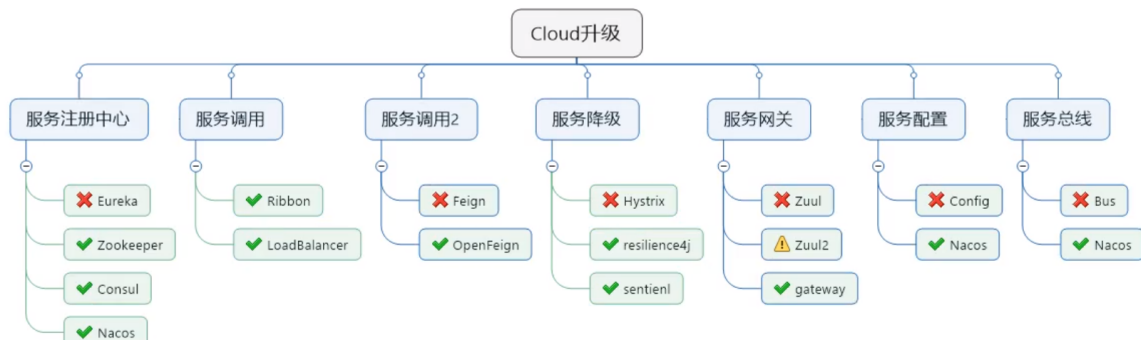


Spring Cloud学习基础

1) Spring Cloud主要组件



2) 父工程Project

- 字符编码，全盘UTF-8。Editor->File Encodings (3个全选，勾选)
- 注解激活。Build->Compiler->Annotation Processors (勾选)
- Java编译版本。Build->Compiler->Java Compiler->Target bytecode version
- File Type过滤。Editor->File Types->Ignored Files and Folders (增加*.idea、*.iml)

3) 父工程pom文件

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.hjy.springcloud</groupId>
  <artifactId>cloud2021</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <!-- 统一管理jar包版本 -->
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <junit.version>4.12</junit.version>
    <log4j.version>1.2.17</log4j.version>
    <lombok.version>1.16.18</lombok.version>
    <mysql.version>8.0.26</mysql.version>
    <durid.version>1.1.16</durid.version>
    <mybatis.spring.boot.version>2.1.1</mybatis.spring.boot.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
```

```

        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-dependencies</artifactId>
        <version>2.3.12.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Hoxton.SR12</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-alibaba-dependencies</artifactId>
        <version>2.2.1.RELEASE</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>${mysql.version}</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>${druid.version}</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>${mybatis.spring.boot.version}</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit.version}</version>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>${log4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
</dependencyManagement>

<build>
    <plugins>
        <plugin>

```

```

        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>2.3.12.RELEASE</version>
        <configuration>
            <fork>true</fork>
            <addResources>true</addResources>
        </configuration>
    </plugin>
</plugins>
</build>
</project>

```

4) 微服务模块构建步骤

- 建立module
- 改POM
- 写yaml
- 主启动
- 业务类
- RestTemplate

RestTemplate: Spring提供的用于访问Rest服务的客户模板工具类

```

@Configuration
public class ApplicationContextConfig {
    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
// restTemplate.postForObject/getForObject等方法

```

5) 热部署Devtools

- 添加devtools到微服务pom

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

```

- 添加插件到父工程pom里

```

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.3.12.RELEASE</version>
      <configuration>
        <fork>true</fork>
        <addResources>true</addResources>
      </configuration>
    </plugin>
  </plugins>
</build>

```

- Idea设置。Setting->Build->Compiler->ADBC ✓
- Setting->Advanced Setting->Compiler->allow automake ✓

6) 工程重构

- 相同可复用代码单独提取到一个工程中，以后直接复用
- 通用jar包pom

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>cn.hutool</groupId>
    <artifactId>hutool-all</artifactId>
    <version>5.7.13</version>
  </dependency>
</dependencies>

```

- 其他微服务引入

```

<dependency>
  <groupId>com.hjy.springcloud</groupId>
  <artifactId>cloud-api-commons</artifactId>
  <version>${project.version}</version>
</dependency>

```

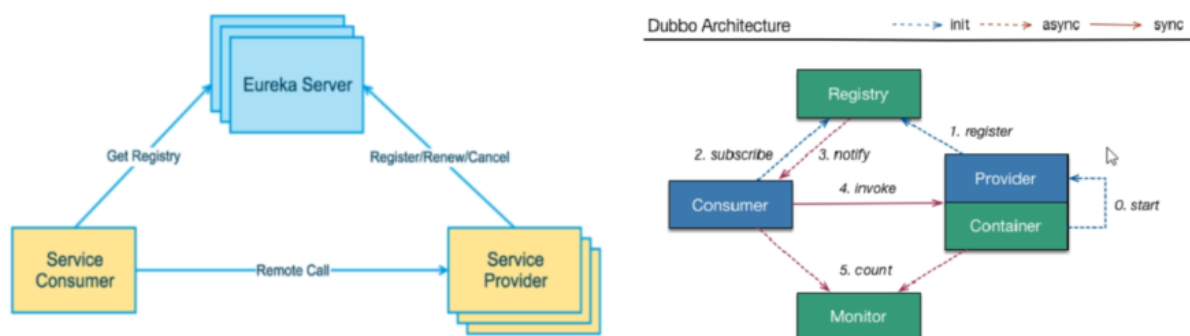
Spring Cloud组件介绍与使用

1) 服务注册与发现

- 服务治理：在传统的rpc远程调用框架中，管理每个服务与服务之间依赖关系比较复杂。所以引入了服务治理观念，管理服务之间的依赖关系，可以实现服务调用、负载均衡、容错等，实现服务治理。
- 服务注册与发现：在服务注册与发现中，有一个注册中心。当服务器启动时，会把当前自己服务器的信息比如服务地址、通讯地址等以别名的方式注册到注册中心上。另一方以该别名的方式去注册中心上获取实际的服务通讯地址，再实现本地RPC调用。

1.1) Eureka

- Eureka使用了CS的设计架构，Eureka Server作为服务注册功能的服务器，它是服务注册中心，而系统中其他为服务，使用Eureka的客户端连接到Eureka Server并维持心跳连接。因此维护人员可通过Eureka Server来监控系统各个微服务是否正常运行。包含Eureka Server和Eureka Client。
- Eureka Server：提供服务注册服务。各个微服务点通过配置启动后，会在Eureka Server中进行注册，这样Eureka Server中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。
- Eureka Client：通过注册中心进行访问。本质是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询负载均衡算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳（默认周期30秒）。如果Eureka Server多个周期内未收到某个节点心跳，将会把它从服务注册表中移除（默认90秒）。



1.1.1) 单机Eureka

- 建立Module，改Pom文件

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <!-- 旧版本 -->
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
    <groupId>com.hjy.springcloud</groupId>
    <artifactId>cloud-api-commons</artifactId>
    <version>${project.version}</version>
  </dependency>
  <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>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
</dependencies>

```

- yml文件

```

server:
  port: 7001
eureka:
  instance:
    hostname: localhost # eureka服务端的实例名称
  client:
    register-with-eureka: false # 表示不像eureka服务端注册自己
    fetch-registry: false # 表示不启用检索服务，自己就是服务端
    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://${eureka.instance.hostname}:${server.port}/eureka/

```

- 主启动

```

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerMain7001 {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerMain7001.class, args);
    }
}

```

- 其他微服务注册进入Eureka

```

<!-- 修改pom, 引入依赖 -->
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <!-- 旧版本
    <artifactId>spring-cloud-starter-eureka</artifactId>
  -->
  <!-- 启动类添加@EnableEurekaClient注解 -->
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

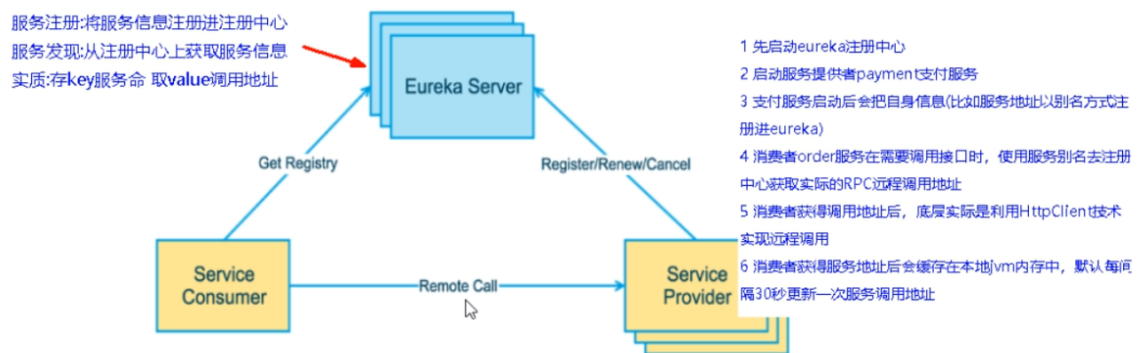
```

```

# yml添加配置
eureka:
  client:
    register-with-eureka: true
    fetchRegistry: true
    service-url:
      # 必须先启动Eureka服务端, 才能注册
      defaultZone: http://localhost:7001/eureka

```

1.1.2) 集群Eureka



- 如单机Eureka一样, 建立多个EurekaServer的module, 但需要修改配置

```

# 7001服务端
eureka:
  instance:
    hostname: eureka7001.com    #eureka服务端的实例名字
  client:
    register-with-eureka: false  #不向注册中心注册自己
    fetch-registry: false       #表示自己就是注册中心, 职责是维护服务实例, 并不需要去检索服务
    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
      #集群之间的Eureka服务端互相注册
      defaultZone: http://eureka7002.com:7002/eureka/

# 7002服务端
eureka:
  instance:
    hostname: eureka7002.com    #eureka服务端的实例名字
  client:
    register-with-eureka: false  #表示不向注册中心注册自己
    fetch-registry: false       #表示自己就是注册中心, 职责是维护服务实例, 并不需要去检索服务
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/

```

- 其他微服务注册进入Eureka集群

```
eureka:
  client:
    register-with-eureka: true
    fetchRegistry: true
    service-url:
      # 向多个eureka集群注册
      defaultZone: http://eureka7001.com:7001/eureka/,
http://eureka7002.com:7002/eureka/
```

- 为防止提供服务的生产者类微服务挂掉导致整个服务宕机，因此建立微服务集群，步骤类似于新建Module，不过使用不同端口号和相同的spring.application.name（Eureka使用这个来注册服务）。

1.1.3) Eureka的负载均衡和使用注册服务

- 之前是通过RestTemplate固定IP地址端口的访问，如 `private static String PAYMENT_URL_PREFIX = "http://localhost:8001/payment/";`，使用Eureka后直接使用 `private static String PAYMENT_URL_PREFIX = "http://CLOUD-PAYMENT-SERVICE/payment/"`。
- 同时，为了让RestTemplate知道所使用的微服务是集群中的哪一个，使用@LoadBalanced注解。

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

1.1.4) 完善actuator信息

- 目前Eureka管理页面中，服务会暴露主机名，且链接未显示端口和ip地址，不便于管理。

```
eureka:
  instance:
    # 服务主机名
    instance-id: payment8001
    # 显示IP地址端口
    prefer-ip-address: true
```

1.1.5) Discovery发现

- 可用于获取注册进入Eureka的微服务信息
- 修改主启动类，启用Discovery服务


```

@SpringBootApplication
@EnableEurekaClient
@EnableDiscoveryClient
public class PaymentMain8001 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain8001.class, args);
    }
}

```

- 修改controller, 通过api获取信息

```

@Resource
private DiscoveryClient discoveryClient;

@GetMapping("/discovery")
public CommonResult<Object> discovery(){
    List<String> services = discoveryClient.getServices();
    List<ServiceInstance> instances =
        discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
    Map<String, Object> result = new HashMap<>();
    result.put("services", services);
    result.put("payment-instances", instances);
    return new CommonResult<>(200, "success", result);
}

```

1.1.6) Eureka自我保护

- 某时刻某一个微服务不可用了, Eureka不会立刻清理, 依旧会对该微服务的信息进行保存 (默认开启)
- 关闭自我保护

```

# Eureka Server端关闭
eureka:
  server:
    enable-self-preservation: false

```

```

# 微服务端修改发送心跳时间
eureka:
  instance:
    # 发送心跳间隔
    lease-renewal-interval-in-seconds: 1
    # 注销间隔
    lease-expiration-duration-in-seconds: 2

```

1.2) Zookeeper

1.2.1) 替代Eureka

- 随着Eureka的停更, Zookeeper成为替代Eureka的一种选择。

1.2.2) 服务注册进入Zookeeper

- 新建微服务工程，引入Zookeeper

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

- 修改配置，注册进入Zookeeper

```
spring:
  cloud:
    zookeeper:
      connect-string: 192.168.29.130:2181
```

- 主启动类依旧使用@EnableDiscoveryClient注解

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

1.2.3) 使用Zookeeper注册服务

- 新建工程，pom, application.yml, 主启动类修改与上面类似
- 配置ApplicationContextConfig的RestTemplate，使用@LoadBalanced负载均衡注解
- controller使用方式与Eureka一致，也是使用服务名称如<http://cloud-payment-service>

1.2.4) Zookeeper服务节点

- Zookeeper服务节点为临时性的，微服务关闭后，经过一定的心跳时间即注销微服务，等到下次重新连接上时再重新注册

1.3) Consul

- Consul是一套开源的分布式服务发现与配置管理系统。提供了微服务的服务治理、配置中心、控制总线等功能，可根据需要单独使用。
- Consul的基本功能：服务发现（HTTP或DNS方式）、健康监测（HTTP、TCP、Docker、Shell脚本定制化）、KV存储（类似于Redis）、多数据中心、可视化WEB界面（对比Zookeeper）

1.3.1) 开启Consul服务端

- 下载consul.exe，打开后输入命令consul agent -dev打开开发模式，在Localhost:8500查看相关信息

1.3.2) 服务注册进入Consul

- 新建Module，引入consul

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

- 修改yml

```
spring:
  application:
    name: cloud-providerconsul-payment
  cloud:
    consul:
      hostname: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
        heartbeat:
          enabled: true
```

- 主启动使用注解@EnableDiscoveryClient

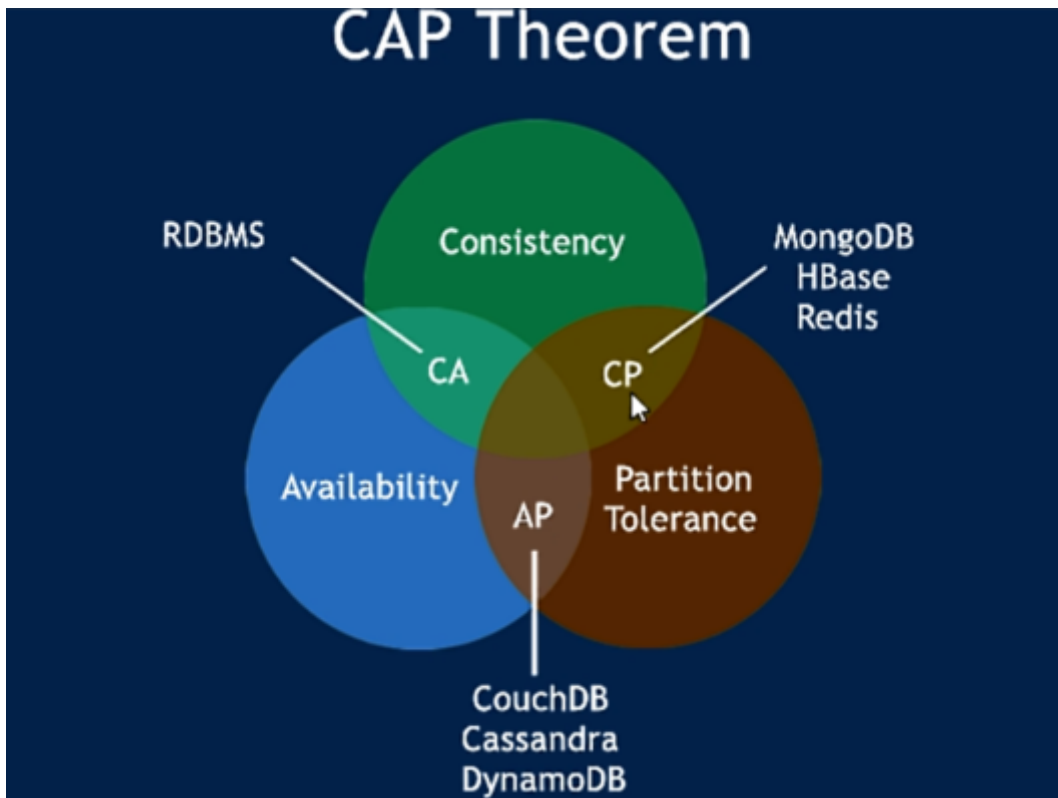
```
@SpringBootApplication
@EnableDiscoveryClient
public class PaymentMain8006 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentMain8006.class, args);
    }
}
```

1.3.3) 使用Consul注册服务

- 新建工程, pom, application.yml, 主启动类修改与上面类似
- 配置ApplicationContextConfig的RestTemplate, 使用@LoadBalanced负载均衡注解
- controller使用方式与Eureka一致, 也是使用服务名称如<http://cloud-payment-service>

1.4) 三个服务注册中心的区别 (根据CAP理论)

- CAP理论: Consistency (强一致性)、Availability (可用性)、Partition tolerance (分区容错)。不可能同时满足CAP三个要求。
- CA-单点集群, 满足一致性、可用性的系统, 通常可扩展性上不强
- CP-满足一致性、分区容错的系统, 通常性能不高 (Zookeeper, Consul)
- AP-满足可用性、分区容错的系统, 通常一致性不高 (Eureka)



2) 服务调用

2.1) Ribbon

- Ribbon是Netflix发布的开源工具，用于客户端的软件负载均衡算法与服务调用。提供了一系列配置项如连接超时，重试等，也可自定义负载均衡算法。目前虽然进入维护阶段了，但core、eureka、loadbalancer等仍在大规模使用，未来考虑使用Spring Cloud LoadBalancer替代。Ribbon=负载均衡+RestTemplate。
- Ribbon本地负载均衡，是在调用微服务接口时，会在注册中心上获取注册信息服务列表之后缓存到JVM本地，从而在本地实现RPC远程调用技术。而Nginx是服务器负载均衡，客户端的所有请求都发到Nginx上，由Nginx进行转发。
- Ribbon属于进程内负载均衡，将负载均衡逻辑集成在消费方进程中，消费方从服务注册中心中获知哪些地址可用，再自行选择；Nginx属于集中式负载均衡，即在消费方和服务方直接提供一个独立的设施，通过该设施负责进行访问的转发与地址的选择。

2.1.1) Ribbon与Eureka结合

- 最新的Eureka里面已经结合了Ribbon，在引入Eureka时就已经引入了Ribbon

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

- RestTemplate的ForObject和ForEntity区别：ForObject返回的是响应体，而Entity则是ResponseEntity，除了响应体还有响应中的其他信息，如响应头、响应状态码等。

```
@GetMapping("/get/{id}")
```

```

public CommonResult<Payment> getPaymentById(@PathVariable("id") Long id){
    return restTemplate.getForObject(PAYMENT_URL_PREFIX + "get/" + id,
CommonResult.class);
}

@GetMapping("/getEntity/{id}")
public CommonResult<Payment> getPaymentEntityById(@PathVariable("id") Long id){
    ResponseEntity<CommonResult> entity = restTemplate.getForEntity(
        PAYMENT_URL_PREFIX + "getEntity/" + id, CommonResult.class);
    if (entity.getStatusCode().is2xxSuccessful()){
        return entity.getBody();
    }else{
        return new CommonResult<>(444, "error");
    }
}
}

```

2.1.2) Ribbon负载均衡规则IRule

- com.netflix.loadbalancer.RoundRobinRule——轮询
- com.netflix.loadbalancer.RandomRule——随机
- com.netflix.loadbalancer.RetryRule——先按照轮询取服务，服务失败时在指定时间内重试
- WeightedResponseTimeRule——轮询的扩展，响应速度越快的加权越大，越容易被选择
- BestAvailableRule——先过滤多次访问故障而处于断路器跳闸的服务，选择并发量最小的服务
- AvailabilityFilteringRule——先过滤故障，在选择并发较小的实例
- ZoneAvoidanceRule——默认规则，复合判断server所在区域的性能和server的可用性选择服务器

2.1.3) Ribbon负载均衡规则的替换

- 需要注意官方明确提示，自定义配置类不也能放在@ComponentScan所扫描的包下，否则该配置类将被所有Ribbon客户端共享，达不到特殊化定制的目的。@SpringBootApplication集成了@ComponentScan，因此自定义规则最好是放在别的包下
- 自定义规则类（不在主要类的包下）

```

package com.hjy.myrule;

import com.netflix.loadbalancer.IRule;
import com.netflix.loadbalancer.RandomRule;
import org.springframework.boot.SpringBootConfiguration;
import org.springframework.context.annotation.Bean;

@SpringBootConfiguration
public class MySelfRule {
    @Bean
    public IRule myRule(){
        return new RandomRule();
    }
}

```

- 启动类添加注解@RibbonClient，指明对应名称服务及其所使用的规则

```

@SpringBootApplication
@EnableEurekaClient
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)
public class OrderMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderMain80.class, args);
    }
}

```

2.1.4) 自定义负载均衡规则

- 先去掉RestTemplate的@LoadBalanced注解，它包含了负载均衡处理，以免影响自定义的规则
- 自定义LoadBalancer接口，并实行

```

public interface LoadBalancer {
    // 从多个可选服务中按照自定义规则选择一个
    ServiceInstance instances(List<ServiceInstance> serviceInstances);
}

```

```

@Component
public class MyLoadBalancer implements LoadBalancer{
    private AtomicInteger atomicInteger = new AtomicInteger(0);

    private final int getAndIncrement(){
        int current;
        int next;
        do {
            current = atomicInteger.get();
            next = current >= 2147483647 ? 0 : current + 1;
        }while (!this.atomicInteger.compareAndSet(current, next));
        return next;
    }

    @Override
    public ServiceInstance instances(List<ServiceInstance> serviceInstances) {
        int index = getAndIncrement() % serviceInstances.size();
        return serviceInstances.get(index);
    }
}

```

- 修改Controller，使用自定义规则选择服务提供者

```

@Resource
private DiscoveryClient discoveryClient;

@Resource
private LoadBalancer loadBalancer;

@GetMapping("/getByMyRule/{id}")
public CommonResult<Payment> getByMyRule(@PathVariable("id") Long id){
    List<ServiceInstance> serviceInstanceList =
        discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");
    ServiceInstance selectedServiceInstance =
        loadBalancer.instances(serviceInstanceList);
    URI uri = selectedServiceInstance.getUri();
}

```

```
return restTemplate.getForObject(uri + "/payment/get/" + id,
CommonResult.class);
}
```

2.2) OpenFeign

- Feign是一个声明式的web服务客户端，只需要创建接口并在接口上添加注解即可使用。通过完成对服务提供方的接口绑定，简化了使用Ribbon时需要自动封装服务调用客户端的开发量。
- Feign集成了Ribbon，也实现了客户端的负载均衡。
- Feign和OpenFeign的区别：
 - Feign是Spring Cloud组件中一个轻量级RESTful的HTTP服务客户端，内置了Ribbon，用来做客户端负载均衡，去调用服务注册中心的服务。
 - OpenFeign是在Feign的基础上支持了Spring MVC的注解，如OpenFeign的@FeignClient可以解析@RequestMapping注解下的接口，并通过动态代理的方法产生实现类，实现类中做负载均衡并调用服务

2.2.1) OpenFeign的使用

- 新建Module，引入依赖（OpenFeign可结合Eureka使用）

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

- 修改yml，连接eureka

```
server:
  port: 80
eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,
http://eureka7002.com:7002/eureka
    register-with-eureka: false
```

- 主启动类添加注解

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class OrderMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderMain80.class, args);
    }
}
```

- 业务类创建接口使用@FeignClient接口，指明服务名称和使用的接口地址

```

@Service
@FeignClient(value = "CLOUD-PAYMENT-SERVICE")
public interface OrderService {
    @GetMapping("/payment/get/{id}")
    CommonResult<Payment> getPaymentById(@PathVariable("id") Long id);
}

```

- Feign用于集成了Ribbon，因此可以使用@RibbonClient修改负载均衡配置

2.2.2) Feign的超时配置

- OpenFeign默认超时1s，1s后未收到回复将返回500错误，Read timed out executing
- 可修改超时配置时间

```

ribbon:
  ReadTimeout: 8000
  ConnectTimeout: 8000

```

2.2.3) Feign配置日志

- Feign提供了日志功能，用于对Http请求的细节进行记录，反馈接口的调用情况
- 日志等级分为四类：
 - NONE：默认的，不显示任何日志
 - BASIC：仅记录请求方法、URL、响应状态码和执行时间
 - HEADERS：除了BASIC的信息，包括请求和响应的头信息
 - FULL：除了HEADERS，包括请求和响应的正文和元数据
- 配置日志，定义日志等级

```

@SpringBootConfiguration
public class FeignConfig {
    @Bean
    Logger.Level feignLoggerLevel(){
        return Logger.Level.FULL;
    }
}

```

- 配置需要开启日志的业务

```

logging:
  level:
    com.hjy.springcloud.service.OrderService: debug

```

3) 服务降级

3.1) Hystrix

- 服务雪崩：由于复杂分布式系统中存在微服务之间的依赖，甚至多层依赖（扇出）。如果扇出的链路上某个微服务调用时间过长或不可用，将导致前一个微服务占用的资源越来越多，逐级影响，最终导致整个系统奔溃，出现雪崩效应。

- Hystrix是一个用于处理分布式系统的延迟和容错的开源库，当依赖出现超时、异常等情况时，保证不会导致整个系统服务失败，避免级联故障，以提高分布式系统的弹性。
- 三个主要概念：
 - 服务降级：当服务器压力剧增的情况下，根据实际业务情况及流量，对一些服务和页面有策略的不处理或换种简单的方式处理，从而释放服务器资源以保证核心业务正常运作或高效运作。说白了，就是尽可能的把系统资源让给优先级高的服务。广泛的意义上，熔断也属于降级的一种。触发降级的条件有程序运行异常、超时、熔断、线程池/信号量满等。
 - 服务熔断：当某个服务单元发生故障时，通过断路器的故障监控，向调用方返回一个符合预期的、可处理的备选响应，而不是长时间等待或抛出无法处理的异常。这样就保证了服务调用方的线程不会被长时间、不必要的占用，从而避免了故障在系统中的蔓延，防止雪崩。
 - 服务限流：在一定时间段内限制服务的请求量以保护系统，主要用于防止突发流量而导致的服务崩溃。

3.1.1) Hystrix服务端降级fallback

- 新建服务提供者Module，引入Hystrix，yml正常注册服务进入Eureka服务端

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

- 服务提供者降级，业务类中。这里添加了@HystrixCommand注解，fallbackMethod为降级的处理方法。其中commandProperties为触发降级的情况。每种情况用@HystrixProperty的键值对表示，键可以在HystrixCommandProperties的源码中找到对应配置的类名。当出现commandProperties中的条件，或者方法本身出现异常时，都会转为降级处理。

```
@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler",
  commandProperties = {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",
      value = "3000")})
public String paymentInfo_TimeOut(Integer id) {
  //int timeUnit = 3;
  int timeUnit = 5;
  // int age = 10 / 0;
  try{
    TimeUnit.SECONDS.sleep(timeUnit);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  return "线程池" + Thread.currentThread().getName() +
    " paymentInfo_TimeOut, id:" + id + ",耗时(s):" + timeUnit;
}

public String paymentInfo_TimeOutHandler(Integer id) {
  return "线程池" + Thread.currentThread().getName() +
    " paymentInfo_TimeOutHandler, id:" + id + ",异常";
}
```

- 启动类添加注解@EnableCircuitBreaker

```

@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public class PaymentHystrixMain8001 {
    public static void main(String[] args) {
        SpringApplication.run(PaymentHystrixMain8001.class, args);
    }
}

```

3.1.2) Hystrix消费端降级fallback

- 新建module引入Hystrix, 修改yml, 启用feign自带的hystrix

```

feign:
  hystrix:
    enabled: true

```

- 启动类添加注解@EnableHystrix

```

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableHystrix
public class OrderHystrixFeignMain80 {
    public static void main(String[] args) {
        SpringApplication.run(OrderHystrixFeignMain80.class, args);
    }
}

```

- 修改controller类, 自定义fallback方法

```

@GetMapping("/getnok/{id}")
@HystrixCommand(fallbackMethod = "paymentInfo_TimeOutHandler",
    commandProperties = {
        @HystrixProperty(name =
"execution.isolation.thread.timeoutInMilliseconds", value = "1500")
    })
String paymentInfo_TimeOut(@PathVariable("id")Integer id){
    return orderService.paymentInfo_TimeOut(id);
}

public String paymentInfo_TimeOutHandler(@PathVariable("id")Integer id){
    return "消费者端超时或异常";
}

```

3.1.3) 默认fallback

- 为了避免给每一个需要降级的方法都指定一个fallback方法, 绝大多数都采用同一个默认的fallback可以有效减少代码量。通过@DefaultProperties指定默认fallback, 需要降级的业务添加@HystrixCommand, 不加其他配置。

```

@RestController
@RequestMapping("/order")
@Slf4j

```

```

@DefaultProperties(defaultFallback = "defaultFallbackMethod")
public class OrderController {
    @Resource
    private OrderService orderService;

    @GetMapping("/getnok/{id}")
    @HystrixCommand
    String paymentInfo_TimeOut(@PathVariable("id") Integer id){
        return orderService.paymentInfo_TimeOut(id);
    }

    public String defaultFallbackMethod(){
        return "默认超时或异常返回";
    }
}

```

3.1.4) fallback与业务代码分开

- 实现feign业务类接口

```

@Component
public class OrderFallbackService implements OrderService {
    @Override
    public String paymentInfo_OK(Integer id) {
        return "ok fallback";
    }

    @Override
    public String paymentInfo_TimeOut(Integer id) {
        return "nok fallback";
    }
}

```

- feign业务接口类添加注解配置

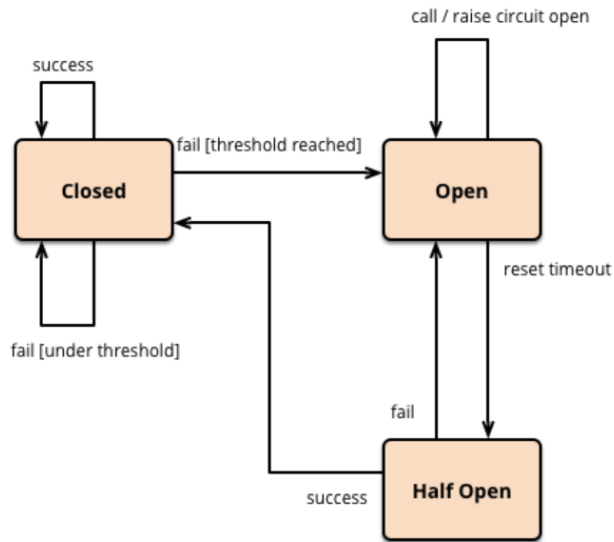
```

@FeignClient(value = "CLOUD-PAYMENT-HYSTRIX-SERVICE", fallback =
OrderFallbackService.class)
public interface OrderService {...}

```

3.1.5) Hystrix熔断

- 新建Module，引入Hystrix，主程序开启@EnableCircuitBreaker熔断器，业务类配置熔断。当10秒内出现10次请求60%的比例出错时，将进行熔断，后续的请求将直接返回fallback信息。熔断后10秒进行一次请求尝试，若请求正常则恢复服务，关闭熔断器，否则将保持熔断状态，持续10秒后再次尝试。



```

@Service
@Slf4j
public class PaymentServiceImpl implements PaymentService {
    @Override
    @HystrixCommand(fallbackMethod = "paymentInfo_break", commandProperties = {
        // 熔断开关
        @HystrixProperty(name = "circuitBreaker.enabled", value = "true"),
        // 请求次数
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value =
"10"),
        // 时间窗口期
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",
value = "10000"),
        // 触发熔断的出错比例
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value =
"60")
    })
    public String paymentInfo_OK(Integer id) {
        if (id < 0){
            throw new RuntimeException("id不能为负数");
        }
        return "线程池" + Thread.currentThread().getName() + " paymentInfo_OK,
id:" + id;
    }

    public String paymentInfo_break(Integer id) {
        return "多次异常，触发熔断";
    }
}

```

3.1.6) Hystirx可视化

- 新建Module, 引入依赖

```

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

```

- 修改yml，配置允许监控路径

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

- 主启动类添加注解启用Hystrix Dashboard

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardMain9001 {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardMain9001.class, args);
    }
}
```

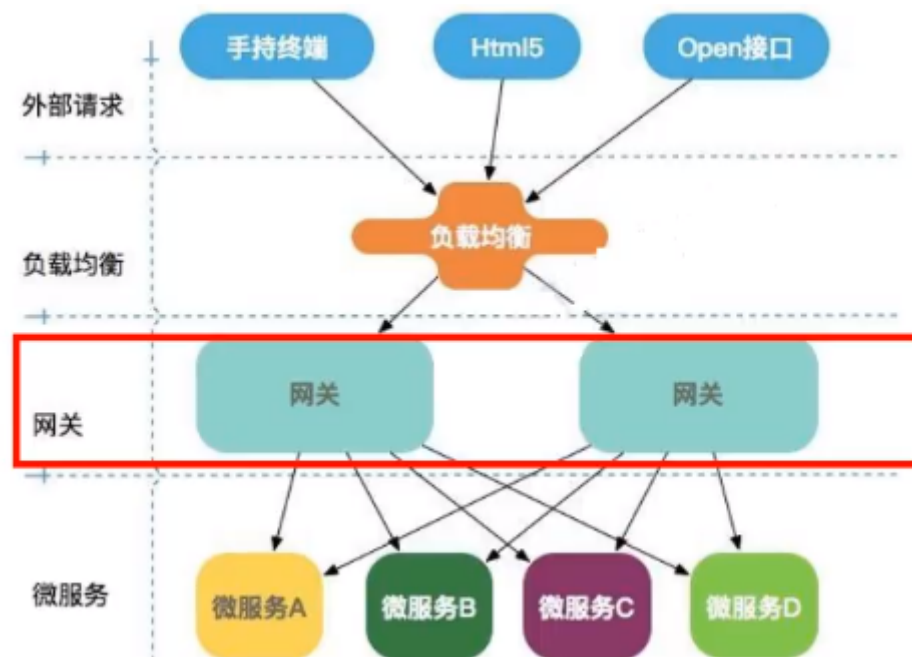
- 接受实时监控的微服务必须依赖spring actuator，并在主启动中添加监控路径

```
@Bean
public ServletRegistrationBean getServlet(){
    HystrixMetricsStreamServlet streamServlet = new
    HystrixMetricsStreamServlet();
    ServletRegistrationBean registrationBean = new
    ServletRegistrationBean(streamServlet);
    registrationBean.setLoadOnStartup(1);
    registrationBean.addUrlMappings("/hystrix.stream");
    registrationBean.setName("HystrixMetricsStreamServlet");
    return registrationBean;
}
```

4) 服务网关

4.1) Gateway

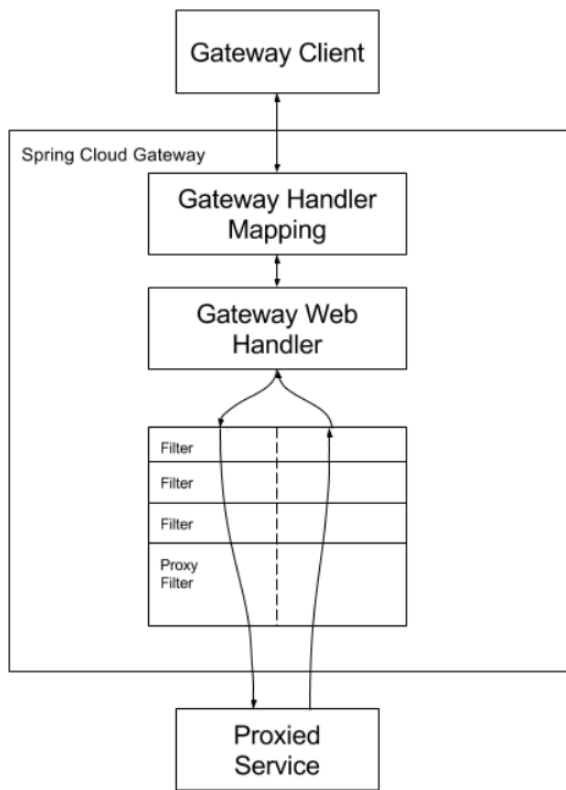
- Gateway是基于Spring 5.0 + Spring Boot 2.0 + Project Reactor等技术开发的网关，基于框架底层使用了高性能的Reactor模式通信框架Netty实现的WebFlux框架，旨在为微服务架构提供一种简单有效的统一的API路由管理方式。包括安全、监控、限流等。
- Gateway提供反向代理、鉴权、流量控制、熔断、日志监控等功能



- Gateway特性
 - 动态路由：能够匹配任何请求属性
 - 可以对路由指定Predicate和Filter，且易于编写
 - 集成Hystirx的断路器功能
 - 集成Spring Cloud服务发现功能
 - 请求限流功能
 - 支持路径重写
- Spring webFlux是Spring 5.0引入的，区别于Spring MVC，它不依赖servlet API，它是完全异步非阻塞的，并且基于Reactor来实现响应流规范。

4.1.1) Gateway的三大主要概念

- 路由Route：是构成网关的基本模块，由ID、目标URI、一系列断言和过滤器组成，若断言为true则匹配该路由
- 断言Predicate：用于匹配请求
- 过滤器Filter：可以在请求被路由前或者之后对请求进行修改。发送代理请求前，过滤器可以做参数检验、权限检验、流泪监控、日志输出、协议转换等。发送代理请求后，过滤器可以做响应内容、响应头的修改、日志输出、流量监控等。



4.1.2) Gateway的使用

- 新建Module, 修改Pom, 引入Gateway, 同时引入eureka, hystrix等

```

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

```

- 新建Yml, 正常注册服务进入eureka里面, 正常创建启动类

```

server:
  port: 9527
spring:
  application:
    name: cloud-gateway-service
eureka:
  client:
    fetch-registry: true
    register-with-eureka: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,
http://eureka7002.com:7002/eureka
    instance:
      hostname: cloud-gateway-service

```

- 新增路由配置方式——yml配置

```

spring:
  cloud:
    gateway:
      routes:
        # id必须唯一

```

```
#此时访问localhost:9527/payment/get/**相当于localhost:8001/payment/get/**
- id: payment_route1
  uri: http://localhost:8001
  predicates:
    - Path=/payment/get/**

- id: payment_route2
  uri: http://localhost:8001
  predicates:
    - Path=/payment/lb/**
```

- 新增路由配置方式——config bean

```
@SpringBootConfiguration
public class GatewayConfig {
    @Bean
    public RouteLocator customRouteLocator1(RouteLocatorBuilder
routeLocatorBuilder){
        RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
        routes.route("custom_route_1", r ->
r.path("/guonei").uri("http://news.baidu.com/")).build();
        return routes.build();
    }

    @Bean
    public RouteLocator customRouteLocator2(RouteLocatorBuilder
routeLocatorBuilder){
        RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
        routes.route("custom_route_2", r ->
r.path("/guoji").uri("http://news.baidu.com/guoji")).build();
        return routes.build();
    }
}
```

4.1.3) Gateway通过微服务名实现动态路由

- uri使用协议lb表示使用负载均衡

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          # 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
          enabled: true
      routes:
        - id: payment_route1
          uri: lb://cloud-payment-service
          predicates:
            - Path=/payment/get/**

        - id: payment_route2
          uri: lb://cloud-payment-service
          predicates:
            - Path=/payment/lb/**
```


4.1.4) Gateway Predicates的使用

- Route Predicates Factories: Spring Cloud Gateway包括许多内置的Route Predicate工厂，所有的这些Predicate都与http请求的不同属性匹配，多个Route Predicate工厂可以进行组合。使用RoutePredicateFactory创建的Predicate对象可以赋值给Route。
- Predicate Factory类配置方式（只展示yaml配置方式）与作用
 - After: 匹配指定时间之后的请求（时间格式为ZonedDateTime类）

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: http://news.baidu.com
          predicates:
            - After=2021-10-15T16:38:07.968+08:00[Asia/Shanghai]
```

- Before: 匹配指定时间之前的请求

```
spring:
  cloud:
    gateway:
      routes:
        - id: before_route
          uri: http://news.baidu.com
          predicates:
            - Before=2021-10-15T16:38:07.968+08:00[Asia/Shanghai]
```

- Between: 匹配指定时间之间的请求

```
spring:
  cloud:
    gateway:
      routes:
        - id: between_route
          uri: http://news.baidu.com
          predicates:
            - Between=2021-10-15T16:38:07.968+08:00[Asia/Shanghai], 2021-10-25T16:38:07.968+08:00[Asia/Shanghai]
```

- Cookie: 匹配请求头的cookie中指定字段值匹配正则表达式的请求

```
spring:
  cloud:
    gateway:
      routes:
        - id: cookie_route
          uri: http://news.baidu.com
          predicates:
            - Cookie=username, .*admin
```

- Header: 匹配请求头的指定字段值匹配正则表达式的请求

```
spring:
  cloud:
    gateway:
      routes:
        - id: header_route
          uri: http://news.baidu.com
          predicates:
            - Header=access-control-allow-methods, POST
```

- Host: 添加允许访问的主机域名，只有匹配的主机域名的请求才允许访问

```
spring:
  cloud:
    gateway:
      routes:
        - id: host_route
          uri: http://news.baidu.com
          predicates:
            - Host=*.baidu.com
```

- Method: 允许访问的请求方式

```
spring:
  cloud:
    gateway:
      routes:
        - id: method_route
          uri: http://news.baidu.com
          predicates:
            - Method=GET, POST
```

- Path: 匹配路径进行转发，例如localhost:9527/guoji转发news.baidu.com/guoji

```
spring:
  cloud:
    gateway:
      routes:
        - id: path_route
          uri: http://news.baidu.com
          predicates:
            - Path=/guoji
```

- Query: 请求url中带有参数且值匹配正则表达式（可选）

```
spring:
  cloud:
    gateway:
      routes:
        - id: query_route
          uri: http://news.baidu.com
          predicates:
            - Query=userId, \d+
```

- RemoteAddr: 指定可访问的IP和掩码（可选）(? 是否包括掩码下同局域网IP ?)

```

spring:
  cloud:
    gateway:
      routes:
        - id: remote_route
          uri: http://news.baidu.com
          predicates:
            - RemoteAddr=192.168.1.1/24, 192.168.1.2/24

```

- Weight: 根据权重分配路由，负载均衡的一种

```

spring:
  cloud:
    gateway:
      routes:
        - id: weightHigh_route
          uri: http://weightHigh.com
          predicates:
            - Weight=group1, 8

        - id: weightLow_route
          uri: http://weightLow.com
          predicates:
            - Weight=group1, 2

```

4.1.5) Gateway Filter的使用

- Route Filter 路由过滤器是用于修改进入的HTTP请求和返回的HTTP响应，只能指定路由进行生效。使用方式和Predicates基本一致，有31种，相关使用可参考spring.io使用。
- Global Filter 全局过滤器，针对全部路由生效。
- 自定义Global Filter，主要实现两个接口GlobalFilter，Ordered。

```

@Component
@Slf4j
public class MyLogGlobalFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        log.info("-----Global Filter My Log-----");
        String username =
exchange.getRequest().getQueryParams().getFirst("username");
        if (StringUtils.isBlank(username)){
            log.info("username is null or empty");
            exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
            return exchange.getResponse().setComplete();
        }
        // 过滤器放行
        return chain.filter(exchange);
    }

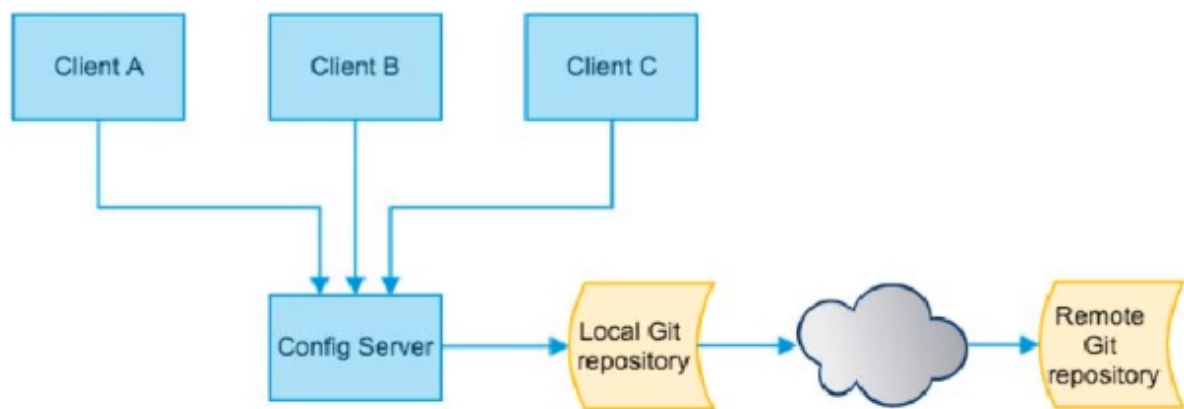
    @Override
    // 指全局过滤器的加载顺序，数值越小优先级越高
    public int getOrder() {
        return 0;
    }
}

```

5) 服务配置

5.1) Config

- Spring Cloud Config提供了一个配置中心，配置服务器Config Server为微服务架构种的微服务提供集中化的外部配置支持。
- Spring Cloud Config分为服务端和客户端两部分。
 - 服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。
 - 客户端通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取配置信息。配置服务器默认采用git来存储配置信息，这样有助于对环境配置进行版本管理，并且可以通过git客户端工具来管理和访问配置内容。



- Spring Cloud Config的作用：
 - 集中管理配置文件
 - 不同环境不同配置，动态化的配置更新，分环境部署比如dev/prod/test/beta/release
 - 运行时动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务回向配置中心统一拉取配置
 - 当配置发生变动时，服务不需要重启即可感知配置的变化并应用
 - 将配置信息以REST接口的形式暴露，post/curl访问刷新

5.1.1) Config服务端配置与使用

- 首先在GitHub上创建仓库，并添加相关的配置文件并上传
- 新建Module，并引入Config

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

- 新建yml，并配置git地址和分支名

```
spring:
  application:
    name: cloud-config-server
  cloud:
    config:
```

```

server:
  git:
    # git地址
    uri: https://gitee.com/jia_hope/cloud2021config.git
    username: ****
    password: ****
    # 搜索的文件夹路径下配置文件
    search-paths:
      - springcloud-config
    # 分支
    label: master

```

- 新建主启动类，使用@EnableConfigServer注解

```

@SpringBootApplication
@EnableConfigServer
@EnableDiscoveryClient
public class ConfigServerMain3344 {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerMain3344.class, args);
    }
}

```

- 修改下host文件，本地测试下效果（不修改直接使用localhost也可以）

```
127.0.0.1 config-3344.com
```

```

// http://localhost:3344/master/application.yml结果如下
{
  "name": "master",
  "profiles": ["application.yml"],
  "label": null,
  "version": "730dd11f8b04903062a34f5570482e92e72ae435",
  "state": null,
  "propertySources": [
    {
      "name": "https://gitee.com/jia_hope/cloud2021config.git/springcloud-
config/application.yml",
      "source": {
        "spring.application.name": "cloud-payment-service",
        "server.port": 8001,
        // ...
      }
    }
  ]
}

```

```

# http://localhost:3344/master/application-prod.yml结果如下
spring:
  application:
    name: cloud-payment-service
server:
  port: 8100
# ...

```

- 配置读取的规则映射

```
/ {application} / {profile} [ / label ]
/ {application} - {profile} . yml
/ {label} / {application} - {profile} . yml (推荐方式)
/ {application} - {profile} . properites
/ {label} / {application} - {profile} . properties
```

5.1.2) Config客户端配置与使用

- 新建Module, 引入config

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

- 新建系统级别配置文件bootstrap.yml。相对比application是用户级的配置, bootstrap是系统级的, 优先级更高。Spring会根据此新建一个Bootstrap Context, 作为Application Context的父上下文。默认情况下不会被本地配置所覆盖, 可以保证两者配置的分离。**注: 如果bootstrap.yml和application-prod同时定义了server.port, 根据加载顺序, 最终会是application的生效**

```
spring:
  application:
    name: cloud-config-client
  cloud:
    config:
      # git仓库分支
      label: master
      # 配置文件名称
      name: application
      # 配置文件后缀
      profile: prod
      # config服务端地址
      uri: http://localhost:3344
```

- 随便写个业务类来测试下 (test.version在git仓库的application-prod.yml上)

```
@RestController
@RequestMapping("/config")
public class ConfigController {
    @Value("${test.version}")
    private String version;

    @GetMapping("/getVersion")
    public String getVersion(){
        return version;
    }
}
```

5.1.3) Config客户端动态刷新

- 由于Config服务端是可以直接去取git的数据的，所以直接更改git上的配置文件，服务端能第一时间响应。但是，客户端是在开始启动时去从服务端接收配置的，因此git上做了修改，它并不会第一时间感应到，除非重启。为了解决上述问题，可以实现Config客户端的动态刷新
- 确定Config客户端引入了actuator

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

- 修改yml，通过actuator暴露刷新接口

```
management:
  endpoints:
    web:
      exposure:
        # 暴露所有接口
        include: "*"

```

- 修改需要动态刷新配置的业务类，使用@RefreshScope注解

```
@RestController
@RequestMapping("/config")
@RefreshScope
public class ConfigController {
    @Value("${test.version}")
    private String version;

    @GetMapping("/getVersion")
    public String getVersion(){
        return version;
    }
}
```

- 通过Post请求进行刷新

```
http://localhost:3355/actuator/refresh
```

- Spring Cloud Config配合Spring Cloud Bus使用可以实现配置的动态刷新

6) 服务总线

6.1) Bus

- Spring Cloud Bus连接分布式系统的节点与轻量级的消息代理，能被用于管理和传播分布式系统间的消息（如配置修改）。同时，Bus也可作为微服务节点之间的通信管道，目前支持RabbitMQ和Kafka。

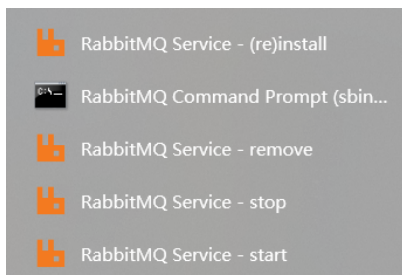
- 在微服务架构的系统中，通常会使用轻量级的消息代理来构建一个共用的消息主题，并让系统中的所有微服务都连接上来。由于该主题中所产生的消息被所有微服务监听和消费，所以称它为消息总线。
- ConfigClient都监听MQ中的同一个Topic（默认是springCloudBus）。当一个服务刷新数据时，它会把消息放入这个Topic中，这样其他监听同一个Topic的服务就能得到通知，并更新自身的配置。

6.1.1) RabbitMQ环境配置

- RabbitMQ是实现了高级消息队列协议（AMQP）的开源消息代理软件（亦称面向消息的中间件），是用Erlang语言编写的。因此需要先下载Erlang环境，再下载RabbitMQ软件。
- 输入以下命令启动管理功能，添加可视化插件

```
rabbitmq-plugins enable rabbitmq_management
```

```
F:\RabbitMQ\rabbitmq_server-3.9.8\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@DESKTOP-B53974U:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DESKTOP-B53974U...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
started 3 plugins.
```



- 打开后输入localhost:15672，密码guest

Overview

Queued messages last minute

Currently idle

Message rates last minute

Currently idle

Global counts

Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0

Name	File descriptors	Socket descriptors	Erlang processes	Memory	Disk space	Uptime	Info	Reset stats
rabbit@DESKTOP-B53974U	0 65536 available	0 58893 available	380 1048576 available	62 MiB 4.7 GiB high watermark	109 GiB MiB low watermark	18m 59s	basic disc 1 rss	This node All nodes

Churn statistics

Ports and contexts

Export definitions

Import definitions

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

6.1.2) Bus的广播使用

- 参照Config客户端新建一个客户端，测试广播功能。
- 通过Bus总线去刷新配置两种方式：
 - 利用消息总线，触发一个客户端/bus/refresh，而刷新所有客户端的配置
 - 利用消息总线，触发一个服务端/bus/refresh，而刷新所有客户端的配置
 - 考虑到微服务的职责应该单一性，客户端本身应该主攻业务类，不应该承担刷新职责，所以选择服务端来触发刷新才是更推荐的方法。
- 给Config服务端和客户端添加消息总线支持

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

```
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest

management:
  endpoints:
    web:
      exposure:
        include: "bus-refresh"
```

- 向服务端发送Post刷新请求，实现广播，处处都通知到

```
http://localhost:3344/actuator/bus-refresh
```

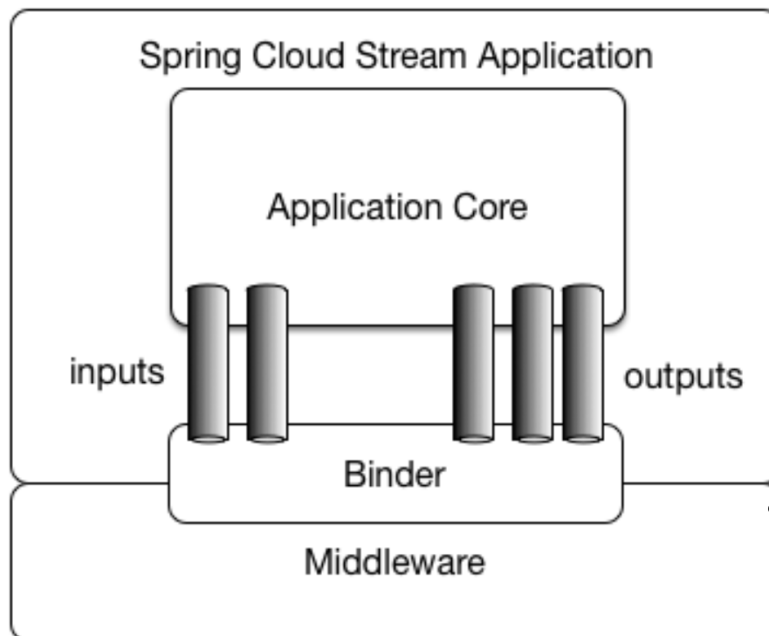
6.1.3) Bus的定点通知

- 指定实例

```
http://localhost:3344/actuator/bus-refresh/{destination}
http://localhost:3344/actuator/bus-refresh/cloud-config-client:3355
```

7) 消息驱动 Spring Cloud Stream

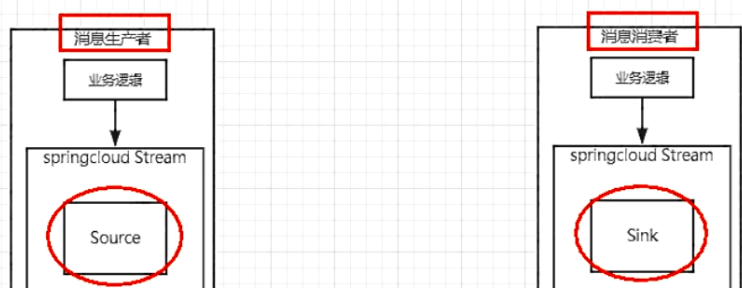
- Spring Cloud Stream是一个构建与共享消息系统连接的高可扩展的事件驱动微服务的框架。应用程序通过inputs和outputs与Spring Cloud Stream中的binder对象进行交互，而binder对象负责与消息中间件（RabbitMQ或Kafka）进行交互，inputs对应消费者，outputs对应生产者。通过定义绑定器Binder作为中间层，实现了应用程序与消息中间件细节直接的隔离。
- Spring Cloud Stream通过使用Spring Integration来连接消息代理中间件以实现消息事件驱动，同时引入了发布-订阅、消费组、分区等概念来屏蔽中间件之间的差异。因此，使用Spring Cloud Stream可屏蔽底层消息中间件的差异。



- Binder绑定器，Channel消息通道，Source发送端的流，Sink接收端的流

7.1) Spring Cloud Stream的使用 (RabbitMQ)

- 新建消息生产者Module，引入相关依赖



```
<dependency>

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

- 新建生产者配置文件，配置rabbitMQ以及消息通道等

```
spring:
  application:
    name: cloud-stream-rabbitmq-provider
  cloud:
    stream:
      binders: # 绑定中间件
      defaultRabbit: # 定义的中间件名称
      type: rabbit # 中间件类型
      environment: # 配置当前中间件的环境
      spring:
        rabbitmq:
          host: localhost
          port: 5672
          username: guest
          password: guest
```

```

bindings: # 绑定消息通道
  output: # 消息通道的名称, 可自定义, 不一定非要output
  destination: testExchange # 消息主题
  content-type: application/json # 消息类型, 文本则设置为text/plain
  binder: defaultRabbit # 设置具体绑定的消息服务代理

eureka:
  client:
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka,
http://eureka7002.com:7002/eureka
  instance:
    instance-id: send-8801.com # 显示名称
    lease-renewal-interval-in-seconds: 2 # 心跳时间
    lease-expiration-duration-in-seconds: 5 # 持续时间
    prefer-ip-address: true # 是否显示Ip

```

- 实现生产者的发送信息方法, 通过@EnableBinding绑定是发送的类型 (Source.class) 还是接收的类型 (Sink.class)

```

@EnableBinding(Source.class)
public class MessageProviderImpl implements IMessageProvider {
    @Resource
    private MessageChannel output;

    @Override
    public void send() {
        String serial = UUID.randomUUID().toString();
        output.send(MessageBuilder.withPayload(serial).build());
    }
}

```

- 新建消费者Module, 参照生产者Module, 引入依赖, 修改配置文件

```

spring:
  cloud:
    stream:
      bindings: # 绑定消息通道
      input: # 消息通道的名称, 可自定义, 不一定非要input
      destination: testExchange # 消息主题
      content-type: application/json # 消息类型, 文本则设置为text/plain
      binder: defaultRabbit # 设置具体绑定的消息服务代理

```

- 新建消费者接受消息的处理业务, 绑定Sink, 使用@StreamListener绑定消息通道

```

@Component
@EnableBinding(Sink.class)
@Slf4j
public class MessageServiceImpl implements IMessageService {
    @Value("${server.port}")
    private String port;

    @Override
    @StreamListener(Sink.INPUT)
    public void receive(Message<String> message) {
        log.info("receive : " + message + ", port : " + port);
    }
}

```

- 消息的分组与持久化配置

```

spring:
  cloud:
    stream:
      bindings: # 绑定消息通道
        input: # 消息通道的名称，可自定义，不一定非要input
          destination: testExchange # 消息主题
          content-type: application/json # 消息类型，文本则设置为text/plain
          binder: defaultRabbit # 设置具体绑定的消息服务代理
          group: group1

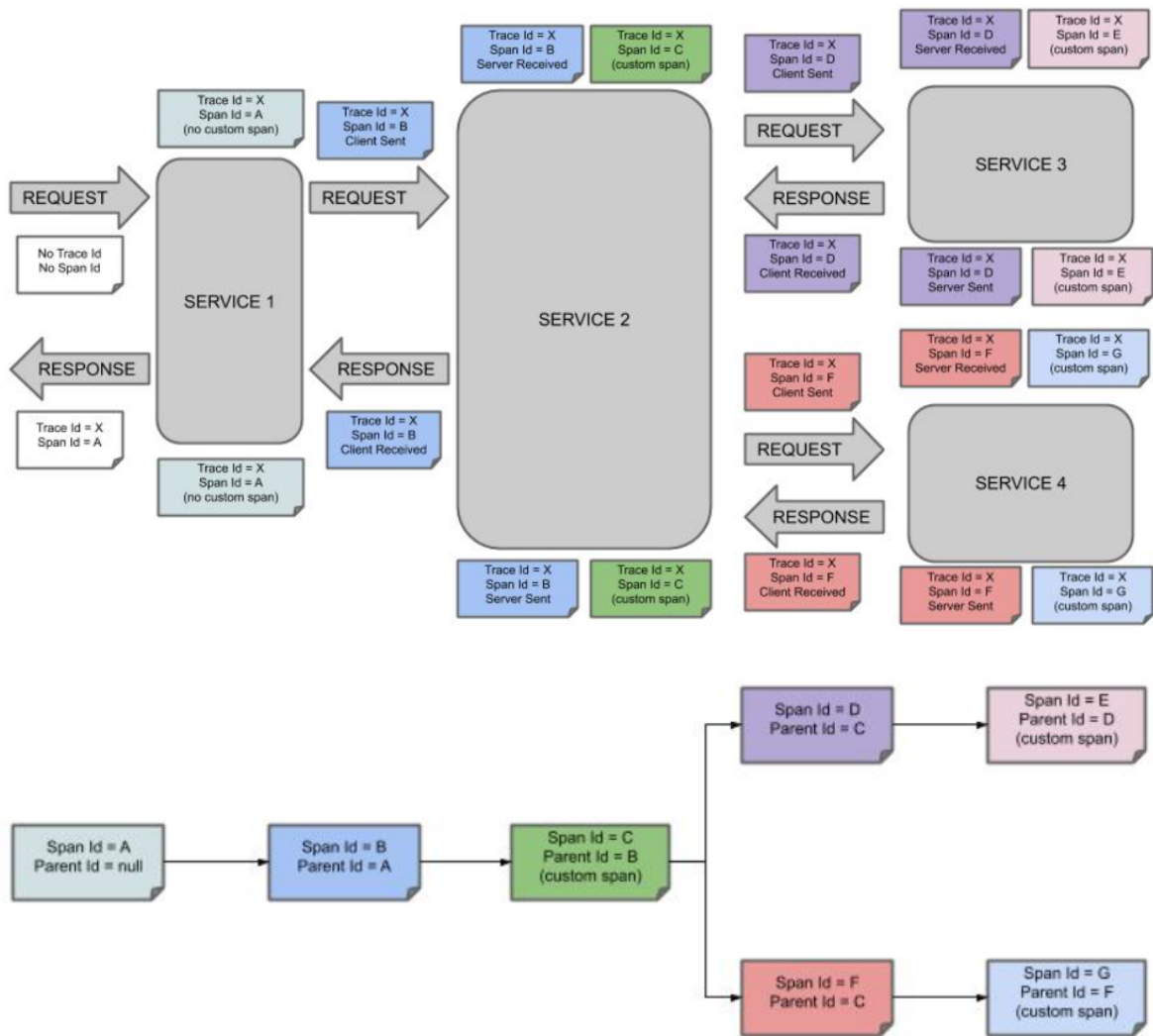
```

8) 分布式请求链路追踪Spring Cloud Sleuth

- 微服务框架中，一个由客户端发起的请求在后端系统中会经过多个不同的服务节点调用来协同产生最终的结果。每一个前端的请求都会形成一条复杂的分布式调用链路，链路中的每一环出现高延时或错误都会引起整个请求最后的失败。
- Spring Cloud Sleuth提供了一套完整的服务跟踪的解决方案，在分布式系统中提供追踪解决方案并且兼容支持了zipkin（一款开源的分布式实时数据追踪系统，主要功能是聚集来自各个异构系统的实时监控数据）。帮助提供快速定位服务故障点的能力。

8.1) Zipkin

- Zipkin的两个主要概念：
 - Span：表示调用链路来源，也可以理解为span就是一次请求信息，通过parentId进行链接上下级span
 - Trace：类似于树结构的span集合，表示一条调用链路，存在唯一标识



- 在<https://zipkin.io/>的QuickStart上寻找Java下的latest release下载可执行jar包文件（如zipkin-server-2.23.4-exec.jar），执行

```
java -jar zipkin-server-2.23.4-exec.jar
```

- 运行控制台 <http://localhost:9411/zipkin>

8.2) Spring Cloud Sleuth

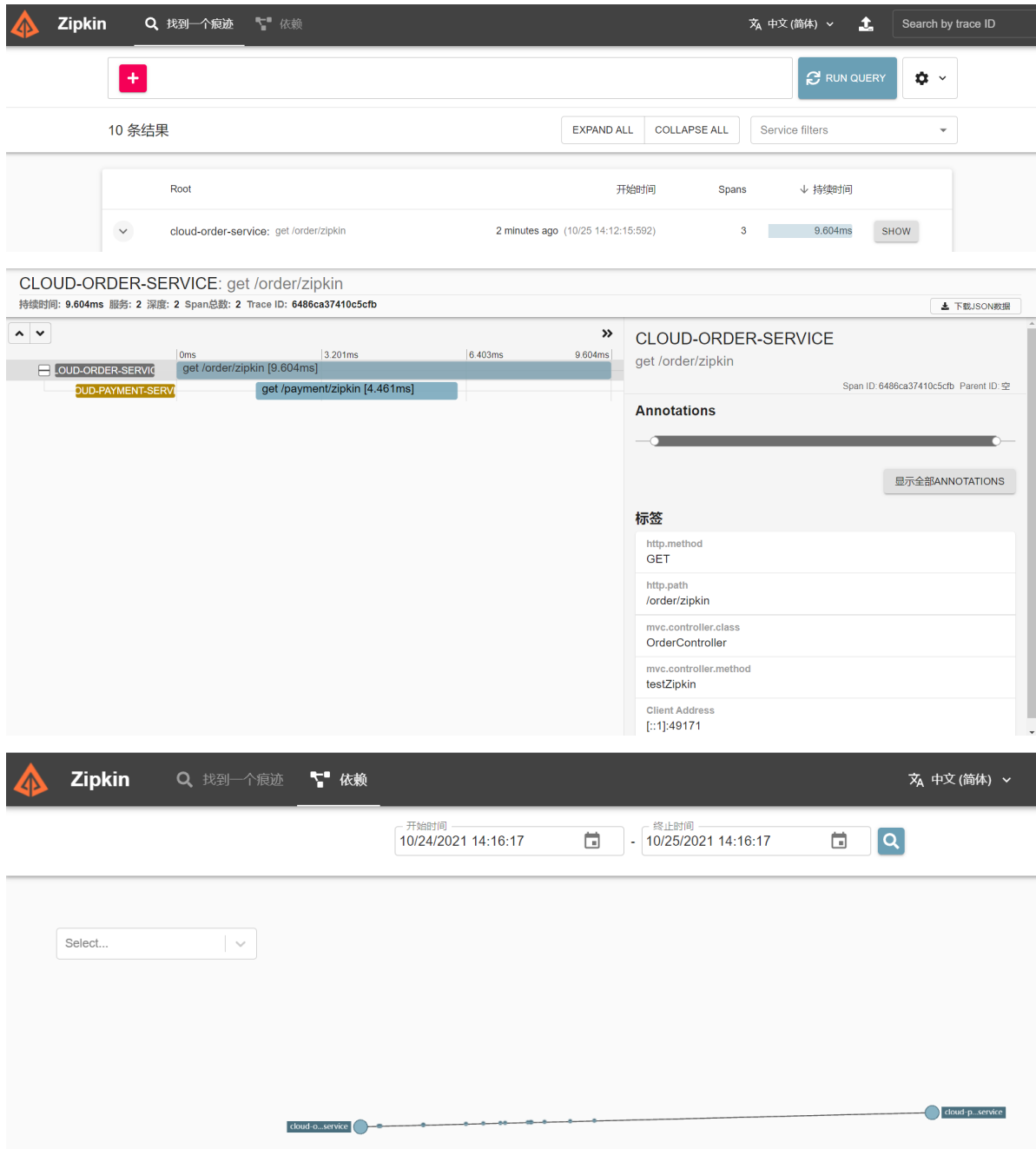
- module引入相关依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <!-- 包含spring cloud sleuth和zipkin -->
  <artifactId>spring-cloud-starter-zipkin</artifactId>
</dependency>
```

- 增加相关配置

```
spring:
  zipkin:
    base-url: http://localhost:9411 # zipkin服务器地址
  sleuth:
    sampler:
      probability: 1 # 多少概率的请求链路会被记录, 1表示100%, 默认0.1表示10%
```

- 进行前端请求，查看zipkin控制台信息



Spring Cloud Alibaba

1) Spring Cloud Alibaba介绍

1. 由于Spring Cloud的主要组件提供者Netflix公司对Spring Cloud组件进入维护状态，且不再更新。在此背景下，阿里巴巴于2018年开始发布自己的Spring Cloud组件，并持续进行更新与维护，成为替代Spring Cloud Netflix组件的更优选择
2. 目前Spring Cloud Alibaba提供的功能如下：

主要功能

- **服务限流降级**: 默认支持 WebServlet、WebFlux、OpenFeign、RestTemplate、Spring Cloud Gateway、Zuul、Dubbo 和 RocketMQ 限流降级功能的接入，可以在运行时通过控制台实时修改限流降级规则，还支持查看限流降级 Metrics 监控。
- **服务注册与发现**: 适配 Spring Cloud 服务注册与发现标准，默认集成了 Ribbon 的支持。
- **分布式配置管理**: 支持分布式系统中的外部化配置，配置更改时自动刷新。
- **消息驱动能力**: 基于 Spring Cloud Stream 为微服务应用构建消息驱动能力。
- **分布式事务**: 使用 @GlobalTransactional 注解，高效并且对业务零侵入地解决分布式事务问题。
- **阿里云对象存储**: 阿里云提供的海量、安全、低成本、高可靠的云存储服务。支持在任何应用、任何时间、任何地点存储和访问任意类型的数据。
- **分布式任务调度**: 提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。同时提供分布式的任务执行模型，如网格任务。网格任务支持海量子任务均匀分配到所有 Worker（schedulerx-client）上执行。
- **阿里云短信服务**: 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。

3. Spring Cloud Alibaba主要组件如下:

组件

Sentinel: 把流量作为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。

Nacos: 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。

RocketMQ: 一款开源的分布式消息系统，基于高可用分布式集群技术，提供低延时的、高可靠的消息发布与订阅服务。

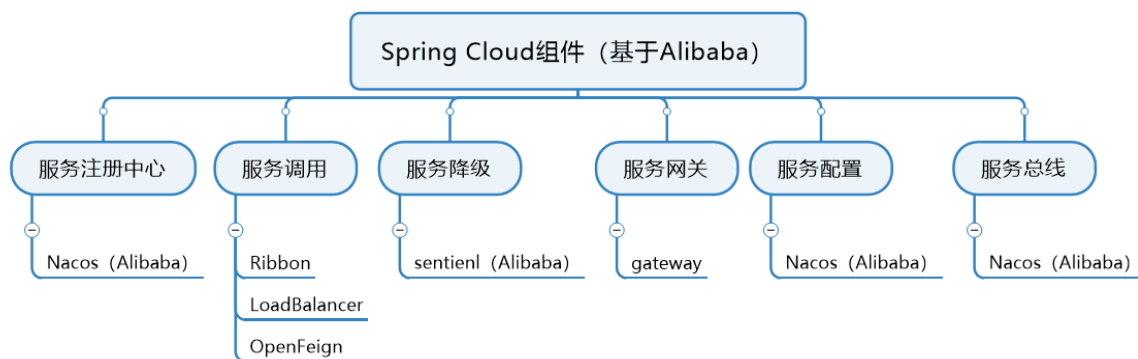
Dubbo: Apache Dubbo™ 是一款高性能 Java RPC 框架。

Seata: 阿里巴巴开源产品，一个易于使用的高性能微服务分布式事务解决方案。

Alibaba Cloud OSS: 阿里云对象存储服务（Object Storage Service，简称 OSS），是阿里云提供的海量、安全、低成本、高可靠的云存储服务。您可以在任何应用、任何时间、任何地点存储和访问任意类型的数据。

Alibaba Cloud SchedulerX: 阿里中间件团队开发的一款分布式任务调度产品，提供秒级、精准、高可靠、高可用的定时（基于 Cron 表达式）任务调度服务。

Alibaba Cloud SMS: 覆盖全球的短信服务，友好、高效、智能的互联化通讯能力，帮助企业迅速搭建客户触达通道。



2) Spring Cloud Alibaba Nacos

2.1) Nacos介绍

- Nacos全称Dynamic Naming And Configuration Services，是一个为动态服务注册发现与配置服务而设计的易于使用的平台，相当于注册中心（Eureka）+配置中心（Config + Bus）。作为微服务架构中最重要的一层，Nacos几乎支持所有类型的服务，如Dubbo、RESTFul、或Kubernetes服务。
- Nacos提供四个主要的功能：
 - Service Discovery and Service Health Check: Nacos提供注册服务与通过DNS或HTTP访问服务的能力，并且提供实时服务健康检测，以避免向不健康的地址发送请求。
 - Dynamic Configuration Management: Nacos使得配置修改时无需重新部署应用与服务，使得配置修改更高效灵活。

- Dynamic DNS Service: Nacos支持加权路由，使您更容易在数据中心的生环境中实现中端负载均衡、灵活的路由策略、流量控制和简单的DNS解析服务。它帮助您轻松实现基于DNS的服务发现，并防止应用程序耦合到特定于供应商的服务发现API。
- Service and MetaData Management: Nacos提供了一个易于使用的服务仪表板，帮助您管理服务元数据、配置、kubernetes DNS、服务运行状况和度量统计数据。
- Nacos与其他服务注册中心比较

服务注册与发现框架	CAP模型	控制台管理	社区活跃度
Eureka	AP	支持	低（2.x版本闭源）
Zookeeper	CP	不支持	中
Consul	CP	支持	高
Nacos	AP	支持	高

2.2) Nacos的使用

- 下载Nacos最新版本，<https://github.com/alibaba/nacos/releases>。后续启动时需注意，有集群模式和单例模式，集群模式需要配置数据库，conf文件夹下有数据库的初始化sql文件，以及需要修改配置中的数据库连接细节；或者使用单例模式，在控制台中表明单例模式 `startup.cmd -m standalone`，使用内置的数据库，无需配置数据库。



2.2.1) Nacos作为服务注册中心

- 父pom引入Spring Cloud Alibaba，此后子Pom不用指明版本

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.alibaba.cloud</groupId>
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>
      <version>${alibaba.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- 新建基于Nacos的服务提供者（即生产者）Module，引入依赖

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


- 修改生产者配置文件，绑定注册中心地址以及暴露健康检测的接口。主启动类同样使用@EnableDiscoveryClient注解。

```
spring:
  application:
    name: cloud-nacos-payment-service
  cloud:
    nacos:
      discovery:
        server-addr: http://localhost:8848
  management:
    endpoints:
      web:
        exposure:
          include: "*"

```

- 随便写个生产者业务类进行测试

```
@RestController
@RequestMapping("/payment/nacos")
public class PaymentController {
    @Value("${server.port}")
    private String port;

    @GetMapping("/{id}")
    public String getPayment(@PathVariable("id") String id){
        return "this comes from nacos payment service, port : " + port + ", id : " + id;
    }
}

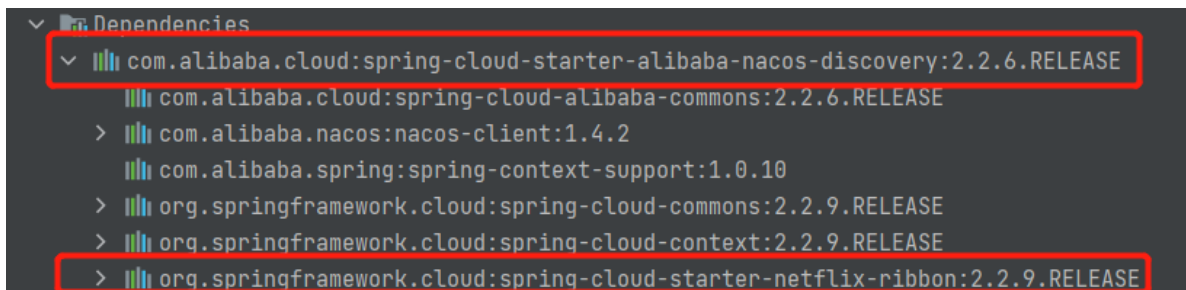
```

- 进行请求访问后，查看nacos控制台



服务名称	分组名称	实例数目	实例数	健康实例数	服务保护策略	操作
cloud-nacos-payment-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除

- 新建基于Nacos的服务消费者Module，引入依赖spring-cloud-starter-alibaba-nacos-discovery，由于该依赖包含Ribbon，因此也支持负载均衡。



```
Dependencies
├── com.alibaba.cloud:spring-cloud-starter-alibaba-nacos-discovery:2.2.6.RELEASE
│   ├── com.alibaba.nacos:nacos-client:1.4.2
│   └── org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.9.RELEASE
├── com.alibaba.spring:spring-context-support:1.0.10
├── org.springframework.cloud:spring-cloud-commons:2.2.9.RELEASE
├── org.springframework.cloud:spring-cloud-context:2.2.9.RELEASE
└── org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.9.RELEASE

```

- 新建消费者配置文件，并直接在配置文件里面设定使用的微服务地址

```

spring:
  application:
    name: cloud-nacos-consumer-service
  cloud:
    nacos:
      discovery:
        server-addr: http://localhost:8848
  service-url:
    nacos-payment-service: http://cloud-nacos-payment-service

```

- 随便写个消费者业务类，进行消费

```

@RestController
@RequestMapping("/order/nacos")
public class OrderController {
    @Resource
    private RestTemplate restTemplate;

    @Value("${service-url.nacos-payment-service}")
    private String payment_service_url_prefix;

    @GetMapping("/{id}")
    public String getPayment(@PathVariable("id") String id){
        return restTemplate.getForObject(payment_service_url_prefix +
            "/payment/nacos/" + id, String.class);
    }
}

```

- 进行消费端的访问请求，`http://localhost/order/nacos/1` 能看到负载均衡正常起作用
- Nacos支持CP和AP的转换，可根据需要进行切换

2.2.2) Nacos作为服务配置中心

- 新建Module，引入依赖

```

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

```

- 新建配置文件，基础配置使用bootstrap.yml避免被覆盖

```

# bootstrap.yml
spring:
  application:
    name: cloud-nacos-config-service
  cloud:
    nacos:
      discovery:
        server-addr: http://localhost:8848
      config:
        server-addr: http://localhost:8848 # 配置中心地址
        file-extension: yaml # 配置文件的类型，可选yaml或者properties

```

```
# application.yml
spring:
  profiles:
    active: dev
```

- 新建配置刷新测试业务类，使用@RefreshScope实现动态刷新

```
@RestController
@RequestMapping("/config")
@RefreshScope
public class ConfigController {
    @Value("${config.info}")
    private String info;

    @GetMapping("/nacos/config")
    public String getConfigInfo(){
        return info;
    }
}
```

- 在Nacos控制台新增配置文件，参照原则如下，并按照这个规则新增配置

说明：之所以需要配置 `spring.application.name`，是因为它是构成 Nacos 配置管理 `dataId` 字段的一部分。

在 Nacos Spring Cloud 中，`dataId` 的完整格式如下：

```
${prefix}-${spring.profiles.active}.${file-extension}
```

- `prefix` 默认为 `spring.application.name` 的值，也可以通过配置项 `spring.cloud.nacos.config.prefix` 来配置。
- `spring.profiles.active` 即为当前环境对应的 profile，详情可以参考 [Spring Boot文档](#)。注意：当 `spring.profiles.active` 为空时，对应的连接符 - 也将不存在，dataId 的拼接格式变成 `${prefix}.${file-extension}`
- `file-extension` 为配置内容的数据格式，可以通过配置项 `spring.cloud.nacos.config.file-extension` 来配置。目前只支持 `properties` 和 `yaml` 类型。

新建配置

* Data ID:

* Group:

[更多高级选项](#)

描述:

配置格式: ☐ TEXT ☐ JSON ☐ XML ☒ YAML ☐ HTML ☐ Properties

* 配置内容:  :

```
1 config:
2   info: "from nacos config center, cloud-nacos-config-service-dev.yaml, version=1"
```

- 进行获取配置信息访问请求，`http://localhost:3377/config/nacos/config`，可以发现直接在Nacos控制台上进行配置修改时会自动刷新配置
- Nacos支持使用Namespace（部署环境）+ Group（微服务分组）+ DataId三种方式来使用不同环境下的配置文件。
 - DataId方式：即使用默认Namespace，默认Group，通过`${prefix}-${spring.profiles.active}.${file-extension}`来匹配。
 - Group方式：即使用Group + DataId来匹配，在微服务的配置文件中`spring.cloud.nacos.config.group`来指明Group。
 - Namespace方式：即新建dev、test等命名空间，在里面使用Group+DataId进行匹配，根据不同的部署环境来，配置文件中`spring.cloud.nacos.config.namespace`来指明Namespace。

2.3) Nacos集群的部署

- Nacos支持三种部署模式：
 - 单机模式：用于测试与单机试用
 - 集群模式：用于生产环境，确保高可用
 - 多集群模式：用于多数据中心场景
- 部署参照手册：<https://nacos.io/zh-cn/docs/cluster-mode-quick-start.html>
- 集群部署架构图推荐模式：

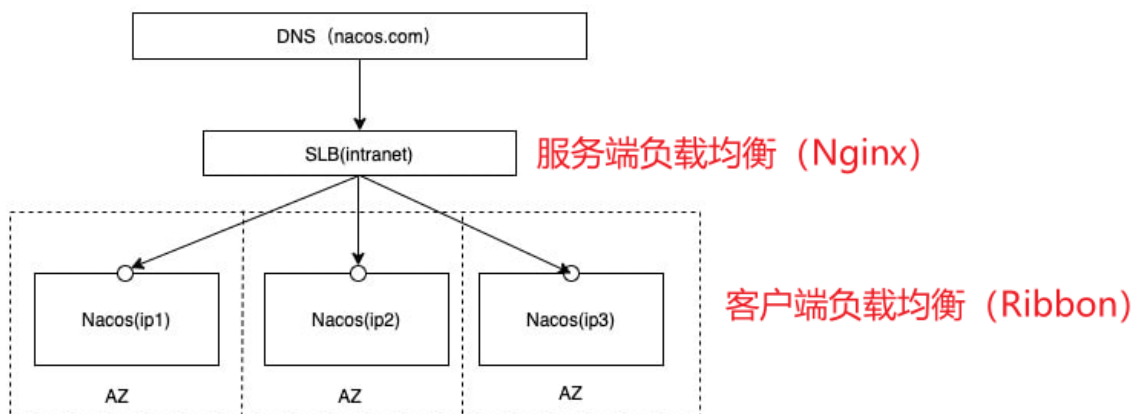
集群部署架构图

因此开源的时候推荐用户把所有服务列表放到一个vip下面，然后挂到一个域名下面

`http://ip1:port/openAPI` 直连ip模式，机器挂则需要修改ip才可以使用。

`http://SLB:port/openAPI` 挂载SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，直连SLB即可，下面挂server真实ip，可读性不好。

`http://nacos.com:port/openAPI` 域名 + SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，可读性好，而且换ip方便，推荐模式



- 默认Nacos使用嵌入式数据库（derby）实现数据的存储，为了解决集群化部署数据存储的一致性问題，Nacos采用集中化存储的方式，目前支持Mysql。

2.3.1) Linux下Nginx + Nacos集群部署

0. 环境准备：64位Linux系统 + JDK1.8以上 + Maven 3.2.x以上 + 3个及以上Nacos节点 + MySQL数据库
1. 配置数据库：Mysql数据库执行 `/nacos/conf/nacos-mysql.sql` 文件
2. 修改nacos配置文件 `/nacos/conf/application.properties`：

```
spring.datasource.platform=mysql
db.num=1
db.url.0=jdbc:mysql://127.0.0.1:3306/nacos_config?characterEncoding=utf-8
db.user=root
db.password=root
```

3. 编写nacos集群配置文件 `/nacos/conf/cluster.conf`：

```
192.168.100.100:8848
192.168.100.200:8848
192.168.100.201:8848
```

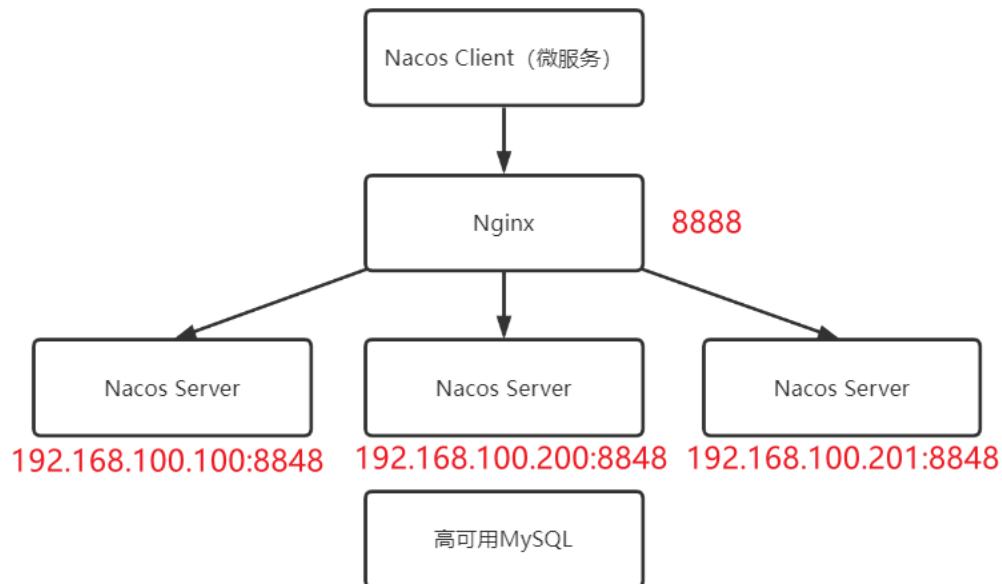
4. 修改nginx配置文件 `nginx.conf`：

```
upstream cluster{
    server 192.168.100.100:8848;
    server 192.168.100.200:8848;
    server 192.168.100.201:8848;
}

server{
    listen 8888;
    server_name localhost;

    location / {
        proxy_pass http://cluster
    }
}
```

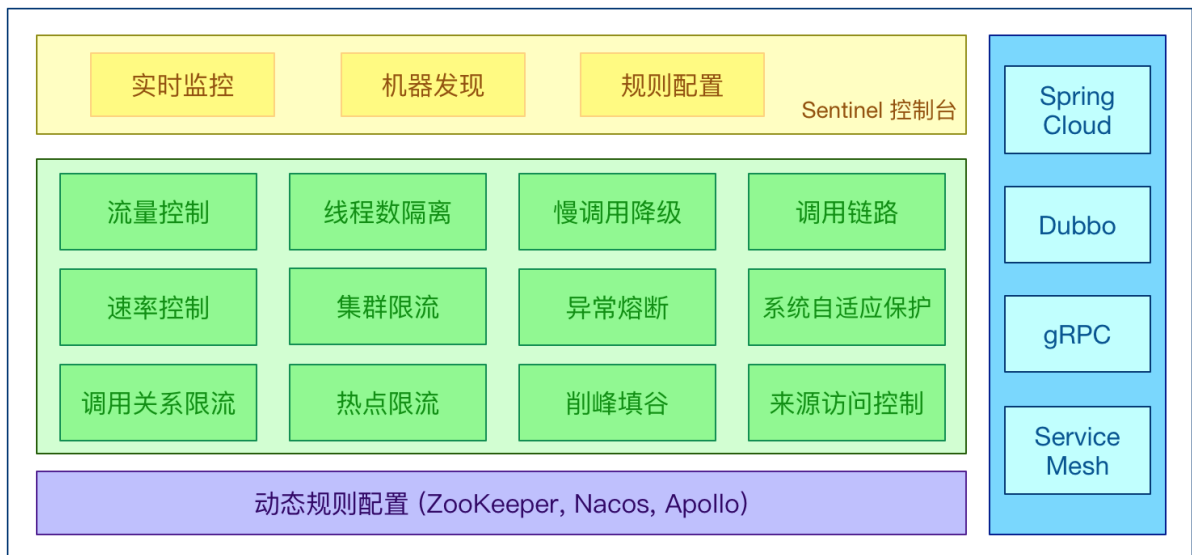
5. 外部通过 `192.168.100.100:8888/nacos` 访问Nacos控制台，微服务通过 `spring.cloud.nacos.config/discovery.server-addr=192.168.100.100:8888` 绑定Nacos集群。
6. 部署结构图：



3) Spring Cloud Alibaba Sentinel

3.1) Sentinel介绍

- Sentinel是一个高效强力的流量控制组件，以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度保护服务的稳定性。强大的流量控制组件，支持微服务的可靠性、弹性和监控。包含以下特性：



3.2) Sentinel的使用

- 下载Sentinel控制台最新版本，<https://github.com/alibaba/sentinel/releases>。确保jdk1.8环境及8080端口未被占用（sentinel-dashboard.jar的默认端口），执行jar包，前端打开<http://localhost:8080> 输入sentinel账号密码。
- 同时启动nacos，新建Module，引入依赖

```
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

- 新建配置文件，配置Nacos注册地址和Sentinel交互地址与交互端口

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: http://localhost:8848
    sentinel:
      transport:
        dashboard: http://localhost:8080
        port: 8719 # 默认8719，被占用时进行+1轮询，直到未占用
  management:
    endpoints:
      web:
        exposure:
          include: "*"

```

- 由于sentinel是懒加载，需要先访问资源才会对资源进行监控。启动并进行访问请求，可以在Sentinel控制台看到如下信息



3.2.1) 流量控制

- 流量控制规则

Field	说明	默认值
resource	资源名，资源名是限流规则的作用对象	
count	限流阈值	
grade	限流阈值类型，QPS 或线程数模式	QPS 模式
limitApp	流控针对的调用来源	default，代表不区分调用来源
strategy	调用关系限流策略：直接、链路、关联	根据资源本身（直接）
controlBehavior	流控效果（直接拒绝 / 排队等待 / 慢启动模式），不支持按调用关系限流	直接拒绝

- 流控效果：
 - 直接拒绝：达到阈值后续请求直接拒绝，抛出FlowException
 - 慢启动模式（冷启动）：当系统长期处于低水位时，突然的流量增加可能导致系统崩溃。慢启动模式将阈值逐渐增加，在一定时间内逐渐增加达到阈值上限。
 - 排队等待（匀速通过）：当处理间隔性的突发流量时，希望系统能在空闲时间逐渐处理这些请求而不是直接拒绝。匀速通过模式能严格的限制请求通过的时间间隔。
- 流控策略：
 - 直接：根据资源名本身进行限制
 - 关联：根据关联的资源，当该关联资源的请求达到阈值时，限制本资源的请求
 - 链路：存在多个入口调用同一个微服务的情况，可以指定监控某个入口。可以在代码中使用@SentinelResource将某个方法标注为Sentinel的资源。
- 注意事项：
 1. Sentinel 从 1.6.3版本开始，Sentinel Web Filter 默认收敛所有URL的入口Context，因此链路限流不生效。
 2. 1.7.0版本开始，官方在CommonFilter中引入了一个WEB_CONTEXT_UNIFY参数，用于控制是否收敛context。默认为true(默认收敛所有)，配置为false则可根据不同URL进行链路的限流操作。
 3. Spring Cloud Alibaba 在2.1.1.RELEASE版本后，可以根据配置spring.cloud.sentinel.filter.enabled: false来关闭自动收敛。
- 增加配置以启用链路限流

```
<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>sentinel-web-servlet</artifactId>
</dependency>
```

```
@Configuration
public class FilterContextConfig {
    @Bean
    public FilterRegistrationBean sentinelFilterRegistration() {
        FilterRegistrationBean registrationBean = new FilterRegistrationBean();
        registrationBean.setFilter(new CommonFilter());
        registrationBean.addUrlPatterns("/");
        // 入口资源关闭聚合
        registrationBean.addInitParameter(CommonFilter.WEB_CONTEXT_UNIFY,
            "false");
        registrationBean.setName("sentinelFilter");
        registrationBean.setOrder(1);
        return registrationBean;
    }
}
```



```
}  
}
```

```
spring:  
  cloud:  
    sentinel:  
      filter:  
        enabled: false # 不使用sentinel自带的Filter
```

3.2.2) 熔断降级

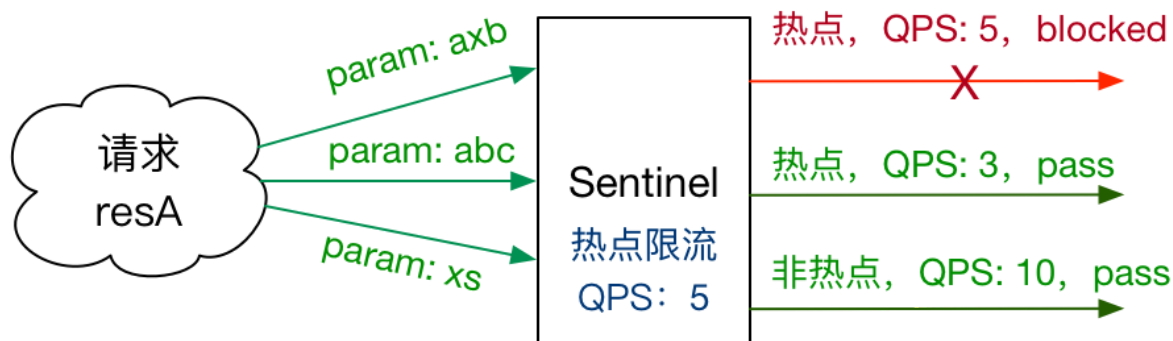
- 熔断降级规则：

Field	说明	默认值
resource	资源名，即规则的作用对象	
grade	熔断策略，支持慢调用比例/异常比例/异常数策略	慢调用比例
count	慢调用比例模式下为慢调用临界 RT（超出该值计为慢调用）；异常比例/异常数模式下为对应的阈值	
timeWindow	熔断时长，单位为 s	
minRequestAmount	熔断触发的最小请求数，请求数小于该值时即使异常比率超出阈值也不会熔断（1.7.0 引入）	5
statIntervalMs	统计时长（单位为 ms），如 60*1000 代表分钟级（1.8.0 引入）	1000 ms
slowRatioThreshold	慢调用比例阈值，仅慢调用比例模式有效（1.8.0 引入）	

- 熔断策略：
 - 慢调用比例：设置允许的慢调用RT（即最大的响应时间），请求的响应时间大于该值时统计为慢调用。单位统计时长内请求数大于设置的最小请求数，且慢调用比例大于阈值时，在接下来的熔断时长内请求会被自动熔断。熔断时间后进入探测阶段（Half-Open阶段），根据下一个请求响应时间是否长于RT决定是否关闭熔断。
 - 异常比例：单位统计时长内请求数大于设置的最小请求数，且异常比例大于阈值时，在接下来的熔断时长内请求会被自动熔断。熔断时间后进入探测阶段（Half-Open阶段），根据下一个请求是否异常决定是否关闭熔断。
 - 异常数：单位统计时长内请求数大于设置的最小请求数，且异常数大于阈值时，在接下来的熔断时长内请求会被自动熔断。熔断时间后进入探测阶段（Half-Open阶段），根据下一个请求是否异常决定是否关闭熔断。

3.2.3) 热点参数限流

- 热点即经常访问的数据。热点参数限流会统计传入参数中的热点参数，并根据配置的限流阈值与模式，对包含热点参数的资源调用进行限流。热点参数限流可以看做是一种特殊的流量控制，仅对包含热点参数的资源调用生效。



- 热点参数规则：

Field	说明	默认值
resource	资源名，必填	
count	限流阈值，必填	
grade	限流模式	QPS 模式
durationInSec	统计窗口时间长度（单位为秒），1.6.0 版本开始支持	1s
controlBehavior	流控效果（支持快速失败和匀速排队模式），1.6.0 版本开始支持	快速失败
maxQueueingTimeMs	最大排队等待时长（仅在匀速排队模式生效），1.6.0 版本开始支持	0ms
paramIdx	热点参数的索引，必填，对应 <code>Sphu.entry(xxx, args)</code> 中的参数索引位置	
paramFlowItemList	参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 <code>count</code> 阈值的限制。 仅支持基本类型和字符串类型	
clusterMode	是否是集群参数流控规则	<code>false</code>
clusterConfig	集群流控相关配置	

3.2.4) Sentinel的降级处理方法

- 其中blockHandler属性处理的是因为Sentinel控制台限流造成的异常，fallback属性是处理因业务本身造成的异常，exceptionsToIgnore是忽视某种异常。若fallback或者blockHandler方法不在本类中需要注明属性fallbackHandlerClass或者blockHandlerClass；处理方法注意参数个数与格式需要对应上。

```

@GetMapping("/testHotkey")
@SentinelResource(value = "testHotkey",
                  blockHandler = "hotkeyBlockHandler", fallback =
"hotkeyFallback")
public String testHotkey(@RequestParam(value = "user", required = false) String
user,
                        @RequestParam(value = "id", required = false) String
id){
    if (user.equals("abc")){

```

```

        throw new IllegalArgumentException();
    }
    return "testHotkey";
}

public String hotkeyFallback(String user, String id, Throwable e){
    return "hotkey fallback";
}

public String hotkeyBlockHandler(String user, String id, BlockException
exception){
    return "hotkey block handler";
}
}

```

3.2.5) 系统自适应保护

- Sentinel 系统自适应保护从整体维度对应用入口流量进行控制，结合应用的 Load、总体平均 RT、入口 QPS 和线程数等几个维度的监控指标，让系统的入口流量和系统的负载达到一个平衡，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：

Field	说明	默认值
highestSystemLoad	load1 触发值，用于触发自适应控制阶段	-1 (不生效)
avgRt	所有入口流量的平均响应时间	-1 (不生效)
maxThread	入口流量的最大并发数	-1 (不生效)
qps	所有入口资源的 QPS	-1 (不生效)
highestCpuUsage	当前系统的 CPU 使用率 (0.0-1.0)	-1 (不生效)

3.2.6) Sentinel规则持久化

- Sentinel规则默认是临时化的，当微服务重启后规则将清空，因此为了规则的持久化，考虑将规则存放在Nacos中，每次微服务启动时都将从Nacos中加载Sentinel规则。若不清楚如何书写Sentinel规则的json，可以在Sentinel上添加一条规则，之后根据微服务使用的actuator接口，访问 ip:port/actuator/sentinel获取。
- 引入依赖文件

```

<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>

```

- 在配置文件中指明Sentinel规则所存放的数据库地址

```

spring:
  cloud:
    sentinel:
      datasource:
        ds1:
          nacos:
            server-addr: http://localhost:8848
            dataId: ${spring.application.name}
            groupId: DEFAULT_GROUP
            data-type: json
            rule-type: flow

```

3.3) Sentinel与其他服务熔断框架对比

框架	Sentinel	Hystrix	resilience4j
隔离策略	信号量隔离（并发线程数限流）	线程池隔离/信号量隔离	信号量隔离
熔断降级策略	基于响应时间、异常比率、异常数	基于异常比率	基于异常比率、响应时间
实时统计实现	滑动窗口（LeapArray）	滑动窗口（基于 RxJava）	Ring Bit Buffer
动态规则配置	支持多种数据源	支持多种数据源	有限支持
扩展性	多个扩展点	插件的形式	接口的形式
基于注解的支持	支持	支持	支持
限流	基于 QPS，支持基于调用关系的限流	有限的支持	Rate Limiter
流量整形	支持预热模式、匀速递器模式、预热排队模式(流量规则处可配置)	不支持	简单的 Rate Limiter 模式
系统自适应保护	支持	不支持	不支持
控制台	提供开箱即用的控制台，可配置规则、查看秒级监控、机器发现等	简单的监控查看	不提供控制台，可对接其它监控系统

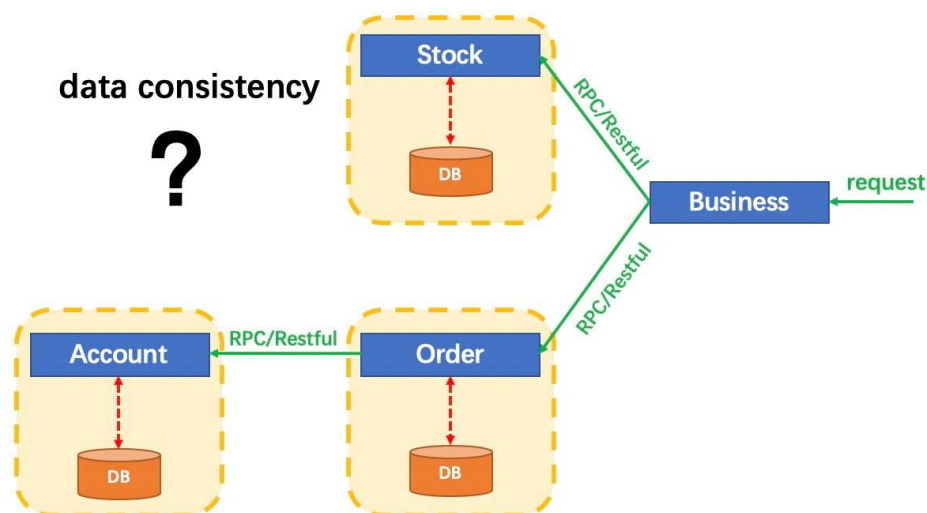
4) Spring Cloud Seata

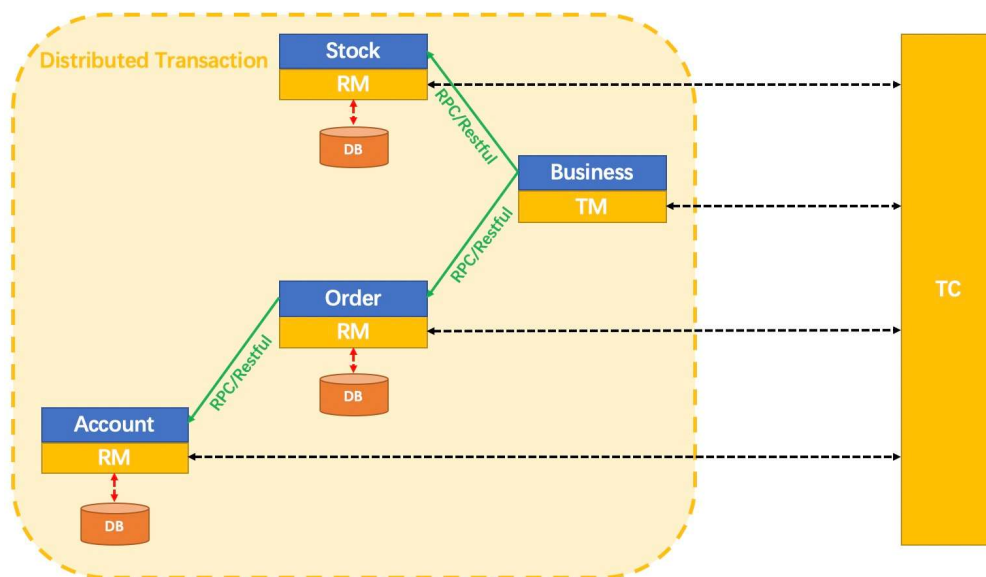
4.0) 分布式事务模式（了解下两段式三段式）

分布式事务模式	介绍	技术栈
AT模式	无侵入的分布式事务解决方案，适用于不希望对业务进行改造的场景，几乎0学习成本（sql都由框架托管统一执行，会存在脏写问题）	seata、shardingsphere
TCC模式	高性能分布式事务解决方案，适用于核心系统等对性能有很高要求的场景（第一阶段会产生行锁，事务执行太久会锁行很久）	seata、service-comb
Saga模式	长事务解决方案，适用于业务流程长且需要保证事务最终一致性的业务系统（第一阶段就操作DB，会存在脏读问题）	seata、shardingsphere、service-comb
XA模式	分布式强一致性的解决方案，但性能低而使用较少。	seata、shardingsphere

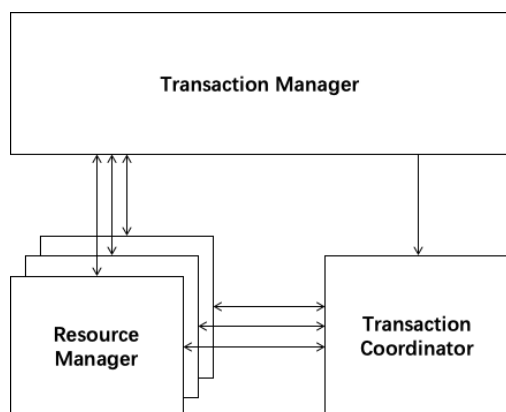
4.1) Seata介绍

- Seata全称Simple Extensible Autonomous Transaction Architecture。是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 AT、TCC、SAGA 和 XA 事务模式。
- 随着微服务与分布式系统的流行，微服务之间互相调用，但所使用的数据库不一致时，如何保持整个系统的数据一致性成为问题。而使用Seata则可以解决这个问题。

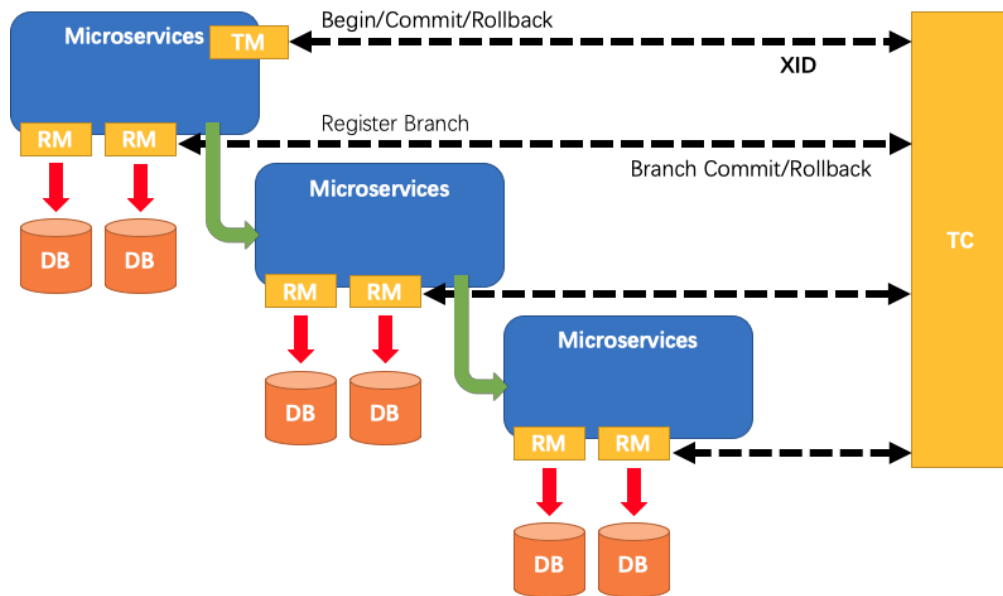




- Distributed Transaction 分布式事务是由多个分支事务组成的全局事务，分支事务通常是数据库本身的事务。
- Seata框架中三个主要成员：
 - Transaction Coordinator (TC)：维护全局事务和分支事务状态，驱动全局提交或回滚。
 - Transaction Manager (TM)：定义全局事务范围:开始全局事务、提交或回滚全局事务。
 - Resource Manager (RM)：管理分支事务所处理的资源，与TC对话，以便注册分支事务和报告分支事务的状态，并驱动分支事务提交或回滚。



- Seata管理分布式事务的典型生命周期：
 1. TM向TC请求开启一个新的全局事务，TC生成一个全新的全局事务并用XID唯一标识。
 2. XID通过微服务的调用链进行传播。
 3. RM向TC注册本地事务作为对应的XID全局事务的分支事务。
 4. TM根据XID对应的全局事务向TC请求提交或回滚。
 5. TC驱动XID对应的全局事务下的所有分支事务进行提交或回滚。



4.2) Seata使用

4.2.1) Seata服务端

- 下载Seata最新版本, <https://github.com/seata/seata/releases>。结合Nacos使用, 可先在Nacos中建立Seata专属的命名空间用于隔离Seata的服务与配置。

NACOS 2.0.3

配置管理

服务管理

权限控制

命名空间

集群管理

命名空间

新建命名空间

刷新

命名空间名称	命名空间ID	配置数	操作
public(保留空间)		1	详情 删除 编辑
seata	1df1f25b-3da0-409b-8c3c-6f336f8ea6cb	0	详情 删除 编辑

- 修改 /conf 下的配置文件, 首先修改 registry.conf, 配置Seata注册与配置中心Nacos。

```
registry {
  # file 、nacos 、eureka、redis、zk、consul、etcd3、sofa
  type = "nacos"

  nacos {
    application = "seata-server"
    serverAddr = "127.0.0.1:8848"
    group = "SEATA_GROUP"
    namespace = "1df1f25b-3da0-409b-8c3c-6f336f8ea6cb"
    cluster = "default"
    username = "nacos"
    password = "nacos"
  }
}

config {
  # file、nacos 、apollo、zk、consul、etcd3
  type = "nacos"

  nacos {
    serverAddr = "127.0.0.1:8848"
    namespace = "1df1f25b-3da0-409b-8c3c-6f336f8ea6cb"
    group = "SEATA_GROUP"
  }
}
```

```

    username = "nacos"
    password = "nacos"
    dataId = "seataServer.properties"
  }
}

```

- 若使用file模式则修改 `file.conf`，更改事务日志的存放数据库配置（如果上面选择了Nacos则不需要这一步）。

```

## transaction log store, only used in seata-server
store {
  ## store mode: file、db、redis
  mode = "db"
  ## rsa decryption public key
  publicKey = ""

  db {
    ## the implement of javax.sql.DataSource
    datasource = "druid"
    ## mysql/oracle/postgresql/h2/oceanbase etc.
    dbType = "mysql"
    driverClassName = "com.mysql.cj.jdbc.Driver"
    url = "jdbc:mysql://127.0.0.1:3306/seata?rewriteBatchedStatements=true"
    user = "root"
    password = "root"
    minConn = 5
    maxConn = 100
    globalTable = "global_table"
    branchTable = "branch_table"
    lockTable = "lock_table"
    queryLimit = 100
    maxWait = 5000
  }
}

```

- 修改Seata source里的 `/script/config-server/config.txt` 内部细节，并上传至Nacos。

```

service.vgroupMapping.order-service-group=default
service.vgroupMapping.storage-service-group=default
service.vgroupMapping.account-service-group=default
service.vgroupMapping.business-service-group=default
store.mode=db
store.db.driverClassName=com.mysql.cj.jdbc.Driver
store.db.url=jdbc:mysql://127.0.0.1:3306/seata?
useUnicode=true&rewriteBatchedStatements=true
store.db.user=root
store.db.password=root

```

- windows使用python命令执行上传config.txt到Nacos中

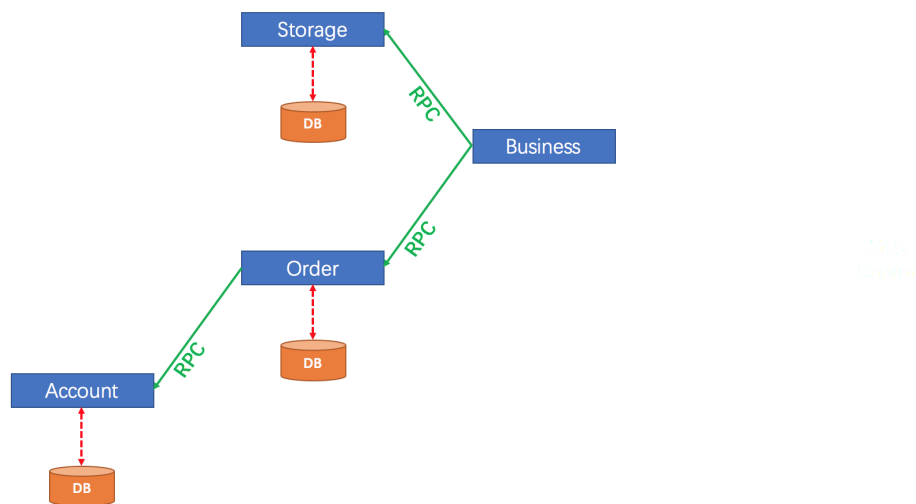
```
$ python3 nacos-config.py localhost:8848 1df1f25b-3da0-409b-8c3c-6f336f8ea6cb
```


public seata				
配置管理 seata 1df1f25b-3da0-409b-8c3c-6f336f8ea6cb 查询结果: 共查询到 67 条满足要求的配置。				
Data ID:	添加通配符"*"进行模糊查询	Group:	添加通配符"*"进行模糊查询	查询 高级查询 导入配置
<input type="checkbox"/>	Data Id ↕↑	Group ↕↑	归属应用: ↕↑	操作
<input type="checkbox"/>	transport.type	SEATA_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	transport.server	SEATA_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	transport.heartbeat	SEATA_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	transport.enableClientBatchSendRequest	SEATA_GROUP		详情 示例代码 编辑 删除 更多
<input type="checkbox"/>	transport.threadFactory.bossThreadPrefix	SEATA_GROUP		详情 示例代码 编辑 删除 更多

- seata数据库执行seata source里的 `/script/server/db/mysql.sql` , 生成服务端的数据表 `branch_table`、`global_table`、`lock_table`。

4.2.2) 分布式事务Demo

- 架构图



- 业务准备数据库，准备三个数据库 `storedb`、`orderdb`、`accountdb`。为了实现Seata AT模式，每个数据库都必须存在 `undo_log` 表，建表文件参考Seata Source里的 `/script/client/at/mysql.sql`。除此之外，创建各自的业务表。
- 新建Module，引入依赖。为了保证所使用的版本与Seata服务端版本一致，需要排除并手动引入指定版本

```

<dependency>
  <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
  <exclusions>
    <exclusion>
      <groupId>io.seata</groupId>
      <artifactId>seata-spring-boot-starter</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-spring-boot-starter</artifactId>
  <version>1.4.2</version>
</dependency>

```

- 每个微服务均要有一个registry.conf，配置基本和服务端的一致，新建配置文件

```
spring:
  application:
    name: cloud-alibaba-seata-storage-service
  cloud:
    nacos:
      discovery:
        server-addr: http://localhost:8848
        namespace: 1df1f25b-3da0-409b-8c3c-6f336f8ea6cb
seata:
  application-id: ${spring.application.name}
  enabled: true
  enable-auto-data-source-proxy: false
  service:
    vgroup-mapping:
      storage_service_group: default
tx-service-group: storage_service_group #要与配置文件中的vgroupMapping一致
registry: #registry根据seata服务端的registry配置
  type: nacos
  nacos:
    server-addr: localhost:8848
    username: nacos
    password: nacos
    namespace: 1df1f25b-3da0-409b-8c3c-6f336f8ea6cb
    group: SEATA_GROUP
config:
  type: nacos
  nacos:
    server-addr: localhost:8848
    group: SEATA_GROUP
    username: nacos
    password: nacos
    namespace: 1df1f25b-3da0-409b-8c3c-6f336f8ea6cb
```

- 添加主启动类，排除数据源自动配置

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@EnableDiscoveryClient
@EnableFeignClients
@MapperScan("com.hjy.springcloud.dao")
public class StorageMain8881 {
    public static void main(String[] args) {
        SpringApplication.run(StorageMain8881.class, args);
    }
}
```

- 新增数据源代理配置类

```
@Configuration
public class DataSourceProxyConfig {
    @Bean
    @ConfigurationProperties(prefix = "spring.datasource")
    public DataSource dataSource() {
        return new DruidDataSource();
    }
}
```

```

@Bean
public DataSourceProxy dataSourceProxy(DataSource dataSource) {
    return new DataSourceProxy(dataSource);
}

@Bean
public SqlSessionFactory sqlSessionFactoryBean(DataSourceProxy
dataSourceProxy) throws Exception {
    SqlSessionFactoryBean sqlSessionFactoryBean = new
SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dataSourceProxy);
    // 由于使用了自定义的数据源代理，因此部分配置失效，需要手动配置
    sqlSessionFactoryBean.setMapperLocations(
        new PathMatchingResourcePatternResolver().
        getResources("classpath:mapper/*.xml"));

    sqlSessionFactoryBean.setTypeAliasesPackage("com.hjy.springcloud.entity");
    return sqlSessionFactoryBean.getObject();
}
}

```

- 事务的发起者使用@GlobalTransactional注解

```

@Override
@GlobalTransactional
public void purchase(String userId, String commodityCode, int orderCount) {
    storageFeignService.deduct(commodityCode, orderCount);
    orderFeignService.create(userId, commodityCode, orderCount);
}

```

- 出现异常时回滚

```

i.seata.tm.api.DefaultGlobalTransaction : Suspending current transaction, xid = 192.168.29.1:8091:4575850110958276609
i.seata.tm.api.DefaultGlobalTransaction : [192.168.29.1:8091:4575850110958276609] rollback status: Rollbacked

io.seata.rm.AbstractRMHandler           : Branch Rollbacking: 192.168.29.1:8091:4575850110958276609
i.s.r.d.undo.AbstractUndoLogManager     : xid 192.168.29.1:8091:4575850110958276609 branch 457585011
io.seata.rm.AbstractRMHandler           : Branch Rollbacked result: PhaseTwo_Rollbacked

```

- 正常提交

```

i.seata.tm.api.DefaultGlobalTransaction : Begin new global transaction [192.168.29.1:8091:4575850110958276616]
i.seata.tm.api.DefaultGlobalTransaction : Suspending current transaction, xid = 192.168.29.1:8091:4575850110958276616
i.seata.tm.api.DefaultGlobalTransaction : [192.168.29.1:8091:4575850110958276616] commit status: Committed

io.seata.rm.AbstractRMHandler           : Branch committing: 192.168.29.1:8091:4575850110958276616
io.seata.rm.AbstractRMHandler           : Branch commit result: PhaseTwo_Committed

```

Bug或使用方式

环境类

1) Lombok注解不起作用

- Error: 无法将类中的构造器应用到给定类型,
- 1.16.18无效, 改成1.18.10解决问题

2) MySQL8.0出现连接异常Public Key Retrieval is not allowed

- Error: 不允许检索公钥
- mysql8以上版本默认使用 sha256_password 认证, 密码在传输过程中必须加密保护, 如果无法使用 TLS, 就需要使用 RSA 公钥加密。可以在连接字符串中通过 ServerRSAPublicKeyFile 指定服务器的 RSA 公钥; 或者设置AllowPublicKeyRetrieval=True参数以允许客户端从服务器获取公钥, 但可能会导致恶意的代理通过中间人攻击(MITM)获取到明文密码, 所以默认是关闭的, 必须显式开启。

3) 初步发现, 引入包并不会引入包的依赖

- 即通用jar包中引入了Lombok, devtools等依赖。其他jar包引入了该通用jar包, 并不能直接使用Lombok、devtools, 还是要自己继承。

4) Eureka Client ServiceUrl配置出错

- Error: Failed to bind properties under "eureka.client.service-url" to java.util.Map<java.lang.String, java.lang.String>
- 即yml缩进和空格不对, 冒号: 后面要加空格, 与父标题要缩进两格

5) Zookeeper版本对应不上

- Error: KeeperException\$UnimplementedException
- 考虑是Zookeeper版本与依赖版本对应不上, 进行排除并重新引入

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
  <!--排除zk-->
  <exclusions>
    <exclusion>
      <groupId>org.apache.zookeeper</groupId>
      <artifactId>zookeeper</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<!--添加zk 指定版本-->
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>${zookeeper.version}</version>
</dependency>
```

6) Consul出现HttpClientErrorException\$NotFound: 404 Not Found

- Error: HttpClientErrorException\$NotFound: 404 Not Found: [no body]
- 出错原因可能在于配置文件写错了

```
spring:
  application:
    name: cloud-providerconsul-payment
  cloud:
    consul:
      # 此时hostname和port是在consul下的，若放在discovery下，则会出现上述问题
      hostname: localhost
      port: 8500
    discovery:
      service-name: ${spring.application.name}
      heartbeat:
        enabled: true
```

7) Feign日志未显示

- Error: yml配置正常，FeignLoggerConfig使用的Logger.level也没错，但是未显示日志
- 出错原因：可能在于FeignLoggerConfig文件未放在SpringBootApplication所扫描的包或子包下，导致该配置未被扫描而生效，将配置类放在可被扫描的路径下后日志可正常显示。

8) 通过Feign或者RestTemplate调用微服务时Load balancer does not have available server for client: CLOUD-PAYMENT-SERVICE

- Error: Load balancer does not have available server for client: CLOUD-PAYMENT-SERVICE
- 出错原因：可能是因为服务调用者先于服务提供者启动，因此服务调用者在向注册中心获取服务节点时检索不到可用服务，导致出错。

9) 使用Spring Cloud Config启动失败，获取不到git上的配置

- 出错原因：可能是因为客户端微服务使用的配置文件名称和git上的相同，例如都为application，那么优先加载本地的，就取不到git上的配置。最好就是客户端微服务使用bootstrap作为配置文件名称，确保可以加载git上的配置

10) 使用RestTemplate请求微服务时无法识别微服务名作为url一部分的情况

- Error: java.net.UnknownHostException: CLOUD-PAYMENT-SERVICE
- 出错原因：RestTemplate类定义时可能未使用@LoadBalanced注解

11) Mybatis扫描不到Dao类，无法注入

- Error: 已经使用@Mapper注解，但是任然扫描不到
- 出错原因：可能是由于使用Seata的过程中使用自定义的数据源代理，导致的。加入@MapperScan。

12) 开启分布式全局事务出错, Tm事务错误, application_id太长

- Error: io.seata.core.exception.TmTransactionException: TransactionException[begin global request failed. xid=null, msg=Data truncation: Data too long for column 'application_id' at row 1]
- 出错原因: Seata数据库的global_table表application_id默认长度为32, 修改为255

13) Invalid bound statement (not found), 使用了自定义数据源代理

- Error: Invalid bound statement (not found)
- 出错原因: 使用了自定义的数据源代理后, 配置文件中的部分配置, 如Mybatis的扫描配置将失效, 需要手动写入sqlSessionFactoryBean.setMapperLocations(new PathMatchingResourcePatternResolver().getResources("classpath:mapper/*.xml"));

代码类

1) mybatis插入语句返回主键

- useGeneratedKeys="true", 使用自增
- keyColumn="id", 表的主键
- keyProperty="id", 对象的字段, 用对象返回主键值

2) @RequestBody和@RequestParam区别

- @RequestBody是放在请求体requestBody的数据
- @RequestParam是放在请求头即url后面拼接的数据中

3) RestTemplate出现422 Unprocessable Entity

- 意思为请求格式正确, 但是请求参数出错误, 仔细确认参数是否缺少或者名称有出入