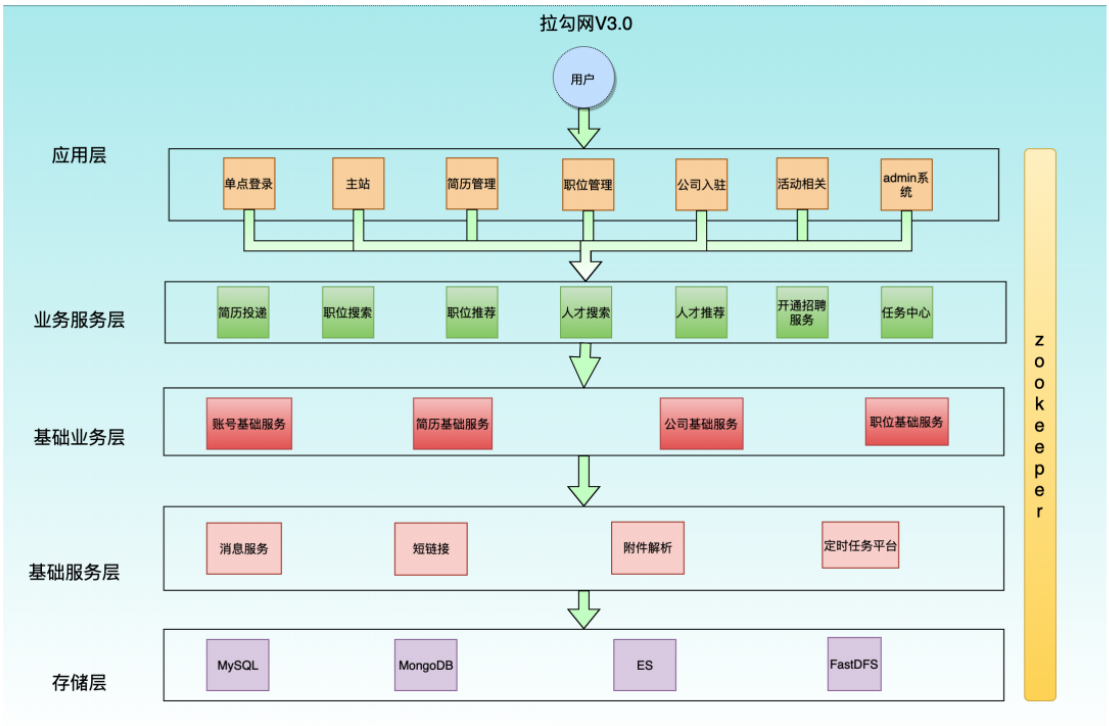
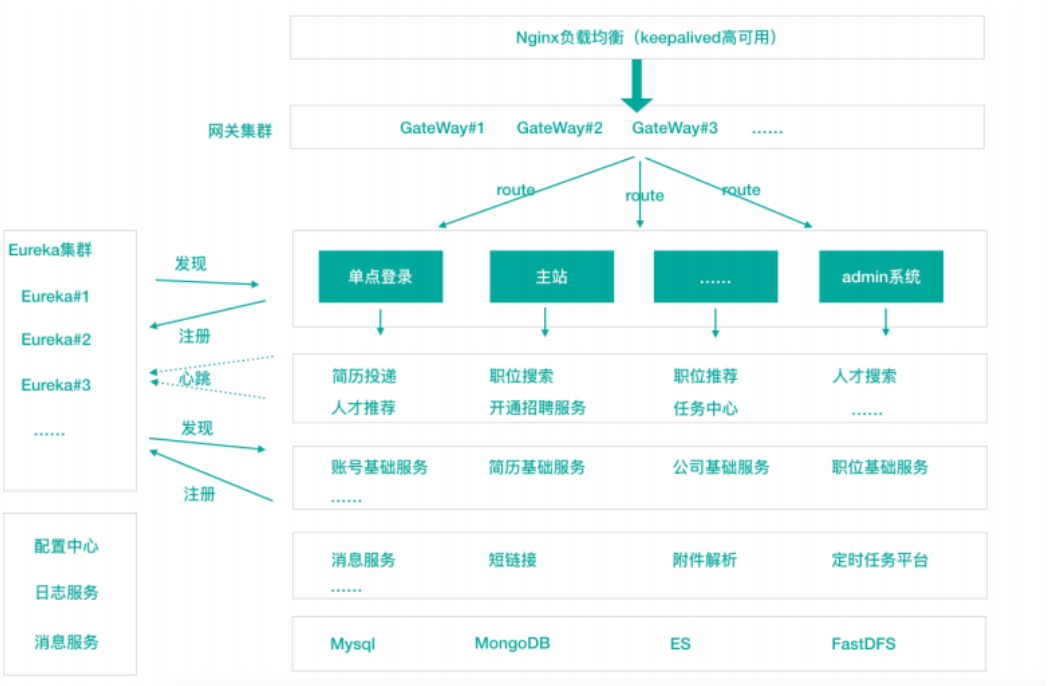
什么是微服务应用架构，和SOA区别是什么?

SOA:



微服务：



微服务可以说是SOA架构的一种拓展，比SOA拆分粒度更小、服务更独立，不同的服务可以使用不同的开发语言和存储，服务之间往往通过Restful等轻量 级通信。微服务架构强调的一个重点是“**业务需要彻底的组件化和服务化**”。

微服务架构和SOA架构相似又不同：

服务拆分粒度的不同，从服务拆分上来说变化并不大，只是引入了相对完整的新一代Spring Cloud微服务技术。

Spring Cloud是什么?

Spring Cloud是一系列框架的有序集合（是一个规范），Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

Spring Cloud 解决什么问题?

Spring Cloud 规范及实现意图要解决的问题其实就是微服务架构实施过程中存在的一些问题，比如微服务架构中的服务注册发现问题、网络问题（比如熔断场景）、统一认证安全授权问题、负载均衡问题、链路追踪等问题。

Zookeeper、Eureka、Consul、Nacos对比

- **Zookeeper**

  Zookeeper它是一个分布式服务框架，是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式 应用配置项的管理等。

  简单来说zookeeper本质=存储+监听通知。

  zNode

  Zookeeper 用来做服务注册中心，主要是因为它具有节点变更通知功能，只要客户端监听相关服 务节点，服务节点的所有变更，都能及时的通知到监听客户端，这样作为调用方只要使用 Zookeeper 的客户端就能实现服务节点的订阅和变更通知功能了，非常方便。另外，Zookeeper 可用性也可以，因为只要半数以上的选举节点存活，整个集群就是可用的。

- **Eureka**

  由Netflix开源，并被Pivatal集成到SpringCloud体系中，它是基于 RestfulAPI 风格开发的服务注册与发现组件。

- **Consul**

  Consul是由HashiCorp基于Go语言开发的支持多数据中心分布式高可用的服务发布和注册服务软件，采用Raft算法保证服务的一致性，且支持健康检查。

- **Nacos**

  Nacos是一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。简单来说 Nacos 就是注册中心 + 配置中心的组合，帮助我们解决微服务开发必会涉及到的服务注册 与发现，服务配置，服务管理等问题。Nacos 是 Spring Cloud Alibaba 核心组件之一，负责服务注册与发现， 还有配置。

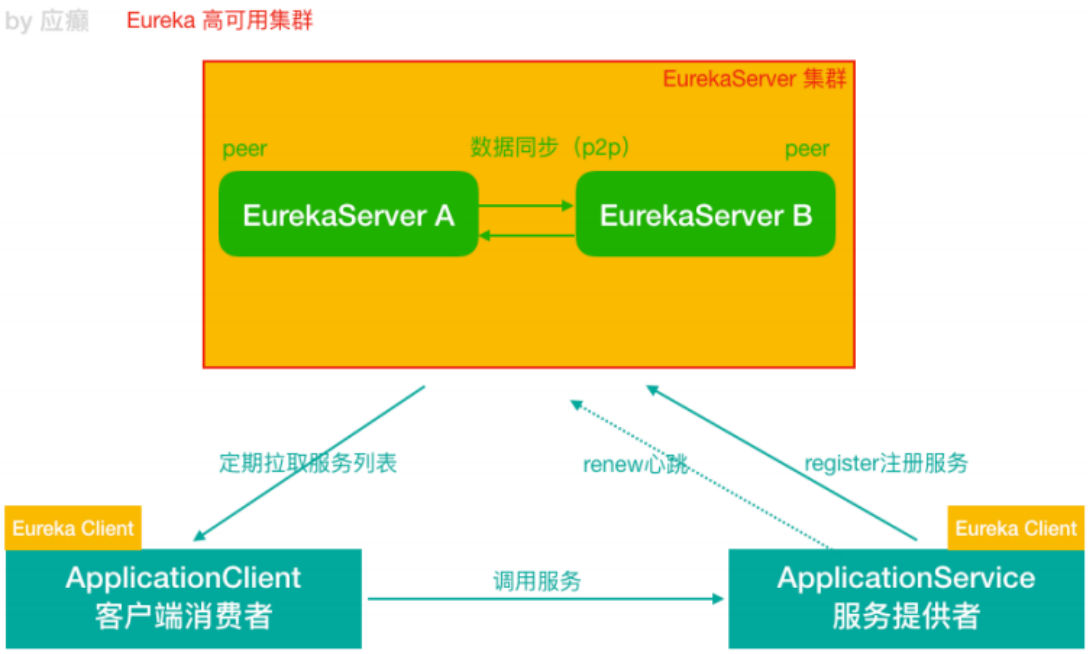
组件名	语言	CAP	对外暴露接口
Eureka	Java	AP（自我保护机制，保证可用）	HTTP
Consul	Go	CP	HTTP/DNS
Zookeeper	Java	CP	客户端
Nacos	Java	支持AP/CP切换	HTTP

技术选型	CAP模型	适用规模(建议)	控制台管理	社区活跃度
Eureka	AP	<30k	支持	低
Zookeeper	CP	<20k	不支持	中
Consul	AP	<5k	支持	高
Nacos	AP/CP	100k+	支持	靠大家啦☺

Eureka 和 Zookeeper 的最大区别： Eureka 是 AP 模型，Zookeeper 是 CP 模型。在出现脑裂等场景时，Eureka 可用性是每一位，也就是说出现脑裂时，每个分区仍可以独立提供服务，是去中心化的

Eureka注册中心

配置Eureka高可用集群：



EurekaServer配置：

```
#启动类添加注解：
@EnableEurekaServer
#添加依赖：
 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
 </dependency>
#yml配置：
spring:
 profiles: LagouCloudEurekaServerB
server:
 port: 8762
```

```
eureka:
 instance:
    hostname: LagouCloudEurekaServerB
 client:
    register-with-eureka: true
    fetch-registry: true
     service-url: #客户端与EurekaServer交互的地址，如果是集群，也需要写其它Server的地址
        defaultZone: http://LagouCloudEurekaServerA:8761/eureka
 spring:
    application:
         name: lagou-cloud-eureka-server
```

EurekaClient配置：

```
#启动类添加注解：
@EnableDiscoveryClient
#添加依赖：
<dependency>
     <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
#yml配置：
server:
 port: 8090
eureka:
 client:
    service-url: # eureka server的路径
        defaultZone:
http://lagoucloudeurekaservera:8761/eureka/,http://lagoucloudeurekaserverb
:8762/eureka/ #把 eureka 集群中的所有 url 都填写了进来，也可以只写一台，因为各个eureka server 可以同步注册表
 instance:
 #使用ip注册，否则会使用主机名注册了（此处考虑到对老版本的兼容，新版本经过实验都是ip）
    prefer-ip-address: true
 #自定义实例显示格式，加上版本号，便于多版本管理，注意是ip-address，早期版本是ipAddress
    instance-id:
${spring.cloud.client.ipaddress}:${spring.application.name}:${server.port}:@projec
t.version@
spring:
    application:
         name: lagou-cloud-eureka-client
```

调用EurekaClient配置：

```
//RestTemplate方式：
String url = "http://lagou-service-resume/resume/openstate/" + userId;
Integer forObject = restTemplate.getForObject(url, Integer.class);
//Feign方式：参考feign模块
```

url解释：

lagou-service-resume是spring.application.name，resume和openstate见下图：

```java
@RestController
@RequestMapping("/resume")
public class ResumeController {

    @Autowired
    private ResumeService resumeService;

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

    //"/resume/openstate/1545132"
    @GetMapping("/openstate/{userId}")
    public Integer findDefaultResumeState(@PathVariable Long userId) {
        //return resumeService.findDefaultResumeByUserId(userId).getIsOpenResume();
        System.out.println("====>>>>>>>>>>>>>>>我是8081，访问到我这里了......");
        return port;
    }
```

EurekaServer和EurekaClient配置的区别和相同点：

1. eureka.client.service-url.defaultZone参数EurekaServer只需要配置其他EurekaServer的路径或者只配置一个EurekaServer的路径也可以因为各个eureka server 可以同步注册表，EurekaClient则需要配置所有EurekaServer的路径。
2. EurekaServer和EurekaClient都要在启动类加@EnableDiscoveryClient或@EnableDiscoveryClient（开启服务发现注解）

> @EnableDiscoveryClient和@EnableEurekaClient二者的功能是一样的。但是如果选用的是 eureka 服务器，那么就推荐@EnableEurekaClient，如果是其他的注册中心，那么推荐使用@EnableDiscoveryClient，考虑到通用性，后期我们可以使用@EnableDiscoveryClient

Eureka元数据：

Eureka的元数据有两种：

- 标准元数据：主机名、IP地址、端口号等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。
- 自定义元数据：可以使用eureka.instance.metadata-map配置，符合KEY/VALUE的存储格式。这些元数据可以在远程客户端中访问。
  - 自定义元数据：

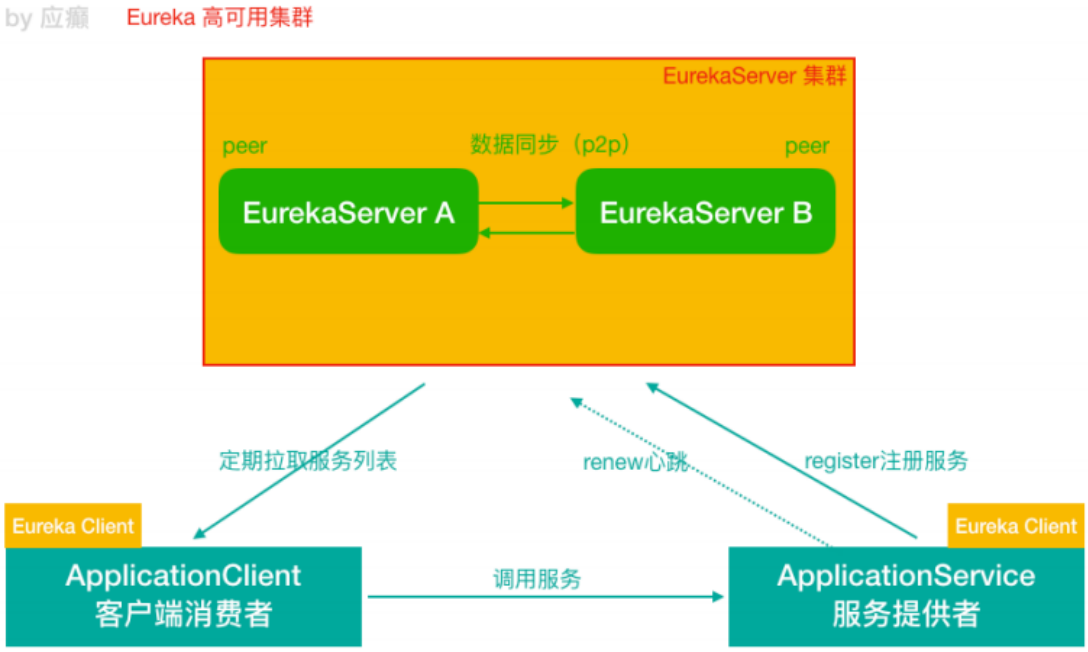
```yaml
instance:
    prefer-ip-address: true
    metadata-map:
        # 自定义元数据(kv自定义)
         cluster: cl1
         region: rn1
```

  - 元数据结构图：

获取指定微服务元数据：

```
public class AutodeliverApplicationTest {
    @Autowired
    private DiscoveryClient discoveryClient;
    @Test
    public void test() {
        // 从EurekaServer获取指定微服务实例
        List<ServiceInstance> serviceInstanceList =
discoveryClient.getInstances("lagou-service-resume");
        // 循环打印每个微服务实例的元数据信息
        for (int i = 0; i < serviceInstanceList.size(); i++) {
            ServiceInstance serviceInstance = serviceInstanceList.get(i);
            System.out.println(serviceInstance);
     }
    }
}
```

Eureka客户端和服务端详解：



Eureka客户端详解：

1. Eureka注册中心把服务的信息（包括元数据）保存在Map中。
2. 服务提供者renew心跳：服务每隔30秒会向注册中心续约(心跳)一次（也称为报活），如果没有续约，租约在90秒后到期，然后服务会被失效。每隔30秒的续约操作我们称之为心跳检测

```yaml
eureka:
 instance:
    # 租约续约间隔时间，默认30秒
    lease-renewal-interval-in-seconds: 30
    # 租约到期时间，默认90秒
    lease-expiration-duration-in-seconds: 90
```

3. 客户端消费者定期拉取服务列表：每隔30秒服务会从注册中心中拉取一份服务列表，这个时间可以通过配置修改。往往不需要我们调整（服务消费者启动时，从 EurekaServer服务列表获取只读备份，**缓存到本地**，每隔30秒（时间可配置），会重新获取并更新数据）

```yaml
eureka:
    client:
         # 每隔多久拉取一次服务列表
        registry-fetch-interval-seconds: 30
```

Eureka服务端详解：

1. **服务下线：**当服务正常关闭操作时，会发送服务下线的REST请求给EurekaServer。服务中心接受到请求后，将该服务置为下线状态。
2. **失效剔除：**Eureka Server会定时（间隔值是eureka.server.eviction-interval-timer-in-ms，默认60s）进行检查， 如果发现实例在在一定时间（此值由客户端设置的eureka.instance.lease-expiration-duration-inseconds定义，默认值为90s）内没有收到心跳，则会注销此实例。
3. **自我保护（意义是有可能提供者和注册中心网络有问题不代表提供者不可用）：**

   服务提供者 —> 注册中心

   自我保护进入条件：如果在15分钟内超过85%的客户端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，Eureka Server自动进入自我保护机制。

进入自我保护模式时：

1）**不会剔除任何服务实例**（可能是服务提供者和EurekaServer之间网络问题），保证了大多数服务依然可用

2）Eureka Server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它节点上**，保证当前节点依然可用，当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中。

3）在Eureka Server工程中通过eureka.server.enable-self-preservation配置可用关停自我保护，默认值是打开（生产环境建议打开）

Eureka Server故障判断：

这里存在一个问题，如何判断是Eureka Server故障，还是服务故障，Eureka Server提供的判断条件是，**当出现大量的服务续约超时**，那么就会认为自己出现了问题。如果出现**少量服务续约超时，则认为服务故障。**

EurekaServer崩溃恢复：

1. **重启：**

   Spring Cloud Eureka 启动时，在初始化 EurekaServerBootstrap#initEurekaServerContext 时会调用 PeerAwareInstanceRegistryImpl#syncUp 从其它 Eureka 中同步数据。

2. **脑裂:**

   - 一是脑裂很快恢复，一切正常；
   - 二是该实例已经自动过期，则重新进行注册；
   - 三是数据冲突，出现不一致的情况，则需要发起同步请求，其实也就是重新注册一次，同时踢除老的实例。（数据会不会冲突是通过最近变更时间来判断的）

Eureka同步过程：

- Eureka Server也是一个Client，在启动时，通过请求其中一个节点（Server），将自身注册到Server上，并获取注册服务信息；（1.为什么是读本地的实例信息 2.怎么将他自身的信息发给别的实例）
- 每当Server信息变更后（client发起注册，续约，注销请求），就将信息通知给其他Server，来保持数据同步；
- **在执行同步（复制）操作时，可能会有数据冲突，是通过lastDirtyTimestamp，最近一次变更时间来保证是最新数据；**

比如 Eureka Server A 向 Eureka Server B 复制数据，数据冲突有2种情况：

（1）A 的数据比 B 的新，B 返回 404，A 重新把这个应用实例注册到 B。

（2）A 的数据比 B 的旧，B 返回 409，要求 A 同步 B 的数据。

Eureka的缺点?

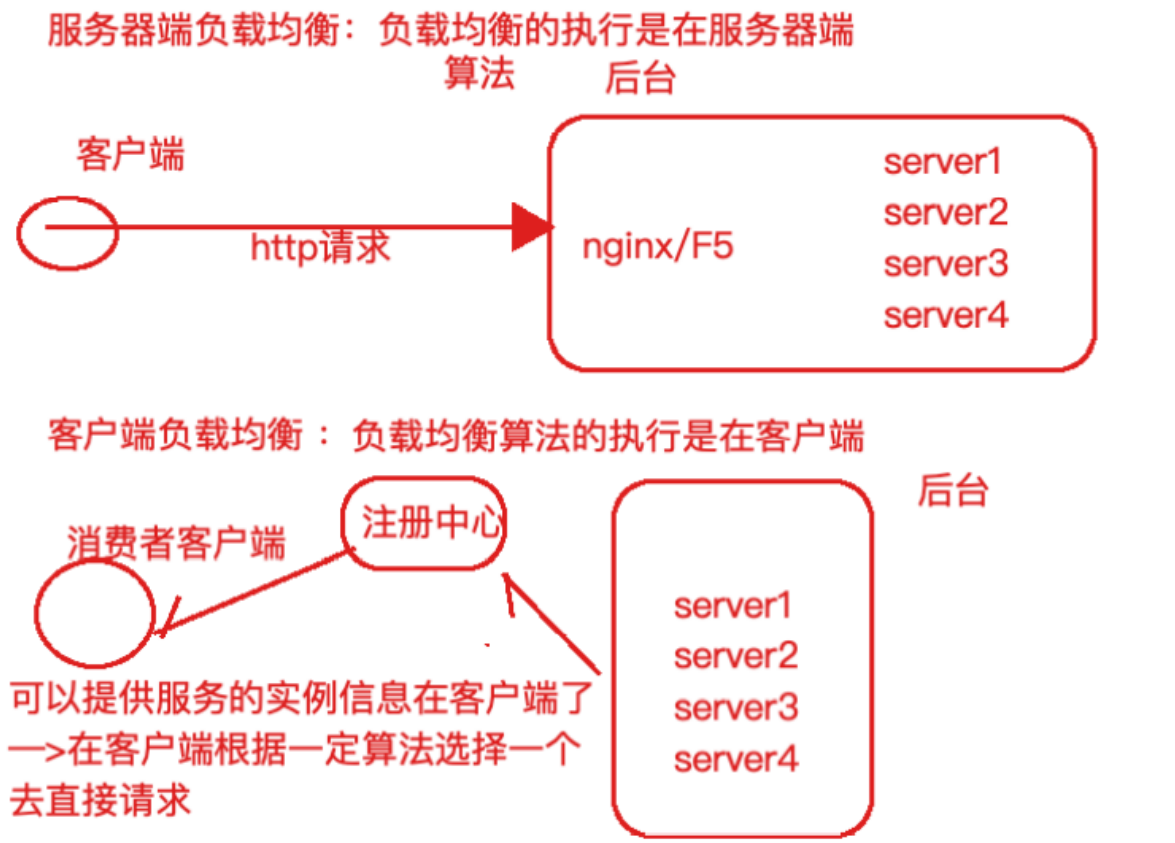
ZooKeeper基于CP，不保证高可用，如果zookeeper正在选主，或者Zookeeper集群中半数以上机器不可用，那么将无法获得数据。Eureka基于AP，能保证高可用，即使所有机器都挂了，也能拿到本地缓存的数据。作为注册中心，其实配置是不经常变动的，只有发版和机器出故障时会变。对于不经常变动的配置来说，CP是不合适的，而AP在遇到问题时可以用牺牲一致性来保证可用性，既返回旧数据，缓存数据。

所以理论上Eureka是更适合作注册中心。而现实环境中大部分项目可能会使用ZooKeeper，那是因为集群不够大，并且基本不会遇到用做注册中心的机器一半以上都挂了的情况。所以实际上也没什么大问题。

Ribbon负载均衡

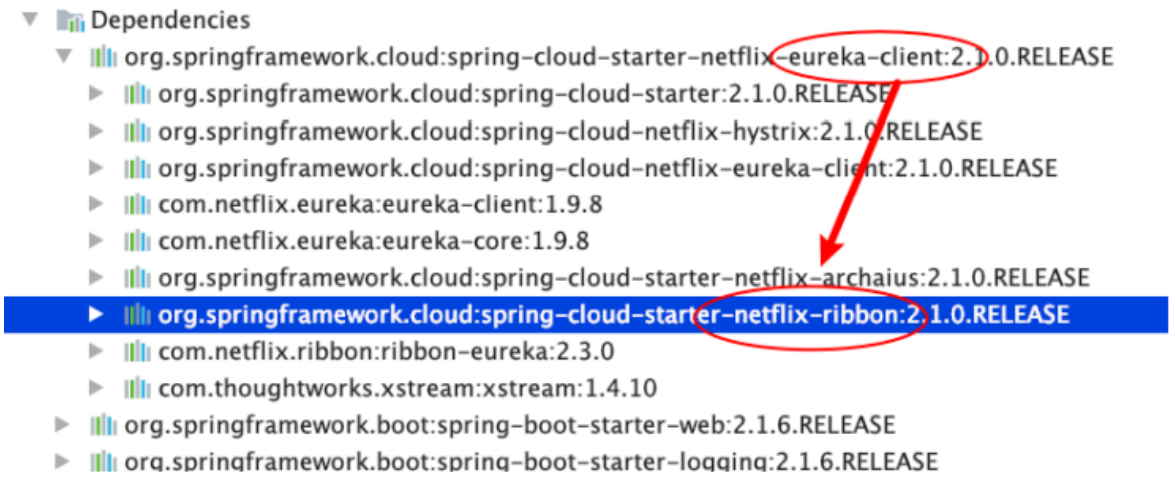
Ribbon使用：

服务端和客户端负载均衡的区别：负载均衡算法一个在后台，一个在客户端。



Ribbon用法：

不需要引入额外的Jar坐标，因为在服务消费者中我们引入过eureka-client，它会引入Ribbon相关Jar



RestTemplate方式：

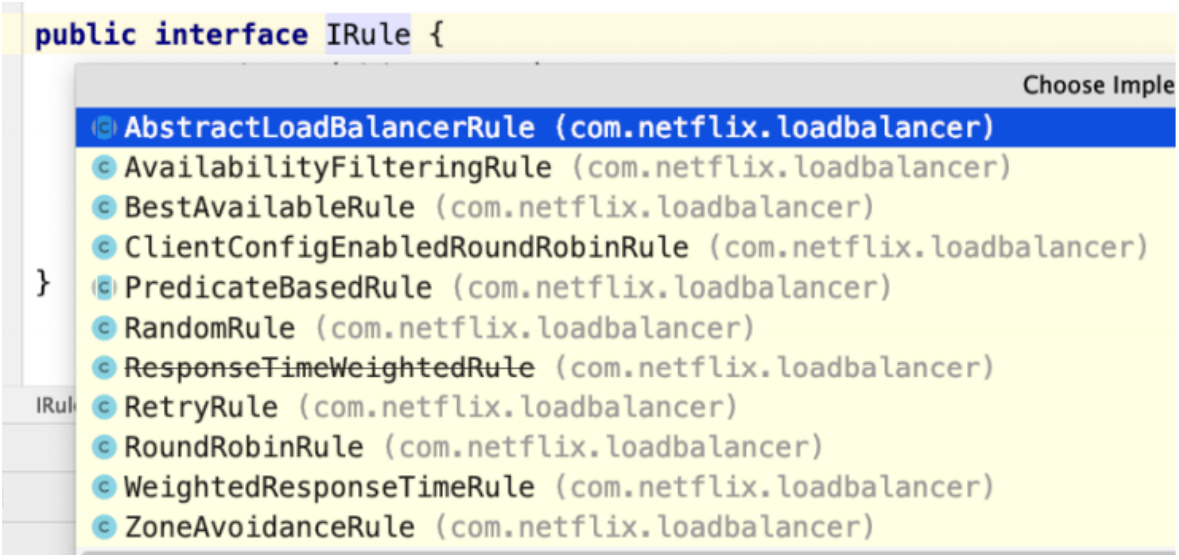
```
    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

 /**
   * 使用Ribbon负载均衡
   * @param userId
   * @return
   */
   @GetMapping("/checkState/{userId}")
   public Integer findResumeOpenState(@PathVariable Long userId) {
    // 使用ribbon不需要我们自己获取服务实例然后选择一个那么去访问了（自己的负载均衡）
    String url = "http://lagou-service-resume/resume/openstate/" + userId;
    //两个服务提供者的spring.application.name都是lagou-service-resume
    Integer forObject = restTemplate.getForObject(url, Integer.class);
    return forObject;
  }
```

Feign方式：yml加配置项即可

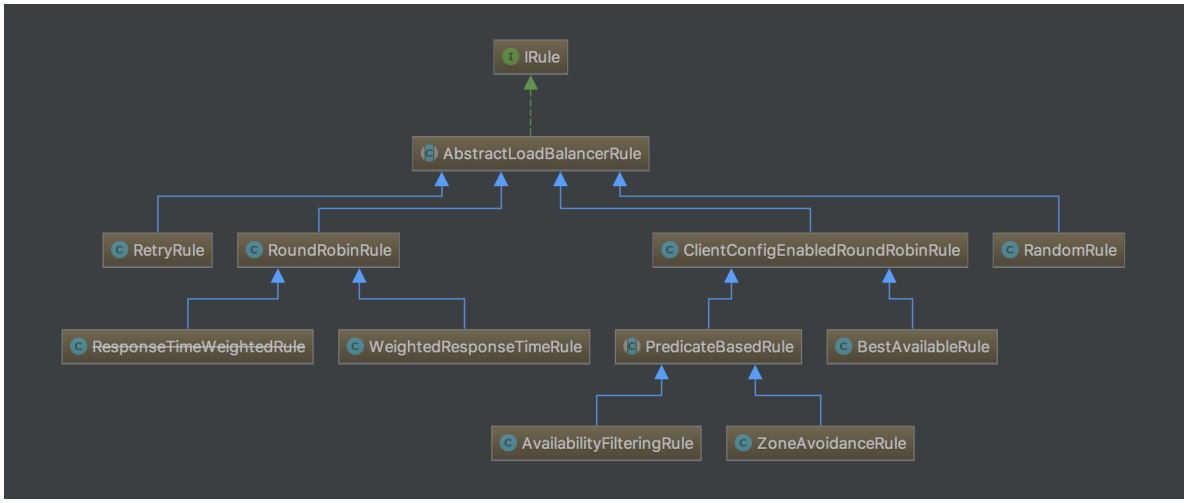
```
#针对的被调用方微服务名称,不加就是全局生效
lagou-service-resume:
  ribbon:
    #请求连接超时时间（超时触发后会调用另外实例）
    ConnectTimeout: 2000
    #请求处理超时时间
    ReadTimeout: 3000
    #对所有操作都进行重试
    OkToRetryOnAllOperations: true
    #根据如上配置，当访问到故障请求的时候，它会再尝试访问一次当前实例（次数由MaxAutoRetries配置），
    #如果不行，就换一个实例进行访问，如果还不行，再换一次实例访问（更换次数由MaxAutoRetriesNextServer配置）。
    #如果依然不行，返回失败信息。
    MaxAutoRetries: 0 #对当前选中实例重试次数，不包括第一次调用
    MaxAutoRetriesNextServer: 0 #切换实例的重试次数
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule #负载策略调整
```

Ribbon内置了多种负载均衡策略，内部负责复杂均衡的顶级接口为 com.netflix.loadbalancer.IRule ， 类树如下



负载均衡策略	描述
RoundRobinRule：轮询策略	默认超过10次获取到的server都不可用，会返回一个空的server
RandomRule：随机策略	如果随机到的server为null或者不可用的话，会while不停的循环选取
RetryRule：重试策略	一定时限内循环重试。默认继承RoundRobinRule，也支持自定义 注入，RetryRule会在每次选取之后，对选举的server进行判断， 是否为null，是否alive，并且在500ms内会不停的选取判断。而 RoundRobinRule失效的策略是超过10次，RandomRule是没有失 效时间的概念，只要serverList没都挂。
BestAvailableRule：最小连接数策略	遍历serverList，选取出可用的且连接数最小的一个server。该算 法里面有一个LoadBalancerStats的成员变量，会存储所有server 的运行状况和连接数。如果选取到的server为null，那么会调用 RoundRobinRule重新选取。
AvailabilityFilteringRule：可用过滤策略	扩展了轮询策略，会先通过默认的轮询选取一个server，再去判断 该server是否超时可用，当前连接数是否超限，都成功再返回。
ZoneAvoidanceRule：区域权衡策略（默认策略）	扩展了轮询策略，继承了2个过滤器：ZoneAvoidancePredicate和AvailabilityPredicate，除了过滤超时和链接数过多的server，还会过滤掉不符合要求的zone区域里面的所有节点，AWS --ZONE 在**一个区域/机房内的服务**实例中**轮询**

Ribbon源码：



入口：RibbonAutoConfiguration

Hystrix熔断器

Hystrix简介：

Hystrix能够提升系统的可用性与容错性，Hystrix主要通过以下几点实现延迟和容错：

1. **包裹请求：**使用HystrixCommand包裹对依赖的调用逻辑。 自动投递微服务方法 （@HystrixCommand添加Hystrix控制） ——调用简历微服务
2. **跳闸机制：**当某服务的错误率超过一定的阈值时，Hystrix可以跳闸，停止请求该服务一段时间。
3. **资源隔离：**Hystrix为每个依赖都维护了一个小型的线程池(舱壁模式)（或者信号量）。如果该线程池已满， 发往该依赖的请求就被立即拒绝，而不是排队等待，从而加速失败判定。
4. **监控：**Hystrix可以近乎实时地监控运行指标和配置的变化，例如成功、失败、超时、以及被拒绝的请求等。
5. **回退机制：**当请求失败、超时、被拒绝，或当断路器打开时，执行回退逻辑。回退逻辑由开发人员自行提供，例如返回一个缺省值。
6. **自我修复：**断路器打开一段时间后，会自动进入“半开”状态。

什么是雪崩效应（流量暴增）：

扇入：微服务调用次数

扇出：微服务调用其他微服务次数（扇入大是好事，扇出大不一定是好事）

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又调用其它的微服务，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**”。

解决雪崩效应方案：

1. **服务熔断**：**一般和服务降级一起用**，服务断掉，当检测到该节点微服务调用响应正常后，恢复调用链路。
2. **服务降级：**通俗讲就是整体资源不够用了，先将一些不关紧的服务停掉（调用我的时候，给你返回一个预留的值， 也叫做兜底数据），待渡过难关高峰过去，再把那些服务打开。
3. **服务限流：**服务降级是当服务出问题或者影响到核心流程的性能时，暂时将服务屏蔽掉，待高峰或者问题解决后再打开；但是有些场景并不能用服务降级来解决，比如秒杀业务这样的核心功能，这个时候可以结合服务限流来限制这些场景的并发/请求量

   限流措施也很多，比如

- 限制总并发数（比如数据库连接池、线程池）
- 限制瞬时并发数（如nginx限制瞬时并发连接数）
- 限制时间窗口内的平均速率（如Guava的RateLimiter、nginx的limit_req模块，限制每秒的平均速率）
- 限制远程接口调用速率、限制MQ的消费速率等

限流算法：

保护高并发系统的三把利器：限流、缓存、降级。

常见的限流算法（主要用漏桶和令牌桶）：

1、计数器（固定窗口）算法

2、滑动窗口算法

3、漏桶算法

4、令牌桶算法

- **固定窗口算法：**

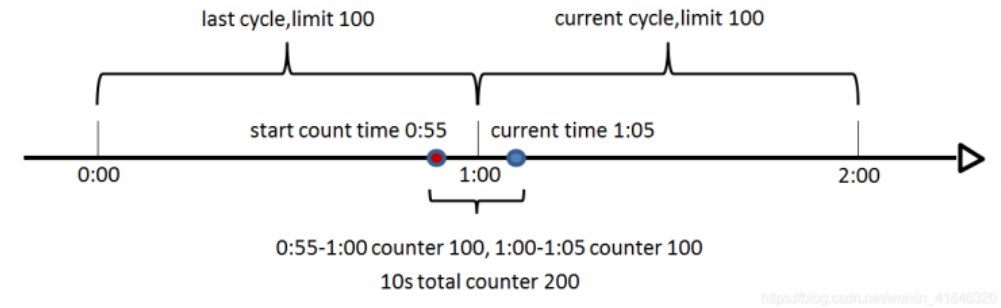
请求通过，计数值加1，当计数值超过预先设定的阈值时，就拒绝单位时间内的其他请求。如果单位时间已经结束，则将计数器清零，开启下一轮的计数。（等于说固定时间窗口，每个时间窗口只执行这么多请求）

代码实现：

```
public class FixedWindow {
    private long time = new Date().getTime();
    private Integer count = 0; // 计数器
    private final Integer max = 100; // 请求阈值
    private final Integer interval = 1000; // 窗口大小
    public boolean trafficMonitoring() {
        long nowTime = new Date().getTime();
        if (nowTime < time + interval) {
            // 在时间窗口内
            count++;
            return max > count;
        } else {
            time = nowTime; // 开启新的窗口
            count = 1; // 初始化计数器,由于这个请求属于当前新开的窗口,所以记录这个请求
            return true;
        }
    }
}
```

固定窗口算法存在的问题：

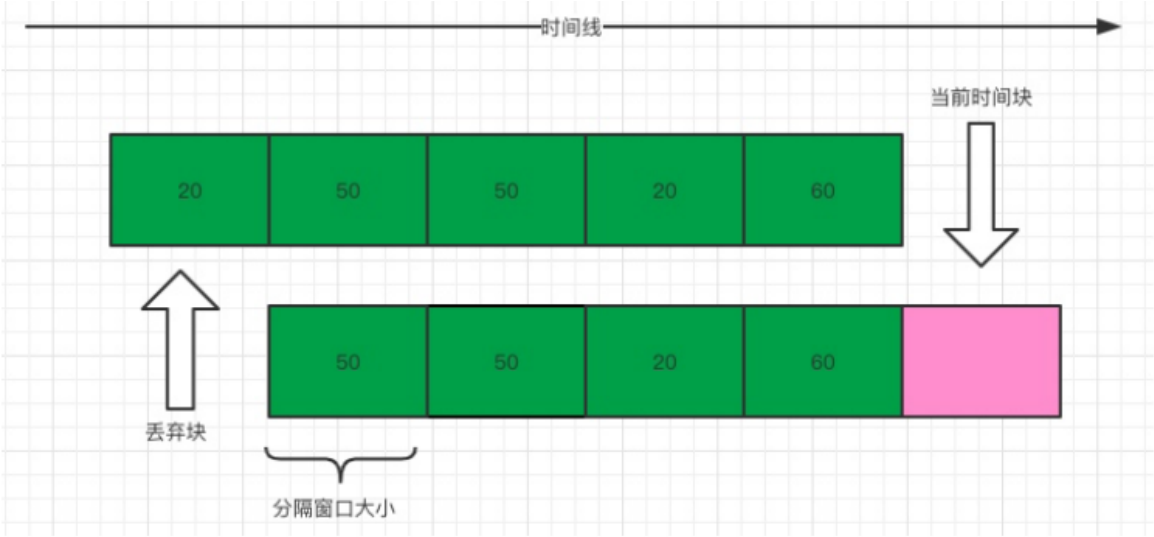
> 临界值问题：



如上图，时间窗口分为两段，每1s固定执行100个请求，但是假如第一段只有最后100ms才执行100个请求，而第二段开始100ms执行100个请求，这两段合起来的单位时间请求数量显然超过了阈值，但没有限流。（这种情况也叫突刺现象）

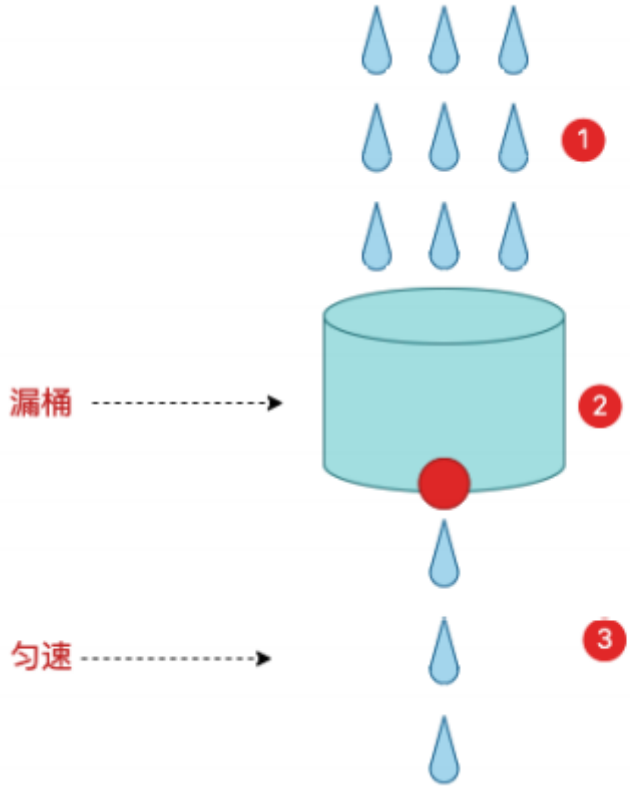
- **滑动窗口算法：（需再百度清楚）**

滑动窗口算法解决了上面的临界值问题，假设我们仍然设定1秒内允许通过的请求是100个，但是在这里我们需要把1秒的时间分成多格，假设分成5格（格数越多，流量过渡 越平滑），每格窗口的时间大小是200毫秒，每过200毫秒，就将窗口向前移动 一格。为了便于理解，可以看下图：



由上图可知，每200ms允许通过的请求数是20。流量的过渡是否平滑依赖于我们设置的窗口格数也就是统计时间间隔，格数越多，统计越精确，但是具体要分多少格.....

- **漏桶算法：**



如上所示，所有请求都要装入桶中（桶一般使用队列实现，会固定大小，如果超过大小则采用拒绝或服务降级），然后顺序执行请求。

> Nginx按请求速率限速模块使用的是漏桶算法，即能够强行保证请求的实时处理速度不会超过设置的阈值。

缺点：无法应对短时间的突发流量。

- **令牌桶算法：**

  算法思想是：

  - 令牌以固定速率产生，并缓存到令牌桶中；（如果桶容量为100，而令牌的产生速度为10/s，加入前9s没有处理请求，那么这时桶就可以同时处理90容量的请求，所以令牌桶可以应对短时间的突发流量）
  - 令牌桶放满时，多余的令牌被丢弃；
  - 请求要消耗等比例的令牌才能被处理；
  - 令牌不够时，请求被缓存。

令牌桶和漏桶对比：

- 令牌桶是按照固定速率往桶中添加令牌，请求是否被处理需要看桶中令牌是否足够，**当令牌数减为零时则拒绝新的请求**；漏桶则是按照常量**固定速率流出请求**，**流入请求速率任意**，当流入的请求数累积到漏桶容量时，则新流入的请求被拒绝；
- 令牌桶限制的是平均流入速率，**允许突发请求**，只要有令牌就可以处理；漏桶限制的是常量流出速率，即流出速率是一个**固定常量值**，比如都是1的速率流出，而不能一次是1，下次又是2，从而平滑突发流入速率；
- 令牌桶允许一定程度的突发，而漏桶主要目的是平滑流出速率。

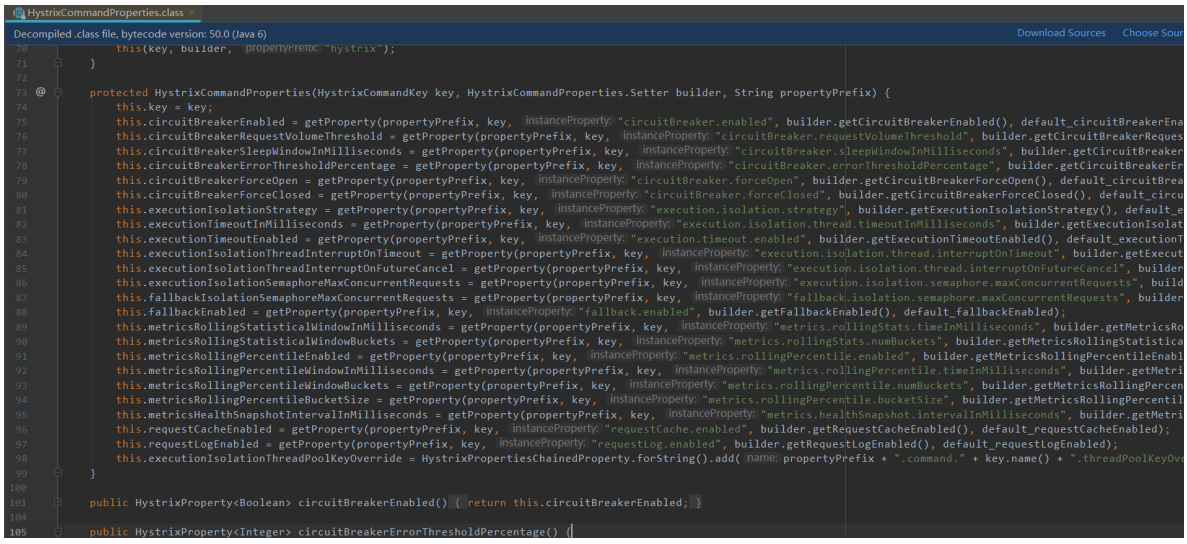
Hystrix使用：

调用者配置就好：

```
//加入依赖
<dependency>
    <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
//启动类加入注解
@EnableCircuitBreaker
```

```
// 使用@HystrixCommand注解进行熔断控制
@HystrixCommand(
        // 线程池标识，要保持唯一，不唯一的话就共用了
        threadPoolKey = "findResumeOpenStateTimeout",
        // 线程池细节属性配置
        threadPoolProperties = {
                @HystrixProperty(name="coreSize",value = "1"), // 线程数
                @HystrixProperty(name="maxQueueSize",value="20") // 等待队列长度
        },
        // commandProperties熔断的一些细节属性配置
        commandProperties = {
                // 每一个属性都是一个HystrixProperty
                @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="2000")
        }
)
@GetMapping("/checkStateTimeout/{userId}")
public Integer findResumeOpenStateTimeout(@PathVariable Long userId) {
    // 使用ribbon不需要我们自己获取服务实例然后选择一个那么去访问了（自己的负载均衡）
    String url = "http://lagou-service-resume/resume/openstate/" + userId;  // 指定服务名
    Integer forObject = restTemplate.getForObject(url, Integer.class);
    return forObject;
}
```

Hystrix的属性配置都在HystrixCommandProperties类的构造方法里面（如下图）：




```
// 超时+服务降级配置
 commandProperties = {
/*每一个属性都是一个HystrixProperty*/
@HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="2000")
.fallbackMethod = "myFallBack"  // 回退方法（触发超时情况下会调用自定义myFallBack（）返回结果）
```

```
hystrix:
  threadpool:
    default:
      coreSize: 10 #并发执行的最大线程数，默认10
      maxQueueSize: 1500 #BlockingQueue的最大队列数，默认值-1（等于-1时该配置不起作用）
      queueSizeRejectionThreshold: 1000 #队列大小拒绝阈值（当线程队列中有五个请求时，之后的所有请求都会被拒绝）
```

Hystrix 推荐配置

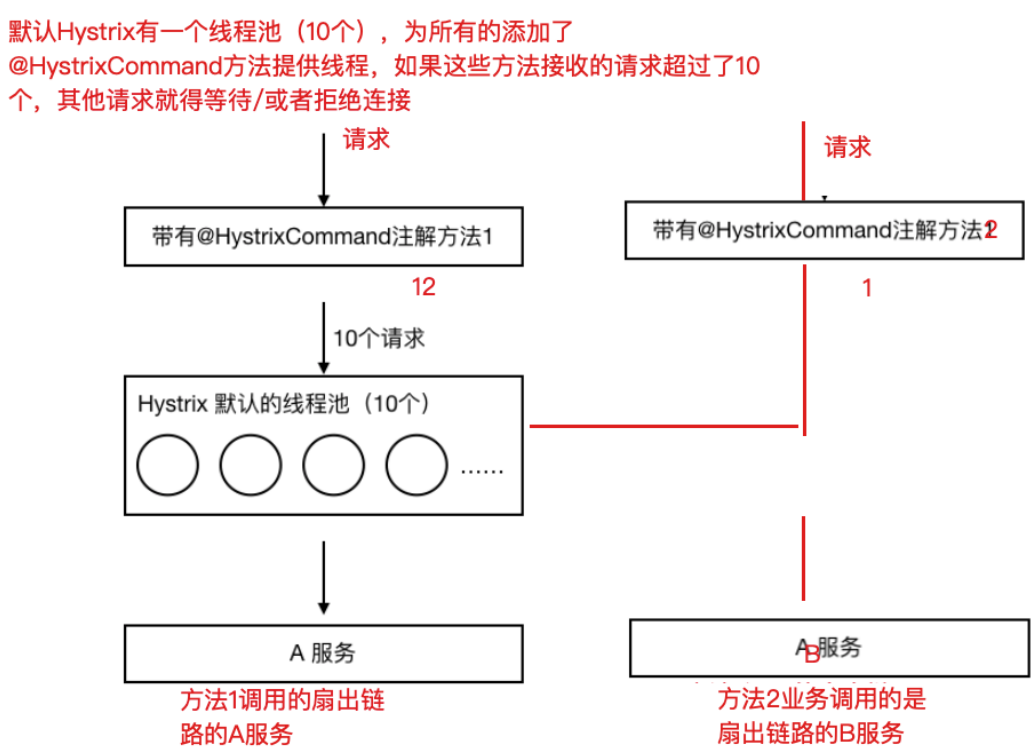
关于Hystrix线程池配置没有通用答案，具体问题具体分析。

- 线程池默认大小为10
- spring cloud 官方文档对于hystrix线程池配置的建议是10-20个
- CPU核数

注意：core默认为10，不代表每秒处理请求的能力为10。Hystrix线程池的配置 取决于接口性能及设置超时时间等因素。

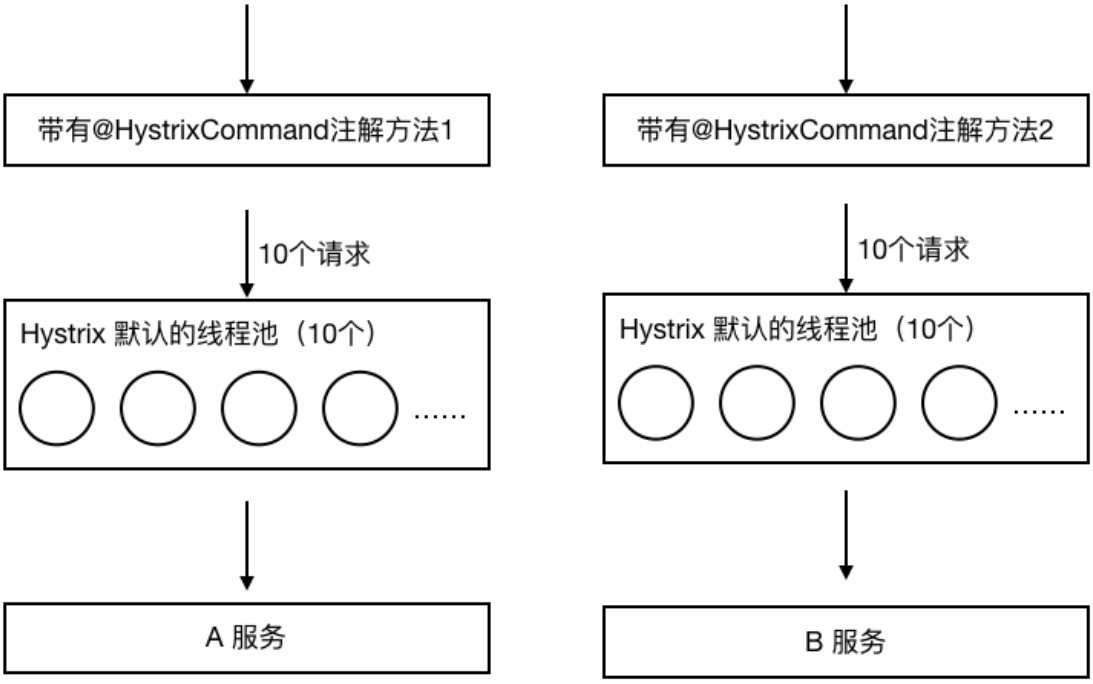
舱壁模式：

介绍：每个添加@HystrixCommand的方法都使用同一个Hystrix线程池，如果线程池个数不够，会造成请求等待/拒绝连接，并不是请求自己的问题，如下图：



针对上面的问题，使用每个添加@HystrixCommand的方法都有自己的Hystrix线程池，这样互不影响，这就是舱壁模式。

by 应癫 Hystrix舱壁模式



舱壁模式使用：

```
//每个加了@HystrixCommand方法添加下面的注解，形成线程池隔离
// 线程池标识，要保持唯一，不唯一的话就共用了
threadPoolKey = "findResumeOpenStateTimeoutFallback",
// 线程池细节属性配置
threadPoolProperties = {
    @HystrixProperty(name="coreSize",value = "2"), // 线程数
    @HystrixProperty(name="maxQueueSize",value="20") // 等待队列长度
},
```

可 以 使 用

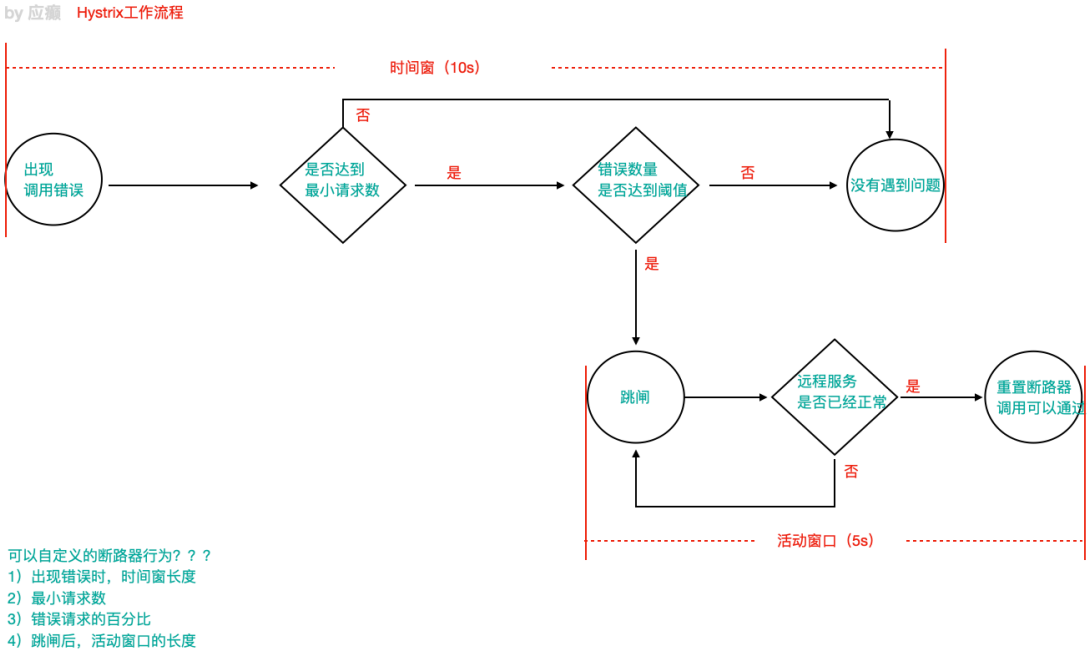
```
yingdian — -bash — 122×35
localhost:~ yingdian$ jps  jps命令查看java进程
1138 RemoteMavenServer36
2419 AutodeliverApplication
2420 Launcher
1396 LagouEurekaServerApp8762
1401 LagouResumeApplication8080
2458 Jps
987
1404 LagouResumeApplication8081  使用jstack查看指定进程中的线程信息，
1391 LagouEurekaServerApp8761    过滤出和hystrix有关的线程信息
localhost:~ yingdian$ which jps
/Library/Java/JavaVirtualMachines/jdk-11.0.5.jdk/Contents/Home/bin/jps
localhost:~ yingdian$ jstack 2419 | grep hystrix
```

发起请求，可以使用PostMan模拟批量请求

```
localhost:~ yingdian$ jstack 2419 | grep hystrix
"hystrix-AutodeliverControlle-1" #83 daemon prio=5 os_prio=31 cpu=20.25ms elapsed=80.00s tid=0x00007f8b252e0800 nid=0x139
03 waiting on condition  [0x000070000e3ea000]   10个线程
"hystrix-AutodeliverControlle-2" #86 daemon prio=5 os_prio=31 cpu=3.40ms elapsed=79.93s tid=0x00007f8b23a13000 nid=0xc603
 waiting on condition  [0x000070000e6f3000]
"hystrix-AutodeliverControlle-3" #88 daemon prio=5 os_prio=31 cpu=3.64ms elapsed=77.85s tid=0x00007f8b22d49000 nid=0x1350
3 waiting on condition  [0x000070000e8f9000]
"hystrix-AutodeliverControlle-4" #90 daemon prio=5 os_prio=31 cpu=3.56ms elapsed=75.80s tid=0x00007f8b22cab800 nid=0xcd03
 waiting on condition  [0x000070000eaff000]
"hystrix-AutodeliverControlle-5" #92 daemon prio=5 os_prio=31 cpu=4.02ms elapsed=65.72s tid=0x00007f8b232e0800 nid=0x1310
3 waiting on condition  [0x000070000ec02000]
"hystrix-AutodeliverControlle-6" #93 daemon prio=5 os_prio=31 cpu=3.62ms elapsed=65.67s tid=0x00007f8b23975800 nid=0x12e0
3 waiting on condition  [0x000070000ed05000]
"hystrix-AutodeliverControlle-7" #94 daemon prio=5 os_prio=31 cpu=9.25ms elapsed=63.58s tid=0x00007f8b22bd5000 nid=0x12d0
3 waiting on condition  [0x000070000ee08000]
"hystrix-AutodeliverControlle-8" #95 daemon prio=5 os_prio=31 cpu=3.62ms elapsed=61.55s tid=0x00007f8b22c48800 nid=0x12a0
3 waiting on condition  [0x000070000ef0b000]
"hystrix-AutodeliverControlle-9" #96 daemon prio=5 os_prio=31 cpu=15.41ms elapsed=51.46s tid=0x00007f8b25257000 nid=0xd20
3 waiting on condition  [0x000070000f00e000]
"hystrix-AutodeliverControlle-10" #97 daemon prio=5 os_prio=31 cpu=16.99ms elapsed=49.42s tid=0x00007f8b23990800 nid=0x12
803 waiting on condition  [0x000070000f111000]
localhost:~ yingdian$
```

跳闸+自我修复机制：

如下图，当请求一个添加@HystrixCommand方法时，如果请求数达到设置的值并且请求错误次数小于原先设定的阈值，那么请求正常放行，大于阈值，启动跳闸，后面再隔一段时间（活动窗口，默认5s）放行一个请求，如果请求正常那么说明该方法恢复正常，如果还是报错，那么还是跳闸状态。



跳闸+自我修复机制使用：

```yaml
# springboot中暴露健康检查等断点接口
management:
  endpoints:
    web:
      exposure:
        include: "*"
  # 暴露健康接口的细节
  endpoint:
    health:
      show-details: always
```

```
// 统计时间窗口定义
@HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds",value = "8000"),
// 统计时间窗口内的最小请求数
@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value = "2"),
// 统计时间窗口内的错误数量百分比阈值
@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value = "50"),
// 自我修复时的活动窗口长度
@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value = "3000")
```

Feign远程调用组件

Feign简介:

- 是Netflix开发的一个轻量级RESTful的HTTP服务客户端（用它来发起请求，远程调用的），**是以Java接口注解的方式调用Http请求**，而不用像Java中通过封装HTTP请求报文的方式直接调用，Feign被广泛应用在Spring Cloud 的解决方案中。
- 使用Feign非常简单，创建一个接口（在消费者--服务调用方这一端），并在接口上添加一些注解，代码就完成了
- SpringCloud对Feign进行了增强，使Feign支持了SpringMVC注解（OpenFeign）

Feign使用:

```
//启动类添加注解:
@EnableFeignClients
@EnableDiscoveryClient

//添加依赖:
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>

//建立调用接口
@FeignClient(value = "lagou-service-email")
public interface EmailServiceFeignClient {
    // Feign要做的事情就是，拼装url发起请求
    // 我们调用该方法就是调用本地接口方法，那么实际上做的是远程请求
    @GetMapping("/email/sendEmail/{email}/{code}")
    Boolean sendAuthEmail(@PathVariable("email") String
email,@PathVariable("code") String code);
}
```

@FeignClient注解的name属性用于指定要调用的服务提供者名称，和服务提供者yml文件中spring.application.name保持一致

Feign对负载均衡的支持：

```
#针对的被调用方微服务名称,不加就是全局生效
lagou-service-resume:
  ribbon:
  #请求连接超时时间
  #ConnectTimeout: 2000
  #请求处理超时时间
  #ReadTimeout: 5000
  #对所有操作都进行重试
  OkToRetryOnAllOperations: true
  #根据如上配置，当访问到故障请求的时候，它会再尝试访问一次当前实例（次数由MaxAutoRetries配置），
  #如果不行，就换一个实例进行访问，如果还不行，再换一次实例访问（更换次数MaxAutoRetriesNextServer配置），
####如果依然不行，返回失败信息。
  MaxAutoRetries: 0 #对当前选中实例重试次数，不包括第一次调用
  MaxAutoRetriesNextServer: 0 #切换实例的重试次数
  NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule #负载策略调整
```

Feign对熔断器的支持：

```
# 开启Feign的熔断功能
feign:
  hystrix:
  enabled: true
```

Feign对请求压缩和响应压缩的支持：

Feign 支持对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。通过下面的参数 即可开启请求与响应的压缩功能：

```
feign:
    compression:
        request:
            enabled: true # 开启请求压缩
            mime-types: text/html,application/xml,application/json # 设置压缩的数据类型，此处也是默认值
            min-request-size: 2048 # 设置触发压缩的大小下限，此处也是默认值
     response:
        enabled: true # 开启响应压缩
```

Feign的日志级别配置：

Feign是http请求客户端，类似于咱们的浏览器，它在请求和接收响应的时候，可以打印出比较详细的一些日志信息（响应头，状态码等等） 如果我们想看到Feign请求时的日志，我们可以进行配置，默认情况下Feign的日志没有开启

1) 开启Feign日志功能及级别

```java
// Feign的日志级别（Feign请求过程信息）
// NONE：默认的，不显示任何日志----性能最好
// BASIC：仅记录请求方法、URL、响应状态码以及执行时间----生产问题追踪
// HEADERS：在BASIC级别的基础上，记录请求和响应的header
// FULL：记录请求和响应的header、body和元数据----适用于开发及测试环境定位问题
@Configuration
public class FeignConfig {
    @Bean
    Logger.Level feignLevel() {
        return Logger.Level.FULL;
    }
}
```

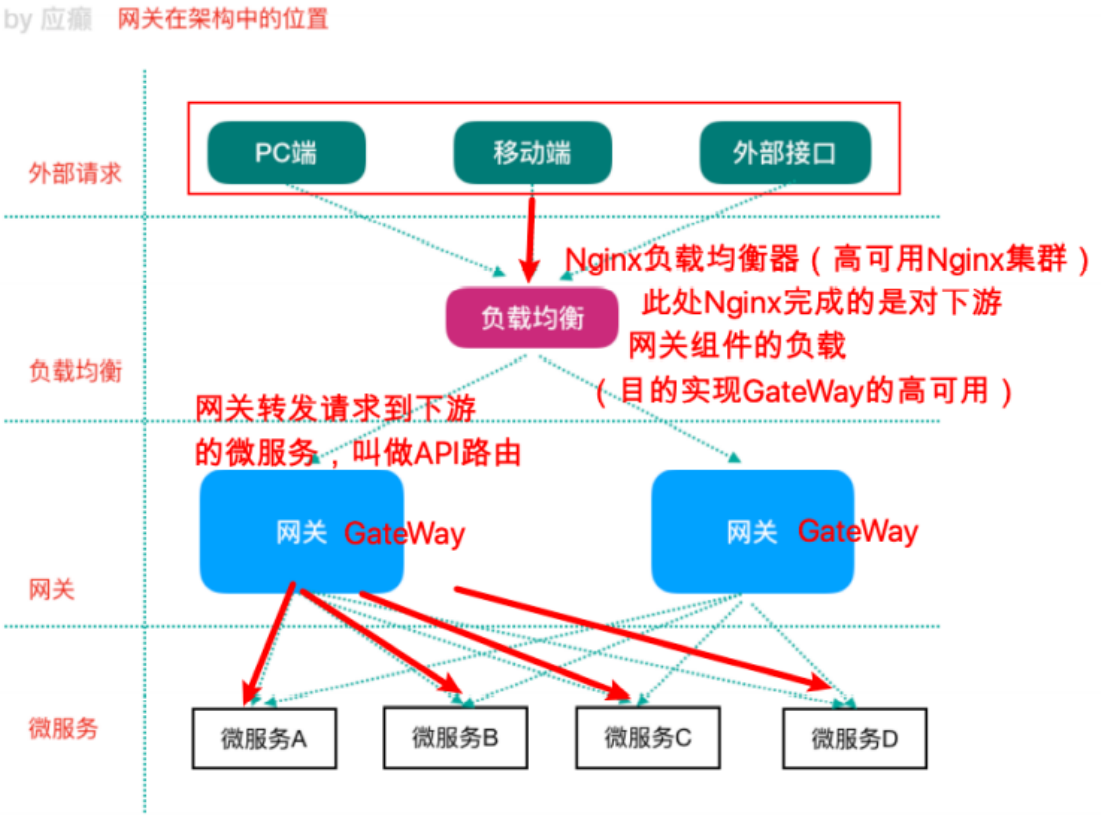
2) 配置log日志级别为debug

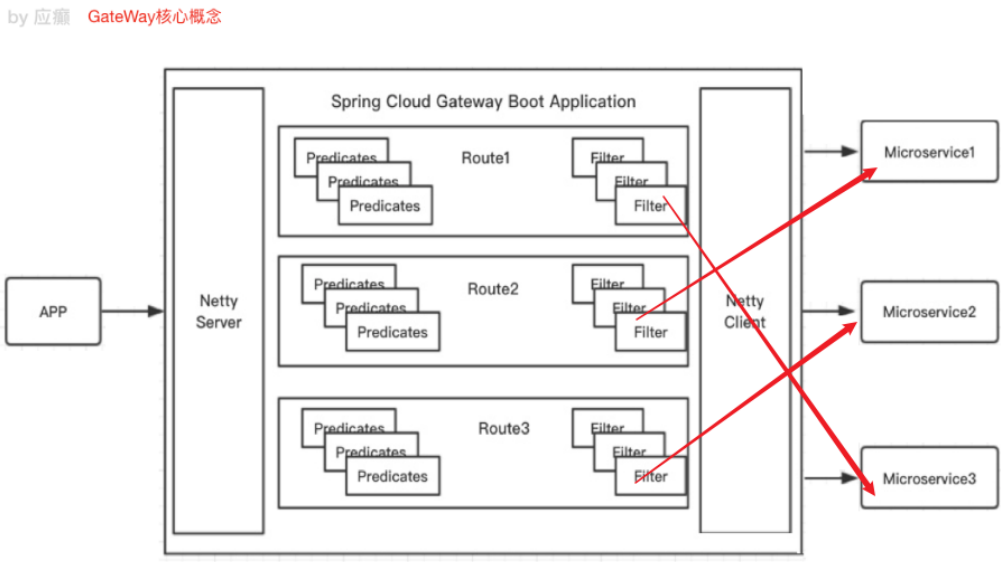
```yaml
logging:
    level:
       # Feign日志只会对日志级别为debug的做出响应
       com.lagou.edu.controller.service.ResumeServiceFeignClient: debug
```

GateWay网关组件

我们学习的GateWay-->Spring Cloud GateWay（它只是众多网关解决方案中的一种）

Spring Cloud GateWay 是 Spring Cloud 的一个全新项目，目标是取代 Netflix Zuul，它基于 Spring5.0+SpringBoot2.0+WebFlux（基于高性能的Reactor模式响应式通信框架Netty，异步非阻塞模 型）等技术开发，性能高于Zuul，官方测试，**GateWay是Zuul的1.6倍（GateWay是异步非阻塞，Zuul1.X 是阻塞式，2.X基于Netty）**，旨在为微服务架构提供一种简 单有效的统一的API路由管理方式。 Spring Cloud GateWay不仅提供统一的路由方式（反向代理）并且基于 Filter(定义过滤器对请求过滤， 完成一些功能) 链的方式提供了网关基本的功能，例如：**鉴权、流量控制、熔断、路径重写、日志监控 等。**





其中，Predicates**断言就是我们的匹配条件**，而**Filter就可以理解为一个无所不能的拦截器**，有了这两个元素，结合目标URL，就可以实现一个具体的路由转发。

GateWay核心逻辑：路由转发+执行过滤器链

过滤器分请求之前（pre）和请求之后（post）

Filter在“pre”类型过滤器中可以做参数校验、权限校验、流量监控、日志输出、协议转换等，在“post”类型的过滤器中可以做响应内容、响应头的修改、日志的输出、流量监控等。

- 路由（route）： 网关最基础的部分，也是网关比较基础的工作单元。路由由一个ID、一个目标URL（最终路由到的地址）、一系列的断言（匹配条件判断）和Filter过滤器（精细化控制）组 成。如果断言为true，则匹配该路由。
- 断言（predicates）：参考了Java8中的断言java.util.function.Predicate，开发人员可以匹配Http 请求中的所有内容（包括请求头、请求参数等）（类似于nginx中的location匹配一样），**如果断言与请求相匹配则路由。**

- 过滤器（filter）：一个标准的Spring webFilter，使用过滤器，**可以在请求之前或者之后执行业务逻辑。**

GateWay使用：

引入依赖：

```
<!--GateWay 网关-->
 <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
 </dependency>
```

```
spring:
  application:
    name: lagou-service-gateway
  cloud:
    gateway:
      routes: # 路由可以有多个
        - id: service-user-8080 # 服务1 我们自定义的路由 ID
          uri: lb://lagou-service-user #动态路由的配置，路由地址，和服务的spring.application.name一致（路由匹配后访问的地址）
          predicates: # 断言：路由条件
            - Path=/api/user/** #到该服务的地址
          filters:
            - StripPrefix=1
        - id: service-code-8081 # 服务2
          uri: lb://lagou-service-code
          predicates:
            - Path=/api/code/**
          filters:
            - StripPrefix=1
        - id: service-code-8082 # 服务3
          uri: lb://lagou-service-emailw
          predicates:
            - Path=/api/email/**
          filters:
            - StripPrefix=1
```

fliter定义（全局）：

```
@Component
public class IPFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain){
    }

    @Override
    public int getOrder() {
        return 0;
    }
}
```

fliter定义（局部）：https://www.cnblogs.com/wangjunwei/p/12898780.html

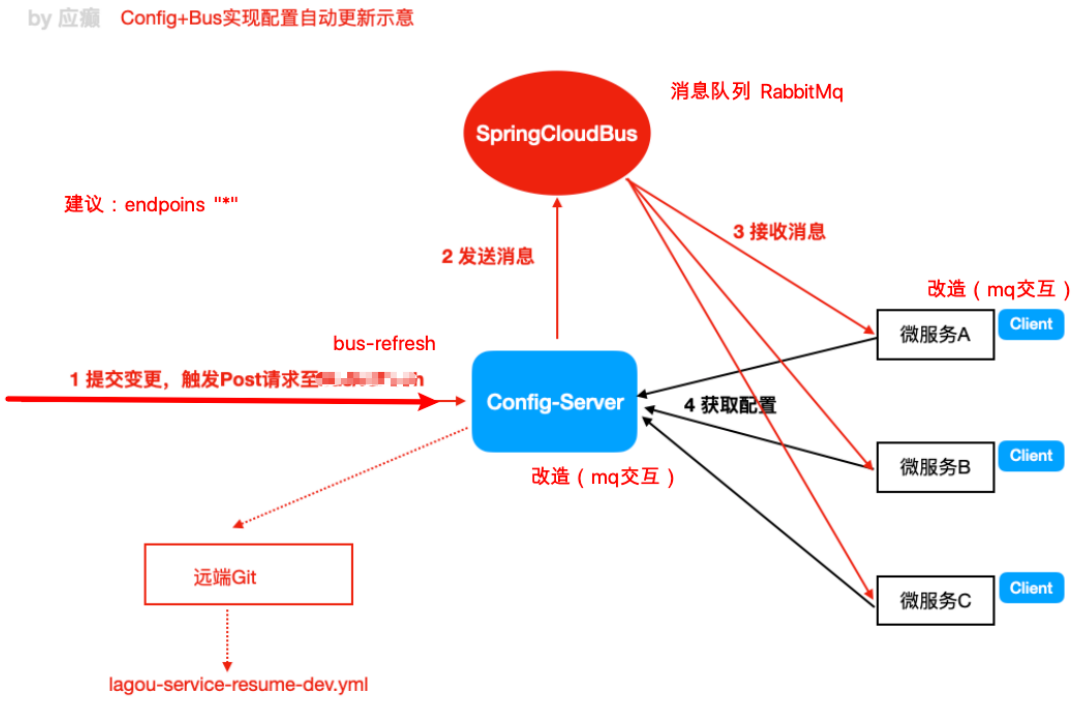
```java
@Component
public class IPFilter extends AbstractGatewayFilterFactory {

}
```

GateWay高可用配置（使用nginx）：

```
#配置多个GateWay实例
upstream gateway {
 server 127.0.0.1:9002;
 server 127.0.0.1:9003;
}
location / {
 proxy_pass http://gateway;
}
```

Spring Cloud Config 分布式配置中心



Config Server配置：

导入依赖：

```xml
<dependencies>
    <!--eureka client 客户端依赖引入-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eurekaclient</artifactId>
    </dependency>
    <!--config配置中心服务端-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
</dependencies>
```

启动类加入注解：

```
@EnableConfigServer // 开启配置服务器功能
```

application.yml配置：

```yaml
spring:
    application:
        name: lagou-service-autodeliver
    cloud:
        config:
            server:
                git:
                    uri: https://github.com/5173098004/lagou-config-repo.git #配置git服务地址
                    username: 517309804@qq.com #配置git用户名
                    password: yingdian12341 #配置git密码
                    search-paths:
                        - lagou-config-repo
            # 读取分支
            label: master
# springboot中暴露健康检查等断点接口
management:
     endpoints:
        web:
            exposure:
                include: "*"
 # 暴露健康接口的细节
 endpoint:
    health:
        show-details: always
```

Config Client配置：

添加依赖：

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

添加注解：

```
@RefreshScope //Client客户端使用到配置信息的类上添加该注解
```

bootstrap.yml：bootstrap.yml是系统级别的，优先级比application.yml高

```yaml
management:
    endpoints:
        web:
            exposure:
                include: refresh
#也可以暴露所有的端口
management:
    endpoints:
        web:
            exposure:
                include: "*"
```

SpringCloud和Dubbo的区别

Dubbo 使用的是 RPC 通信，二进制传输，占用带宽小；

Spring Cloud 使用的是 HTTP RESTFul 方式

RPC 和 Http的区别：

- RPC速度比http快，虽然底层都是TCP，但是http协议的信息往往比较臃肿（RPC服务基于TCP/IP协议（传输层）；HTTP服务基于HTTP协议（应用层））
- RPC使用较为复杂，http相对比较简单
- 灵活性来看，http更胜一筹，因为它不关心实现细节，跨平台、跨语言。

两者都有不同的使用场景：

- 如果对效率要求更高，并且开发过程使用统一的技术栈，那么用RPC还是不错的。
- 如果需要更加灵活，跨语言、跨平台，显然http更合适（微服务，更加强调的是独立、自治、灵活。而RPC方式的限制较多，因此微服务框架中，一般都会采用基于Http的Rest风格服务。）

SpringCloud和Dubbo的通讯性能比较：

线程数	Dubbo	Spring Cloud
10线程	2.75	6.52
20线程	4.18	10.03
50线程	10.3	28.14
100线程	20.13	55.23
200线程	42	110.21

SpringCloud和Dubbo的组件比较：

核心要素	Dubbo	Spring Cloud
服务注册中心	Zookeeper、Redis	Eureka
服务调用方式	RPC	REST API
服务网关	无	Zuul、GateWay

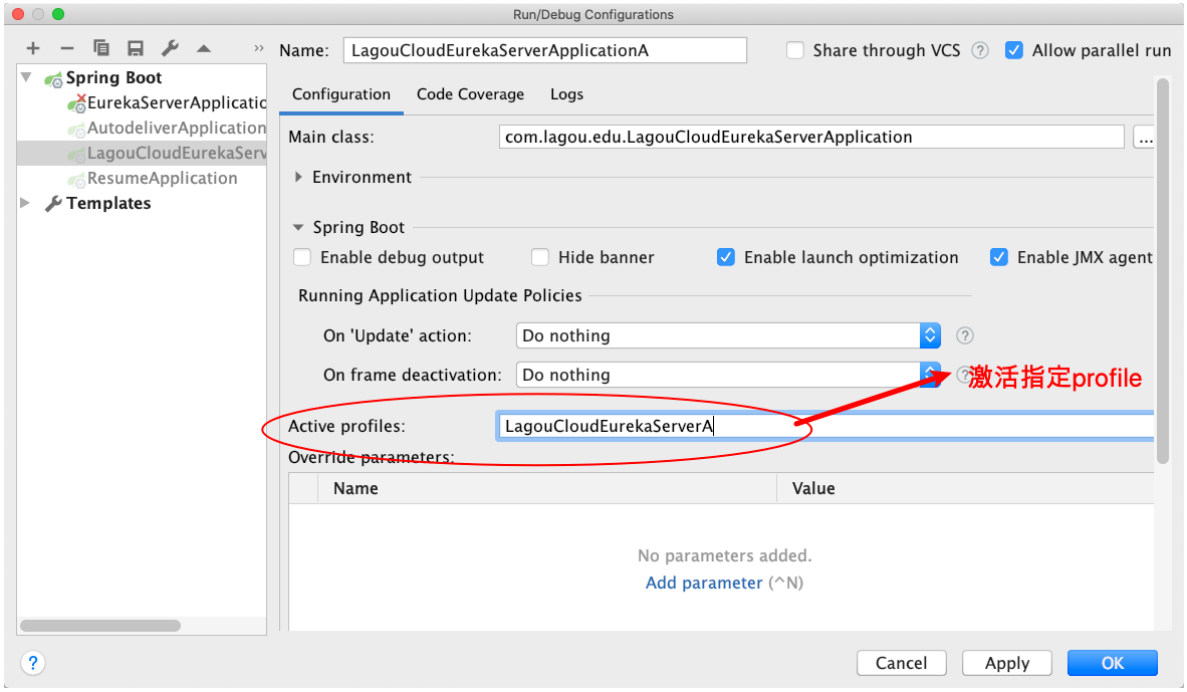
核心要素	Dubbo	Spring Cloud
断路器	不完善	Hystrix
分布式配置	无	Spring Cloud Config
分布式追踪系统	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
批量任务	无	Spring Cloud Task

springcloud的yml配置讲解：

```yaml
spring:
 profiles: LagouCloudEurekaServerA
spring:
 profiles: LagouCloudEurekaServerB
```

profiles的作用：

比如在同一个yml配置了两个profiles，在启动时要指定名字，如下图（其实一般都是配两个工程）




```yaml
server:
  port: 8090
spring:
 application:
    name: lagou-cloud-eureka-server # 应用名称，会在Eureka中作为服务的id标识
eureka:
 instance:
    hostname: localhost #下面的eureka的defaultZone有用到该参数
    # 租约续约间隔时间，默认30秒
    lease-renewal-interval-in-seconds: 30
```

```
    # 租约到期时间，默认90秒
    lease-expiration-duration-in-seconds: 90
 client:
    service-url: # 客户端与EurekaServer交互的地址，如果是集群，也需要写其它Server的地址
     # 每隔多久拉取一次服务列表
    registry-fetch-interval-seconds: 30
        defaultZone: http://${eureka.instance.hostname}:${server.port}:@project.version@/eureka/
    register-with-eureka: false # 是否注册自己到eureka
    fetch-registry: false #是否从Eureka Server获取服务信息,默认为true，置为false
```

eureka.instance.hostname的作用：

如果没有配置该参数，则eureka.client.service-url.defaultZone **默认**将使用http://${spring.application.name}:${server.port}/eureka/，等于说配置了它eureka的注册地址就可以使用ip进行访问了，而不是spring.application.name访问，当然还有其他用途。

:@project.version@的作用：

该配置为配置版本号，便于多版本管理，路径如下图：

面试题：

微服务，多服务之间频繁调用导致系统接口超时，怎么优化?

1. 从设计上解决，或者使用cache减少频繁调用
2. 增加熔断、降级、限流等措施，保证服务高可用，防止服务雪崩
3. 使用异步方式例如MQ进行削峰
4. 加机器，加配置