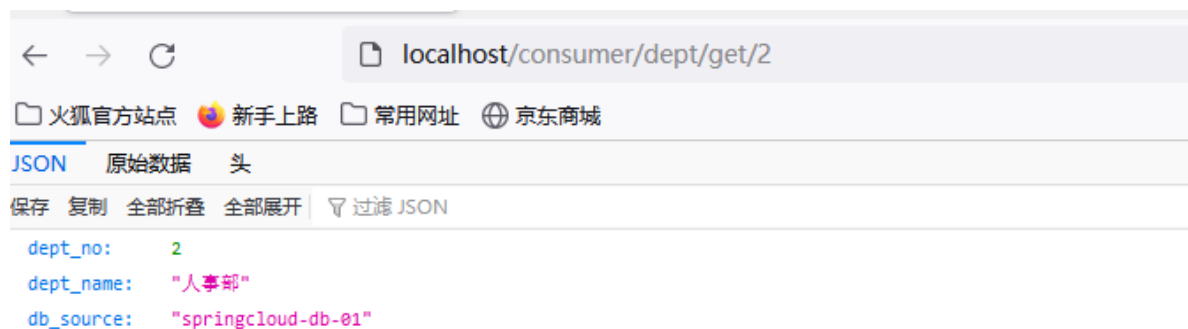
```
@HystrixCommand(fallbackMethod="queryById_hystrix") //如果根据id查询出现异常,则走
hystrixGet这段备选代码
@GetMapping("/dept/find/{id}")
public Dept queryById(@PathVariable("id") Long dept_no) {
    Dept dept = deptService.queryById(dept_no);
    if (dept == null) {
        throw new RuntimeException("id=>" + dept_no + "不存在");
    }
    return dept;
}

/**
 * @author youxin
 * @date 2021-11-12 17:24
 * @param dept_no 根据dept_no查询备用方案（熔断机制）
 * @return com.youxin.springcloud.pojo.Dept
 * @throws
 * @since
 */
public Dept queryById_hystrix(@PathVariable("id") Long dept_no) {
    return new Dept().setDept_no(dept_no)
        .setDept_name("id=>" + dept_no + ",数据库中没有对应的部门信息!")
        .setDb_source("没有对应数据库");
}
```

开启hystrix @EnableCircuitBreaker：

```
@SpringBootApplication
@EnableEurekaClient //在服务启动后自动注册到eureka中!
@EnableDiscoveryClient //注册服务发现
@EnableCircuitBreaker // 添加对熔断的支持注解
public class DeptProviderHystrix_8001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptProviderHystrix_8001.class,args);
    }
}
```


当获取正确请求时：



当请求错误时：



发现并没有抛出异常，而是走了我们自定义的熔断机制方法。

服务降级

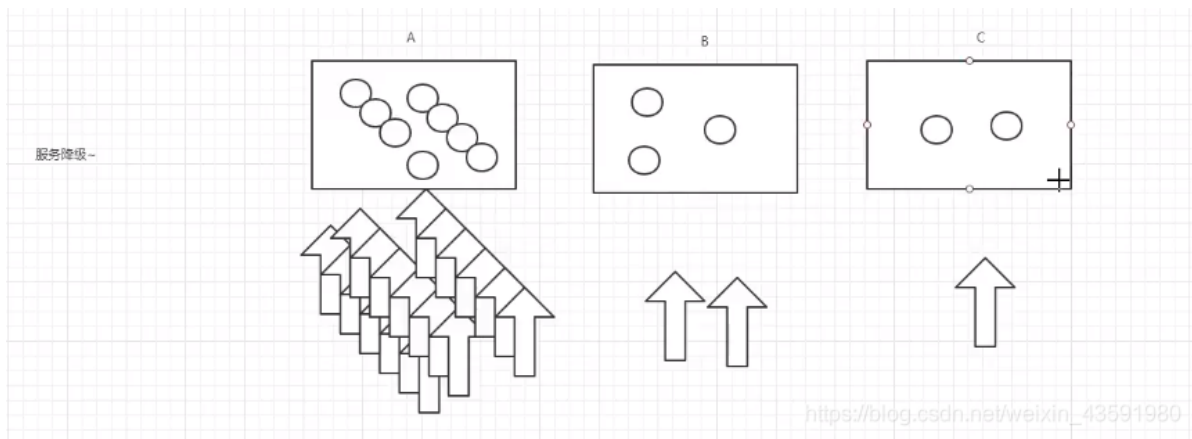
什么是服务降级？

服务降级是指 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业务正常运作或高效运作。说白了，**就是尽可能的把系统资源让给优先级高的服务。**

资源有限，而请求是无限的。如果在并发高峰期，不做服务降级处理，一方面肯定会影响整体服务的性能，严重的话可能会导致宕机某些重要的服务不可用。所以，一般在高峰期，为了保证核心功能服务的可用性，都要对某些服务降级处理。比如当双11活动时，把交易无关的服务统统降级，如查看蚂蚁森林，查看历史订单等等。

服务降级主要用于什么场景呢？当整个微服务架构整体的负载超出了预设的上限阈值或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，可以将一些 不重要 或 不紧急 的服务或任务进行服务的 延迟使用 或 暂停使用。

降级的方式可以根据业务来，可以延迟服务，比如延迟给用户增加积分，只是放到一个缓存中，等服务平稳之后再执行；或者在粒度范围内关闭服务，比如关闭相关文章的推荐。



由上图可得，当某一时间内服务A的访问量暴增，而B和C的访问量较少，为了缓解A服务的压力，这时候需要B和C暂时关闭一些服务功能，去承担A的部分服务，从而为A分担压力，叫做服务降级。

服务降级需要考虑的问题

- 1) 那些服务是核心服务，哪些服务是非核心服务
- 2) 那些服务可以支持降级，那些服务不能支持降级，降级策略是什么
- 3) 除服务降级之外是否存在更复杂的业务放通场景，策略是什么？

自动降级分类

- 1) 超时降级：主要配置好超时时间和超时重试次数和机制，并使用异步机制探测回复情况
- 2) 失败次数降级：主要是一些不稳定的api，当失败调用次数达到一定阈值自动降级，同样要使用异步机制探测回复情况
- 3) 故障降级：比如要调用的远程服务挂掉了（网络故障、DNS故障、http服务返回错误的状态码、rpc服务抛出异常），则可以直接降级。降级后的处理方案有：默认值（比如库存服务挂了，返回默认现货）、兜底数据（比如广告挂了，返回提前准备好的一些静态页面）、缓存（之前暂存的一些缓存数据）
- 4) 限流降级：秒杀或者抢购一些限购商品时，此时可能会因为访问量太大而导致系统崩溃，此时会使用限流来进行限制访问量，当达到限流阈值，后续请求会被降级；降级后的处理方案可以是：排队页面（将用户引导到排队页面等一会重试）、无货（直接告知用户没货了）、错误页（如活动太火爆了，稍后重试）。

案例

api中添加降级服务配置类：

```
package com.youxin.springcloud.service;

import com.youxin.springcloud.pojo.Dept;
import feign.hystrix.FallbackFactory;
import org.springframework.stereotype.Component;

import java.util.List;

/**
 * @author youxin
 * @program springcloud
 * @description Hystrix服务降级
 * @date 2021-11-12 18:25
 */
@Component
public class DeptClientFallbackService implements FallbackFactory {
```

```

    public DeptClientService create(Throwable throwable) {
        return new DeptClientService() {
            public boolean addDept(Dept dept) {
                return false;
            }

            public List<Dept> queryAll() {
                return null;
            }

            public Dept queryById(Long dept_no) {
                return new Dept()
                    .setDept_no(dept_no)
                    .setDept_name("id=>" + dept_no + ",由于客户端提供了服务降级,
所以现服务已被关闭")
                    .setDb_source("no data searched!");
            }
        };
    }
}

```

在DeptClientService中指定降级配置类DeptClientFallBackFactoryService

```

@Service
// @FeignClient:微服务客户端注解,value:指定微服务的名字,这样就可以使Feign客户端直接找到对应的微服务
@FeignClient(value = "springcloud-provider-dept",fallbackFactory =
DeptClientFallbackService.class) //fallbackFactory指定降级配置类
public interface DeptClientService {

    //这里的GetMapping要跟服务提供者一样
    @PostMapping("/dept/add")
    public boolean addDept(Dept dept);

    @GetMapping("/dept/findAll")
    public List<Dept> queryAll();

    @GetMapping("/dept/find/{id}")
    public Dept queryById(@PathVariable("id") Long dept_no);
}

```

80客户端开启服务降级配置:

```

server:
  port: 80

# eureka配置
eureka:
  client:
    register-with-eureka: false #表示不向eureka注册中心发现自己
    service-url:
      # 向集群中获取服务
      defaultZone:
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/

```

```
# 在客户端中开启降级
feign:
  hystrix:
    enabled: true
```

关闭服务提供端8001:



服务熔断和降级的区别

- **服务熔断—>服务端**: 某个服务超时或异常, 引起熔断~, 类似于保险丝(自我熔断)
- **服务降级—>客户端**: 从整体网站请求负载考虑, 当某个服务熔断或者关闭之后, 服务将不再被调用, 此时在客户端, 我们可以准备一个 FallBackFactory, 返回一个默认的值(缺省值)。会导致整体的服务下降, 但是好歹能用, 比直接挂掉强。
- 触发原因不太一样, 服务熔断一般是某个服务(下游服务)故障引起, 而服务降级一般是从整体负荷考虑; 管理目标的层次不太一样, 熔断其实是一个框架级的处理, 每个微服务都需要(无层级之分), 而降级一般需要对业务有层级之分(比如降级一般是从最外围服务开始)
- 实现方式不太一样, 服务降级具有代码侵入性(由控制器完成/或自动降级), 熔断一般称为**自我熔断**。

熔断, 降级, 限流:

限流: 限制并发的请求访问量, 超过阈值则拒绝;

降级: 服务分优先级, 牺牲非核心服务(不可用), 保证核心服务稳定; 从整体负荷考虑;

熔断: 依赖的下游服务故障触发熔断, 避免引发本系统崩溃; 系统自动执行和恢复

Dashboard 流监控

新建springcloud-consumer-hystrix-dashboard模块

引入80端口依赖并添加dashboard依赖:

```
<!-- 监控页面依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard</artifactId>
  <version>2.2.9.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
  <version>2.2.9.RELEASE</version>
</dependency>
```

主启动类:

```
@SpringBootApplication
// 开启Dashboard监控
@EnableHystrixDashboard
public class DeptConsumerDashboard_9001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptConsumerDashboard_9001.class, args);
    }
}
```

配置文件yaml:

```
server:
  port: 9001

hystrix:
  dashboard:
    proxy-stream-allow-list: localhost

eureka:
  client:
    # 关闭注册自己
    register-with-eureka: false
    fetch-registry: false
```

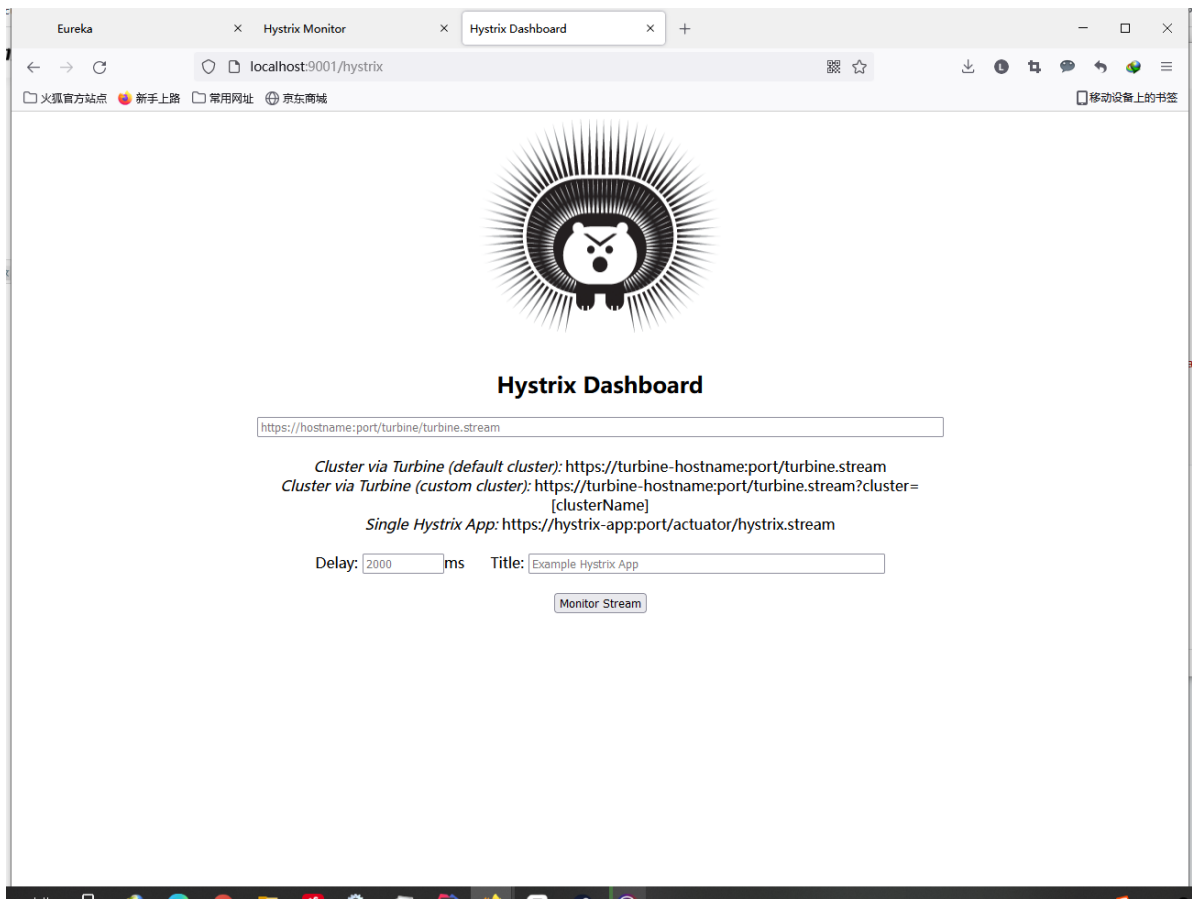
给springcloud-provider-dept-hystrix-8001模块下的主启动类添加如下代码,添加监控

```
@SpringBootApplication
@EnableEurekaClient //在服务启动后自动注册到eureka中!
@EnableDiscoveryClient //注册服务发现
@EnableCircuitBreaker // 添加对熔断的支持注解
public class DeptProviderHystrix_8001 {
    public static void main(String[] args) {
        SpringApplication.run(DeptProviderHystrix_8001.class, args);
    }

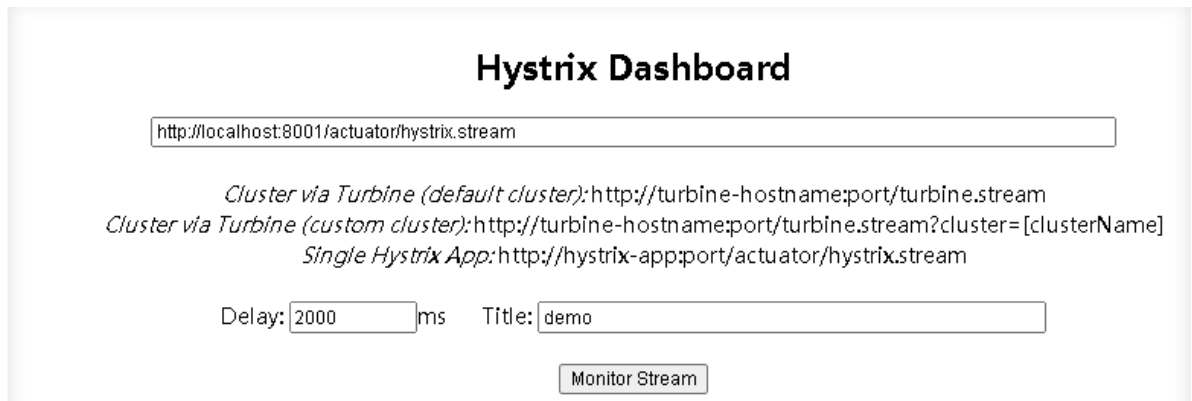
    //增加一个 Servlet
    @Bean
    public ServletRegistrationBean getServletRegistrationBean() {
        ServletRegistrationBean servletRegistrationBean = new
        ServletRegistrationBean(new HystrixMetricsStreamServlet());
        servletRegistrationBean.addUrlMappings("/actuator/hystrix.stream"); //访问该页面就是监控页面
        return servletRegistrationBean;
    }
}
```

注意: 必须添加熔断机制才可以使用dashboard监控

访问: <http://localhost:9001/hystrix>



进入监控页面: <http://localhost:8001/actuator/hystrix.stream>




localhost:9001/hystrix/monitor?stream=http%3A%2F%2Flocalhost%3A8001%2Ffactua

火狐官方网站 新手上路 常用网址 京东商城

Hystrix Stream: demo

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#) Success | St



queryByld
6 0 0.0 %
0 0
0 0

Host: 0.6/s
Cluster: 0.6/s
Circuit **Closed**

Hosts	1	90th	0ms
Median	0ms	99th	0ms
Mean	0ms	99.5th	0ms

Thread Pools Sort: [Alphabetical](#) | [Volume](#)

DeptController

Host: 0.6/s
Cluster: 0.6/s

Active	0	Max Active	1
Queued	0	Executions	6
Pool Size	10	Queue Size	5

localhost:8001//dept/find/1

localhost:8001//dept/find/1

火狐官方网站 新手上路 常用网址 京东商城

JSON 原始数据 头

保存 复制 全部折叠 全部展开 过滤 JSON

```

dept_no: 1
dept_name: "开发部"
db_source: "springcloud-db-01"
  
```

各个字符含义:



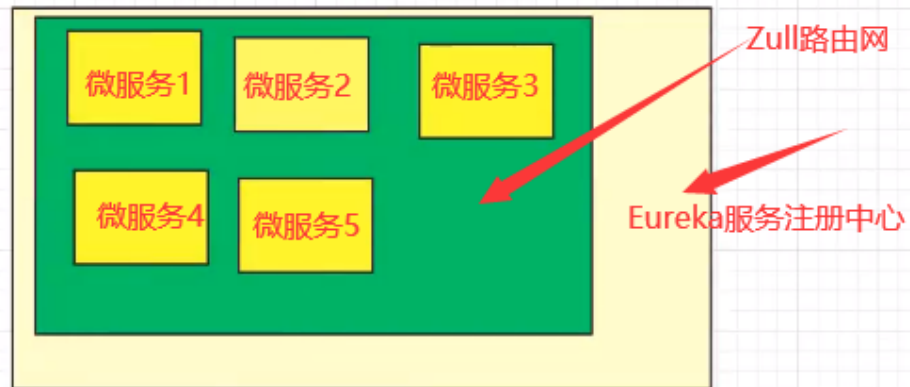
https://blog.csdn.net/weixin_43591980

Zuul路由网关

什么是zuul?

Zuul包含了对请求的**路由**(用来跳转的)和**过滤**两个最主要功能:

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。



https://blog.csdn.net/weixin_43591980

注意: Zuul 服务最终还是会注册进 Eureka

提供: 代理 + 路由 + 过滤 三大功能!

zuul能干嘛

- 路由
- 过滤

官方文档: <https://github.com/Netflix/zuul/>

案例:

新建springcloud-zuul-gateway-10086模块, 并导入依赖

```
<!-- zuul路由网关依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
  <version>2.2.9.RELEASE</version>
</dependency>
```

配置文件:

```
server:
  port: 10086

spring:
  application:
    name: springcloud-zuul

#eureka注册中心配置
eureka:
```



```

client:
  service-url:
    defaultZone:
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
  instance:
    instance-id: sprincloud-zuul-8001 # 修改eureka默认描述信息
    prefer-ip-address: true #自动获取本机url

info:
  app.author: youxin
  company.address: com.youxin.com

```

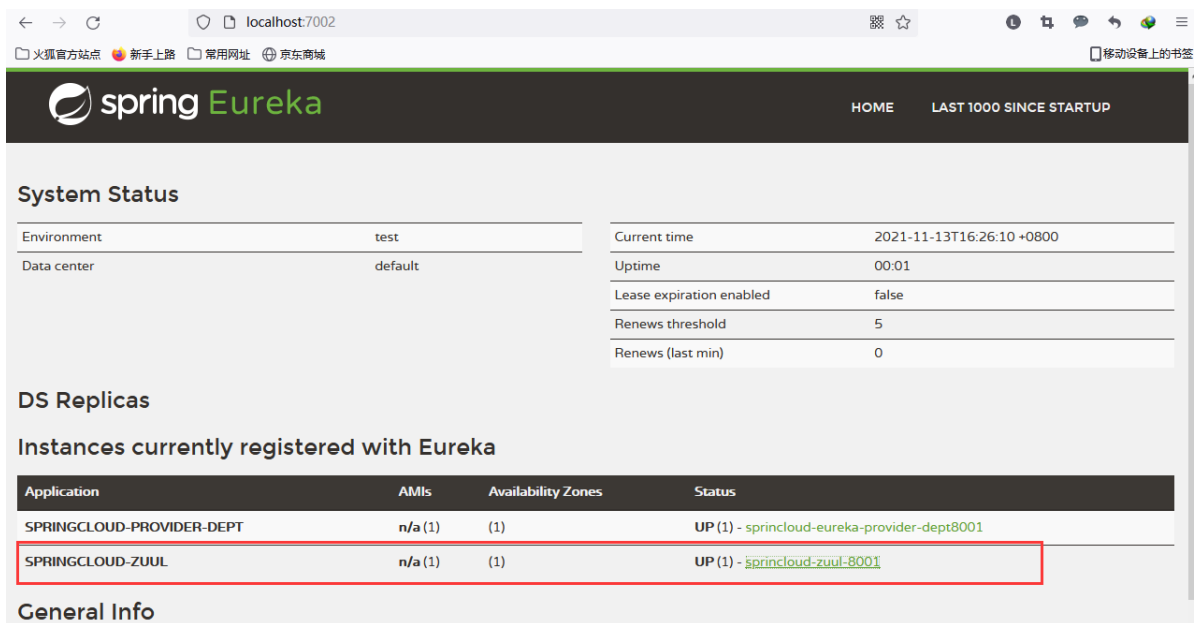
主启动类:

```

@SpringBootApplication
//开启zuul功能
@EnableZuulProxy
public class ZuulGatewayDept_10086 {
    public static void main(String[] args) {
        SpringApplication.run(ZuulGatewayDept_10086.class, args);
    }
}

```

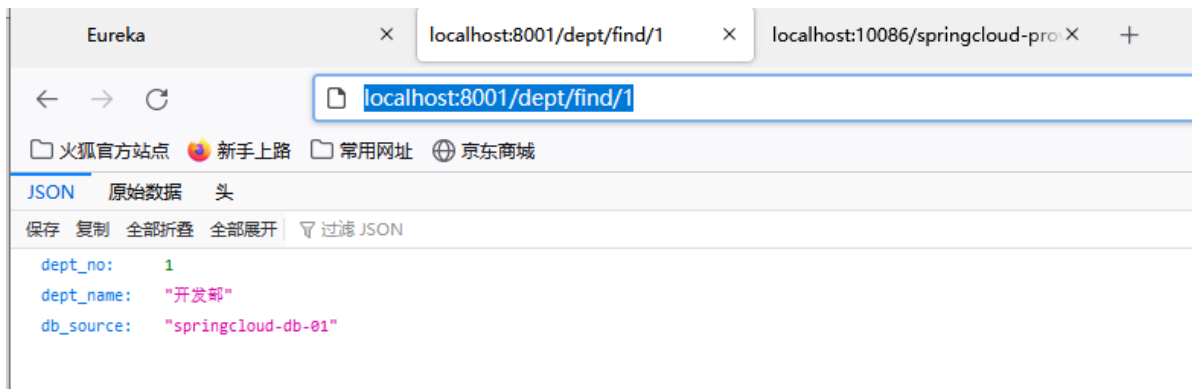
测试:



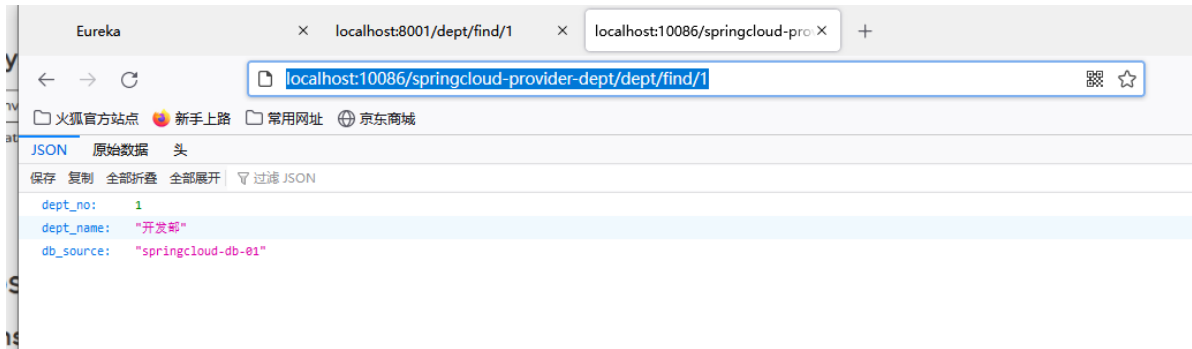
The screenshot shows the Spring Eureka web interface in a browser. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, there's a 'System Status' section with two tables. The first table shows 'Environment: test' and 'Data center: default'. The second table shows 'Current time: 2021-11-13T16:26:10 +0800', 'Uptime: 00:01', 'Lease expiration enabled: false', 'Renews threshold: 5', and 'Renews (last min): 0'. Below this is a 'DS Replicas' section. The main part of the page is titled 'Instances currently registered with Eureka' and contains a table with columns: 'Application', 'AMIs', 'Availability Zones', and 'Status'. The table lists two instances: 'SPRINGCLOUD-PROVIDER-DEPT' and 'SPRINGCLOUD-ZUUL'. The 'SPRINGCLOUD-ZUUL' instance is highlighted with a red box, showing its status as 'UP (1) - sprincloud-zuul-8001'. At the bottom, there's a 'General Info' section.

Application	AMIs	Availability Zones	Status
SPRINGCLOUD-PROVIDER-DEPT	n/a (1)	(1)	UP (1) - sprincloud-eureka-provider-dept8001
SPRINGCLOUD-ZUUL	n/a (1)	(1)	UP (1) - sprincloud-zuul-8001

访问<http://localhost:8001/dept/find/1>



再访问<http://localhost:10086/springcloud-provider-dept/dept/find/1>:



同样可以请求到结果

上图是没有经过Zuul路由网关配置时，服务接口访问的路由，可以看出直接用微服务(服务提供方)名称去访问，这样不安全，不能将微服务名称暴露！

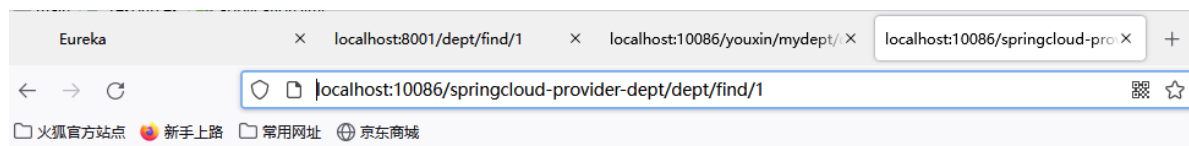
所以经过Zuul路由网关配置后，访问的路由为：

修改zuul配置：

```
# zuul配置
zuul:
  # 路由相关配置
  # 原来访问路由 eg:http://localhost:10086/springcloud-provider-dept/dept/get/1
  # zuul路由配置后访问路由 eg:http://localhost:10086/youxin/mydept/dept/get/1
  routes:
    # serviceId和path是K,V键值对不可以写错，前缀可以自定义
    youxin.serviceId: springcloud-provider-dept
    youxin.path: /mydept/**
    # 不能再使用原来这个路径访问了，*: 忽略,隐藏全部的服务名称~
    ignored-services: "*"
  # ignored-services: springcloud-provider-dept
  # 设置公共的前缀
  prefix: /youxin
```

再次启动：

使用原来的地址不能再访问了：



Whitelabel Error Page

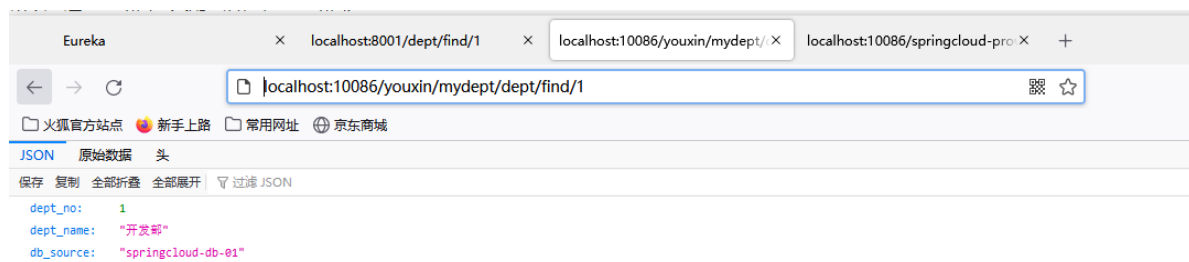
This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Nov 13 16:59:34 CST 2021

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

No message available

使用新地址：



我们看到，微服务名称被替换并隐藏，换成了我们自定义的微服务名称mydept，同时加上了前缀youxin，这样就做到了对路由访问的加密处理！

Spring Cloud Config 分布式配置

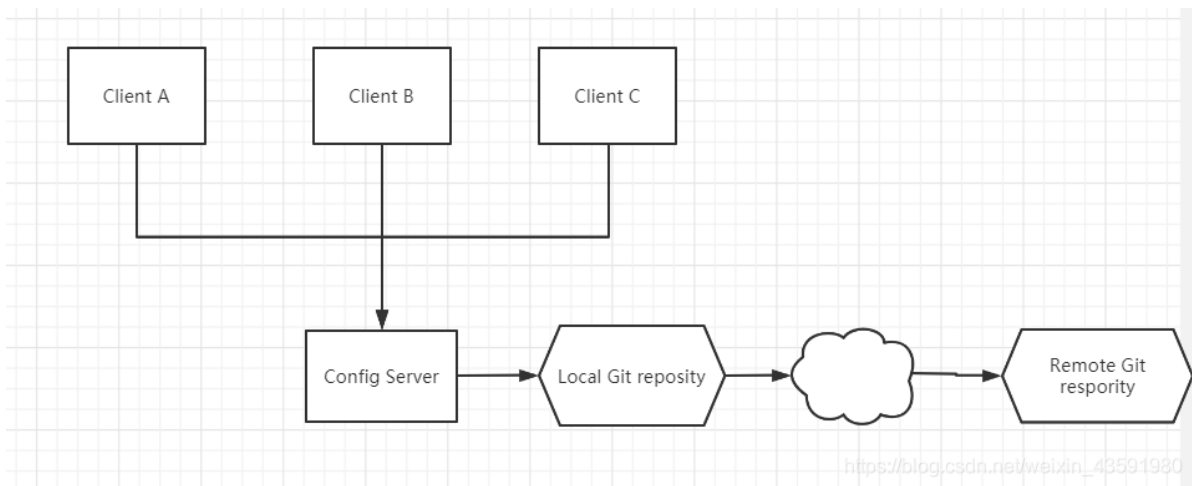
Spring Cloud Config为分布式系统中的外部配置提供服务器和客户端支持。使用Config Server，您可以在所有环境中管理应用程序的外部属性。客户端和服务端的概念映射与Spring Environment 和 PropertySource 抽象相同，因此它们与Spring应用程序非常契合，但可以与任何以任何语言运行的应用程序一起使用。随着应用程序通过从开发人员到测试和生产的部署流程，您可以管理这些环境之间的配置，并确定应用程序具有迁移时需要运行的一切。服务器存储后端的默认实现使用git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。很容易添加替代实现，并使用Spring配置将其插入。

概述

分布式系统面临的-配置文件问题

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，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。spring cloud提供了configServer来解决这个问题，我们每一个微服务自己带着一个application.yml，那上百个的配置文件修改起来，令人头疼！

什么是SpringCloud config分布式配置中心？



spring cloud config 为微服务架构中的微服务提供集中化的外部支持，配置服务器为各个不同微服务应用的所有环节提供了一个**中心化的外部配置**。

spring cloud 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。并且可用通过git客户端工具来方便的管理和访问配置内容。

spring cloud config 分布式配置中心能干嘛？

- 集中式管理配置文件
- 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
- 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配置自己的信息
- 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
- 将配置信息以REST接口的形式暴露

spring cloud config 分布式配置中心与GitHub整合

由于spring cloud config 默认使用git来存储配置文件 (也有其他方式，比如自持SVN 和本地文件)，但是最推荐的还是git，而且使用的是 http / https 访问的形式。

入门案例

gitee上创建远程仓库springcloud-config并克隆到本地：

添加application.yml:

```
spring:
  profiles:
    active: dev

---
spring:
  profiles: dev
  application:
    name: springcloud-config-dev

---
spring:
  profiles: test
```

```
application:
  name: springcloud-config-test
```

提交到远程仓库:

```
git add .
```

```
git commit -m "first commit"
```

```
git push origin master
```

服务端

新建springcloud-config-server-3366模块导入pom.xml依赖

```
<dependencies>

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

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.2.9.RELEASE</version>
  </dependency>

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

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
    <version>2.2.8.RELEASE</version>
  </dependency>

</dependencies>
```

编写配置文件:

```
server:
  port: 3366
spring:
  application:
    name: springcloud-config
  # 连接码云远程仓库
cloud:
  config:
    server:
      git:
        # 注意是https的而不是ssh
        uri: https://gitee.com/LYx1NZz/springcloud-config.git
```

```
# 不加这个配置会报Cannot execute request on any known server 这个错：连接Eureka服务端地址不对
# 或者直接注释掉eureka依赖 这里暂时用不到eureka
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

编写主启动类：

```
@SpringBootApplication
@EnableConfigServer// 开启spring cloud config server服务
public class ConfigServer_3366 {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServer_3366.class, args);
    }
}
```

定位资源的默认策略是克隆一个git仓库（在 `spring.cloud.config.server.git.uri`），并使用它来初始化一个迷你 `SpringApplication`。小应用程序的 `Environment` 用于枚举属性源并通过JSON端点发布。

HTTP服务具有以下格式的资源：

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

/application}-{
  profile}.yaml

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

/application}-{
  profile}.properties

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

其中“应用程序”作为 `SpringApplication` 中的 `spring.config.name` 注入（即常规的Spring Boot应用程序中通常是“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”）。

测试访问<http://localhost:3366/application-dev.yml>

```
localhost:3366/application-dev.yml

spring:
  profiles:
    active: dev
  application:
    name: springcloud-config-dev
```

测试访问 <http://localhost:3366/application/test/master>

localhost:3366/application/test/master

JSON 原始数据 头

保存 复制 全部折叠 全部展开 过滤 JSON

```
{
  "name": "application",
  "profiles": {
    "test": {
      "label": "master",
      "version": "905d24a9cc86934829dbfcb42082e1238a2a9ce8",
      "state": null
    }
  },
  "propertySources": [
    {
      "name": "https://gitee.com/LYx1NZz/springcloud-config.git/application.yml (document #2)",
      "source": {
        "spring.profiles": "test",
        "spring.application.name": "springcloud-config-test"
      }
    },
    {
      "name": "https://gitee.com/LYx1NZz/springcloud-config.git/application.yml (document #0)",
      "source": {
        "spring.profiles.active": "dev"
      }
    }
  ]
}
```

客户端

将本地git仓库springcloud-config文件夹下新建的springcloud-config-client.yml提交到码云仓库:

master	LYx1NZz	update springcloud-config-client.yml	2e77024	1分钟前	5次提交
		.gitignore	Initial commit	8小时前	
		LICENSE	Initial commit	8小时前	
		README.en.md	Initial commit	8小时前	
		README.md	Initial commit	8小时前	
		application.yml	first commit	8小时前	
		springcloud-config-client.yml	update springcloud-config-client.yml	1分钟前	

新建一个springcloud-config-client-8202模块，并导入依赖:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.2.9.RELEASE</version>
  </dependency>
</dependencies>
```

```

</dependency>

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

<!-- config客户端依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
    <version>2.2.8.RELEASE</version>
</dependency>
</dependencies>

```

编写配置文件:

```

# 系统级别的配置
spring:
  cloud:
    config:
      profile: test #环境
      name: springcloud-config-client #文件名
      label: master # 分支
      uri: http://localhost:3366 # 获取端口地址

```

编写控制类获取远程yml配置:

```

@RestController
public class ConfigClientController {

    @Value("${spring.application.name}")
    private String applicationName;

    @Value("${eureka.client.service-url.defaultZone}")
    private String eurekaServer;

    @Value("${server.port}")
    private String serverPort;

    @RequestMapping("/config")
    public String getConfig() {
        return "applicationName=>" + applicationName + "\n" +
            "eurekaServer=>" + eurekaServer + "\n" +
            "serverPort=>" + serverPort;
    }
}

```

测试:

使用configserver获取配置信息<http://localhost:3366/springcloud-config-client-dev.yml>:


```
localhost:3366/springcloud-config-client-dev.yml

spring:
  profiles:
    active: dev
  application:
    name: springcloud-config
server:
  port: 8201
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
  instance:
    instance-id: springcloud-config8201
    prefer-ip-address: true
info:
  app:
    name: youxin-springcloud
  company:
    name: www.youxin.com
  auther: 我是谁
```

使用client获取配置信息：<http://localhost:8202/config> 并发现配置信息是远程仓库中调用的（本地并没有写端口）

```
localhost:8202/config

applicationName=>springcloud-config eurekaServer=>http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/ serverPort=>8202
```

测试成功！

测试案例：

上传springcloud-provider-config.yml

```
spring:
  profiles:
    active: dev

---

server:
  port: 8001

#mybatis配置
mybatis:
  type-aliases-package: com.youxin.springcloud.pojo
  mapper-locations: classpath:mybatis/mapper/*.xml
  config-location: classpath:mybatis/mybatis.xml

#spring配置
spring:
  profiles: dev
  application:
    name: springcloud-provider-dept
  datasource:
    # 数据源
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/springcloud-db-01?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    username: root
    password: 1234

#springboot默认是不注入这些属性值的，需要自己绑定
#druid 数据源专有配置
```

```

    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true

# eureka配置, 表示服务需要注册到哪里
eureka:
  client:
    service-url:
#      单机:
#      defaultZone: http://localhost:7001/eureka/
#      集群: 需要向每个集群中注册进去
    defaultZone:
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
  instance:
    instance-id: springcloud-eureka-provider-dept8001 # 修改eureka默认描述信息
    prefer-ip-address: true #自动获取本机url

#info配置, 这里面的东西都是自己定义配置的
info:
  app.name: youxin-springcloud
  company.name: www.youxin.com
  auther: 廖友鑫

---
server:
  port: 8004

#mybatis配置
mybatis:
  type-aliases-package: com.youxin.springcloud.pojo
  mapper-locations: classpath:mybatis/mapper/*.xml
  config-location: classpath:mybatis/mybatis.xml

#spring配置
spring:
  profiles: test
  application:
    name: springcloud-provider-dept
  datasource:
#    数据源
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/springcloud-db-03?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    username: root

```

```
password: 1234
```

```
#springboot默认是不注入这些属性值的，需要自己绑定
```

```
#druid 数据源专有配置
```

```
initialSize: 5
```

```
minIdle: 5
```

```
maxActive: 20
```

```
maxWait: 60000
```

```
timeBetweenEvictionRunsMillis: 60000
```

```
minEvictableIdleTimeMillis: 300000
```

```
validationQuery: SELECT 1 FROM DUAL
```

```
testWhileIdle: true
```

```
testOnBorrow: false
```

```
testOnReturn: false
```

```
poolPreparedStatements: true
```

```
# eureka配置，表示服务需要注册到哪里
```

```
eureka:
```

```
  client:
```

```
    service-url:
```

```
#      单机:
```

```
#      defaultZone: http://localhost:7001/eureka/
```

```
#      集群: 需要向每个集群中注册进去
```

```
    defaultZone:
```

```
http://localhost:7001/eureka/,http://localhost:7002/eureka/,http://localhost:7003/eureka/
```

```
  instance:
```

```
    instance-id: springcloud-eureka-provider-dept8001 # 修改eureka默认描述信息
```

```
    prefer-ip-address: true #自动获取本机url
```

```
#info配置,这里面的东西都是自己定义配置的
```

```
info:
```

```
  app.name: youxin-springcloud
```

```
  company.name: www.youxin.com
```

```
  auther: 廖友鑫
```

重新生成springcloud-provider-dept-config模块:

用bootstrap.yml文件导入远程配置文件:

```
spring:
  cloud:
    config:
      name: springcloud-provider-config
      label: master
      uri: http://localhost:3366
      profile: test
```

测试发现8004端口可用且配置文件修改成功:

localhost:3366/springcloud-provi × Eureka × localhost:8004/dept/find/1 × +

← → ↻

localhost:8004/dept/find/1

火狐官方网站 新手上路 常用网址 京东商城

JSON 原始数据 头

保存 复制 全部折叠 全部展开 过滤 JSON

dept_no: 1

dept_name: "开发部"

db_source: "springcloud-db-03"