

微服务

微服务是一种经过良好架构设计的分布式架构方案

单一职责：微服务拆分粒度更小，每一个服务都对应唯一的业务能力，做到单一职责，避免重复业务开发

面向服务：微服务对外暴露业务接口

自治：团队，技术，数据，部署独立

隔离性强：服务调用做好隔离，容错，降级，避免出现级联问题

操作：

服务集群 暴露接口，让**注册中心** 拉取或注册 服务信息，**配置中心** 拉取 配置信息，用户经过**服务网关**，服务网关对请求路由 负载均衡 到**服务集群**

要求：

不同微服务不要重复开发相同业务

微服务数据独立，不要访问其它微服务数据库，有自己独立数据库

微服务可以将自己业务暴露为接口，供其它微服务调用

springcloud

父项目引入依赖

微服务框架

注意springboot要和springcloud版本兼容，上网搜

我们用hoxton.sr10，springboot对应版本是2.3.x

```
<?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>groupId</groupId>
    <artifactId>SpringCloud</artifactId>
    <version>1.0-SNAPSHOT</version>

    <!-- 对子项目的管理-->
    <modules>
        <module>Cloud02</module>
        <module>Cloud01</module>
        <module>eureka-server</module>
    </modules>

    <packaging>pom</packaging>
```

```

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.9.RELEASE</version>
    <relativePath/>
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
    <spring-cloud.version>Hoxton.SR10</spring-cloud.version>
    <mysql.version>5.1.47</mysql.version>
    <mybatis.version>2.1.1</mybatis.version>
</properties>

<dependencyManagement>
    <dependencies>
        <!-- springCloud -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!--mysql-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>
        <!--mybatis-->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>
</project>

```

子项目引入依赖

```

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

```

```
<groupId>com.example</groupId>
<artifactId>Cloud02</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>Cloud02</name>
<description>Cloud02</description>
<!-- 指定父亲-->
<parent>
    <groupId>groupId</groupId>
    <artifactId>SpringCloud</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>

<properties>
    <java.version>1.8</java.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <aliyun-spring-boot.version>1.0.0</aliyun-spring-boot.version>
    <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
</properties>

<dependencies>

    <dependency>
        <!--springboot-->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!--      用来转化成json-->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>fastjson</artifactId>
        <version>1.2.4</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>2.1.3</version>
    </dependency>
    <!-- mybatis逆向工程jar包 -->
    <dependency>
        <groupId>org.mybatis.generator</groupId>
        <artifactId>mybatis-generator-core</artifactId>
        <version>1.3.4</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>aliyun-redis-spring-boot-starter</artifactId>
    </dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
</dependencies>

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

<build>
    <plugins>

        <!-- mybatis逆向工程jar包-->
        <plugin>
            <groupId>org.mybatis.generator</groupId>
            <artifactId>mybatis-generator-maven-plugin</artifactId>
            <version>1.3.2</version>
            <configuration>
```

```

<overwrite>true</overwrite>

<configurationFile>src/main/resources/generator/generatorConfig.xml</configurationFile>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
    </configuration>
</plugin>
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>2.3.7.RELEASE</version>
    <configuration>

<mainClass>com.example.cloud02.cloud02Application</mainClass>
    </configuration>
    <executions>
        <execution>
            <id>repackage</id>
            <goals>
                <goal>repackage</goal>
            </goals>
        </execution>
    </executions>
</plugin>
</plugins>
</build>

</project>

```

远程调用方式分析

如何在java里面发请求

```

//配置类里
@Bean
public RestTemplate restTemplate()
{
    return new RestTemplate();
}

```

在service层

```

//注入restTemplate对象
@Autowire
private RestTemplate restTemplate;
//利用RestTemplate发送http请求
String url="http://localhost:8081/user/"+id;
User user=restTemplate.getForObject(url,user.class);

```

提供者和消费者

服务提供者：提供接口的

服务消费者：调用接口的

一个服务也可以既是提供者也是消费者

Eureka

eureka-server注册中心会有服务的注册服务信息，并且与服务有心跳续约，每30秒一次。

消费者如何获取提供者具体信息？

提供则会启动时向eureka注册自己信息，eureka保存信息，消费者根据服务名称向eureka拉取提供者信息

多个提供者，消费者怎么选择？

消费者利用负载均衡算法选一个

消费者可以感谢提供者健康状态？

提供者每隔30秒向eurekaserver发送心跳请求，报告健康状态，eureka更新记录服务列表信息，心跳不正确会被剔除，消费者可以拉取到最新的信息

搭建eurekaserver

引入依赖

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

    <groupId>org.example</groupId>
    <artifactId>eureka-server</artifactId>
    <version>1.0-SNAPSHOT</version>
    <!--指定父项目-->
    <parent>
        <groupId>groupId</groupId>
        <artifactId>SpringCloud</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <!-- 引入依赖-->
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
            <version>2.2.7.RELEASE</version>
        </dependency>
    </dependencies>
```

```
</project>
```

开启，加注解

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

写配置信息

```
server:  
  port: 10086 #服务端口  
spring:  
  application:  
    name: eurekaserver #服务名称  
eureka:  
  client:  
    service-url: #eureka地址信息  
    defaultZone: http://127.0.0.1:10086/eureka
```

注册

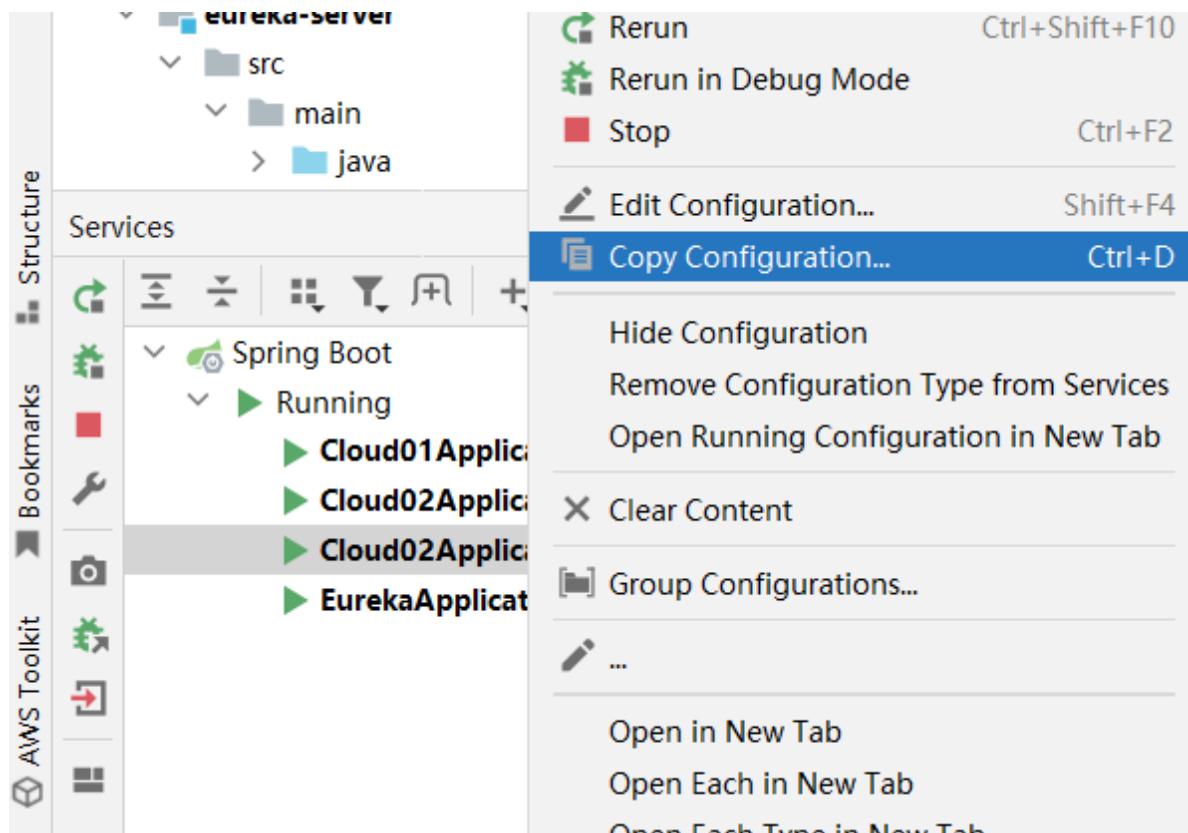
在提供者和消费者中引入依赖

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

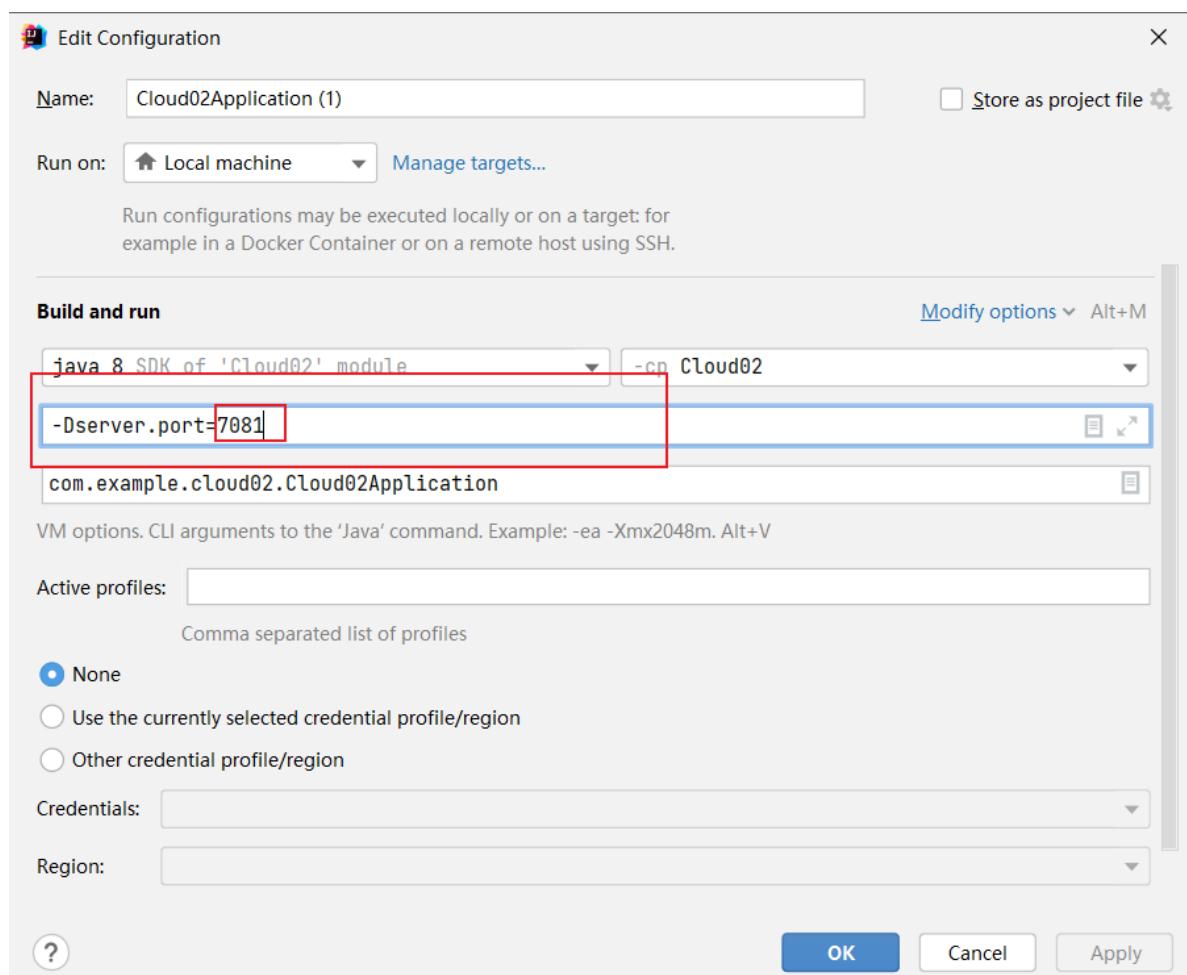
写配置文件

```
spring:  
  application:  
    name: orderservice #该服务名称  
eureka:  
  client:  
    service-url:  
    defaultZone: http://127.0.0.1:10086/eureka #eureka地址信息
```

可以拷贝配置



但是要修改端口号



服务拉取

原先的url路径解耦为用服务名代替ip，端口

```
@RequestMapping("/test")
public TbUser test(){
    return restTemplate.getForObject("http://userServer/user/", TbUser.class);
}
//此处userServer对应的是另一个服务器中
spring:
application:
name: userServer #服务名称
```

在@Bean获取RestTemplate添加负载均衡注解@LoadBalanced

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

负载均衡Ribbon

RestTemplate发起请求: http://服务名称

loadbalancerinterceptor负载均衡拦截器拦截到请求

RibbonLoadBanlancerClient客户端获取url中的服务器id发送到到DynamicServerListLoadBalancer

DynamicServerListLoadBalancer拉取eureka-server服务

然后根据服务负载均衡规则选择某个服务

修改负载均衡

在服务中对所有服务器的请求

```
@Bean
public IRule randomRule()
{
    //从轮询变成随机
    return new RandomRule();
}
```

在服务中对某个服务器的请求

```
userservice: #针对某个服务器
ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule #负载均衡规则
```

饥饿加载

修改配置

ribbon默认是采用懒加载，即第一次访问才会去创建loadbalanceclient，请求时间会很长，而饥饿加载则会在项目启动时创建，降低第一次访问的耗时。

```
ribbon:  
  eager-load:  
    enabled: true #开启饥饿加载  
    #clients: 指定对某个服务饥饿加载  
    clients是数组  
    k: [v1,v2,v3]  
  k:  
  
    -- v1  
  
    -- v2
```

Nacos服务注册

nacos比eureka功能更加丰富

启动错误

https://blog.csdn.net/qg_41848006/article/details/108816658?ops_request_misc=&request_id=&biz_id=102&utm_term=java.lang.IllegalArgumentException&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-2-108816658.142^v50^control_1,201^v3^add_ask&spm=1018.2226.3001.4187

启动

D:\nacos\bin的start.cmd后获取网址，账号密码为nacos

父引入依赖

```
<dependencyManagement>  
  <dependencies>  
    <!-- nacos的管理依赖 -->  
    <dependency>  
      <groupId>com.alibaba.cloud</groupId>  
      <artifactId>spring-cloud-alibaba-dependencies</artifactId>  
      <version>2.2.5.RELEASE</version>  
      <type>pom</type>  
      <scope>import</scope>  
    </dependency>
```

子工程

```
<!-- nacos客户端依赖包-->  
  <dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>  
  </dependency>
```

子工程修改配置

```
spring:  
  application:  
    name: orderservice #该服务名称  
  cloud:  
    nacos:  
      discovery:  
        server-addr: localhost:8848 #nacos服务地址
```

服务，集群，实例

一个服务可以有多个实例，多个不同或者相同实例可以形成集群

```
cloud:  
  nacos:  
    discovery:  
      server-addr: localhost:8848 #nacos服务地址  
      cluster-name: GD #集群名称
```

在服务中优先选择同一个集群，然后随机负载均衡

```
userServer:  
  ribbon:  
    NLoadBalancerRuleClassName: com.alibaba.cloud.nacos.ribbon.NacosRule
```

权重

在nacos修改权重，则可改变请求，权重为0不会被访问

环境隔离

在naocos创建命名空间，然后每个namespace都有唯一id

```
cloud:  
  nacos:  
    discovery:  
      server-addr: localhost:8848 #nacos服务地址  
      cluster-name: GD  
      namespace: f9093cc9-de53-4740-baf4-d3c946428abc #命名空间id
```

不同环境之间无法交流

临时实例

eureka中消费者只会定时拉取服务。

nacos除了定时拉取服务外会主动推送变更消息push

nacos：临时实例采用心跳检测，下线会被剔除，变更不及时

非临时实例nacos主动询问，下线不会被剔除，变更及时

默认实例都是临时实例

```
ephemeral: true
```

配置管理

配置管理服务 读取配置

注册中心对配置进行热更新

DataID格式：服务名称-环境.文件后缀

配置内容是新更新的内容，只要有变化的配置

配置获取步骤

原先 项目启动读取本地配置文件application.yml 创建spring容器 加载bean

现在 项目启动读取nacos中配置文件 读取本地配置文件application.yml 创建spring容器 加载bean
bootstrap.yml

提供者和消费者服务引入依赖后新建bootstrap.yml，因为这个文件优先级高于application.yml

```
<!-- nacos配置管理依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
```

写bootstrap.yml，目的是让其知道DataID格式：服务名称-环境.文件后缀

```
spring:
  application:
    name: orderservice #服务名称
  profiles:
    active: dev #环境
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
      config:
        file-extension: yaml #文件后缀
```

```
用@Value("${pattern.dateformat}")
```

对应配置详情orderservice-dev.yml中的pattern:

dateformat: yyyy-MM-dd HH:mm:ss

热更新

```
@RestController
@RequestMapping("/order")
public class OrderController {
```

或者不用@RequestMapping

推荐

直接用

首先配置文件文件设置初始值，前置代号.**属性**

然后在类上面@**ConfigurationProperties**

即给类的属性初始化值，注意类的属性要有**set, get**方法

但是要有@Component, @Service, @Controller, @Repository //将类定义为一个bean的注解
或者 在配置类上@EnableConfigurationProperties({类名.**class**})定义为Bean。

多环境配置共享

```
spring:  
  application:  
    name: orderservice #服务名称  
  profiles:  
    active: dev #环境  
  cloud:  
  nacos:  
    discovery:  
      server-addr: localhost:8848  
    config:  
      file-extension: yaml #文件后缀
```

读到

服务名称-环境.文件后缀

和

服务名称.文件

优先级

服务名-profile.yaml>服务名称.yaml>本地配置

nacos集群

将\nacos\conf\下的cluster.conf.example文件改为cluster.conf

将\nacos\conf\下的application.properties文件

```
server.port=8848
#多台nacos, port不一样
### If use MySQL as datasource:
spring.datasource.platform=mysql

### Count of DB:
db.num=1

### Connect URL of DB:
db.url.0=jdbc:mysql://127.0.0.1:3306/nacos?
characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true
&useUnicode=true&useSSL=false&serverTimezone=UTC
db.user.0=nacos
db.password.0=nacos
#删掉, db.num是mysql数量, url和user, password都要修改
```

将startup.cmd的
rem set MODE="cluster"

set MODE="standalone"
改为 set MODE="cluster"
cmd下直接startup.cmd

feign

```
restTemplate.getForObject("http://userServer/user/", Tbuser.class);
```

存在以下问题：

代码可读性差，编程体验不统一

参数复杂URL难以维护

引入依赖

```
<!-- feign客户端依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

开启自动装装配

```
@SpringBootApplication
@EnableFeignClients
public class Cloud01Application {
```

用法

```
@FeignClient("userServer") //服务名称
public interface UserClient {
    //请求
    @GetMapping("/user")
    String findById();
}
```

最佳实践

创建一个maven引入依赖

```
<dependencies>
    <!-- feign客户端依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-openfeign</artifactId>
    </dependency>
</dependencies>
```

将功能写到这里

```
package clients;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(value = "userServer" ,configuration = {FeignConfiguration.class}) //服务名称
public interface UserClient {
    //请求
    @GetMapping("/user")
    String findById();
}
```

```
package clients;

import feign.Logger;
import org.springframework.context.annotation.Bean;

public class FeignConfiguration {

    @Bean
    public Logger.Level loggerLevel() {
        return Logger.Level.BASIC;
    }
}
```

```
package clients;

import java.io.Serializable;

public class TbUser implements Serializable {
    ....
}
```

然后所要用的服务引入该api

```
<!-- 引入自定义api-->
<dependency>
    <groupId>groupId</groupId>
    <artifactId>feign-api</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

开启feign客户端

```
@SpringBootApplication
@Slf4j
@EnableFeignClients
public class Cloud01Application {
```

扫描不到UserClient，是另一个springboot

```
@Autowired
UserClient userClient;
@RequestMapping("/test")
public String test() {
    //return restTemplate.getForObject("http://userServer/user/", TbUser.class);
    return userClient.findById();
}
```

要扫描@EnableFeignClients(clients = UserClient.class)

日志

失效原因

跟springboot冲突了，所以要修一下

logging.level.com.dudu=DEBUG: com.dudu包下所有class以DEBUG级别输出

配置文件

```
feign:
  client:
    config:
      default:
        loggerLevel: FULL
```

Bean

麻烦需要再了解

连接池

引入依赖

```
<!--      httpclient依赖-->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-httpclient</artifactId>
</dependency>
```

开启配置

```
feign:
  httpclient:
    enabled: true
    max-connections: 200
    max-connections-per-route: 50
```

网关gateway

对请求进行身份认证和权限校验

然后进行服务路由，负载均衡

还可以进行请求限由

引入依赖

网关本身也是个服务，所以需要nacos发现，也需要nacos依赖

```
<!--      nacos服务注册发现依赖-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-
discovery</artifactId>
</dependency>
<!--      网关gateway-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
</dependencies>
```

```
server:
  port: 1010
spring:
  application:
    name: gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848 #nacos地址
    gateway:
      routes:
        - id: user-service #路由标识，必须唯一
          uri: lb://userservice #统一资源标识符，路由的目的地址，userservice是服务名
称！
      predicates: #判断请求是否符合规则，路径工厂，还有很多请求判断
        - Path=/user/** #路径判断，判断路径是否以/user开头
```

```
- After=2031-04-13T15:14:47.433+08:00[Asia/Shanghai]
- id: order-service
  uri: lb://orderservice
  predicates:
    - Path=/order/**
```

路由过滤器gatewayFilter

对网关的请求进行处理

```
gateway:
  routes: #路由
    - id: user-service #路由标识，必须唯一
      uri: lb://userservice #路由的目的地址
      predicates: #判断请求是否符合规则
        - Path=/user/** #路径判断，判断路径是否以/user开头
          #- After=2031-04-13T15:14:47.433+08:00[Asia/Shanghai]
      filters: #路由过滤器
        - AddRequestHeader=Truth,zyj #对userservice服务添加请求头
    - id: order-service
      uri: lb://orderservice
      predicates:
        - Path=/order/**

  default-filters: #默认过滤器，对所有请求都生效
```

全局过滤器

```
package com.gateway;

import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpStatus;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

@Order(0)//过滤器顺序，越小越高
@Component
public class AuthorizedFilter implements GlobalFilter {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //获取请求参数
        ServerHttpRequest request = exchange.getRequest();
        MultiValueMap<String, String> queryParams = request.getQueryParams();
        //获取参数authorization参数
        String authorization = queryParams.getFirst("authorization");
        //判断是否等于admin
        if ("admin".equals(authorization))
        {
            //是，放行
        }
    }
}
```

```
        return chain.filter(exchange);
    }
    //否，拦截
    //设置状态码
    exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
    return exchange.getResponse().setComplete();
}
}
```

过滤器级别

order值越小越高

路由过滤器和defaultfilter的order由spring指定，默认从1开始递增

当三个过滤器出现一样时候，会按照defaultFilter>路由过滤器>globalfilter顺序执行

跨域

域名不同或者域名相同端口不同

浏览器禁止ajax跨域请求，即服务器和端口号不同于当前的网址端口号

网关处理跨域采用CORS方案，只需要简单配置即可实现

```
gateway:
  globalcors: #全局跨域处理
    add-to-simple-url-handler-mapping: true #解决option请求被拦截问题
    cors-configurations:
      allowedOrigins: #允许哪些网站跨域请求
        - "http://localhost:8080"
      allowedMethods: #允许请求方式
        - "GET"
        - "POST"
        - "DELETE"
        - "PUT"
      allowedHeaders: "*" #允许请求中携带的头信息
      allowCredentials: true #是否允许携带cookie
      maxAge: 3306 #跨域检测有效期
```

Docker部署

docker和虚拟机差异

docker是一个系统进程，而虚拟机是在操作系统中的操作系统

docker体积小启动速度快，性能好。虚拟机体积大启动速度慢，性能一般

镜像

docker将应用程序及其所需的依赖，函数库，环境，配置等文件打包在一起，称为镜像。

容器

镜像中的应用程序运行后形成的进程就是容器，只是docker会给容器做隔离，对外不可见。

docker结构

服务端：接收命令或远程请求，操作镜像或容器

客户端：发送命令或者请求到docker服务端

dockerhub

镜像托管服务器

docker安装

linux下执行

```
yum install -y yum-utils \
    device-mapper-persistent-data \
    lvm2 --skip-broken
```

更新本地镜像源

```
yum-config-manager \
--add-repo \
https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

sed -i 's/download.docker.com/mirrors.aliyun.com\//docker-ce/g' \
/etc/yum.repos.d/docker-ce.repo

yum makecache fast
```

下载docker

```
yum install -y docker-ce
```

启动docker前关闭防火墙

直接简单粗暴防火墙都关了

```
# 关闭
systemctl stop firewalld
# 禁止开机启动防火墙
systemctl disable firewalld
```

启动docker

```
systemctl start docker
```

查看状态

```
systemctl status docker
```

重启

```
systemctl restart docker
```

配置镜像

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<- 'EOF'
{
  "registry-mirrors": ["https://dfjmvoef.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

docker基本操作

镜像相关命令

镜像名称一般分为repository: tag

例如mysql:5.7

如果没有指定tag则下载最新

```
帮助手册
docker --help
具体命令帮忙
docker 命令 --help
例如
docker save --help
```

```
docker images    *# 查看镜像列表*
docker rmi      *# 移出镜像*
docker pull     *# 拉取镜像*
docker push     *# 推送镜像到服务*
docker save     *# 保存镜像到本地*
docker load     *# 本地加载镜像*
```

dockerhub网址

<https://hub.docker.com/>

容器相关命令

docker run 镜像变成容器运行

docker pause 暂停

docker unpause 不暂停

docker stop [容器名字] 停止

docker start 运行

docker rm 删除指定容器

```
docker run --name some-nginx -d -p 8080:80 some-content-nginx
docker run: 创建并运行一个容器
--name: 给容器起一个名字
-d: 后台运行容器
-p: 将宿主机端口和容器端口映射
some-content-nginx: 镜像名称
```

```
[root@VM-8-2-centos ~]# docker run --name mynginx -p 80:80 -d nginx
基于nginx镜像创建一个后台运行的容器mynginx, 映射宿主端口80到容器端口80
```

docker ps 查看运行中的容器信息

docker ps -a 查看容器信息

docker logs [容器名] 查看日志信息

docker stop [容器名] 停止

进入容器

步骤：

1) 进入容器。进入我们刚刚创建的nginx容器的命令为：

```
docker exec -it mn bash
也可以直接
直接进入并执行redis-cli
docker exec -it mn redis-cli
```

命令解读：

- docker exec：进入容器内部，执行一个命令
- -it：给当前进入的容器创建一个标准输入、输出终端，允许我们与容器交互
- mn：要进入的容器的名称
- bash：进入容器后执行的命令，bash是一个linux终端交互命令

```
出去容器
exit
```

容器修改问题

不便于修改

数据不可复用

容器与数据耦合，升级维护困难

数据卷

数据卷是一个虚拟目录，指向宿主机文件系统中的某个目录

docker volume [command]

docker volume命令是数据卷操作，根据命令后跟随的command来确定下一步的操作

create 创建一个volume

inspect [volumeName] 某一个volume详细信息

ls 列出所有volume

prune 删除未使用的volume

rm 删除指定volume

挂载

```
docker run --name some-nginx -d -p 8080:80 some-content-nginx -v [数据卷]:具体位置
```

docker run: 创建并运行一个容器

--name: 给容器起一个名字

-d: 后台运行容器

-p: 将宿主机端口和容器端口映射

some-content-nginx: 镜像名称

-v: 容器挂载到数据卷

```
[root@VM-8-2-centos ~]# docker run --name mynginx -p 80:80 -d nginx
```

基于nginx镜像创建一个后台运行的容器mynginx, 映射宿主端口80到容器端口80

例如我们进入nginx容器内部知道nginx的html目录所在位置为/usr/share/nginx/html

我们首先创建html数据卷docker volume html后

```
docker run --name mn -d -p 80:80 -v html:/usr/share/nginx/html nginx
```

由于当我们创建一个数据卷时，它存储在 Docker 主机上的一个目录中。数据卷由 Docker 管理，并与主机的核心功能隔离。

我们根据docker volume inspect 数据卷名

即可详细知道它挂载到主机的位置进行修改

不创建数据卷直接

```
docker run --name mn -d -p 80:80 -v html:/usr/share/nginx/html nginx
```

也会帮我们创建数据卷

宿主机目录直接挂载到容器

```
docker run \
--name mysql \
-e MYSQL_ROOT_PASSWORD=mdjfbzyj515 \
-p 3306:3306 \
-v /tmp/mysql/conf/hmy.cnf:/etc/mysql/conf.d/hmy.cnf \
-v /tmp/mysql/data:/var/lib/mysql \
-d mysql:5.7.25
```

```
docker run -d -p 3306:3306 --name demo -v /usr/local/mysql/data:/var/lib/mysql -v
/etc/localtime:/etc/localtime -e MYSQL_ROOT_PASSWORD=mdjfbzyj515 mysql:5.7.25
```

数据卷好处是不需要提前知道目录位置

宿主机目录好处是自己管理

镜像

镜像是分层结构，每一层称为一个layer

baseimage层：包括基本的系统函数库，环境变量，文件系统

entrypoint：入口，是镜像中应用启动的命令

其它：在baseimage基础上添加依赖，安装程序，完成整个应用的安装和配置

dockerfile

dockerfile本质是一个文件，通过指令描述镜像的构建过程

dockerfile第一行必须是from，从一个基础镜像来构建，基础镜像可以是基本操作系统如ubuntu，

```
# 指定基础镜像
FROM ubuntu:16.04
# 配置环境变量，JDK的安装目录
ENV JAVA_DIR=/usr/local

# 拷贝jdk和java项目的包
COPY ./jdk8.tar.gz $JAVA_DIR/
COPY ./docker-demo.jar /tmp/app.jar

# 安装JDK
RUN cd $JAVA_DIR \
    && tar -xf ./jdk8.tar.gz \
    && mv ./jdk1.8.0_144 ./java8

# 配置环境变量
ENV JAVA_HOME=$JAVA_DIR/java8
ENV PATH=$PATH:$JAVA_HOME/bin

# 暴露端口
EXPOSE 8090
# 入口，java项目的启动命令
ENTRYPOINT java -jar /tmp/app.jar
```

也可以是其它人制作好的镜像，

例如 java:8-alpine

```
# 指定基础镜像
FROM java:8-alpine
# 暴露端口
EXPOSE 8090
# 入口，java项目的启动命令
ENTRYPOINT java -jar /tmp/app.jar
```

自定义镜像

首先是当前文件下有dockerfile，

如果不是用镜像java:8-alpine

可以不需要当前文件下有jdk8.tar.gz

其次还需要我们需要运行的jar包，

接着创建镜像

docker build -t [repository]:[tag] .

.代表在当前位置创建镜像

dockercompose

要先下载docker-compose到/usr/local/bin

用linux加执行权

chmod +x /usr/local/bin/docker-compose

```
# 补全命令
curl -L
https://raw.githubusercontent.com/docker/compose/1.29.1/contrib/completion/bash/
docker-compose > /etc/bash_completion.d/docker-compose
```

如果这里出现错误，需要修改自己的hosts文件：

```
echo "199.232.68.133 raw.githubusercontent.com" >> /etc/hosts
```

compose文件是一个文本文件，通过指令定义集群中的每一个容器如何运行，基于compose部署微服务

微服务部署

是一些繁琐的记住操作，等需要再用

https://www.bilibili.com/video/BV1LQ4y127n4/?is_story_h5=false&p=59&share_from=ugc&share_medium=iphone&share_plat=ios&share_source=WEIXIN&share_tag=s_ixtamp=1664192927&unique_k=EKd8YEL&vd_source=003e9f54d7ed05b77aee32ac0f194b0a

镜像仓库

不学了妈的

mq (meesageQueue)

消息队列，就是存放消息

网址

<https://www.rabbitmq.com/>

下载

拉取mq.tar到linux后加载镜像

运行

```
docker run \
-e RABBITMQ_DEFAULT_USER=root \
-e RABBITMQ_DEFAULT_PASS=mdjfbzyj515 \
--name mq \
--hostname mq1 \
-p 15672:15672 \
-p 5672:5672 \
-d rabbitmq:3-management
```

hostname是主机名

15672是ui界面端口

5672是消息端口

amqp

advanced message queuing protocol是用于在应用程序或之间传递业务消息的开放标准，该协议与语言和平台无关，更符合微服务中独立性的要求

springAmqp

父工程引入依赖

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

不用

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

则子工程不用引入了

配置文件

```
spring:
  rabbitmq:
    host: 175.178.236.116 #ip地址
    port: 5672 #端口
    username: root #账号
    password: 123456 #密码
    virtual-host: / #虚拟主机的位置
```

发送消息

```
@Autowired  
private RabbitTemplate rabbitTemplate;  
@Test  
public void testSendMessage()  
{  
    rabbitTemplate.convertAndSend("simple.que", "hello,mq");  
}
```

处理消息

```
@Component  
public class Listener {  
    @RabbitListener(queues ="simple.queue" )  
    public void listener(String message)  
    {  
        System.out.println("获取到的消息:"+message);  
    }  
}  
//消息一旦消费就从队列删除
```

消息预取机制

默认是平均分给消费者，即轮询，因为默认消息预取是最大值，则消费者平均分配了，我们可以将预取设为1则能者多劳了

channel通道先预取消息

```
rabbitmq:  
    host: 175.178.236.116 #ip地址  
    port: 5672 #端口  
    username: root #账号  
    password: 123456 #密码  
    virtual-host: / #虚拟主机的位置  
    listener:  
        simple:  
            prefetch: #通道预取信息
```

RabbitMQ 四大核心概念

- 1、生产者：产生数据发送消息的程序是生产者。
- 2、交换机：交换机是 RabbitMQ 非常重要的一个部件，一方面它接收来自生产者的消息，另一方面他讲消息推送到队列中。交换机必须确切的知道如何处理它接受的消息，是将这些消息推送到特定队列还是送到多个队列，亦或者是把消息丢弃，这个得由交换机类型决定。交换分为如下及中：直连交换机 (direct exchange) 、主题交换机 (topic exchange) 、标题交换机 (headers exchange) 、扇出交换机 (fanout exchange)
- 3、队列：队列是 RabbitMQ 内部使用的一种数据结构，尽管消息流经 RabbitMQ 和应用程序，但它们只能存储在队列中。队列仅受主机的内存和磁盘限制的约束。本质上是一个大的消息缓冲区。许多生产者可以将消息发送到一个队列中，许多消费者可以从队列中接收数据，这就是我们使用队列的方式。
- 4、消费者：消费者和接受具有相似的含义。消费者大多时候是一个等待接受消息的程序，请注意生产者、消费者、消息中间件大多时候并不在统一机器上，统一个应用既可以是生产者也可以是消费者。

简单模式和工作模型

1 "Hello World!"

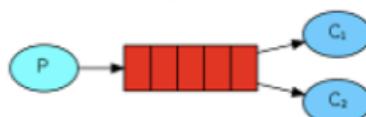
The simplest thing that does something



- [Python](#)
- [Java](#)

2 Work queues

Distributing tasks among workers (the [competing consumers pattern](#))



简单模式：一个消费者消费一个生产者生产的信息

工作模式：一个生产者生产信息，多个消费者进行消费，但是一条消息只能消费一次

结构都为生产者-队列-消费者，消息一旦消费就从队列删除

发布订阅模式

交换机的作用

接收publisher发送的消息

将消息按照规则路由到与之绑定的队列

不能缓存消息，路由失败消息丢失

声明队列，交换机，绑定关系的bean是什么

Queue, FanoutExchange, Binding

四种交换机

Header Exchange (头交换机)

与routingKey无关，匹配机制是匹配消息头中的属性信息。在绑定消息队列与交换机之前声明一个map键值对，通过这个map对象实现消息队列和交换机的绑定。当消息发送到RabbitMQ时会取到该消息的headers与Exchange绑定时指定的键值对进行匹配；如果完全匹配则消息会路由到该队列，否则不会路由到该队列。

匹配规则x-match有下列两种类型：

x-match = all : 表示所有的键值对都匹配才能接受到消息

x-match = any : 表示只要有键值对匹配就能接受到消息

扇出交换机

Fanout Exchange (扇出交换机)，类似于广播，只要队列与该类型的交换机绑定，所有发送到该交换机的信息都会被转发到所有与之绑定的队列，与routingKey无关。

绑定

```
@Configuration
public class FanoutConfig {
    //交换机
    @Bean
    public FanoutExchange fanoutExchange() {
        //交换机名字为fanoutExchange
        return new FanoutExchange("fanoutExchange");
    }
    //队列
    @Bean
    //import org.springframework.amqp.core.Queue;
    public Queue fanoutQueue1() {
        //队列名字为fanoutQueue1
        return new Queue("fanoutQueue1");
    }
    //队列1
    @Bean
    //import org.springframework.amqp.core.Queue;
    public Queue fanoutQueue2() {
        //队列名字为fanoutQueue2
        return new Queue("fanoutQueue2");
    }
    //绑定队列fanoutQueue1到交换机fanoutExchange
    @Bean
    public Binding fanoutBinding(Queue fanoutQueue1, FanoutExchange
fanoutExchange) {
        //绑定队列1到交换机
        return BindingBuilder.bind(fanoutQueue1).to(fanoutExchange);
    }
    //绑定队列fanoutQueue2到交换机fanoutExchange
    @Bean
    public Binding fanoutBinding1(Queue fanoutQueue2, FanoutExchange
fanoutExchange) {
        //绑定队列2到交换机
        return BindingBuilder.bind(fanoutQueue2).to(fanoutExchange);
    }
}
```

Exchange: fanoutExchange

▼ Overview

Message rates last minute ?

Currently idle

Details

Type	fanout
Features	durable: true
Policy	

▼ Bindings

This exchange



To	Routing key	Arguments	
fanoutQueue1			<button>Bind</button> <button>Unbind</button>
fanoutQueue2			<button>Bind</button> <button>Unbind</button>

订阅

```
@RabbitListener(queues ="fanoutQueue1" )
public void fanoutQueue1listener(String message)
{
    System.out.println("交换机队列1获取到的消息:"+message);
}
@RabbitListener(queues ="fanoutQueue2" )
public void fanoutQueue2listener(String message)
{
    System.out.println("交换机队列2获取到的消息:"+message);
}
```

发布

```

@RequestMapping("fanout")
public String testFanout()
{
    //交换机名称
    String exchange="fanoutExchange";
    //消息
    String message="交换机发送消息测试";
    //发送
    rabbitTemplate.convertAndSend(exchange,"",message);
    return "\\"+exchange+"发送消息测试\"";
}

```

直连交换机

Direct Exchange (直连交换机)

直连交换机的特点是消息队列通过routingKey与交换机进行绑定，相同的routingKey会获得相同的消息。一个队列可以通过多个不同的routingKey与交换机进行绑定。不同的队列也可以通过相同的routingKey绑定交换机。

绑定与订阅

```

@RabbitListener(bindings =
    @QueueBinding(value = @Queue(name = "directQueue1"),
                  exchange = @Exchange(name = "directExchange",type =
ExchangeTypes.DIRECT),
                  key = {"red","blue"})
)
public void direct1listener(String message)
{
    System.out.println("直连交换机1:"+message);
}

@RabbitListener(bindings =
    @QueueBinding(value = @Queue(name = "directQueue2"),
                  exchange = @Exchange(name = "directExchange",type =
ExchangeTypes.DIRECT),
                  key = {"red","yellow"})
)
public void direct2listener(String message)
{
    System.out.println("直连交换机2:"+message);
}

```

发布

```

@RequestMapping("direct")
public String testDirect(String routingKey)
{
    //交换机名称
    String exchange="directExchange";
    //消息
    String message="直连交换机发送消息测试";
    //根据routingkey发送
    rabbitTemplate.convertAndSend(exchange,routingKey,message);
    return "\"交换机发送消息测试\"";
}

```

主题交换机

Topic Exchange（主题交换机）应用范围最广的交换机类型，消息队列通过消息主题与交换机绑定。一个队列可以通过多个主题与交换机绑定，多个消息队列也可以通过相同消息主题和交换机绑定。并且可以通过通配符（*或者#）进行多个消息主题的适配。

消息主题的一般格式为xxx.xxx.xxx（x为英文字母，每个单词用英文句号隔开）。*通配符可以适配一个单词，#可以适配零个或者多个单词。

通配符适配如下:*.xxx.#。此主题可以适配xxx前面只有一个单词后面有零个或者多个单词的所有消息主题。

绑定和订阅

```

@RabbitListener(bindings =
    @QueueBinding(
        value = @Queue(name = "topicQueue1"),
        exchange = @Exchange(name = "topicExchange",type =
ExchangeTypes.TOPIC),
        key = "china.#"
    )
)
public void topicQueue1(String msg)
{
    System.out.println("主题交换机地区:"+msg);
}

@RabbitListener(bindings =
@QueueBinding(
    value = @Queue(name = "topicQueue2"),
    exchange = @Exchange(name = "topicExchange",type = ExchangeTypes.TOPIC),
    key = "#.news"
)
)
public void topicQueue2(String msg)
{
    System.out.println("主题交换机新闻:"+msg);
}

```

发布

```
@RequestMapping("topic")
public String testTopic(String routingKey)
{
    //交换机名称
    String exchange="topicExchange";
    //消息
    String message="主题交换机发送消息测试";
    //根据routingkey发送
    rabbitTemplate.convertAndSend(exchange,routingKey,message);
    return "主题交换机";
}
```

发布详解

```
public void convertAndSend(String exchange, String routingKey, Object object)
throws AmqpException {
    ....
}
exchange是交换机
routingKey是路由键
object传string类型可以，但传对象的话会序列化

content_type: application/x-java-serialized-object
序列化为:
r00ABXNyABFqYXZhLnV0aWwuSGFzaE1hcAUH2sHDFmDRAwACRgAKbG9hZEZhY3RvckkACXRocmVzaG9s
ZHhwP0AAAAAAAAX3CAAAABAAAABdAAd2V5c3IA
EwphdmEubGFuZy5JbnR1Z2VyEuKgpPeBhzgCAAFJAAV2Ywx1ZXhyABBqYXZhLmxhbmcuTnVtYmVyhqyV
HQuU4IsCAAB4cAAAAA94
```

消息转换器

spring对消息对象的处理是由org.springframework.amqp.support.converter.messageconverter处理，而默认实现是由simplemessageconverter，基于jdk的objectoutputstream完成序列化

修改只需要顶一个messageconverter类型的bean即可以，推荐用json方式序列化

先在父工程引入依赖

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

修改org.springframework.amqp.support.converter.messageconverter的bean

```
//import org.springframework.amqp.support.converter.MessageConverter;
public MessageConverter messageConverter()
{
    return new Jackson2JsonMessageConverter();
}
```

mq消息可靠性

发送时丢失

生产者到交换机

交换机到队列

mq宕机，队列丢失消息

消费者接收消息后未消费就宕机

生产者消息确认

publisher-confirm 发送者确认

消息成功投递到交换机，返回ack

消息未投递到交换机，返回nack

消息发送过程中出现异常，没有收到回执

publisher-return，发送者回执

消息投递到交换机，但没有路由到队列，返回ack，及路由失败原因

注意确认机制发送消息时，需要给每一个消息设置一个全局唯一id，以区分不同消息，避免ack冲突

```
//生产者服务器
rabbitmq:
  publisher-confirm-type: correlated
    #simplie是同步等待confirm结果，直到超时
    # correlated异步回调，定义confirmCallback，MQ返回结果时候会回调这个confirmCallback
  publisher-returns: true
    #消息投递到交换机，但没有路由到队列，返回ack，及路由失败原因，基于回调机制，定义
  returncallback
    template:
      mandatory: true
        #注意定义了publisher-returns要定义消息路由失败策略，true调用returnCallback，false丢
        弃消息
```

ConfirmCallback对消息发送到交换机的处理

```
@Test
public void testSendMessage2SimpleQueue() throws InterruptedException {
    // 1.准备消息
    String message = "hello, spring amqp!";
    // 2.准备CorrelationData
    // 2.1.消息ID
    CorrelationData correlationData = new
    CorrelationData(UUID.randomUUID().toString());
    // 2.2.准备confirmCallback
    correlationData.getFuture().addCallback(result -> {
        // 判断结果
        if (result.isAck()) {
            // ACK
            log.debug("消息成功投递到交换机！消息ID: {}", correlationData.getId());
        } else {
            // NACK
            log.error("消息投递到交换机失败！消息ID: {}", correlationData.getId());
            // 重发消息
        }
    });
}
```

```

}, ex -> {
    // 记录日志
    log.error("消息发送失败! ", ex);
    // 重发消息
});
// 3.发送消息
rabbitTemplate.convertAndSend("amq.topic", "a.simple.test", message,
correlationData);
}

```

returncallback对交换机发消息到队列失败的处理

```

@Slf4j
@Configuration
public class CommonConfig implements ApplicationContextAware {

    //bean工厂获取对象处理
    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
throws BeansException {
        // 获取RabbitTemplate对象
        RabbitTemplate rabbitTemplate =
applicationContext.getBean(RabbitTemplate.class);
        // 配置ReturnCallback,设置回调函数, 即消息发送到队列失败
        rabbitTemplate.setReturnCallback((message, replyCode, replyText,
exchange, routingKey) -> {
            // 判断是否是延迟消息
            Integer receivedDelay =
message.getMessageProperties().getReceivedDelay();
            if (receivedDelay != null && receivedDelay > 0) {
                // 是一个延迟消息, 忽略这个错误提示
                return;
            }
            // 记录日志
            log.error("消息发送到队列失败, 响应码: {}, 失败原因: {}, 交换机: {}, 路由
key: {}, 消息: {}",
                    replyCode, replyText, exchange, routingKey,
                    message.toString());
            // 如果有需要的话, 重发消息
        });
    }
}

```

消息持久化

交换机是持久化，队列不是。

创建队列可在可视化界面durability设为durable 持久化设置为持久化的

创建消息也需声明持久化

java代码

```

//创建交换机
@Bean
public DirectExchange Exchange()

```

```

{
    //名称，是否持久化，当没有queue与其绑定是否自动删除
    return new DirectExchange("exchange",true,false);
}
//创建队列
@Bean
public Queue queue()
{
    return QueueBuilder.durable("queue").build();
}
//创建消息持久化
public void testDurableMessage()
{
    Message message =
    MessageBuilder.withBody("context".getBytes(StandardCharsets.UTF_8)) //设置内容
    .setDeliveryMode(MessageDeliveryMode.PERSISTENT)      //设置持久化
    .build();
    //发送消息
    new RabbitTemplate().convertAndSend("queue",message);
}
//spring默认情况下交换机，队列，消息都是持久化

```

消费者消息确认

rabbitmq支持消费者确认机制，即消费者处理消息后可以向mq发送ack回执，mq收到ack回执后才会删除该消息。

而springamqp允许三种确认模式

manual：手动ack，需要在业务代码结束后，调用api发送ack

auto：自动ack，由spring检测listener代码是否出现异常，没有异常则返回ack，抛出异常则返回nack

none：关闭ack，mq假定消费者获取消息后成功处理，因此消息投递后立即被删除

配置消费者服务器的配置文件

```

rabbitmq:
  listener:
    simple:
      prefetch: 1
      acknowledge-mode: auto

```

失败重试机制

消费者出现异常后，消息会不断requeue重新入队到队列，再重新发送给消费者，然后再次异常，再次requeue，无限循环，导致mq的消息处理飙升，带来不必要的压力

我们可以利用spring的retry机制，在消费者出现异常时利用本地重试，而不是无限制的requeue到mq队列

```

rabbitmq:
  listener:
    retry:
      enabled: true #开启消费者失败重试
      initial-interval: 1000      #初始的失败等待时长为1秒
      multiplier: 3                #下次失败等待时长倍数, 下次等待时长
      =multiplier*lastInterval
      max-attempts: 4            #最大重试次数
      stateless: true #如果业务包含事务就写为false, 默认是true

```

消费者失败消息处理策略

在开启重试模式后，重试次数耗尽，如果消息依然失败，则需要有MessageRecoverer接口来处理，它包含三种不同的实现：

`RejectAndDontRequeueRecoverer`: 重试耗尽后，直接reject，丢弃消息。默认就是这种方式

`ImmediateRequeueMessageRecoverer`: 重试耗尽后，返回nack，消息重新入队

`RepublishMessageRecoverer` : 重试耗尽后，将失败消息投递到指定的交换机

//覆盖spring的bean的RepublishMessageRecoverer来处理失败重试最终还是失败的消息

`@Bean`

```
public MessageRecoverer republisherMessageRecoverer(RabbitTemplate
```

```
rabbitTemplate)
```

```
{
```

```
  //rabbit, 交换机, 队列
```

```
  return new
```

```
RepublishMessageRecoverer(rabbitTemplate, "errExchange", "errorKey");
```

```
}
```

可靠性总结

开启生产者确认机制，确保生产者的消息能到达队列

开启持久化功能，确保消息未消费前能再队列中不会丢失

开启消费者确认机制为auto，由spring确认消息处理成功后完成ack

mq死信交换机（做延时发消息）

当一个队列中消息满足下列情况之一，可以成为死信

一个消息发出去但消费失败

消息是一个过期消息，超时无人消费

投递的队列消息堆积满了，最早的消息可能成为死信

如果该队列配置了dead-letter-exchange属性，指定了一个交换机，那么队列中的死信就会投递到这个交换机中，而这个交换机称为死信交换机（Dead letter exchange，简称DLX）

如何给队列绑定死信交换机

给队列设置dead-letter-exchange和dead-letter-routing-key指定交换机和队列

ttl也就是time-to-live，如果一个队列中的消息ttl结束仍未被消费，则会变成死信，ttl超时分为两种情况：

消息所在队列设置了存活时间

消息本身设置了存活时间

```

//给队列设置延时时间
@Bean
public Queue ttlQueue(){
    return QueueBuilder
        .durable("ttl.queue")          //持久化创建队列
        .ttl(10000)                   //存活毫秒
        .deadLetterExchange("dl.direct") //死信交换机
        .deadLetterRoutingKey("dl")     //死信key
        .build();
}

```

```

//给消息设置延时时间
@Test
public void testTTLMessage() {
    // 1.准备消息
    Message message = MessageBuilder
        .withBody("hello, ttl messsage".getBytes(StandardCharsets.UTF_8))//内
容
        .setDeliveryMode(MessageDeliveryMode.PERSISTENT)      //持久消息， 默认也是
持久的
        .setExpiration("5000")                                //设置存活时间
        .build();

    // 2.发送消息
    rabbitTemplate.convertAndSend("ttl.direct", "ttl", message);
    // 3.记录日志
    log.info("消息已经成功发送！");
}

```

注意如果给队列和消息都设置延时时间！则以较短的时间为准！！则会做到延时发信

原理都是让消息成为死信，配置一个死信交换机。

还可以使用插件扩展简易操作，但该内容就不扩展了

mq惰性队列

消息堆积问题：当生产者发送消息的速度超过了消费者处理消息的速度，就会导致队列中的消息堆积，直到队列存储消息达到上限，最早接收到的消息，可能就会成为死信，会被丢弃，这就是消息堆积问题

处理消息堆积：

增加更多消费者，提高消费速度

消费者内开启线程池加快消息处理速度

扩大队列容积，提高堆积上限

惰性队列：

接收到的消息后直接存入磁盘而非内存

读取时加载到内存

支持数百万条的消息存储

缺点：

- 基于磁盘存储，消息时效性会降低
- 性能受限于磁盘的IO

设置一个队列为惰性队列，只需要在声明队列时，指定`x-queue-mode`属性为`lazy`即可。可以通过[命令行]将一个运行中的队列修改为惰性队列

```
rabbitmqctl set_policy Lazy "^lazy-queue$" '{"queue-mode":"lazy"}' --apply-to-queues
```

或者用`springamqp`声明惰性队列

```
//bean
@Bean
public Queue lazyQueue() {
    return QueueBuilder.durable("lazy.queue")
        .lazy()
        .build();
}

//注解，利用 arguments = @Argument(name = "x-queue-mode", value = "lazy"
@RabbitListener(queuesToDeclare = @Queue(
    name = "lazy.queue",
    durable = "true",
    arguments = @Argument(name = "x-queue-mode", value = "lazy"))
))

public void listenLazyQueue(String msg) {
    log.info("接收到了 lazy.queue 队列的消息:{}" ,msg);
}
```

mq集群

elasticsearch

elasticserach是elasticStack核心

elasticsearch是一个开源的分布式搜索引擎，可以用来实现搜索，日志统计，分析，系统监控等功能

倒排索引将文档分成多个名词，存入词条term，然后记录其文档id

查询时候会将查询字段 分成一个或者多个词条 去词条列表 查询其文档id最后根据其id 查数据库。

eliasticsearch是面向文档存储的，可以是数据库中一条商品数据，一个订单信息。文档数据会序列化为json格式后存储在elasticsearch中

索引是相同类型的文档的集合

概念对比

mysql	elasticsearch	说明
table	index	索引index就是文档的集合
row	document	文档document就是一条条数据，文档是json格式
column	field	字段，就是json文档中的字段
schema	mapping	mapping映射就是索引中对文档的约束
sql	dsl	dsl是elasticsearch提供的json风格的请求语句，用来操作elasticsearch，实现crud

mysql擅长事务类型操作，可以确保数据的安全和一致性

elasticsearch擅长海量数据的搜索，分析，计算

docker部署

docker创建网络

```
docker network create es-net
```

```
docker run -d \
--name es \
-e "ES_JAVA_OPTS=-Xms512m -Xmx512m" \
-e "discovery.type=single-node" \
-v es-data:/usr/share/elasticsearch/data \
-v es-plugins:/usr/share/elasticsearch/plugins \
--privileged \
--network es-net \
-p 9200:9200 \
-p 9300:9300 \
elasticsearch:latest
```

ES_JAVA_OPTS=-Xms512m -Xmx512m 内存是512m

discovery.type=single-node 单点不是集群

-v es-data:/usr/share/elasticsearch/data 数据卷挂载
-v es-plugins:/usr/share/elasticsearch/plugins 数据卷挂载

9300端口是做集群用

--privileged 是有root权限

--network es-net 是属于哪个网络

部署kibana，可以给我们提供一个elasticsearch的可视化界面，便于我们学习

```
docker run -d \
--name kibana \
-e ELASTICSEARCH_HOSTS=http://es:9200 \
--network=es-net \
-p 5601:5601 \
kibana:latest
```

注意kibana版本要跟elastic版本一样

在Docker中，默认情况下容器与容器、容器与外部宿主机的网络是隔离开来的。当你安装Docker的时候，docker会创建一个桥接器docker0，通过它才让容器与容器之间、与宿主机之间通信

与elasticsearch在同一个网络，-e ELASTICSEARCH_HOSTS=<http://es:9200>, <http://es:9200>即是知道es的ip地址和es的端口号，让kibana可以连接es

请求

```
POST /_analyze
{
  "analyzer": "standard"
  "text": "黑马程序员学习java太棒了"
}

//中文分词不怎么样，要另外安装分词器
```

语法说明

post 请求方式

/analyze 请求路径，这里省略了<http://es>的ip：es的端口号，有kibana帮我们补充

请求参数，json风格：

analyzer 分词器类型，这里默认是standard分词器

text 是分词的内容

中文分词器安装

plugin 插件

将中文分词器插件安装到docker容器es的数据卷es-plugins

找到数据卷位置

docker volume inspect es-plugins

解压ik上传到指定数据卷位置最后重启下容器 docker restart es，查看下日志docker logs es

ik分词器有两种模式

ik_smart 最粗粒度分词

ik_max_word 最细粒度分词

ik分词的拓展和停用

修改ik分词器目录中的config目录中IkAnalyzer.cfg.xml文件

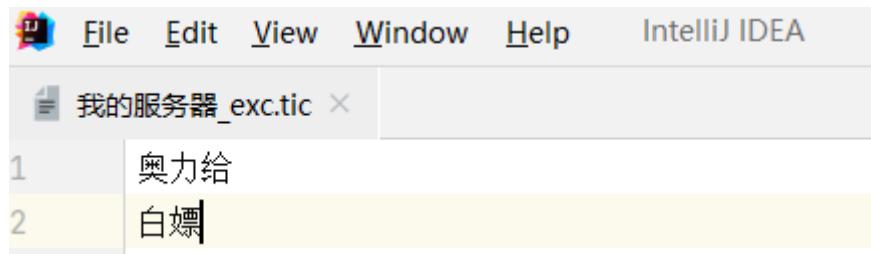
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>IK Analyzer 扩展配置</comment>
    <!--用户可以在这里配置自己的扩展字典-->
    <entry key="ext_dict">ext.dic</entry>
    <!--用户可以在这里配置自己的扩展停止词字典-->
    <entry key="ext_stopwords">stopword.dic</entry>
    <!--用户可以在这里配置远程扩展字典-->
    <!--<entry key="remote_ext_dict">words_location</entry>-->
    <!--用户可以在这里配置远程扩展停止词字典-->
    <!--<entry key="remote_ext_stopwords">words_location</entry>-->
</properties>
```

ext.dic是扩展文件

stopword.dic是扩展停止文件

在IkAnalyzer.cfg.xml同个目录下创建

文件名	大小	类型	修改时间	权限	用户/用户组
exc.tic	17 B	TIC 文件	2022/09/28 09:43	-rw-r--r--	root/root
extra_main.dic	5 MB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
extra_single_word_f...	61.7 KB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
extra_single_word_l...	10.6 KB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
extra_single_word.dic	61.7 KB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
extra_stopword.dic	156 B	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
IKAnalyzer.cfg.xml	644 B	XML 文档	2022/09/28 09:42	-rw-r--r--	root/root
main.dic	2.9 MB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
preposition.dic	123 B	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
quantifier.dic	1.8 KB	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
stopword.dic	164 B	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
suffix.dic	192 B	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root
surname.dic	752 B	DIC 文件	2022/09/28 09:35	-rw-r--r--	root/root



注意配置文件或者插件修改就要重启

索引库操作

mapping

mapping是对索引库中文档的约束

常见的mapping属性包括

type 字段数据类型
字符串**text** (可分词文本)

字符串**keyword** (精确值)

数值 **long**, **integer**,

布尔值 **boolean**

日期 **date**

对象 **object**

index是否创建索引默认为**true**

analyzer使用哪种分词器

properties 子字段

创建索引库

```
#创建索引库
PUT /test
{
  "mappings": {
    "properties": {
      "info": {
        "type": "text",
        "analyzer": "ik_smart"
      },
      "email": {
        "type": "keyword",
        "index": false
      },
      "name": {
        "type": "object",
        "properties": {
          "firstName": {
            "type": "keyword"
          },
          "lastName": {
            "type": "keyword"
          }
        }
      }
    }
  }
}
```

查看索引库

GET /索引库名

删除索引库

DELETE /索引库名

不允许修改，但可以添加

```
#添加新字段
PUT /test/_mapping
{
  "properties": {
    "age": {
      "type": "integer"
    }
  }
}
```

文档操作

添加文档

```
POST /索引库名/_doc/文档id
{
  "字段1": "值1",
  "字段2": "值2",
  "字段3": {
    "子属性1": "值3",
    "子属性2": "值4"
  }
}
#插入文档
POST /test/_doc/1
{
  "info": "黑马程序",
  "email": "123",
  "name": {
    "firstName": "一",
    "lastName": "二"
  }
}
```

查询文档

```
#查询
GET /索引库名/_doc/文档id
GET /test/_doc/1
```

删除文档

```
#删除
DELETE /索引库名/_doc/文档id
DELETE /test/_doc/1
```

修改文档

先删后增，没有的话算增

```
#全量修改，所有字段都改
PUT /test/_doc/文档id
{
    "字段1": "值1",
    "字段2": "值2",
    "字段3": {
        "子属性1": "值3",
        "子属性2": "值4"
    }
}
```

```
#增量修改，修改指定id
POST /test/_update/文档id
{
    "doc": {
        "字段1": "值1",
        "字段2": "值2"
    }
}
//注意是post还有doc！！！
```

数据库和es

```
#酒店的mapping
PUT /hotel
{
    "mappings": {
        "properties": {
            "id": {
                "type": "keyword"
            },
            "name": {
                "type": "text",
                "analyzer": "ik_max_word"
            },
            "copy_to": "all"
        },
        "address": {
            "type": "keyword",
            "index": false
        },
        "price": {
            "type": "integer"
        },
        "copy_to": "all"
    }
},
```

```
"score":{  
    "type": "integer"  
},  
"brand":{  
    "type": "keyword"  
},  
"city":{  
    "type": "keyword"  
},  
"starName":{  
    "type": "keyword"  
},  
"business":{  
    "type": "keyword"  
},  
"location":{  
    "type": "geo_point"  
},  
"pic":{  
    "type": "keyword",  
    "index": false  
},  
"all":{  
    "type": "text",  
    "analyzer": "ik_max_word"  
}  
}  
}  
}  
}
```

引入依赖

```
<properties>  
    <java.version>1.8</java.version>  
    <!--      统一版本-->  
    <elasticsearch.version>7.12.1</elasticsearch.version>  
</properties>  
<dependencies>  
    <!--      elasticsearch-->  
    <dependency>  
        <groupId>org.elasticsearch.client</groupId>  
        <artifactId>elasticsearch-rest-high-level-client</artifactId>  
        <version>7.12.1</version>  
    </dependency>
```

索引库操作基本步骤

初始化RestHighLevelClient

创建XxxIndexRequest, XXX是create, get, delete

准备dsl (create时需要)

发送请求, 调用RestHighLevelClient.indices.xxx () 方法

xxx是create, exists, delete

创建索引

```
private RestHighLevelClient client;
@Test
void test() {
    System.out.println(client);
}
@Test
void createHotelIndex() throws IOException {
    //创建request对象
    CreateIndexRequest request = new CreateIndexRequest("hotel");
    //准备请求的参数，DSL语句
    /*
    MAPPING_TEMPLATE是
    PUT /hotel
    {
        ...
    }中的
    {
        ...
    }
    */
    request.source(MAPPING_TEMPLATE, XContentType.JSON);
    //发送请求
    client.indices().create(request, RequestOptions.DEFAULT);
}
@BeforeEach
void setUp(){
    this.client=new RestHighLevelClient(RestClient.builder(
        HttpHost.create("175.178.236.116:9200")
    ));
}

@AfterEach
void tearDown() throws IOException {
    this.client.close();
}
```

删除索引

```
void testDelete() throws IOException {
    DeleteIndexRequest request=new DeleteIndexRequest("hotel");
    client.indices().delete(request, RequestOptions.DEFAULT);
}
```

判断索引是否存在

```
@Test
void testExist() throws IOException {
    GetIndexRequest request=new GetIndexRequest("hotel");
    System.out.println(
client.indices().exists(request, RequestOptions.DEFAULT));
}
```

文档操作基本步骤

初始化RestHighLevelClient

创建XxxRequest, xxx是index, get, update, delete

准备参数, index和update需要

发送请求, 调用RestHighLevelClient.xxx()方法 , xxx是index, get, update, delete

添加文档

```
@Autowired
private IHotelService hotelService;
@Test
void testAddDocument() throws IOException {
    //根据id查询酒店数据
    Hotel hotel = hotelService.getById(61083L);
    //转换为hotel文档类型
    HotelDoc hotelDoc = new HotelDoc(hotel);
    //准备request对象
    IndexRequest request= new
IndexRequest("hotel").id(hotel.getId().toString());
    //准备json文档,将对象序列化成json
    request.source(JSON.toJSONString(hotelDoc),XContentType.JSON);
    //发送请求
    client.index(request,RequestOptions.DEFAULT);
}
```

查询文档

```
@Test
void testGetDocument() throws IOException {
    //准备request
    GetRequest request=new GetRequest("hotel","61083");
    //发送请求得到响应
    GetResponse documentFields = client.get(request, RequestOptions.DEFAULT);
    //解析结果
    String json = documentFields.getSourceAsString();
    HotelDoc hotelDoc=JSON.parseObject(json,HotelDoc.class);
    System.out.println(hotelDoc);
}
```

修改文档

将添加文档的操作再次进行就是全量更新, 即删除旧文档添加新文档

局部更新, 更新部分字段

```

@Test
void testUpdateDocument() throws IOException {
    //准备request
    UpdateRequest request = new UpdateRequest("hotel", "61083");
    //准备请求参数
    request.doc("price", "952",
               "starName", "四钻"
    );
    //发送请求
    client.update(request, RequestOptions.DEFAULT);
}

```

删除文档

```

@Test
void testDeleteDocument() throws IOException {
    //准备request
    DeleteRequest request = new DeleteRequest("hotel", "61083");
    //发送请求
    client.delete(request, RequestOptions.DEFAULT);
}

```

批量处理

```

@Test
void testBulkRequest() throws IOException {
    //批量查询酒店数据
    List<Hotel> hotels=hotelservice.list();
    //创建request
    BulkRequest request = new BulkRequest();
    //准备参数，添加多个到新增的request
    for (Hotel hotel:hotels)
    {
        //转换为文档类型的request
        HotelDoc hotelDoc = new HotelDoc(hotel);
        request.add(new IndexRequest("hotel")
                   .id(hotelDoc.getId().toString())
                   .source(JSON.toJSONString(hotelDoc),XContentType.JSON));
    }
    //发送请求
    client.bulk(request, RequestOptions.DEFAULT);
}

```

GET /hotel/_search获取索引中所有文档

数据的搜索

常见的查询类型包括

查询所有: match_all

全文搜索: 利用分词器对用户输入内容分词, 然后去倒排索引库匹配,
match_query,multi_match_query

精确查询：根据精确词条查找数据，一般是查找keyword，数值，日期，boolean等类型字段，ids，range，term

地理查询：根据经纬度查询，例如geo_distance,geo_bounding_box

复合查询：上面的查询组合一起，bool，function_score

查询所有

```
GET /索引库名/_search
{
  "query": {
    "查询类型": {}
  }
}
GET /hotel/_search
{
  "query": {
    "match_all": {}
  }
}
```

全文搜索match

```
"name": {
  "type": "text",
  "analyzer": "ik_max_word"
  , "copy_to": "all"
},
"price": {
  "type": "integer"
  , "copy_to": "all"
},
"all": {
  "type": "text",
  "analyzer": "ik_max_word"
}
```

//前面创建索引库已经将name和price都关联到all

所以

```
GET /hotel/_search
{
  "query": {
    "match": {
      "all": "外滩如家"
    }
  }
}
```

针对all字段就是搜索name字段和price字段含有外滩如家的

全文搜索multi_match

```
GET /hotel/_search
{
  "query": {
    "multi_match": {
      "query": "外滩如家",
      "fields": ["brand", "name"]
    }
  }
}
```

针对多个字段有外滩如家查询

match和multi_match区别

match根据单字段查询

multi_match根据多字段查询，参与查询字段越多，性能越差

精确查询term

根据词条精确值查询

```
GET /hotel/_search
{
  "query": {
    "term": {
      "city": {
        "value": "上海"
      }
    }
  }
}
```

精确查询range

根据值的范围查询

```
GET /hotel/_search
{
  "query": {
    "range": {
      "price": {
        "gte": 100,
        "lte": 300
      }
    }
  }
}
gt是大于
gte是大于等于
lt是小于
lte是小于等于
```

地理查询geo_bounding_box

根据其geo_point值落在某个矩形范围的所有文档

地理查询geo_distance

根据其geo_point值落在某个圆形范围的所有文档

```
GET /hotel/_search
{
  "query": {
    "geo_distance": {
      "distance": "15km",
      "location": "31.21,121.5"
    }
  }
}
```

location是字段，经纬度之间有逗号

复合查询

可以实现其简单查询组合起来，实现更复杂的搜索逻辑，例如

function score：算分函数查询，可以控制文档相关性算分，控制文档排名，例如百度竞价

score计算！！！

tf算法

tf-idf算法

tf (term frequency)：查询的文本中的词条在document中出现了多少次，出现次数越多，相关度越高

idf (inverse document frequency)：查询的文本中的词条在索引的文档中出现了多少次，出现的次数越多，相关度越少

有很多不同的数学公式可以用来计算TF-IDF。这边的例子以上述的数学公式数学公式？

fromModule=lemma_inlink)来计算。词频 (TF) 是一词语出现的次数除以该文件的总词语数。假如一篇文件的总词语数是100个，而词语“母牛”出现了3次，那么“母牛”一词在该文件中的词频就是 $3/100=0.03$ 。一个计算文件频率 (IDF) 的方法是文件集里包含的文件总数除以测定有多少份文件出现过“母牛”一词。所以，如果“母牛”一词在1,000份文件出现过，而文件总数是10,000,000份的话，其逆向文件频率就是 $\lg(10,000,000 / 1,000)=4$ 。最后的TF-IDF的分数为 $0.03 * 4=0.12$ 。

现在用bm25算法

BM25是基于TF-IDF改进的文本匹配算法 (BM是Best Matching的意思)，被称为下一代的TF-IDF，BM25在传统TF-IDF的基础上增加了几个可调节的参数，使得它在应用上更佳灵活和强大，具有较高的实用性。

<https://www.jianshu.com/p/344bd4dfcb5a>

<https://www.modb.pro/db/330342>

function score query

修改算分

```
GET /hotel/_search
{
  "query": {
    "function_score": {
      "query": {
        "match": {"all": "外滩"},      //原始查询条件，搜索文档根据相关性打分
      },
      "functions": [
        {
          "filter": {"term": {"id": "1"}},   //过滤条件，符合条件的文档才会重新
          "weight": 10           //算分函数，算法函数的结果称为function score，将来
          算法
          //会与query score运算，得到新算分，常见算分函数有weight给一个常量值作为函数结果，
          field_value_factor用文档中某个字段作为函数结果，random_score随机生成一个值作为函数结果，
          script_score自定义计算公式，公式结果作为函数结果
        }
      ],
      "boost_mode": "multiply"     //加权模式，定义function score与query score
      的运算方式，包括multiply两者相乘，默认就是这个，replace用function score替换query score，
      其它sum, avg, max, min
    }
  }
}
```

复合查询boolean query

布尔查询是一个或多个查询子句的组合，子查询的组合方式有

must: 必须匹配每一个子查询，与

should: 选择性匹配

must_not: 必须不匹配，不参与算分

filter: 必须匹配，不参与算分

```
GET /hotel/_search
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "name": "如家"
          }
        }
      ]
    },
    "must_not": [
      {
        "range": {
          "price": {
            "gt": 400
          }
        }
      }
    ]
  }
}
```

```
        }
    }
]
, "filter": [
{
    "geo_distance": {
        "distance": "10km",
        "location": {
            "lat": 40.73,
            "lon": -74.1
        }
    }
}
]
}
}
```

搜索结果处理

升序

```
GET /hotel/_search
{
    "query": {
        "term": {
            "city": {
                "value": "上海"
            }
        }
    }
, "sort": [
{
    "price": {
        "order": "asc"
    }
}
]
```

也可以这样优雅一点

```
"sort": [
{
    "score": "desc"
}]
```

```
GET /hotel/_search
{
    "query": {
        "term": {
            "city": {
                "value": "上海"
            }
        }
    }
}
```

```
, "sort": [
  {
    "price": {
      "order": "asc"
    }
  , "_geo_distance": {
      "location": {
        "lat": 31.034661,
        "lon": 121.612282
      },
      "order": "asc"
      , "unit": "km"
    }
  }
]
}
```

分页

数据结构先获取前from+size条数据再截取，如果es做了集群，数据会拆分到不同的服务器，若是想找前1000，需要每个服务器找前1000汇总后再找前1000。

默认from+size不能超过一万

关于分页的详细知识需要再了解！！！！！！！

默认返回top10

from是分页开始的位置，默认为0

size是期望获取的文档总数

```
GET /hotel/_search
{
  "query": {
    "match_all": {}
  }
, "sort": [
  {
    "price": {
      "order": "asc"
    }
  }
]
, "from": 10
, "size": 10
}
```

高亮

将关键字用标签标记出来，在页面中给标签添加css样式

```
GET /hotel/_search
{
  "query": {
    "match": {
```

```

        "brand": "如家"
    }
}
, "highlight": {
"fields": {
"price": {
"require_field_match": "false",
"pre_tags": "<em>",
"post_tags": "/<em>"
}
}
}
}

默认就是用<em></em>可以不写
注意需要查询字段和高亮字段一致查询字段才高亮，即
"highlight" :
{
    "brand" : [
        "<em>如家/<em>"
    ]
}

```

java实现查询

创建SearchRequest对象

准备request.source()也就是DSL

使用查询request.source().query也就是DSL的查询，查询条件封装在QueryBuilders.matchAllQuery()

发送请求得到结果

对结果进行解析

matchAll

```

json为
{
    "took" : 0,
    "timed_out" : false,
    "_shards" : {
        "total" : 1,
        "successful" : 1,
        "skipped" : 0,
        "failed" : 0
    },
    "hits" : {
        "total" : {
            "value" : 50,
            "relation" : "eq"
        },
        "max_score" : 1.0,
        "hits" : [
            {
                "_index" : "hotel",
                "_type" : "_doc",
                "_id" : "38609",
                "_score" : 1.0
            }
        ]
    }
}

```

```
"_score" : 1.  
.....
```

```
@Test  
void MatchAll() throws IOException {  
    //发送请求到哪一个索引  
    SearchRequest request = new SearchRequest("hotel");  
    //请求的source即是dsl，然后调用其查询动作，传入查询条件  
    QueryBuilders.matchAllQuery()  
        request.source().query(QueryBuilders.matchAllQuery());  
    //发送请求  
    SearchResponse search = client.search(request, RequestOptions.DEFAULT);  
    //查询命中对象  
    SearchHits searchHits = search.getHits();  
    //获取总条数  
    long value = searchHits.getTotalHits().value;  
    System.out.println(value);  
    //获取结果数组  
    SearchHit[] hits = searchHits.getHits();  
    for (SearchHit hit:hits)  
    {  
        HotelDoc hotelDoc = JSON.parseObject(hit.getSourceAsString(),  
HotelDoc.class);  
        System.out.println(hotelDoc);  
    }  
}
```

其余查询

match

```
request.source().query(QueryBuilders.matchQuery("field","value"));
```

multi_match

```
request.source().query(QueryBuilders.multiMatchQuery("value","field1","field2"))  
;
```

term

```
request.source().query(QueryBuilders.termQuery("field","value"));
```

range

```
QueryBuilders.rangeQuery("field").gte(50)
```

booleanQuery

```
//创建布尔查询
BoolQueryBuilder boolQuery = QueryBuilders.boolQuery();
//添加must条件
boolQuery.must(QueryBuilders.termQuery("field","value"));
//查询
request.source().query(boolQuery);
```

排序和分页

```
GET /hotel/_search
{
  "query": {
    "match_all": {}
  }
  , "sort": [
    {
      "price": {
        "order": "asc"
      }
    }
  ]
  , "from": 10
  , "size": 10
}
```

分析以上代码query可知sort和from size与其同级

则调用排序和分页是用request.source().xxxxx

```
request.source().from(0).size(5);
request.source().sort("field", SortOrder.ASC);
request.source().sort(
  SortBuilders.
    geoDistanceSort("locaton",
      new
GeoPoint(location)).order(SortOrder.ASC).unit(DistanceUnit.KILOMETERS));
```

高亮

```
{
  "_index" : "hotel",
  "_type" : "_doc",
  "_id" : "1455383931",
  "_score" : 1.9649367,
  "_source" : {
    "address" : "西乡河西金雅新苑34栋",
    "brand" : "如家",
    "business" : "宝安商业区",
    "city" : "深圳",
    "id" : 1455383931,
    "location" : "22.590272, 113.881933",
    "name" : "如家酒店(深圳宝安客运中心站店)",
    "pic" :
https://m.tuniucdn.com/fb3/s1/2n9c/2w9cbbpzjjsyd2wRhFrnUpBMT8b4\_w200\_h200\_c1\_t0.jpg,
  }
```

```

    "price" : 169,
    "score" : 45,
    "starName" : "二钻"
},
"highlight" : {
    "name" : [
        "<em>如家</em>酒店(深圳宝安客运中心站店)"
    ]
}
}

```

高亮的结果处理相对比较麻烦，因为source不会高亮，是有

```

"highlight" : {
    "name" : [
        "<em>如家</em>酒店(北京良乡西路店)"
    ]
}

```

```

request.source().query(QueryBuilders.matchQuery("all", "如家"));
request.source().highlighter(new
HighlightBuilder().field("name").requireFieldMatch(false));
查询和高亮不一致要加requireFieldMatch(false)

```

```

HotelDoc hotelDoc = JSON.parseObject(hit.getSourceAsString(), HotelDoc.class);
System.out.println(hotelDoc);
//处理高亮
Map<String, HighlightField> highlightFields = hit.getHighlightFields();
//获取高亮里的key-value
HighlightField highlightField = highlightFields.get("name");
System.out.println("不要碎片化"+highlightFields);
//获取碎片fragment中的值
System.err.println("高亮"+highlightField.getFragments()[0].string());

```

结果为：

不要碎片化{name=[name], fragments[[如家酒店(北京良乡西路店)]]}
高亮如家酒店(北京上地安宁庄东路店)

functionScore查询

Function Score查询可以控制文档的相关性算分，使用方式如下：

```

// 7.function score
FunctionScoreQueryBuilder functionScoreQueryBuilder =
    QueryBuilders.functionScoreQuery(
        QueryBuilders.matchQuery("name", "外滩"),
        new FunctionScoreQueryBuilder.FilterFunctionBuilder[]{
            new FunctionScoreQueryBuilder.FilterFunctionBuilder(
                QueryBuilders.termQuery("brand", "如家"),
                ScoreFunctionBuilders.weightFactorFunction(5)
            )
        }
    );
sourceBuilder.query(functionScoreQueryBuilder);

```

```

GET /hotel/_search
{
    "query": {
        "function_score": {
            "query": {
                "match": {
                    "name": "外滩"
                }
            },
            "functions": [
                {
                    "filter": {
                        "term": {
                            "brand": "如家"
                        }
                    },
                    "weight": 5
                }
            ]
        }
    }
}

```

```

//复合查询boolQuery
//算分控制
FunctionScoreQueryBuilder isAD = QueryBuilders.functionScoreQuery(
    boolQuery,
    new FunctionScoreQueryBuilder.FilterFunctionBuilder[]{
        new FunctionScoreQueryBuilder.FilterFunctionBuilder(
            (
                QueryBuilders.termQuery("isAD", true),
                ScoreFunctionBuilders.weightFactorFunction(10)
            )
        );
    }
);
//查询
request.source().query(isAD);

```

聚合aggregation

什么是聚合?

聚合是对文档数据的统计，分析，计算

聚合的常见种类有哪些?

bucket: 对文档数据分组，并统计每组数量

metric: 对文档数据做计算，例如avg

pipeline: 基于其他聚合结果再做聚合

参与聚合的字段类型必须是：

keyword，数值，日期，布尔

bucket

```

GET /hotel/_search
{
  "size": 0, //设置size为0，结果不包含文档，只包含聚合结果
  "aggs": {
    "brandAgg": {
      "terms": {
        "field": "brand", //参与聚合的字段
        "order": {
          "_count": "asc" //按照_count升序
        },
        "size": 10 //希望获取聚合的结果数量
      }
    }
  }
}

```

默认bucket聚合会统计bucket文档数量，记为_count，并且降序排序

```
{
  "took" : 4,
```

```

"timed_out" : false,
"_shards" : {
  "total" : 1,
  "successful" : 1,
  "skipped" : 0,
  "failed" : 0
},
"hits" : {
  "total" : {
    "value" : 201,
    "relation" : "eq"
  },
  "max_score" : null,
  "hits" : [ ]
},
"aggregations" : {
  "brandAgg" : {
    "doc_count_error_upper_bound" : 0,
    "sum_other_doc_count" : 39,
    "buckets" : [
      {
        "key" : "7天酒店",
        "doc_count" : 30
      },
      {
        "key" : "如家",
        "doc_count" : 30
      },
      .....
    ]
  }
}
}

```

默认情况下bucket是会对索引库的所有文档做聚合，我们也可以限定聚合范围，添加query条件

```

GET /hotel/_search
{
  "query": {},

```

metric

```

GET /hotel/_search
{
  "size": 0,
  "aggs": {
    "brandAgg": {
      "terms": {
        "field": "brand",
        "order": {
          "_count": "asc"
        },
        "size": 10
      }
    }, "aggs": { //是brands聚合的子聚合，对分组后每组分别进行计算

```

```
"score_stats": { //聚合名称
    "max": { //可以计算max, min, avg, 所有都罗列的话是要用stats
        "field": "score" //聚合字段
    }
}
}
}
}
}

如果要根据品牌的分数score进行排序
则需要
GET /hotel/_search
{
    "size": 0,
    "aggs": {
        "brandAgg": {
            "terms": {
                "field": "brand",
                "order": {
                    "score_stats.value": "asc" //针对score_stats中的value进行排序,
                }
            },
            "size": 10
        }
    , "aggs": {
        "score_stats": {
            "avg": {
                "field": "score"
            }
        }
    }
}
}
```

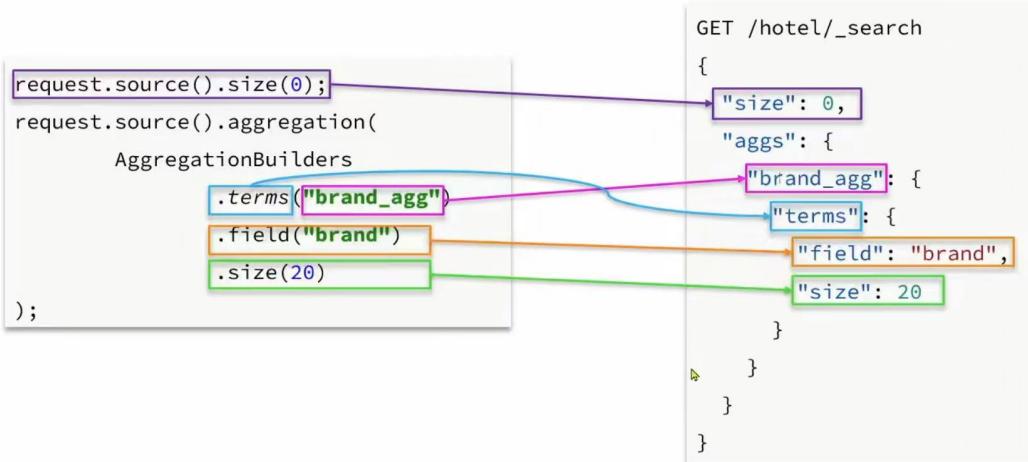
桶里的结果

```
"buckets" : [
    {
        "key" : "万丽",
        "doc_count" : 2,
        "score_stats" : {
            "value" : 47.0
        }
    }
]
```

java代码

RestAPI实现聚合

我们以品牌聚合为例，演示下Java的RestClient使用，先看请求组装：



```

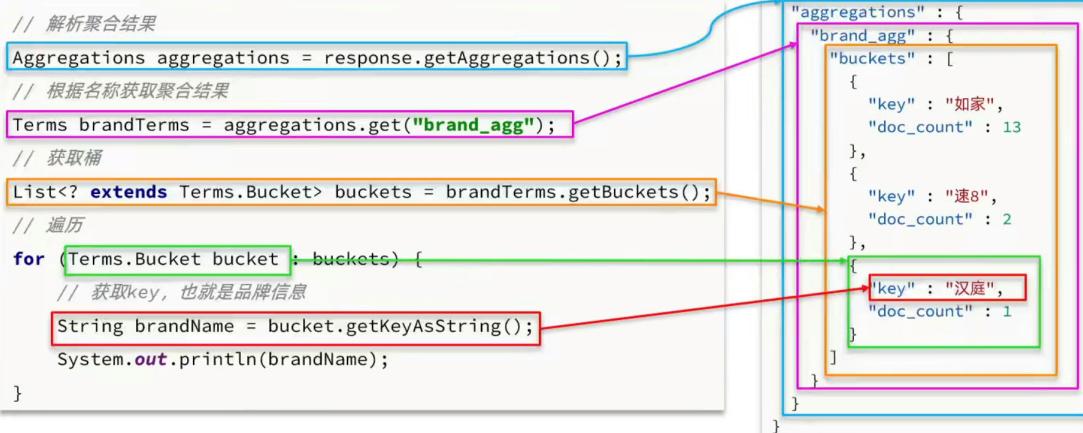
//准备request
SearchRequest request = new SearchRequest("hotel");
//准备ds1

request.source().aggregation(AggregationBuilders.terms("brand_agg").field("brand")
    ".size(20));
//发出请求
SearchResponse response = client.search(request,
RequestOptions.DEFAULT);

```

RestAPI实现聚合

再看下聚合结果解析



```

//结果解析
//获取聚合
Aggregations aggregations = response.getAggregations();
//根据名称获取聚合结果
Terms brand_agg = aggregations.get("brand_agg");
//获取桶
List<? extends Terms.Bucket> buckets = brand_agg.getBuckets();
//遍历

```

```

for (Terms.Bucket bucket:buckets)
{
    //获取key
    String keyAsString = bucket.getKeyAsString();
    System.out.println(keyAsString);
}

```

拼音分词器

先elastic先下载拼音分词器，然后重启服务器

基本用法

```

POST /_analyze
{
  "text": ["如家酒店还不错"]
  , "analyzer": "pinyin"
}

```

analyzer三部分

自定义分词器

elasticsearch中分词器 (analyzer) 的组成包含三部分：

- character filters: 在tokenizer之前对文本进行处理。例如删除字符、替换字符
- tokenizer: 将文本按照一定的规则切割成词条 (term) 。例如keyword，就是不分词；还有ik_smart
- tokenizer filter: 将tokenizer输出的词条做进一步处理。例如大小写转换、同义词处理、拼音处理等



自定义分词器

创建索引库时候自定义分词器

```

// 自定义拼音分词器
PUT /test
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": {      //自定义分词器名字
          "tokenizer": "ik_max_word",      //分割
          "filter": "py"                  //进一步处理
        }
      },
      //但是该拼音分词器不好，每个字去拼英，以及取句子的首字母，所以要做一定修改
      "filter": {

```

```

    "py": {
        "type": "pinyin",
        "keep_full_pinyin": false,
        "keep_joined_full_pinyin": true,
        "keep_original": true,
        "limit_first_letter_length": 16,
        "remove_duplicated_term": true,
        "none_chinese_pinyin_tokenize": false
    }
}
},
{
"mappings": {
    "properties": {
        "name": {
            "type": "text",
            "analyzer": "my_analyzer",           //创建时用的分词器
            "search_analyzer": "ik_smart" //搜索时用的分词器
        }
    }
}
}

//拼音分词器适合在创建倒排索引时候使用，不能在搜索时候使用
//用两个分词器是为了防止搜索到同音词，搜索时不需要拼音分词器

```

自动补全

- 参与补全查询的字段必须是completion类型。
- 字段的内容一般是由多个词条形成的数组。

```

// 创建索引库
PUT test
{
    "mappings": {
        "properties": {
            "title": {
                "type": "completion"
            }
        }
    }
}

```

插入

```

// 示例数据
POST test/_doc
{
    "title": ["Sony", "WH-1000XM3"]
}
POST test/_doc
{
    "title": ["SK-II", "PITERA"]
}

```

查询

```
// 自动补全查询
GET /test/_search
{
  "suggest": {
    "title_suggest": { //查询所起的名字
      "text": "s", // 关键字
      "completion": {
        "field": "title", // 补全查询的字段
        "skip_duplicates": true, // 跳过重复的
        "size": 10 // 获取前10条结果
      }
    }
  }
}
```

修改测试数据结构

```
#查看酒店数据结构
GET /hotel/_mapping
```

```
// 酒店数据索引库
PUT /hotel
{
  "settings": {
    "analysis": {
      "analyzer": {
        "text_analyzer": {
          "tokenizer": "ik_max_word",
          "filter": "py"
        },
        "completion_analyzer": {
          "tokenizer": "keyword",
          "filter": "py"
        }
      },
      "filter": {
        "py": {
          "type": "pinyin",
          "keep_full_pinyin": false,
          "keep_joined_full_pinyin": true,
          "keep_original": true,
          "limit_first_letter_length": 16,
          "remove_duplicated_term": true,
          "none_chinese_pinyin_tokenize": false
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "id": {
        "type": "keyword"
      },
      "name": {
        "type": "string"
      }
    }
  }
}
```

```
"name":{  
    "type": "text",  
    "analyzer": "text_analyzer",  
    "search_analyzer": "ik_smart",  
    "copy_to": "all"  
},  
"address":{  
    "type": "keyword",  
    "index": false  
},  
"price":{  
    "type": "integer"  
},  
"score":{  
    "type": "integer"  
},  
"brand":{  
    "type": "keyword",  
    "copy_to": "all"  
},  
"city":{  
    "type": "keyword"  
},  
"starName":{  
    "type": "keyword"  
},  
"business":{  
    "type": "keyword",  
    "copy_to": "all"  
},  
"location":{  
    "type": "geo_point"  
},  
"pic":{  
    "type": "keyword",  
    "index": false  
},  
"all":{  
    "type": "text",  
    "analyzer": "text_analyzer",  
    "search_analyzer": "ik_smart"  
},  
"suggestion":{  
    "type": "completion",  
    "analyzer": "completion_analyzer"  
}  
}  
}  
}
```

代码

先看请求参数构造的API:

```
// 1. 准备请求
SearchRequest request = new SearchRequest("hotel");
// 2. 请求参数
request.source()
    .suggest(new SuggestBuilder().addSuggestion(
        "mySuggestion",
        SuggestBuilders
            .completionSuggestion("title")
            .prefix("h")
            .skipDuplicates(true)
            .size(10)
    ));
// 3. 发送请求
client.search(request, RequestOptions.DEFAULT);
```

```
// 自动补全查询
GET /test/_search
{
    "suggest": {
        "mySuggestion": {
            "text": "h", // 关键字
            "completion": {
                "field": "title", // 补全字段
                "skip_duplicates": true,
                "size": 10 // 获取前10条结果
            }
        }
    }
}
```

```
SearchRequest request=new SearchRequest("hotel");
request.source().suggest(
    new SuggestBuilder().addSuggestion("suggestions",

SuggestBuilders.completionSuggestion("suggestion").prefix("h").skipDuplicates(true).size(10))
);
SearchResponse response = client.search(request,
RequestOptions.DEFAULT);
System.out.println(response);
```

结果解析

RestAPI实现自动补全

再来看结果解析:

```
// 4. 处理结果
Suggest suggest = response.getSuggest();
// 4.1. 根据名称获取补全结果
CompletionSuggestion suggestion = suggest.getSuggestion("hotelSuggestion");
// 4.2. 获取options并遍历
for (CompletionSuggestion.Entry.Option option : suggestion.getOptions()) {
    // 4.3. 获取一个option中的text, 也就是补全的词条
    String text = option.getText().string();
    System.out.println(text);
}
```

```
{
    "took" : 1,
    "timed_out" : false,
    "_shards" : {...},
    "hits" : [...],
    "suggest" : {
        "title_suggest" : [
            {
                "text" : "s",
                "offset" : 0,
                "length" : 1,
                "options" : [
                    {
                        "text" : "SK-II",
                        ...
                    },
                    {
                        "text" : "Sony",
                        ...
                    },
                    {
                        "text" : "switch",
                        ...
                    }
                ]
            }
        ]
    }
}
```

```
SearchResponse response = client.search(request, RequestOptions.DEFAULT);
//解析结果
Suggest suggest = response.getSuggest();
//获取补全结果
CompletionSuggestion suggestions = suggest.getSuggestion("suggestions");
//获取options
List<CompletionSuggestion.Entry.Option> options = suggestions.getOptions();
for (CompletionSuggestion.Entry.Option option:options)
{
    //遍历
    String s = option.getText().toString();
    System.out.println(s);
}
```

数据同步

同步调用

写入数据库时候再调用elastic更新的接口。数据耦合，由于多了elastic性能下降。

异步通知

写入数据库后发布消息到消息队列给elastic监听消息处理，缺点是依赖mq可靠性

监听binlog

完全解除服务间耦合，但开启binlog增加数据库负担，实现复杂度高

elasticSearch集群问题

数据存储问题：将海量数据拆分成n份存储到多个节点

单点故障问题：将分片数据再不同节点备份

利用docker-compose搭建集群

然后创建索引库时候做分片以及备份

```
put /itcast
{
    "settings": {
        "number_of_shards": 3           //分片数量
        "number_of_replicas": 1         //副本数量
    }
}
```

雪崩问题

服务故障可能是因为大量访问，或者依赖于其他服务，请求一直阻塞，会导致服务器资源耗尽。其他请求进不来。

微服务调用链路中的某个服务故障，引起整个链路中的所有微服务不可用，就叫雪崩

解决方案

超时处理：设定超时时间，超过一定时间返回错误信息，不会无休止等待。

缺点：没有从根本解决问题，若是请求速度大于返回速度，最终也会服务器资源耗尽崩溃。

船舱模式：限定每一个业务能使用的线程数，避免耗尽整个tomcat资源，也叫线程隔离。

缺点：即使该业务无法解决问题，请求也会进来该业务。浪费资源

熔断降级：由断路器统计业务执行异常比例，超出阈值熔断该业务，拦截访问该业务的一切请求。常使用该方案

以上解决方案是避免因服务故障引起雪崩问题，是发生问题后解决问题

以下是直接让问题不会发生

流量控制：限制业务访问的QPS（Query Per Second，每秒处理请求数），避免服务因流量的突增而故障

流量控制中间件sentinel

下载

下载GitHub下载sentinel的jar包，默认是8080端口，账号密码是sentinel

配置项	默认值	说明
server.port	8080	服务端口
sentinel.dashboard.auth.username	sentinel	默认用户名
sentinel.dashboard.auth.password	sentinel	默认密码

举例说明

```
java -jar sentinel-dashboard-1.8.1.jar --server.port=8090
```

依赖

```
<!-- sentinel-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

配置

```
spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:8090
```

The screenshot shows the Sentinel Control Panel version 1.8.1. On the left, there's a sidebar with navigation items like '实时监控' (Real-time Monitoring), '流控规则' (Flow Control Rules), '热点规则' (Hotspot Rules), '系统规则' (System Rules), '授权规则' (Authorization Rules), '集群流控' (Cluster Flow Control), and '机器列表' (Machine List). The '簇点链路' (Clustered Path) item is highlighted with a red box. The main area shows the 'userservice' configuration with a tree view of paths: sentinel_default_context, sentinel_spring_web_context, /user, /*, and /error. A modal window titled '新增流控规则' (Add Flow Control Rule) is open, showing fields for '资源名' (/user), '针对来源' (default), '阈值类型' (QPS, selected), '单机阈值' (5), and '是否集群' (unchecked). In the background, there's a list of flow control rules with columns for '分钟拒绝' (Minute Rejection), '操作' (Operation), and a red box highlighting a specific rule.

簇点链路

簇点链路就是项目的调用链路，control---servcie---mapper就是一条链路，而在链路中被监控的每个接口就是一个资源。

默认情况下sentinel会监控控制层的@RequestMapping("name")

注意

要先服务器被访问，sentinel才会获取该服务器

每秒请求数量，多了的话会拦截并报错，fastjson版本不能太低，否则无法正常使用

流控模式高级选项

The screenshot shows the 'Edit Flow Control Rule' dialog. It contains the following fields:

- 资源名:** /user
- 针对来源:** default
- 阈值类型:** QPS (radio button selected)
- 单机阈值:** 5
- 是否集群:**
- 流控模式:** 直接 (radio button selected)
- 流控效果:** 快速失败 (radio button selected)

At the bottom, there are '保存' (Save) and '取消' (Cancel) buttons.

流控模式：

关联：其他资源，当其他资源触发阈值时候，对当前资源限流，给谁限流就给谁做关联。

链路：对指定链路访问到本资源的请求做统计，判断是否超过阈值，

关联场景应用两个有竞争关系的资源，一个优先级高一个优先级低，对优先级低做限流。例如在查询和修改争抢数据库锁，产生竞争关系，修改业务触发阈值时候，对查询业务做限流。

service方法资源被监控

修改配置

原先是

▼ sentinel_spring_web_context	0
/two	0
▼ /one	0
nextTest	0
sentinel_default_context	0

```
sentinel:  
  transport:  
    dashboard: localhost:8090  
  web-context-unify: false #sentinel默认将controller方法做context整合，要关闭context整合  
  其目的是使得不要在同一个链路，不要流控在一起
```

现在是不要在sentinel_default_context下，则不流控在一起

▼ /one	0	0
▼ /one	0	0
nextTest	0	0
▼ /two	0	0
/two	0	0
sentinel_default_context	0	0

然后service层

```
@SentinelResource("nextTest")  
public String two() {  
    return "test";  
}
```

▼ /order/save	0	0	0	0
▼ /order/save	0	0	0	0
goods	0	0	0	0
▼ /order/query	0	0	0	0
▼ /order/query	0	0	0	0
goods	0	0	0	0

用法是链路：对指定链路访问到本资源的请求做统计，判断是否超过阈值。同一个资源，不同的链路

新增流控规则

资源名	goods		
针对来源	default		
阈值类型	<input checked="" type="radio"/> QPS <input type="radio"/> 线程数	单机阈值	单机阈值
是否集群	<input type="checkbox"/>		
流控模式	<input type="radio"/> 直接 <input type="radio"/> 关联 <input checked="" type="radio"/> 链路		
入口资源	/order/save		
流控效果	<input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待		

[关闭高级选项](#)

流控效果

warm up也叫预热模式，是应对服务冷启动但是一种方案，请求阈值初始值为threshold/coldfactor，持续指定时长后逐渐提高到threshold，而coldfactor的默认值是3

例如qps的threshold为10，预热时间为5秒，初始阈值就是 $10/3$ 也就是3，5秒后逐渐增长到10

排队等待：所有请求进入一个队列中，按照阈值允许的时间间隔一次执行，后来的请求必须等前面执行成功完成，如果请求预期的等待时间超过最大时长则会被拒绝。例如QPS=5,意味着每200ms处理一个队列中的请求，timeout=2000，意味着预期等待超过2000ms的请求会被拒绝

热点参数限制

之前的限流是统计访问某个资源的所有请求，则热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值

```

@RequestMapping ("order")
public class OrderController{
    @Autowired
    private OrderService orderService;
    @SentinelResource ("hot")
    @GetMapping ("{orderId}")
    public Order queryOrderById(@PathVariable("orderId") Long orderId) {
        .....
    }
}

```



是对参数值@GetMapping ("{orderId}")的rest风格参数orderId进行热点参数限制

隔离和降级

虽然限流可以尽量避免因高并发而引起的服务故障，但服务还会因其他原因而故障。所以我们不能只做流量控制，还要做线程隔离（舱壁模式）和熔断降级手段，这两种方式都是生产者去调用消费者，生产者是通过feign实现的，所以要整合feign和sentinel

feign整合sentinel步骤

目的是开启feign的sentinel功能

```

feign:
  sentinel:
    enabled: true

```

在feign-api中写请求失败后的降级逻辑

```
package com.clients;

import com.fallback.UserClientFallbackFactory;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;

@FeignClient(value = "userservice" ,fallbackFactory =
UserClientFallbackFactory.class) //服务名称
public interface UserClient {
    //请求
    @GetMapping("/user")
    String findById();
}
```

```
package com.fallback;

import com.clients.UserClient;
import feign.hystrix.FallbackFactory;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class UserClientFallbackFactory implements FallbackFactory<UserClient> {

    @Override
    public UserClient create(Throwable throwable) {
        return new UserClient() {
            @Override
            public String findById() {
                //降级的业务逻辑
                log.error("查询用户异常",throwable);
                return null;
            }
        };
    }
}
```

```
public class FeignConfiguration {

    @Bean
    public Logger.Level loggerLevel() {
        return Logger.Level.BASIC;
    }
    @Bean
    public UserClientFallbackFactory userClientFallbackFactory()
    {
        return new UserClientFallbackFactory();
    }
}
```

注意在配置类生成bean，不需要@Configuration，在消费者中用注解

```
@EnableFeignClients(clients = UserClient.class,defaultConfiguration =
FeignConfiguration.class)
```

报错

将父工程pom.xml修改<spring-cloud.version>Hoxton.SR8</spring-cloud.version>

线程隔离

线程池隔离和

信号量隔离 (sentinel默认采用) ,基于计数器实现

	优点	缺点	场景
线程池隔离	支持主动超时, 异步调用	线程额外开销大	低扇出
信号量隔离	轻量级无额外开销	不支持主动调用, 异步调用	高频调用, 高扇出



利用fallbackFactory可以捕获异常信息并返回默认降级结果

熔断

超过阈值熔断服务拦截一切请求，服务回复，断路器放行访问该服务的请求

The screenshot shows the configuration interface for adding a circuit breaker rule. The 'Breaker Strategy' field is highlighted with a red box, showing 'Slow Ratio' selected. Other fields like 'Resource Name' and 'Maximum RT' are also visible. A list of existing rules is shown on the right, with one specific rule highlighted by a red box.

慢调用在统计时长内，请求数超过最小请求数后，请求时间超过最大RT的比例超过比例阈值，就熔断一定时间，时间为熔断时长

还有异常比例和异常数，是个人都看到就会用。

授权和持久化不学了

事务协调者

解决分布式事务，各个子系统之间必须能感知到彼此的事务状态，才能保证状态，因此需要一个事务协调者来协调每一个事务的参与者。

解决分布式事务

全局事务：整个分布式事务

分支事务：分布式事务包含的每一个子系统的事务

最终一致思想：各分支事务分别执行并提交，如果有不一致的情况，再想办法恢复数据

强一致思想：各分支事务执行完业务不要提交，等待彼此结果。而后统一提交或回滚

CAP

指的是在一个分布式系统中，Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性），最多只能同时三个特性中的两个，三者不可兼得（每个服务实例发生异常是无法避免的）。

一致性：数据复制的时候，按照强一致性的方式进行数据复制。保证了在读操作总是能够读取到之前写入的数据，无论从那个主数据或者副本数据。

可用性：数据写入成功后，正在进行数据复制时，任何一个副本节点发生异常也不会影响此次写入操作。可以理解为，此时数据的复制采用的是弱一致性，数据的读写操作在单台集器发生故障的情况下仍然可以正常执行。

分区容错性：在服务实例发生异常时，分布式系统任然能够满足一致性和可用性。

BASE

BASE是 Basically Available（基本可用）、soft state（软状态）和 Eventually consistent（最终一致性）三个短语的缩写

基本可用：是指分布式系统在出现不可预知故障的时候，允许损失部分可用性。

软状态：允许部分节点的数据存在一定的延时，这个延时不影响可用性。

最终一致性：软状态允许有一定延时，所以这个最终一致含义就是在一定的延时过去之后，所有节点的数据必须保持一致。

Seata

网址

<https://seata.io/zh-cn/>

角色

seata事务管理中有三个重要角色

TC transactionCoordinator事务协调者，维护全局和分支事务的状态，协调全局事务提交或回滚

TM transactionManager事务管理器，定义全局事务的范围，开始全局事务，提交或回滚全局事务

RM resourceManager资源管理器，管理分支事务处理的资源，与TC交谈以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

注册配置

\seata-server-1.4.2\conf\registry.conf

对注册中心的模板进行修改

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

    nacos {
        application = "seata-tc-server" #服务名字
        serverAddr = "127.0.0.1:8848" #nacos端口
        group = "DEFAULT_GROUP" #NACOS的组
        namespace = "" #NACOS的命名空间
        cluster = "default" #NACOS的集群
        username = "nacos" #nacos注册中心nacos账号
        password = "nacos" #nacos注册中心nacos密码
    }
}

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

    nacos {
        serverAddr = "127.0.0.1:8848"
        namespace = ""
        group = "SEATA_GROUP"
        username = "nacos"
        password = "nacos"
        dataId = "seataServer.properties" #nacos配置列表的配置文件
    }
}
```

模板

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

    nacos {
        application = "seata-server"
        serverAddr = "127.0.0.1:8848"
        group = "SEATA_GROUP"
        namespace = ""
        cluster = "default"
        username = ""
    }
}
```

```
password = ""
}

eureka {
    serviceUrl = "http://localhost:8761/eureka"
    application = "default"
    weight = "1"
}

redis {
    serverAddr = "localhost:6379"
    db = 0
    password = ""
    cluster = "default"
    timeout = 0
}

zk {
    cluster = "default"
    serverAddr = "127.0.0.1:2181"
    sessionTimeout = 6000
    connectTimeout = 2000
    username = ""
    password = ""
}

consul {
    cluster = "default"
    serverAddr = "127.0.0.1:8500"
    aclToken = ""
}

etcd3 {
    cluster = "default"
    serverAddr = "http://localhost:2379"
}

sofa {
    serverAddr = "127.0.0.1:9603"
    application = "default"
    region = "DEFAULT_ZONE"
    datacenter = "DefaultDataCenter"
    cluster = "default"
    group = "SEATA_GROUP"
    addressWaitTime = "3000"
}

file {
    name = "file.conf"
}

}

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

    nacos {
        serverAddr = "127.0.0.1:8848"
        namespace = ""
        group = "SEATA_GROUP"
        username = ""
        password = ""
        dataId = "seataServer.properties"
    }

    consul {
```

```

serverAddr = "127.0.0.1:8500"
aclToken = ""
}
apollo {
    appId = "seata-server"
    ## apolloConfigService will cover apolloMeta
    apolloMeta = "http://192.168.1.204:8801"
    apolloConfigService = "http://192.168.1.204:8080"
    namespace = "application"
    apolloAccesskeySecret = ""
    cluster = "seata"
}
zk {
    serverAddr = "127.0.0.1:2181"
    sessionTimeout = 6000
    connectTimeout = 2000
    username = ""
    password = ""
    nodePath = "/seata/seata.properties"
}
etcd3 {
    serverAddr = "http://localhost:2379"
}
file {
    name = "file.conf"
}
}

```

在nacos添加配置

修改下数据库

```

# 数据存储方式, db代表数据库
store.mode=db
store.db.datasource=druid
store.db.dbType=mysql
store.db.driverClassName=com.mysql.jdbc.Driver
store.db.url=jdbc:mysql://127.0.0.1:3306/seata?
useUnicode=true&rewriteBatchedStatements=true
store.db.user=root
store.db.password=mdjfbzyj515
store.db.minConn=5
store.db.maxConn=30
store.db.globalTable=global_table
store.db.branchTable=branch_table
store.db.queryLimit=100
store.db.lockTable=lock_table
store.db.maxWait=5000
# 事务、日志等配置
server.recovery.committingRetryPeriod=1000
server.recovery.asynCommittingRetryPeriod=1000
server.recovery.rollbackingRetryPeriod=1000
server.recovery.timeoutRetryPeriod=1000
server.maxCommitRetryTimeout=-1
server.maxRollbackRetryTimeout=-1
server.rollbackRetryTimeoutUnlockEnable=false
server.undo.logSaveDays=7

```

```

server.undo.logDeletePeriod=86400000

# 客户端与服务端传输方式
transport.serialization=seata
transport.compressor=none
# 关闭metrics功能，提高性能
metrics.enabled=false
metrics.registryType=compact
metrics.exporterList=prometheus
metrics.exporterPrometheusPort=9898

```

运行

\seata-server-1.4.2\bin\seata-server.bat

引入依赖

```

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    <exclusions>
        <!-- 版本较低，1.3.0，因此排除-->
        <exclusion>
            <artifactId>seata-spring-boot-starter</artifactId>
            <groupId>io.seata</groupId>
        </exclusion>
    </exclusions>
</dependency>
<!-- seata starter 采用1.4.2版本-->
<dependency>
    <groupId>io.seata</groupId>
    <artifactId>seata-spring-boot-starter</artifactId>
    <version>${seata.version}</version>
</dependency>

```

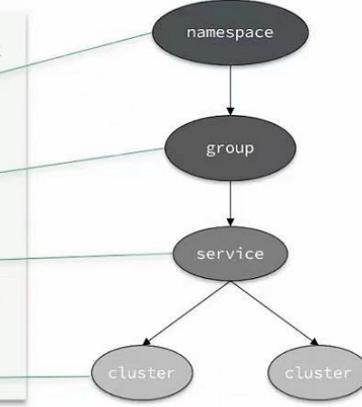
配置文件

2. 然后，配置application.yml，让微服务通过注册中心找到seata-tc-server：

```

seata:
  registry: # TC服务注册中心的配置，微服务根据这些信息去注册中心获取tc服务地址
    # 参考tc服务自己的registry.conf中的配置,
    # 包括：地址、namespace、group、application-name、cluster
    type: nacos
    nacos: # tc
      server-addr: 127.0.0.1:8848
      namespace: "##"
      group: DEFAULT_GROUP
      application: seata-tc-server # tc服务在nacos中的服务名称
      tx-service-group: seata-demo # 事务组，根据这个获取tc服务的cluster名称
  service:
    vgroup-mapping: # 事务组与TC服务cluster的映射关系
      seata-demo: SH

```



注意一个服务需要命名空间，组，服务，集群才能找到，但这里有点绕，还需要一个事务组

```

seata:
  registry: # TC服务注册中心的配置，微服务根据这些信息去注册中心获取tc服务地址

```

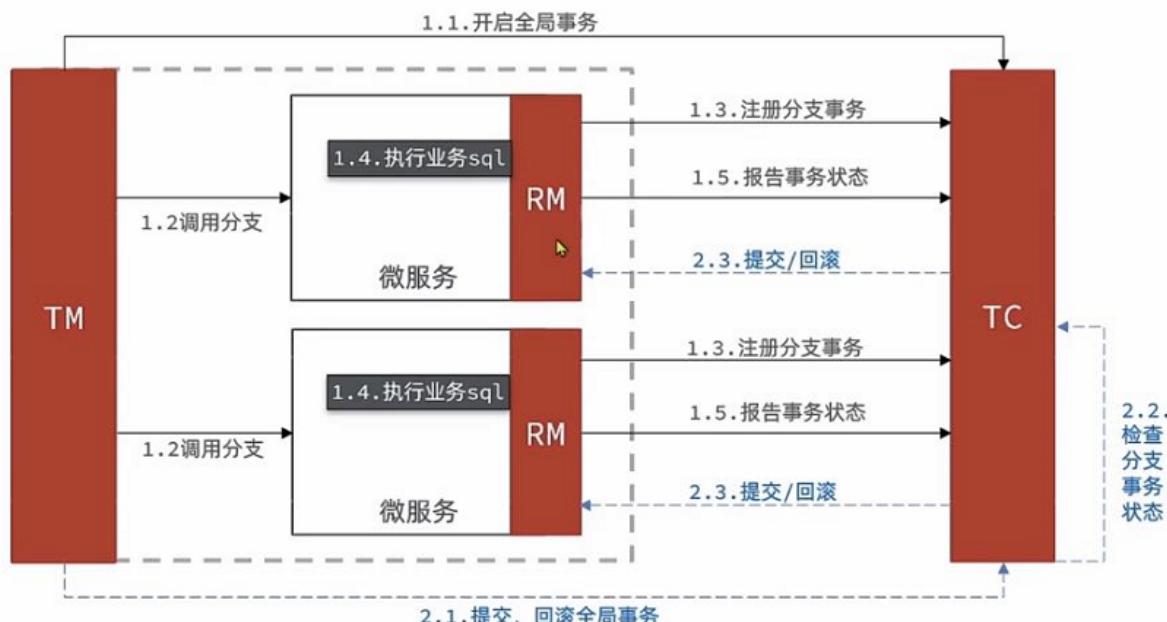
```

# 参考tc服务自己的registry.conf中的配置
type: nacos
nacos: # tc
  server-addr: 127.0.0.1:8848
  namespace: ""      #默认不写就是public
  group: DEFAULT_GROUP
  application: seata-tc-server # tc服务在nacos中的服务名称
  cluster: SH
tx-service-group: seata-demo # 事务组, 根据这个获取tc服务的cluster名称
service:
  vgroup-mapping: # 事务组与TC服务cluster的映射关系
    seata-demo: default

```

XA模式原理

RM准备好数据库操作后跟TC报告，若是所有RM都报告成功，则TC下达指令要求所有RM提交，否则就回滚。



优点是强一致，缺点是占用资源性能差，依赖关系数据库实现事务

代码

修改配置开启数据源代码的XA模式

```

seata:
  data-source-proxy-mode: XA
// accountclient, storageclient以及本服务器都要修改配置

```

发起全局事务@GlobalTransactional

```

@Override
@Transactional
public Long create(Order order) {
  // 创建订单
  orderMapper.insert(order);
  try {
    // 扣用户余额

```

```

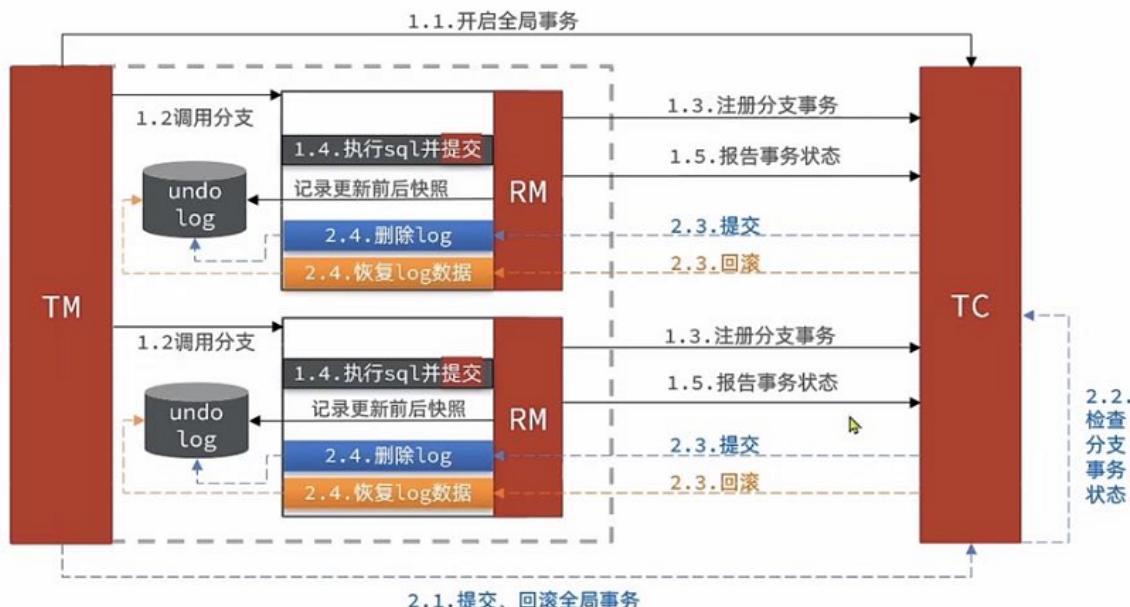
        accountClient.deduct(order.getUserId(), order.getMoney());
        // 扣库存
        storageClient.deduct(order.getCommodityCode(), order.getCount());

    } catch (FeignException e) {
        log.error("下单失败, 原因:{}" , e.contentUTF8() , e);
        throw new RuntimeException(e.contentUTF8() , e);
    }
    return order.getId();
}

```

AT模式原理

AT模式同样是分阶段提交的事务模型，弥补了XA模式中资源锁定周期过长的缺陷。只不过是记录更新前后快照数据库。若有RM失败可以全都回滚且删除快照，全都成功就删除快照。



代码

AT模式所需要的表global_table、branch_table、lock_table要导入到tc服务关联的数据库

```

create table branch_table
(
    branch_id bigint not null
        primary key,
    xid varchar(128) not null,
    transaction_id bigint null,
    resource_group_id varchar(32) null,
    resource_id varchar(256) null,
    branch_type varchar(8) null,
    status tinyint null,
    client_id varchar(64) null,
    application_data varchar(2000) null,
    gmt_create datetime(6) null,
    gmt_modified datetime(6) null
)charset=utf8;
create index idx_xid on branch_table (xid);

```

```

create table global_table
(
    xid varchar(128) not null
        primary key,
    transaction_id bigint null,
    status tinyint not null,
    application_id varchar(32) null,
    transaction_service_group varchar(32) null,
    transaction_name varchar(128) null,
    timeout int null,
    begin_time bigint null,
    application_data varchar(2000) null,
    gmt_create datetime null,
    gmt_modified datetime null
)charset=utf8;
create index idx_gmt_modified_status on global_table (gmt_modified, status);
create index idx_transaction_id on global_table (transaction_id);

create table lock_table
(
    row_key varchar(128) not null
        primary key,
    xid varchar(96) null,
    transaction_id bigint null,
    branch_id bigint not null,
    resource_id varchar(256) null,
    table_name varchar(32) null,
    pk varchar(36) null,
    gmt_create datetime null,
    gmt_modified datetime null
)charset=utf8;
create index idx_branch_id on lock_table (branch_id);

```

各分支数据库要导入undo_lock表

```

create table health_ums.undo_log
(
    branch_id bigint not null comment 'branch transaction id',
    xid varchar(100) not null comment 'global transaction id',
    context varchar(128) not null comment 'undo_log context,such as
serialization',
    rollback_info longblob not null comment 'rollback info',
    log_status int not null comment '0:normal status,1:defense status',
    log_created datetime(6) not null comment 'create datetime',
    log_modified datetime(6) not null comment 'modify datetime',
    constraint ux_undo_log
        unique (xid, branch_id)
)
comment 'AT transaction mode undo table' charset=utf8;

```

修改配置

```

seata:
    data-source-proxy-mode: AT

```

使用注解：

发起全局事务@GlobalTransactional

脏写问题

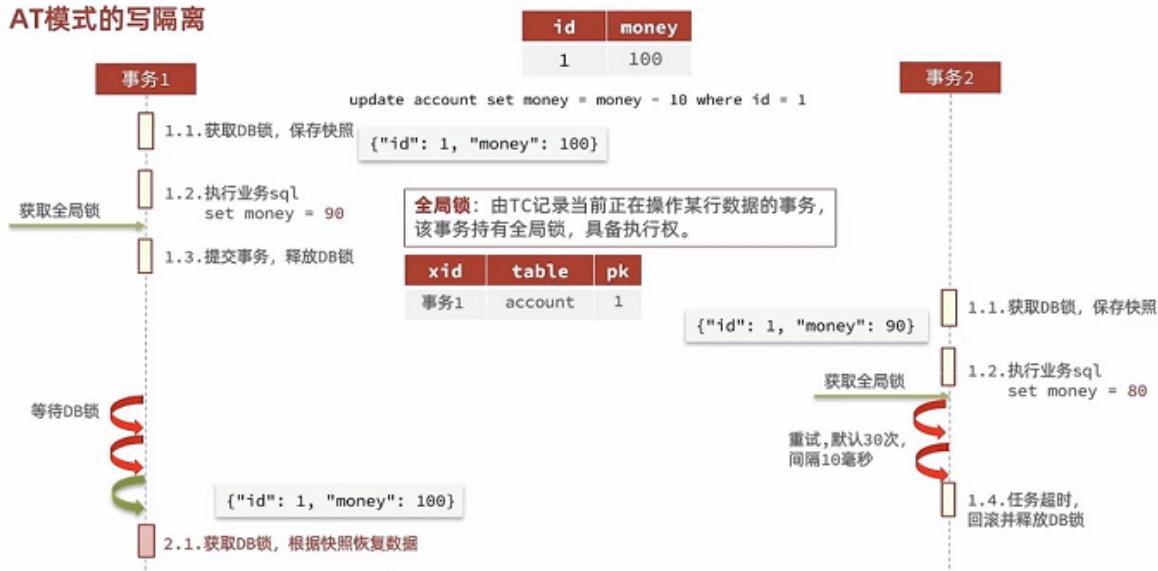
问题在于回滚前已经更新了数据。快照里应该修改为80但是是100

AT模式的脏写问题



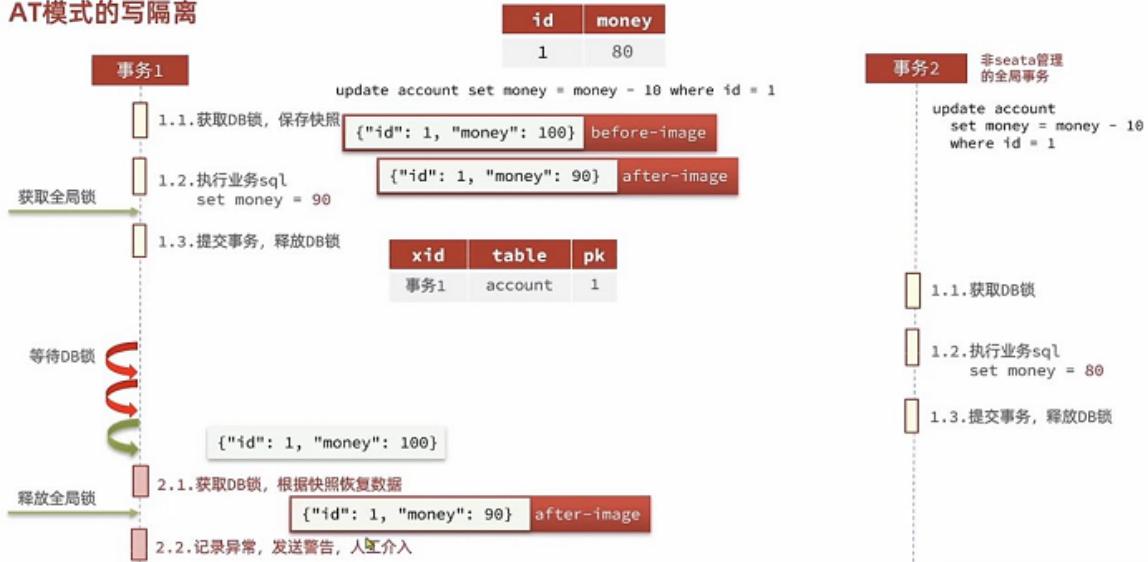
AT模式利用全局锁进行写隔离，即锁定某一行数据不可修改。2.1中回滚快照数据，快照数据是正确的，因为事务2获取全局锁执行sql失败任务超时回滚并释放db锁，没有对数据进行修改。

AT模式的写隔离



若是事务2非seata管理的全局事务，事务1快照后90与当前数据库80不一致，报告异常

AT模式的写隔离



优缺点

优点是：

一阶段完成直接提交事务，释放数据库资源，性能比较好

利用全局锁实现读写隔离

没有代码侵入，框架自动完成回滚和提交

缺点是：

两阶段之间属于软状态，属于最终一致

框架的快照功能会影响性能，但比XA模式要好很多

Xa和AT模式区别在于

xa模式一阶段不提交事务，锁定资源，at模式一阶段直接提交，不锁定资源。

xa模式依赖数据库机制实现回滚，at模式利用数据快照实现数据回滚

xa模式强一致，at模式最终一致

TCC模式

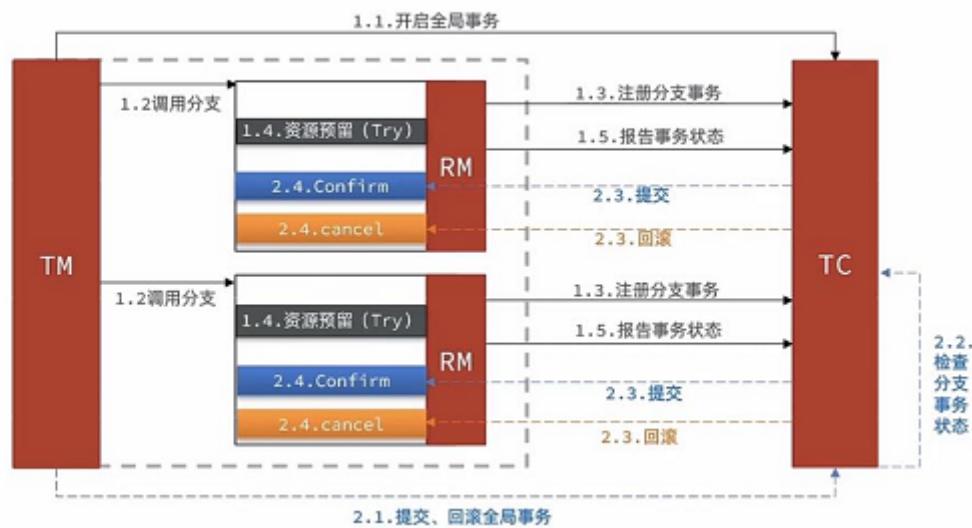
tcc与at模式非常相似，每阶段都是独立事务，不同的是tcc通过人工编码来实现数据恢复，需要实现三个方法

try：资源的检测和预留（检测资源并且冻结资源）（准备阶段）

confirm：完成资源操作业务，要求try成功confirm一定要能成功（运行阶段）

cancel：预留资源释放，try的反向操作

TCC的工作模型图：



空回滚和业务悬挂

当某分支事务的try阶段阻塞时候，可能导致全局事务超时而触发二阶段的cancel操作，在未执行try操作时候先执行了cancel操作，这时cancel不能做回滚，就是空回滚。

对于已经空回滚的业务，如果以后继续执行try，就永远不可能confirm或者cancel，这就是业务悬挂，应当阻止执行空回滚后的try操作，避免悬挂。

幂等问题

防止多次执行try-confirm-cancel

代码

业务自己的写，况且这个逻辑都是围着这个表，框架难道不成帮你把你sql表自动生成吗？？

多一句没有，少一句不行，用最短时间，教会最实用的技术！

业务分析

为了实现空回滚、防止业务悬挂，以及幂等性要求。我们必须在数据库记录冻结金额的同时，记录当前事务id和执行状态，为此我们设计了一张表：

```
CREATE TABLE `account_freeze_tbl` (
  `xid` varchar(128) NOT NULL,
  `user_id` varchar(255) DEFAULT NULL COMMENT '用户id',
  `freeze_money` int(11) unsigned DEFAULT '0' COMMENT '冻结金额',
  `state` int(1) DEFAULT NULL COMMENT '事务状态, 0:try, 1:confirm, 2:cancel',
  PRIMARY KEY (`xid`) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 ROW_FORMAT=COMPACT;
```

Try业务

- 记录冻结金额和事务状态到 account_freeze表
- 扣减account表可用金额

Confirm业务

- 根据xid删除 account_freeze表的冻结记录

Cancel业务

- 修改account_freeze表，冻结金额为0, state为2
- 修改account表，恢复可用金额

如何判断是否空回滚

- cancel业务中，根据xid查询account_freeze，如果为null则说明try还没做，需要空回滚

如何避免业务悬挂

- try业务中，根据xid查询 account_freeze，如果已经存在则证明Cancel已执行，拒绝执行try业务

@LocalTCC

```
public interface AccountTCCService {
```

```

/*
 * @TwoPhaseBusinessAction 中name, commitMethod, rollbackMethod即try-confirm-
cancel三个方法的名字
 */
@TwoPhaseBusinessAction(name = "deduct", commitMethod =
"confirm", rollbackMethod = "cancel")
/*
 * @BusinessActionContextParameter即参数名,目的是让commitMethod和rollbackMethod方法
也可以获取参数
 */
void deduct(@BusinessActionContextParameter(paramName = "userId") String
userId,@BusinessActionContextParameter(paramName = "money") int money);

/**
 *
 * @param ctx BusinessActionContext是获取try方法的参数,例如
ctx.getActionContext("userId").toString()
 * @return 执行是否成功
 */
boolean confirm(BusinessActionContext ctx);

boolean cancel(BusinessActionContext ctx);

}

```

```

public class AccountTCCServiceImpl implements AccountTCCService {
    @Autowired
    private AccountMapper accountMapper;
    @Autowired
    private AccountFreezeMapper freezeMapper;
    @Override
    @Transactional //写事务
    public void deduct(String userId, int money) {
        //业务悬挂, 防止重复执行try
        //判断是否有冻结记录, 查询事务id
        AccountFreeze old = freezeMapper.selectById(RootContext.getXID());
        if (old!=null)
        {
            //已经执行过try了, 拒绝业务
            return;
        }
        //扣钱
        accountMapper.deduct(userId,money);
        //冻结金额, 事务状态
        AccountFreeze freeze=new AccountFreeze();
        freeze.setUserId(userId);
        freeze.setFreezeMoney(money);
        freeze.setState(AccountFreeze.State.TRY);
        freeze.setXid(RootContext.getXID());           //设置事务id
        freezeMapper.insert(freeze);                  //冻结的数据要写入冻结表中
    }

    @Override
    public boolean confirm(BusinessActionContext ctx) {
        //获取事务id并且删除冻结记录, 代表提交成功
    }
}

```

```

        int count=freezeMapper.deleteById(ctx.getXid());
        return count==1;//是否修改数据成功
        //confirm删除业务天生幂等，删除一次跟一百次都一样
    }

    @Override
    public boolean cancel(BusinessActionContext ctx) {
        //查询冻结记录
        AccountFreeze freeze = freezeMapper.selectById(ctx.getXid());
        //空回滚的判断，没有try，则查不到冻结记录
        if (freeze==null)
        {
            //查不到数据要做空回滚
            freeze=new AccountFreeze();
            //获取try的数据
            freeze.setUserId(ctx.getActionContext("userId").toString());
            freeze.setFreezeMoney(0);
            freeze.setState(AccountFreeze.State.CANCEL);
            freeze.setXid(RootContext.getXID());           //设置事务id
            freezeMapper.insert(freeze);                  //冻结的数据要写入冻结表中
            return true;
        }
        //幂等判断
        if (freeze.getState()==AccountFreeze.State.CANCEL)
        {
            //已经cancel过一次了
            return true;
        }
        //try的反向操作，不能删除冻结数据，要做空回滚和业务悬挂的判断
        accountMapper.refund(freeze.getUserId(),freeze.getFreezeMoney());
        //修改冻结记录以及冻结状态后修改冻结表数据
        freeze.setFreezeMoney(0);
        freeze.setState(AccountFreeze.State.CANCEL);
        int count=freezeMapper.updateById(freeze);
        return count==1;//是否修改数据成功
    }
}

```

优缺点

优点：

一阶段完成直接提交事务，释放数据库资源，性能好。

相比at模型，无需生成快照，无需使用全局锁，性能最强

不依赖数据库事务，而是依赖补偿操作，可以用非事务型数据库

缺点：

代码侵入，人为编写try，confirm，cancel

软状态，事务是最终一致

需要考虑confirm和cancel失败情况，做好幂等处理

saga模式

一阶段：直接提交本地事务

二阶段：成功则什么都不做，失败则编写补偿业务来回滚

优缺点

优点：

事务参与者可以基于事件驱动实现异步调用，吞吐高

一阶段直接提交事务，无锁，性能好

不用编写tcc中三个阶段，实现简单

缺点：

软状态持续时间不确定，时效性差

没有锁，没有事务隔离，会有脏写

四种模式对比

	XA	AT	TCC	SAGA
一致性	强一致	弱一致	弱一致	最终一致
隔离性	完全隔离	基于全局锁隔离	基于资源预留隔离	无隔离
代码侵入	无	无	有，要编写三个接口	有，要编写状态机和补偿业务
性能	差	好	非常好	非常好
场景	对一致性隔离性有高要求的业务	基于关系型数据库的大多数分布事务场景都可以	对性能要求较高的事务，有非关系型数据库要参与的事务	业务流程长与多，参与者包括其他公司或遗留系统业务，无法提供tcc模式要求的三个接口