Spring Cloud（2023/11）

一、项目准备

1）技术选型

spring-cloud：2022.0.4

spring-boot：3.0.2

spring-cloud-alibaba：2022.0.0.0

2）maven的dependencyManagement 和 dependencies

在Maven中，`<dependencyManagement>` 和 `<dependencies>` 是两个关键的元素，用于管理和定义项目的依赖关系。让我们分别讨论它们的作用和用法：

1. `<dependencyManagement>` 元素：

`<dependencyManagement>` 元素用于**集中管理项目中所有模块的依赖版本。**通常，当项目中有多个模块时，可能会有一些共享的依赖，而这些依赖的版本可能在不同的模块中会有所不同。通过使用 `<dependencyManagement>`，可以在父项目的顶层指定所有模块中共享的依赖及其版本，从而确保它们在整个项目中的一致性。

示例：

```
<project>
    ...
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.example</groupId>
                <artifactId>common-library</artifactId>
                <version>1.0.0</version>
            </dependency>
            <!-- 可以在这里定义其他共享的依赖 -->
        </dependencies>
    </dependencyManagement>
    ...
</project>
```

然后，在子模块中，可以直接引用这些依赖，而不需要指定版本号：

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.example</groupId>
            <artifactId>common-library</artifactId>
        </dependency>
        <!-- 其他模块特定的依赖 -->
    </dependencies>
    ...
</project>
```

2. `<dependencies>` 元素：

`<dependencies>` 元素用于声明项目的依赖关系。在这个元素中，你指定项目所需的各种库、框架和其他依赖项，并在其中指定相应的坐标（groupId、artifactId、version等）。

示例：

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.1</version>
        </dependency>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>30.1-jre</version>
        </dependency>
        <!-- 其他依赖项 -->
    </dependencies>
    ...
</project>
```

在`<dependencies>`中声明的依赖项将被Maven用于构建项目时自动下载所需的库文件。

总体来说，`<dependencyManagement>` 用于集中管理项目中的依赖版本，而 `<dependencies>` 用于声明项目的实际依赖项。当两者一起使用时，可以实现更好的依赖管理和版本控制。

二、服务注册中心

1）Eureka

1.介绍

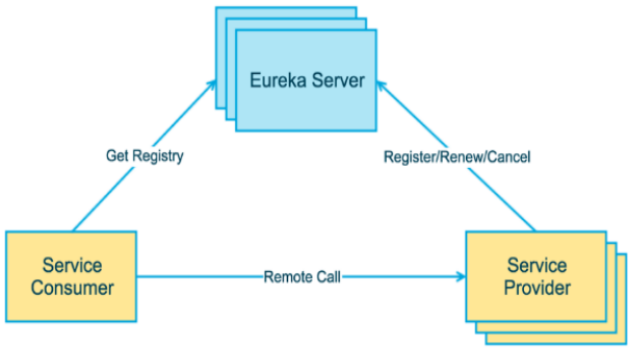
Eureka 是 Netflix 开源的一款**服务发现和注册工具**，用于**构建和管理微服务架构中的服务。**它允许应用程序在一个分布式系统中发现其他服务的位置和元数据，并且它还提供了一种简单的方式来实现**负载均衡和故障转移。**Eureka 的设计目标是**高可用性、可伸缩性和易用性。**

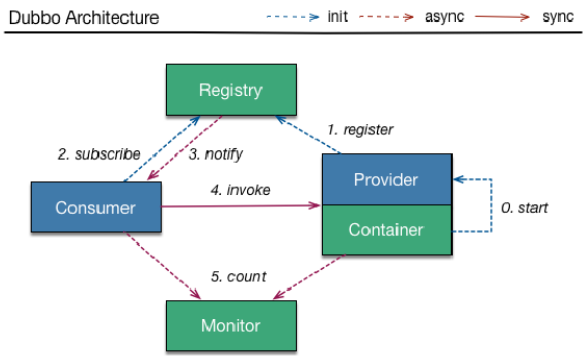
以下是一些 Eureka 的关键特性和概念：

1. **服务注册与发现：** 微服务可以在启动时向 Eureka 注册自己，包括服务的实例 ID、主机和端口等信息。其他服务可以查询 Eureka 服务器以发现可用的服务实例。
2. **客户端负载均衡：** Eureka 提供了内置的客户端负载均衡机制，可以根据服务实例的健康状态和负载情况，智能地分发请求到不同的实例。
3. **自我保护机制：** Eureka 包含一种自我保护机制，可以防止因为网络分区或其他原因导致的整个系统的崩溃。当 Eureka 服务器在一段时间内未收到心跳时，它将保护自己的实例信息，确保服务的基本可用性。
4. **区域感知和多区域部署：** Eureka 支持多区域的部署，可以根据服务的地理位置进行注册和发现。这对于构建分布式系统的全球性部署非常有用。
5. **可插拔的存储后端：** Eureka 允许使用不同的存储后端，包括内存、数据库等。这使得 Eureka 可以根据实际需求进行定制。
6. **Spring Cloud 集成：** Eureka 是 Spring Cloud 生态系统的一部分，可以方便地与其他 Spring Cloud 组件集成，如 Ribbon（客户端负载均衡）、Feign（声明式 REST 客户端）等。

在微服务架构中，Eureka 被广泛应用于服务注册和发现的场景，它为构建弹性、可伸缩的分布式系统提供了一个基础设施组件。请注意，由于 Netflix 在后续已经宣布停止对 Eureka 的更新和维护，因此在选择服务注册和发现的解决方案时，您可能还需要考虑其他替代方案，如 Consul、etcd 等。

下左图是Eureka系统架构，右图是Dubbo的架构，请对比





2.服务治理和服务注册

1. 服务治理（Service Governance）：

服务治理是一种**管理和控制微服务架构中各种服务的方法。**它涉及到确保服务的可用性、稳定性、可扩展性和安全性，以及监控和调整系统以适应不同的负载和需求。

关键概念和任务包括：

- **服务注册和发现：** 服务治理需要一种机制来注册服务并使其可被其他服务发现。这通常通过服务注册表实现，服务在启动时将自己注册到注册表，其他服务可以查询注册表来发现可用的服务。
- **负载均衡：** 管理服务实例之间的负载均衡，确保请求在多个实例之间均匀分布，提高系统的性能和可伸缩性。

- **容错和故障恢复：** 处理服务实例的故障，包括自动重新连接、重试机制和降级策略，以确保系统在面对故障时仍能保持可用。
- **安全性：** 管理服务之间的安全通信，包括身份验证、授权和数据加密，以确保系统的安全性。
- **监控和日志：** 实现对服务的监控、日志和度量，以便实时了解系统的状态，并在发生问题时进行调试和故障排除。

2. 服务注册（Service Registration）：

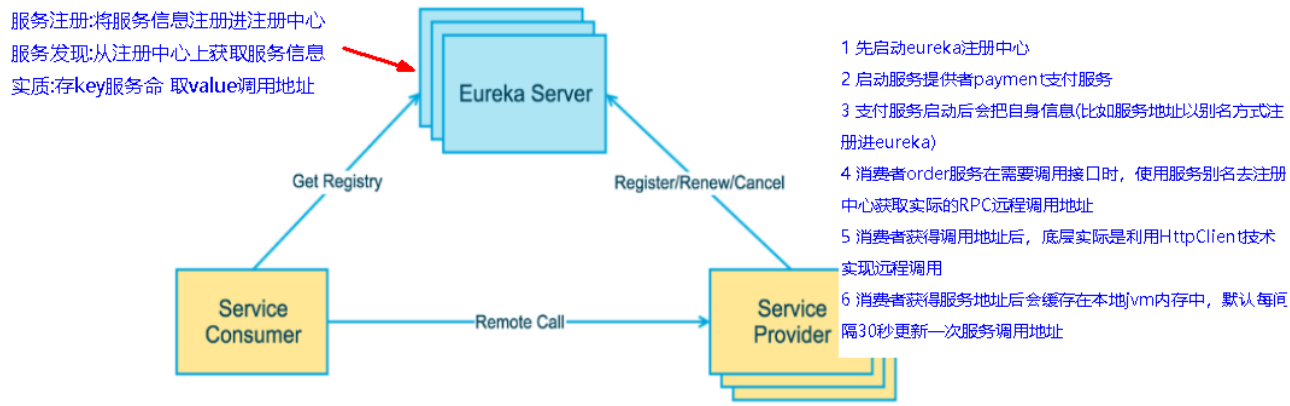
服务注册是服务治理的一个关键组成部分。它是指在一个分布式系统中，服务实例向一个中心化的服务注册表注册自己的信息，以便其他服务可以查询并发现可用的服务实例。

服务注册的关键特点和流程包括：

- **注册服务：** 在服务启动时，服务实例会向服务注册表注册自己的信息，包括服务名称、实例ID、IP地址、端口号等。
- **发现服务：** 其他服务需要调用某个服务时，它可以查询服务注册表以获取可用的服务实例信息，然后使用该信息进行通信。
- **心跳机制：** 注册的服务实例通常会周期性地发送心跳以告知注册表它们仍然存活。这有助于及时发现不可用的服务实例。
- **服务元数据：** 除了基本的地址信息外，服务注册表还可以包含与服务实例相关的其他元数据，如版本、环境、健康状态等。

服务注册和发现的实现可以借助于工具和平台，例如 Eureka、Consul、etcd 和 Kubernetes 等。这些工具提供了方便的方式来管理服务之间的关系，并确保系统在不断变化的环境中仍然稳定和可靠。

3.eureka的集群模式



Eureka的集群原理主要涉及到服务注册、服务发现、心跳机制以及集群之间的通信。以下是Eureka集群的一般工作原理：

1. **服务注册：** 微服务启动时，通过Eureka客户端向Eureka服务器注册自己。注册过程中，微服务提供了关于自己的元数据信息，包括服务名称、实例ID、IP地址、端口号等。Eureka服务器维护了一个服务注册表，记录了所有已注册的微服务实例。
2. **服务发现：** 其他微服务需要调用某个服务时，首先通过Eureka客户端向Eureka服务器发起查询请求。Eureka服务器返回相应服务的所有可用实例信息，包括它们的IP地址和端口号。
3. **心跳机制：** 微服务在注册后，会定期发送心跳请求给Eureka服务器，以表明自己仍然是活跃的。如果Eureka服务器在一定时间内没有收到某个微服务的心跳，它会将该实例标记为不可用，但并不立即将其从服务注册表中删除。这种自我保护机制有助于防止因网络问题或微服务瞬时故障而误删实例。

4. **集群通信：** Eureka服务器之间通过HTTP通信来共享服务注册表的信息。当一个Eureka服务器接收到服务注册或注销的请求时，它会将这个变更信息广播给所有集群中的其他Eureka服务器。这样，整个集群都能保持同步，确保每个Eureka服务器都包含完整的服务注册表。
5. **负载均衡和故障容错：** Eureka客户端在查询服务时，会从所有可用的Eureka服务器中选择一个进行查询。这样就实现了负载均衡，而且由于Eureka服务器之间保持了同步，即使某个Eureka服务器宕机，客户端仍然能够从其他可用的服务器中获取服务信息。

总体而言，Eureka的集群原理基于服务注册、服务发现、心跳机制和集群通信，通过这些机制，它能够在分布式环境中实现高可用性、负载均衡和故障容错。这种架构使得微服务能够动态地加入和退出，同时保持整个系统的稳定性。

4.eureka的自我保护机制

Eureka是Netflix开源的一款用于实现服务注册与发现的工具，它可以帮助构建基于微服务架构的应用程序。Eureka的自我保护机制是其一个重要特性，旨在应对网络分区或其他异常情况下的服务注册中心的问题。

自我保护机制的主要目标是确保服务注册中心（Eureka Server）在面临网络故障或其他问题时能够保持可用性，防止服务实例被误认为不可用而被剔除。在网络分区的情况下，服务实例可能因为无法与注册中心进行心跳检测而被标记为不健康。为了防止因网络问题导致的误判，Eureka引入了自我保护机制。

自我保护机制的工作原理如下：

1. **期望的心跳比例下降：** Eureka Server期望在一定的时间内收到每个服务实例的心跳。如果在这个时间内没有收到足够数量的心跳，Eureka Server就会认为发生了问题。
2. **启动自我保护模式：** 一旦发现心跳比例下降，Eureka Server会启动自我保护模式。在这个模式下，Eureka Server将保留所有注册的服务实例，不再剔除因为心跳超时而被标记为不健康的实例。
3. **自我保护模式期间的行为：** 在自我保护模式期间，Eureka Server会记录每个心跳失败的实例，并在日志中输出警告。这样，开发者可以注意到可能存在的问题。
4. **自我保护模式的退出：** 一旦Eureka Server检测到心跳比例恢复正常，它将退出自我保护模式，恢复正常的实例剔除策略。

通过自我保护机制，Eureka能够在面临一些网络故障或不稳定的环境中保持服务注册中心的可用性，确保服务实例的注册信息不会轻易被剔除。这对于微服务架构中的稳定性和可靠性非常重要。

关闭自我保护机制：

```yaml
eureka:
  server:
    enable-self-preservation: false # 关闭自我保护机制
```

5.使用

①服务提供者provider

- 主启动类：
  - 加上注释：@EnableDiscoveryClient

- controller
  - 可以注入：DiscoveryClient 来查看服务信息
- 配置类yml

```yaml
server:
  port: 8001

spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db2023?useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: vanky
  freemarker:
    cache: false

eureka:
  instance:
    instance-id: payment8001
    prefer-ip-address: true
  client:
    #表示是否将自己注册进EurekaServer默认为true。
    register-with-eureka: true
    #是否从EurekaServer抓取已有的注册信息，默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
    fetchRegistry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,http://eureka7002.com:7002/eureka
```

mybatis:

```
mapperLocations: classpath:mapper/*.xml
type-aliases-package: com.vanky.springcloud.entities    # 所有Entity别名类所在包
configuration:
  map-underscore-to-camel-case: true
```

```
•

##### ②eureka服务端

* 主启动类

  * 加上注释：@EnableEurekaServer

* 配置类yml
```

```yaml
server:
  port: 7001

eureka:
  instance:
    hostname: eureka7001.com #eureka服务端的实例名称
#  server:
#    enable-self-preservation: false # 关闭自我保护机制
  client:
    #false表示不向注册中心注册自己。
    register-with-eureka: 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
    fetch-registry: false
    service-url:
      #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
```

5.消费者

- 主启动类
  - 添加注释：@EnableDiscoveryClient
- 配置信息yaml

```yaml
server:
  port: 80

spring:
  application:
    name: cloud-order-service
  devtools:
    restart:
      enabled: true # 启动热部署
      additional-paths: src/main/java
      exclude: WEB-INF/**
  freemarker:
    cache: false

eureka:
  client:
    #表示是否将自己注册进EurekaServer默认为true。
    register-with-eureka: true
    #是否从EurekaServer抓取已有的注册信息，默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
    fetchRegistry: true
    service-url:
      defaultZone: http://localhost:7001/eureka
```

2）zookeeper

1.介绍

ZooKeeper（动物园管理员）是一个分布式协调服务，经常被用作微服务架构中的注册中心。微服务注册中心的主要作用是允许服务实例进行注册和发现，以便构建动态可伸缩的系统。以下是使用ZooKeeper作为微服务注册中心时的一些关键方面：

1. **服务注册：** 微服务在启动时向ZooKeeper注册自己的信息，包括服务名称、IP地址、端口等。这通常涉及在ZooKeeper的命名空间中创建一个**临时节点**，该节点包含有关服务实例的信息。临时节点的一个重要特性是，如果服务实例异常退出，**相应的临时节点会被自动删除，从而实现自动注销。**
2. **服务发现：** 其他微服务实例可以查询ZooKeeper，以获取已注册的服务实例列表。这通常通过在ZooKeeper上设置监听器，以便在服务实例注册或注销时得到通知。服务发现机制使得微服务能够动态地找到和调用其他服务，而无需硬编码服务实例的位置。
3. **节点的组织结构：** ZooKeeper通常以**树状结构组织数据**，每个注册的服务实例都对应于树中的一个节点。例如，可以使用服务名称作为根节点，其下有每个服务实例的子节点。
4. **分布式协调：** ZooKeeper提供了一致性和分布式协调的能力，这是构建可靠的微服务体系结构所必需的。它使用ZAB（ZooKeeper Atomic Broadcast）协议来实现**数据的一致性。**ZooKeeper集群中的多个节点协同工作，**确保数据的强一致性。**
5. **容错性：** ZooKeeper的设计**考虑了分区容错。**即使在网络分区的情况下，ZooKeeper仍然能够提供可用的服务。这对于构建可靠的分布式系统至关重要。
6. **ZooKeeper客户端：** 在微服务应用中，需要使用ZooKeeper客户端库与ZooKeeper进行交互。一些流行的ZooKeeper客户端库包括Apache Curator、ZooKeeper自带的Java客户端等。
7. **安全性：** 对于微服务环境中的安全性，需要确保与ZooKeeper的通信是安全的。此外，可以采取适当的措施来保护服务注册信息的隐私。

使用ZooKeeper作为微服务注册中心的好处包括其稳定性、高可用性、分布式特性以及成熟的生态系统。然而，在使用之前，需要仔细考虑和规划ZooKeeper集群的配置、网络拓扑，以及应用程序的容错和性能方面的需求。

> ZooKeeper和Eureka都是用于微服务架构中服务注册与发现的工具，它们有一些不同的设计理念和特点。以下是ZooKeeper相较于Eureka的一些优势：
>
> 1. **一致性和分区容错：** ZooKeeper是一个强一致性的分布式协调服务，它的设计目标之一是保证数据的一致性。ZooKeeper通过使用ZAB（ZooKeeper Atomic Broadcast）协议来实现这种一致性。这使得ZooKeeper在分布式环境中更容易处理网络分区和节点故障，确保系统的稳定性和可用性。
> 2. **更广泛的用途：** ZooKeeper并不仅仅用于服务注册与发现，它还可以用于分布式锁、配置管理、领导选举等场景。这使得ZooKeeper成为一个更为通用的分布式协调工具，而Eureka更专注于服务注册与发现。
> 3. **数据持久性：** ZooKeeper的数据是持久性的，即使在服务实例退出或重启后，注册信息仍然可以保留。这有助于确保在系统重新启动后，注册中心的数据仍然可用。
> 4. **严格的一致性：** ZooKeeper的一致性要求相对较高，这意味着在集群中的各个节点之间始终保持相同的视图。这对于某些需要强一致性的场景非常重要，比如分布式事务。
> 5. **分布式协调原语：** ZooKeeper提供了一组分布式协调原语，如锁、队列等，这些原语可以用于构建更复杂的分布式系统。Eureka相对而言更专注于服务注册与发现，不提供类似的原语。
> 6. **成熟性和稳定性：** ZooKeeper是一个经过时间验证、成熟稳定的项目，有一个活跃的社区和大量的使用案例。这使得它在企业级应用中更受欢迎，尤其是对于那些对一致性和稳定性要求较高的场景。

> 总体而言，ZooKeeper在一致性、分布式协调和更广泛用途方面具有优势。然而，Eureka在一些特定的场景中可能更简单易用，特别是在构建基于Spring Cloud的微服务架构时。选择使用哪个取决于具体的需求和项目的特点。

2.使用

①服务提供者

- 配置

```yaml
server:
  port: 8004

spring:
  application:
    name: cloud-provider-payment
  cloud:
    zookeeper:
      connect-string: 192.168.200.142:2181
```

- 主启动类：

```java
@SpringBootApplication
@EnableDiscoveryClient
public class PaymentMain8004 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain8004.class,args);
    }
}
```

②消费者

- 配置

```yaml
server:
  port: 80

spring:
  application:
    name: cloud-consumer-order
  cloud:
    zookeeper:
      connect-string: 192.168.200.142:2181
```

- controller：注意访问地址

```java
@RestController
@Slf4j
public class OrderZKController {
```

```
    public static final String INVOKE_URL = "http://cloud-provider-payment";

    @Resource
    private RestTemplate restTemplate;

    @GetMapping("/consumer/payment/zk")
    public String paymentInfo(){
        String res = restTemplate.getForObject(INVOKE_URL + "/payment/zk", String.class);
        log.info("消费者调用支付服务（zookeeper）---->result：" + res);

        return res;
    }

}
```

- 配置类

```
@Configuration
public class ApplicationContextConfig {

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

}
```

3）Consul

1.介绍

Spring Cloud Consul是Spring Cloud生态系统的一部分，用于在微服务架构中提供服务注册和发现功能。它使用HashiCorp Consul作为后端服务注册中心，为分布式系统提供了一种方便的方式来进行服务的注册、发现和配置。

下面是关于Spring Cloud Consul的一些重要概念和特点：

1. **服务注册与发现：** Spring Cloud Consul允许微服务将自己注册到Consul中心，并且其他微服务可以通过Consul中心发现已注册的服务。这有助于构建动态的、可伸缩的微服务架构。
2. **Consul作为后端：** Spring Cloud Consul使用HashiCorp Consul作为其默认的服务注册中心。Consul是一个开源的工具，提供了分布式服务注册、发现、健康检查等功能。它是一个高度可扩展的工具，适用于构建大规模的微服务系统。
3. **Consul配置中心：** Spring Cloud Consul还提供了配置中心的功能，可以将应用程序的配置存储在Consul中心，并实现动态配置的更新。这使得微服务能够在运行时动态获取配置信息，而不需要重新启动应用。
4. **健康检查：** Consul允许微服务注册健康检查，以确保只有健康的服务实例被路由到。Spring Cloud Consul利用Consul的健康检查机制来提供对服务状态的监控和故障转移支持。
5. **Spring Cloud集成：** Spring Cloud Consul与Spring Cloud的其他组件无缝集成，比如通过`@EnableDiscoveryClient`注解启用服务注册与发现。它还可以与Spring Cloud的其他功能，如断路器、配置中心等一起使用。

6. **服务Mesh支持：** Consul Connect是Consul的一部分，它提供了一种服务Mesh的解决方案，可以用于微服务之间的安全通信。Spring Cloud Consul可以与Consul Connect集成，以提供更复杂的网络安全性。
7. **分布式锁：** Consul还支持分布式锁的实现，Spring Cloud Consul可以利用这个功能来实现分布式系统中的互斥操作。

总体而言，Spring Cloud Consul为开发者提供了一个强大的工具，以在微服务架构中实现服务注册、发现和配置管理。通过与Consul的集成，它为构建可靠的、分布式的微服务体系结构提供了一系列功能。

2.使用

①服务提供者

- pom

```
<!--SpringCloud consul-server -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-consul-discovery</artifactId>
</dependency>
```

- yml

```
###consul服务端口号
server:
  port: 8006

spring:
  application:
    name: consul-provider-payment
  ####consul注册中心地址
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        #hostname: 127.0.0.1
        service-name: ${spring.application.name}
```

- 主启动类加上注释@EnableDiscoveryClient

②服务消费者

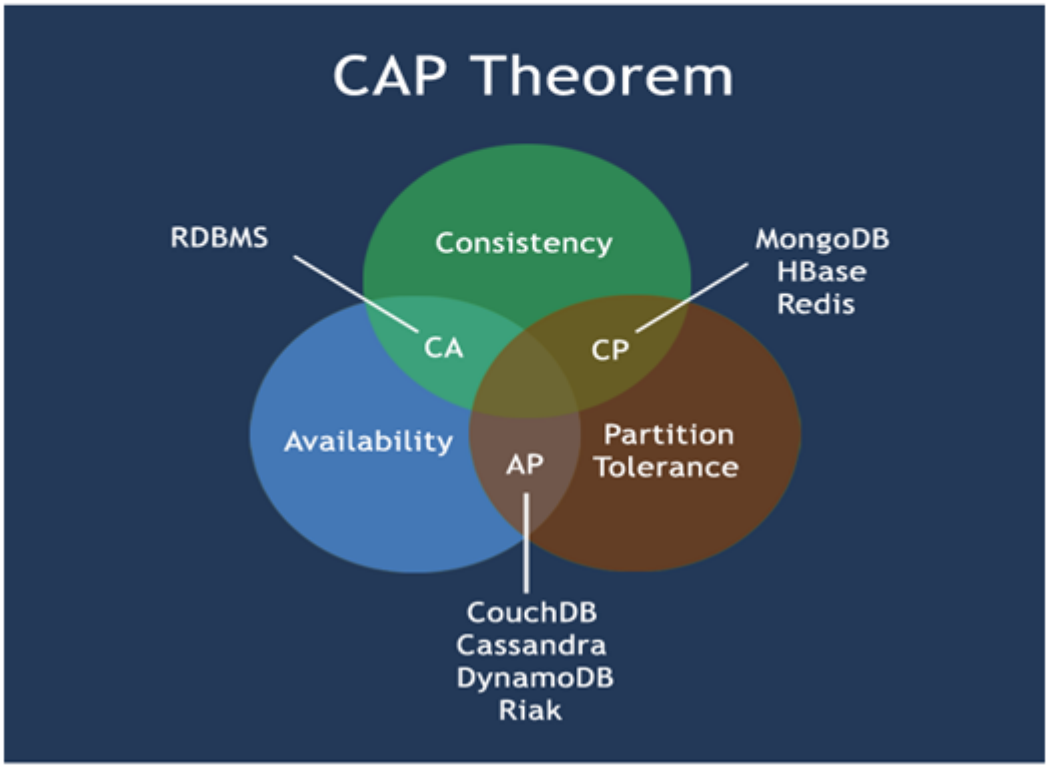
- yaml、pom和上面一样，改改服务名和端口号

4）三种注册中心总结

1.异同点

Eureka、ZooKeeper、和Consul都是用于微服务架构中的服务注册与发现的工具，但它们在设计理念、特性以及适用场景上有一些异同点。

组件名	语言	CAP	服务健康检查	对外暴露接口	Spring Cloud集成
Eureka	Java	AP	可配支持	HTTP	已集成
Consul	Go	CP	支持	HTTP/DNS	已集成
Zookeeper	Java	CP	支持	客户端	已集成



异同点：

1. **一致性和分区容错：**
   - **Eureka：** Eureka设计上更侧重于AP（可用性和分区容错）的一致性模型。在面临网络分区时，Eureka倾向于保持可用性，采用自我保护机制，不容易剔除服务实例。
   - **ZooKeeper：** ZooKeeper追求强一致性和CP（一致性和分区容错）模型。它保证了在整个集群中的节点间的一致性，适用于一些对一致性要求较高的场景。
   - **Consul：** Consul采用CP模型，强调一致性和分区容错。它在服务注册和发现中提供一致性保证，以确保系统的稳定性。
2. **数据持久性：**
   - **Eureka：** Eureka默认情况下服务实例的注册信息是有限时的，如果服务实例不再发送心跳，注册信息会过期。这意味着在服务重启后需要重新注册。
   - **ZooKeeper：** ZooKeeper的数据是持久性的，即使服务实例退出或重启，注册信息仍然会保留在ZooKeeper中。
   - **Consul：** Consul默认情况下也支持持久性的注册信息。
3. **功能广泛性：**
   - **Eureka：** Eureka更专注于服务注册与发现，相对较简单，适用于一些中小型项目。

- **ZooKeeper：** ZooKeeper是一个通用的分布式协调服务，除了服务注册与发现外，还提供了分布式锁、队列等原语，适用于更广泛的用例。
- **Consul：** Consul综合了服务注册与发现、健康检查、分布式配置等功能，更全面地支持构建微服务系统。

4. **多数据中心支持：**
   - **Eureka：** Eureka的多数据中心支持较弱，需要在不同数据中心间手动配置。
   - **ZooKeeper：** ZooKeeper通过WAN感知的方式提供多数据中心之间的服务发现和同步。
   - **Consul：** Consul天生支持多数据中心，具有较好的跨数据中心扩展性。

共同点：

1. **服务注册与发现：** 所有三个工具都提供了服务注册与发现的基本功能，允许微服务注册自己的信息，并通过查询来发现其他服务。
2. **健康检查：** Eureka、ZooKeeper、和Consul都支持对服务实例进行健康检查，以确保只有健康的实例被路由到。
3. **多语言支持：** 它们都提供了多种语言的客户端库，以方便在各种编程语言中使用。
4. **安全性：** 这三个工具都支持通过TLS/SSL来加密服务之间的通信，提供一定程度的安全性。
5. **Spring Cloud集成：** Eureka和Consul都与Spring Cloud有较好的集成支持。

选择使用哪个服务注册中心，取决于具体的需求、项目的规模和复杂度，以及对一致性、可用性等方面的要求。

2.CAP定理

CAP 定理，又称为 Brewer's Theorem，是由计算机科学家Eric Brewer于2000年提出的一个理论。CAP 定理描述了在分布式系统设计中三个重要目标之间的不可避免的权衡关系。这三个目标分别是一致性（Consistency）、可用性（Availability）、和分区容错性（Partition Tolerance）。

CAP 定理的表述：

在一个分布式系统中，不可能同时满足以下三个条件：

1. **一致性（Consistency）：** 所有节点在同一时刻看到的数据是相同的。在强一致性系统中，任何一个读操作都能够读取到最新的写操作的结果。
2. **可用性（Availability）：** 系统能够对客户的请求做出响应，而且这个响应是在有限的时间内完成的。高可用性的系统能够保证服务在大多数时间内都是可用的。
3. **分区容错性（Partition Tolerance）：** 系统能够在网络分区的情况下仍然保持正常运行。网络分区是指系统中的节点由于网络故障等原因无法互相通信。

CAP 定理的权衡关系：

CAP 定理指出，在发生网络分区的情况下，分布式系统只能同时满足一致性和可用性中的一个。这是因为在网络分区的情况下，不同节点之间可能无法达成一致的共识，因此在追求一致性的同时可能导致系统不可用。

具体来说：

- **CA系统（Consistent and Available）：** 在没有网络分区的情况下，系统追求一致性和可用性。这意味着所有节点在同一时刻看到的数据是相同的，并且系统能够保证服务的可用性。然而，在发生网络分区时，系统可能会牺牲可用性以保持一致性。
- **CP系统（Consistent and Partition Tolerant）：** 系统追求一致性和分区容错性。这意味着系统会在网络分区的情况下保持一致性，但可能会导致一些节点无法响应请求。

- **AP系统（Available and Partition Tolerant）：** 系统追求可用性和分区容错性。这意味着系统会在网络分区的情况下保持可用性，但可能导致数据一致性的延迟。

实际上，不同的系统在设计时会根据具体的应用场景和需求做出不同的权衡选择。 CAP 定理为分布式系统提供了一个理论基础，帮助开发者更好地理解和权衡系统设计中的重要因素。

三、负载均衡Ribbon

1）介绍

Ribbon是Netflix开源的一个**基于HTTP和TCP客户端的负载均衡器。**它主要用于在微服务架构中，通过在**服务消费方部署 Ribbon**，实现对多个服务提供方的请求负载均衡，提高系统的可用性和弹性。

以下是关于Ribbon的一些重要特点和功能：

1. **负载均衡：** Ribbon提供了多种负载均衡策略，如轮询、随机、权重等，用于在服务消费方选择合适的服务实例进行请求分发。这有助于避免单一服务实例的过载，提高整个系统的性能和可用性。
2. **服务列表和服务发现：** Ribbon通过服务注册中心（如Eureka、Consul等）或配置文件中的服务列表，获取可用的服务实例信息。这使得Ribbon能够动态地感知到服务的变化，实现服务的发现和注册。
3. **重试机制：** Ribbon内置了请求重试的机制，可以在发生故障时尝试重新发送请求，增加了系统的稳定性。
4. **断路器（Circuit Breaker）：** Ribbon集成了断路器模式，可以在检测到某个服务实例故障时，暂时中断对该实例的请求，防止故障的服务实例继续对系统产生负面影响。
5. **自定义规则：** 开发者可以通过自定义规则来扩展Ribbon的负载均衡行为。这使得可以根据实际需求，实现更复杂的负载均衡策略。
6. **集成Spring Cloud：** Ribbon是Spring Cloud中的一部分，与Spring Cloud的其他组件（如Eureka、Zuul等）无缝集成。在Spring Cloud应用中，通过使用`@LoadBalanced`注解，可以让RestTemplate和Feign客户端具备Ribbon的负载均衡能力。
7. **多协议支持：** Ribbon支持多种协议，包括HTTP、TCP、UDP等，使得它更加灵活地适应不同的应用场景。
8. **自适应负载均衡：** Ribbon支持根据服务实例的响应时间和状态来自适应地调整负载均衡策略，确保对性能较好的实例进行更多的请求分发。

总体而言，Ribbon是一个强大的客户端负载均衡工具，通过其丰富的特性，能够帮助开发者在微服务架构中实现动态的负载均衡，提高系统的可用性和性能。然而需要注意的是，Netflix已经宣布停止Ribbon项目的更新和维护，未来可能会逐步淘汰，因此在新的项目中可能需要考虑使用Spring Cloud LoadBalancer等替代方案。

与nginx的区别?

Ribbon和Nginx都是负载均衡的工具，但它们有一些重要的区别，主要体现在以下几个方面：

1. **位置：**

- **Ribbon：** 是一个在应用层（即七层负载均衡）的负载均衡器，通常用于在微服务架构中进行服务间的负载均衡。Ribbon通常作为客户端的库存在，嵌入在服务消费方的应用中，由客户端负责负载均衡的实现。
- **Nginx：** 是一个在网络层（即四层负载均衡）的负载均衡器，通常用于在服务器之间进行负载均衡。Nginx既可以作为反向代理服务器，也可以作为负载均衡器，集中管理对一组服务器的请求分发。

2. **负载均衡粒度：**

- **Ribbon：** 通过**在客户端实现负载均衡**，能够根据服务实例的状态、权重等信息进行动态的负载均衡决策。适用于微服务架构，更关注于服务之间的负载均衡。
- **Nginx：** **在网络层进行负载均衡**，主要基于IP地址和端口号的分发，对上层协议（如HTTP、HTTPS）不够了解，无法像Ribbon那样深入到应用层进行更细粒度的负载均衡。

3. **应用场景：**

- **Ribbon：** 更适用于微服务架构中，能够通过与服务注册中心（如Eureka）集成，动态地发现和分发请求到各个服务实例。适用于基于Spring Cloud的微服务体系结构。
- **Nginx：** 适用于传统的Web应用负载均衡，反向代理，以及静态文件服务等场景。Nginx的强大性能和灵活性使其成为处理大量并发请求的理想选择。

4. **可配置性和扩展性：**

- **Ribbon：** 通过配置文件或代码进行负载均衡的配置，有一些默认的负载均衡算法和策略，但相比Nginx，配置项相对较少。在微服务框架中，通常与Spring Cloud集成，使用起来比较简单。
- **Nginx：** 具有非常丰富的配置选项，可以通过配置文件进行高度定制化。Nginx支持多种负载均衡算法，同时支持反向代理、缓存、SSL终端等多种功能。

5. **部署和管理：**

- **Ribbon：** 部署在应用内，需要在每个服务消费方进行配置和集成。管理相对分散，每个微服务都需要维护自己的Ribbon配置。
- **Nginx：** 部署在独立的服务器上，中心化管理。可以作为独立的负载均衡服务器，通过配置文件统一管理所有服务的负载均衡策略。

总体而言，Ribbon更适用于微服务架构，而Nginx更适用于传统的Web应用负载均衡和反向代理场景。在实际应用中，两者可以根据具体的需求和架构选择合适的负载均衡解决方案。同时，需要注意的是，Netflix已经宣布停止Ribbon项目的更新和维护，因此在新的项目中可能需要考虑使用Spring Cloud LoadBalancer等替代方案。

2）负载均衡的规则

Ribbon作为一个客户端负载均衡库，提供了多种负载均衡规则，可以根据实际需求进行配置。以下是一些常见的Ribbon负载均衡规则：

1. **轮询规则（Round Robin Rule）：**
   - 描述： 请求按顺序轮询地分配到每个服务实例上。
   - 配置方式： 默认规则，无需特殊配置。

   > 原理：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标 ， 每次服务重启动后rest接口计数从1开始。

2. **随机规则（Random Rule）：**
   - 描述： 每次从所有服务实例中随机选择一个进行请求。
   - 配置方式： 在Ribbon的配置文件中设置规则为`RandomRule`。
3. **权重轮询规则（Weighted Round Robin Rule）：**
   - 描述： 根据服务实例的权重进行轮询分配请求，权重越高的实例获得更多的请求。
   - 配置方式： 在Ribbon的配置文件中设置规则为`WeightedResponseTimeRule`，并配置服务实例的权重。
4. **权重随机规则（Weighted Random Rule）：**

- 描述： 根据服务实例的权重进行随机分配请求，权重越高的实例被选中的概率越大。
- 配置方式： 在Ribbon的配置文件中设置规则为`WeightedResponseTimeRule`，并配置服务实例的权重。

5. **最小并发数规则（Best Available Rule）：**
   - 描述： 选择并发请求数最小的实例进行请求。适用于具有异构性的服务实例，能够避免请求集中在某一台实例上。
   - 配置方式： 在Ribbon的配置文件中设置规则为`BestAvailableRule`。
6. **性能权重规则（Availability Filter Rule）：**
   - 描述： 根据服务实例的性能指标进行加权，性能越好的实例获得更多的请求。
   - 配置方式： 在Ribbon的配置文件中设置规则为`AvailabilityFilteringRule`。
7. **响应时间加权规则（Response Time Weighted Rule）：**
   - 描述： 根据服务实例的平均响应时间进行加权，响应时间越短的实例获得更多的请求。
   - 配置方式： 在Ribbon的配置文件中设置规则为`WeightedResponseTimeRule`。
8. **重试规则（Retry Rule）：**
   - 描述： 在一定的重试时间内，尝试重新选择可用的服务实例。适用于网络环境不稳定的情况。
   - 配置方式： 在Ribbon的配置文件中设置规则为`RetryRule`。
9. **区域感知规则（Zone Avoidance Rule）：**
   - 描述： 根据服务实例的区域信息进行分配，避免请求跨越多个区域。
   - 配置方式： 在Ribbon的配置文件中设置规则为`ZoneAvoidanceRule`。
10. **自定义规则：**
    - 描述： 开发者可以实现自定义的负载均衡规则，根据具体的需求和业务逻辑来实现请求分发策略。
    - 配置方式： 实现`IRule`接口，并在Ribbon的配置文件中指定自定义规则。

这些规则提供了在不同场景下选择合适负载均衡策略的灵活性。通过合理选择负载均衡规则，可以根据实际需求来平衡服务实例的负载，提高系统的性能和可用性。

3）自定义规则

要自定义Ribbon的负载均衡规则，你需要实现`IRule`接口。`IRule`接口是Ribbon负载均衡规则的核心接口，定义了如何选择下一个要请求的服务实例的策略。下面是一个简单的例子，演示如何自定义一个负载均衡规则。

首先，创建一个实现`IRule`接口的类，例如：

```
import com.netflix.loadbalancer.ILoadBalancer;
import com.netflix.loadbalancer.IRule;
import com.netflix.loadbalancer.Server;

public class MyCustomRule implements IRule {

    private ILoadBalancer lb;

    @Override
    public Server choose(Object key) {
        // 在这里实现自定义的负载均衡逻辑
        // 可以根据实际需求选择具体的服务实例

        // 返回选择的服务实例
```

```
        return null;
    }

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

    @Override
    public ILoadBalancer getLoadBalancer() {
        return lb;
    }
}
```

在`choose`方法中，你可以根据自定义的负载均衡策略选择合适的服务实例，并将其返回。

然后，在Spring Boot应用的配置类中，使用`@RibbonClient`注解指定要使用的负载均衡规则。例如：

```
import org.springframework.cloud.netflix.ribbon.RibbonClient;
import org.springframework.context.annotation.Configuration;

@Configuration
@RibbonClient(name = "your-service-name", configuration = MyCustomRuleConfig.class)
public class RibbonConfig {
    // 这里的配置会覆盖掉默认的Ribbon配置
}
```

在上述代码中，`your-service-name`是你要进行负载均衡的服务的名称，而`MyCustomRuleConfig`是一个配置类，用于注入你自定义的负载均衡规则。

接着，在`MyCustomRuleConfig`类中，使用`@Bean`注解注入自定义的规则：

```
import org.springframework.context.annotation.Bean;

public class MyCustomRuleConfig {

    @Bean
    public IRule myCustomRule() {
        return new MyCustomRule();
    }
}
```

通过以上步骤，你就成功地自定义了Ribbon的负载均衡规则。在`choose`方法中，你可以根据具体的业务逻辑实现自定义的负载均衡策略，例如基于权重、性能、自定义标签等因素进行选择。

4）手写轮询

原理：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标

- 接口

```
public interface LoadBalancer {
    //获取当前有哪些注册的服务
    ServiceInstance instances(List<ServiceInstance> serviceInstances);
}
```

- 实现类

```
@Component
@Slf4j
public class MyLB implements LoadBalancer{
    //原子整数类，用于进行原子操作的整数
    private AtomicInteger atomicInteger = new AtomicInteger(0);
    //更新atomicInteger：获取访问次数
    public final int getAndIncrement(){
        int current;
        int next;

        do{
            current = this.atomicInteger.get();
            next = current == Integer.MAX_VALUE ? 0 : current + 1;
        }while (!this.atomicInteger.compareAndSet(current,next));

        log.info("**第 {} 次访问**",next);
        return next;
    }

    @Override
    public ServiceInstance instances(List<ServiceInstance> serviceInstances) {
        if (serviceInstances.size() == 0 || serviceInstances == null){
          //当前没有服务
            return null;
        }
         //选取第几个服务
        int index = getAndIncrement() % serviceInstances.size();
        return serviceInstances.get(index);
    }
}
```

四、OpenFeign 服务接口调用

1）介绍

OpenFeign是一个声明式的、模板化的HTTP客户端，它简化了使用RESTful服务的开发。它是Spring Cloud生态系统中的一部分，用于简化微服务架构中服务之间的通信。通过OpenFeign，开发者可以通过注解方式来定义和实现HTTP请求，而无需手动编写HTTP客户端代码。

以下是OpenFeign的一些主要特点和用法：

1. **声明式 REST 客户端：**

OpenFeign支持使用注解方式声明REST客户端接口，类似于Spring MVC的`@RequestMapping`注解。通过这些注解，开发者可以定义需要调用的服务端点、请求方法、请求参数等信息。

```
@FeignClient(name = "example-service")
public interface MyFeignClient {

    @GetMapping("/api/resource/{id}")
    ResponseEntity<Resource> getResourceById(@PathVariable("id") Long id);
}
```

2. **集成了Ribbon：**

OpenFeign集成了Netflix的Ribbon负载均衡器，因此它可以轻松地与服务发现组件（如Eureka）一起工作，实现在服务之间的负载均衡。

3. **支持多种注解：**

- `@FeignClient`：标注在接口上，用于指定目标服务的名称、负载均衡器配置等。
- `@RequestMapping`、`@GetMapping`、`@PostMapping`等：用于定义请求的路径、方法等信息。
- `@RequestParam`、`@PathVariable`、`@RequestBody`等：用于处理请求参数。

4. **集成了 Hystrix：**

OpenFeign集成了Netflix的Hystrix，提供了对服务的容错能力。通过在`@FeignClient`注解上添加`fallback`属性，可以指定服务降级的处理类。

```
@FeignClient(name = "example-service", fallback = MyFeignClientFallback.class)
public interface MyFeignClient {

    // ...
}
```

5. **支持 Spring Cloud Contract：**

OpenFeign支持使用Spring Cloud Contract来定义和生成服务契约，这有助于在服务提供者和消费者之间定义和维护API的规范。

6. **支持自定义编码器和解码器：**

可以通过实现`RequestInterceptor`和`Encoder`、`Decoder`等接口来自定义请求拦截器以及请求和响应的编码解码逻辑。

7. **易于集成和使用：**

OpenFeign的集成相对简单，只需在项目中添加相应的依赖，并通过注解方式定义接口即可。与Ribbon、Hystrix等组件的集成也是无缝的，可以在Spring Cloud中方便地构建和管理微服务架构。

使用OpenFeign时，只需声明接口并使用注解描述服务调用，OpenFeign会根据这些注解自动生成具体的实现。这样，开发者无需手动编写HTTP客户端代码，大大简化了服务之间通信的实现。

2）使用

①创建feign接口

```
@Component
@FeignClient("CLOUD-PAYMENT-SERVICE")
public interface PaymentFeignService {

    @GetMapping("/payment/get/{id}")
    CommonResult<Payment> getPaymentById(@PathVariable("id") Long id);

    @GetMapping("/payment/feign/timeout")
    String paymentFeignTimeout();

}
```

②在controller调用feignService的方法

```
@GetMapping("/consumer/payment/get/{id}")
public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id){
  log.info("{} >>>> 通过feign调用接口~", LocalDateTime.now());
  return paymentFeignService.getPaymentById(id);
}

@GetMapping("/consumer/payment/feign/timeout")
public String paymentFeignTimeOut() {
  return paymentFeignService.paymentFeignTimeout();
}
```

3）Feign调用的日志

①创建配置类FeignConfig

```
@Configuration
public class FeignConfig {
    @Bean
    Logger.Level feignLoggerLevel(){
      //日志级别
        return Logger.Level.BASIC;
    }
}
```

> - `Logger.Level.NONE`：禁用日志（默认）。
> - `Logger.Level.BASIC`：仅记录请求方法、URL、响应状态码以及执行时间。
> - `Logger.Level.HEADERS`：记录 BASIC 级别的信息，并且记录请求和响应的头信息。
> - `Logger.Level.FULL`：记录所有请求和响应的明细，包括请求体、响应体等。

②配置yml

```yaml
logging:
  level:
    # feign日志以什么级别监控哪个接口
    com.vanky.springcloud.service.PaymentFeignService: debug
```

五、Hystrix 断路器

1）介绍

Hystrix是Netflix开源的一个用于处理分布式系统的延迟和故障的库。它提供了一种通过添加延迟容错和故障切换的逻辑来控制分布式系统之间的交互的方式。Hystrix使得我们能够在分布式系统中构建具有高可用性和弹性的服务。

以下是Hystrix的一些主要特点和用法：

1. **容错机制：**

Hystrix通过实现断路器模式来防止在复杂的分布式系统中的故障进行传递。断路器模式会在服务调用失败达到一定阈值后打开，阻止进一步的请求，以免对故障的服务进行不必要的调用。一段时间后，断路器会尝试半开，如果服务调用成功，则断路器会继续关闭，否则它会继续打开。

2. **延迟和超时处理：**

Hystrix可以定义每个服务调用的超时时间，当服务调用超过指定时间仍未返回时，Hystrix将视为超时，并采取相应的容错策略。

3. **降级机制：**

当远程服务调用失败或超时时，Hystrix可以提供一个备选的降级逻辑，返回一个默认的值或执行备选的逻辑。这有助于防止整个系统崩溃，提高系统的稳定性。

4. **隔离策略：**

Hystrix使用隔离技术，通过将不同的服务调用隔离到独立的线程池中，防止由于某个服务调用的故障导致整个系统的线程池耗尽。

5. **指标收集和监控：**

Hystrix提供了丰富的指标（metrics）收集，包括每个服务调用的成功、失败、超时等情况。这些指标可以通过Hystrix Dashboard进行监控，方便运维人员实时查看服务的健康状况。

6. **自动熔断：**

Hystrix通过自动检测服务调用的失败率和延迟情况来实现自动熔断。当服务调用失败率超过阈值或延迟超过设定的时间时，Hystrix会自动打开断路器，避免对故障的服务进行进一步的调用。

7. **Fallback机制：**

Hystrix提供了Fallback机制，允许在服务调用失败时执行备选逻辑。开发者可以定义一个Fallback方法，当主逻辑执行失败时，自动调用Fallback方法。

8. **集成Spring Cloud：**

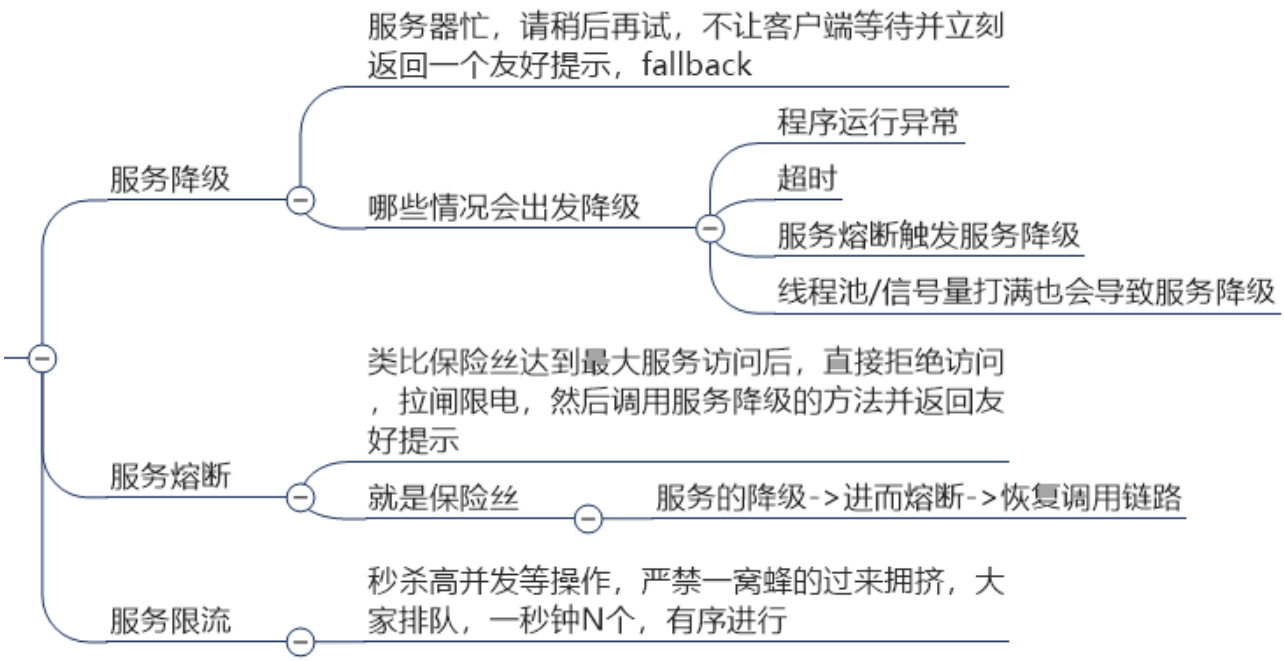
Hystrix广泛用于Spring Cloud中，通过使用`@HystrixCommand`注解，可以很方便地将Hystrix的功能集成到微服务架构中。

9. **支持缓存：**

Hystrix支持对服务调用的结果进行缓存，从而提高对于相同请求的响应速度。

总体而言，Hystrix提供了一套强大的工具和机制，使得在分布式系统中更容易实现高可用性和弹性。通过合理使用Hystrix，可以有效地处理分布式系统中不可避免的故障和延迟。

2）服务熔断、服务降级、服务限流



服务熔断、服务降级和服务限流是在分布式系统中处理故障和异常的关键策略。它们的目标是提高系统的稳定性、可用性，并在面临故障或高负载时保护系统免受更严重的问题。以下是它们的概念和区别：

1. **服务熔断（Circuit Breaker）：**

服务熔断是一种防止故障在整个系统中传播的机制。当系统中的某个服务达到一定的失败率或响应时间阈值时，熔断器打开，阻止进一步的请求流向失败的服务。打开后，熔断器会在一段时间内拒绝所有请求，然后定期检测服务的健康状况，如果服务恢复正常，熔断器逐渐关闭，允许流量再次通过。服务熔断的目的是防止故障的服务对整个系统产生连锁反应，从而提高系统的稳定性。

2. **服务降级（Fallback）：**

服务降级是在面临高负载或异常情况时，为了保护系统整体稳定性而采取的一种策略。当某个服务不可用或响应时间超过阈值时，系统可以通过提供一个备选方案，返回一个默认值或执行一个备选逻辑，而不是返回错误。服务降级的目的是在面临不可避免的故障时，最小化对用户的影响，提供系统的基本功能。

3. **服务限流（Rate Limiting）：**

服务限流是通过限制系统中某个服务的请求速率来保护系统。通过设置每秒或每分钟可以处理的请求数量，可以防止系统因过多请求而超负荷。服务限流可以在应对恶意攻击、保护系统资源、防止过载等方面发挥作用。例如，可以使用令牌桶算法或漏桶算法来实现服务限流。

区别和综合应用：

- **服务熔断和服务降级：**
  - 服务熔断更注重于阻止故障的服务对整个系统的影响，而服务降级更注重于在面临高负载或异常情况时提供系统的基本功能。
  - 在一些场景中，服务熔断和服务降级可以结合使用，例如，当服务熔断打开时，可以提供一个降级逻辑，返回默认值或执行备选逻辑。
- **服务降级和服务限流：**
  - 服务降级通常是为了在面临故障时提供备选方案，而服务限流是为了防止系统过载。二者可以在系统设计中结合使用，通过限流防止系统过载，通过降级提供基本功能。
- **服务熔断和服务限流：**
  - 服务熔断通常是为了防止故障的服务对整个系统产生连锁反应，而服务限流是为了防止系统过载。结合使用可以有效保护系统不受故障和过载的影响。

在实际系统设计中，这三种策略往往会综合应用，以构建更加健壮、稳定和可用的分布式系统。

3）服务降级的实现

不知道是是不是hystrix太老，而且很多配置已经被弃用，这里无法实现服务降级，而是直接报错了

1.服务提供端

①service实现类

```
@HystrixCommand(fallbackMethod = "timeout_fallback")     //服务降级的兜底方法
    public String timeout(Integer id){
        int time = 5;
        //int i = 10/0;
        try {
            TimeUnit.SECONDS.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

```
        return "线程: " + Thread.currentThread().getName() + "paymentInfo_OK [" + id + "]  耗时(秒) : " + time;
    }

    public String timeout_fallback(Integer id){
        return "服务超时了: 我是兜底的方法~~";
    }
```

②主启动类

加注解：@EnableHystrix

2.服务消费端

①创建一个service实现类实现feignService接口，重写方法用于备选方法。

```
@Component
public class PaymentFallbackService implements PaymentHystrixService {
    @Override
    public String paymentInfo_OK(Integer id) {
        return "PaymentFallbackService >>> paymentInfo_OK o(╥﹏╥)o";
    }

    @Override
    public String timeout(Integer id) {
        return "PaymentFallbackService >>> timeout o(╥﹏╥)o";
    }
}
```

②在feignService上加上注释：

```
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT",fallback = PaymentFallbackService.class)
```

③主启动类：`@EnableHystrix`

4）服务熔断的实现

在服务提供者的service实现类方法上注释

```
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback",commandProperties = {
  @HystrixProperty(name = "circuitBreaker.enabled",value = "true"),
  @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value = "10"),
  @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value = "10000"),
  @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value = "60"),
})
public String paymentCircuitBreaker(@PathVariable("id") Integer id)
{
```

```
    if(id < 0)
    {
      throw new RuntimeException("******id 不能负数");
    }
    String serialNumber = IdUtil.simpleUUID();

    return Thread.currentThread().getName()+"\t"+"调用成功，流水号: " + serialNumber;
}

public String paymentCircuitBreaker_fallback(@PathVariable("id") Integer id)
{
    return "id 不能负数，请稍后再试，/(ToT)/~~   id: " +id;
}
```

这段代码使用了Hystrix库中的注解，该库是一个用于隔离对远程系统、服务和第三方库的访问点的延迟和容错库。`@HystrixCommand`注解用于声明一个将被包装成断路器的方法。

1. `fallbackMethod = "paymentCircuitBreaker_fallback"`: 指定在带有注解的方法失败或遇到异常时要调用的回退方法的名称。
2. `commandProperties = { ... }`: 定义一个`@HystrixProperty`注解数组，用于配置Hystrix命令的各种属性。
    - `@HystrixProperty(name = "circuitBreaker.enabled", value = "true")`: 启用带有注解的方法的断路器。
    - `@HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "10")`: 设置在一个滚动窗口中触发断路器的最小请求数。在这种情况下，如果在滚动窗口中的请求数少于10个，则无论错误百分比如何，断路器都不会打开。
    - `@HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "10000")`: 设置在触发断路器后，在允许再次尝试执行失败操作之前的时间量（以毫秒为单位）。在这种情况下，设置为10,000毫秒（10秒）。
    - `@HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "60")`: 设置触发断路器的错误百分比阈值。如果错误百分比超过此阈值，断路器将打开。在这种情况下，如果60%或更多的请求失败，断路器将打开。

总体而言，此Hystrix命令配置了一个断路器，如果在一个滚动窗口中有10个或更多的请求，并且错误率为60%或更高，则断路器将打开。如果断路器打开，将调用指定的回退方法（`paymentCircuitBreaker_fallback`）来处理后续的请求，直到断路器在指定的休眠窗口后关闭。

六、GateWay 网关

1）介绍

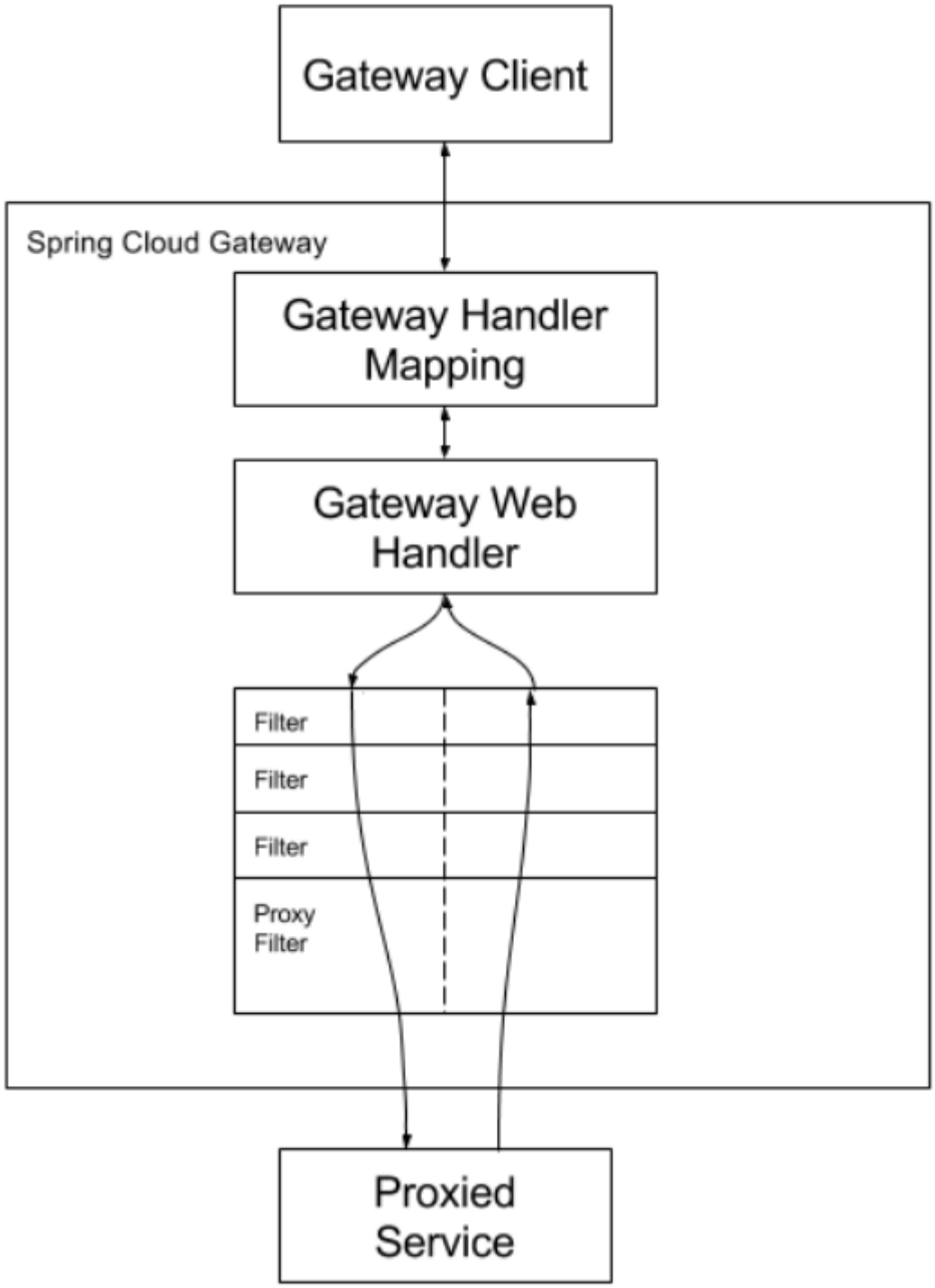
Spring Cloud Gateway 是 Spring Cloud 生态系统中的一个项目，用于构建微服务架构中的**网关服务**。它提供了一种灵活的方式来**路由请求、过滤请求以及执行一些其他的任务，比如熔断、限流等**。Spring Cloud Gateway 的设计目标是提供一种简单而有效的方式来处理基于 HTTP 的请求，同时具备高度的可扩展性。

以下是 Spring Cloud Gateway 的一些主要特性和概念：

1. **路由（Routing）：** Spring Cloud Gateway 可以**根据预定义的路由规则将请求转发到不同的微服务。**这些路由规则可以通过配置文件或者代码进行定义。
2. **断言（Predicate）：** 通过使用断言，可以定义哪些请求会匹配到特定的路由。例如，可以根据请求的路径、主机、请求方法等条件来进行路由。
3. **过滤器（Filter）：** 过滤器是 Spring Cloud Gateway 中非常重要的一部分，它允许在请求被路由前或者之后执行一些操作。过滤器可以用于修改请求、修改响应、执行认证、记录日志等。
4. **全局过滤器（Global Filters）：** 这是一类应用于所有路由的过滤器，可以用于一些全局的任务，例如安全认证、日志记录等。
5. **熔断器（Circuit Breaker）：** Spring Cloud Gateway 提供了断路器的支持，可以防止微服务之间的故障扩散。
6. **限流（Rate Limiting）：** 通过集成第三方库，Spring Cloud Gateway 支持请求的限流，防止某个微服务被过度请求。
7. **集成 Spring Cloud DiscoveryClient：** 可以通过整合服务注册中心（如 Eureka、Consul）实现动态路由的功能，自动感知微服务的变化。

Spring Cloud Gateway 构建在 Spring Framework 5、Project Reactor 和 Spring Boot 的基础上，采用了响应式编程的思想，可以处理大量的并发请求。通过使用 Spring Cloud Gateway，开发人员可以更加灵活地管理微服务架构中的请求流量、路由和过滤逻辑。

2）工作流程



Spring Cloud Gateway的工作流程可以概括为以下几个步骤：

1. **客户端发起请求：** 整个流程的起点是客户端向网关发起请求。这可以是HTTP请求，例如来自浏览器或移动应用程序的请求。
2. **路由（Routing）：** 接收到请求后，Spring Cloud Gateway首先根据预定义的路由规则将请求路由到相应的目标服务。路由规则可以基于请求的路径、主机、请求方法等条件进行定义。这一步骤允许网关将请求导向不同的微服务。
3. **断言（Predicate）：** 在路由过程中，可以使用断言定义哪些请求将匹配特定的路由规则。断言可以基于请求的各种属性，如路径、主机、请求方法等。匹配成功的请求将被路由到相应的目标服务。
4. **过滤器（Filter）：** 一旦请求被路由到目标服务，网关会应用一系列过滤器。过滤器可以用于修改请求、修改响应、执行认证、记录日志等。过滤器的使用可以在请求的不同生命周期阶段执行。

5. **目标服务处理请求：** 经过网关路由和过滤器的处理后，请求最终被路由到相应的目标服务。目标服务执行业务逻辑，产生响应并将其返回给网关。
6. **响应返回客户端：** 目标服务的响应经过网关，可以通过一系列过滤器进行修改。最终，网关将处理后的响应返回给发起请求的客户端。

总的来说，Spring Cloud Gateway通过路由和过滤器的组合，实现了对请求的动态路由和各种处理逻辑的灵活控制。这使得开发人员能够更好地管理和控制微服务架构中的请求流量，实现各种功能，如路由、认证、日志记录等。

3）配置路由的两种方法

1.yml配置

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: payment_routh
          uri: http://localhost:8001
          predicates:    # 断言，当路径符合时，转发到上面的uri
            - Path=/payment/get/**
        - id: payment_routh2
          uri: http://localhost:8001
          predicates:
            - Path=/payment/lb/**
```

2.配置类

访问 localhost:9527/guonei 时，转发到 -----》》 http://news.baidu.com/guonei

```java
@Configuration
public class GateWayConfig {
    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder builder){
        RouteLocatorBuilder.Builder routes = builder.routes();

        RouteLocator path1 = routes.route("path1", r -> r.path("/guonei")
                                          .uri("http://news.baidu.com/guonei")).build();

        return path1;
    }
}
```

4）动态路由（网关负载均衡）

```yaml
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true # 开启从注册中心动态创建路由的功能，利用微服务名进行路由
      routes:
        - id: payment_routh
#          uri: http://localhost:8001
          uri: lb://cloud-payment-service
          predicates:
            - Path=/payment/get/**
        - id: payment_routh2
#          uri: http://localhost:8001
          uri: lb://cloud-payment-service
          predicates:
            - Path=/payment/lb/**
```

5）断言类型

Spring Cloud Gateway提供了多种内建的断言类型，这些断言类型基于请求的不同属性进行匹配。以下是一些常见的断言类型：

1. **Path（路径匹配）：**
   - **示例：**

     ```yaml
     - predicates:
         - Path=/api/**
     ```

     这个断言表示如果请求的路径匹配 `/api/**`，则这个条件成立。
2. **Host（主机名匹配）：**
   - **示例：**

     ```yaml
     - predicates:
         - Host=**.example.com
     ```

     这个断言表示如果请求的主机名符合 `**.example.com` 的模式，则这个条件成立。
3. **Method（请求方法匹配）：**
   - **示例：**

     ```yaml
     - predicates:
         - Method=GET
     ```

     这个断言表示如果请求的方法是 GET，则这个条件成立。

4. **RemoteAddr（远程地址匹配）：**
   - **示例：**

     ```
     - predicates:
         - RemoteAddr=192.168.1.1
     ```

     这个断言表示如果请求的远程地址是 `192.168.1.1`，则这个条件成立。
5. **Header（请求头匹配）：**
   - **示例：**

     ```
     - predicates:
         - Header=X-Request-Header, \d+
     ```

     这个断言表示如果请求头中的 `X-Request-Header` 存在且其值是数字，则这个条件成立。
6. **Query（查询参数匹配）：**
   - **示例：**

     ```
     - predicates:
         - Query=param=value
     ```

     这个断言表示如果请求的查询参数中包含 `param=value`，则这个条件成立。
7. **Cookie（Cookie匹配）：**
   - **示例：**

     ```
     - predicates:
         - Cookie=cookieName, cookieValue
     ```

     这个断言表示如果请求中包含名为 `cookieName` 且值为 `cookieValue` 的 Cookie，则这个条件成立。
8. **Method（请求方法匹配）：**
   - **示例：**

     ```
     - predicates:
         - Method=GET
     ```

     这个断言表示如果请求的方法是 GET，则这个条件成立。
9. **CompositeRoutePredicateFactory（组合断言）：**
   - **示例：**

```
- predicates:
    - name: Path
      args:
        pattern: "/api/**"
    - name: Query
      args:
        param: "param"
        regexp: "value"
```

这个断言表示如果请求的路径匹配 `/api/**` 且查询参数中包含 `param=value`，则这个条件成立。

10. **After/Before/Between（时间断言）**：
    - 实例：

```
- predicates:
    - After=2023-11-27T15:12:26.965396900+08:00[Asia/Shanghai]
```

上述示例中的配置是在Spring Cloud Gateway的路由配置中使用的，它定义了一系列断言来匹配请求的不同属性。断言的组合和配置可以根据具体的需求来灵活定义，以实现对请求的精确匹配和路由。除了这些内建的断言类型，Spring Cloud Gateway还支持自定义断言，使得开发人员能够更加灵活地定制路由规则。

6）自定义过滤器

①带@Component注释

②实现GlobalFilter 和 Ordered 两个接口

③重写方法

```
@Component
@Slf4j
public class MyLogGateWayFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        log.info("********进入过滤器:  MyLogGateWayFilter <<<< " + new Date());
        String uname = exchange.getRequest().getQueryParams().getFirst("uname");
        if (uname == null){
            log.info("*******uname为null，非法用户！！！");
            exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
            return exchange.getResponse().setComplete();
        }
      //放行到下一个过滤器
        return chain.filter(exchange);
    }

    @Override

    public int getOrder() {
```

```
        //过滤器执行层级，越小越先执行
        return 0;
    }
}
```

七、Config 配置中心

1）介绍

Spring Cloud Config 是 Spring Cloud 生态系统中的一个项目，用于**集中管理和配置分布式系统中的应用程序的配置信息。**它提供了一种轻量级、易扩展的方式，允许开发人员在不重新部署应用程序的情况下动态更新配置。

以下是 Spring Cloud Config 的主要特性和组件：

1. **配置存储：** Spring Cloud Config 支持**将配置信息存储在不同的后端存储**中，例如 Git 仓库、SVN 仓库、本地文件系统等。这使得配置信息可以轻松地集中存储和管理。
2. **配置服务：** Spring Cloud Config Server 是**配置的中心化服务，它提供 RESTful API 用于获取配置信息。**通过配置服务，应用程序可以从 Config Server 动态获取其配置信息。
3. **配置客户端：** Spring Cloud Config Client 是一个用于**集成配置服务的库，它允许应用程序在启动时从配置服务中获取配置信息。**客户端可以通过定期轮询或使用 Webhook 的方式来获取配置的更新。
4. **分布式配置：** Spring Cloud Config 可以被集成到分布式系统中，使得不同服务可以通过配置服务来集中管理配置信息。这对于微服务架构中的多个服务协同工作、灵活配置非常有用。
5. **环境和应用分离：** Spring Cloud Config 支持在不同的环境和应用之间进行配置的分离。通过命名规范和目录结构，可以轻松管理不同环境（如开发、测试、生产）和应用之间的配置。
6. **加密和解密：** Spring Cloud Config 提供了对配置信息的加密和解密支持，保护敏感信息的安全性。

下面是一个简单的 Spring Cloud Config Server 的配置示例：

```
spring:
  application:
    name: config-server
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-username/config-repo
```

上述配置表示将配置存储在 GitHub 上的一个 Git 仓库中。Config Server 将会在启动时从这个仓库中读取配置信息，并通过 RESTful API 提供给客户端。

Spring Cloud Config 的使用可以显著简化配置管理和更新的过程，使得系统更具灵活性和可维护性。

2）服务端

①配置类

```yaml
server:
  port: 3344

spring:
  application:
    name:  cloud-config-center #注册进Eureka服务器的微服务名
  cloud:
    config:
      server:
        git:
          uri: https://github.com/vankykoo/spring-cloud-config.git #GitHub上面的git仓库名字
          ####搜索目录
          timeout: 3000
          search-paths:
            - spring-cloud-config
      ####读取分支
      label: main

#服务注册到eureka地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka
```

②主启动类

```java
@SpringBootApplication
@EnableConfigServer
public class ConfigCenterMain3344 {
    public static void main(String[] args) {
            SpringApplication.run(ConfigCenterMain3344.class, args);
    }
}
```

3）服务类

bootstrap.yml

```yaml
server:
  port: 3355

spring:
  application:
    name: config-client
  cloud:
    #Config客户端配置
    config:
      label: main #分支名称
```

```
      name: config #配置文件名称
      profile: dev #读取后缀名称   上述3个综合：master分支上config-dev.yml的配置文件被读取
http://config-3344.com:3344/master/config-dev.yml
      uri: http://config-3344.com:3344 #配置中心地址k

#服务注册到eureka地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka

# 暴露监控端点
# 用于动态更新
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

②controller

```
@RestController
@RefreshScope
public class ConfigClientController {
    @Value("${config.info}")
    private String configInfo;

    @GetMapping("/configInfo")
    public String getConfigInfo() {
        return configInfo;
    }
}
```

③主启动类

```
@SpringBootApplication
@EnableDiscoveryClient
public class ConfigClientMain3355 {
    public static void main(String[] args) {
        SpringApplication.run(ConfigClientMain3355.class,args);
    }
}
```

八、Bus

1）介绍

Spring Cloud Bus 是 Spring Cloud 生态系统中的一个模块，它用于在分布式系统中轻松地将**消息传播（Message Brokering）和事件广播**引入到微服务架构中。Spring Cloud Bus 的目标是**简化分布式系统中的配置更改、事件传播等任务。**

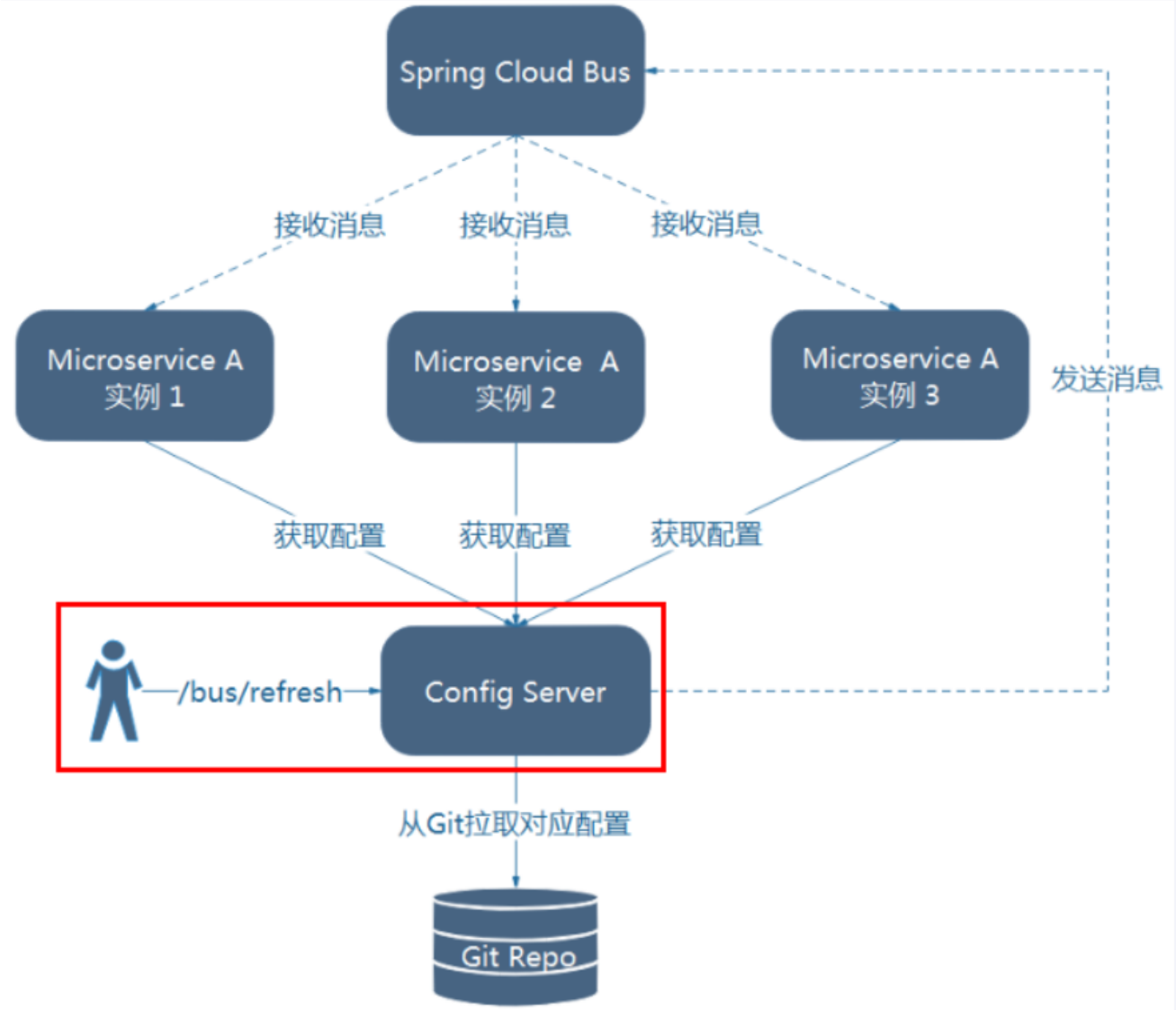
以下是 Spring Cloud Bus 的主要特点和概念：

1. **消息传播：** Spring Cloud Bus 使用消息代理（通常是消息队列，如 RabbitMQ 或 Kafka）来传播消息。这意味着当一个微服务节点上的配置发生变化时，这个变化会通过消息传播到其他微服务节点，从而实现配置的同步。
2. **配置中心：** Spring Cloud Bus 通常与 Spring Cloud Config 配置中心搭配使用。当配置中心的配置发生变化时，Spring Cloud Bus 将这个变化通知给所有连接到消息代理的微服务实例，以便它们可以动态地刷新配置。
3. **事件广播：** 除了配置的刷新，Spring Cloud Bus 还可以用于广播其他自定义事件。这使得在整个微服务架构中触发和处理事件变得更加容易。
4. **支持多种消息代理：** Spring Cloud Bus 支持多种消息代理，包括 RabbitMQ、Kafka 等，因此你可以选择适合你架构和需求的消息代理。
5. **轻量级：** Spring Cloud Bus 是一个轻量级的模块，易于集成到 Spring Cloud 应用程序中。

使用 Spring Cloud Bus 的一般步骤包括：

1. **添加依赖：** 在你的 Spring Cloud 项目中添加 Spring Cloud Bus 的依赖。
2. **配置消息代理：** 配置你的 Spring Cloud Bus 使用的消息代理，例如 RabbitMQ 或 Kafka。
3. **触发事件：** 在一个微服务实例上触发事件，例如刷新配置。
4. **监听事件：** 在其他微服务实例中监听这些事件，并执行相应的操作。

通过使用 Spring Cloud Bus，你可以实现配置的动态刷新、事件的广播等功能，使得微服务架构更加灵活、可扩展。



2）server端

都要引入依赖：

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

①application.yml

```
server:

  port: 3344
```

```yaml
spring:
  application:
    name:  cloud-config-center #注册进Eureka服务器的微服务名
  cloud:
    config:
      server:
        git:
          uri: https://github.com/vankykoo/spring-cloud-config.git #GitHub上面的git仓库名字
          ####搜索目录
          timeout: 3000
          search-paths:
            - spring-cloud-config
      ####读取分支
      label: main

  #rabbitmq相关配置
rabbitmq:
  host: 192.168.200.142
  port: 15672
  username: admin
  password: 123456

#服务注册到eureka地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka

##rabbitmq相关配置,暴露bus刷新配置的端点
management:
  endpoints: #暴露bus刷新配置的端点
    web:
      exposure:
        include: 'bus-refresh'
```

3）客户端

```yaml
server:
  port: 3366

spring:
  application:
    name: config-client
  cloud:
    #Config客户端配置
    config:
      label: main #分支名称
      name: config #配置文件名称

      profile: dev #读取后缀名称   上述3个综合：master分支上config-dev.yml的配置文件被读取
```

```
http://config-3344.com:3344/master/config-dev.yml
      uri: http://localhost:3344 #配置中心地址k
  #rabbitmq相关配置 15672是Web管理界面的端口；5672是MQ访问的端口
  rabbitmq:
    host: 192.168.200.142
    port: 15672
    username: admin
    password: 123456

#服务注册到eureka地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:7001/eureka

# 暴露监控端点
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

九、Spring Cloud Stream

1）介绍

Spring Cloud Stream 是一个用于构建消息驱动微服务的框架，它基于 Spring Boot 构建，利用 Spring Integration 提供的强大的消息处理能力。Spring Cloud Stream 的目标是**简化开发者构建分布式系统中的消息驱动应用程序的过程。**

以下是 Spring Cloud Stream 的一些主要特性和概念：

1. **Binder：** Spring Cloud Stream **提供了 Binder 抽象层，它允许开发者将应用程序与消息中间件连接起来。** Binder 封装了与消息中间件的通信细节，使得应用程序可以**方便地切换底层消息中间件的实现**，例如 RabbitMQ、Kafka、Apache Kafka、Apache RocketMQ 等。
2. **消息通道：** Spring Cloud Stream 引入了消息通道的概念，它是用于在应用程序组件之间传递消息的抽象。应用程序可以定义输入通道和输出通道，消息会通过这些通道进行传递。
3. **消息转换器：** Spring Cloud Stream 支持消息的自动序列化和反序列化，开发者可以使用各种格式的消息，例如 JSON、Avro 等。这是通过消息转换器（Message Converter）来实现的。
4. **Binder 的配置：** 开发者可以通过配置文件或者代码来配置 Binder，以便与特定的消息中间件进行集成。配置包括连接到消息代理所需的信息，如主机、端口、凭据等。
5. **简化的发布-订阅模型：** Spring Cloud Stream 简化了发布-订阅模型的实现，开发者可以更专注于业务逻辑而不用过多关注消息传递细节。
6. **内置的消息处理：** Spring Cloud Stream 结合 Spring Integration，提供了一些内置的消息处理功能，例如分组、分区、重试、延迟等。
7. **支持微服务架构：** Spring Cloud Stream 被设计为适用于微服务架构，它可以与 Spring Cloud 中的其他组件（如 Eureka、Config Server、Zipkin 等）结合使用，提供全面的微服务支持。

一个简单的 Spring Cloud Stream 应用通常包括以下组件：

- **Source：** 负责产生消息并发送到消息通道。源通常是消息的生产者。
- **Processor：** 接收来自输入通道的消息，进行处理后再发送到输出通道。处理器通常是消息的处理中间环节。
- **Sink：** 接收来自输入通道的消息，处理后执行特定的操作。汇通常是消息的消费者。

通过这种方式，开发者可以将应用程序拆分为更小、更可维护的组件，并通过消息通道进行通信。

Spring Cloud Stream 提供了与多种消息中间件的 Binder 实现，因此可以轻松地切换底层消息中间件，而不用修改应用程序代码。这种抽象层的设计使得 Spring Cloud Stream 成为构建消息驱动微服务的强大工具。

2）常用注解和 API

Spring Cloud Stream 提供了一系列注解和 API，用于简化消息驱动微服务的开发。以下是一些常用的注解和 API：

注解：

1. **@Input：**
   - **作用：** 用于定义输入通道，表示应用程序从消息代理接收消息。
   - **示例：**

     ```
     public interface MySource {
         @Input("myInputChannel")
         SubscribableChannel myInput();
     }
     ```

2. **@Output：**
   - **作用：** 用于定义输出通道，表示应用程序向消息代理发送消息。
   - **示例：**

     ```
     public interface MySource {
         @Output("myOutputChannel")
         MessageChannel myOutput();
     }
     ```

3. **@StreamListener：**
   - **作用：** 用于定义消息监听器，监听指定的输入通道，并在接收到消息时执行相应的处理逻辑。
   - **示例：**

     ```
     @StreamListener("myInputChannel")
     public void handleInputMessage(String message) {
         // 处理接收到的消息
     }
     ```

4. **@SendTo：**

- **作用：** 用于指定处理消息的方法的返回值发送到指定的输出通道。
- **示例：**

```
@StreamListener("myInputChannel")
@SendTo("myOutputChannel")
public String handleInputMessage(String message) {
    // 处理接收到的消息并返回结果
}
```

5. **@StreamMessageConverter：**
   - **作用：** 用于自定义消息转换器，控制消息的序列化和反序列化。
   - **示例：**

```
@Bean
@StreamMessageConverter
public MyMessageConverter myMessageConverter() {
    // 自定义消息转换器
    return new MyMessageConverter();
}
```

API：

1. **BinderAwareChannelResolver：**
   - **作用：** 用于根据通道名称解析消息通道，获取 `MessageChannel` 实例。
   - **示例：**

```
@Autowired
private BinderAwareChannelResolver resolver;

public void sendMessage(String channelName, String message) {
    MessageChannel channel = resolver.resolveDestination(channelName);
    channel.send(MessageBuilder.withPayload(message).build());
}
```

2. **BindingService：**
   - **作用：** 提供了用于创建和管理绑定关系的 API，例如声明性地创建绑定关系。
   - **示例：**

```
@Autowired
private BindingService bindingService;

public void createBinding() {
    bindingService.bindConsumer(myInputChannel(), "myGroup");
    bindingService.bindProducer(myOutputChannel());
}
```

这些注解和 API 使得开发者能够方便地定义和管理与消息中间件的绑定关系，以及实现消息的生产和消费逻辑。通过这些抽象，Spring Cloud Stream 提供了一种简单而强大的方式来构建消息驱动微服务。

3）消息生产者

①pom引入依赖

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

②service实现类

```
@Service
@Slf4j
public class MessageProviderImpl implements IMessageProvider {

    @Resource
    private StreamBridge streamBridge;

    //@Resource
    //private MessageChannel output;

    @Override
    public String send() {
        //老方法
        //String uuid1 = UUID.randomUUID().toString();
        //output.send(MessageBuilder.withPayload(uuid1).build());
        //log.info("MessageChannel  发送消息：{}",uuid1);

        //新方法
        String uuid2 = UUID.randomUUID().toString();
        streamBridge.send("output",MessageBuilder.withPayload(uuid2).build());
        log.info("StreamBridge  发送消息：{}",uuid2);
        return null;
    }
}
```

③controller

```
@RestController
public class SendMessageController
{
    @Resource
    private IMessageProvider messageProvider;

    @GetMapping(value = "/sendMessage")
    public String sendMessage()
    {
        return messageProvider.send();
    }
}
```

3）消息消费者

①yml

```
server:
  port: 8802

spring:
  application:
    name: cloud-stream-consumer
  cloud:
    stream:
      binders: # 在此处配置要绑定的rabbitmq的服务信息;
        defaultRabbit: # 表示定义的名称，用于于binding整合
          type: rabbit # 消息组件类型
          environment: # 设置rabbitmq的相关的环境配置
            spring:
              rabbitmq:
                host: 192.168.200.142
                port: 5672
                username: admin
                password: 123456
      bindings: # 服务的整合处理
        inputChannel-in-0: # 这个名字是一个通道的名称
          destination: studyExchange # 表示要使用的Exchange名称定义
          content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设
置“text/plain”

eureka:
  client: # 客户端进行Eureka注册的配置
    service-url:
      defaultZone: http://localhost:7001/eureka
  instance:
    lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）
    lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）
    instance-id: receive-8802.com  # 在信息列表时显示主机名称

    prefer-ip-address: true     # 访问的路径变为IP地址
```

②接收消息

```
@Service
@Slf4j
public class ReceiveMessageListener{
    @Bean
    public Consumer<String> myChannel(){
        return message -> log.info("收到消息" + message);
    }
}
```

4）分组消费

在 Spring Cloud Stream 中，分组消费和持久化通常与消息中间件的特性相关，主要涉及 RabbitMQ 和 Kafka 这两种常见的消息中间件。

同一个组为竞争关系，不同组为群发。

RabbitMQ：

分组消费：

在 RabbitMQ 中，分组消费主要与队列的特性有关。RabbitMQ 中的队列支持分组（group）概念，一个队列可以有多个消费者，而这些消费者可以属于同一个分组。分组消费的主要目的是实现消息的负载均衡，确保同一组内的多个消费者中只有一个消费者接收到消息。

在 Spring Cloud Stream 中，你可以通过配置属性来指定消费者的分组。例如：

```
spring:
  application:
    name: cloud-stream-consumer
  cloud:
    stream:
      binders: # 在此处配置要绑定的rabbitmq的服务信息;
        defaultRabbit: # 表示定义的名称，用于于binding整合
          type: rabbit # 消息组件类型
          environment: # 设置rabbitmq的相关的环境配置
            spring:
              rabbitmq:
                host: 192.168.200.142
                port: 5672
                username: admin
                password: 123456
      bindings: # 服务的整合处理
        input-in-0: # 这个名字是一个通道的名称
          destination: studyExchange # 表示要使用的Exchange名称定义
          content-type: application/json # 设置消息类型，本次为对象json，如果是文本则设
```

```
置“text/plain”
          group: groupA #配置分组
```

持久化：

RabbitMQ 支持持久化消息，即消息在发布到队列时可以标记为持久的。这样即使 RabbitMQ 服务器宕机，消息也不会丢失。

在 Spring Cloud Stream 中，默认情况下，消息是持久的。可以通过配置 `spring.cloud.stream.rabbit.bindings.<channelName>.producer.required-groups` 属性来指定哪些组需要消息持久化。例如：

```
spring.cloud.stream.rabbit.bindings.output.producer.required-groups: myGroup
```

Kafka：

分组消费：

在 Kafka 中，分组消费是通过 Consumer Group 的概念来实现的。一个 Consumer Group 可以包含多个消费者，而同一个 Consumer Group 内的消费者将共同消费一个主题（Topic）的消息，实现了消息的负载均衡。

在 Spring Cloud Stream 中，你可以通过配置 `spring.cloud.stream.bindings.<channelName>.group` 属性来指定消费者的分组。例如：

```
spring.cloud.stream.bindings.input.group: myGroup
```

持久化：

Kafka 中的消息本身是持久化的，因为消息写入到 Kafka 主题（Topic）后，默认会保留一段时间。持久性主要与 Kafka 的主题配置有关，可以通过设置主题的 `retention.ms` 等属性来控制消息的保留时间。

在 Spring Cloud Stream 中，默认情况下，消息是被持久化的。如果你需要更精细的配置，可以参考 Kafka 的主题配置。

总的来说，分组消费和持久化在 Spring Cloud Stream 中是与底层消息中间件的特性相关的，可以通过配置适当的属性来实现。在使用时，建议详细了解所选消息中间件的特性和配置，以确保满足业务需求。

十、micrometer 监控

1）介绍

Micrometer 是一个用于应用程序度量和监控的度量库，它旨在为开发者提供一种简单、一致的方式来收集、导出和报告应用程序的指标。Micrometer 提供了一个通用的 API，支持多种监控系统，包括 Prometheus、Graphite、StatsD、InfluxDB、Datadog 等。

以下是 Micrometer 的主要特点和概念：

1. **通用 API：** Micrometer 提供了一套通用的 API，使得开发者可以使用相同的代码收集和导出指标，而不受监控系统的差异影响。这使得应用程序能够更轻松地适配不同的监控系统，或者在需要时切换监控系统。

2. **多种监控系统适配器：** Micrometer 提供了多个适配器，支持与不同的监控系统集成。这些适配器包括 Prometheus、Graphite、StatsD、InfluxDB、Datadog、Wavefront 等。通过选择适配器，开发者可以将应用程序的度量数据导出到他们选择的监控系统。
3. **度量类型：** Micrometer 支持各种常见的度量类型，包括计数器（Counter）、计时器（Timer）、直方图（DistributionSummary）、长任务计时器（LongTaskTimer）、计量器（Gauge）等。这些度量类型涵盖了应用程序性能和行为的不同方面。
4. **标签和维度：** Micrometer 支持标签（tags）和维度（dimensions），这使得度量数据能够更细粒度地组织和查询。标签可以用于区分不同实例、环境、版本等，从而更精确地分析和监控应用程序的性能。
5. **Spring Boot 集成：** Micrometer 在 Spring Boot 中有很好的集成，通过 Spring Boot 的 `Actuator` 模块，开发者可以方便地暴露和查看应用程序的度量端点。这也使得 Spring Boot 应用程序更容易与 Micrometer 集成。
6. **自定义度量：** 开发者可以使用 Micrometer 提供的 API 创建和记录自定义的度量指标。这使得应用程序能够根据具体的需求，记录和导出与业务相关的度量数据。
7. **支持多语言：** 尽管 Micrometer 的主要应用场景是 Java 应用程序，但它还提供了对其他语言的支持，如 Kotlin、Scala、Groovy、JavaScript、Go 等。这使得 Micrometer 不仅适用于纯 Java 应用程序，也可以在混合语言的环境中使用。

总体而言，Micrometer 提供了一种简单而强大的方式，帮助开发者收集、导出和监控应用程序的度量数据。它的设计理念是提供一致的 API 和多样化的监控系统适配器，使得开发者能够更灵活地选择和切换监控系统，而不用过多关注不同监控系统的细节。

2）使用

①在 dependencyManagement 中统一配置版本

```
<dependencyManagement>
  <dependencies>

    <dependency>
      <groupId>io.micrometer</groupId>
      <artifactId>micrometer-tracing-bom</artifactId>
      <version>${micrometer-tracing.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

  </dependencies>
</dependencyManagement>
```

②在每个服务的 pom 中配置如下

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-observation</artifactId>
</dependency>
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
  <groupId>io.zipkin.reporter2</groupId>
  <artifactId>zipkin-reporter-brave</artifactId>
  <version>2.16.3</version>
</dependency>
```

③每个服务的 yml 中配置如下

```
management:
  tracing:
    sampling:
      probability: 0.1 #这是默认的
```

`management.tracing.sampling.probability: 0.1` 表示设置采样率为 0.1。这里的采样率是一个介于 0 到 1 之间的值，**表示在整个请求流程中，有多少比例的请求会被跟踪记录。**1.0 表示所有请求都被记录，0.0 表示不记录任何请求。

④启动zipkin

1. 下载 zipkin 的jar包
2. 启动 zipkin
3. 访问 http://localhost:9411 默认网站

```
E:\java\zipkin>java -jar zipkin-server-3.0.0-rc0-exec.jar
```

十一、Nacos

1）介绍

Nacos（中文名“诺科斯”）是一个开源的服务发现、配置管理和服务管理平台，由阿里巴巴集团推出。Nacos 提供了一种简单而强大的方式，帮助开发者在微服务架构中实现服务注册与发现、动态配置管理、服务健康监测等功能。

以下是 Nacos 的一些主要特点和功能：

1. **服务发现与注册：** Nacos 提供了服务注册和发现功能，允许微服务在启动时向 Nacos 注册自己的信息，使得其他服务可以动态地发现和调用它们。
2. **动态配置管理：** Nacos 提供了动态配置管理的功能，允许开发者将配置信息存储在 Nacos 上，并在运行时动态地获取和更新配置。这有助于实现微服务架构中的配置中心。
3. **服务健康监测：** Nacos 具有服务健康监测的功能，可以监控注册在 Nacos 上的服务的健康状态，帮助实现服务的高可用性和容错。
4. **多数据中心和跨区域复制：** Nacos 支持多数据中心的部署，并能够在不同的数据中心之间进行跨区域的复制，以提高系统的可用性和容灾能力。
5. **一致性保证：** Nacos 使用了 Raft 算法来保证分布式系统中的一致性。这使得 Nacos 具有较强的数据一致性和可靠性。
6. **开放式 API：** Nacos 提供了丰富的开放式 API，可以通过 HTTP、gRPC 等方式与 Nacos 进行交互。这使得开发者可以轻松地集成 Nacos 到他们的应用程序中。
7. **可视化管理界面：** Nacos 提供了用户友好的可视化管理界面，方便开发者对注册的服务、配置信息等进行查看和管理。
8. **支持多种语言和框架：** Nacos 不仅支持 Java 开发的应用程序，还提供了针对多种语言和框架的客户端，如 Go、Python、Node.js、Spring Cloud 等，以方便各种技术栈的开发者使用。

总体而言，Nacos 是一个功能丰富、易于使用的开源平台，为微服务架构提供了服务发现、配置管理、服务健康监测等核心功能，是构建现代分布式系统的有力工具。

2）安装运行

3）使用nacos 配置消息生产者和消费者

①消息生产者

1. 引入 pom 依赖

```
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2. 写 application.yml

```
server:
  port: 9001

spring:
  application:
    name: nacos-payment-provider
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #配置Nacos地址

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

3. 主启动类

```
//加上注释
@EnableDiscoveryClient
@SpringBootApplication
public class PaymentMain9001
{
    public static void main(String[] args) {
            SpringApplication.run(PaymentMain9001.class, args);
    }
}
```

②消息消费者

1. 引入 pom 依赖

```
<!--高版本spring cloud alibaba已不再引入ribbon, 改用spring cloud loadbalancer -->
<!-- 用于负载均衡 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>

<!--SpringCloud ailibaba nacos -->
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2. 写application.yml

```
server:
  port: 83

spring:
  application:
    name: nacos-order-consumer
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 # 注册中心地址

#消费者将要去访问的微服务名称(注册成功进nacos的微服务提供者)
service-url:
  nacos-user-service: http://nacos-payment-provider
```

3. 主启动类

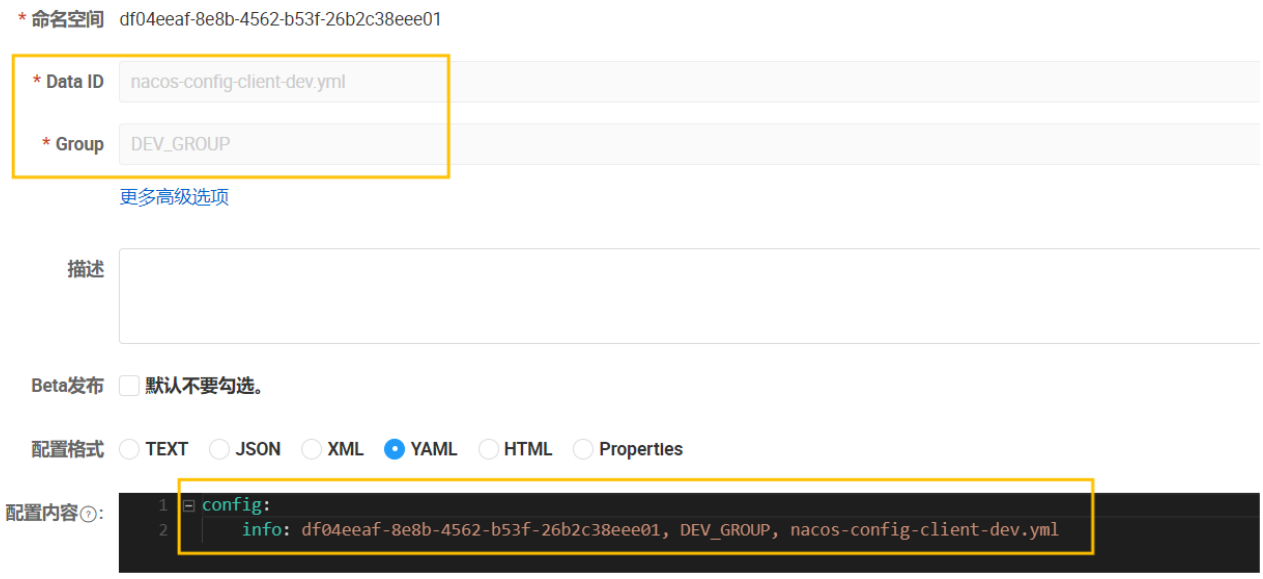
```
@EnableDiscoveryClient
@SpringBootApplication
public class OrderNacosMain83
{
    public static void main(String[] args)
    {
        SpringApplication.run(OrderNacosMain83.class,args);
    }
}
```

4）服务配置中心功能

①新建命名空间

②写配置文件

可以根据不同名，不同组来写不同的配置文件



③写yml

- bootstrap.yml

```
# nacos配置

server:
```

```
  port: 3377

spring:
  application:
    name: nacos-config-client
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #Nacos服务注册中心地址
      config:
        server-addr: localhost:8848 #Nacos作为配置中心地址
        file-extension: yml #指定yaml格式的配置
        group: DEV_GROUP
        namespace: df04eeaf-8e8b-4562-b53f-26b2c38eee01
```

- application.yml

```
spring:
  profiles:
    active: dev
```

配置文件匹配规则：

5）集群 与 持久化

> MySql用于持久化nacos中的配置文件。
>
> 集群搭建参考文章：Linux系统中nacos集群搭建看这一篇就够了_双机 nacos-CSDN博客

在Linux上搭建Nacos集群并使用外部MySQL数据库，以及使用Nginx进行负载均衡，可以按照以下步骤进行操作：

1. 下载Nacos

从Nacos的官方 GitHub 仓库下载最新版本的 Nacos。

```
wget https://github.com/alibaba/nacos/releases/download/2.0.3/nacos-server-2.0.3.tar.gz
tar -zxvf nacos-server-2.0.3.tar.gz
```

2. 配置Nacos集群

编辑Nacos的配置文件 `conf/application.properties`，设置如下属性：

```
# 修改为自己的 MySQL 地址、用户名和密码
spring.datasource.platform=mysql
db.num=1
db.url.0=jdbc:mysql://your-mysql-host:3306/nacos?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
db.user=nacos
db.password=nacos

# 修改自己的端口和 IP 地址
server.port=3333
nacos.server.ip=your-ip
```

3. 启动Nacos节点

分别在不同的节点上启动Nacos服务：

```
cd nacos/bin
./startup.sh
```

4. 配置Nginx负载均衡

安装和配置Nginx，创建一个配置文件（例如 `/etc/nginx/conf.d/nacos.conf`）：

```
upstream cluster{
    server 127.0.0.1:3333;
    server 127.0.0.1:4444;
    server 127.0.0.1:5555;
}

server {
    listen 80;
    server_name nacos.example.com; # 修改为你的域名或IP

    location / {
        proxy_pass http://cluster;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
```

```
}
```

5. 重启Nginx

```
sudo service nginx restart
```

6. 验证

通过访问Nginx的地址，验证Nacos集群是否正常工作。例如，如果你使用了域名 `nacos.example.com`，则在浏览器中输入 `http://nacos.example.com`。

以上步骤提供了一个基本的搭建过程，实际部署中需要根据具体情况进行调整和优化。请注意修改配置中的IP、端口、数据库信息等，以适应你的实际环境。

十二、Sentinel

1）介绍

Alibaba Sentinel（又称Sentinel）是一款由阿里巴巴开源的流量控制和防护组件，旨在**解决分布式系统中的流量防护和流量控制问题**。它是阿里巴巴开源的一部分，专注于**提供高可用性、高稳定性的流量控制和防护解决方案**。Sentinel的目标是帮助开发者确保微服务架构下的应用稳定运行，防止潜在的系统故障和崩溃。

以下是Sentinel的一些关键特性：

1. **实时监控：** Sentinel提供了实时的**流量监控和统计**，可以迅速发现系统中的异常流量和性能问题。
2. **流量控制：** Sentinel支持基于QPS（每秒查询率）的流量控制，可以限制对应用程序的请求流量，防止突发的流量超载。
3. **规则引擎：** Sentinel使用规则引擎来定义流量控制策略，开发者可以根据业务需求自定义规则，灵活应对不同的场景。
4. **熔断降级：** Sentinel支持熔断降级机制，可以在系统压力过大或异常情况下，自动降低对某些服务的调用，防止整体系统崩溃。
5. **系统适配：** Sentinel可以方便地集成到不同的框架和应用中，适用于各种Java应用、Spring Cloud、Dubbo等。
6. **丰富的统计信息：** 提供了丰富的统计信息，包括成功的请求、异常的请求、响应时间等，帮助开发者全面了解系统的运行状态。

Sentinel的开源使得更多的开发者可以使用并贡献代码，有助于构建更加健壮、稳定的分布式系统。在微服务架构中，流量控制和防护是保障系统稳定性的重要组成部分，而Sentinel为开发者提供了一个强大而灵活的工具来应对这些挑战。

2）流控规则

1.介绍

- 资源名：唯一名称，默认请求路径
- 针对来源：Sentinel可以针对调用者进行限流，填写微服务名，默认default（不区分来源）
- 阈值类型/单机阈值：
    - QPS（每秒钟的请求数量）：当调用该api的QPS达到阈值的时候，进行限流
    - 线程数：当调用该api的线程数达到阈值的时候，进行限流
- 是否集群：不需要集群
- 流控模式：
    - 直接：api达到限流条件时，直接限流
    - 关联：当关联的资源达到阈值时，就限流自己
    - 链路：只记录指定链路上的流量（指定资源从入口资源进来的流量，如果达到阈值，就进行限流）【api级别的针对来源】
- 流控效果：
    - 快速失败：直接失败，抛异常
    - Warm Up：根据codeFactor（冷加载因子，默认3）的值，从阈值/codeFactor，经过预热时长，才达到设置的QPS阈值
    - 排队等待：匀速排队，让请求以匀速的速度通过，阈值类型必须设置为QPS，否则无效

Sentinel 的流控规则是**通过定义一组规则来控制系统中的流量**，以确保系统的稳定性和可靠性。这些规则可以基于不同的维度进行配置，如 QPS（每秒查询率）、线程数、并发数等。以下是 Sentinel 流控规则的一些关键概念和配置项：

1. **资源（Resource）：** 流控规则针对的是系统中的资源，可以是一个具体的接口、方法，或者是一个资源的组合。
2. **阈值（Threshold）：** 阈值是用来限制流量的数量，可以是 QPS、线程数、并发数等。当资源的流量达到或超过阈值时，流控规则开始生效。
3. **流控模式（Control Behavior）：** 流控规则可以配置为两种主要的流控模式：直接拒绝（直接拒绝请求，返回错误）和预热模式（阶梯式放行请求，逐渐增加通过的流量）。
4. **关联资源（Association）：** 可以设置关联资源，使得不同资源之间的流量控制关联起来。这样，在一个资源达到流量限制时，可能会影响到与之关联的其他资源。
5. **流控效果（Control Effect）：** 流控规则可以配置为直接拒绝请求，或者通过预热模式逐渐放行请求，以缓解系统的压力。
6. **控制台配置：** Sentinel 提供了图形化的控制台，可以通过控制台进行动态的流控规则配置，方便运维人员实时调整系统的流量控制策略。

以下是一个简单的 Sentinel 流控规则的示例：

- **资源名（Resource）：** com.example.service.UserService
- **阈值（Threshold）：** 10 QPS
- **流控模式（Control Behavior）：** 直接拒绝

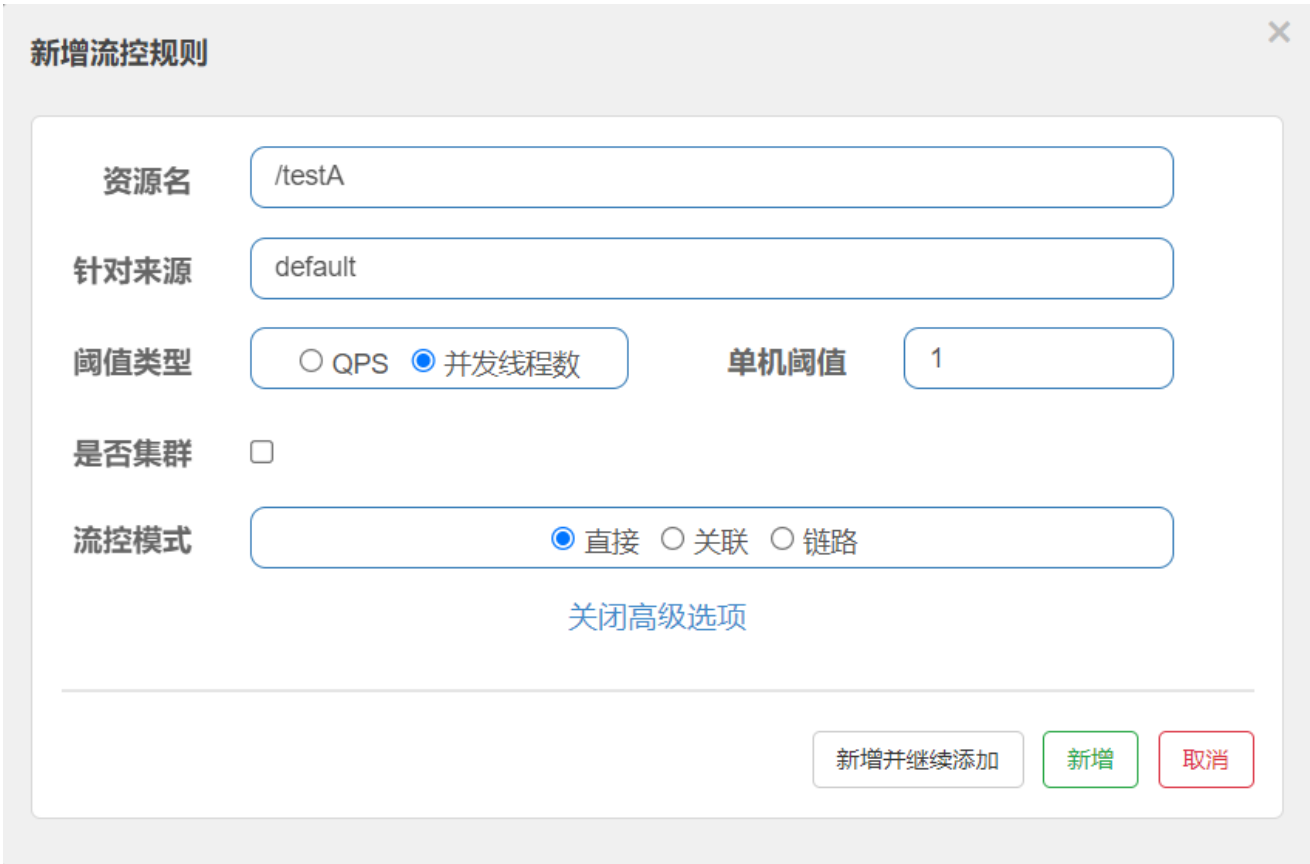
这个规则的含义是，当 UserService 这个资源的 QPS 达到或超过 10 时，直接拒绝多余的请求，确保系统的负载在可接受的范围内。

这样的流控规则可以通过 Sentinel 提供的 API 或者控制台进行配置和管理，使得系统在面临流量激增或异常情况时能够有效地保护自身，防止过载和崩溃。

2. QPS + 直接失败

3.线程数+直接失败

线程数流控是 Sentinel 中一种基于线程数的流量控制方式，它关注的是系统中运行的线程数量，以防止系统超负荷运行，导致性能下降或系统崩溃。在一些高并发的场景下，线程数流控可以有效地限制并发请求，确保系统稳定运行。



4.关联

在 Sentinel 中，流控规则的关联模式是一种机制，用于将不同资源之间的流量控制关联起来。通过关联模式，当一个资源的流控规则触发时，可能会影响到与之关联的其他资源，以便更灵活地管理系统的整体流量。

以下是一些关于流控规则关联模式的关键概念和使用方式：

1. **资源关联（Resource Association）：** 在流控规则中，可以设置关联资源，使得不同资源之间产生关联。这种关联可以是一对一，也可以是一对多的关系。当一个资源的流控规则触发时，关联的资源也可能受到影响。
2. **关联模式（Association Mode）：** 关联模式定义了关联资源之间的流控关系。在 Sentinel 中，关联模式分为直接关联和链路关联两种。
    - 直接关联：当主资源的流控规则触发时，直接关联的资源也受到相同的流控效果。这种模式适用于一对一的关联关系。
    - 链路关联：当主资源的流控规则触发时，链路上的所有关联资源都受到相同的流控效果。这种模式适用于一对多的关联关系，其中关联资源之间存在依赖链路。
3. **关联限流阈值（Associated Limit Applicable）：** 可以配置关联资源的流控阈值。当关联资源的流量超过关联限流阈值时，将触发关联资源的流控效果。
4. **关联触发条件（Association Condition）：** 可以设置关联资源触发条件，即在什么情况下关联资源的流控规则生效。这有助于更精细地控制关联资源的流控策略。

使用关联模式的一个典型案例是在微服务架构中，当某个服务出现异常或流量激增时，可能会影响到与之相关的其他服务。通过配置关联模式，可以使得这种影响在系统层面得到控制，防止异常传播到整个系统。

示例：

假设有两个服务，A 和 B，它们之间存在关联关系。当 A 的流控规则触发时，希望限制 B 的流量。

- **主资源名（Main Resource）：** com.example.service.AService
- **关联资源名（Associated Resource）：** com.example.service.BService
- **关联模式（Association Mode）：** 直接关联
- **关联限流阈值（Associated Limit Applicable）：** 30 QPS

这个配置的含义是，当 AService 的 QPS 达到或超过阈值时，直接限制 BService 的 QPS 不超过 30，以防止异常或高流量扩散到 BService。

5.预热 效果

在 Sentinel 中，流控规则中的预热（Warm Up）效果是一种流控模式，它用于在系统刚启动或系统负载突然增加时，逐渐放宽对流量的限制，以防止系统被大量请求拖垮。预热模式可以有效地缓解系统的压力，使得系统能够更平稳地处理流量波动。

以下是一些关于预热效果的关键概念和使用方式：

1. **流控模式（Control Behavior）：** 在流控规则中，预热效果是通过选择预热模式来实现的。预热模式与直接拒绝模式相对，直接拒绝模式是在达到阈值后立即拒绝多余的请求，而预热模式则会逐渐增加通过的请求。

2. **预热时长（Warm Up Period）：** 预热时长是指从流控规则开始生效到达到阈值的过程中所经过的时间。在预热模式下，系统会逐渐增加通过的请求，而预热时长就是这个逐渐增加的时间段。
3. **阈值（Threshold）：** 预热模式下，阈值表示系统最终允许的流量上限。在预热时长内，系统逐渐增加通过的请求，直到达到最终的阈值。
4. **冷加载因子（Cold Factor）：** 冷加载因子表示在预热时长内，每秒允许通过的请求数的增加速率。通过调整冷加载因子，可以控制预热时长内逐渐增加的速率。
5. **关联资源（Association）：** 预热模式也支持关联资源，即可以将预热效果应用于与主资源关联的其他资源。

使用预热效果的一个典型场景是系统启动阶段，当系统刚刚启动或者从负载较低的状态突然增加到高负载状态时，直接将系统的流量限制到固定的阈值可能导致性能问题。通过配置预热模式，可以在系统启动时逐渐增加通过的请求，使系统能够平稳过渡到高负载状态，提高系统的稳定性和性能。

示例：

- **流控模式（Control Behavior）：** 预热模式
- **预热时长（Warm Up Period）：** 5 秒
- **阈值（Threshold）：** 100 QPS
- **冷加载因子（Cold Factor）：** 3

这个配置的含义是，在预热时长内，每秒允许通过的请求数以每秒3个的速率逐渐增加，直到达到最终的阈值 100 QPS。这样可以确保系统在启动时逐渐适应增加的流量，防止系统启动阶段的压力过大。

6.排队等待效果

在 Sentinel 中，排队等待效果是一种流控模式，也称为排队等待机制（Queue Waiting）。这个机制**允许在流量超过阈值时，请求不会被立即拒绝，而是被放入一个队列中等待执行。**这样可以有效地缓解短时间内的流量峰值，防止系统被突发的大量请求拖垮，同时确保请求按照一定的速率被处理。

以下是排队等待效果的一些关键概念和使用方式：

1. **流控模式（Control Behavior）：** 在流控规则中，排队等待效果是通过选择排队等待模式来实现的。排队等待模式与直接拒绝模式相对，直接拒绝模式是在达到阈值后立即拒绝多余的请求，而排队等待模式则允许请求被放入队列等待执行。
2. **队列大小（Queue Size）：** 队列大小表示可以等待执行的请求的最大数量。当队列已满时，新的请求将被立即拒绝。
3. **超时时间（Max Timeout）：** 超时时间表示请求在队列中等待执行的最大时间。如果请求在超时时间内没有得到执行，将被认为是超时请求，可以根据配置的超时策略进行处理。
4. **流控效果（Control Effect）：** 在排队等待模式下，请求会被放入队列等待执行，直到资源可用。这种方式可以确保资源在有限的队列中按照一定速率被处理，防止系统瞬时被大量请求压垮。

使用排队等待效果的一个典型场景是在系统的处理能力有限、无法立即响应大量请求的情况下。通过配置排队等待模式，可以让请求在队列中等待执行，而不是立即被拒绝，从而更好地利用系统资源，提高系统的稳定性。

示例：

- **流控模式（Control Behavior）：** 排队等待模式
- **队列大小（Queue Size）：** 100
- **超时时间（Max Timeout）：** 2000 毫秒

这个配置的含义是，当流量超过阈值时，最多允许将 100 个请求放入队列等待执行，每个请求最大等待执行时间为 2000 毫秒。这样可以确保系统在瞬时的流量高峰期，通过排队等待方式逐渐处理请求，防止系统被突发的大量请求拖垮。

3）降级规则

1.介绍

在 Sentinel 中，降级规则用于在系统出现异常或超过流量限制时，临时降低某个资源的可用性，以防止整个系统受到影响。降级规则是 Sentinel 流控策略的一部分，通过定义规则来设置降级策略。

以下是 Sentinel 降级规则的一些关键概念和配置项：

1. **资源（Resource）：** 降级规则针对的是系统中的某个具体资源，可以是一个接口、方法，或者其他需要进行流量控制的业务逻辑单元。
2. **降级阈值（Threshold）：** 降级阈值是用来触发降级的条件，通常表示资源的 QPS（每秒查询率）或响应时间阈值。当资源的 QPS 或响应时间超过降级阈值时，降级规则开始生效。
3. **降级模式（Degrade Behavior）：** 降级规则可以配置为直接降级或者渐进降级两种模式。
    - 直接降级：当资源达到降级阈值时，直接拒绝请求或返回预设的降级结果。
    - 渐进降级：当资源达到降级阈值时，系统将逐渐减小通过的请求比例，直到最终达到降级比例。这种模式下，可以逐渐降低对资源的压力，而不是突然拒绝所有请求。
4. **降级比例（Degrade Rate）：** 在渐进降级模式下，降级比例表示允许通过的请求占总请求的比例。例如，降级比例为 0.5 表示允许通过的请求占总请求的一半。

5. **最小请求数（Min Request Amount）：** 在渐进降级模式下，最小请求数表示在进行降级比例计算时，至少需要有多少个请求作为参考。如果请求数量未达到最小请求数，降级规则将不会生效。
6. **统计时长（Stat Interval）：** 统计时长表示触发降级规则的统计时间窗口，系统在该时间窗口内统计资源的 QPS 和响应时间。

通过降级规则，Sentinel 可以在系统压力过大、出现异常或超过流量限制时，自动降低对某个资源的请求，防止整个系统崩溃。这是一种保护机制，确保系统在异常情况下依然能够提供稳定的服务。

示例：

- **资源名（Resource）：** com.example.service.UserService
- **降级阈值（Threshold）：** 1000 QPS
- **降级模式（Degrade Behavior）：** 渐进降级
- **降级比例（Degrade Rate）：** 0.5
- **最小请求数（Min Request Amount）：** 5
- **统计时长（Stat Interval）：** 1 分钟

这个配置的含义是，当 UserService 的 QPS 超过 1000 时，系统将逐渐降低通过的请求比例，直到降低到总请求的一半。这样可以在系统压力大时，逐渐降低对 UserService 的请求压力，确保系统的稳定性。

2.慢比例调用

3.异常比例


编辑熔断规则
资源名
/testD
熔断策略
慢调用比例
异常比例
异常数
比例阈值
0.2
熔断时长
1
s
最小请求数
5
统计时长
1000
ms
在1000ms内，至少要有5个请求，
且异常比例达到0.2，就熔断 1s
保存
取消

4.异常数


编辑熔断规则
资源名
/testE
熔断策略
慢调用比例
异常比例
异常数
异常数
2
熔断时长
3
s
最小请求数
5
统计时长
1000
ms
在统计时长1000ms内，要求至少有5个请求，
且异常数达到2个时，就会熔断3s
保存
取消

4）热点规则

1.介绍

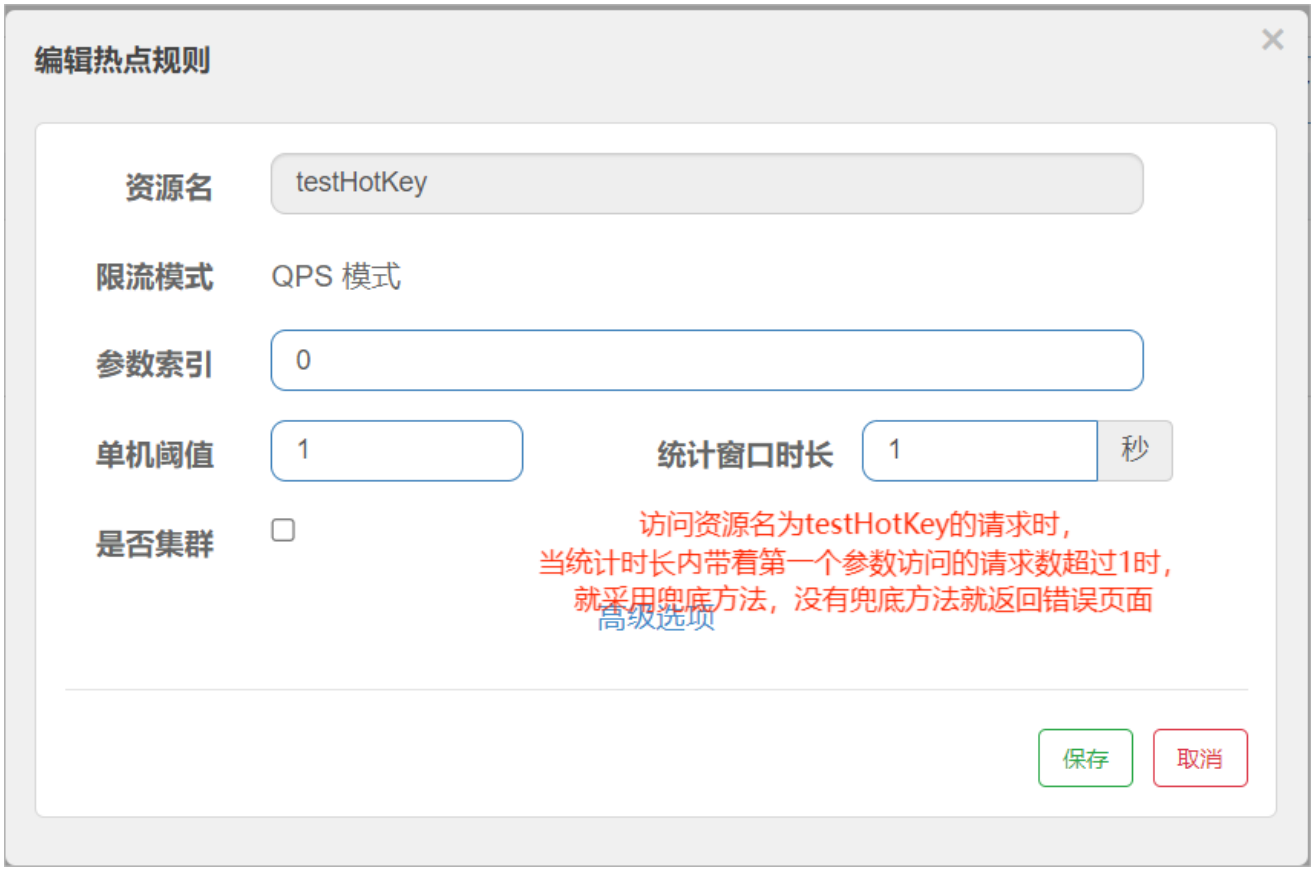
Sentinel 是一款阿里巴巴开源的流量控制和服务保护的框架，用于解决分布式系统中的流量控制、熔断降级、系统负载保护等问题。Sentinel 的热点规则是其流量控制的一种重要策略，用于对特定资源的访问进行限制，防止由于某个资源的过度访问而导致系统不稳定或崩溃。

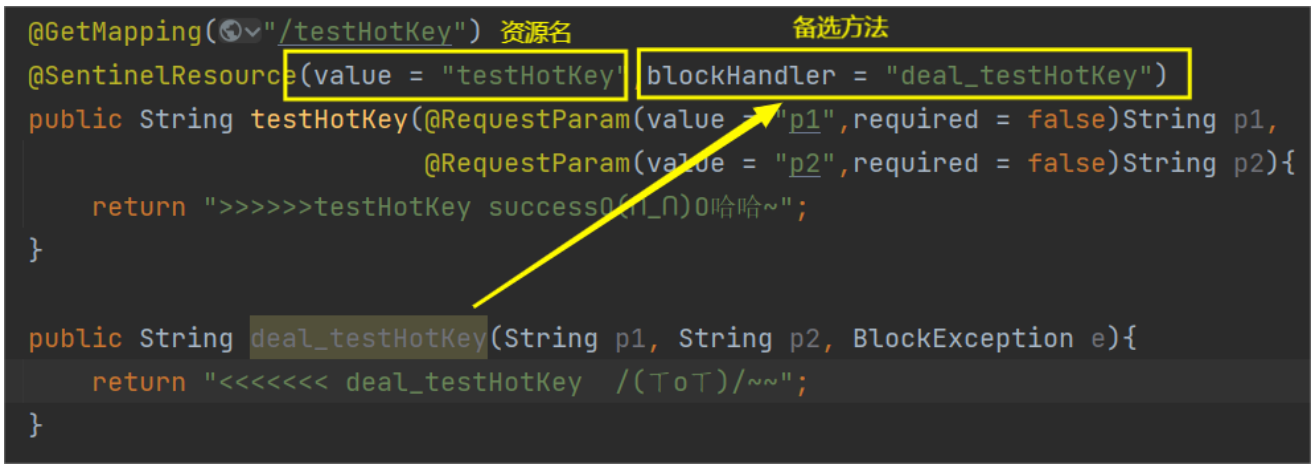
以下是 Sentinel 热点规则的主要特点和用法：

1. **热点参数：**
    - 热点规则针对的是某个特定的“热点参数”，这个参数可以是业务上的某种标识，比如用户 ID、商品 ID 等。热点参数是触发规则的关键。
2. **流控模式：**
    - 热点规则支持两种流控模式：QPS（每秒钟的请求次数）和并发线程数。你可以选择其中一种或两种同时生效。
3. **限制阈值：**
    - 可以设置每秒钟的请求数或并发线程数的阈值。当热点参数的访问超过这个阈值时，规则将触发。
4. **冷却时间：**
    - 规定了在触发了热点规则后的冷却时间。在冷却时间内，即使热点参数的访问再次超过阈值，规则也不会立即再次触发，以防止因为短时间内的波动而导致频繁触发。
5. **触发策略：**
    - 触发了热点规则后，可以选择不同的触发策略，比如直接拒绝请求、慢启动模式、匀速器模式等。
6. **参数例外项：**
    - 可以配置例外项，使得某些特定的热点参数不受规则限制，这在一些特殊情况下可能会很有用。
7. **自定义逻辑（扩展项）：**
    - Sentinel 提供了自定义逻辑的扩展点，可以通过实现接口来定义自己的规则触发逻辑。
8. **动态规则：**
    - Sentinel 支持通过控制台或 API 动态修改热点规则，而不需要重启应用。

使用热点规则可以在高并发场景下，对访问某个特定资源的请求进行限制，防止因为某个资源的过度访问而影响整个系统的稳定性。配置热点规则需要根据具体的业务场景和访问模式进行调整，确保合理有效地控制流量。

2.备选方法


编辑热点规则
资源名
testHotKey
限流模式
QPS 模式
参数索引
0
单机阈值
1
统计窗口时长
1
秒
是否集群
访问资源名为testHotKey的请求时，
当统计时长内带着第一个参数访问的请求数超过1时，
就采用兜底方法，没有兜底方法就返回错误页面
高级选项
保存
取消


@GetMapping("/testHotKey") 资源名
备选方法
@SentinelResource(value = "testHotKey", blockHandler = "deal_testHotKey")
public String testHotKey(@RequestParam(value = "p1",required = false)String p1,
@RequestParam(value = "p2",required = false)String p2){
return ">>>>>>testHotKey success(￣_￣)O哈哈~";
}
public String deal_testHotKey(String p1, String p2, BlockException e){
return "<<<<<<< deal_testHotKey /(ㄒoㄒ)/~~";
}

3.热点规则的例外项

注意：

```
@SentinelResource(value = "testHotKey",blockHandler = "deal_testHotKey")
```

这个 blockHandler 只会处理不符合在sentinel配置的规则的请求，不会处理程序出现的异常。

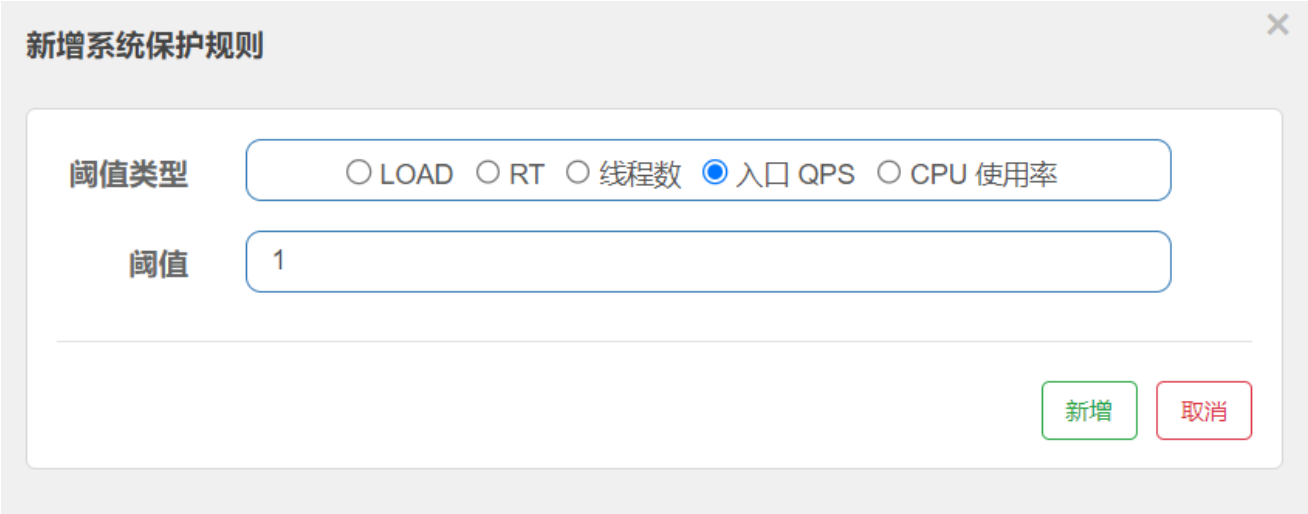
5）系统规则

系统保护规则是从应用级别的入口流量进行控制，从单台机器的总体 Load、RT、入口 QPS 和线程数四个维度监控应用数据，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。

系统保护规则是应用整体维度的，而不是资源维度的，并且**仅对入口流量生效**。入口流量指的是进入应用的流量（`EntryType.IN`），比如 Web 服务或 Dubbo 服务端接收的请求，都属于入口流量。

系统规则支持以下的阈值类型：

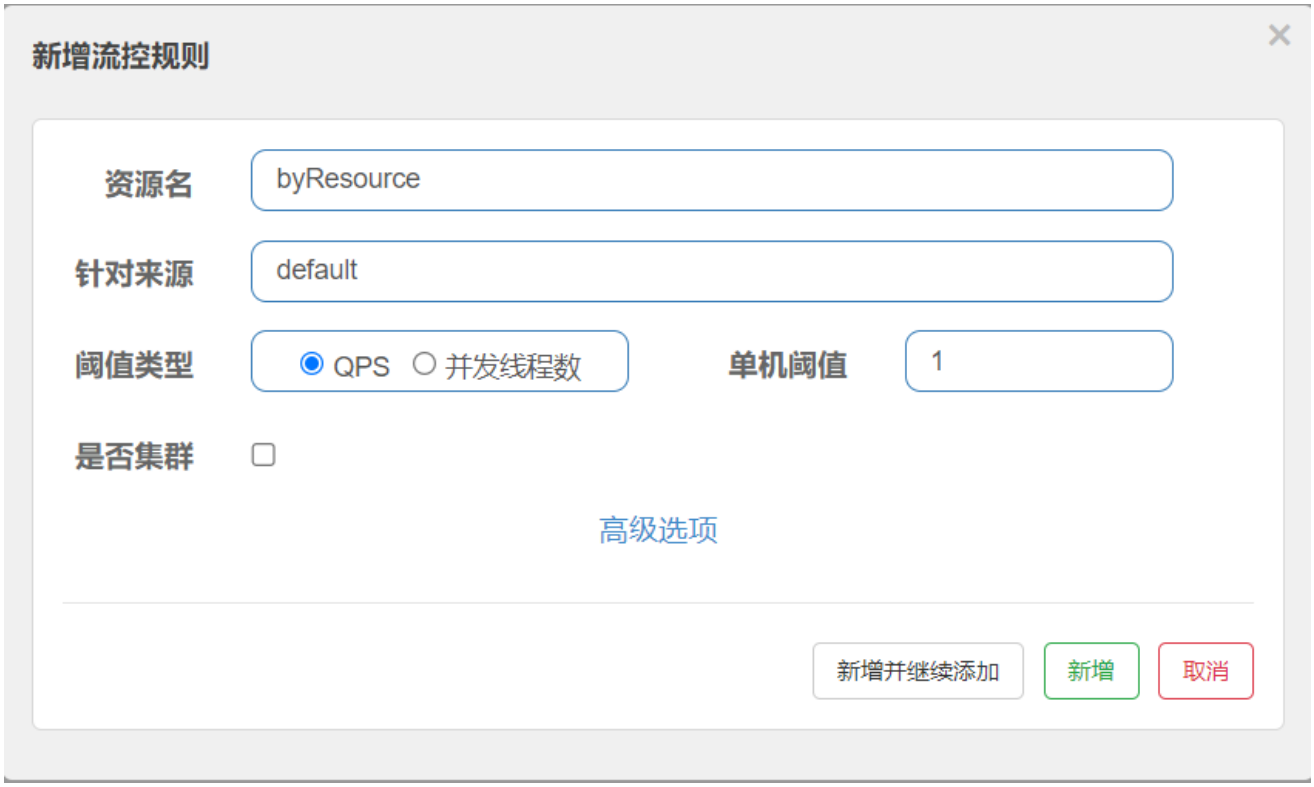
- **Load**（仅对 Linux/Unix-like 机器生效）：当系统 load1 超过阈值，且系统当前的并发线程数超过系统容量时才会触发系统保护。系统容量由系统的 `maxQps * minRt` 计算得出。设定参考值一般是 `CPU cores * 2.5`。
- **CPU usage**（1.5.0+ 版本）：当系统 CPU 使用率超过阈值即触发系统保护（取值范围 0.0-1.0）。
- **RT**：当单台机器上所有入口流量的平均 RT 达到阈值即触发系统保护，单位是毫秒。
- **线程数**：当单台机器上所有入口流量的并发线程数达到阈值即触发系统保护。
- **入口 QPS**：当单台机器上所有入口流量的 QPS 达到阈值即触发系统保护。



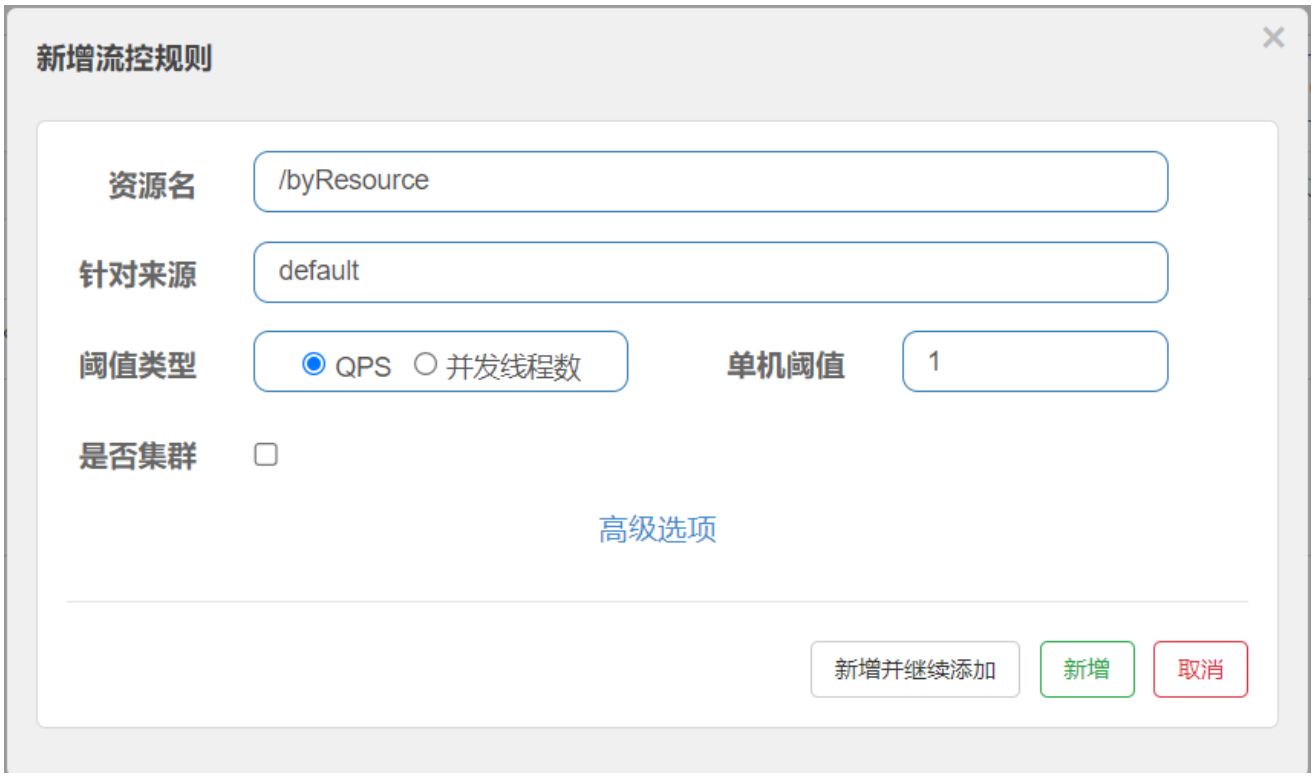
6）@SentinelResource 注释

1. 资源名规则

```
@GetMapping("/byResource")
@SentinelResource(value = "byResource",blockHandler = "handleException")
public CommonResult byResource()
{
  return new CommonResult(200,"按资源名称限流测试OK",new Payment(2020L,"serial001"));
}
public CommonResult handleException(BlockException exception)
{
  return new CommonResult(444,exception.getClass().getCanonicalName()+"\t 服务不可用");
}
```



2. 地址配置规则



3.全局备选方法

①创建一个 handler 类

②写逻辑处理方法

```
public class CustomerBlockHandler {

    public static CommonResult exceptionHandler(BlockException ex){
        return new CommonResult(4444," exceptionHandler >>>> 1");
    }

    public static CommonResult exceptionHandler2(BlockException ex){
        return new CommonResult(4444," exceptionHandler >>>> 2");
    }
}
```

③在@SentinelResource 注释中 配置 类和方法

以下为，如果不符合sentinel配置的规则，就选`CustomerBlockHandler`类中的`exceptionHandler2`方法为备选方法

```
@GetMapping("/byHandler")
@SentinelResource(value = "ByHandler",
                  blockHandlerClass = CustomerBlockHandler.class,
                  blockHandler = "exceptionHandler2")
public CommonResult byHandler(){
  return new CommonResult(200,"按资源名称限流测试OK",new Payment(2020L,"serial002"));
}
```

4.fallback

```
@SentinelResource(value = "fallback",fallback = "handlerFallback")
```

当发生异常或者不符合sentinel规则的时候，都用fallback的备选方法。

5.blockHandler

```
@SentinelResource(value = "fallback",blockHandler = "myBlockHandler")
```

只对不符合 sentinel 规则的请求使用备选方法：myBlockHandler

6.fallback + blockHandler

```
@SentinelResource(value = "fallback", fallback = "handlerFallback", blockHandler =
"myBlockHandler")
```

如果同时发生java 异常 和 不符合 sentinel 规则，优先用blockHandler，因为sentinel是大门。

7.exceptionsToInnore

```
@SentinelResource(value = "fallback", fallback = "handlerFallback", blockHandler =
"myBlockHandler", exceptionsToIgnore = NullPointerException.class)
```

就是遇到 NullPointerException 异常时，不使用备选方法，直接报错。

7）Feign + Sentinel的备选方法

① Feign 服务接口

```
@FeignClient(value = "nacos-payment-provider",fallback = FallbackFeignServiceImpl.class)
public interface FeignService {

    @GetMapping(value = "/paymentSQL/{id}")
    CommonResult<Payment> paymentSQL(@PathVariable("id") Long id);
}
```

②Feign 服务接口实现类，也是备选方法处理逻辑

```
@Component  //这个注释别忘了
public class FallbackFeignServiceImpl implements FeignService {
    @Override
    public CommonResult<Payment> paymentSQL(Long id) {
        return new CommonResult<>(44444,"这是备选方法[Feign]版,
FallbackFeignServiceImpl《《《《");
    }
}
```

③controller

```
@Resource
private FeignService feignService;
@GetMapping(value = "/consumer/paymentSQL/{id}")
CommonResult<Payment> paymentSQL(@PathVariable("id") Long id){
  return feignService.paymentSQL(id);
}
```

8）持久化规则

1.说明

在 Sentinel 中，持久化是指**将规则配置持久化到外部存储介质，以便在应用重启后能够保留规则配置。**这有助于在应用程序启动时能够加载之前的规则配置，从而维持规则的连续性。目前，Sentinel 支持将规则配置持久化到本地文件系统、Nacos、Apollo 等。

2.实现

将sentinel的配置规则都持久化进nacos中：

①pom

```
<!--SpringCloud ailibaba sentinel-datasource-nacos -->
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

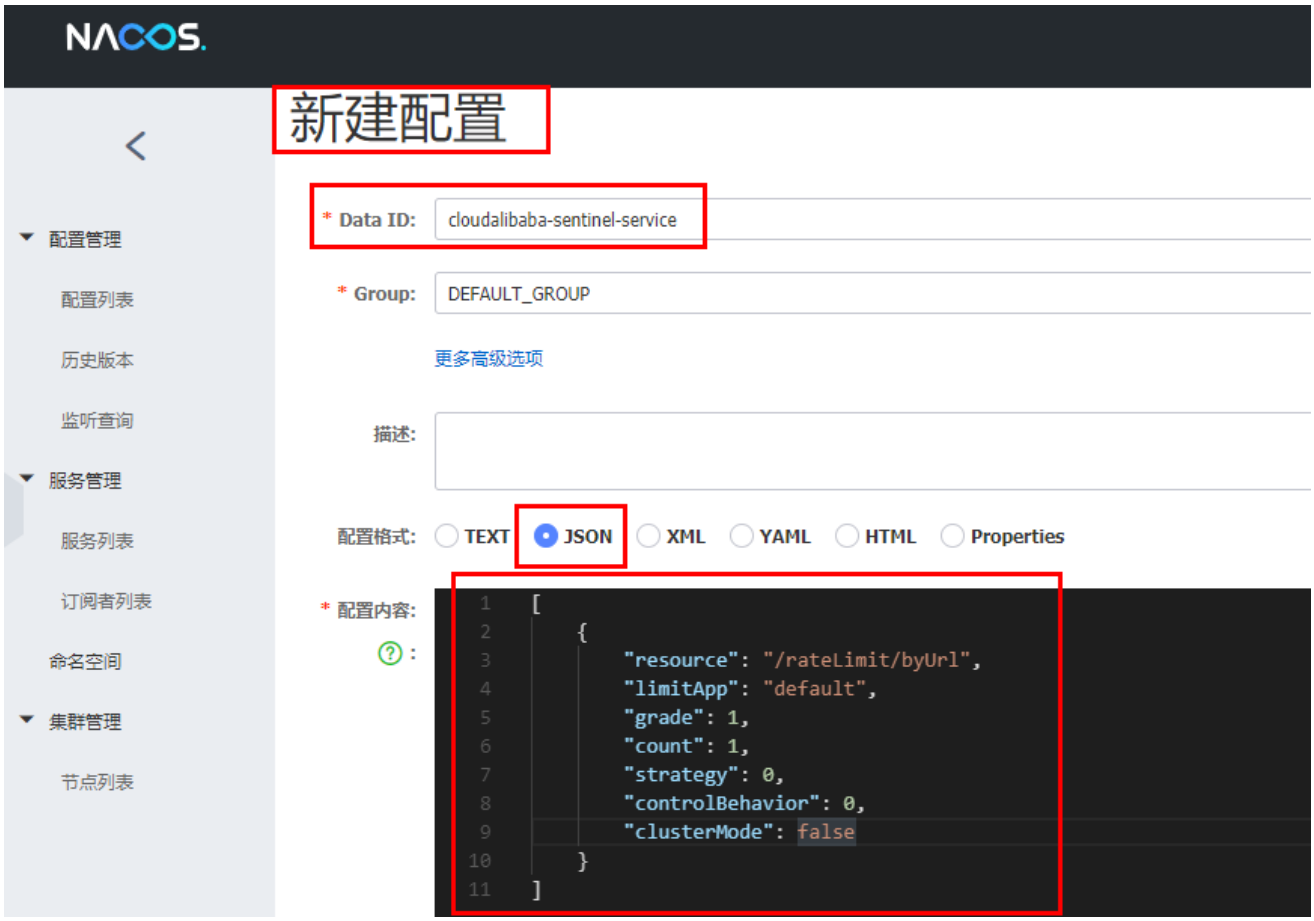
②yml

```
server:
  port: 8401

spring:
  application:
    name: cloudalibaba-sentinel-service
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    sentinel:
      transport:
        dashboard: localhost:8080
        port: 8719
      datasource:    # 主要是这个datasource
        ds1:
          nacos:
            server-addr: localhost:8848
            dataId: cloudalibaba-sentinel-service
            groupId: DEFAULT_GROUP
            data-type: json
            rule-type: flow

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

feign:
  sentinel:
    enabled: true # 激活Sentinel对Feign的支持
```

③在nacos中新建配置

```
[
    {
        "resource": "/rateLimit/byUrl",
        "limitApp": "default",
        "grade": 1,
        "count": 1,
        "strategy": 0,
        "controlBehavior": 0,
        "clusterMode": false
    }
]
```

> resource：资源名称； limitApp：来源应用； grade：阈值类型，0表示线程数，1表示QPS； count：单机阈值； strategy：流控模式，0表示直接，1表示关联，2表示链路； controlBehavior：流控效果，0表示快速失败，1表示Warm Up，2表示排队等待； clusterMode：是否集群。



十三、Seata

1）介绍

Seata（Simple Extensible Autonomous Transaction Architecture）是一种开源的分布式事务解决方案，旨在解决分布式事务的一致性问题。Seata 提供了一套完整的分布式事务解决方案，包括全局事务的协调（Transaction Coordinator）、分支事务的管理（Transaction Manager），以及支持不同存储介质的分布式事务日志存储。

Seata 的核心组件包括：

1. **Transaction Coordinator（TC）：**
   - TC是全局事务的协调者，负责全局事务的协调和控制。
   - TC协调各个参与者的事务操作，确保全局事务的一致性。
   - 全局事务可以包含多个分支事务。
2. **Transaction Manager（TM）：**
   - TM是分布式事务的管理者，负责全局事务的启动、提交和回滚。
   - TM协调TC和各个分支事务，确保全局事务的正确执行。
   - Seata 支持两种事务模型：AT（TCC）模型和 Saga 模型。
3. **Resource Manager（RM）：**
   - RM是分支事务的管理者，负责分支事务的具体执行。
   - RM将本地事务的执行结果通知给TM，协助TM进行全局事务的最终提交或回滚。
4. **分布式事务日志存储：**
   - Seata 支持不同的分布式事务日志存储方案，如数据库存储、文件存储等。
   - 这些存储方案用于记录全局事务的状态和操作，以保障分布式事务的一致性。

Seata 主要关注两种分布式事务模型：

- **AT（TCC）模型：** 基于悲观锁的两阶段提交，可以确保分布式事务的一致性。AT 模型要求各个参与者提供 Try、Confirm 和 Cancel 三个操作来实现事务的一致性。
- **Saga 模型：** 基于补偿事务的模型，通过一系列的事务步骤来实现分布式事务的一致性。Saga 模型要求各个分支事务提供正向和补偿两个操作。

Seata 的目标是为微服务架构中的分布式事务提供简单、可扩展、自治的解决方案，使开发者能够更容易地实现和管理分布式事务，确保系统的数据一致性。 Seata 可以与多种技术栈集成，包括 Spring Cloud、Dubbo、gRPC 等。

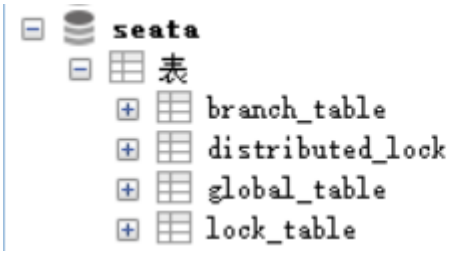
2）seata微服务搭建----服务端

①配置conf文件夹中的application.conf

```
server:
  port: 7091  # Seata服务器端口

spring:
  application:
    name: seata-server  # Spring应用名称

logging:
  config: classpath:logback-spring.xml  # 使用指定的Logback配置文件
  file:
    path: ${log.home:${user.home}/logs/seata}  # 指定日志文件路径

  extend:
```

```
    logstash-appender:
      destination: 127.0.0.1:4560  # Logstash服务器地址
    kafka-appender:
      bootstrap-servers: 127.0.0.1:9092  # Kafka服务器地址
      topic: logback_to_logstash  # Kafka主题

console:
  user:
    username: seata  # 控制台用户名
    password: seata  # 控制台密码

seata:
  config:
    type: nacos  # Seata配置类型为Nacos
    nacos:
      server-addr: 127.0.0.1:8848  # Nacos服务器地址
      namespace:  # Nacos命名空间
      group: SEATA_GROUP  # Nacos组名
      username:  # Nacos用户名
      password:  # Nacos密码
      context-path:  # Nacos上下文路径
      data-id: seataServer.properties  # Nacos数据ID
  registry:
    type: nacos  # Seata注册中心类型为Nacos
    nacos:
      application: seata-server  # Nacos应用名称
      server-addr: 127.0.0.1:8848  # Nacos服务器地址
      group: SEATA_GROUP  # Nacos组名
      namespace:  # Nacos命名空间
      cluster: default  # Nacos集群
  store:
    mode: db  # Seata事务日志存储模式为数据库
    db:
      datasource: druid  # 使用Druid数据源
      db-type: mysql  # 数据库类型为MySQL
      driver-class-name: com.mysql.cj.jdbc.Driver  # MySQL驱动类
      url: jdbc:mysql://127.0.0.1:3306/seata?rewriteBatchedStatements=true  # 数据库连接URL
      user: root  # 数据库用户名
      password: vanky  # 数据库密码
  service:
    vgroupMapping:
      my_test_tx_group: fsp_tx_group  # Seata事务组映射
  security:
    secretKey: SeataSecretKey0c382ef121d778043159209298fd40bf3850a017  # Seata密钥
    tokenValidityInMilliseconds: 1800000  # 令牌有效期（毫秒）
    ignore:
      urls:
/,/**/*.css,/**/*.js,/**/*.html,/**/*.map,/**/*.svg,/**/*.png,/**/*.jpeg,/**/*.ico,/api/v1/auth/
login  # 忽略的URL
```

②到mysql数据库下建seata数据库，到 seata\script\server\db 下打开mysql.sql文件，建立相关表



> Seata 服务端对应的数据库包含四个核心表，用于支持分布式事务的管理和跟踪。这四个表的含义和作用如下：
>
> 1. **Global Table（全局事务表）：**
>   - **表名：** `global_table`
>   - **作用：** 记录全局事务的信息。每个全局事务在这个表中有一条记录，用于标识一个分布式事务的发起。
> 2. **Branch Table（分支事务表）：**
>   - **表名：** `branch_table`
>   - **作用：** 记录分支事务的信息。每个分支事务在这个表中有一条记录，用于标识一个全局事务的参与者，以及记录分支事务的状态（已提交、已回滚等）。
> 3. **Lock Table（分布式锁表）：**
>   - **表名：** `lock_table`
>   - **作用：** 用于支持分布式锁的实现。Seata 在全局事务的提交和回滚阶段需要对资源进行加锁，以确保事务的一致性。这个表用于记录分布式锁的信息。
> 4. **Id Generator Table（ID生成器表）：**
>   - **表名：** `id_generator`
>   - **作用：** 用于生成全局唯一的 XID（XID 是 Seata 中的一个核心概念，代表全局事务的唯一标识）。Seata 使用这个表来生成全局唯一的事务 ID。
>
> 这些表的设计和作用是 Seata 实现分布式事务的关键。通过这些表的协同工作，Seata 能够确保分布式事务的 ACID 特性，实现全局事务的提交和回滚，以及资源的正确加锁和解锁。具体的表结构和字段可能因不同的版本而有所变化，上述说明是基于通用概念的解释。在使用 Seata 时，建议参考具体版本的文档和数据库脚本。

③一个微服务对应一个数据库，每个数据库都应该有undo_log 表

> 在使用 Seata 分布式事务框架时，每个涉及分布式事务的数据库都需要创建一个名为 `undo_log` 的表。这是因为 Seata 通过 Undo Log 机制来实现分布式事务的 ACID 特性（原子性、一致性、隔离性、持久性）。以下是为什么需要 `undo_log` 表的原因：
>
> 1. **Undo Log 机制：** Seata 使用 Undo Log 机制来保障分布式事务的原子性。在事务执行阶段，Seata会将每个参与分布式事务的数据源的变更操作记录到 `undo_log` 表中。这个表的作用是在发生回滚操作时，通过读取 `undo_log` 表的记录来执行相应的撤销（undo）操作，从而回滚之前的事务。
> 2. **分布式事务的原子性：** 当涉及多个数据库时，Seata要确保整个分布式事务要么完全提交，要么完全回滚，以维持 ACID 特性。`undo_log` 表的存在为 Seata 提供了一种在分布式事务回滚时能够撤销之前对数据库的修改的方式。
> 3. **支持多种存储：** `undo_log` 表的设计使得 Seata 能够支持多种存储引擎。Seata本身并不限定具体数据库或存储引擎，因此 `undo_log` 表的结构相对通用，适应了不同数据库的差异。

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` bigint(20) NOT NULL,
  `xid` varchar(100) NOT NULL,
  `context` varchar(128) NOT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  `ext` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `ux_undo_log` (`xid`,`branch_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

3）seata微服务搭建----客户端

①pom

```
<dependencies>
  <!--nacos-->
  <dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
  </dependency>
    <!--负载均衡器-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    <version>4.0.3</version>
  </dependency>
  <!--seata-->
  <dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    <exclusions>
      <exclusion>
        <artifactId>seata-spring-boot-starter</artifactId>
        <groupId>io.seata</groupId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <artifactId>seata-spring-boot-starter</artifactId>
    <groupId>io.seata</groupId>
    <version>1.7.1</version>
  </dependency>
  <!--feign-->
  <dependency>
    <groupId>org.springframework.cloud</groupId>
```

```
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>4.0.3</version>
  </dependency>
  <!--web-actuator-->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <!--mysql-druid-->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.2.20</version>
  </dependency>
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```

②yml配置

```
server:
  port: 2001  # 服务器端口

spring:
  application:
    name: seata-order-service  # Spring应用名称
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848  # Nacos服务发现地址

  datasource:
```

```
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/seata_order     # 当前服务操作的数据库
    username: root
    password: vanky

feign:
  hystrix:
    enabled: false
  client:
    config:
      default:
        connectTimeout: 5000
        readTimeout: 1000

logging:
  level:
    io:
      seata: info  # Seata日志级别设置为info

mybatis:
  mapperLocations: classpath:mapper/*.xml  # MyBatis映射文件位置

seata:
  enabled: true  # 启用Seata分布式事务
  application-id: vanky-abc      # Seata服务名
  tx-service-group: fsp_tx_group     # Seata服务端定义的组
  service:
    vgroup-mapping:
      fsp_tx_group: default
  registry:
    type: nacos
    nacos:
      server-addr: 127.0.0.1:8848  # Nacos服务器地址
      namespace: public
      group: SEATA_GROUP         # Nacos组名
      application: seata-server      # Nacos服务名称
```

③调用其他微服务时，要写其他服务的接口，用openFeign 调用

```
@FeignClient(value = "seata-account-service")
public interface AccountService {
    @PostMapping("/account/decrease")
    CommonResult decrease(@RequestParam("userId")Long userId, @RequestParam("money") BigDecimal
money);
}
```

④当前服务的接口与实现类

@GlobalTransactional 注释：

`@GlobalTransactional(name = "service-order",rollbackFor = Exception.class)`：表示这个服务开启全局事务，要么全部成功，要么全部失败。

rollbackFor 表示的是遇到什么异常时回滚。

```
//接口定义
public interface OrderService {
    void create(Order order);
}

//=============================================================================================
//实现类
@Service
@Slf4j
public class OrderServiceImpl implements OrderService {

    @Resource
    private OrderDao orderDao;

    @Resource
    private AccountService accountService;

    @Resource
    private StorageService storageService;


    @Override
    @GlobalTransactional(name = "service-order",rollbackFor = Exception.class)
    public void create(Order order) {
        log.info("》》》》》开始构建订单《《《《《");
        orderDao.create(order);

        log.info("》》》》构建订单完成，开始减少库存《《《《");
        storageService.decrease(order.getProductId(),order.getCount());

        log.info(">>>> 库存扣减完成，开始修改账户信息 <<<<");
        accountService.decrease(order.getUserId(),order.getMoney());

        log.info(" >>> 账户信息修改完成， 开始修改订单状态 <<< ");
        orderDao.update(order.getProductId(), 0);

        log.info("订单完成！！！");
    }
}
```

⑤controller和mapper正常写