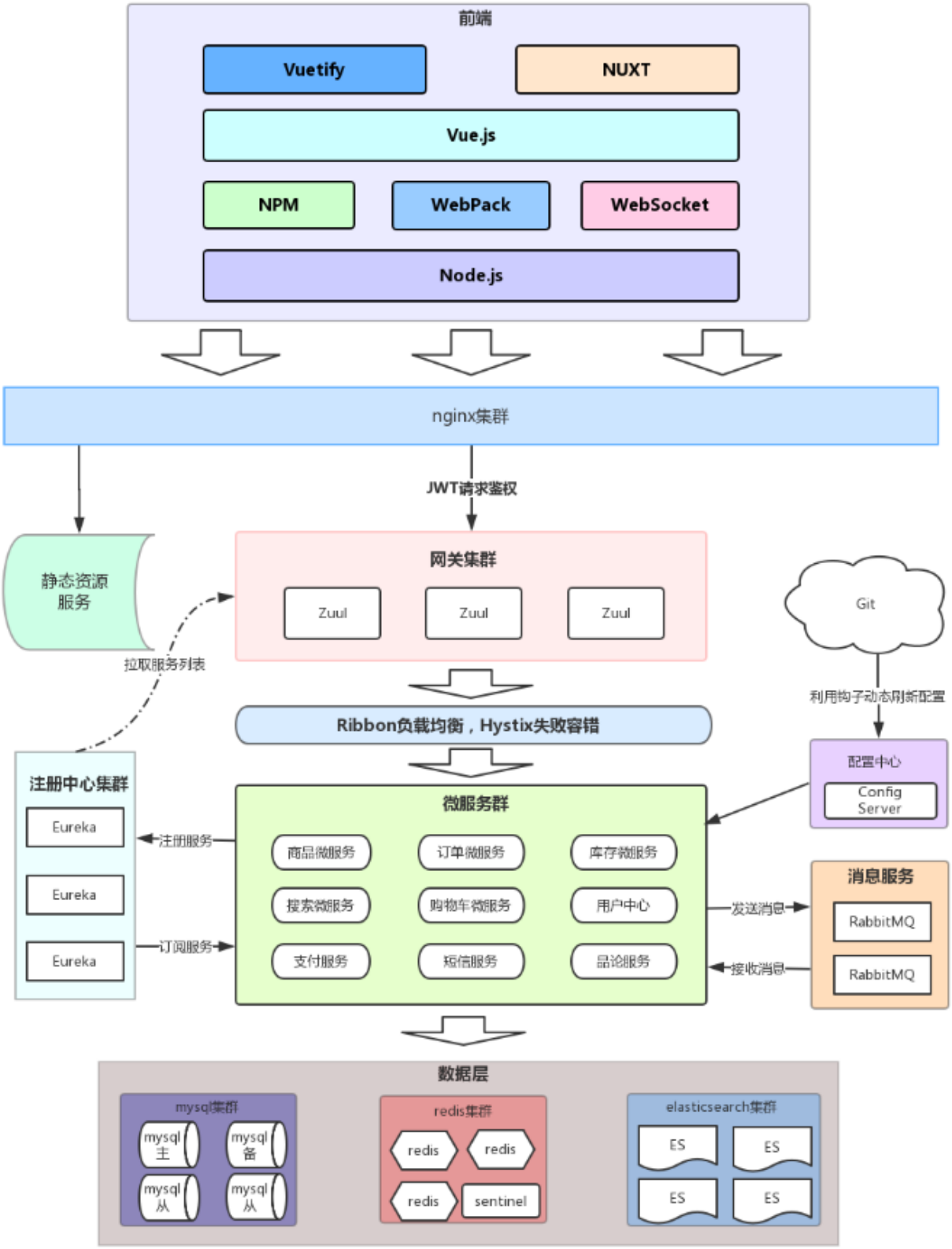
1.项目概述

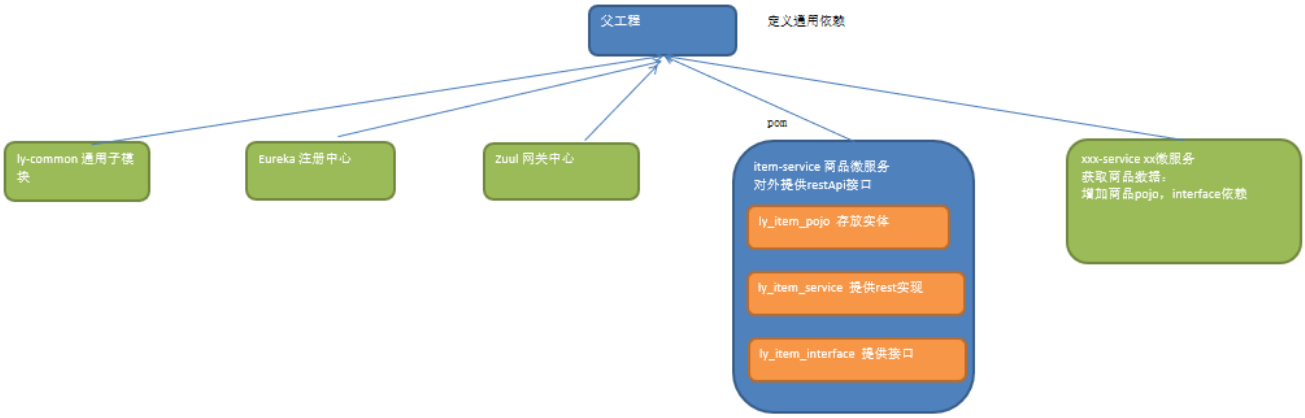
1.1.常见电商模式

- B2C：商家对个人，如：亚马逊、当当等
- C2C平台：个人对个人，如：闲鱼、拍拍网、ebay
- B2B平台：商家对商家，如：阿里巴巴、八方资源网等
- O2O：线上和线下结合，如：饿了么、电影票、团购等
- P2P：在线金融，贷款，如：网贷之家、人人聚财等。
- B2C平台：天猫、京东、一号店等

1.2.系统架构


前端
Vuetify
NUXT
Vue.js
NPM
WebPack
WebSocket
Node.js
nginx集群
JWT请求鉴权
静态资源服务
拉取服务列表
网关集群
Zuul
Zuul
Zuul
Git
利用钩子动态刷新配置
Ribbon负载均衡，Hystix失败容错
配置中心
Config Server
注册中心集群
Eureka
Eureka
Eureka
注册服务
订阅服务
微服务群
商品微服务
订单微服务
库存微服务
搜索微服务
购物车微服务
用户中心
支付服务
短信服务
品论服务
发送消息
接收消息
消息服务
RabbitMQ
RabbitMQ
数据层
mysql集群
mysql 主
mysql 备
mysql 从
mysql 从
redis集群
redis
redis
redis
sentinel
elasticsearch集群
ES
ES
ES
ES

2.后台项目构建



注意:实体的划分:

DTO:(Data Transfer Object)数据库传输对象,用于服务之间的调用

PO:(Persistence Object)持久化对象,用于跟数据库进行交互

VO:(View Object)视图对象,用于页面展示数据的

BO:(Business Object)业务对象

POJO:(plain ordinary java object)无规则简单java对象.是一个中间对象,可以转化为PO,DTO,VO.

3.实体类相关注解

3.1.Lambok

```
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
```

- @Setter:生成Setter方法,final变量不包
- @Getter:生成Getter方法
- @NoArgsConstructor:生成空参构造
- @AllArgsConstructor:生成全部参数构造
- @RequiredArgsConstructor:将标记为@NoNull的属性生成一个构造器
    - 如果运行中标记为@NoNull的属性为null,会抛出空指针异常
- @ToString:生成所有属性的toString( )方法
- @EqualsAndHashCode:生成equals( )方法和hashCode( )方法
- **@Data**:@Data = @Setter+@Getter+@EqualsAndHashCode+@NoArgsConstructor.getter所有变量,setter所有不为final的变量
- @Builder 构造Builder模式的结构,通过内部类Builder( )进行构建对象
- @Value 与@Data相对应,两个注解的主要区别就是如果变量不加@NonFinal,@Value会给所有的弄给final的.当然如果是final的话,就没有set方法.

- @Synchronized:同步方法
- **@Accessors(chain = true)** 支持链式风格编程

3.2.实体实例

```
@Data
@Table(name = "tb_brand")
public class Brand {
    @Id
    @KeySql(useGeneratedKeys = true)
    private Long id;
    private String name;
    private String image;
    private Character letter;
    private Date createTime;
    private Date updateTime;
}
```

对于通用Mapper的使用,参考通用Mapper.

4.统一异常处理

对于异常处理:我们首先想到的是:

`return ResponseEntity.status(HttpStatus.BAD_REQUEST).boby("价格不能为空");`

但对于以上的异常处理,首先`HttpStatus.BAD_REQUEST`这是一个字符串,那么我们方法的返回值就要写成`ResponseEntity<Srtring>`,显然不合适,所以在出现问题的地方我们不要返回,而是转为异常,然后再统一处理异常.

1.定义异常枚举,这样可以规范异常处理.在该枚举上标注@Getter注解,这样就可以得到该枚举的status和message属性.方便后续程序进行操作

```
/**
 * 异常 状态码 提示消息 枚举类型
 */
@Getter
public enum ExceptionEnum {
    //一定在最上方定义选择项
    OK(200, "操作成功"),
    PRICE_NOT_NULL(400, "价格不能为空"),
    ERROR(400, "操作失败"),
    OTHERS(500, "其他异常"),

    INVALID_FILE_TYPE(400, "无效的文件类型！"),
    INVALID_PARAM_ERROR(400, "无效的请求参数！"),
    INVALID_PHONE_NUMBER(400, "无效的手机号码"),
    INVALID_VERIFY_CODE(400, "验证码错误！"),
    INVALID_USERNAME_PASSWORD(400, "无效的用户名和密码！"),
    INVALID_SERVER_ID_SECRET(400, "无效的服务id和密钥！"),
    INVALID_NOTIFY_PARAM(400, "回调参数有误！"),
    INVALID_NOTIFY_SIGN(400, "回调签名有误！"),
```

```
    CATEGORY_NOT_FOUND(404, "商品分类不存在！"),
    BRAND_NOT_FOUND(404, "品牌不存在！"),
    SPEC_NOT_FOUND(404, "规格不存在！"),
    GOODS_NOT_FOUND(404, "商品不存在！"),
    CARTS_NOT_FOUND(404, "购物车不存在！"),
    APPLICATION_NOT_FOUND(404, "应用不存在！"),
    ORDER_NOT_FOUND(404, "订单不存在！"),
    ORDER_DETAIL_NOT_FOUND(404, "订单数据不存在！"),

    DATA_TRANSFER_ERROR(500, "数据转换异常！"),
    INSERT_OPERATION_FAIL(500, "新增操作失败！"),
    UPDATE_OPERATION_FAIL(500, "更新操作失败！"),
    DELETE_OPERATION_FAIL(500, "删除操作失败！"),
    FILE_UPLOAD_ERROR(500, "文件上传失败！"),
    DIRECTORY_WRITER_ERROR(500, "目录写入失败！"),
    FILE_WRITER_ERROR(500, "文件写入失败！"),
    SEND_MESSAGE_ERROR(500, "短信发送失败！"),
    INVALID_ORDER_STATUS(500, "订单状态不正确！"),

    UNAUTHORIZED(401, "登录失效或未登录！");

    private Integer status;
    private String message;

    //提供私有构造,该构造默认就是私有的
    ExceptionEnum(Integer status, String message) {
        this.status = status;
        this.message = message;
    }
}
```

2.自定义异常,该自定义异常继承RuntimeException,构造函数使用的是传入的ExceptionEnum,我们将该枚举的message数据传入给自定义异常的父异常,将该枚举的status状态码数据传入给该自定义异常的status属性.

```
@Getter
public class LyException extends RuntimeException {
    //定义返回的状态码
    private Integer status;

    public LyException(ExceptionEnum exceptionEnum) {
        super(exceptionEnum.getMessage());
        this.status = exceptionEnum.getStatus();
    }

    public LyException(ExceptionEnum exceptionEnum, Throwable cause) {
        super(exceptionEnum.getMessage(), cause);
        this.status = exceptionEnum.getStatus();
    }
}
```

3.返回异常结果 这边的DateTime.now( ).toString( )方法使用了日期的工具类:JodaTime,我们要引入依赖在ly-common中.

```
@Getter
public class ExceptionResult {
    private Integer status;
    private String messgae;
    private String timestamp;

    public ExceptionResult(LyException e){
        this.status = e.getStatus();
        this.messgae = e.getMessage();
        this.timestamp = DateTime.now().toString("yyyy-MM-dd HH:mm:ss");
    }
}
```

4.统一异常处理

我们使用SpringMVC提供的统一异常拦截器.

@ControllerAdvice:默认情况下,会拦截所有加了@Controller的类

@ExceptionHandler(RuntimeException.class);作用在方法上,声明要处理的异常类型,可以有多个.被声明的方法可以看做是一个SpringMVC的handler.此处使用了Spring的注解,因此需要引入spring-web依赖.

```
@ControllerAdvice
//处理controller层抛出异常,全部统一处理
public class BasicExceptionAdvice {

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> runtimeException(RuntimeException e){
        return ResponseEntity.status(500).body(e.getMessage());
    }

    @ExceptionHandler(LyException.class)
    public ResponseEntity<ExceptionResult> lyException(LyException e){
        ExceptionResult exceptionResult = new ExceptionResult(e);
        return ResponseEntity.status(e.getStatus()).body(exceptionResult);
    }
}
```

5.Nginx反向代理

1.Nginx可以通过命令行来启动,操作命令

- 启动: `start nginx.exe`
- 停止: `nginx.exe -s stop`
- 重新加载: `nginx.exe -s reload`

2.修改Nginx的配置文件,实现反向代理.

- 我们在Nginx的conf文件夹下修改nginx.conf.

```
    keepalive_timeout  65;

    #gzip  on;

    #引入当前目录下的.conf文件
    include vhost/*.conf;

    server {
        listen       80;
        server_name  localhost;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;
```

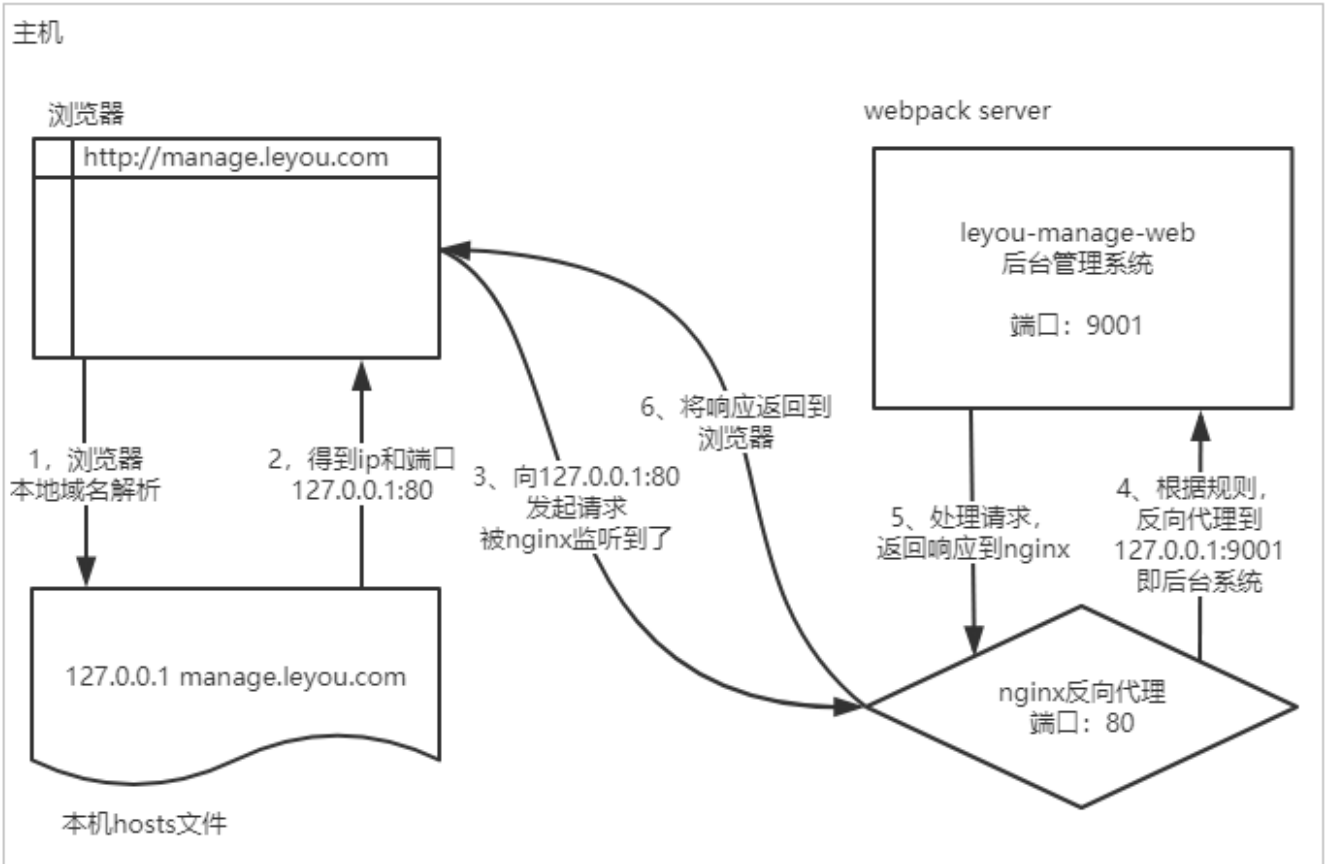
- 我们新建一个vhost文件夹,在文件夹中编写leyou.conf

```
#定义集群节点 默认轮询访问 通过 server 127.0.0.1:9001 weight = 3;配置权重
upstream leyou-manage{
    server 127.0.0.1:9001;
}
upstream leyou-gateway{
    server 127.0.0.1:10010;
}
upstream leyou-portal{
    server 127.0.0.1:9002;
}

#配置反向代理
server {
    listen 80;
    server_name manage.leyou.com;
    location / {
        proxy_pass http://leyou-manage;
        proxy_connect_timeout 600;
        proxy_read_timeout 5000;
    }
}
server {
    listen 80;
    server_name  www.leyou.com;
    location /item {
        root html;
    }
    location / {
        proxy_pass http://leyou-portal;
        proxy_connect_timeout 600;
        proxy_read_timeout 5000;
    }
```

```
}
server {
    listen 80;
    server_name  api.leyou.com;
    location / {
        proxy_pass http://leyou-gateway;
        proxy_connect_timeout 600;
        proxy_read_timeout 5000;
    }
}

server {
    listen 80;
    server_name  image.leyou.com;
    location /images {
        root html;
    }
}
```

- upstream：定义一个负载均衡集群，例如leyou-manage
  - server：集群中某个节点的ip和port信息，可以配置多个，实现负载均衡，默认轮询
- server：定义一个监听服务配置
  - listen：监听的端口
  - server_name：监听的域名
  - location：匹配当前域名下的哪个路径。例如：`/`，代表的是一切路径
    - proxy_pass：监听并匹配成功后，反向代理的目的地，可以指向某个ip和port，或者指向upstream定义的负载均衡集群，nginx反向代理时会轮询中服务列表中选择。

反向代理的流程:



- 浏览器准备发送请求.访问http://manage.leyou.com,但是需要进行域名解析.
- 优先进行本地本机的hosts文件中查找域名映射的IP地址,如果查找到就返回IP,没找就进行域名解析器解析.因为我们修改了hosts,所以解析成功,得到地址127.0.0.1.
- 请求被发往解析得到ip,并且默认使用80端口 http://127.0.0.1:80,本机的nginx一直监听80端口,因此捕获这个请求.
- nginx中配置了反向代理规则,将manage.leyou.com代理到http://127.0.0.1:9001
- 主机上的后台系统的webpack server监听的端口是9001,得到请求并处理,完成后将响应返回到nginx.
- nginx将得到的结果返回到浏览器.

6.跨域问题处理

跨域是指跨域名的访问,以情况都是属于跨域.

跨域原因说明	示例
域名不同	`www.jd.com` 与 `www.taobao.com`
域名相同，端口不同	`www.jd.com:8080` 与 `www.jd.com:8081`
二级域名不同	`item.jd.com` 与 `miaosha.jd.com`

但是跨域不一定会有跨域问题.因为跨域问题是浏览器对于ajax请求的一种安全限制:**一个页面发起的ajax请求,只能是与当前页同域名的路径**,这能有效的阻止跨站攻击.因此:**跨域问题,是针对ajax的一种限制.**

6.1.CORS解决跨域

cors是W3C标准,全称是"跨域资源共享"(Cross-origin resource sharing).

浏览器端:目前所有浏览器都支持该功能(IE10以下不行),整个CORS通信过程,都是浏览器自动完成,不需要用户参与.

服务端:CORS通信与ajax没有任何差别,因此你不需要改变以前的业务逻辑.只不过,浏览器会在请求中携带一些头信息.我们需要以此判断是否运行其跨域,然后再响应头中加入一些信息即可.者一般通过过滤器完成即可.

6.2.CORS的原理

浏览器会将ajax请求分为两类,处理方案略有差异:简单请求和特殊请求.

6.2.1.简单请求

只要同时满足以下两大条件，就属于简单请求。：

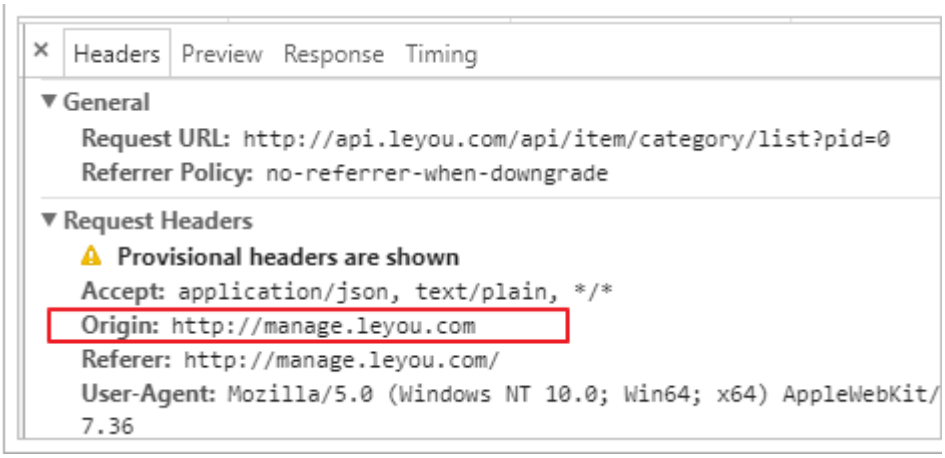
(1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

(2) HTTP的头信息不超出以下几种字段：

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type：只限于三个值`application/x-www-form-urlencoded`、`multipart/form-data`、`text/plain`

当浏览器发现ajax请求是简单请求时,会在请求头中携带一个字段:`Origin`.



Origin中会指出当前请求属于哪个域,服务会根据这个值决定是否允许其跨域.如果想要服务器允许跨域,需要在返回的响应头中携带下面信息

```
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
```

- Access-Control-Allow-Origin：可接受的域，是一个具体域名或者*，代表任意

- Access-Control-Allow-Credentials：是否允许携带cookie，默认情况下，cors不会携带cookie，除非这个值是true

注意：

如果跨域请求要想操作cookie，需要满足3个条件：

- 服务的响应头中需要携带Access-Control-Allow-Credentials并且为true。
- 浏览器发起ajax需要指定withCredentials 为true
- 响应头中的Access-Control-Allow-Origin一定不能为*，必须是指定的域名

6.2.2.特殊请求

特殊请求会在真实通信之前,增加一次HTTP查询请求,称为"预检"(preflight)请求.

浏览器先询问服务器,当前网页所在域名是否在服务器的许可名单之中,以及可以使用哪些Http方法和头信息字段,只有得到肯定答复,浏览器才会正式发出Ajax请求,否则报错.

```
OPTIONS /cors HTTP/1.1
Origin: http://manage.leyou.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.leyou.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

与简单请求相比，除了Origin以外，多了两个头：

- Access-Control-Request-Method：接下来会用到的请求方式，比如PUT
- Access-Control-Request-Headers：会额外用到的头信息

服务器收到预检请求,如果许可跨域,会发出响应

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Max-Age: 1728000
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

除了`Access-Control-Allow-Origin`和`Access-Control-Allow-Credentials`以外，这里又额外多出3个头：

- Access-Control-Allow-Methods：允许访问的方式
- Access-Control-Allow-Headers：允许携带的头

- Access-Control-Max-Age：本次许可的有效时长，单位是秒,过期之前的ajax请求就无需在此进行预检了.

6.3.CorsFilter

SpringMVC已经帮我们写好了CORS的跨域过滤器:CorsFilter.

```
@Configuration
public class GlobalCorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        CorsConfiguration corsConfiguration = new CorsConfiguration();
        //允许请求头的信息
        corsConfiguration.addAllowedHeader("*");
        //允许请求方法
        corsConfiguration.addAllowedMethod("OPTIONS");
        corsConfiguration.addAllowedMethod("GET");
        corsConfiguration.addAllowedMethod("POST");
        corsConfiguration.addAllowedMethod("PUT");
        corsConfiguration.addAllowedMethod("DELETE");
        //允许的域
        corsConfiguration.addAllowedOrigin("http://manage.leyou.com");
        //预请求的有效时间
        corsConfiguration.setMaxAge(86400L);
        //是否允许携带cookie
        corsConfiguration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**",corsConfiguration);
        CorsFilter corsFilter = new CorsFilter(source);

        return corsFilter;
    }
}
```

6.4.代码优化

参考SpringBoot的XXXAutoConfiguration,我们可以将配置放在配置文件上

yml文件中的配置如下:

```
ly:
  cors:
    allowedOrigins:
      - http://manage.leyou.com
    allowedCredentials: true
    allowedHeaders:
      - "*"
    allowedMethods:
      - GET
      - POST
      - DELETE
```

```
      - PUT
      - OPTIONS
      - HEAD
    maxAge: 86400
    filterPath: "/**"
```

属性类,与yml中的属性值进行绑定

```
@Data
@ConfigurationProperties(prefix = "ly.cors")
public class GlobalCorsProperties {
    private List<String> allowedOrigins;
    private Boolean allowedCredentials;
    private List<String> allowedMethods;
    private List<String> allowedHeaders;
    private Long maxAge;
    private String filterPath;
}
```

配置类

```
@Configuration
@EnableConfigurationProperties(GlobalCorsProperties.class)
public class GlobalCorsConfig {
    @Bean
    public CorsFilter corsFilter(GlobalCorsProperties prop) {
        //1.添加CORS配置信息
        CorsConfiguration config = new CorsConfiguration();
        //1) 允许的域,不要写*，否则cookie就无法使用了
        prop.getAllowedOrigins().forEach(config::addAllowedOrigin);
        //2) 是否发送Cookie信息
        config.setAllowCredentials(prop.getAllowedCredentials());
        //3) 允许的请求方式
        prop.getAllowedMethods().forEach(config::addAllowedMethod);
        //4) 允许的头信息
        prop.getAllowedHeaders().forEach(config::addAllowedHeader);
        //5) 配置有效期
        config.setMaxAge(prop.getMaxAge());

        //2.添加映射路径，我们拦截一切请求
        UrlBasedCorsConfigurationSource configSource = new
UrlBasedCorsConfigurationSource();
        configSource.registerCorsConfiguration(prop.getFilterPath(), config);

        //3.返回新的CORSFilter.
        return new CorsFilter(configSource);
    }
}
```

7.图片上传

目前我们所知道的是:

- 请求方式:上传肯定是POST.
- 请求参数:文件,参数名是file,SpringMVC会封装为一个接口:MultipleFile.
- 返回结果:这里上传与表单分离.文件不是跟随表单一起提交,而是单独上传并得到结果,随后把结果跟随表单提交,因此上传成功之后需要返回一个可以访问的文件url路径.

7.1.上传到本地的Nginx服务器

1.controller层代码

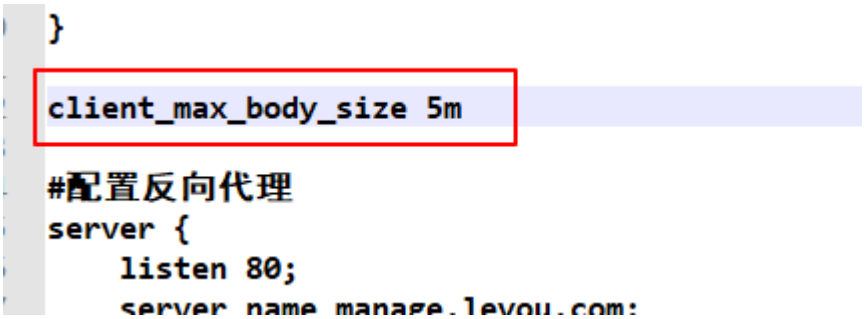
```
@RestController
public class UploadController {

    @Autowired
    private UploadService uploadService;

    @PostMapping("image")
    public ResponseEntity<String> uploadImage(@RequestParam("file") MultipartFile file) {
        // 返回200，并且携带url路径
        return ResponseEntity.ok(this.uploadService.upload(file));
    }
}
```

2.service层我们需要检验文件的大小,检验文件的媒体类型,检验文件的内容.

```
@Service
public class UploadService {

    // 支持的文件类型
    private static final List<String> suffixes = Arrays.asList("image/png", "image/jpeg",
"image/bmp");

    public String upload(MultipartFile file) {
        // 1、图片信息校验
        // 1)校验文件类型
        String type = file.getContentType();
        if (!suffixes.contains(type)) {
            throw new LyException(ExceptionEnum.INVALID_FILE_TYPE);
        }

        // 2)校验图片内容
        BufferedImage image = null;
        try {
            image = ImageIO.read(file.getInputStream());
        } catch (IOException e) {
            throw new LyException(ExceptionEnum.INVALID_FILE_TYPE);
        }
        if (image == null) {
            throw new LyException(ExceptionEnum.INVALID_FILE_TYPE);
        }
```

```
        // 2、保存图片
        // 2.1、生成保存目录,保存到nginx所在目录的html下，这样可以直接通过nginx来访问到
        File dir = new File("C:\\develop\\nginx-1.12.2\\html\\");
        if (!dir.exists()) {
            dir.mkdirs();
        }
        try {
            // 2.2、保存图片
            file.transferTo(new File(dir, file.getOriginalFilename()));

            // 2.3、拼接图片地址
            return "http://image.leyou.com/" + file.getOriginalFilename();
        } catch (Exception e) {
            throw new LyException(ExceptionEnum.FILE_UPLOAD_ERROR);
        }
    }
}
```

如果上传一个超过1M大小的文件,就会收到错误响应,`413 Request Entity Too Large`.我们可以在nginx配置文件中设置大小的限制为5M.



当我们再次进行测试,会发现Zuul网关报错,为什么?因为文件上传会经过Zuul,再到上传微服务.而zuul这里发现文件过大,于是报错.默认情况下,所有的缓存经过Zuul网关的代理,默认会通过SpringMVC预先对请求进行进行处理,缓存.普通请求并不会有什么影响,但是对于文件上传,会造成不必要的网络负担.在高并发时,可能会导致网络阻塞,Zuul网关不可用,这样我们的整个系统就会瘫痪.所以我们上传文件需要绕过请求需要绕过请求的缓存,直接到达目标微服务.我们需要修改到以/Zuul为前缀.

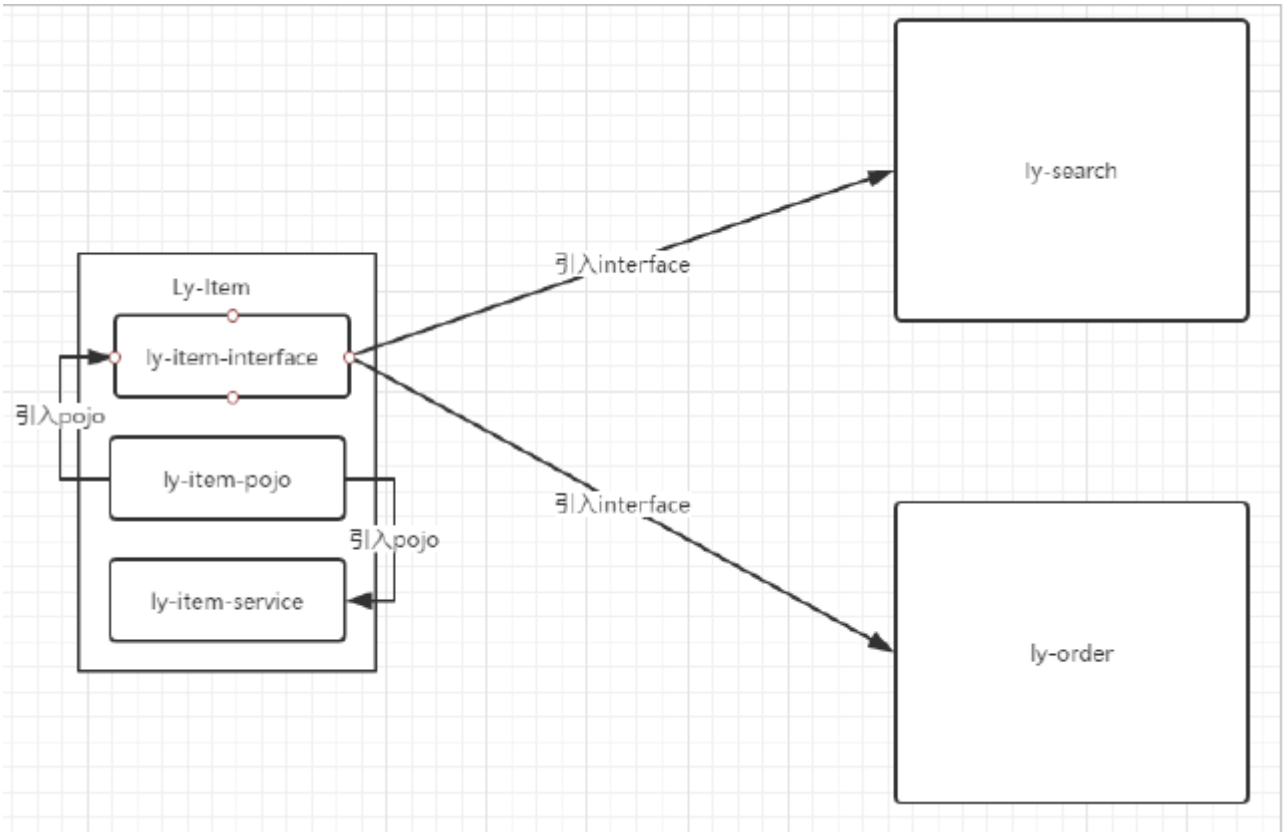
```
client_max_body_size 5m;
server {
    listen       80;
    server_name  api.leyou.com;

    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Server $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    location /api/upload {
        rewrite "^/(.*)$" /zuul/$1;
    }

    location / {
        proxy_pass http://127.0.0.1:10010;
        proxy_connect_timeout 600;
```

```
        proxy_read_timeout 600;
    }
}
```

7.2.上传到阿里云对象存储OSS

8.商品搜索

8.1.导入到索引库.

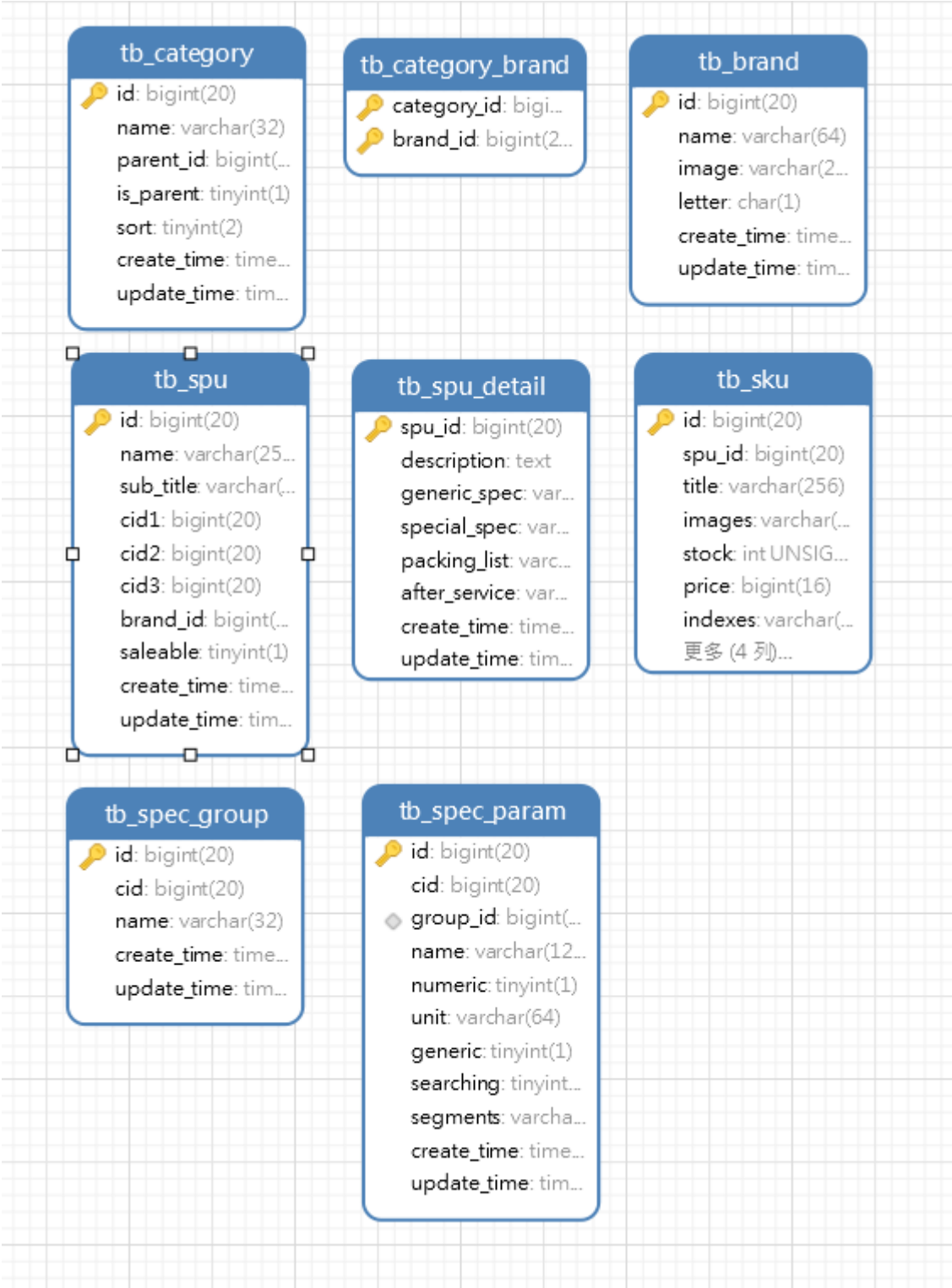


在创建搜索微服务的时候,由于我们需要使用到商品的微服务,所以我们在ly-item-interface中建立fegin的客户端,将来在搜索微服务中添加ly-item-interface的依赖,在ly-item-interface中添加ly-item-pojo的依赖.这样我们在搜索微服务去使用fegin的客户端的时候,就可以通过注入fegin的客户端来调用商品微服务.具体的架构如上图所示.

```
@FeignClient(value = "item-service")
public interface GoodsClient {

    @GetMapping("/spu/page")
    PageResult<SpuDTO> querySpuByPage(
            @RequestParam(name = "page",required = false,defaultValue = "1") Integer page,
            @RequestParam(name = "rows",required = false,defaultValue = "5") Integer rows,
            @RequestParam(name = "key",required = false) String key,
            @RequestParam(name = "saleable",required = false) Boolean saleable
    );
}
```

上述代码只是feign客户端的部分代码.

分析清除表关系,选择好将来我们需要存入索引库的内容:这样方便我们去封装一个实体Bean去存入索引库.




```
@Data
@Document(indexName = "goods", type = "docs", shards = 1, replicas = 1)
public class Goods {
    @Id
    @Field(type = FieldType.Keyword)
    private Long id; // spuId
    @Field(type = FieldType.Keyword, index = false)
    private String subTitle;// 卖点
    @Field(type = FieldType.Keyword, index = false)
    private String skus;// sku信息的json结构
```

```
    @Field(type = FieldType.Text, analyzer = "ik_max_word")
    private String all; // 所有需要被搜索的信息，包含标题，分类，甚至品牌
    private Long brandId;// 品牌id
    private Long categoryId;// 商品3级分类id
    private Long createTime;// spu创建时间
    private Set<Long> price;// 价格
    private Map<String, Object> specs;// 可搜索的规格参数，key是参数名，值是参数值
}
```

通过分析:在页面**商品展示**中,我们需要sku的相关数据,包括sku的id,title,price,image,当然还有spu的id和spu的subTitle.在页面**条件搜索**中,我们需要brandId,categoryId,价格区间,规格参数,规格参数由于是键值对的方式,我们可以将其封装到Map集合中.最后,我们将需要被搜索的信息,包含标题,分类,品牌.全部放在放在一个字符串中,并用ik分词器进行分词.

在导入数据之前我们需要建立EasticSearch中goods的映射关系,当然我们也可以在实体类上写好配置,在导入实体的实体后ElasticSearch会自动帮我们创建好映射关系.

```
PUT /goods
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 1
  },
  "mappings": {
    "docs":{
      "properties": {
        "id":{
          "type": "keyword"
        },
        "subTitle":{
          "type": "keyword",
          "index": false
        },
        "skus":{
          "type": "keyword",
          "index": false
        },
        "all":{
          "type": "text",
          "analyzer": "ik_max_word"
        }
      },
      "dynamic_templates": [
        {
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "keyword"
            }
          }
        }
      ]
```

```
    }
  }
}
```

以下是构建构建Goods的Service

```
@Service
public class GoodsBuildServcie {
    //在使用的时候,Spring会为我们创建代理对象
    @Autowired
    private GoodsClient goodsClient;

    public Goods buildGoods(SpuDTO spu) {
        // 1 商品相关搜索信息的拼接：名称、分类、品牌、规格信息等
        // 1.1 分类
        String categoryNames = itemClient.queryCategoryByIds(spu.getCategoryIds())
                .stream().map(CategoryDTO::getName).collect(Collectors.joining(","));
        // 1.2 品牌
        BrandDTO brand = itemClient.queryBrandById(spu.getBrandId());
        // 1.3 名称等，完成拼接
        String all = spu.getName() + categoryNames + brand.getName();

        // 2 spu下的所有sku的JSON数组
        List<SkuDTO> skuList = itemClient.querySkuBySpuId(spu.getId());
        // 准备一个集合，用map来代替sku，只需要sku中的部分数据
        List<Map<String, Object>> skuMap = new ArrayList<>();
        for (SkuDTO sku : skuList) {
            Map<String, Object> map = new HashMap<>();
            map.put("id", sku.getId());
            map.put("price", sku.getPrice());
            map.put("title", sku.getTitle());
            map.put("image", StringUtils.substringBefore(sku.getImages(), ","));
            skuMap.add(map);
        }

        // 3 当前spu下所有sku的价格的集合
        Set<Long> price =
skuList.stream().map(SkuDTO::getPrice).collect(Collectors.toSet());

        // 4 当前spu的规格参数
        Map<String, Object> specs = new HashMap<>();

        // 4.1 获取规格参数key，来自于SpecParam中当前分类下的需要搜索的规格
        List<SpecParamDTO> specParams = itemClient.querySpecParams(null, spu.getCid3(),
true);
        // 4.2 获取规格参数的值，来自于spuDetail
        SpuDetailDTO spuDetail = itemClient.querySpuDetailById(spu.getId());
        // 4.2.1 通用规格参数值
        Map<Long, Object> genericSpec = JsonUtils.toMap(spuDetail.getGenericSpec(),
Long.class, Object.class);
        // 4.2.2 特有规格参数值
        Map<Long, List<String>> specialSpec =
JsonUtils.nativeRead(spuDetail.getSpecialSpec(), new TypeReference<Map<Long, List<String>>>
```

```
() {
        });

        for (SpecParamDTO specParam : specParams) {
            // 获取规格参数的名称
            String key = specParam.getName();
            // 获取规格参数值
            Object value = null;
            // 判断是否是通用规格
            if (specParam.getGeneric()) {
                // 通用规格
                value = genericSpec.get(specParam.getId());
            } else {
                // 特有规格
                value = specialSpec.get(specParam.getId());
            }
            // 判断是否是数字类型
            if(specParam.getNumeric()){
                // 是数字类型，分段
                value = chooseSegment(value, specParam);
            }
            // 添加到specs
            specs.put(key, value);
        }

        Goods goods = new Goods();
        // 从spu对象中拷贝与goods对象中属性名一致的属性
        goods.setBrandId(spu.getBrandId());
        goods.setCategoryId(spu.getCid3());
        goods.setId(spu.getId());
        goods.setSubTitle(spu.getSubTitle());
        goods.setCreateTime(spu.getCreateTime().getTime());
        goods.setSkus(JsonUtils.toString(skuMap));// spu下的所有sku的JSON数组
        goods.setSpecs(specs); // 当前spu的规格参数
        goods.setPrice(price); // 当前spu下所有sku的价格的集合
        goods.setAll(all);// 商品相关搜索信息的拼接：标题、分类、品牌、规格信息等
        return goods;
    }

    private String chooseSegment(Object value, SpecParamDTO p) {
        if (value == null || StringUtils.isBlank(value.toString())) {
            return "其它";
        }
        double val = parseDouble(value.toString());
        String result = "其它";
        // 保存数值段
        for (String segment : p.getSegments().split(",")) {
            String[] segs = segment.split("-");
            // 获取数值范围
            double begin = parseDouble(segs[0]);
            double end = Double.MAX_VALUE;
            if (segs.length == 2) {
                end = parseDouble(segs[1]);
```

```
            }
            // 判断是否在范围内
            if (val >= begin && val < end) {
                if (segs.length == 1) {
                    result = segs[0] + p.getUnit() + "以上";
                } else if (begin == 0) {
                    result = segs[1] + p.getUnit() + "以下";
                } else {
                    result = segment + p.getUnit();
                }
                break;
            }
        }
        return result;
    }

    private double parseDouble(String str) {
        try {
            return Double.parseDouble(str);
        } catch (Exception e) {
            return 0;
        }
    }
}
```

在测试类中我们手动存入到索引库中.

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class EsBuildGoodsIndex {

    @Autowired
    private GoodsBuildServcie goodsBuildServcie;

    @Autowired
    private GoodsClient goodsClient;

    @Autowired
    private GoodsDao goodsDao;

    @Test
    public void esBuildGoodsIndex() {
        int i = 1;
        PageResult<SpuDTO> spuDTOPageResult;
        do {
            spuDTOPageResult =  goodsClient.querySpuByPage(i, 10, null, null);

spuDTOPageResult.getItems().stream().map(goodsBuildServcie::buidGoods).forEach(goodsDao::sav
e);
            i++;
        } while (spuDTOPageResult.getItems().size()==10);
    }
}
```

8.2.基本搜索

基本搜索我们可以使用elasticsearchTemplate或者在使用dao层的search方法,它们需要的参数都是searchQuery,所以在service层我们可以选择两种方式进行搜索,这里演示的是调用elasticSearchResposity的search( )方法.

```
public PageResult<Goods> findGoodsByPage02(SearchPageRequest searchPageRequest){
        NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
        NativeSearchQuery query =
builder.withPageable(PageRequest.of(searchPageRequest.getPage() - 1,
searchPageRequest.getSize()))
                .withSourceFilter(new FetchSourceFilter(new String[]{"id", "subTitle",
"skus"}, null))
                .withQuery(QueryBuilders.matchQuery("all",
searchPageRequest.getKey())).build();
        Page<Goods> page = goodsDao.search(query);
        List<Goods> content = page.getContent();
        long totalElements = page.getTotalElements();
        int totalPages = page.getTotalPages();
        return new PageResult<>(content,totalElements,totalPages);
    }
```

8.3.聚合搜索

8.3.1.分类和品牌的聚合

在搜索过滤的结果中存在的分类和品牌才能作为过滤项用用户选择.我们怎么知道搜索结果中有哪些分类和品牌呢?我们可以使用ElasticSearch提供的聚合功能,在搜索条件的基础上,对搜索结果聚合,就能知道结果中包含哪些分类和品牌等搜索过滤条件.

由上图所示返回给页面的结果实际上是一个Map<String,List<?>>

Console

```
GET /goods/_search
{
  "size": 0,
  "query":{
    "match":{
      "all": "华为"
    }
  },
  "aggs":{
    "brandName":{
      "terms": {
        "field": "brandId"
      }
    },
    "categoryName":{
      "terms": {
        "field": "categoryId"
      }
    }
  }

}
```

```
{
  "took": 0,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": 80,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "brandName": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": 8557,
          "doc_count": 80
        }
      ]
    },
    "categoryName": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": 76,
          "doc_count": 80
        }
      ]
    }
  }
}
```

```
public Map<String, List<?>> searchFilter(SearchPageRequest searchPageRequest) {

    NativeSearchQueryBuilder builder = new NativeSearchQueryBuilder();
    builder.withQuery(basicQuery(searchPageRequest));
    builder.withSourceFilter(new FetchSourceFilter(null, null));
    builder.withPageable(PageRequest.of(0, 1));
    //添加聚合条件
    String brandAggName = "brandAggName";
    builder.addAggregation(AggregationBuilders.terms(brandAggName).field("brandId"));
    String categoryAggName = "categoryAggName";
    builder.addAggregation(AggregationBuilders.terms(categoryAggName).field("categoryId"));
    //对聚合的结果进行查询
    AggregatedPage<Goods> result = elasticsearchTemplate.queryForPage(builder.build(),
Goods.class);
    Aggregations aggregations = result.getAggregations();
    /*
    查看源码我们可以知道:
    public final <A extends Aggregation> A get(String name)
    并且Terms是Aggregations的一个子类
     */
    Terms brandAggTerms = aggregations.get(brandAggName);
    Terms categoryTerms = aggregations.get(categoryAggName);
    //处理结果
    Map<String, List<?>> resultMap = new LinkedHashMap<>();
    List<Long> brandAggTermsKeyResult = HandlerTermsAndMap(brandAggTerms);
    List<Long> categoryTermsKeyResult = HandlerTermsAndMap(categoryTerms);
```

```
    List<CategoryDTO> categoryDTOS = goodsClient.queryByIds(categoryTermsKeyResult);
    List<BrandDTO> brandDTOS =
brandAggTermsKeyResult.stream().map(goodsClient::findBrandDTObyId).collect(Collectors.toList
());
    resultMap.put("品牌", brandDTOS);
    resultMap.put("分类", categoryDTOS);
    return resultMap;
}

private List<Long> HandlerTermsAndMap(Terms brandAggTerms) {
    List<? extends Terms.Bucket> buckets = brandAggTerms.getBuckets();
    return
buckets.stream().map(Terms.Bucket::getKeyAsNumber).map(Number::longValue).collect(Collectors
.toList());
}
```

聚合搜索的API可以参考Kibana给我们返回的结构:当我们得到AggregatePage之后,可以使用getAggregations方法,得到Aggregations之后,我们可以通过使用get( )方法来获取Terms,在使用Terms的getBuckets来获取Term.Bucket,调用其中的getKeyAsNumber( )获取其中的key值,再使用Number类的longvalue方法来讲Number值转换成long值,然后再根据这些值去数据库总去查找,得到品牌和分类的名称,将其放在map总返回.

8.3.2.规格参数的聚合

首先我们需要思考什么情况下显示有关参数的过滤.如果用户尚未选择商品分类.或者聚合得到的份数大于1,那么就没必要进行规格参数的聚合,因为不同分类的商品,其规格是不同的,我们无法确定到底有多少规格需要聚合,代码无法进行.因此:我们在后台需要对聚合得到的商品分类数量进行判断,如果等于1,我们才继续进行规格参数的聚合.

总结步骤:

1. 用户搜索得到商品,并聚合出商品分类.
2. 判断分类数量是否等于1,如果是则进行规格参数的聚合.
3. 先根据分类,查找可以用来搜索的规格.
4. 在用户搜索结果的基础上,对规格参数进行聚合.
5. 将规格参数聚合结果整理进行返回.

```
private void handlerAllFilter(List<CategoryDTO> categoryDTOS, SearchPageRequest
searchPageRequest, Map<String, List<?>> resultMap) {
    //当分类category确定之后,即categoryDTOS的长度等于1的时候
    if (!CollectionUtils.isEmpty(categoryDTOS) && categoryDTOS.size() == 1) {
        Long id = categoryDTOS.get(0).getId();

        NativeSearchQueryBuilder queryBuilder = new NativeSearchQueryBuilder();
        NativeSearchQueryBuilder nativeSearchQueryBuilder =
queryBuilder.withQuery(basicQuery(searchPageRequest))
                .withSourceFilter(new FetchSourceFilter(null, null))
                .withPageable(PageRequest.of(0, 1));

        List<String> nameList = goodsClient.querySpecParamByGid(null, id, true)
                .stream().map(SpecParamDTO::getName).collect(Collectors.toList());
        for (String s : nameList) {
            nativeSearchQueryBuilder =
nativeSearchQueryBuilder.addAggregation(AggregationBuilders.terms(s).field("specs." + s));
```

```
        }
        NativeSearchQuery build = nativeSearchQueryBuilder.build();
        AggregatedPage<Goods> goods = elasticsearchTemplate.queryForPage(build,
Goods.class);
        Aggregations aggregations1 = goods.getAggregations();
        for (String s : nameList) {
            Terms aggregation = aggregations1.get(s);
            List<? extends Terms.Bucket> buckets = aggregation.getBuckets();
            List<String> collect =
buckets.stream().map(Terms.Bucket::getKeyAsString).collect(Collectors.toList());
            resultMap.put(s, collect);
        }
    }
}
```

8.4.过滤搜索

前端添加过滤条件,后端添加过滤条件.

之前我们的基本查询是:

```
private QueryBuilder basicQuery(SearchRequest request) {
    // 构建基本的match查询
    return QueryBuilders.matchQuery("all", request.getKey()).operator(Operator.AND);
}
```

现在我们需要将页面传递的过滤条件也传入进行,因此不能使用普通的查询,而是要用到BooleanQuery.

```
GET /heima/_search
{
    "query":{
        "bool":{
                "must":{ "match": { "title": "小米手机",operator:"and"}},
                "filter":{
                "terms":{"field": "specs.屏幕尺寸", "value": "5.0~5.5英寸"}
                }
        }
    }
}
```

我们对原来的基本查询进行改造.

```
private QueryBuilder basicQuery(SearchPageRequest searchPageRequest) {
        BoolQueryBuilder boolQueryBuilder = QueryBuilders.boolQuery().
                must(QueryBuilders.matchQuery("all", searchPageRequest.getKey())
                        .operator(Operator.AND));
        if (!CollectionUtils.isEmpty(searchPageRequest.getFilter())) {
            Map<String, String> filter = searchPageRequest.getFilter();
            Set<Map.Entry<String, String>> entries = filter.entrySet();
            for (Map.Entry<String, String> entry : entries) {
                String key = entry.getKey();
```

```
                if("品牌".equals(key)){
                    key = "brandId";
                }else if("分类".equals(key)){
                    key = "categoryId";
                }else{
                    key = "specs."+key;
                }
                String value = entry.getValue();
                boolQueryBuilder.filter(QueryBuilders.matchQuery(key,value));
            }
        }
        return boolQueryBuilder;
    }
```

9.页面静态化

查看商品详情,我们使用的是页面静态化.就是把本来需要动态渲染的页面提前渲染完成,生成静态的html,当用户访问时直接读取静态html,提高响应速度.

Thymeleaf:

- Context:运行上下文:用来保存模型数据,当模板引擎渲染时,可以从context上下文中获取数据用于渲染.当与SpringBoot结合使用,我们放入Model的数据就会被处理到Context,作为模板渲染的数据使用.
- TemplateResolver:模板解析器:用于读取模板相关的配置,例如:模板存放的位置信息,模板文件名称,模板文件的类型等等.当与SpringBoot结合时,TemplateResolver已经由其创建完成,并且各种配置也都有默认值,比如模板存放位置,其默认值就是:templates.比如模板文件类型,器默认值是html.
- TemplateEngine:模板引擎,需要解析模板的引擎,需要使用到上下文,模板解析器,分别从模板中需要的数据,模板文件,然后利用内置的语法规则解析,从而输出解析后的文件.

  `template.process("模板名",context,writer);`

在输出时候,我们可以指定输出的目的地,如果目的地是Response的流,那就是网络响应.如果目的地是本地文件,那就实现静态化了.而在SpringBoot已经自动配置了模板引擎,因此我们不需要关心这个.现在我们做静态化,就是把输出的目的地改成本地文件即可.

```
public Map<String, Object> toPage(Long spuId) {
        SpuDTO spu = goodsClient.querySpuBySpuId(spuId);
        Long cid3 = spu.getCid3();
        List<SpecGroupDTO> specs = goodsClient.querySpecParamBySpu(cid3);
        //封装数据
        Map<String, Object> data = new HashMap<>();
        data.put("categories", spu.getCategories());
        data.put("brand", spu.getBrandDTO());
        data.put("spuName", spu.getName());
        data.put("subTitle", spu.getSubTitle());
        data.put("skus", spu.getSkus());
        data.put("detail", spu.getSpuDetail());
        data.put("specs", specs);
        return data;
    }
    /**
```

```
     * 构建页面静态化
     */
    public void buidStaticPage(Long spuId) {
        Map<String, Object> map = toPage(spuId);
        Context context = new Context();
        context.setVariables(map);
        File fileDir = new File(itemDir);
        if (!fileDir.exists()) {
            fileDir.mkdir();
        }
        File file = new File(fileDir, spuId + ".html");
        try (PrintWriter printWriter = new PrintWriter(file, "UTF-8")) {
            springTemplateEngine.process(itemTemplate, context, printWriter);
        } catch (IOException e) {
            log.error("静态化失败");
            throw new LyException(ExceptionEnum.FILE_WRITER_ERROR);
        }
    }
```

这样我们可以使用搜索服务和静态页服务对外提供操作索引库和静态页接口,商品微服务在商品上下架后,调用接口.但是这存在着一个严重问题就是代码耦合,违背了微服务的独立原则,违背了开闭原则.我们可以通过MQ来解决.

10.RabbitMQ

AMQP(Advanced Message Queuing Protocol)

JMS(Java MesageService)

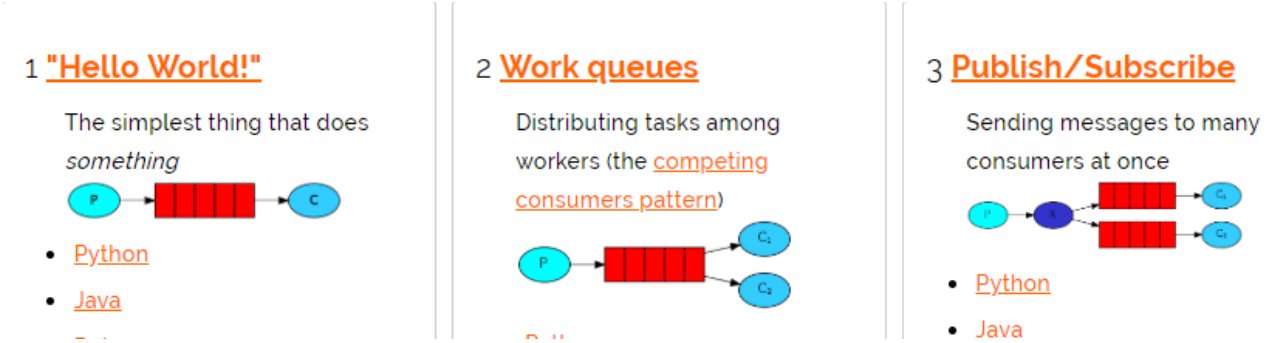
AMQP和JMS的区别和联系:

- JMS是定义了统一的接口,来对消息操作进行同一,AMQP是用过规定协议来统一数据交互.
- JMS限定了必须使用Java语言,AMQP只是协议,不规定时限方式,因此是跨语言的.
- JMS规定了两种消息模型,而AMQP的消息模型更加丰富.

常见的MQ产品:

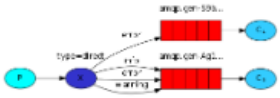
- ActiveMQ:基于JMS,Apache
- RabbitMQ:基于AMQP协议,erlang语言开发,稳定性好.
- RocketMQ:基于JMS,阿里巴巴产品,目前交由Apache基金会.
- Kafka:分布式消息系统,高吞吐量.

RabbitMQ提供了6种消息模型:



4 Routing

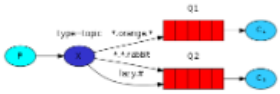
Receiving messages selectively



- Python

5 Topics

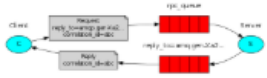
Receiving messages based on a pattern (topics)



- Python

6 RPC

Request/reply pattern example



- Python

Exchange:交换机,一方面,接收生产者发送的消息,知道如何处理消息,例如递交给某个特别的队列,递交给所有的队列等.到底如何操作,取决于Exchange的类型.Exchange有以下3种类型.

- Fanout:广播,将消息交给所有绑定到交换机的队列.
- Direct:定向,把消息交给符合指定的routing key的队列.
- Topic:通配符,把消息交给符合Routing pattern(路由模式)的队列.

Exchange只负责转发消息,不具备存储消息的能力,因此如果没有任何队列与Exchange绑定,或者没有符合路由规则的队列,那么消息会丢失.

> 面试题1:如果解决消息丢失?
>
> - ack(消费者确认):通过消息确认(Acknowlege)机制来实现,当消费者获取消息后,会向RabbitMQ发送执行ACK,告知消息已经被接收,不过这种ACK分两种情况:
> - 自动ACK:消息一旦被接收,消费者自动发送ACK
> - 手动ACK:消息接收后,不会发送ACK.需要手动调用
> - 持久化:我们需要将消息持久化到硬盘,防止服务宕机导致消息丢失.设置消息持久化,前提是:Queue,Exchange都持久化.即**Exchange,Queue,message持久化**
> - 生产者确认:生产者发送消息后,等待mq的ACK,如果没有收到或者收到失败消息,那么就会重试,如果收到成功消息那么业务结束.
> - 可靠消息服务:对于部分不支持生产者确认的消息队列,可以发送消息前,将消息持久化到数据库,并记录消息状态,后续消息发送,消费等过程都依赖于数据库中消息状态的判断和修改.
>
> 面试题2:如何避免消息堆积?
>
> - 通过同一队列多消费者监听,实现消息的争抢,加快消息消费速度
>
> 面试题3:如何保证消息的有序性?
>
> - 保证消息发送时候有序同步发送
> - 保证消息发送被同一队列接收
> - 保证一个队列只有一个消费者,可以有从机(待机状态),实现高可用
>
> 面试题4:如何保证避免消息重复消费?
>
> - 如果,你拿到这个消息做数据库的insert操作,那就容易了,给这个消息做一个唯一主键,那么就算出现了重复消费的情况,就会导致主键冲突.
> - 如果拿到这个消息做Redis的set操作,不用解决,无论set几次结果都是一样,set操作本来就是幂等操作.
> - 如果上述的情况不行,可以准备一个三方介质,来做消费记录,以Redis为例,给消费分配一个全局id,只要消费过该消息,将<id,Message>以K-V形式写入redis,那消费者开始消费前,先去redis中查询有没有消费记录即可.

11.Spring AMQP

需要注意的是AMQP会使用JDK序列化的方式进行处理,传输数据比较大,效率太低,我们可以自定义消息转换器,使用Json来进行处理.

```
@Configuration
public class ItemConfiguration {
    @Bean
    public Jackson2JsonMessageConverter jackson2JsonMessageConverter(){
        return new Jackson2JsonMessageConverter();
    }
}
```

当商品上下架的时候,将spuId发送到rabbitMQ

```
                /**
         * 再改变商品saleable的时候,发送rabbitmq
         */
        if (saleable == true) {
            /**
             * 我们可以在这边使用channel来使用rabbitmq的事务功能.
             * 这边有三个API
             * channel.txSelect();
             * channel.txRollback();
             * channel.txCommit();
             * 当然也可以在配置文件中配置:publisher-confirms:true,来确保生产者端的消息不会丢失.
             */
            amqpTemplate.convertAndSend(MQConstant.Exchange.ITEM,
MQConstant.RoutingKey.UP_ITEM, spuId);
        } else if (saleable == false) {
            amqpTemplate.convertAndSend(MQConstant.Exchange.ITEM,
MQConstant.RoutingKey.DOWN_ITEM, spuId);
        } else {

        }
```

页面静态化的微服务的监听

```
/**
 * 创建静态页面
 * @param spuId
 */
@RabbitListener(
        bindings = @QueueBinding(
                value = @Queue(name = MQConstant.Queue.STATIC_PAGE_UP, durable = "true"),
                exchange = @Exchange(name = MQConstant.Exchange.ITEM,type =
ExchangeTypes.DIRECT),
                key = MQConstant.RoutingKey.UP_ITEM
        )
)
public void listenerForBuildStaticPage(Long spuId) {
    if(spuId!=null){
        buidStaticPage(spuId);
```

```
    }
}
```

索引添加微服务的监听

```
        /**
     * 新建elsticSearch文档
     * @param spuId
     */
    @RabbitListener(
            bindings = @QueueBinding(
                    exchange = @Exchange(name = MQConstant.Exchange.ITEM,type =
ExchangeTypes.DIRECT),
                    value = @Queue(value = MQConstant.Queue.DOCUMENT_UP,durable = "true"),
                    key = MQConstant.RoutingKey.UP_ITEM
            )
    )
    public void amqpListenForCreateDocument(Long spuId){
        if(spuId!=null){
            SpuDTO spuDTO = goodsClient.querySpuBySpuId(spuId);
            Goods goods = buidGoods(spuDTO);
            goodsDao.save(goods);
        }
    }
```

12.Redis

12.1.Redis中常用的5种数据类型

- String:等同于java中的`Map<String,String>`
- list:等同于java中的`Map<String,List<String>>`
- set:等同于java中的`Map<String,Set<String>>`
- sort_set:可排序的set
- hash:等同于java中的:`Map<String,Map<String,String>>`

12.2.通用指令

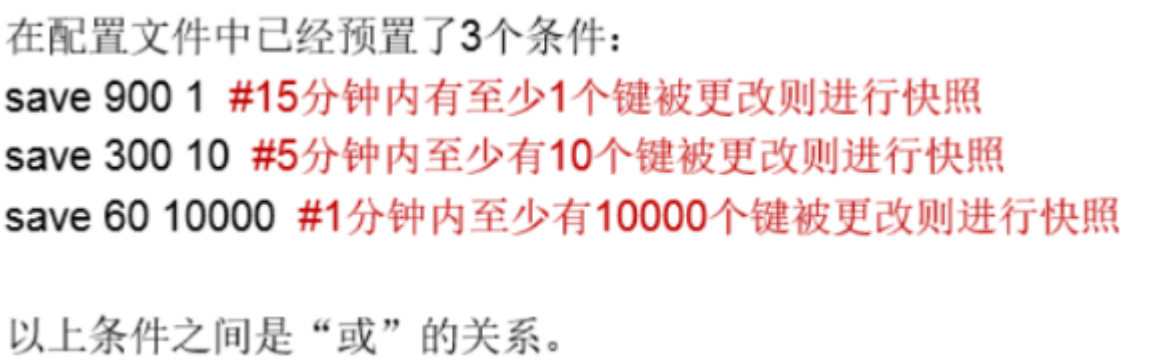
- `keys *`:查询所有的键
- `exists key`:判断一个键是否存在,如果存在返回整数1,否则返回0.
- `del key`:可以删除一个或多个key,返回值是删除key的个数
- `expire key seconds`:设置key的过期时间,超过时间后,将会自动删除该key,成功返回1,失败返回0.
- `TTL key`:设置一个key的过期时间:-1是永不过期,-2是已过期或者不存在
- `persist key`:移除给定key的生存时间,将这个key从带生存时间的key转换成一个不带生存时间,用不过期的key.当生存时间移除成功时,返回1,如果key不存在或key没有设置生存时间,返回0.

12.2.Redis的持久化

1.RDB(redis database)

redis全称是redis database,在指定的时间间隔内将内存中的数据集快照写入磁盘.

条件在redis.conf文件中配置,格式是`save (time) (count)`,表示的意思是在time(单位是秒)时间内,至少进行了count次修改后,触发条件,进行RDB快照.



基本原理:RDB的流程

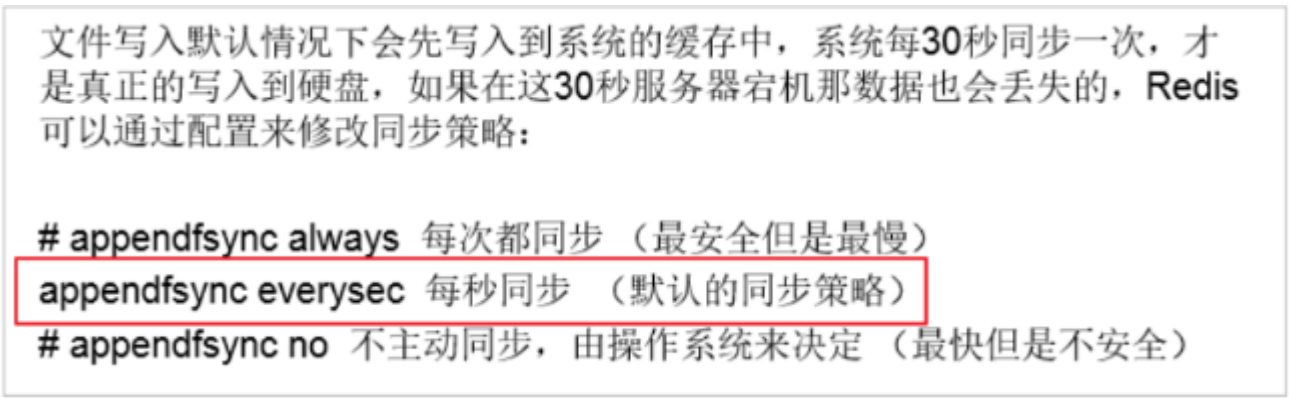
1. Redis使用fork函数来复制一份当前进程(父进程)的副本(子进程)
2. 父进程继续接收并处理请求,子进程开始把内存中的数据写入硬盘中的临时文件.
3. 子进程写完后,会使用临时文件代替旧的RDB文件

2.AOF(Append Only File)

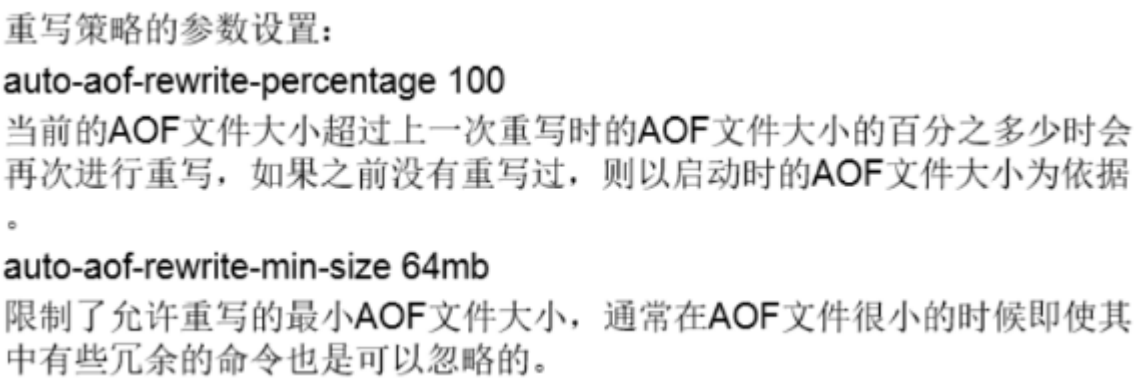
AOF方式默认是关闭的,需要修改配置来开启.

`appendonly yes # 把默认的no改成yes`

AOF持久化的策略是吧每一条接收到的命令都记录下来,每隔一定时间后,写入硬盘的AOF文件中,当服务器重启后,重新执行这些命令,即可恢复数据.



AOF文件重写,当记录命令过多,必然会出现对同一个key的多次写操作,此时只要记录最后一条即可,前面的记录都毫无意义.因此,当满足一定条件时候,Redis会对AOF文件进行重写,移除对同一个key的多次操作命令,保留最后一条,默认的触发条件;



13.Spring Data Redis

13.1.RedisTemplate

Spring Data 为Redis提供了一个模板对象:RedisTemplate.里面封装了对于Redis的五种数据结构的各种操作.

- `redisTemplate.opsForValue()` :操作字符串
- `redisTemplate.opsForHash()` :操作hash
- `redisTemplate.opsForList()` :操作list
- `redisTemplate.opsForSet()` :操作set
- `redisTemplate.opsForZset` :操作zSet

13.2.StringRedisTemplate

RedisTemplate在创建时,可以指定其泛型类型.

- K: 代表key的数据类型
- V: 代表value的数据类型

注意:这里的类型不是Redis中存储的数据类型,而是java中的数据类型,RedisTemplate会自动将Java类型转换为Redis支持的数据类型,字符串,字节,二进制等.

不过RedisTemplate默认会采用JDK自带的序列化来对对象进行转换,生成的数据十分庞大,因此一般我们都会指定key和value为String类型,这样就由我么自己把对象序列化为json字符串来存储即可.因为大部分情况下,我们都会使用key和value都会String的RedisTemplate,因此Spring就默认提供了这样一个实现,StringRedisTemplate,因此Spring就默认提供了这样一个实现.

14.阿里云短信微服务

因为短信发送API调用时长的不确定性,为了提高程序的响应速度,短信发送我们都将采用异步发送的方式.

我们建立一个amqp的监听器,只要接收到发送消息的message,就调用阿里云的发送消息方法.

```
@Service
@Slf4j
public class SMSmq {
    @Autowired
    private SMSUtils smsUtils;
```

```
    @RabbitListener(
            bindings = @QueueBinding(
                    exchange = @Exchange(name = MQConstant.Exchange.ITEM,type =
ExchangeTypes.DIRECT),
                    value = @Queue(name = MQConstant.Queue.SEND_MESSAGE,durable = "true"),
                    key =MQConstant.RoutingKey.send_message
            )
    )
    public void shortMessageListener(Map<String,String> map){
        try {
            String phoneNumber = map.get("phoneNumber");
            String templateCode = map.get("templateCode");
            String templateParam = map.get("templateParam");
            smsUtils.sendSortMessage(phoneNumber,templateCode,templateParam);
        } catch (Exception e) {
            e.printStackTrace();
            //在这里捕获异常,防止生产者重复发送
            log.error("消息发送失败");
            throw new LyException(ExceptionEnum.SEND_MESSAGE_ERROR);
        }
    }
}
```

15.用户中心

15.1.验证码的发送.

对于短信验证功能的业务逻辑:

- 接收页面发送来的手机号码
- 生成一个随机验证码
- 将验证码保存在Redis,可以使用Redis的过期机制来保存,一般保存5分钟
- 发送短信,将验证码发送到用户手机.

```
@Transactional
public void code(String phone) {
    String randomStr = RandomStringUtils.randomNumeric(4);
    log.info(randomStr);
    HashMap<String, String> stringStringHashMap = new HashMap<>();
    stringStringHashMap.put("code",randomStr);
    String s = JsonUtils.toString(stringStringHashMap);
    HashMap<String, String> map = new HashMap<>();
    map.put("phoneNumber",phone);
    map.put("templateCode","SMS_172013390");
    map.put("templateParam",s);

amqpTemplate.convertAndSend(MQConstant.Exchange.ITEM,MQConstant.RoutingKey.send_message,map)
;
    //将数据存入redis,设置过期时间为30分钟
```

```
    stringRedisTemplate.opsForValue().set(prefix+phone,randomStr,30, TimeUnit.MINUTES);
}
```

15.2.用户的注册

用户注册的业务逻辑:

- 效验短信验证码.
- 对密码加密
- 写入数据库

密码加密使用的传统的MD5加密并不安全,这里我么使用的是Spring提供的BCryptPasswordRncoder加密算法,分为加密和验证两个过程.

- 加密:算法会对明文密码随机生成一个salt,使用salt结合密码来加密,得到最终的密文.
- 验证密码:需要先拿到加密后的密码和要验证的密码,根据已加密的密码来推测出salt,然后利用相同的算法和salt对要验证的密码进行加密,与已加密的密码对比即可.为了防止有人能根据秘闻推测出salt,我们需要在使用BCryptPasswordEncoder时配置随机秘钥.

```
 */
@Data
@Component
@ConfigurationProperties("ly.encoder.crypt")
public class PasswordEncoderConfig {

    private int strength;
    private String secret;

    @Bean
    public BCryptPasswordEncoder passwordEncoder(){
        SecureRandom random = new SecureRandom(secret.getBytes());
        return new BCryptPasswordEncoder(strength, random);
    }
}
```

我们先将BCryptPasswordEncoder注入进来,BCryptPasswordEncoder,BCryptPasswordEncoder需要两个参数.一个是加密强度4-31,决定了密码和盐加密时的运算次数.一个是通过UUID生成的随机秘钥.

```
        @Transactional
    public void register(User user, String code) {
        Long phone = user.getPhone();
        String s = stringRedisTemplate.opsForValue().get(prefix + phone);
        if(StringUtils.isNotBlank(s)&&s.equals(code)){
            user.setPassword(passwordEncoder.encode(user.getPassword()));
            int insert = userMapper.insertSelective(user);
            if(insert!=1){
                throw new LyException(ExceptionEnum.INSERT_OPERATION_FAIL);
            }
            stringRedisTemplate.delete(prefix+phone);
            return;
        }
```

```
        throw new LyException(ExceptionEnum.INVALID_USERNAME_PASSWORD);
    }
```

虽然实现了注册,服务端并没有进行数据校验,我们这里使用Hibernate-Validator提供的一个开源框架,使用注解方式非常方便实现服务端的数据效验.Hibernate-Validator提供了JSR303规范中所有内置constraint(约束)的实现,除此之外还有一些附加的constraint.在日常开发中,Hibernate Validator经常用来验证bean的字段,基于注解,方便快捷高效.

```
@Data
@Table(name="tb_user")
public class User {
    @Id
    @KeySql(useGeneratedKeys = true)
    private Long id;
    @Pattern(regexp = RegexPatterns.USERNAME_REGEX,message = "非法的用户名")
    private String username;
    @Length(min = 4,max = 8,message = "密码长度错误")
    private String password;
    //@Pattern(regexp = RegexPatterns.PHONE_REGEX,message = "非法的手机格式")
    private Long phone;
    private Date createTime;
    private Date updateTime;
}
```

在controller层添加@Valid注解,SpringMVC会自动返回错误信息.但返回的数据不是我们想要的格式.

我们在User参数后面跟一个BindingResult参数,不管效验是否通过,就会进入方法内部,BindingResult中会封装错误结果,我们通过result.hasErrors来判断是否有错误,然后通过result.getErrors来获取错误信息.

```
        @PostMapping("/register")
    public ResponseEntity<Void> register(@Validated User user, BindingResult result, String
code) {
        if (result.hasErrors()) {
            String collect =
result.getFieldErrors().stream().map(FieldError::getDefaultMessage).collect(Collectors.joini
ng("|"));
            throw new LyException(400, collect);
        }
        userService.register(user, code);
        return ResponseEntity.status(HttpStatus.CREATED).body(null);
    }
```

15.3. 用户的登陆

```
public UserDTO queryByNameAndPassword(String username, String password) {
    User user = new User();
    user.setUsername(username);
    User user1 = userMapper.selectOne(user);
    if(user1==null){
        throw new LyException(ExceptionEnum.INVALID_USERNAME_PASSWORD);
    }
    if(passwordEncoder.matches(password, user.getPassword())){
        throw new LyException(ExceptionEnum.INVALID_USERNAME_PASSWORD);
    }
    return BeanHelper.copyProperties(user1,UserDTO.class);
}
```

使用passwordEncoder.match方法进行匹配

16.无状态登陆

16.1.有状态登陆

有状态服务,即服务端需要记录每次会话的客户端,从而识别客户端身份,根据用户身份进行请求的处理,典型的设计如tomcat中的session.但是有很多的缺点:

- 服务器保存大量数据,增加服务端压力.
- 服务端保存用户状态,无法水平扩展.
- 客户端请求依赖服务器,多次请求必须访问同一台服务器.

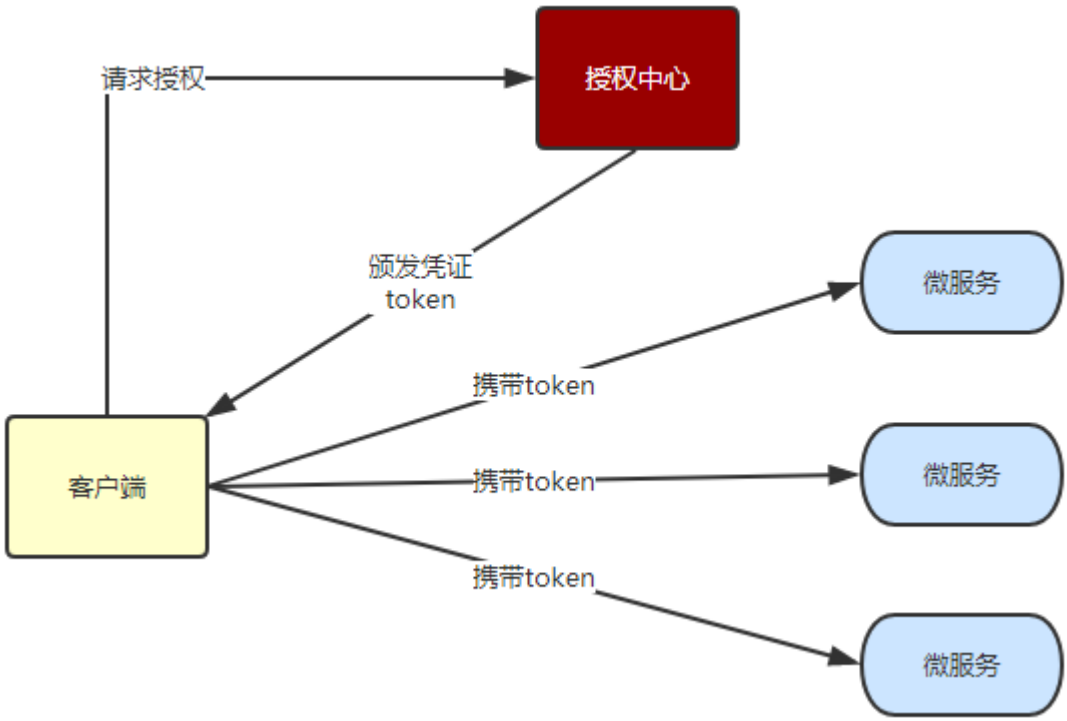
16.2.无状态登陆

优点是:

- 客户端请求不依赖服务端的信息,任何多次请求不必要访问到同一台服务.
- 服务端的集群和状态对客户端透明.
- 服务端可以任意的迁移和伸缩.
- 减小服务端存储压力.

无状态登陆的流程:

- 当客户端第一次请求服务时,服务端对用户进行信息认证
- 认证通过,将用户信息进行加密形成token,返回给客户端,作为登陆凭证.
- 以后每次请求,客户端都携带认证的token
- 服务的对token进行解密.判断是否有效



16.3.JWT

JWT(Json Web Token)是JSON风格的轻量级的授权和身份认证规范,可实现无状态,分布式的应用授权.

数据格式:

- Header:头部,通常头部有两部分信息.(我们会对头部进行base64加密,得到第一部分数据)
  - 声明类型:这里是JWT
  - 签名算法:自定义
- PayLoad:载荷,就是有效数据,一般包含下面数据
  - 用户身份信息(注意:这里因为采用base64加密,可解密,因此不要存放敏感信息)
  - tokenID:当前这个JWT的唯一标识
  - 注册声明:如token的签发时间,过期时间,签发时间
- Signature:qi签名,是整个数据的认证信息,一般根据前两步的数据,再加上服务的密钥(secret),通过加密算法生成,用于验证整个数据的完整性.
-

授权流程:

- 用户请求登陆,携带用户名密码到授权中心
- 授权中心携带用户名密码,到用户中心查询用户.
- 查询如果正确,生成JWT凭证
- 返回JWT给用户

鉴权流程:

- 用户请求某微服务功能,携带JWT
- 微服务将JWT交给授权中心效验
- 授权中心返回效验结果到微服务.
- 微服务判断效验结果,成功或失败.
- 失败返回401,成功则处理业务并返回

但是现在每次微服务都需要鉴权,那么每次鉴权都需要访问授权中心,如果把密钥交给其他微服务就存在安全风险.

16.4.RSA

基本原理:同时生成两把密钥,私钥和公钥,公钥可以下发给信任客户端

- 私钥加密,持有私钥或公钥才可以解密
- 公钥加密,持有私钥才可解密
- 在微服务利用公钥验证签证签名有效性.

17.授权中心

授权中心的主要职责是

- 用户登录鉴权:
  - 接收用户的登陆请求.
  - 通过用户中心的接口效验用户名密码

    - 使用私钥生成JWT并返回
- 用户登录状态的效验:
    - 判断用户是否登录,就是token的效验.
- 用户登出
    - 用户选择退出登陆后,需要让token失效
- 用户登录状态刷新
    - 用户登录一段时间后,JWT可能过期,需要刷新有效期

17.1.登陆功能

```
@Data
@Component
@Slf4j
@ConfigurationProperties("ly.jwt")
public class JwtProperties implements InitializingBean {

    private String pubKeyPath;
    private String priKeyPath;

    private PublicKey publicKey;
    private PrivateKey privateKey;

    private UserProperties user = new UserProperties();

    @Override
    public void afterPropertiesSet() throws Exception {
        try {
            this.publicKey = RsaUtils.getPublicKey(pubKeyPath);
            this.privateKey = RsaUtils.getPrivateKey(priKeyPath);
        } catch (Exception e) {
            log.error("认证微服务初始化失败!");
            e.printStackTrace();
        }
    }

    /**
     * expireTime: 30 #设置token的过期时间,单位是分钟
     * cookieName: LY_TOKEN #设置cookie的名称
     * cookieDomain: leyou.com # cookie的域
     */
    @Data
    public class UserProperties {
        private Integer expireTime;
        private String cookieName;
        private String cookieDomain;
        private Integer minRefreshMinutes;
    }
}
```

加载公钥和私钥.这个属性类只帮我们读取了公钥和私钥地址,那么每次使用公钥我们都需要从硬盘读取,效率很低.加载公钥和私钥的代码应该写在哪里呢?构造函数可以吗?不行.因为构造函数执行时候,Spring还没有完成属性注入,此时公钥地址和私钥地址都没有,我们必须在Spring完成属性初始化之后再加载密钥.

17.2.JWT客户端存储方案

- 目前有两种解决方案：
  - 方案一：存入web存储如：LocalStorage或SessionStorage中
    - 优点：
      - 不用担心cookie禁用问题
      - 不会随着浏览器自动发送，可以减少不必要的请求头大小
    - 缺点：
      - 不会随着浏览器自动发送，需要前段额外代码，携带jwt
      - 会遭到XSS（跨站脚本）攻击
  - 方案二：存入cookie
    - 优点：
      - 会随着浏览器自动发送，客户端不用任何额外代码
      - 使用httponly，避免XSS攻击风险
    - 缺点：
      - 会随着浏览器自动发送，某些时候有些多余
      - 可能遭到CSRF（跨站资源访问）攻击
- 我们采用cookie方案，cookie方案的两个缺陷我们也可以解决：
  - 问题1：会随着浏览器自动发送，某些时候有些多余
    - 解决：后端服务与其它服务资源（如静态资源）采用不同域名，浏览器的同源策略会限制cookie
  - 问题2：可能遭到CSRF（跨站资源访问）攻击
    - 解决：避免get请求操作服务器资源，遵循Rest风格，必要时在token中存入随机码

```
public void login(String username, String password, HttpServletResponse response) {
        UserDTO userDTO = userClient.queryByNameAndPassword(username, password);
        if(userDTO == null){
            throw new LyException(ExceptionEnum.INVALID_USERNAME_PASSWORD);
        }
        UserInfo userInfo = BeanHelper.copyProperties(userDTO, UserInfo.class);
        userInfo.setRole(USER_ROLE);
        String token = JwtUtils.generateTokenExpireInMinutes(userInfo,
jwtProperties.getPrivateKey(), jwtProperties.getUser().getExpireTime());
        CookieUtils.newCookieBuilder()
                .response(response)
                .domain(jwtProperties.getUser().getCookieDomain())
                .httpOnly(true) //是否允许脚本修改cookie,保证安全防止XSS攻击,不允许JS操作Cookie
                .name(jwtProperties.getUser().getCookieName())
                .value(token)
                .build();
    }
```

在登陆成功的同时,根据根据私钥和用户信息(用户信息不能够包含用户的密码),还有过期时间生成token.然而这种方式将token存放在cookie中,Zuul内部会有默认的过滤器,会对请求和响应头信息进行重组,过滤掉敏感头信息.

Zuul内部有默认的过滤器,会对请求和响应头信息进行重组,过滤掉敏感的头信息:

```
@Override
public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    final String requestURI = this.urlPathHelper.getPathWithinApplication(ctx.getRequest());
    Route route = this.routeLocator.getMatchingRoute(requestURI);
    if (route != null) {
        String location = route.getLocation();
        if (location != null) {
            ctx.put(REQUEST_URI_KEY, route.getPath());
            ctx.put(PROXY_KEY, route.getId());
            if (!route.isCustomSensitiveHeaders()) {
                this.proxyRequestHelper
                        .addIgnoredHeaders(this.properties.getSensitiveHeaders().toArray(new String[0]));
            }
            else {
                this.proxyRequestHelper.addIgnoredHeaders(route.getSensitiveHeaders().toArray(new String[0]));
            }

            if (route.getRetryable() != null) {
                ctx.put(RETRYABLE_KEY, route.getRetryable());
            }

            if (location.startsWith(HTTP_SCHEME+":") || location.startsWith(HTTPS_SCHEME+":")) {
                ctx.setRouteHost(getUrl(location));
                ctx.addOriginResponseHeader(SERVICE_HEADER, location);
```

而这个SensitiveHeaders是ZuulProeperties的类,默认值就包含set-cookie,cookie,Authorization.

```
    */
    private Set<String> sensitiveHeaders = new LinkedHashSet<>(
            Arrays.asList("Cookie", "Set-Cookie", "Authorization"));

    /**
```

在配置文件中配置sensitiveHeaders为空,就可以将token存放到客户端的cookie中.

17.3.登陆状态和刷新Token

步骤分析:

- 页面向后台发起请求,携带cookie
- 后台获取cookie中的token
- 沿着token是否有效
    - 有效:解析出里面的用户信息,返回到页面
    - 无效:登陆失效

目前有一定的问题:JWT内部设置了token的有效期,默认是30分钟,30分钟后用户的登陆信息无效了.而且JWT是生成后无法更改,因此我们无法修改token中的有效期.我们可以在cookie即将到期时候,重新生成一个token,比如token的有效期为30分钟,当用户请求我们时候,我们可以判断如果用户的token有效期还剩下15分钟,那么就重新生成token.

```
public UserInfo verfiy(HttpServletRequest request, HttpServletResponse response) {

    try {
        String cookieValue = CookieUtils.getCookieValue(request,
jwtProperties.getUser().getCookieName());
        Payload<UserInfo> payload = JwtUtils.getInfoFromToken(cookieValue,
```

```
jwtProperties.getPublicKey(), UserInfo.class);
        //每次在执行verfiy的时候,都需要去redis中去查询,如果存在,说明token已经无效,抛出异常
        Boolean aBoolean = stringRedisTemplate.hasKey(payload.getId());
        if(aBoolean){
            throw new LyException(ExceptionEnum.UNAUTHORIZED);
        }
        Date expiration = payload.getExpiration();
        DateTime dateTime = new
DateTime(expiration).minusMinutes(jwtProperties.getUser().getMinRefreshMinutes());
        if(DateTime.now().isAfter(dateTime.getMillis())){
            String tokenNew = JwtUtils.generateTokenExpireInMinutes(payload.getUserInfo(),
jwtProperties.getPrivateKey(), jwtProperties.getUser().getExpireTime());
            CookieUtils.newCookieBuilder()
                    .response(response)
                    .domain(jwtProperties.getUser().getCookieDomain())
                    .httpOnly(true) //是否允许脚本修改cookie
                    .name(jwtProperties.getUser().getCookieName())
                    .value(tokenNew)
                    .build();
        }

        return payload.getUserInfo();
    } catch (Exception e) {
        e.printStackTrace();
        throw new LyException(ExceptionEnum.UNAUTHORIZED);
    }
}
```

17.4.注销登陆

因为我们设置了httponly,js是无法操作cookie的,删除cookie必须发起请求到服务端,由服务端来删除cookie.如果只是让用在当前浏览器上的token删除了,但是这个token依然是有效的,JWT是无法控制Token让其失效,如果提前备份token,重新填写到cookie后,登陆状态依然有效.所以,我们**不仅仅要让浏览器清除cookie,还要让这个cookie中的token失效**.

- 用户注销类型操作时候,验证token的有效性,并解析token信息.
- 把token的id存入redis,并设置有效期为token的剩余有效期
- 校验用户登陆状态的接口,除了要正常逻辑外,还必须判断token的id是否存在于redis
- 如果存在,则证明token无法,如果不存在,则证明有效.

等于是在Redis中记录失效token的黑名单,黑名单的时间不用太长,最长也就是token的有效期,30分钟,因此服务端数据存储压力会减小.

```
 public void logout(HttpServletRequest request, HttpServletResponse response) {
        //需要通过request获取cookie,从cookie中获取到token信息.
        String cookieValue = CookieUtils.getCookieValue(request,
jwtProperties.getUser().getCookieName());
        Payload<UserInfo> payload = JwtUtils.getInfoFromToken(cookieValue,
jwtProperties.getPublicKey(), UserInfo.class);
        //如果过期时间超过5秒,直接将用用户的信息存放在Redis中
        long timeLeft = payload.getExpiration().getTime() - new Date().getTime();
        if(timeLeft>=5000){

stringRedisTemplate.opsForValue().set(payload.getId(),payload.getUserInfo().toString(),timeL
eft, TimeUnit.MILLISECONDS);
        }
        //将用户的cookie中的token进行清除

CookieUtils.deleteCookie(jwtProperties.getUser().getCookieName(),jwtProperties.getUser().get
CookieDomain(),response);
    }
```

18.服务鉴权

权限控制的基本流程如下:

- 获取用户的登陆凭证jwt
- 解析jwt,获取用户身份
  - 如果解析失败,证明没有登陆,返回401
  - 如果解析成功,继续向下执行
- 根据身份,查询用户权限信息.
- 获取当前请求资源(通过微服务的接口路径)
- 判断是否有访问资源的权限.

```
@Slf4j
@Component
@EnableConfigurationProperties({JwtProperties.class, FilterProperties.class})
public class AuthFilter extends ZuulFilter {

    @Autowired
    private JwtProperties jwtProp;

    @Autowired
    private FilterProperties filterProp;

    @Override
    public String filterType() {
        return FilterConstants.PRE_TYPE;
    }

    @Override
    public int filterOrder() {
```

```
        return FilterConstants.FORM_BODY_WRAPPER_FILTER_ORDER - 1;
    }

    @Override
    public boolean shouldFilter() {
        // 获取上下文
        RequestContext ctx = RequestContext.getCurrentContext();
        // 获取request
        HttpServletRequest req = ctx.getRequest();
        // 获取路径
        String requestURI = req.getRequestURI();
        // 判断白名单
        return !isAllowPath(requestURI);
    }

    private boolean isAllowPath(String requestURI) {
        // 定义一个标记
        boolean flag = false;
        // 遍历允许访问的路径
        for (String path : this.filterProp.getAllowPaths()) {
            // 然后判断是否是符合
            if(requestURI.startsWith(path)){
                flag = true;
                break;
            }
        }
        return flag;
    }

    @Override
    public Object run() throws ZuulException {
        // 获取上下文
        RequestContext ctx = RequestContext.getCurrentContext();
        // 获取request
        HttpServletRequest request = ctx.getRequest();
        // 获取token
        String token = CookieUtils.getCookieValue(request,
jwtProp.getUser().getCookieName());
        // 校验
        try {
            // 解析token
            Payload<UserInfo> payload = JwtUtils.getInfoFromToken(token,
jwtProp.getPublicKey(), UserInfo.class);
            // 解析没有问题，获取用户
            UserInfo user = payload.getUserInfo();
            // 获取用户角色，查询权限
            String role = user.getRole();
            // 获取当前资源路径
            String path = request.getRequestURI();
            String method = request.getMethod();
            // TODO 判断权限，此处暂时空置，等待权限服务完成后补充
            log.info("【网关】用户{},角色{}。访问服务{} : {}，", user.getUsername(), role,
method, path);
```

```
        } catch (Exception e) {
            // 校验出现异常，返回403
            ctx.setSendZuulResponse(false);
            ctx.setResponseStatusCode(403);
            log.error("非法访问，未登录，地址：{}", request.getRemoteHost(), e );
        }
        return null;
    }
}
```

- 你们使用JWT做登录凭证，如何解决token注销问题

答：jwt的缺陷是token生成后无法修改，因此无法让token失效。只能采用其它方案来弥补，基本思路如下：

```
1）适当减短token有效期，让token尽快失效

2）删除客户端cookie

3）服务端对失效token进行标记，形成黑名单，虽然有违无状态特性，但是因为token有效期短，因此标记
时间也比较短。服务器压力会比较小
```

- 既然token有效期短，怎么解决token失效后的续签问题？

答：在验证用户登录状态的代码中，添加一段逻辑：判断cookie即将到期时，重新生成一个token。比如token有效期为30分钟，当用户请求我们时，我们可以判断如果用户的token有效期还剩下10分钟，那么就重新生成token。因此用户只要在操作我们的网站，就会续签token

- 如何解决异地登录问题？

答：在我们的应用中是允许用户异地登录的。如果要禁止用户异地登录，只能采用有状态方式，在服务端记录登录用户的信息，并且判断用户已经登录，并且在其它设备再次登录时，禁止登录请求，并要求发送短信验证。

- 如何解决cookie被盗用问题？

答：cookie被盗用的可能性主要包括下面几种：

- XSS攻击：这个可以再前端页面渲染时对 数据做安全处理即可，而且我们的cookie使用了Httponly为true，可以防止JS脚本的攻击。
- CSRF攻击：
  - 我们严格遵循了Rest风格，CSRF只能发起Get请求，不会对服务端造成损失，可以有效防止CSRF攻击
  - 利用Referer头，防盗链
- 抓包，获取用户cookie：我们采用了HTTPS协议通信，无法获取请求的任何数据
- 请求重放攻击：对于普通用户的请求没有对请求重放做防御，而是对部分业务做好了幂等处理。运行管理系统中会对token添加随机码，认证token一次有效，来预防请求重放攻击。
- 用户电脑中毒：这个无法防范。
- 如何解决cookie被篡改问题？
- 答：cookie可以篡改，但是签名无法篡改，否则服务端认证根本不会通过
- 如何完成权限校验的？

> - 首先我们有权限管理的服务，管理用户的各种权限，及可访问路径等
> - 在网关zuul中利用Pre过滤器，拦截一切请求，在过滤器中，解析jwt，获取用户身份，查询用户权限，判断用户身份可以访问当前路径
> - 服务端微服务地址不小心暴露了，用户就可以绕过网关，直接访问微服务，怎么办?
> - 答：我们的微服务都做了严格的服务间鉴权处理，任何对微服务的访问都会被验证是否有授权，如果没有则会被拦截。具体实现：此处省略500字，见本节课内容

19.SpringCloud Config

SpringCloud中,有分布式配置中心组件SpringCloud,有分布式配置中心组件SpringCloud Config,分为两个角色,一是config server,二是config client.

- config server是一个可横向扩展,集中式的配置服务器,它用于集中管理应用程序各个环境下的配置,默认使用Git存储配置文件内容,也可以使用SVN存储,或者式本地文件存储.
- config client式Config Server的客户端,用于操作存储在Config Server中的配置内容.微服务在启动时会请求Config Server获取配置文件的内容,请求到后再启动容器.

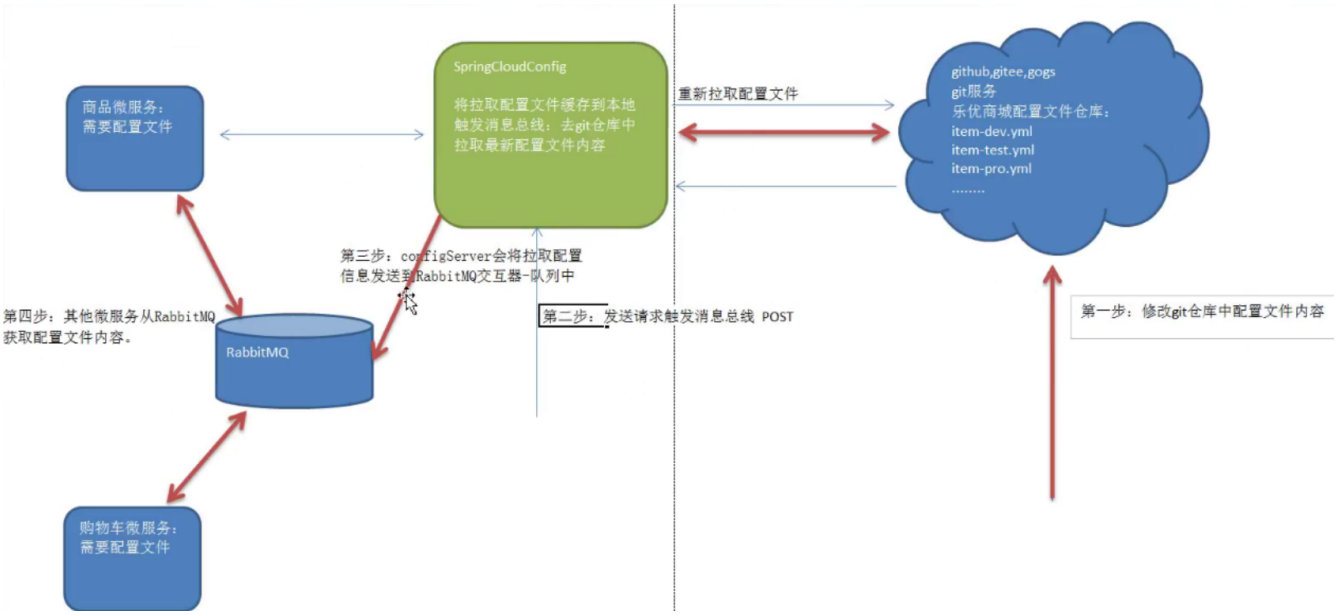
配置中心微服务

```
spring:
  application:
    name: config-service
  cloud:
    config:
      server:
        git:
          uri: https://gitee.com/fechinchu/springcloud-config.git
          username: fechinchu
          password: 320512Grant
```

配置客户端:我们需要将配置文件上传到Gitee,并修改文件名称为{application}-{profile}.yml或{application}-{profile}.properties.当然我们也需要使用在本地微服务使用bootstrap.yml文件

```
spring:
  cloud:
    config:
      name: item
      profile: dev
      label: master
      uri: http://127.0.0.1:12000
```

20.SpringCloud Bus



我们需要修改application.yml,添加rabbitMQ的配置,并暴露触发消息总线的地址

```
rabbitmq:
    host: 127.0.0.1
    port: 5672
    virtual-host: /leyou
    username: fechin
    password: 320512
server:
  port: 12000

management: #暴露触发消息总线的地址
  endpoints:
    web:
      exposure:
        include: bus-refresh
#postman进行测试,url:http://localhost:12000/actuator/bus-refresh method:post
```

这样我们可以使用postman对暴露的地址进行测试,需要使用post方法.

如果我们一个类上需要刷新数据,那么则需要`@RefreshScope`注解.

21.购物车

当用户未登录的时候,应该把数据保存在客户端,这样每个用户保存自己的数据.用户登录后,数据就应该保存在redis中,因为读和写的压力都比较大.我们可以限制购物车最多只能添加商品的数目,这样就能不用担心Redis存储空间问题.

21.1.未登陆购物车

- localStorage -- 没有时间限制的数据存储
- sessionStorage -- 针对一个session的数据存储,当用户关闭浏览器窗口后,数据会被删除.

21.2.已登陆购物车

购物车中的每个业务都需要获取当前登陆的用户信息,我们可以利用AOP的思想,拦截每一个进入controller的请求,统一完成登陆用户的获取.用户的信息获取之后,我们可以将用户的信息存放在**ThreadLocal**中,在并发请求情况下,因为每次请求都有不同的用户信息,我么必须保证每次请求保存的用户信息互相不干扰,线程独立.

我们创建一个工具类

```
public class UserInfoOnThreadLocal {
    private static ThreadLocal<Long> threadLocal = new ThreadLocal<>();

    public static void setUserId(Long id) {
        threadLocal.set(id);
    }

    public static Long getUserId() {
        return threadLocal.get();
    }

    public static void removeId() {
        threadLocal.remove();
    }
}
```

我们在购物车微服务中编写拦截器,将用户的信息存放在ThreadLocal中

```
@Component
public class UserInterceptor implements HandlerInterceptor {
    private final static String COOKIE_NAME = "LY_TOKEN";
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        try {
            String cookieValue = CookieUtils.getCookieValue(request, COOKIE_NAME);
            Payload<UserInfo> infoFromToken = JwtUtils.getInfoFromToken(cookieValue,
UserInfo.class);
            Long id = infoFromToken.getUserInfo().getId();
            UserInfoOnThreadLocal.setUserId(id);
            return true;
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
            return false;
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response,
Object handler, Exception ex) throws Exception {
        UserInfoOnThreadLocal.removeId();
    }
}
```

当然,我们需要对拦截器进行配置

```
@Configuration
public class CartConfiguration implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new UserInterceptor()).addPathPatterns("/**");
    }
}
```

对于购物车,因为增删改频率很高,所以我们需要将数据存放在redis中,Redis中的数据结构有五种,由于我们对购物车中商品的增删改操作,基本需要根据商品id进行判断,为了方便后期处理,我们的购物车也应该是k-v结构,所以我们选择Hash的数据结构.

添加商品到购物车:先查询之前的购物车数据,判断要添加的商品是否存在.如果存在,则直接修改数量后写回redis.如果不存在,新建一条数据,然后写入Redis.

```
/**
     * 在新增商品之前需要去redis中查询该商品是否存在
     * @param cart
     */
    public void addCart(Cart cart) {
        String userId = USER_ID_PREFIX+UserInfoOnThreadLocal.getUserId();
        BoundHashOperations<String, String, String> hashOps =
redisTemplate.boundHashOps(userId);
        //如果reids中存在商品信息,那么需要将商品取出来,并进行数量上的增加
        if(hashOps.hasKey(cart.getSkuId().toString())){
            String JsonOfCart = hashOps.get(cart.getSkuId().toString());
            Cart cart1 = JsonUtils.toBean(JsonOfCart, Cart.class);
            Integer preNum = cart1.getNum();
            cart1.setNum(preNum+cart.getNum());
            //将cart1重新存进reids;
            hashOps.put(cart.getSkuId().toString(),JsonUtils.toString(cart1));
            return;
        }
        //如果reids中并不存在商品的信息,直接增加
        hashOps.put(cart.getSkuId().toString(),JsonUtils.toString(cart));
    }
```

查看商品到购物车:

```
/**抽取出来的方法*/
private BoundHashOperations<String, String, String> getBoundHashOpertion(){
        //去redis中查询商品的信息
        String userId = USER_ID_PREFIX+UserInfoOnThreadLocal.getUserId();
        //根据key进行查询,如果redis中没有key,那么直接报错
        if(!redisTemplate.hasKey(userId)){
            throw new LyException(ExceptionEnum.GOODS_NOT_FOUND);
        }
        BoundHashOperations<String, String, String> hashOps =
redisTemplate.boundHashOps(userId);
        return hashOps;
    }
```

```
public List<Cart> list() {
        BoundHashOperations<String, String, String> boundHashOpertion =
getBoundHashOpertion();
        List<String> values = boundHashOpertion.values();
        List<Cart> collect = values.stream().map(JsonString -> JsonUtils.toBean(JsonString,
Cart.class)).collect(Collectors.toList());
        return collect;
    }
```

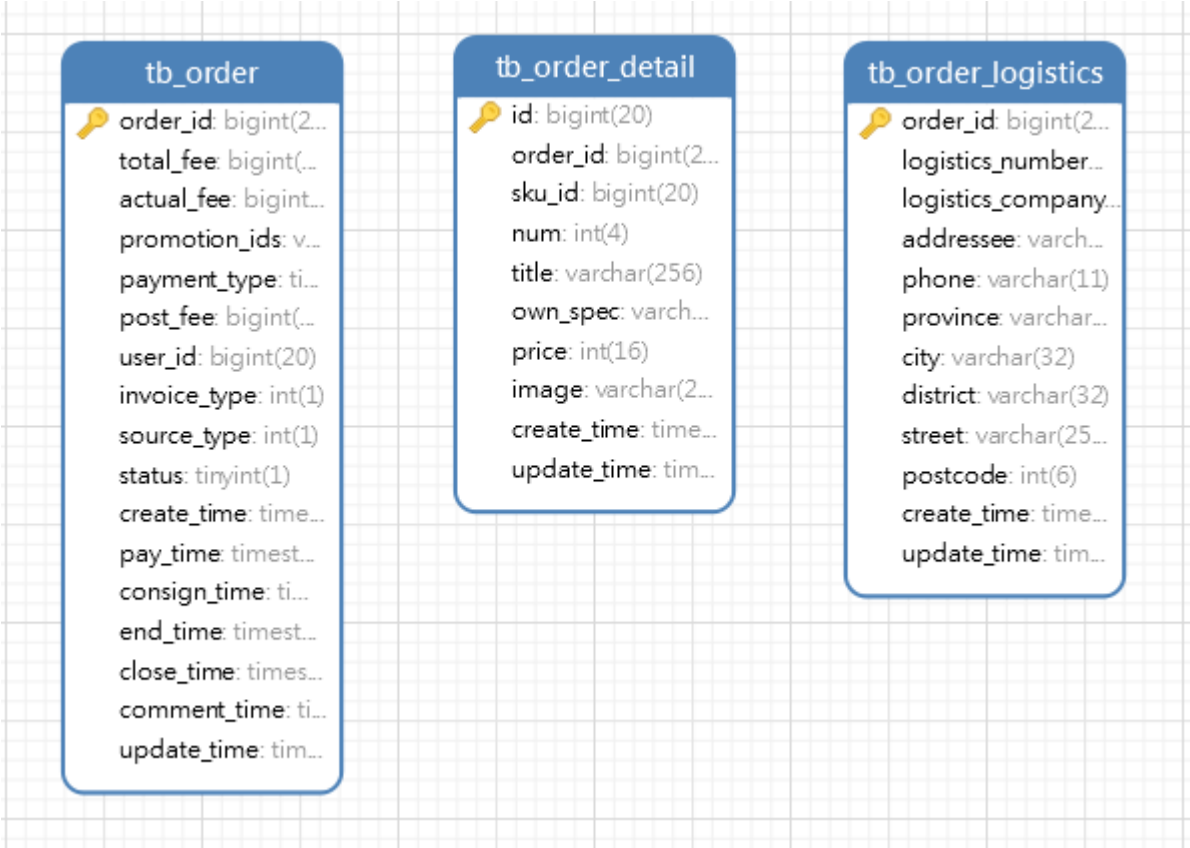
修改购物车商品数量(略)

删除购物车商品数量(略)

登陆后购物车合并

```
/**
     * 对local storage与redis上的数据进行合并
     * @param list
     */
    public void merge(List<Cart> list) {
        BoundHashOperations<String, String, String>  boundHashOpertion =
getBoundHashOpertion();
        for (Cart cart : list) {
            //如果redis中包含该数据,那么需要对数量进行更改
            if(boundHashOpertion.hasKey(cart.getSkuId().toString())){
                String cartOnJson = boundHashOpertion.get(cart.getSkuId().toString());
                Cart cart1 = JsonUtils.toBean(cartOnJson, Cart.class);
                cart1.setNum(cart1.getNum()+cart.getNum());
                boundHashOpertion.put(cart.getSkuId().toString(),JsonUtils.toString(cart1));
            }else {
                boundHashOpertion.put(cart.getSkuId().toString(),JsonUtils.toString(cart));
            }
        }
    }
```

22.下单



订单单独做一个微服务,当然订单也需要当前用户信息,我们添加一个SpringMVC的拦截器,用于获取用户信息.

创建订单逻辑比较复杂,需要组装订单数据,基本步骤如下:

- 组织Order数据,完成新增
  - 订单编号
  - 用户id
  - 订单金额相关数据,需要查询商品信息后逐个运算并累加获取
  - 订单状态数据
- 组织OrderDetail数据,完成新增
  - 需要查询商品信息后,封装为OrderDetail集合后,然后新增
- 组织OrderLogistics数据,完成新增
  - 需要查询到收获人地址
  - 然后根据收货人地址,封装OrderLogistics后,完成新增
- 下单成功后,商品对应库存应该减掉

22.1.生成订单编号

订单数据非常庞大,将来一定会做分库分表,那么这种情况下,要保证id的唯一,就不能靠数据库的自增,而是自己来实现算法,生成唯一id.我们使用的是工具类采用的生成id算法,是由Twitter公司开源的snowflake(雪花)算法.

22.2.减库存

减库存并没有采用先查询库存,判断充足才减库存的方案,那样会有线程安全问题,当然可以通过加锁来解决,不过我们此处为了效率,并没有使用,而是把数据库中的库存列设置为:无符号类型,当库存减到0以下时候,数据库会报错,从而避免超卖.

23.定时任务

23.1.Quartz

```
<!--1.自定义定时任务类交给spring容器-->
<bean id="myTask" class="cn.itcast.web.task.TestTask"></bean>

<!--2.配置jobDetal :  配置定时执行的类和方法-->
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="myTask"></property>
    <property name="targetMethod" value="testTask"></property>
</bean>

<!--3.配置trigger: （触发器）配置时间以及jobdetal关系 -->
<bean id="tigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">
    <!--cron表达式-->
    <property name="jobDetail" ref="jobDetail"></property>
    <property name="cronExpression" value="0/5 * * * * ? *"></property>
</bean>

<!--4.配置定时任务管理器-->
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="tigger"></ref>
        </list>
    </property>
</bean>
```

23.2.Corn表达式

域	允许值	允许的特殊字符
秒（Seconds）	0~59的整数	, - * /
分（*Minutes*）	0~59的整数	, - * /
小时（*Hours*）	0~23的整数	, - * /
日期（*DayofMonth*）	1~31的整数（但是你需要考虑你月的天数）	, - * ? / L W C
月份（*Month*）	1~12的整数或者 JAN-DEC	, - * /
星期（*DayofWeek*）	1~7的整数或者 SUN-SAT （1=SUN）	, - * ? / L C #
年(可选，留空)（*Year*）	1970~2099	, - * /

每个域上一般都是数字，或者指定允许的特殊字符：

特殊字符	说明
*	表示匹配该域的任意值。假如在Minutes域使用, 即表示每分钟都会触发事件
?	只能用在DayofMonth和DayofWeek两个域中的一个。它表示不确定的值
-	表示范围。例如在Hours域使用5-8，表示从5点、6点、7点、8点各执行一次
,	表示列出枚举值。例如：在week域使用FRI,SUN，表示星期五和星期六执行
/	一般用法：x/y，从x开始，每次递增y。如果放在minutes域，5/15，表示每小时的5分钟开始，每隔15分钟一次，即：5分钟、20分钟、35分钟、50分钟时执行
L	表示最后，只能出现在DayofWeek和DayofMonth域。如果在DayOfMonth中，代表每个月的最后一天。如果是在DayOfWeek域，表示每周最后一天（周六），但是如果是：数字+L，如6L表示每月的 最后一个周五
W	表示最近的有效工作日(周一到周五),只能出现在DayofMonth域，系统将在离指定日期的最近的有效工作日触发事件。例如：在 DayofMonth使用5W，如果5日是星期六，则将在最近的工作日：星期五，即4日触发。如果5日是星期天，则在6日(周一)触发；如果5日在星期一到星期五中的一天，则就在5日触发。另外一点，W的最近寻找不会跨过月份 。
LW	两个字符可以连用，表示在某个月最后一个工作日，即最后一个星期五
#	用在DayOfMonth中，确定每个月第几个星期几。例如在4#2，表示某月的第二个星期三（2表示当月的第二周，4表示这周的第4天，即星期三）。

23.3.Spring Schedule

SpringBoot中已经默认对Spring的Schedule实现了自动配置.我们需要开启定时任务功能,只要在启动类上加载一个`@EnableScheduling`注解即可.我们还需要定义任务类,通过`@Component`注解注册到Spring容器,方法上添加注解`@Scheduled(fixedRate = 1000)`来定义任务执行频率.默认情况下,定时任务的线程池大小只有,当任务较多执行频繁时,会出现阻塞等待情况,任务调度器就会出现时间飘移,任务时间将不确定.为了避免这样的情况发生,我们需要自定义线程池大小.

定时策略

`@Scheduled`可以控制定时执行的频率,有三种策略

- fixedRate:按照固定时间频率执行,单位毫秒,即每xx毫秒执行一次.
    - 如果上一个任务阻塞导致任务积压,则会在当前任务执行后,一次把多个积压的任务都执行完成.
- fixedDelay:固定延迟执行,单位毫秒,即前一个任务执行结束后xx毫秒执行第二个任务
- cron:使用cron表达式来定义任务执行策略.

24.分布式任务调度

解决方案有多种:

- 利用锁限制任务串行执行,同一时刻只能有一个节点执行任务.
- 创建分布式任务调度中心,记录定时任务,触发时不执行任务,而是通知微服务集群,利用负载均衡算法,实现任务调度,有许多开源的框架实现了这样的功能.

24.1.分布式锁方案

传统锁:当存在多个线程可以同时修改某个共享变量时候,因为有并发写的情况,可能出现丢失更新,数据不一致等情况,出现线程安全问题.锁的本质是一个标记,为了保证多个线程在一个时刻同一个代码块只能有一个县城可以执行,那么需要在某个地方做标记,这个标记必须每个线程都能看到,当标记不存在时候,可以设置该标记,其余后续线程发现已经有标记了则等待拥有标记的线程结束同步代码块取消标记后再去尝试设置标记,这个标记可以理解为锁.

分布式锁:在单机系统中,因为只有一个JVM进程,进程中有多个线程,因此多个线程可以共享内存空间,每个线程都可以看到这个内存标记,从而实现锁的效果.但是在分布式系统中,因为每个系统部署都是独立的JVM进程,多进程之间不共享内存,写在内存中的标记只对自己可见,锁就失效了.因此,分布式系统下,如果能够把锁的标记放到一个多进程共享的内存中,保证多进程可见,name就实现与单进程锁一样的效果了,这样的锁就是分布式锁.多进程共享的内存实现有多种方式,如:`Redis,Memcache,Zookeeper`.

24.2.任务调度中心方案

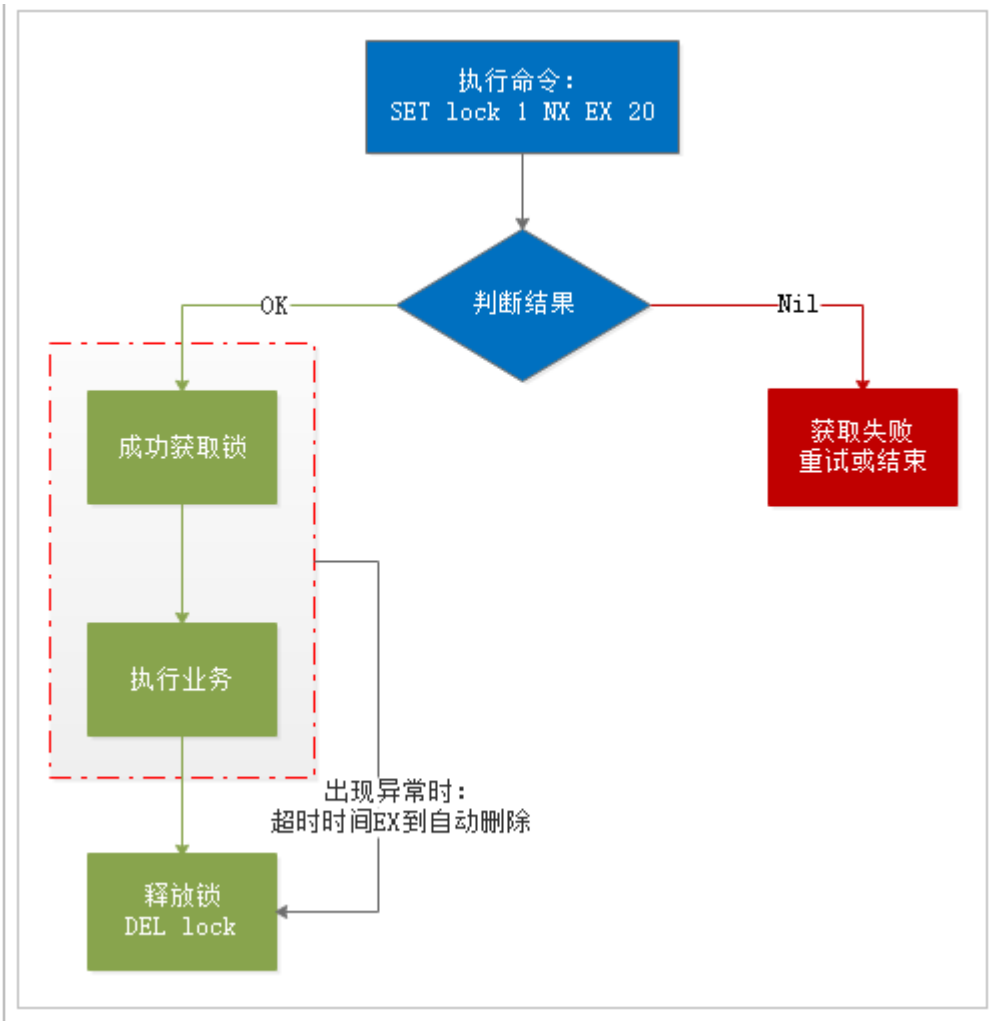
- Quartz：Java上的定时任务标准。但Quartz关注点在于定时任务而非数据，并无一套根据数据处理而定制化的流程。虽然Quartz可以基于数据库实现作业的高可用，但缺少分布式并行调度的功能
- TBSchedule：阿里早期开源的分布式任务调度系统。代码略陈旧，使用timer而非线程池执行任务调度。众所周知，timer在处理异常状况时是有缺陷的。而且TBSchedule作业类型较为单一，只能是获取/处理数据一种模式。还有就是文档缺失比较严重
- elastic-job：当当开发的弹性分布式任务调度系统，功能丰富强大，采用zookeeper实现分布式协调，实现任务高可用以及分片，目前是版本2.15，并且可以支持云开发
- Saturn：是唯品会自主研发的分布式的定时任务的调度平台，基于当当的elastic-job 版本1开发，并且可以很好的部署到docker容器上。
- xxl-job: 是大众点评员工徐雪里于2015年发布的分布式任务调度平台，是一个轻量级分布式任务调度框架，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。

24.3.Redis分布式锁

Redis实现分布式锁原理:

- **多进程可见**:redis本身就是多服务共享的,因此自然满足
- **排它**:同一时刻,只有一个进程获得锁
    - 我们利用Redis的setnx命令来实现,setnx是set when not exits,当多次执行setnx命令时,只有第一次执行的才会成功并返回1,其它情况返回0.
    - 我们定义一个固定的key,多个进行都执行setnx,设置这个key的值,返回1的服务获取锁,0则没有获取锁.
- **避免死锁**:死锁的情况有很多,比如服务宕机后的锁释放问题,我们设置锁时最好设置有效期,如果服务宕机,有效期到时候自动删除锁.`set lock 001 nx ex 20\`
- **高可用**:避免锁服务宕机或处理好宕机的补救措施.

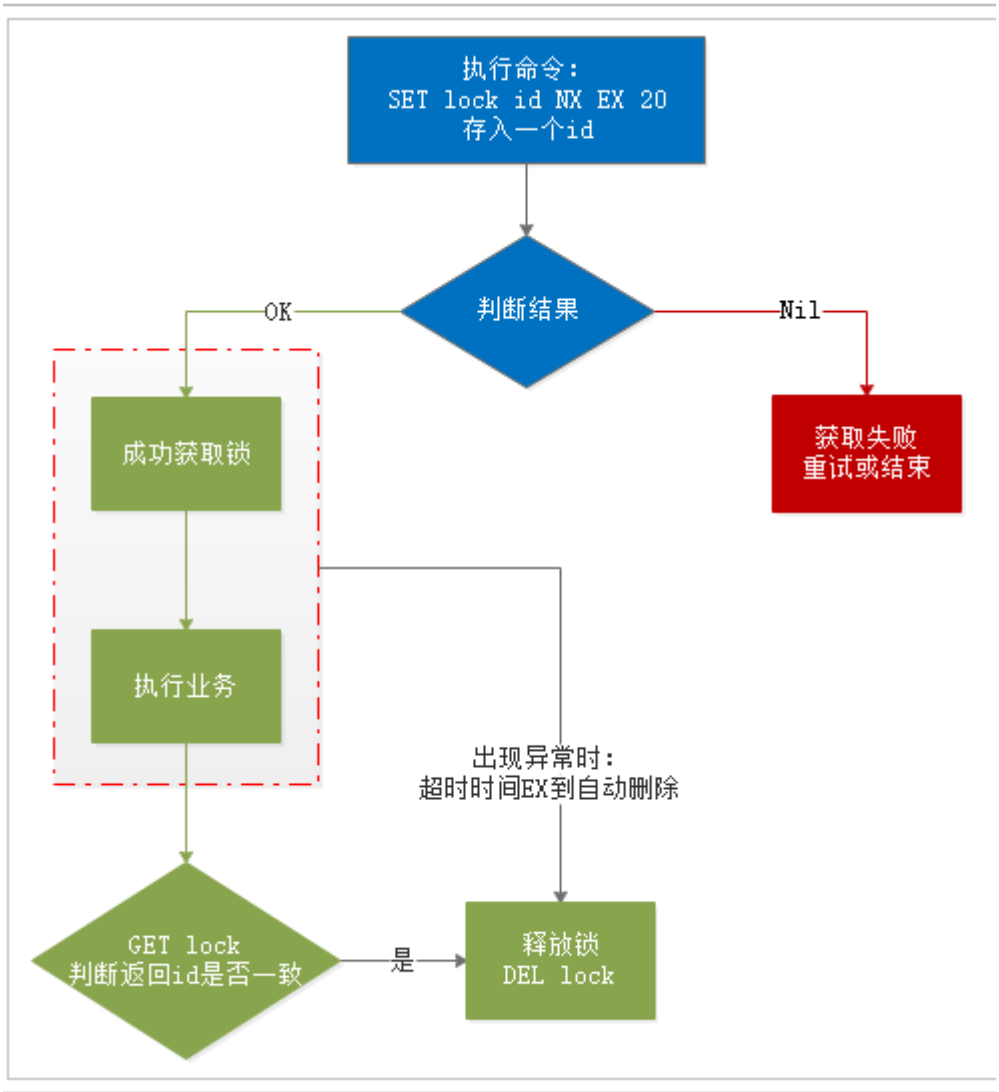
1.版本1




```
package com.leyou.task.utils;

import org.springframework.data.redis.core.StringRedisTemplate;

import java.util.concurrent.TimeUnit;

public class SimpleRedisLock implements RedisLock{

    private StringRedisTemplate redisTemplate;
    /**
     * 设定好锁对应的 key
     */
    private String key;
    /**
     * 锁对应的值，无意义，写为1
     */
    private static final String value = "1";

    public SimpleRedisLock(StringRedisTemplate redisTemplate, String key) {
        this.redisTemplate = redisTemplate;
        this.key = key;
    }

    public boolean lock(long releaseTime) {
```

```
        // 尝试获取锁
        Boolean boo = redisTemplate.opsForValue().setIfAbsent(key, value, releaseTime,
TimeUnit.SECONDS);
        // 判断结果
        return boo != null && boo;
    }

    public void unlock(){
        // 删除key即可释放锁
        redisTemplate.delete(key);
    }
}
```

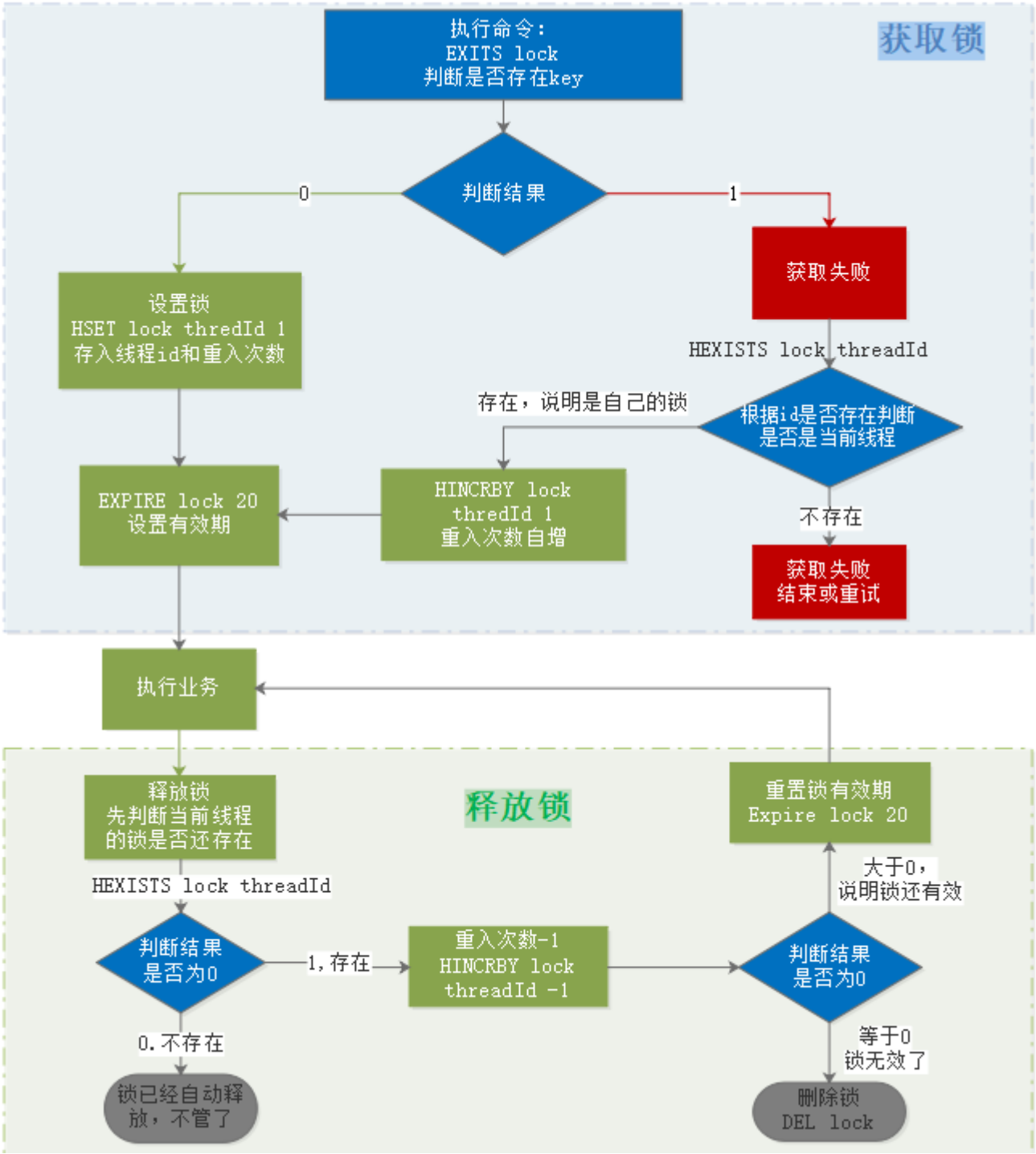
2.版本2




```
package com.leyou.task.utils;

import org.springframework.data.redis.core.StringRedisTemplate;

import java.util.UUID;
import java.util.concurrent.TimeUnit;
```

```
/**
 * @author
 */
public class SimpleRedisLock implements RedisLock{

    private StringRedisTemplate redisTemplate;
    /**
     * 设定好锁对应的 key
     */
    private String key;
    /**
     * 存入的线程信息的前缀，防止与其它JVM中线程信息冲突
     */
    private final String ID_PREFIX = UUID.randomUUID().toString();

    public SimpleRedisLock(StringRedisTemplate redisTemplate, String key) {
        this.redisTemplate = redisTemplate;
        this.key = key;
    }

    public boolean lock(long releaseTime) {
        // 获取线程信息作为值，方便判断是否是自己的锁
        String value = ID_PREFIX + Thread.currentThread().getId();
        // 尝试获取锁
        Boolean boo = redisTemplate.opsForValue().setIfAbsent(key, value, releaseTime,
TimeUnit.SECONDS);
        // 判断结果
        return boo != null && boo;
    }

    public void unlock(){
        // 获取线程信息作为值，方便判断是否是自己的锁
        String value = ID_PREFIX + Thread.currentThread().getId();
        // 获取现在的锁的值
        String val = redisTemplate.opsForValue().get(key);
        // 判断是否是自己
        if(value.equals(val)) {
            // 删除key即可释放锁
            redisTemplate.delete(key);
        }
    }
}
```

3.版本3



如上的版本,在获取锁之后,执行代码的过程中,再次尝试获取锁,执行setnx肯定会失败,因为锁已经存在了,这样就是不可重入锁,有可能导致死锁.

24.4.Redission

配置Redission客户端

```
@Configuration
public class RedisConfig {

    @Bean
    public RedissonClient redissonClient(RedisProperties prop) {
        String address = "redis://%s:%d";
        Config config = new Config();
        config.useSingleServer()
                .setAddress(String.format(address, prop.getHost(), prop.getPort()));
        return Redisson.create(config);
    }
}
```

```
@Slf4j
@Component
public class RedsssionJob {

    @Autowired
    private RedissonClient redissonClient;

    @Scheduled(cron = "0/10 * * * * ?")
    public void hello() {
        // 创建锁对象，并制定锁的名称
        RLock lock = redissonClient.getLock("taskLock");
        // 获取锁,设置自动失效时间为50s
        boolean isLock = lock.tryLock();
        // 判断是否获取锁
        if (!isLock) {
            // 获取失败
            log.info("获取锁失败，停止定时任务");
            return;
        }
        try {
            // 执行业务
            log.info("获取锁成功，执行定时任务。");
            // 模拟任务耗时
            Thread.sleep(500);
        } catch (InterruptedException e) {
            log.error("任务执行异常", e);
        } finally {
            // 释放锁
            lock.unlock();
            log.info("任务执行完毕，释放锁");
        }
    }
}
```

25.定时清理订单

业务分析:

- 设置一个超时时间限制,例如1小时,操作1小时未付款则取消订单.
- 查询当前以及超时未支付的订单.
- 修改这些订单的状态.
- 查询这些订单对应的商品信息,并恢复库存.

清理订单

```
@Transactional
public void closeOverdueOrders(Date deadline) {
    // 查询超时订单
    List<Long> orderIds = orderMapper.queryOverdueUnpaidOrder(deadline);
    if(CollectionUtils.isEmpty(orderIds)){
        log.info("【订单服务】没有需要清理的订单");
        return;
    }
    // 关闭超时订单
    Order record = new Order();
    record.setCloseTime(new Date());
    record.setStatus(OrderStatusEnum.CLOSED.value());
    // 更新订单的条件
    Example example = new Example(Order.class);
    example.createCriteria().andIn("orderId", orderIds);
    // 更新
    int count = orderMapper.updateByExampleSelective(record, example);
    log.info("【订单服务】定时关闭订单数量：{}", count);
    if (count != orderIds.size()) {
        throw new LyException(ExceptionEnum.UPDATE_OPERATION_FAIL);
    }
    // 恢复订单中的商品库存
    Example detailExample = new Example(OrderDetail.class);
    detailExample.createCriteria().andIn("orderId", orderIds);
    // 查询订单对应的订单详情
    List<OrderDetail> orderDetails = detailMapper.selectByExample(detailExample);
    if (CollectionUtils.isEmpty(orderDetails)) {
        throw new LyException(ExceptionEnum.ORDER_DETAIL_NOT_FOUND);
    }
    // 把以商品skuId为key，取出其中对应的库存数据，并叠加求和作为值
    Map<Long, Integer> skuMap = orderDetails.stream()
        .collect(Collectors.groupingBy(
            OrderDetail::getSkuId,
            Collectors.summingInt(OrderDetail::getNum)));
    // 调用加库存功能
    itemClient.plusStock(skuMap);
}
```

定时任务

```
@Slf4j
@Component
public class CloseOrderTask {

    @Autowired
```

```
    private RedissonClient redissonClient;

    @Autowired
    private OrderService orderService;

    /**
     * 定时任务的频率，30分钟
     */
    private static final long TASK_INTERVAL = 1800000;
    /**
     * 定时任务的锁自动释放时间。 \r\n
     * 一般只要大于各服务器的时钟飘移时长+任务执行时长即可。\r\n
     * 此处默认120秒。\r\n
     */
    private static final long TASK_LEASE_TIME = 120;
    /**
     * 订单超时的期限，1小时
     */
    private static final int OVERDUE_SECONDS = 3600;

    private static final String LOCK_KEY = "close:order:task:lock";

    @Scheduled(fixedDelay = TASK_INTERVAL)
    public void closeOrder() {
        // 0.创建锁对象
        RLock lock = redissonClient.getLock(LOCK_KEY);
        try {
            // 1.获取锁
            boolean isLock = lock.tryLock(0, TASK_LEASE_TIME, TimeUnit.SECONDS);
            if(!isLock){
                // 获取锁失败，结束任务
                log.info("【清理订单任务】未能获取任务锁，结束任务。");
                return;
            }
            // 2.开始执行任务
            try {
                // 2.1计算订单清理截止时间
                Date deadline = DateTime.now().minusSeconds(OVERDUE_SECONDS).toDate();
                // 2.2清理订单
                orderService.closeOverdueOrders(deadline);
            } finally {
                // 任务结束，释放锁
                lock.unlock();
                log.info("【清理订单任务】任务执行完毕，释放锁。");
            }

        } catch (InterruptedException e) {
            log.error("【清理订单任务】获取任务锁异常，原因：{}", e.getMessage(), e);
        }
    }
}
```