**第4章-产品附件-对象存储**

**学习目标**

* 完成保险发布保险附件功能开发
* 掌握微服务的如何进行服务化的功能拆分
* 掌握对象存储-文件上传功能流程及设计
* 掌握对象存储-垃圾文件处理流程及设计
* 了解对象存储-微服务化业务功能接口

**1、产品附件**

**1.1、需求分析**

**1.1.1、原型界面分析**

在保险产品的新建中；完成了产品信息——》产品方案——》产品系数项——》产品筛选项 之后；终于到了如下的最后一项：产品附件



在这个界面中；我们可以添加多种类型的资料附件pdf文件；保险详情、详情头图、产品特色三张图片。添加的这些文件都属于产品附属文件内容；也就是一个产品可能包含有多个附件文件。

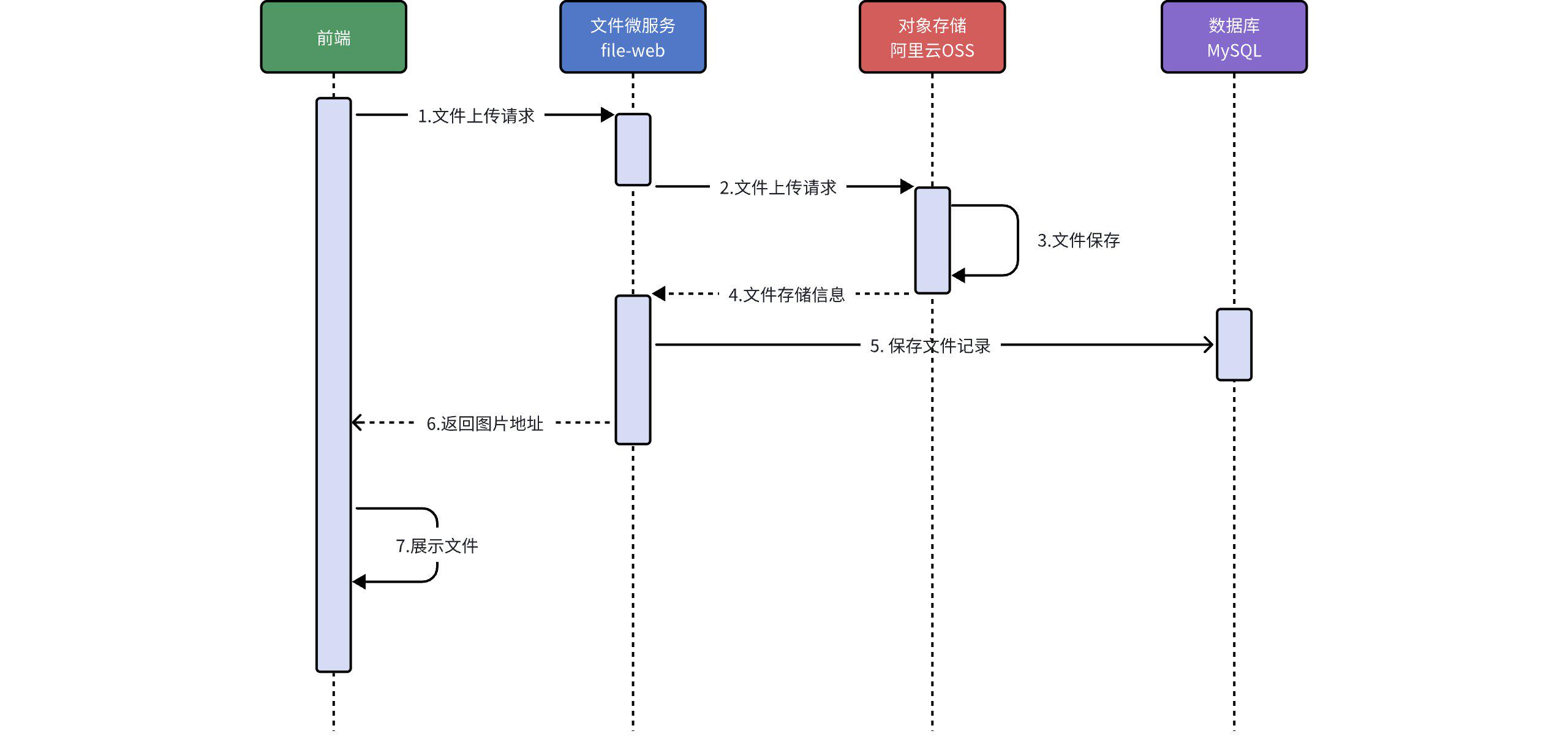
**主要思路**：在开发时需要将这些附件上传保存到存储服务器（阿里云、七牛云、FastDFS、MinIO等）并将这些附件记录、可访问的文件地址存储到数据库中；因为在产品详情页查看时需要使用到。要能够通过产品id查询到对应的附件。

**1.1.2、技术分析**

* 文件存储：存储具体的文件；可以采用学习过的阿里云OSS存储
* 文件记录：直接记录上传的文件信息到数据库表；sfbx-file数据库是专门存放文件、附件的数据库；其中有一个附件的表 tab\_file在这个表中存储附件的信息即可。

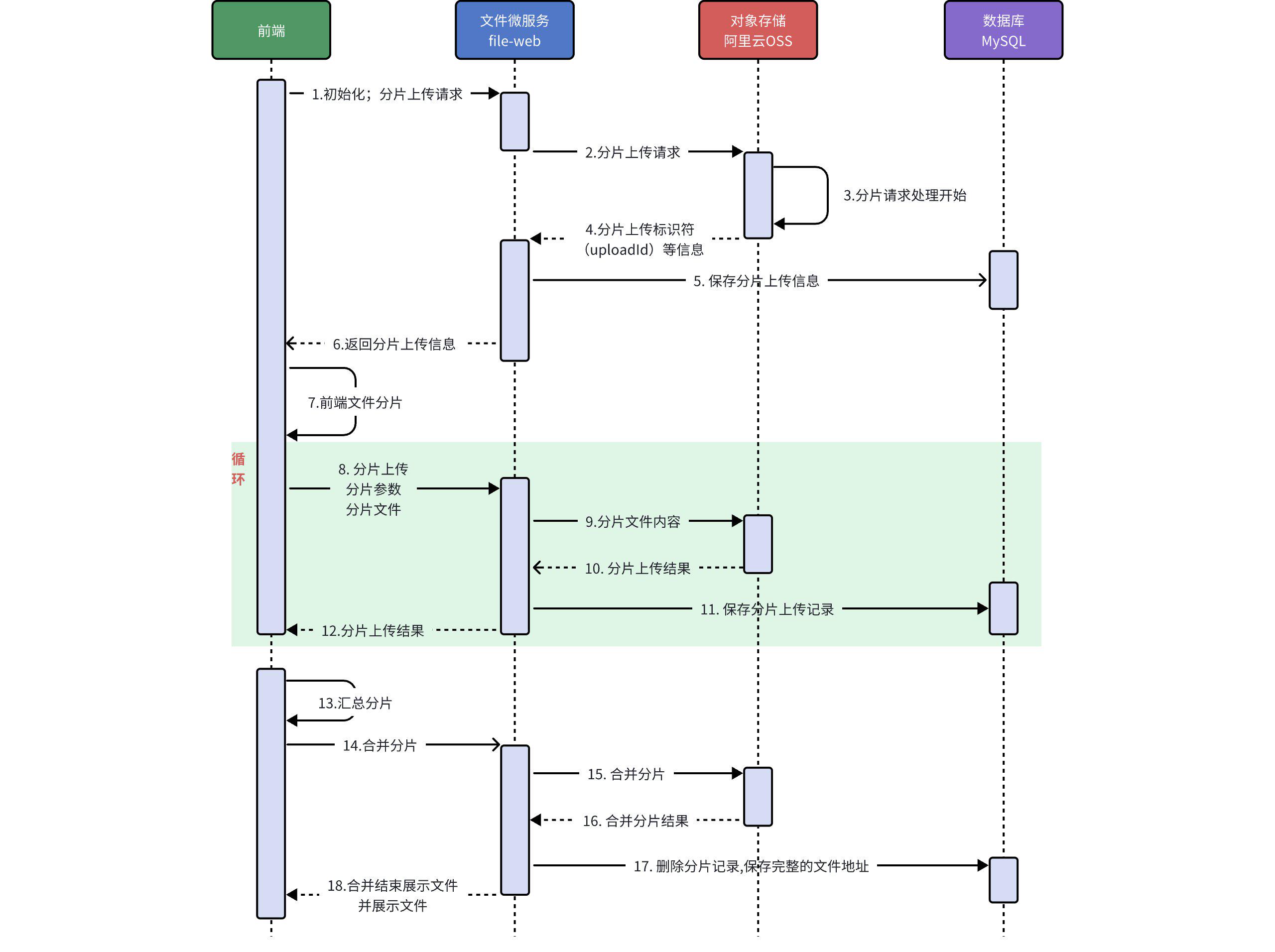
|  |
| --- |
| **特别注意**：  上传的资料都很有可能比较大；那么将这些大文件发送到 后端微服务的时候，很有可能因为网络不稳定而上传失败或者要重试多次，影响体验。可以将这些资料大文件分割为多个1M大小的文件多次上传到后端微服务，提高上传的成功率。 |

**1）简单文件上传时序图**



**2）大文件分片上传时序图**

在资料上传的时候；如果文件大小大于1M的大文件会对文件进行分割；分割后上传

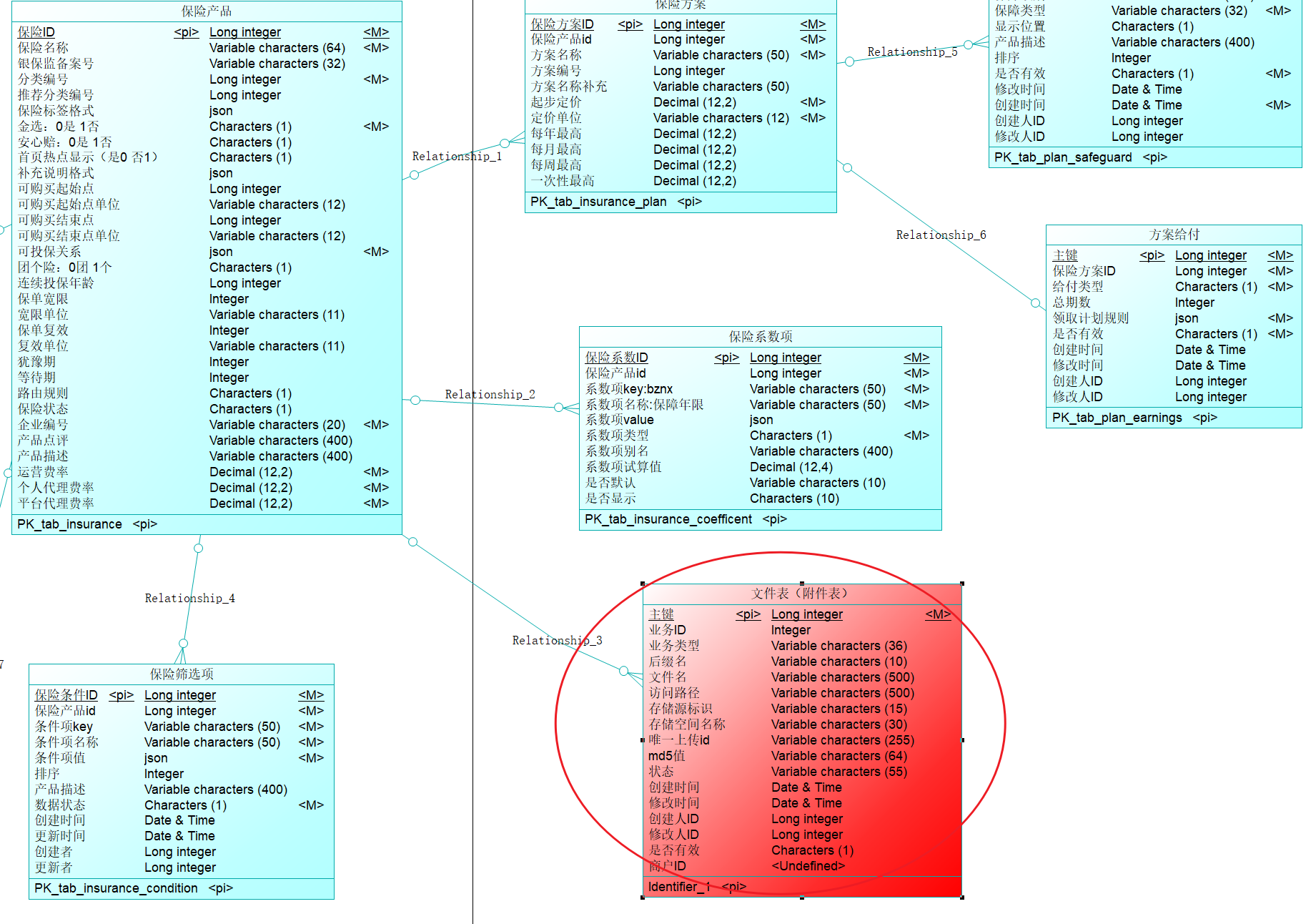


**1.2、数据库表设计**

根据需求分析，一个产品有多个附件，而这些附件都可以利用 sfbx-file 数据库中的已经存在的 附件表 tab\_file来存储。

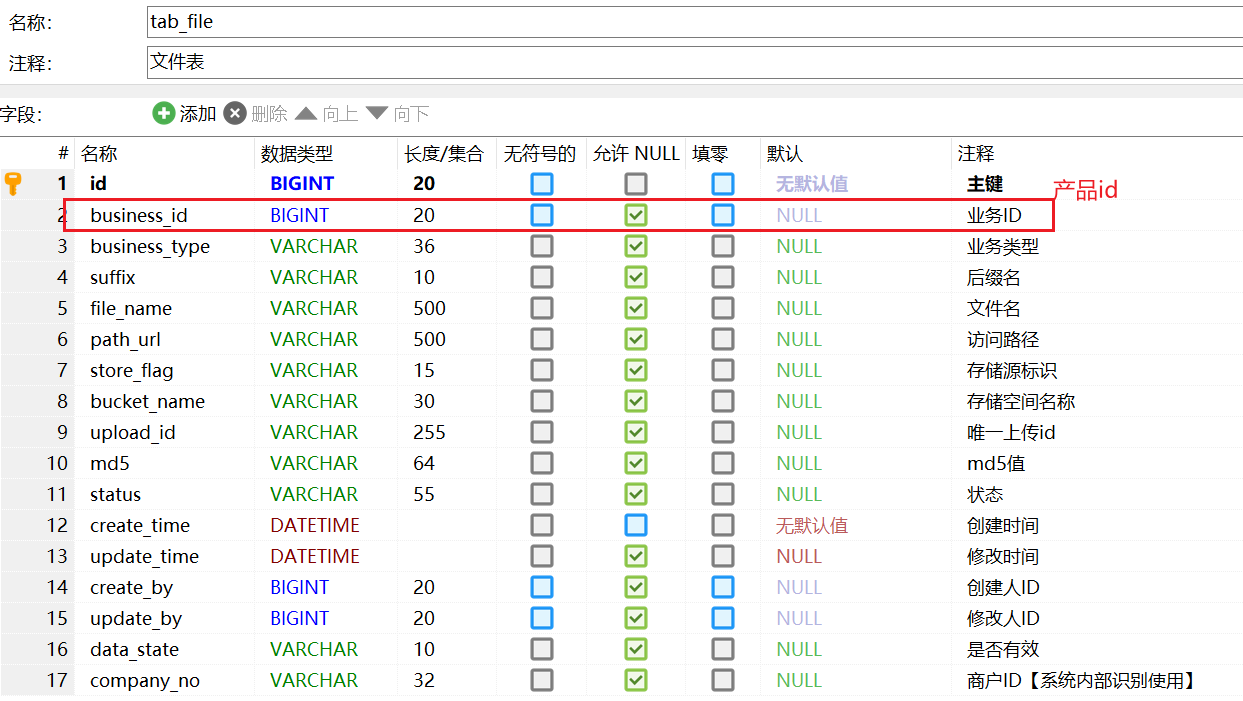
**1.2.3、概念模型**

附件表在与保险产品相关的概念模型如下：



**1.2.4、附件表**

对应附件表的数据库表结构如下：



**1.3、接口分析设计**

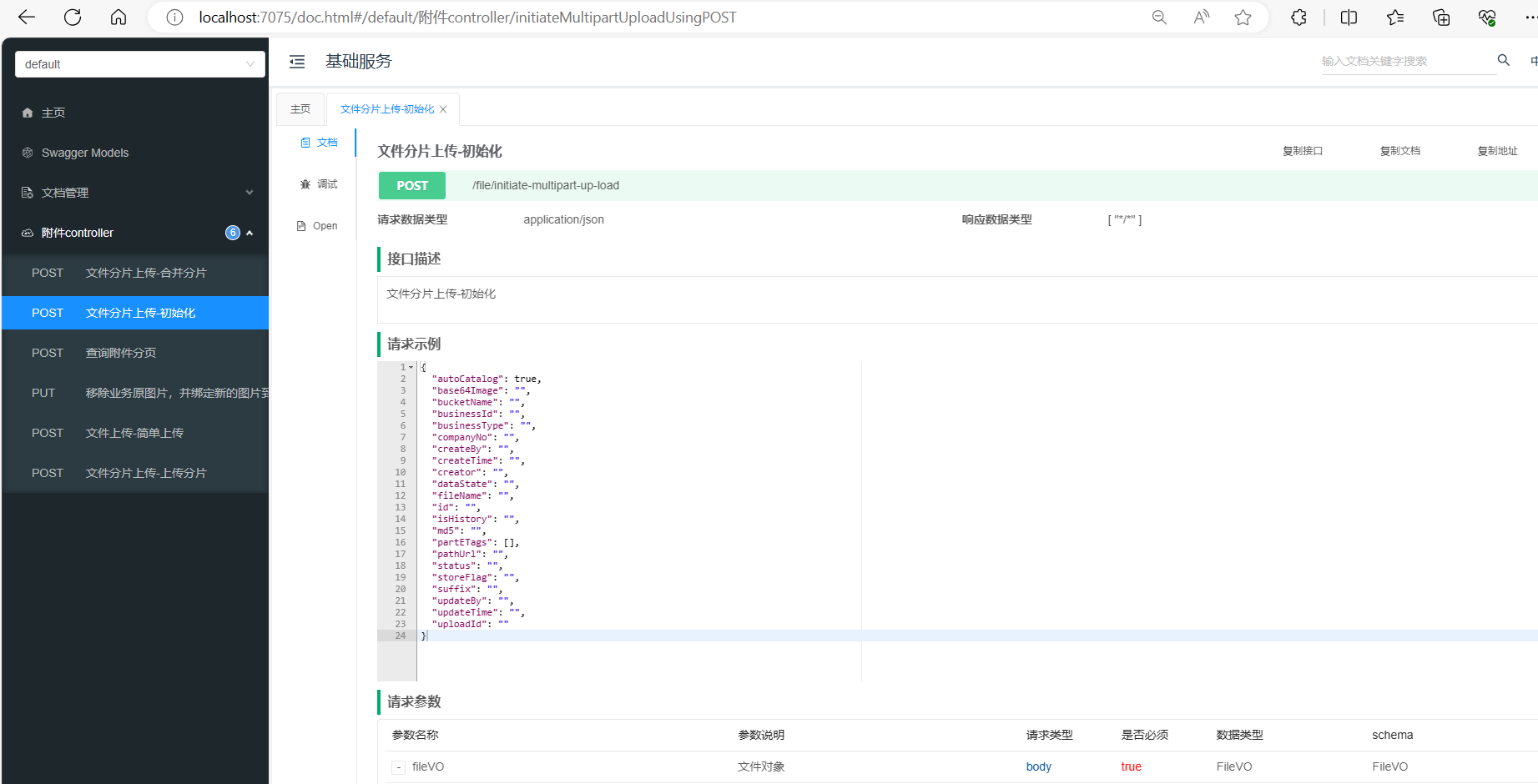
在上传文件的时候；可以在前端指定 storeFlag 来确实采用哪种存储服务器（ aliyunoss表示阿里云、qiniu表示七牛云），具体接口信息查看如下内容：

**1.3.1、上传大文件（资料）**

上传资料大文件的时候；需要对文件至少3次以上的处理；分别处理：分片初始化、n次分片上传、分片合并；所以也将对应3个上传接口：

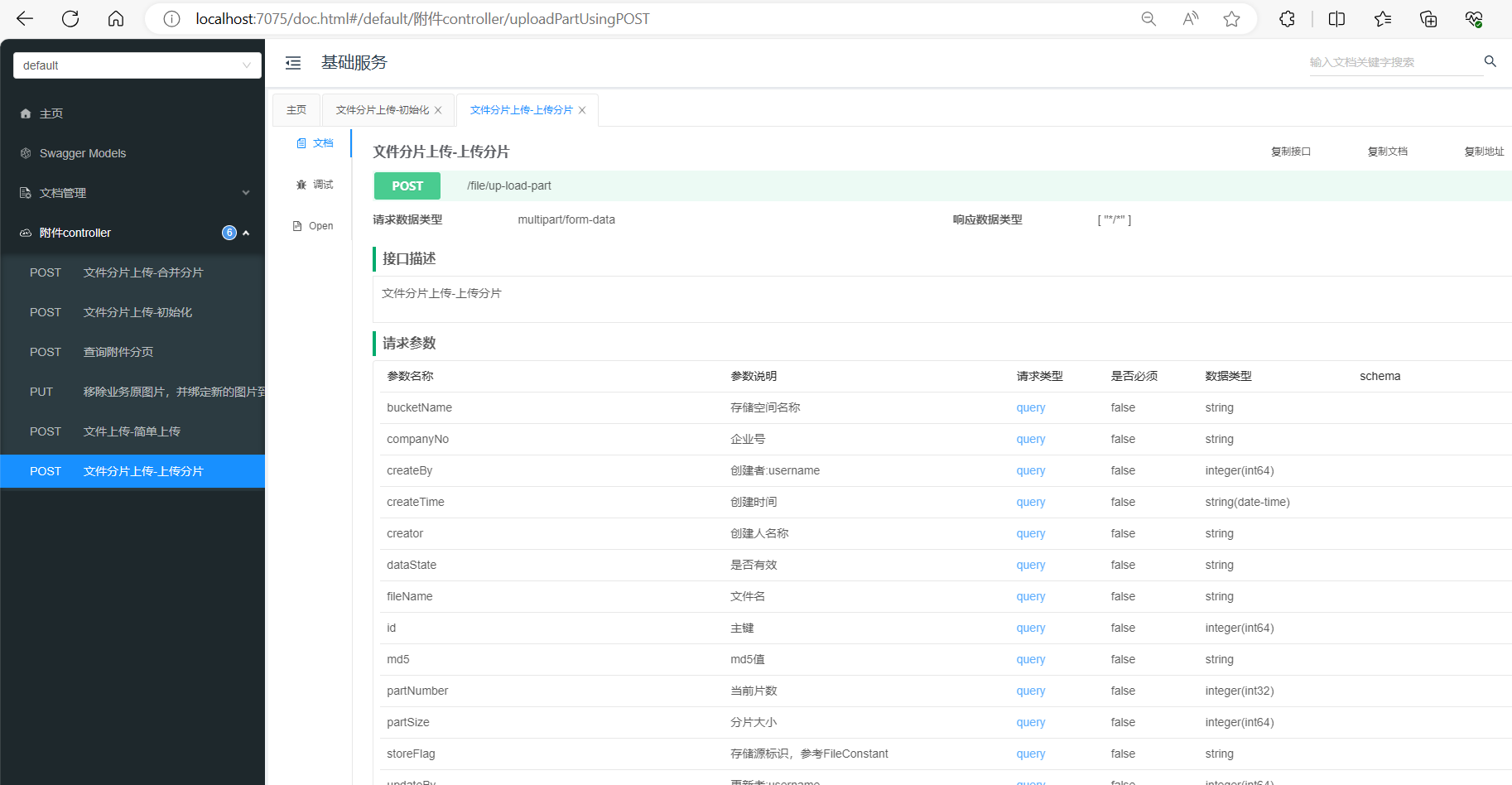
**1）初始化**

接口地址：http://localhost:7075/doc.html#/default/%E9%99%84%E4%BB%B6controller/initiateMultipartUploadUsingPOST



**2）上传分片**

接口地址：http://localhost:7075/doc.html#/default/%E9%99%84%E4%BB%B6controller/uploadPartUsingPOST



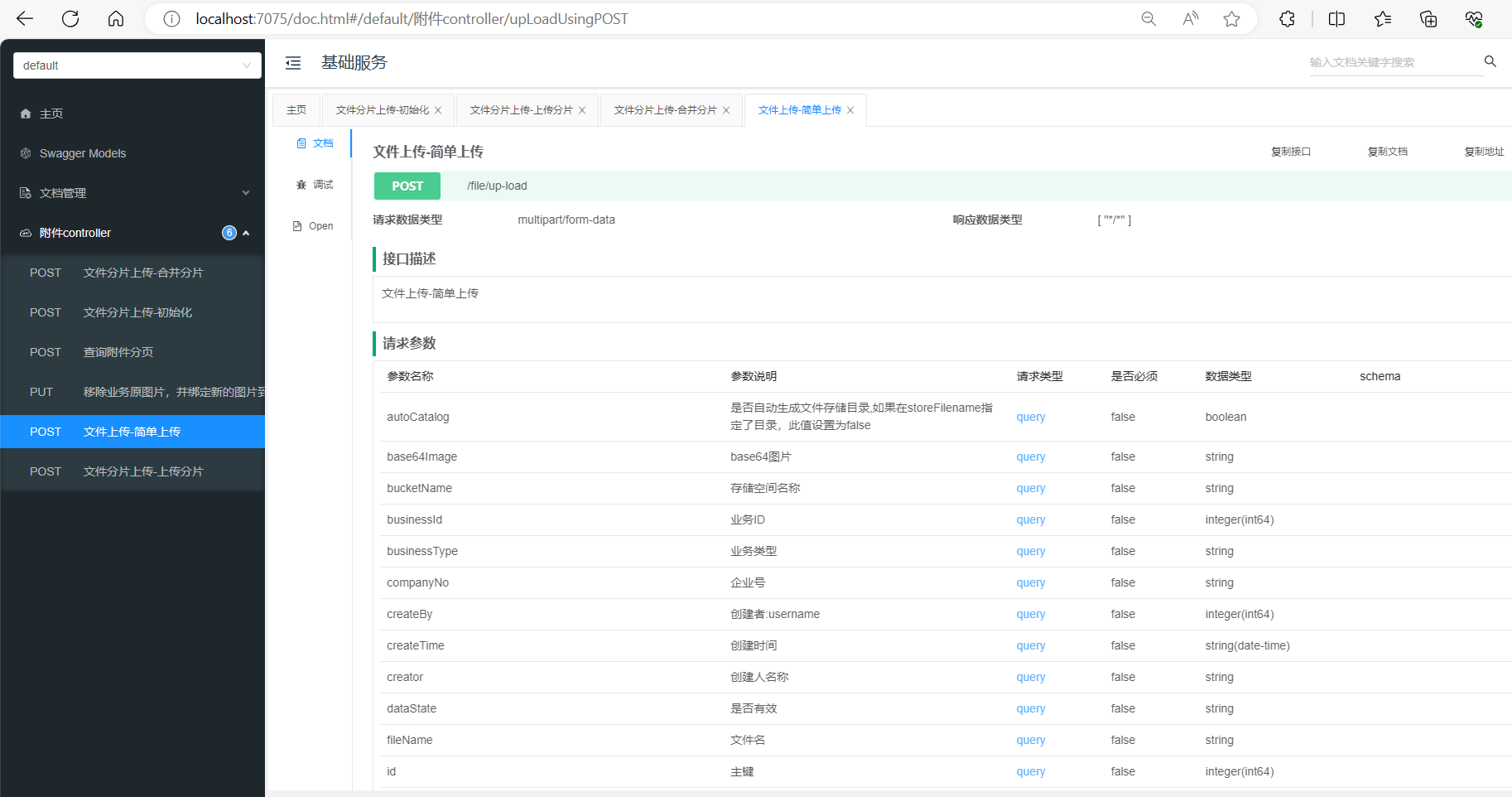
**3）合并分片**

接口地址：http://localhost:7075/doc.html#/default/%E9%99%84%E4%BB%B6controller/completeMultipartUploadUsingPOST



**1.3.2、上传图片**

上传图片主要针对保险详情、详情头图、产品特色三张图片；它们基本属于简单上传文件；一次上传即可，具体接口地址：http://localhost:7075/doc.html#/default/%E9%99%84%E4%BB%B6controller/upLoadUsingPOST



**1.4、开发实现说明**

**1.4.1、微服务划分**

文件上传功能，不只是在 保险产品中使用到；还可能在app端用户的头像上传、后端的其它业务模块中都可能需要上传文件、图片等内容；这类比较通用的业务；可以抽取为一个 文件微服务file-web 。在这个文件微服务中：

* 处理文件上传
* 上传文件的查询
* 文件的删除
* 垃圾文件的处理

说明了这个文件微服务需要记录下来，是由哪些业务上传过来的数据；而且附件的数量也比较庞大。所以需要有对应的数据库来记录业务与附件的关系。

于此；文件微服务对应一个 sfbx-file的数据库，里面存放对应附件相关的表。

那么；在保险管理后台中，需要上传文件怎么办？

**答案就是**：微服务之间进行远程调用；也就是：保险管理微服务insurance-mgt 通过open feign 接口调用 文件微服务file-web；从而实现在产品附件中资料、图片上传的功能。

**1.4.2、文件上传处理器**

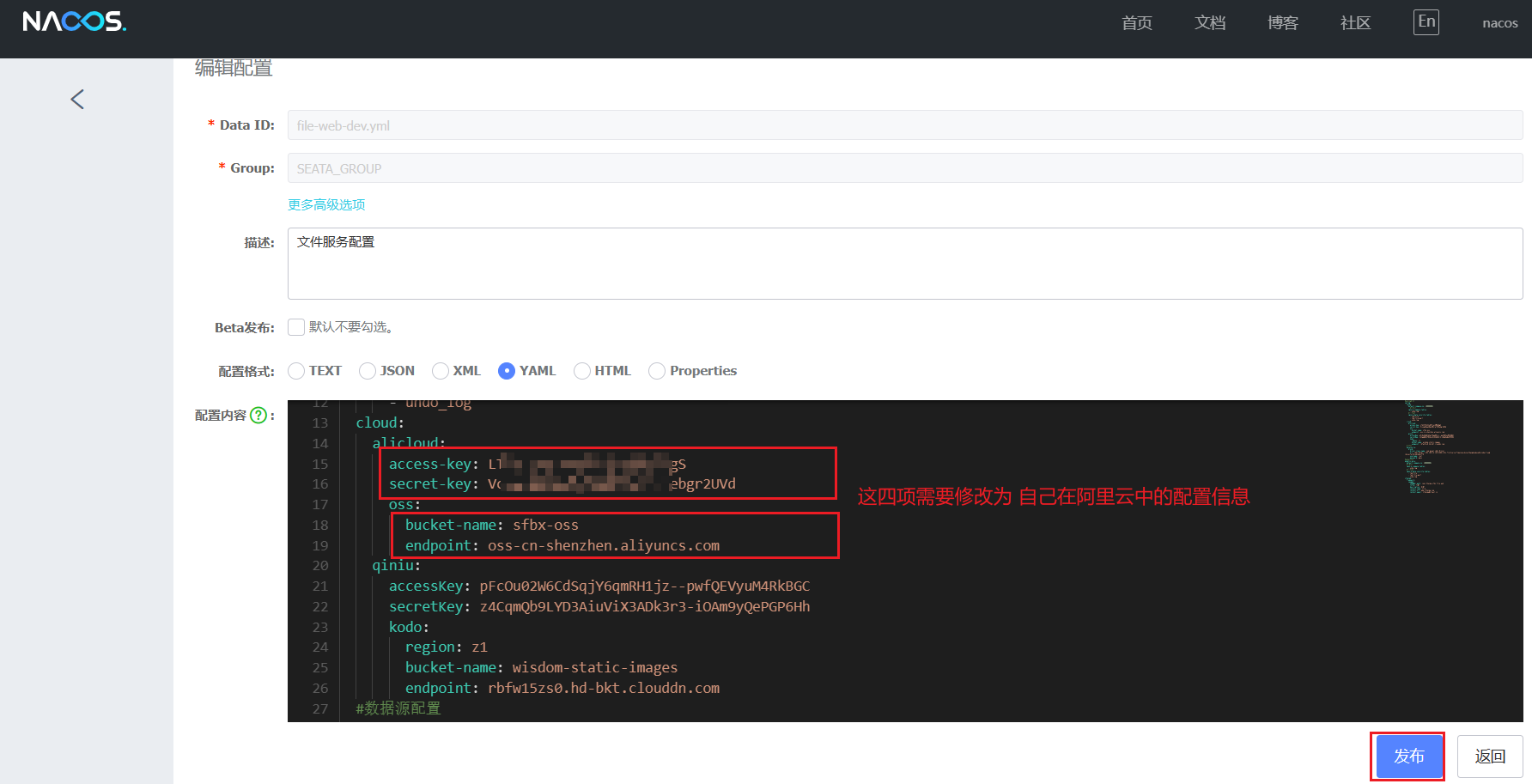
在四方保险中，关于大、小文件、图片上传的实现都在如下这个类中：

sfbx-cloud\sfbx-file\file-web\src\main\java\com\itheima\sfbx\file\web\FileUpLoadController.java

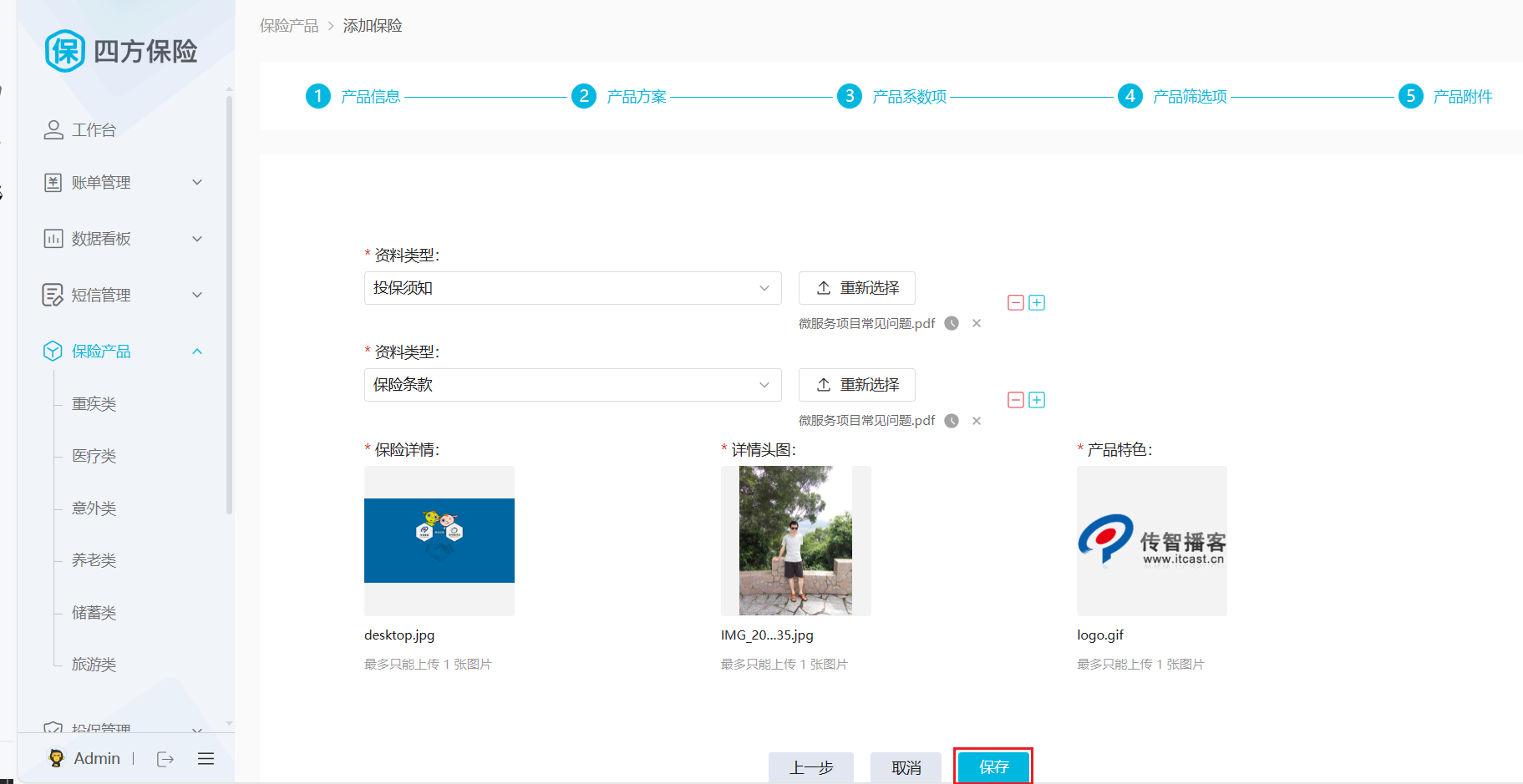
|  |
| --- |
| Java  */\*\**  *\* @ClassName FileUpLoadController.java*  *\* @Description 文件上传接口*  *\*/* @RestController @RequestMapping("file") @Api(tags = "附件controller") @Slf4j public class FileUpLoadController {   @Autowired  IFileService fileService;   @Autowired  private OssAliyunConfigProperties ossAliyunConfigProperties;   */\*\*\**  *\* @description 文件上传-简单上传-前端直接调用*  *\* @param file 上传对象*  *\* @return: com.itheima.travel.req.FileVO*  *\*/* @PostMapping(value = "up-load")  @ApiOperation(value = "文件上传-简单上传",notes = "文件上传-简单上传")  @ApiImplicitParams({  @ApiImplicitParam(paramType = "form", name = "file", value = "文件对象", required = true, dataTypeClass = MultipartFile.class)  })  @ApiOperationSupport(includeParameters = {"fileVO.businessType","fileVO.bucketName","fileVO.storeFlag","fileVO.autoCatalog"})  public ResponseResult<FileVO> upLoad(  @RequestParam("file") MultipartFile file,  FileVO fileVO) throws IOException {  *//因为前端传递了bucketName；但是目前前端不可修改；所以使用配置文件的话需要覆盖一下*  *// 如果要从前端确定bucketName再将下面一行代码删除即可；完全根据前端传递来确定* fileVO.setBucketName(ossAliyunConfigProperties.getBucketName());   fileVO.setCompanyNo(SubjectContent.*getCompanyNo*());  *//构建文件上传对象* UploadMultipartFile uploadMultipartFile = UploadMultipartFile  .*builder*()  .originalFilename(file.getOriginalFilename())  .fileByte(IOUtils.*toByteArray*(file.getInputStream()))  .build();  *//执行文件上传* FileVO fileVOResult = fileService.upLoad(uploadMultipartFile, fileVO);  return ResponseResultBuild.*successBuild*(fileVOResult);   }   @PostMapping(value = "initiate-multipart-up-load")  @ApiOperation(value = "文件分片上传-初始化",notes = "文件分片上传-初始化")  @ApiImplicitParam(name = "fileVO",value = "文件对象",required = true,dataType = "FileVO")  public ResponseResult<FileVO> initiateMultipartUpload(  @RequestBody FileVO fileVO){  *//因为前端传递了bucketName；但是目前前端不可修改；所以使用配置文件的话需要覆盖一下*  *// 如果要从前端确定bucketName再将下面一行代码删除即可；完全根据前端传递来确定* fileVO.setBucketName(ossAliyunConfigProperties.getBucketName());  fileVO.setCompanyNo(SubjectContent.*getCompanyNo*());  *//初始化上传Id* FileVO fileVOResult = fileService.initiateMultipartUpload(fileVO);  return ResponseResultBuild.*successBuild*(fileVOResult);  }   @PostMapping(value = "up-load-part")  @ApiOperation(value = "文件分片上传-上传分片",notes = "文件分片上传-上传分片")  @ApiImplicitParams({  @ApiImplicitParam(paramType = "form", name = "file", value = "文件对象", required = true, dataTypeClass = MultipartFile.class)  })  public ResponseResult<String> uploadPart(  @RequestParam("file") MultipartFile file,  FilePartVO filePartVO)throws IOException {  *//因为前端传递了bucketName；但是目前前端不可修改；所以使用配置文件的话需要覆盖一下*  *// 如果要从前端确定bucketName再将下面一行代码删除即可；完全根据前端传递来确定* filePartVO.setBucketName(ossAliyunConfigProperties.getBucketName());  filePartVO.setCompanyNo(SubjectContent.*getCompanyNo*());  *//构建文件上传对象* UploadMultipartFile uploadMultipartFile = UploadMultipartFile  .*builder*()  .originalFilename(file.getOriginalFilename())  .fileByte(IOUtils.*toByteArray*(file.getInputStream()))  .build();  *//上传分片返回partETagJson* String partETagJson = fileService.uploadPart(uploadMultipartFile,filePartVO);  return ResponseResultBuild.*successBuild*(partETagJson);  }   @PostMapping(value = "complete-multipart-up-load")  @ApiOperation(value = "文件分片上传-合并分片",notes = "文件分片上传-合并分片")  @ApiImplicitParam(name = "fileVO",value = "文件对象",required = true,dataType = "FileVO")  public ResponseResult<String> completeMultipartUpload(  @RequestBody FileVO fileVO)throws IOException {  *//因为前端传递了bucketName；但是目前前端不可修改；所以使用配置文件的话需要覆盖一下；*  *// 如果要从前端确定bucketName再将下面一行代码删除即可；完全根据前端传递来确定* fileVO.setBucketName(ossAliyunConfigProperties.getBucketName());  *//问上传分片返回partETagJson* String eTagJson = fileService.completeMultipartUpload(fileVO);  return ResponseResultBuild.*successBuild*(eTagJson);  }  } |

**1.4.3、修改阿里云OSS配置**

在文件微服务file-web中配置项也是存储在nacos中；对应的Data Id为：file-web-dev.yml文件；如果要修改上传文件的存储为自己的阿里云OSS的话（会影响原添加的一些保险产品的图片、文件）；那么修改如下：



**1.5、测试**



在上述的 产品附件 中如果添加了各类资料、图片之后，点击保存；再次查看这个产品的时候也一样能看到的话说明添加产品附件成功。

|  |
| --- |
| **思考题**：  如果要修改阿里云的终端和空间名的话；应该怎么做？且要保持原有的图片或文件可以访问，新的文件要存储到新空间 |

**2、对象存储**

**2.1、对象存储介绍**

对象存储服务（**O**bject **S**torage **S**ervice）是一种数据存储和管理模型，用于**存储和组织非结构化数据（文件：文本、图片、音频、视频）**，通常以对象（Object）的形式存储数据。每个对象通常包括数据本身、元数据（描述数据的信息），以及一个唯一的标识符。**总的来说；就是存文件的。也一般会称OSS为存储服务器。**

**文件可以存到哪里去？**

* 方式一：可以存储到服务器所在硬盘；（web阶段讲解过）
* 方式二：可以自己搭建存储服务器；比如：MinIO、FastDFS都是可自行搭建的分布式文件存储服务器
* 方式三：可以使用第三方，自己不用搭；直接用就行。比如：阿里云OSS（https://oss.console.aliyun.com/bucket）、华为云OSS、七牛云等

四方保险就是采用了第三方；并且抽象了存储实现这一层，可以通过参数进行灵活的切换存储服务器OSS（阿里云与七牛云之间切换）

**2.2、简单文件上传**

使用Java SDK发起OSS请求，需要配置访问凭证。具体如下：

|  |
| --- |
| Shell # 测试之前需要先设置系统环境变量；打开CMD 执行如下命令： set OSS\_ACCESS\_KEY\_ID=你自己在阿里云上的AccessKey set OSS\_ACCESS\_KEY\_SECRET=你自己在阿里云上的AccessKeySecret  # 永久生效 setx OSS\_ACCESS\_KEY\_ID "%OSS\_ACCESS\_KEY\_ID%" setx OSS\_ACCESS\_KEY\_SECRET "%OSS\_ACCESS\_KEY\_SECRET%"  # 设置之后；可以通过如下命令查看是否设置成功. IDEA中要生效的话，可以重启IDEA echo %OSS\_ACCESS\_KEY\_ID% echo %OSS\_ACCESS\_KEY\_SECRET% |

如下是一个上传文件的简单代码：

参考阿里云文档地址：https://help.aliyun.com/zh/oss/user-guide/simple-upload?spm=a2c4g.11186623.4.2.61435183VvEgPZ&scm=20140722.H\_31848.\_.ID\_31848-OR\_rec-V\_1

演示简单上传如下：

1、添加 file-web 中添加测试依赖；

|  |
| --- |
| XML <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId> </dependency> |

2、编写测试类；sfbx-cloud\sfbx-file\file-web\src\test\java\com\itheima\sfbx\file\AliOSSUploadTest.java

具体示例代码如下：

|  |
| --- |
| Java package com.itheima.sfbx.file;  import com.aliyun.oss.ClientException; import com.aliyun.oss.OSS; import com.aliyun.oss.OSSClientBuilder; import com.aliyun.oss.OSSException; import com.aliyun.oss.common.auth.CredentialsProviderFactory; import com.aliyun.oss.common.auth.EnvironmentVariableCredentialsProvider; import com.aliyun.oss.model.PutObjectRequest; import com.aliyun.oss.model.PutObjectResult; import org.junit.Test;  import java.io.ByteArrayInputStream; import java.io.File;  public class AliOSSUploadTest {   */\*\**  *\* 简单上传*  *\*/* @Test  public void testSimpleUpload() throws Exception {  *// Endpoint以华东1（杭州）为例，其它Region请按实际情况填写。* String endpoint = "https://oss-cn-shenzhen.aliyuncs.com";  *// 从环境变量中获取访问凭证。运行本代码示例之前，请确保已设置环境变量OSS\_ACCESS\_KEY\_ID和OSS\_ACCESS\_KEY\_SECRET。* EnvironmentVariableCredentialsProvider credentialsProvider = CredentialsProviderFactory.*newEnvironmentVariableCredentialsProvider*();  *// 填写Bucket名称，例如examplebucket。* String bucketName = "sfbx-oss";  *// 填写Object完整路径，完整路径中不能包含Bucket名称，例如exampledir/exampleobject.txt。* String objectName = "test/pic/logo.png";  *// 填写本地文件的完整路径，例如D:\\localpath\\examplefile.txt。*  *// 如果未指定本地路径，则默认从示例程序所属项目对应本地路径中上传文件。* String filePath= "D:\\itcast\\pics\\logo.png";   *// 创建OSSClient实例。* OSS ossClient = new OSSClientBuilder().build(endpoint, credentialsProvider);   try {  *// 创建PutObjectRequest对象。* PutObjectRequest putObjectRequest = new PutObjectRequest(bucketName, objectName, new File(filePath));  *// 如果需要上传时设置存储类型和访问权限，请参考以下示例代码。*  *// ObjectMetadata metadata = new ObjectMetadata();*  *// metadata.setHeader(OSSHeaders.OSS\_STORAGE\_CLASS, StorageClass.Standard.toString());*  *// metadata.setObjectAcl(CannedAccessControlList.Private);*  *// putObjectRequest.setMetadata(metadata);*   *// 上传文件。* PutObjectResult result = ossClient.putObject(putObjectRequest);   System.*out*.println(result);  } catch (OSSException oe) {  System.*out*.println("Caught an OSSException, which means your request made it to OSS, "  + "but was rejected with an error response for some reason.");  System.*out*.println("Error Message:" + oe.getErrorMessage());  System.*out*.println("Error Code:" + oe.getErrorCode());  System.*out*.println("Request ID:" + oe.getRequestId());  System.*out*.println("Host ID:" + oe.getHostId());  } catch (ClientException ce) {  System.*out*.println("Caught an ClientException, which means the client encountered "  + "a serious internal problem while trying to communicate with OSS, "  + "such as not being able to access the network.");  System.*out*.println("Error Message:" + ce.getMessage());  } finally {  if (ossClient != null) {  ossClient.shutdown();  }  }  } } |

|  |
| --- |
| 设置环境变量链接：  https://help.aliyun.com/zh/oss/developer-reference/oss-java-configure-access-credentials |

**2.3、分片文件上传**

在上传大文件（超过5 GB）到OSS的过程中，如果出现网络中断、程序异常退出等问题导致文件上传失败，您需要使用分片上传的方式上传大文件。分片上传通过将待上传的大文件分成多个较小的碎片（Part），充分利用网络带宽和服务器资源并发上传多个Part，加快上传完成时间，并在Part上传完成之后调用CompleteMultipartUpload接口将这些Part组合成一个完整的Object。

**2.3.1、使用场景**

* 大文件**加速上传**

当文件大小超过5 GB时，使用分片上传可实现并行上传多个Part以加快上传速度。

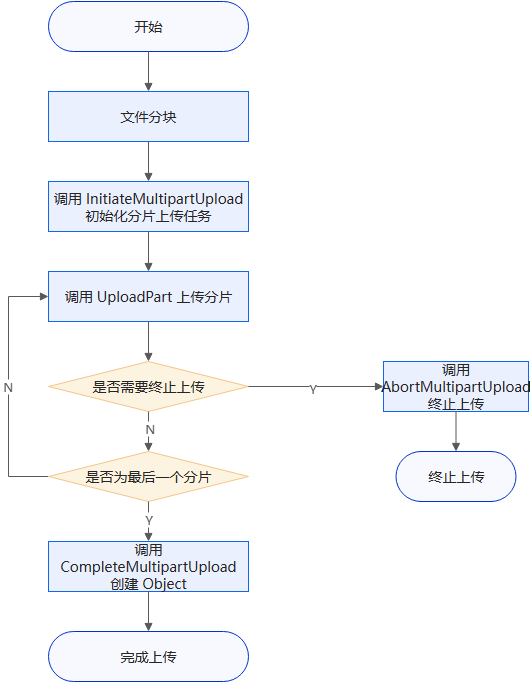
* 网络**环境较差**

网络环境较差时，建议使用分片上传。当出现上传失败的时候，您仅需重传失败的Part。

* 文件**大小不确定**

可以在需要上传的文件大小还不确定的情况下开始上传，这种场景在视频监控等行业应用中比较常见。

**2.3.2、分片上传流程**



**流程说明**：

1. 将待上传文件按照一定大小进行分片。
2. 使用InitiateMultipartUpload接口初始化一个分片上传任务。
3. 使用UploadPart接口上传分片。

文件切分成Part之后，文件顺序是通过上传过程中指定的partNumber来确定，所以您可以并发上传这些碎片。并发数并非越多越快，请结合自身网络状况和设备负载综合考虑。

如果您希望终止上传任务，可调用AbortMultipartUpload接口，成功上传的Part会一并删除。

1. 使用CompleteMultipartUpload接口将Part组合成一个Object。

|  |
| --- |
| **使用限制**：byte 1B ->kb->mb->gb->tb->pb->bb ； 1TB = 1024GB  文件大小不超过48.8TB  part分片数量：1-10000个  单个part分片大小：最小值为100kb，最大值为5GB.最后一个part的大小运行小于100kb |

**2.3.3、代码实现**

参考 https://help.aliyun.com/zh/oss/user-guide/multipart-upload

添加方法：com.itheima.sfbx.file.AliOSSUploadTest#testMultiPartUpload 方法如下：

|  |
| --- |
| Java */\*\**  *\* 分片上传*  *\*/* @Test public void testMultiPartUpload() throws Exception {  *// Endpoint以华东1（杭州）为例，其它Region请按实际情况填写。* String endpoint = "https://oss-cn-shenzhen.aliyuncs.com";  *// 从环境变量中获取访问凭证。运行本代码示例之前，请确保已设置环境变量OSS\_ACCESS\_KEY\_ID和OSS\_ACCESS\_KEY\_SECRET。* EnvironmentVariableCredentialsProvider credentialsProvider = CredentialsProviderFactory.*newEnvironmentVariableCredentialsProvider*();  *// 填写Bucket名称，例如examplebucket。* String bucketName = "sfbx-oss";  *// 填写Object完整路径，完整路径中不能包含Bucket名称，例如exampledir/exampleobject.txt。* String objectName = "test/file/"+ UUID.*randomUUID*()+".pdf";  *// 填写本地文件的完整路径，例如D:\\localpath\\examplefile.txt。*  *// 如果未指定本地路径，则默认从示例程序所属项目对应本地路径中上传文件。* String filePath= "D:\\itcast\\pics\\微服务项目常见问题.pdf";   *// 创建OSSClient实例。* OSS ossClient = new OSSClientBuilder().build(endpoint, credentialsProvider);   try {  *//1、初始化上传* InitiateMultipartUploadRequest initiateMultipartUploadRequest = new InitiateMultipartUploadRequest(bucketName, objectName);  InitiateMultipartUploadResult initiateMultipartUploadResult = ossClient.initiateMultipartUpload(initiateMultipartUploadRequest);  String uploadId = initiateMultipartUploadResult.getUploadId();  System.*out*.println("分片上传初始化完成；uploadId：" + uploadId);  *//2、分片上传文件*  *//2.1、读取要上传的文件* File file = new File(filePath);  *//2.2、计算要分片上传的次数；每个分片大小设置为1M，计算要上传多少次*  *//单个分片文件大小；1MB* long partSize = 1024 \* 1024;  *//文件总长度* long fileLength = file.length();  *//分为几个分片* int partCount = (int) (fileLength / partSize);  if (fileLength % partSize != 0) {  partCount++;  }  System.*out*.println("总共分为：" + partCount + "个分片");  *//2.3、循环上传分片*  *//记录每次每个分片上传之后的eTag* List<PartETag> partETags = new ArrayList<PartETag>();  for (int i = 0; i < partCount; i++) {  *//当前分片的文件起始位置* long startPos = i \* partSize;   *//创建分片请求对象* UploadPartRequest uploadPartRequest = new UploadPartRequest();  *//桶名称* uploadPartRequest.setBucketName(bucketName);  *//上传的文件名* uploadPartRequest.setKey(objectName);  uploadPartRequest.setUploadId(uploadId);  *//分片号* uploadPartRequest.setPartNumber(i + 1);  *//设置当前分片文件内容* FileInputStream fileInputStream = new FileInputStream(file);  fileInputStream.skip(startPos);  uploadPartRequest.setInputStream(fileInputStream);   *//当前这个分片的大小；但是最后一个分片大小可能是不到1M的；所以需要处理* long currentPartSize = (i+1==partCount)?(fileLength-startPos):partSize;  uploadPartRequest.setPartSize(currentPartSize);   *//上传分片* UploadPartResult uploadPartResult = ossClient.uploadPart(uploadPartRequest);   System.*out*.println("Part#" + uploadPartRequest.getPartNumber() + " ETag:" + uploadPartResult.getETag());  *//记录 etag* partETags.add(uploadPartResult.getPartETag());  }   *//3、完成分片上传；合并（阿里云端合并）* CompleteMultipartUploadRequest completeMultipartUploadRequest  = new CompleteMultipartUploadRequest(bucketName, objectName, uploadId, partETags);   ossClient.completeMultipartUpload(completeMultipartUploadRequest);   System.*out*.println("分片上传完成！");  } catch (OSSException oe) {  System.*out*.println("Caught an OSSException, which means your request made it to OSS, "  + "but was rejected with an error response for some reason.");  System.*out*.println("Error Message:" + oe.getErrorMessage());  System.*out*.println("Error Code:" + oe.getErrorCode());  System.*out*.println("Request ID:" + oe.getRequestId());  System.*out*.println("Host ID:" + oe.getHostId());  } catch (ClientException ce) {  System.*out*.println("Caught an ClientException, which means the client encountered "  + "a serious internal problem while trying to communicate with OSS, "  + "such as not being able to access the network.");  System.*out*.println("Error Message:" + ce.getMessage());  } finally {  if (ossClient != null) {  ossClient.shutdown();  }  } } |

四方保险中；超过1M大小的资料文件也是通过上述的思路实现的；只是分片都由阿里云提供的js完成，作为前端只需要引入对应的js并调用即可；自动进行分片上传。

**2.4、垃圾文件处理**

在四方保险项目中；产品附件页面可能会出现如下情况：



在产品附件中；因为附件的上传和产品附件信息的保存并不是同步进行的；所以例如下列操作就会产生垃圾文件：

* 首先，文件上传，此时文件已在对象存储及文件数据库中存在
* 然后，因网络断开或业务人员不去提交业务表单，导致business\_id与file\_id不绑定
* 最后，文件数据库中存在有一个business\_id为空的文件记录，而这个文件是无意义的，需要删除（数据库中的记录和oss的文件）。

处理此问题的方案有2种：

* 定时清理：使用XXL-JOB组件每隔一段时间轮询文件数据库，如果发现文件的创建时间是10分钟前business\_id为空的记录；如果是则认为是垃圾文件——》推荐
* 延迟清理：文件上传后，发送一条延迟消息到rabbitmq中，10分钟后执行消息检测此文件是否关联business\_id，为空则认为是垃圾文件

**3、任务调度-xxl-job**

**任务调度**：可以指定具体的时间或者固定的频率来执行任务（代码）

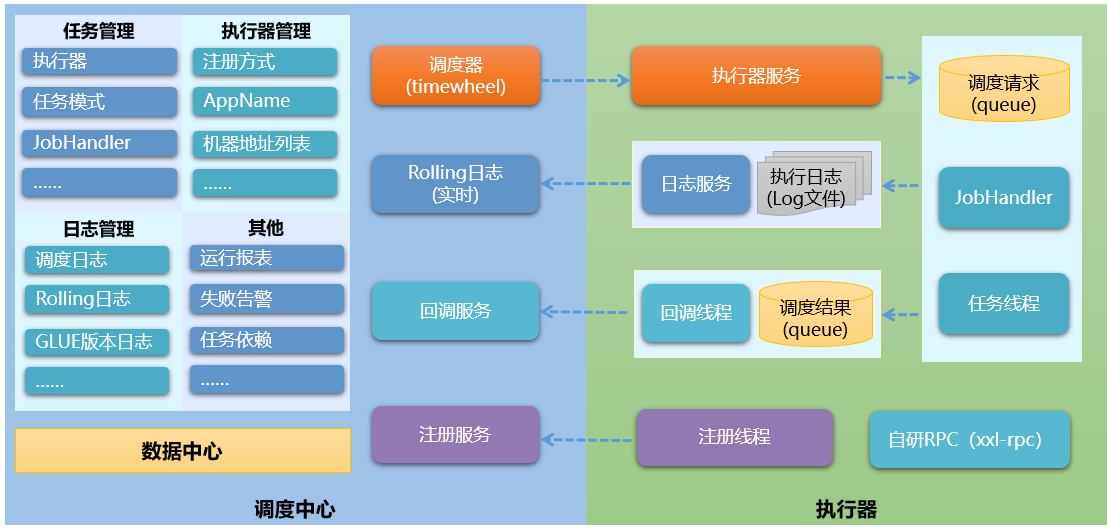
常见的任务调度框架：

* **JDK timer** 只能固定频率执行任务，而且只有应用服务器停止之后才能停止；不方便管理
* **Spring task** 可以指定时间和频率来执行任务；但是不能实现**分布式任务调度（要将任务灵活的划分到不同的集群节点中执行任务）**；如果要分布式任务调度要加分布式锁
* **Quartz** 传统老牌的任务调度框架；可以指定时间和频率来执行任务；
* 简单触发器simpleTrigger：以固定的频率执行任务
* 任务调度触发器CronTrigger：可以指定精细的执行时间和频率执行
* **Elastic Job** 分布式任务调度框架
* **Xxl-job** 分布式任务调度框架；有管理界面控制台可以随时添加、启动和指定执行策略。当下使用的更多
* 阿兹卡班

XXL-JOB是美团点评中间件研发工程师许雪里开发的一个分布式任务调度平台，其核心设计目标是开发迅速、学习简单、轻量级、易扩展。目前已有多家公司接入xxl-job，包括比较知名的大众点评，京东，优信二手车，北京尚德，360金融 (360)，联想集团 (联想)，易信 (网易)等等....

官网地址：<https://github.com/xuxueli/xxl-job>

**3.1、架构图**



**问题**：

1. xxl-job分为哪几个部分？
2. 在调度中心如何监控任务的执行过程？
3. 调度中心如何知道执行器的ip和端口？
4. 在执行器中有哪几个队列？为什么会有队列？

**3.2、快速入门**

**3.2.1、调度中心**

在四方保险中已经在虚拟机中部署好xxl-job调度中心；

访问地址：http://192.168.12.129:8280/xxl-job-admin 用户名：admin；密码：123456



**3.2.2、执行器**

在四方保险中，定时任务的执行都集中到了 task-job 模块工程中；所以启动了这个工程就相当于启动了整个四方保险的与定时任务相关的xxl-job执行器。启动类 com.itheima.sfbx.task.TaskJobStart

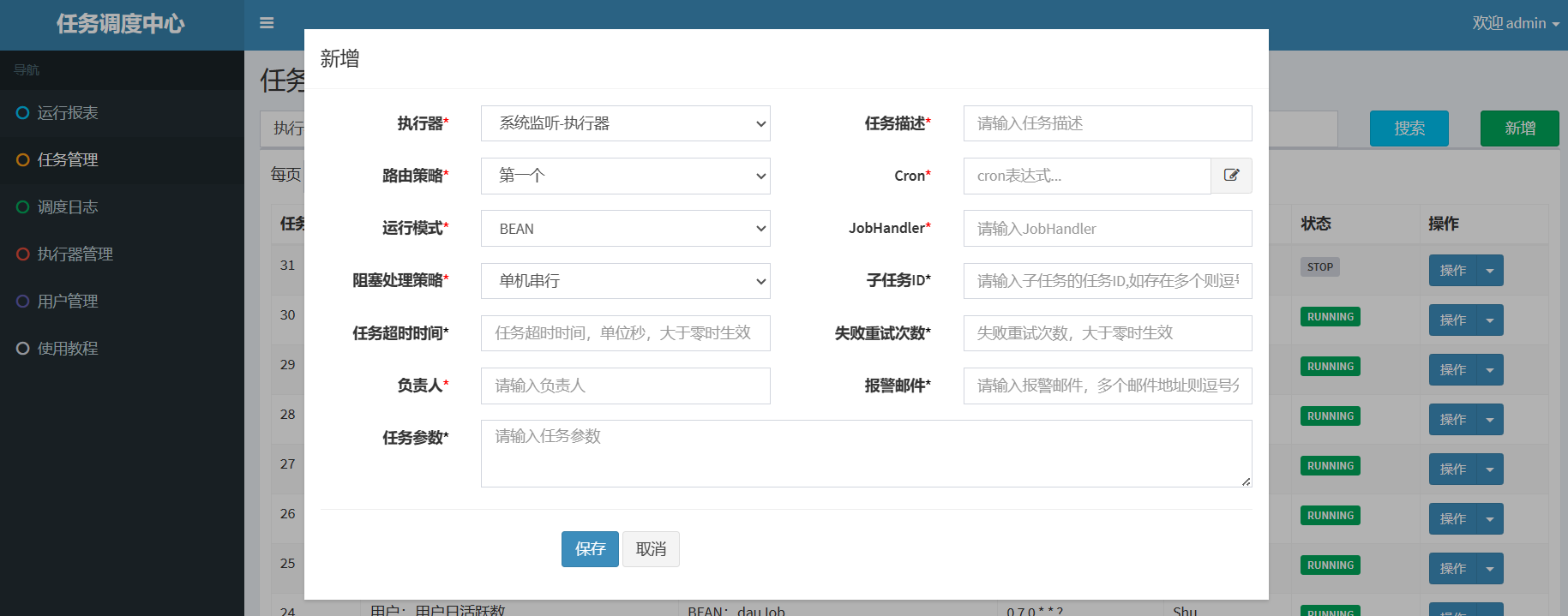
启动执行器之后；就可以在调度中心中 新增执行器；然后再去添加管理要具体执行的任务即可。

**1）新增执行器；（四方保险已经新增）**



**2）新增 任务**





**阻塞处理策略：调度过于密集执行器来不及处理时的处理策略。**

* 单机串行（默认）：调度请求进入单机执行器后，调度请求进入FIFO队列并以串行方式运行；
* 丢弃后续调度：调度请求进入单机执行器后，发现执行器存在运行的调度任务，本次请求将会被丢弃并标记为失败；
* 覆盖之前调度：调度请求进入单机执行器后，发现执行器存在运行的调度任务，将会终止运行中的调度任务并清空队列，然后运行本地调度任务；

**XXL-JOB步骤**：

* 在微服务中引入依赖；
* 配置xxl-job地址
* 编写定时执行的任务；上面需要添加一个注解 @XxlJob（任务名称）
* 访问xxl-job的控制台；添加任务，指定执行时间等信息

**Cron表达式：**

几个域（子表达式） 7个或6个

秒 分 时 日 月 周(一周里面的第几天) 年 注意 日 和 周 必须有一个是?

|  |
| --- |
| \*表示任意，每  / 频率  -范围 , 枚举  ? 日 和 周不能同时使用，如果使用了日则周对应使用? ；如果使用了周，则日使用？；表示不使用该字段；只能使用在周和日两个域中  L 表示最后；放置在日这个域中表示 最后1天  周 从周日开始1-7（周日是一周的第一天，从1开始计算；周一就是2；） |

举例：

2022年12月21日10点30分8秒 8 30 10 21 12 ? 2022

每天某时间点执行任务

每天的 10点50分10秒 执行任务 10 50 10 \* \* ?

每天10点钟至11点钟，每分钟执行一次 0 \* 10 \* \* ?

每月15日11 点执行 0 0 11 15 \* ?

每天12点-1点之间，每间隔10秒执行一次 0/10 \* 12 \* \* ?

每天下午3点20分 第4秒 第7秒 第13秒 第40秒 第55秒 第58秒执行 4,7,13,40,55,58 20 15 \* \* ?

每天下午5点50分 第1秒至第50秒 执行 1-50 50 17 \* \* ?

每天晚上8点-11点 整执行 0 0 20-23 \* \* ?

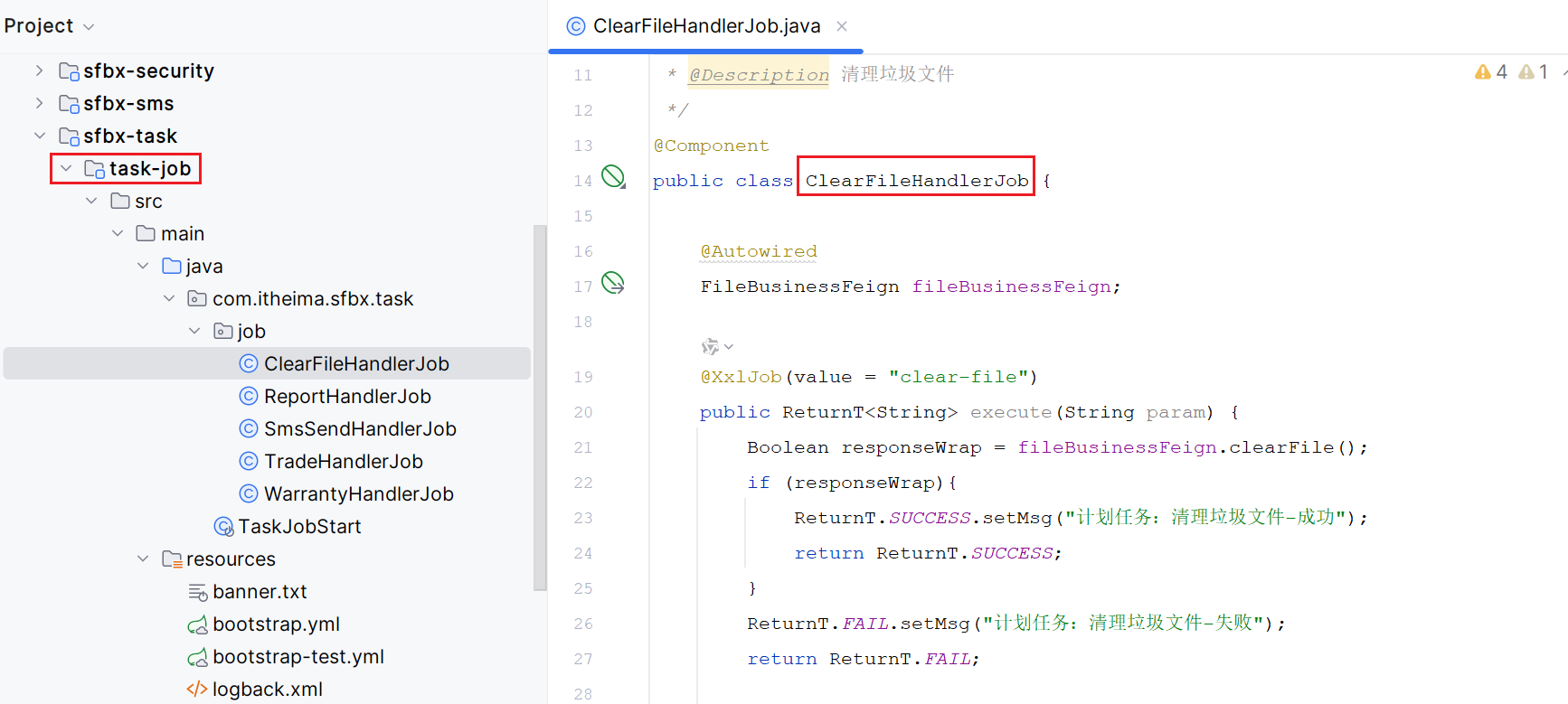
每年奇数月的每一天上午8点至12点 下午 3点至6点 晚上8点 与10点 每间隔2秒执行 0/2 \* 8-12,15-18,20,22 \* 1/2 ?

|  |
| --- |
| **练习**：  1、每个季度的最后一天的下午5点20分发送一个季度报表  2、每周三、周五的上午10点10秒开始每隔3秒执行  3、 2-10 3,5,7,9 \* ? 3-6 1-5 |

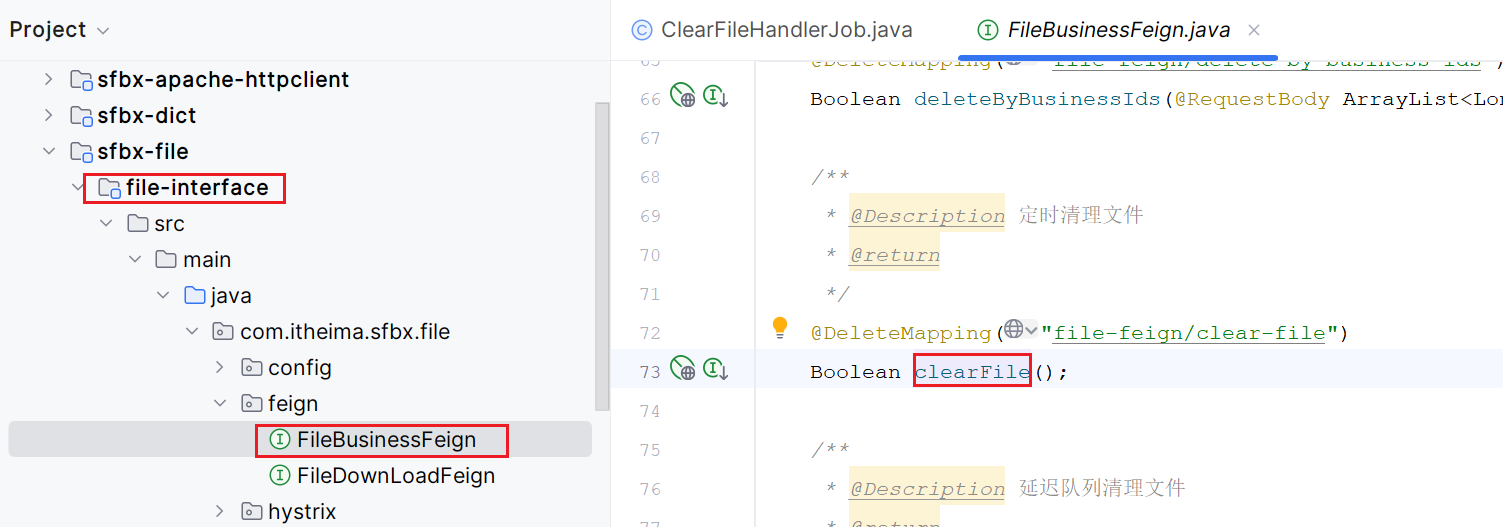
**3.3、定时清理垃圾文件**

使用XXL-JOB每间隔一定时间实现对tab\_file表进行扫描，找出10分钟之内未关联业务记录进行删除。

**1）执行器任务**



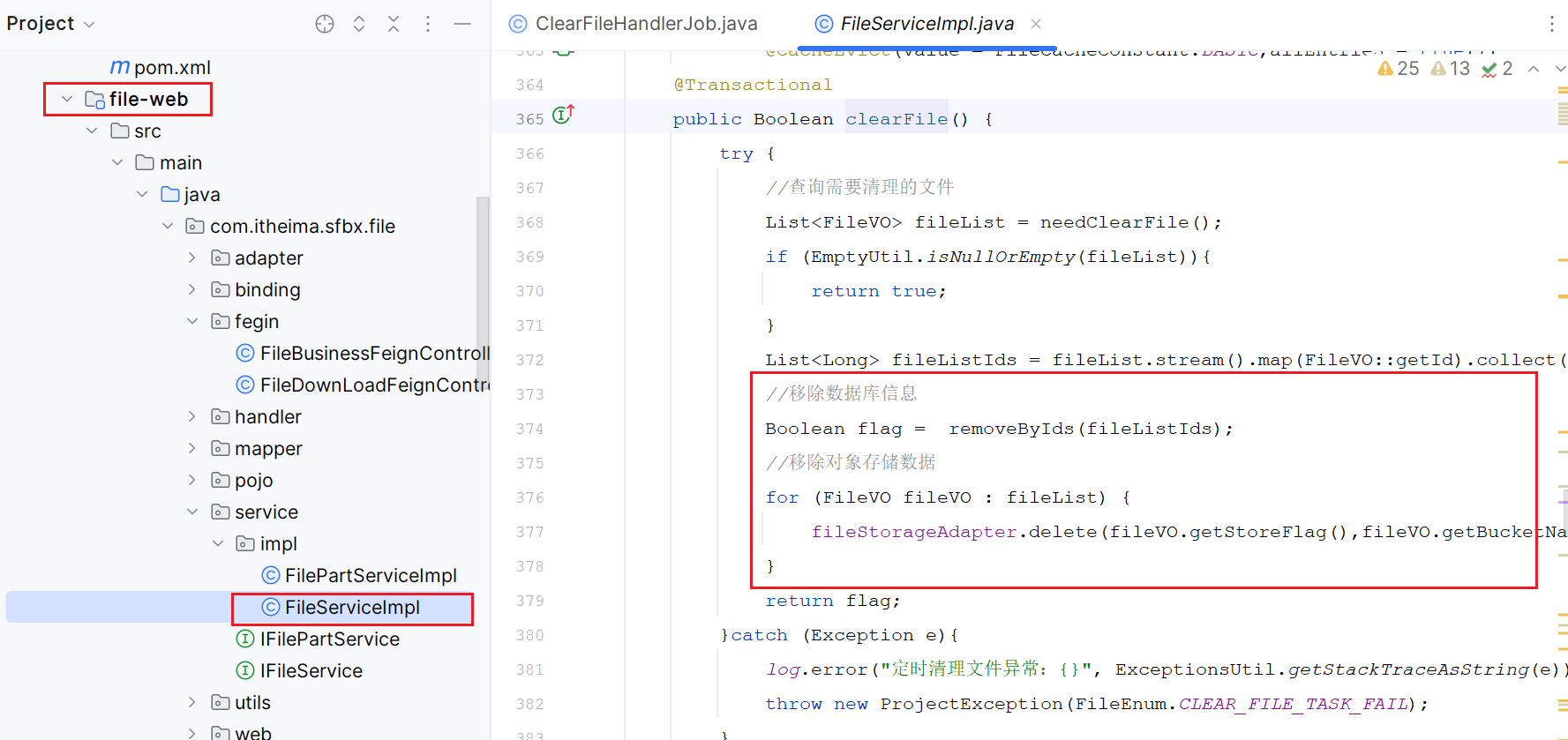
上面代码中 FileBusinessFeign 调用了file-web提供的附件清理远程服务接口：



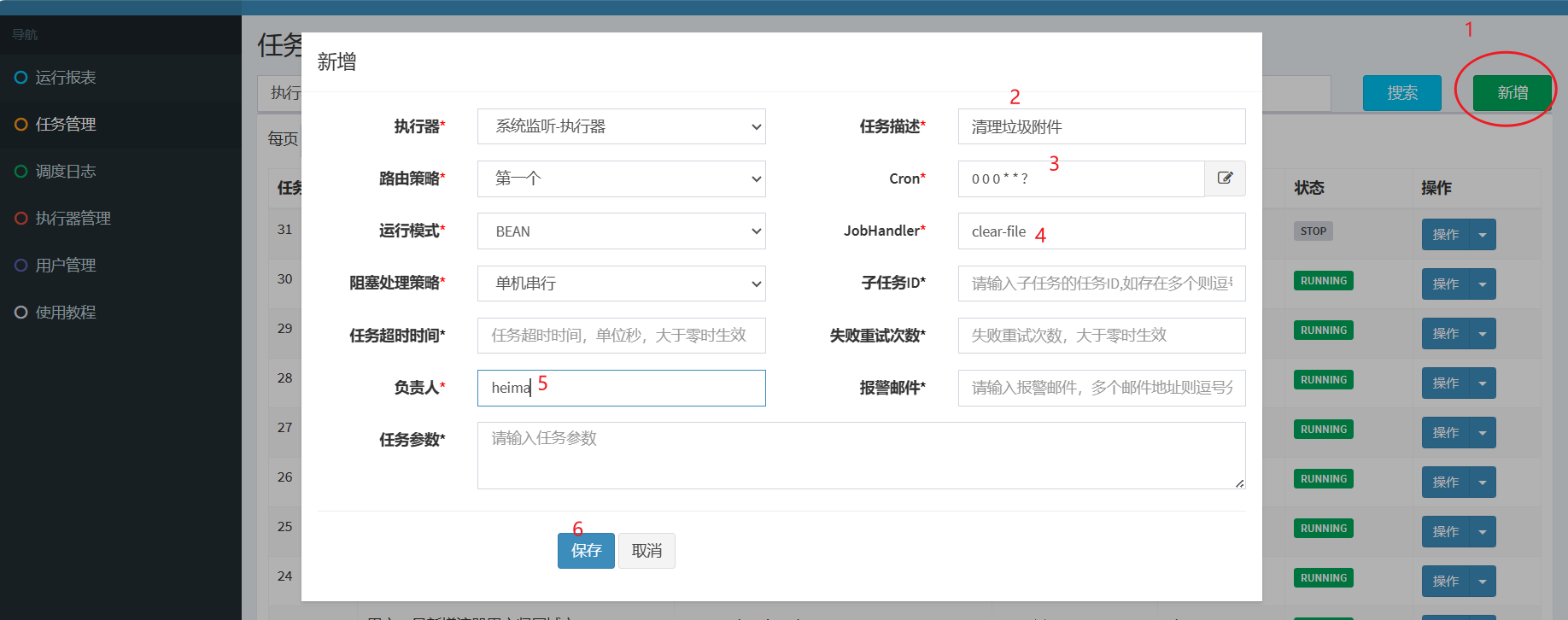
对应的清理接口如下：



在删除垃圾的附件时；不仅删除了数据库中记录还删除了遗留在OSS上的文件。



**2）配置xxl-job任务**



上述配置了每天凌晨执行；如果是配置成每隔10分钟；那么cron表达式为：0 0/10 \* \* \* ?

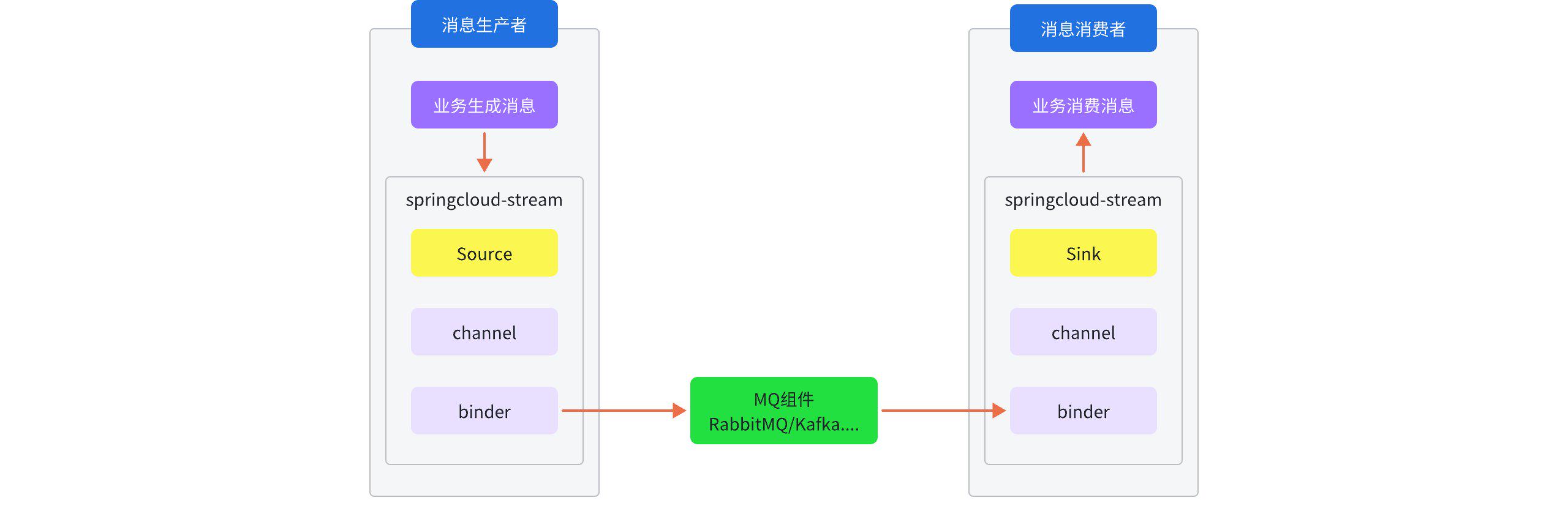
**4、Spring Cloud Stream**

**4.1、简介**

最初我们使用SpringBoot应用要直接与消息中间件进行信息交互的时候，由于各消息中间件构建的初衷不同，它们的实现细节上会有较大的差异性。而SpringCloud-Stream的出现屏蔽底层消息中间件的差异，降低切换版本，统一消息的编程模型的组件，完美地**实现了应用程序与消息中间件细节之间的隔离**。

**Spring Cloud Stream**是一个用于构建 与 共享消息系统 连接的高度可扩展的事件驱动微服务。假设一个开发团队，原来使用的是ActiveMQ；后来发现稳定性和并发能力都更好的RabbitMQ；想要替换的话，那么使用Spring Cloud Stream就只需要修改配置即可，不用修改代码；降低了**更换**消息中间件的成本。**Spring Cloud Stream屏蔽了消息中间件的差异，降低更换中间件的成本，统一消息的编程模型。**

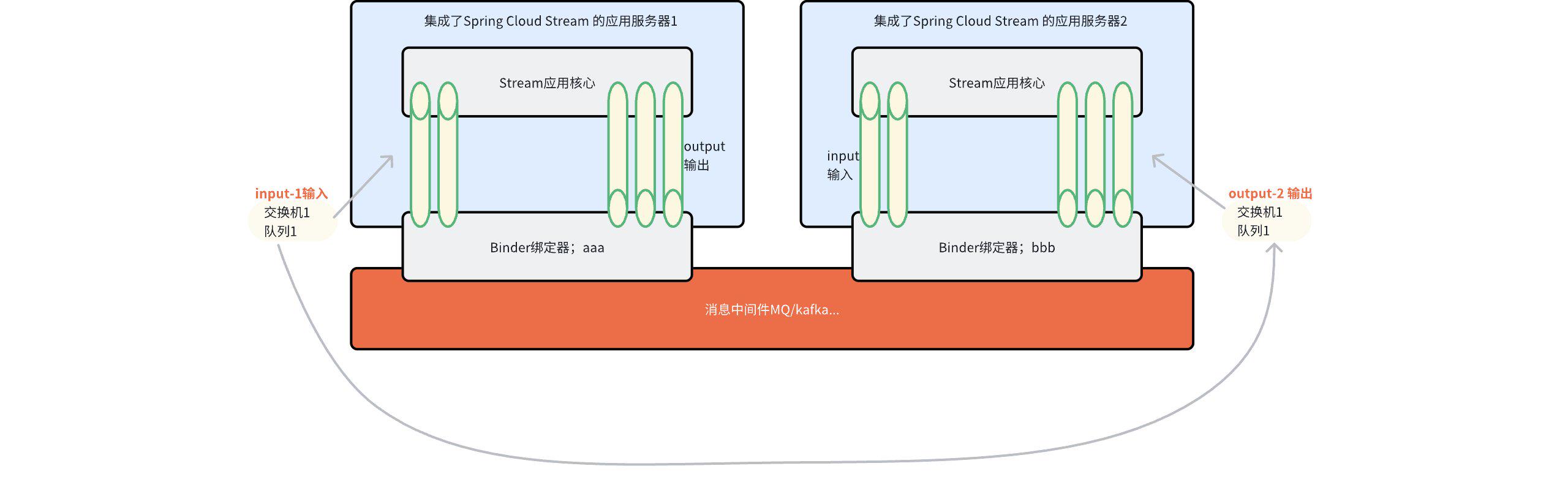
**4.2、架构原理**



* Source：输出消息
* Sink：输入消息
* Channel：通道，是队列 Queue 的一种抽象，在消息通讯系统中就是实现存储和转发的媒介，通过Channel 对队列进行配置；
* Binder：绑定器；用于连接消息中间件，屏蔽 MQ 之间的差异

**4.3、相关概念**

一个Spring Cloud Stream应用程序由一个中间件中立的核心组成。该应用程序通过Spring Cloud Stream注入到其中的输入和输出通道与外界进行通信。应用程序通过 inputs 或者 outputs 来与 SpringCloud Stream 中的 binder（绑定器） 对象交互。通过我们配置来 binding（绑定），而SpringCloud Stream 的 binder 对象负责与消息中间件交互。

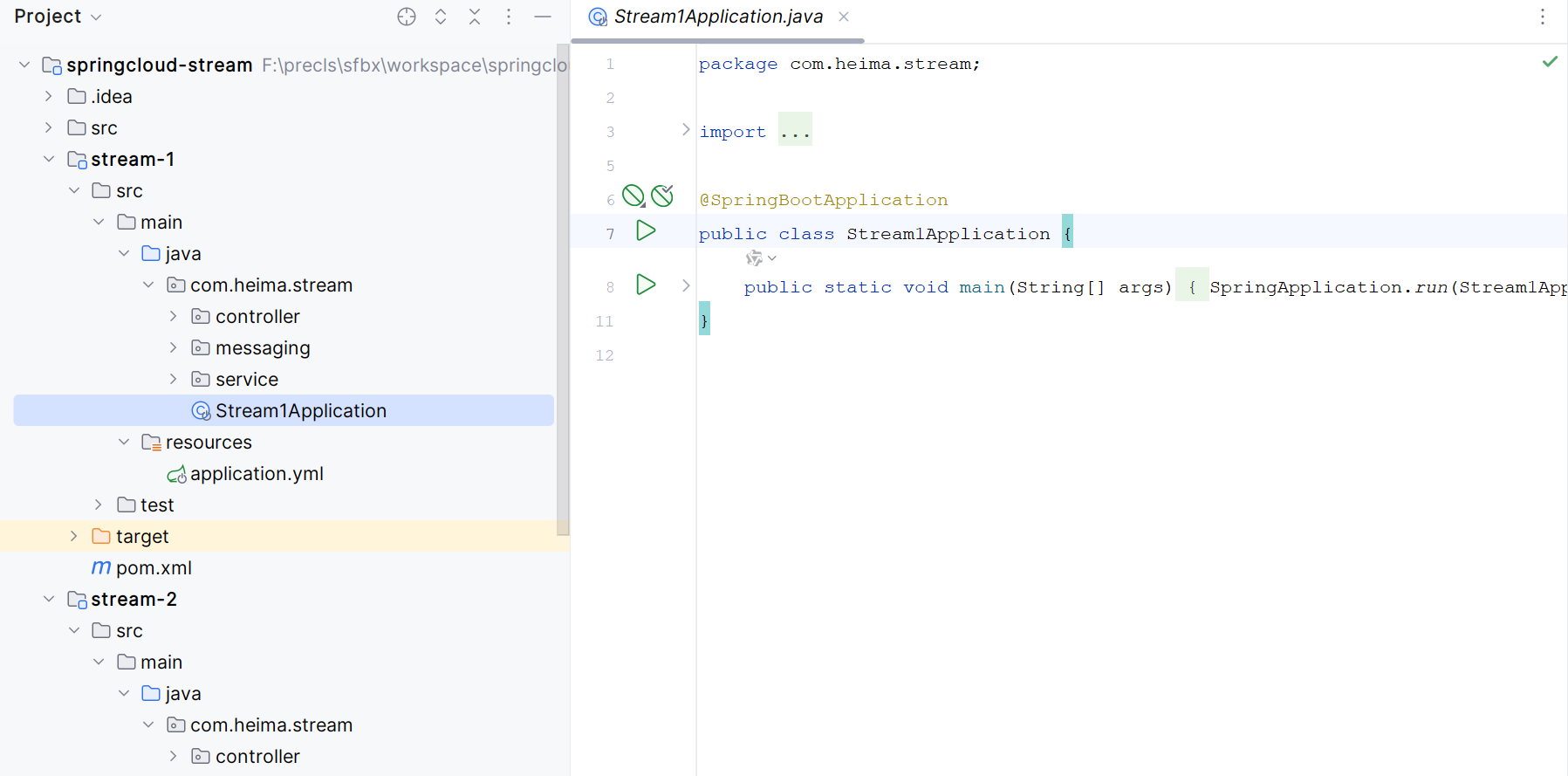


* Binder 绑定器：用于绑定应用服务器与消息中间件；配置文件中可以指定消息队列的连接信息等；可以改变类型（RabbitMQ，kafka）
* input输入：应用服务器从绑定器绑定的通道中接收消息，配置文件中指定交换机、队列；可以使用@Input注解标识
* output输出：应用服务器通过绑定器绑定的通道发送消息，要与input中指定的交换机、队列一致；可以使用@OutPut注解标识
* @StreamListener 监听器，用于接收消费队列中的消息
* @EnableBinding 将通道channel与Exchange绑定在一起

**4.4、快速入门**

**4.4.1、导入工程**

复制 资料/springcloud-stream 工程到工作目录；然后使用new project全新IDEA窗口打开该示例工程；



**4.4.2、发送消息**

**1）配置Stream**

查看阅读：springcloud-stream\stream-1\src\main\resources\application.yml

|  |
| --- |
| YAML server:  port: 8081 spring:  application:  name: stream-1  cloud:  stream:  rabbit:  bindings:  stream1-Input:  consumer:  acknowledge-mode: *manual # 消息手动确认* binders:  defaultRabbitxxx: *# 绑定器名称；自定义* type: rabbit *# 绑定器类型* environment:  spring:  rabbitmq:  host: 192.168.12.129  port: 5672  username: admin  password: pass  vhost: /  bindings:  stream1-Output: *# 发送消息的输出名称；自定义* destination: stream1-destination *# 交换机名称；自定义* group: stream1-queue *# 队列名称；自定义；可以不指定则匿名* content-type: application/json *# 内容类型为json* binder: defaultRabbitxxx *# 绑定器名称与前面配置的要保持一致* stream1-Input: *# 接收消息的输入名称* destination: stream2-exchange  group: stream2-queue  content-type: application/json  binder: defaultRabbitxxx |

**2）编写Source**

在stream中；要发送消息需要使用到Source；但是这个Source默认定义的绑定也就是配置文件中的 stream1-Output 如果写成 output；那么现在这一步 编写Source 就可以省略了；更加简单。但是一般想要自己指定绑定的名称的话；就得重写Source。 按照我们的配置文件；需要写一个：

springcloud-stream\**stream-1**\src\main\java\com\heima\stream\messaging\Stream1Source.java

|  |
| --- |
| Java package com.heima.stream.messaging;  import org.springframework.cloud.stream.annotation.Output; import org.springframework.messaging.MessageChannel;  *// 这个类的编写参考自 org.springframework.cloud.stream.messaging.Source* public interface Stream1Source {   *//管道名称；要与配置文件中的管道名称一致* String *OUTPUT* = "stream1-Output";   *//输出管道* @Output(Stream1Source.*OUTPUT*)  MessageChannel stream1Output(); } |

**3）发送消息**

编写对于的处理器：

springcloud-stream\stream-1\src\main\java\com\heima\stream\controller\Stream1MsgController.java

|  |
| --- |
| Java package com.heima.stream.controller;  import com.heima.stream.service.MessageSendService; import org.springframework.web.bind.annotation.\*;  import javax.annotation.Resource;  @RestController @RequestMapping("/stream") public class Stream1MsgController {   @Resource  private MessageSendService messageSendService;   @GetMapping("/send/{msg}")  public Boolean send(@PathVariable("msg") String msg){  return messageSendService.sendMessage(msg);  }  } |

编写对于的消息发送业务接口、实现类：

springcloud-stream\stream-1\src\main\java\com\heima\stream\service\MessageSendService.java

|  |
| --- |
| Java package com.heima.stream.service;  public interface MessageSendService {   public Boolean sendMessage(String msg);  } |

springcloud-stream\stream-1\src\main\java\com\heima\stream\service\impl\MessageSendServiceImpl.java

|  |
| --- |
| Java package com.heima.stream.service.impl;  import com.heima.stream.messaging.Stream1DelaySource; import com.heima.stream.messaging.Stream1Source; import com.heima.stream.service.MessageSendService; import org.springframework.cloud.stream.annotation.EnableBinding; import org.springframework.messaging.Message; import org.springframework.messaging.support.MessageBuilder; import org.springframework.stereotype.Service;  import javax.annotation.Resource;  @Service *//定义消息的推送和接收管道；如果使用Source则表示输出，且输出管道名称为output；* *//也可以绑定使用Sink则表示输入，且输入管道名称为input* *//@EnableBinding(Source.class)* *//定义消息的推送和接收管道；且自定义输入、输出的管道名称* @EnableBinding({Stream1Source.class}) public class MessageSendServiceImpl implements MessageSendService {   @Resource  private Stream1Source stream1Source;   @Resource  private Stream1DelaySource stream1DelaySource;   @Override  public Boolean sendMessage(String msg) {  Message<String> stringMessage = MessageBuilder.*withPayload*(msg).build();  return stream1Source.stream1Output().send(stringMessage);  }    } |

|  |
| --- |
| 上述的 @EnableBinding 注解很关键；   * 它是一个复合注解，包含了@Configuration其实在上面这个类上是可以不需要@Service了 * 它可以定义Source 输出管道那么就是发送消息 * 它可以定义Sink 输入管道就是接收消息的 |

**4.4.3、接收消息**

**1）配置Stream**

springcloud-stream\**stream-2**\src\main\resources\application.yml 内容如下：

|  |
| --- |
| YAML server:  port: 8082 spring:  application:  name: stream-2  cloud:  stream:  binders:  defaultRabbit: *# 绑定器名称；自定义* type: rabbit *# 绑定器类型* environment:  spring:  rabbitmq:  host: 192.168.12.129  port: 5672  username: admin  password: pass  vhost: /  bindings:  stream2-Output: *# 发送消息的输出名称；自定义* destination: stream2-exchange *# 交换机名称；自定义* group: stream2-queue *# 队列名称；自定义* content-type: application/json  binder: defaultRabbit  stream2-Input: *# 接收消息的输入名称；自定义* destination: stream1-destination # 这个与发送时要一样  group: stream1-queue # 这个与发送时要一样  content-type: application/json # 这个与发送时要一样  binder: defaultRabbit  rabbit:  bindings:  stream2-Input:  consumer:  acknowledge-mode: *manual # 消息手动确认* |

**2）编写Sink**

springcloud-stream\stream-2\src\main\java\com\heima\stream\messaging\Stream2Sink.java

|  |
| --- |
| Java package com.heima.stream.messaging;  import org.springframework.cloud.stream.annotation.Input; import org.springframework.messaging.SubscribableChannel;  public interface Stream2Sink {  *// 定义输入管道名称；要与配置文件中的保持一致* String *INPUT* = "stream2-Input";   *//输入管道* @Input(Stream2Sink.*INPUT*)  SubscribableChannel stream2Input(); } |

**3）接收消息**

springcloud-stream\stream-2\src\main\java\com\heima\stream\listener\MessageListener.java

|  |
| --- |
| Java package com.heima.stream.listener;  import com.heima.stream.messaging.Stream2DelaySink; import com.heima.stream.messaging.Stream2Sink; import lombok.extern.slf4j.Slf4j; import org.springframework.cloud.stream.annotation.EnableBinding; import org.springframework.cloud.stream.annotation.StreamListener; import org.springframework.stereotype.Component;  @Component @EnableBinding({Stream2Sink.class}) @Slf4j public class MessageListener {   @StreamListener(Stream2Sink.*INPUT*)  public void receive(String msg) {  *log*.info("stream2接收到消息：" + msg);  }  } |

|  |
| --- |
| 小扩展练习：  在stream-2中发送一个map结构的消息到stream-1 |

参考接收map的消息监听器的代码如下：

|  |
| --- |
| Java package com.heima.stream.listener;  import com.heima.stream.messaging.Stream1Sink; import com.rabbitmq.client.Channel; import lombok.extern.slf4j.Slf4j; import org.springframework.amqp.support.AmqpHeaders; import org.springframework.cloud.stream.annotation.EnableBinding; import org.springframework.cloud.stream.annotation.StreamListener; import org.springframework.messaging.handler.annotation.Header; import org.springframework.stereotype.Component;  import java.io.IOException; import java.util.Map;  @Component @EnableBinding(Stream1Sink.class) @Slf4j public class MessageListener {   @StreamListener(Stream1Sink.*INPUT*)  public void receive(Map<String, Object> map,  @Header(AmqpHeaders.*CHANNEL*) Channel channel,  @Header(AmqpHeaders.*DELIVERY\_TAG*) Long deliveryTag) throws IOException {  *log*.info("接收到消息：{}", map);  *//引入了手动确认机制，所以需要手动确认；如果没有手动确认，*  *// 消息会一直处于未确认的状态，一直处于队列中，直到超时* channel.basicAck(deliveryTag, false);  } } |

**4.4.4、延迟消息**

**1）发送方配置**

springcloud-stream\stream-1\src\main\resources\application.yml 修改为如下：

|  |
| --- |
| YAML server:  port: 8081 spring:  application:  name: stream-1  cloud:  stream:  rabbit:  bindings:  stream1-Input:  consumer:  acknowledge-mode: *manual # 消息手动确认* stream1-delay-Output:  producer:  delayed-exchange: true *# 开启延迟交换机* binders:  defaultRabbitxxx: *# 绑定器名称；自定义* type: rabbit *# 绑定器类型* environment:  spring:  rabbitmq:  host: 192.168.12.129  port: 5672  username: admin  password: pass  vhost: /  bindings:  stream1-Output: *# 发送消息的输出名称；自定义* destination: stream1-destination *# 交换机名称；自定义* group: stream1-queue *# 队列名称；自定义；可以不指定则匿名* content-type: application/json *# 内容类型为json* binder: defaultRabbitxxx *# 绑定器名称与前面配置的要保持一致* stream1-delay-Output: *# 发送消息的输出名称；自定义* destination: stream1-delay-exchange *# 交换机名称；自定义* group: stream1-delay-queue *# 队列名称；自定义* content-type: application/json  binder: defaultRabbitxxx  stream1-Input: *# 接收消息的输入名称* destination: stream2-exchange  group: stream2-queue  content-type: application/json  binder: defaultRabbitxxx |

|  |
| --- |
| 上述文件中最重要的是要开启发送方 延迟队列 delayed-exchange: true |

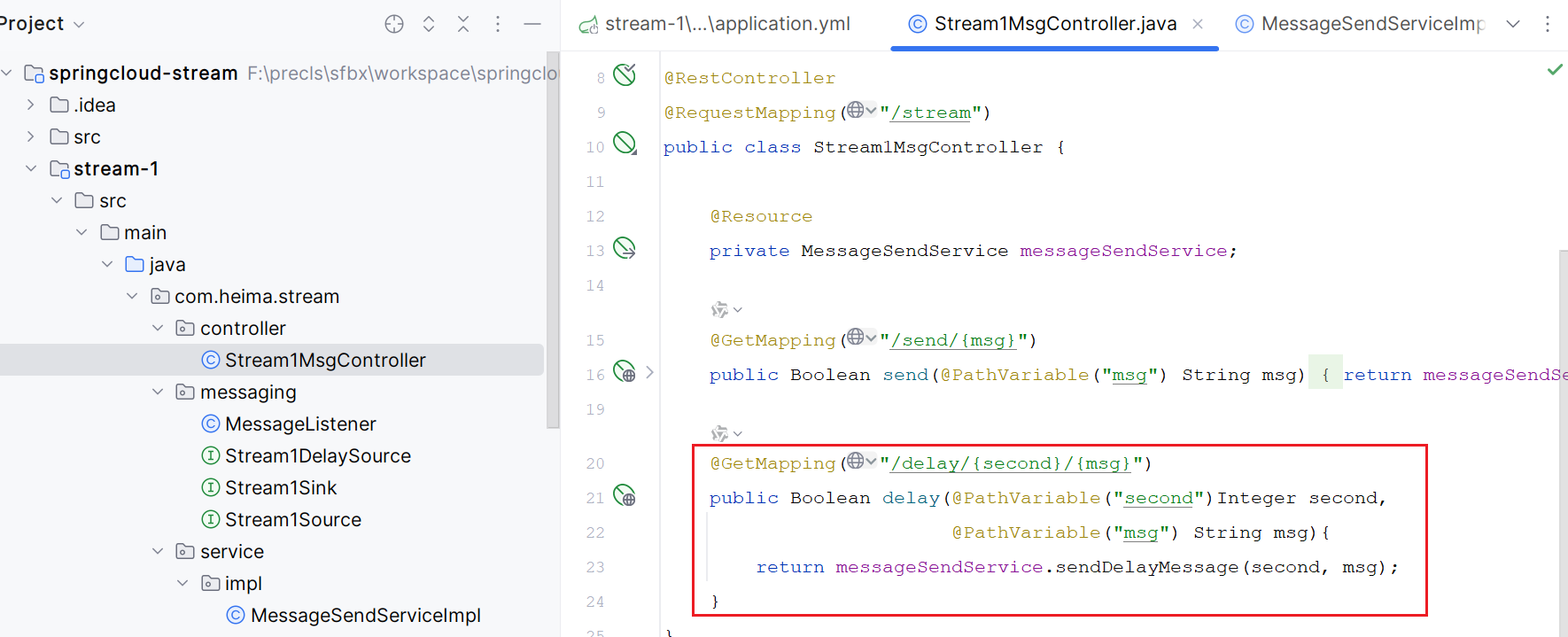
**2）发送延迟消息**

**定义Source类**：

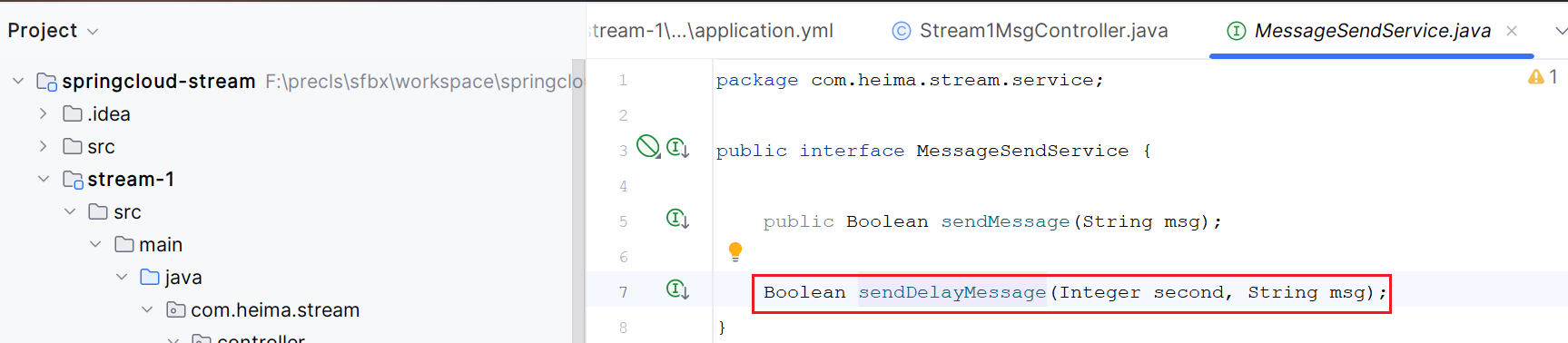
springcloud-stream\stream-1\src\main\java\com\heima\stream\messaging\Stream1DelaySource.java

|  |
| --- |
| Java package com.heima.stream.messaging;  import org.springframework.cloud.stream.annotation.Output; import org.springframework.messaging.MessageChannel;  public interface Stream1DelaySource {   *//管道名称；要与配置文件中的管道名称一致* String *OUTPUT* = "stream1-delay-Output";   *//输出管道* @Output(Stream1DelaySource.*OUTPUT*)  MessageChannel delayOutput(); } |

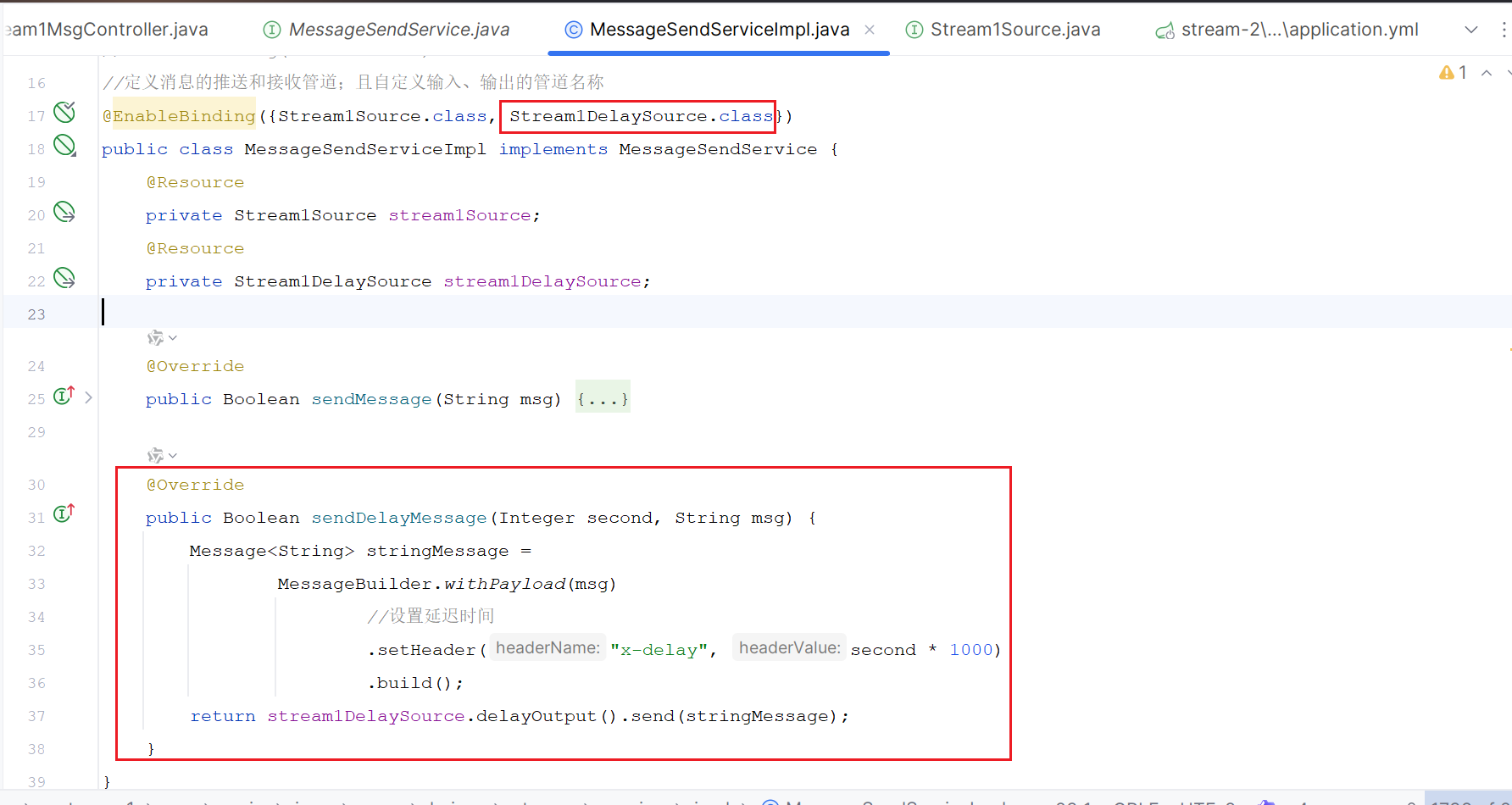
**处理器方法**：



**接口类**：



**实现类**：



**3）接收方配置**

修改配置文件

springcloud-stream\stream-2\src\main\resources\application.yml 为如下：

|  |
| --- |
| YAML server:  port: 8082 spring:  application:  name: stream-2  cloud:  stream:  binders:  defaultRabbit: *# 绑定器名称；自定义* type: rabbit *# 绑定器类型* environment:  spring:  rabbitmq:  host: 192.168.12.129  port: 5672  username: admin  password: pass  vhost: /  bindings:  stream2-Output: *# 发送消息的输出名称；自定义* destination: stream2-exchange *# 交换机名称；自定义* group: stream2-queue *# 队列名称；自定义* content-type: application/json  binder: defaultRabbit  stream2-Input: *# 接收消息的输入名称；自定义* destination: stream1-destination *# 这个与发送时要一样* group: stream1-queue *# 这个与发送时要一样* content-type: application/json *# 这个与发送时要一样* binder: defaultRabbit  stream2-delay-Input: *# 接收消息的输入名称* destination: stream1-delay-exchange  group: stream1-delay-queue  content-type: application/json  binder: defaultRabbit  rabbit:  bindings:  stream2-delay-Input:  consumer:  acknowledge-mode: *manual # 消息手动确认* delayed-exchange: true *# 开启延迟交换机* stream2-Input:  consumer:  acknowledge-mode: *manual # 消息手动确认* |

|  |
| --- |
| 上述配置中，重要是要开启消息input消费要开启延迟队列 |

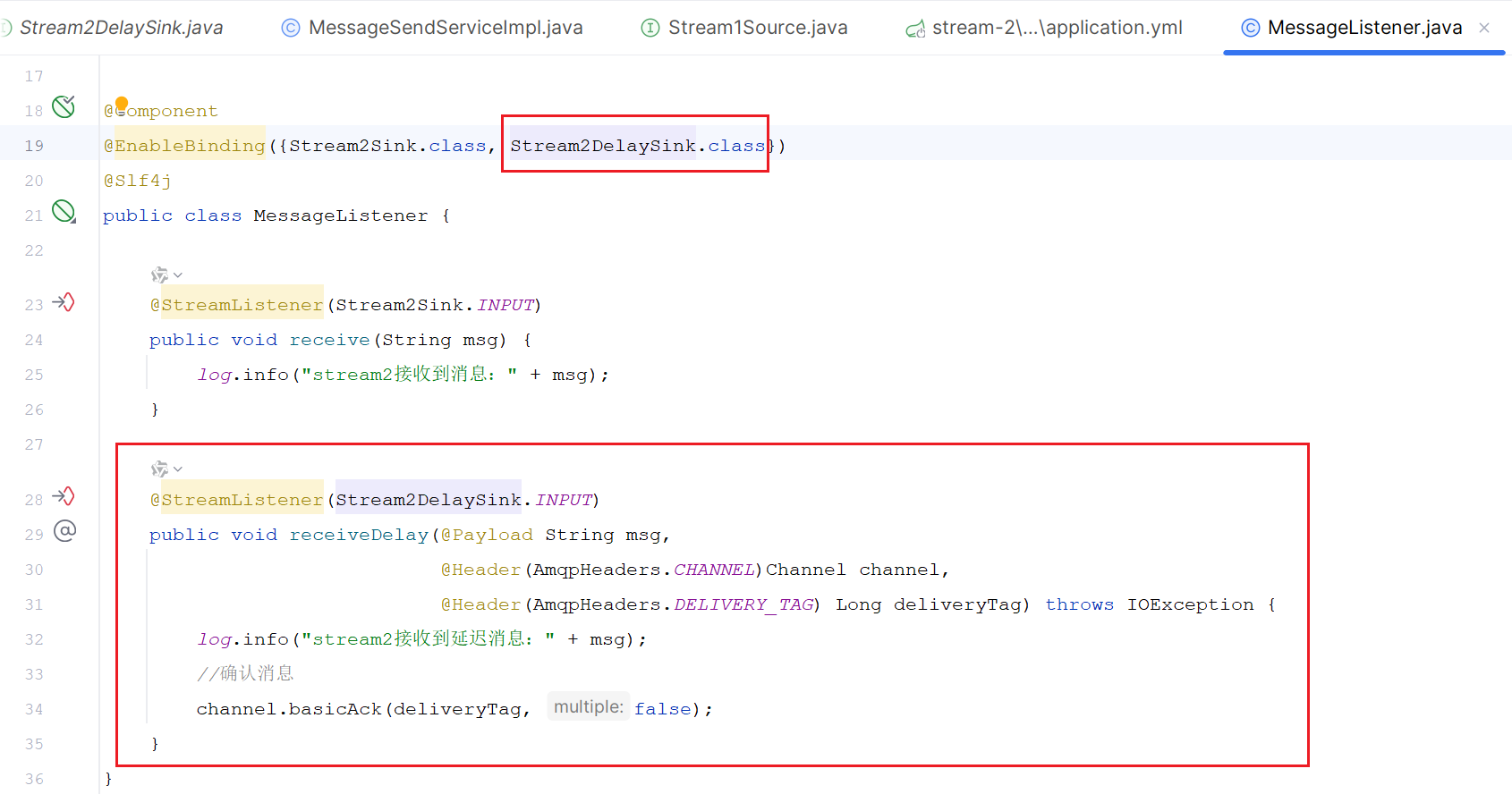
**4）接收延迟消息**

**编写Sink类**：springcloud-stream\stream-2\src\main\java\com\heima\stream\messaging\Stream2DelaySink.java

|  |
| --- |
| Java package com.heima.stream.messaging;  import org.springframework.cloud.stream.annotation.Input; import org.springframework.messaging.SubscribableChannel;  public interface Stream2DelaySink {  *// 定义输入管道名称；要与配置文件中的保持一致* String *INPUT* = "stream2-delay-Input";   *//输入管道* @Input(Stream2DelaySink.*INPUT*)  SubscribableChannel stream2Input(); } |

**编写消息监听方法**：

springcloud-stream\stream-2\src\main\java\com\heima\stream\listener\MessageListener.java



**4.5、延迟清理垃圾文件**

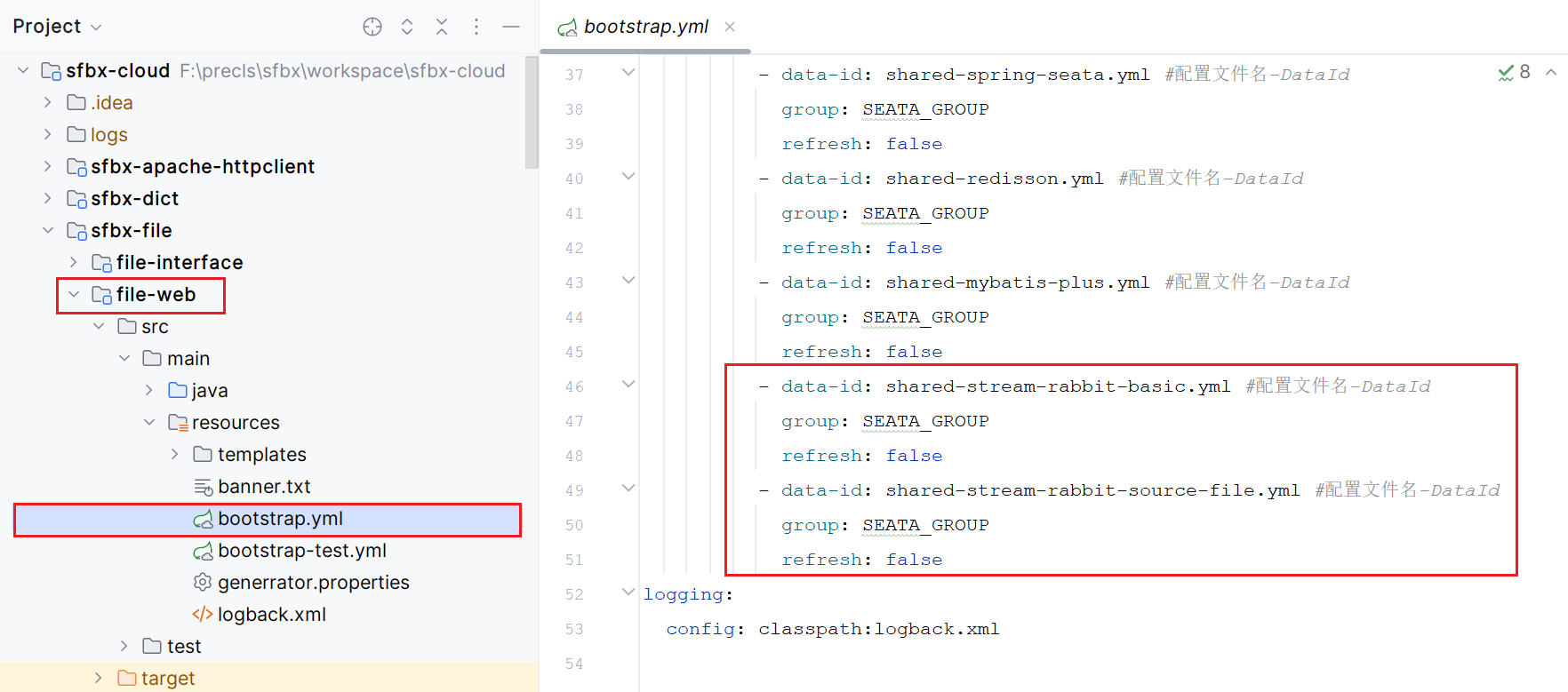
在学习了解了Spring Cloud Stream之后；回到 四方保险；清理垃圾文件的另外一种方案。

当文件上传对象存储且保存数据库后，上传方法最后会发送延迟信息当前文件记录信息到RabbitMQ中，task-listener10分钟后消费到消息，然后通过openfegin调用file-web中的查询处理方法，如果当前文件记录还是没有businessId,则认为是垃圾文件进行删除。

**4.5.1、消息发送端**

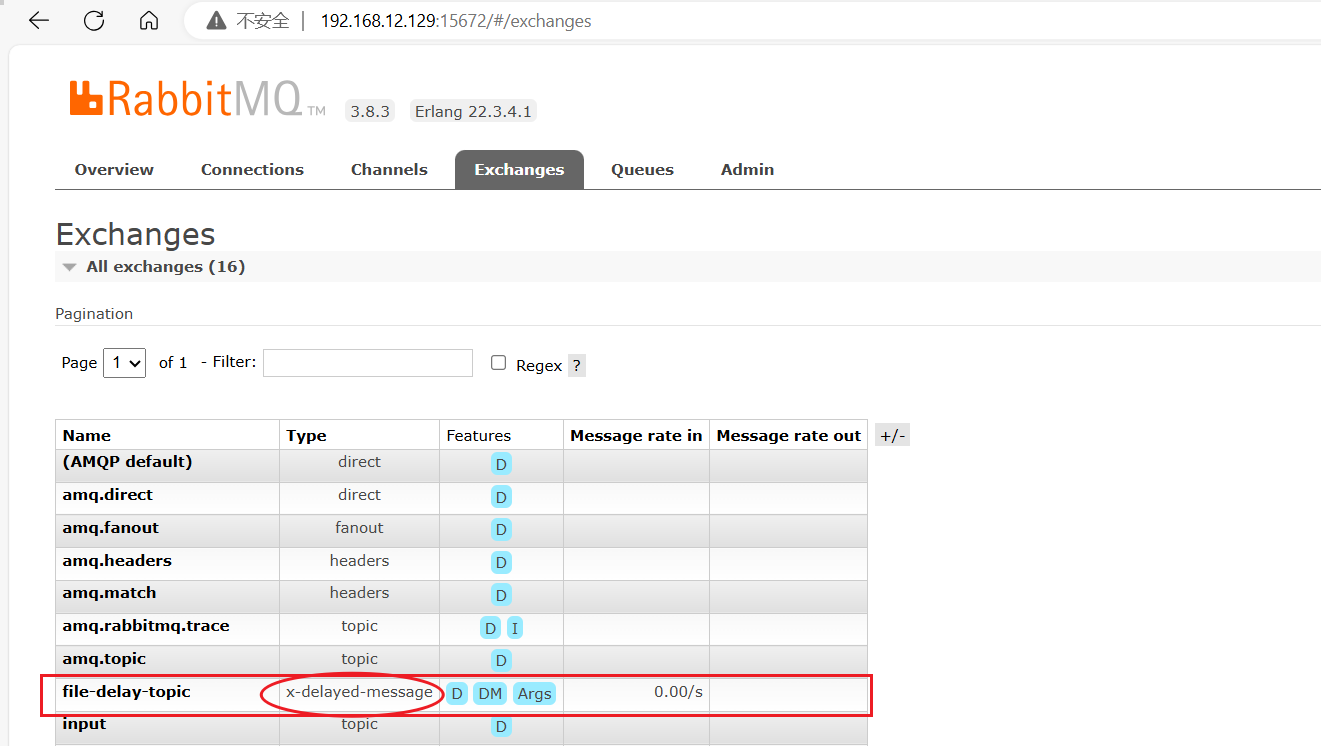
**1）配置Stream**

在file-web模块的bootstrap.yml中引入关于SpringCloud Stream配置。



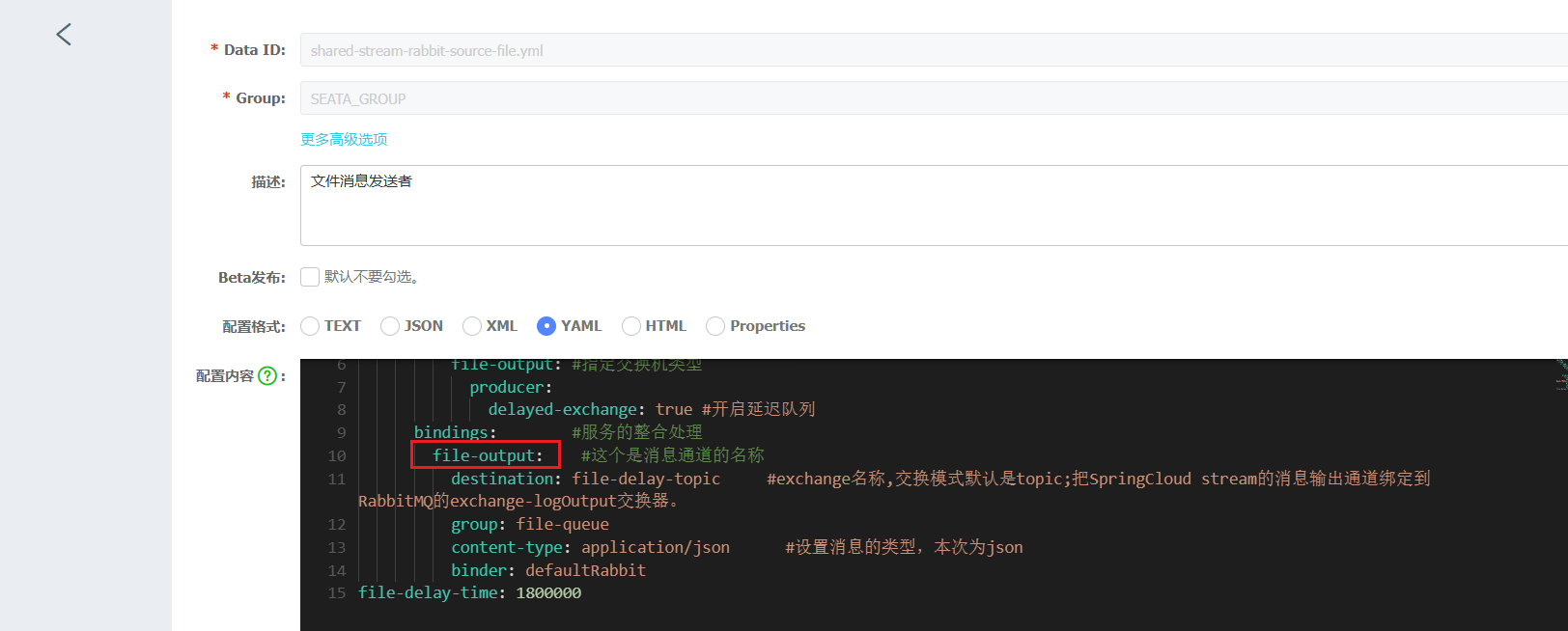
上述的两个文件都已经放置在nacos配置中心中管理；可以访问nacos查阅对应的文件。

项目启动后可以打开RabbitMQ的管理界面http://192.168.12.129:15672查看：



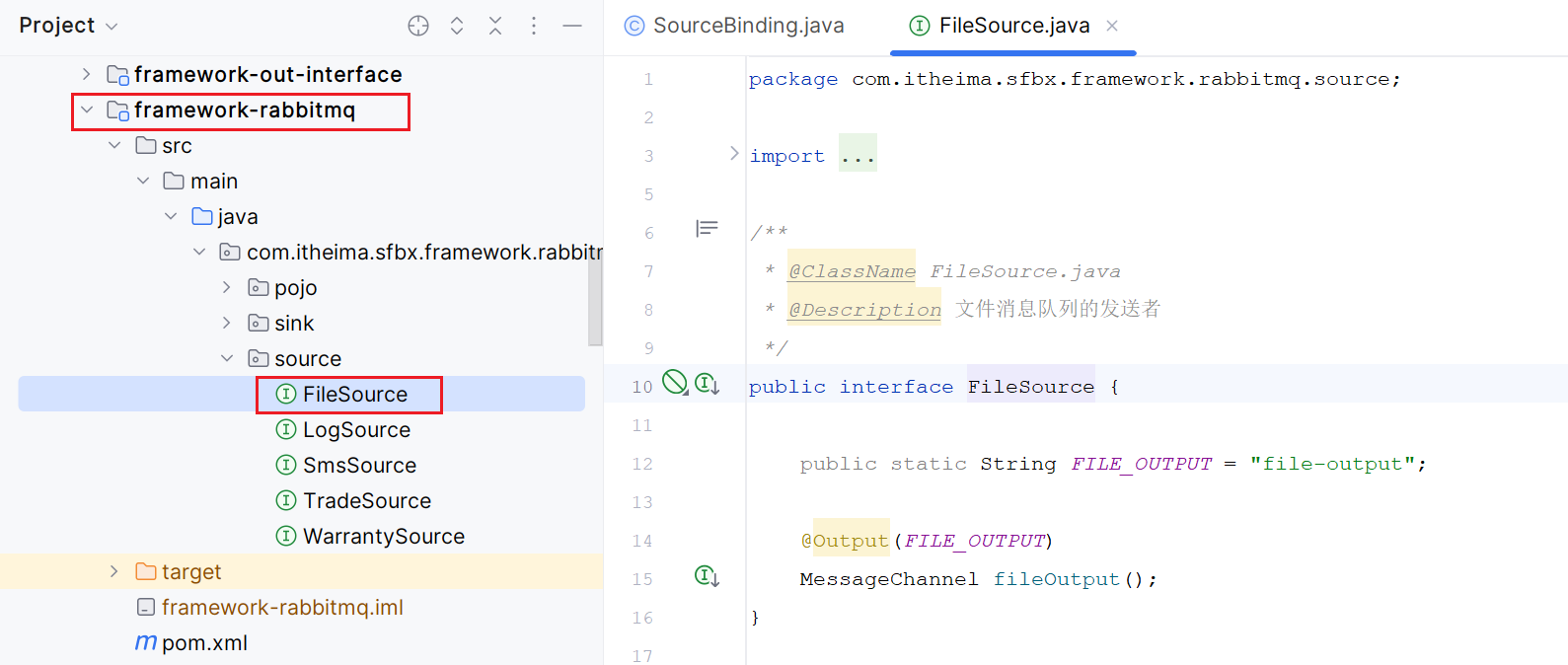
**2）定义Source**

根据nacos中配置文件指定的output；



编写对应的Source类的绑定类和FileSource类如下：





**3）发送消息**



**4.5.2、消息接收端**

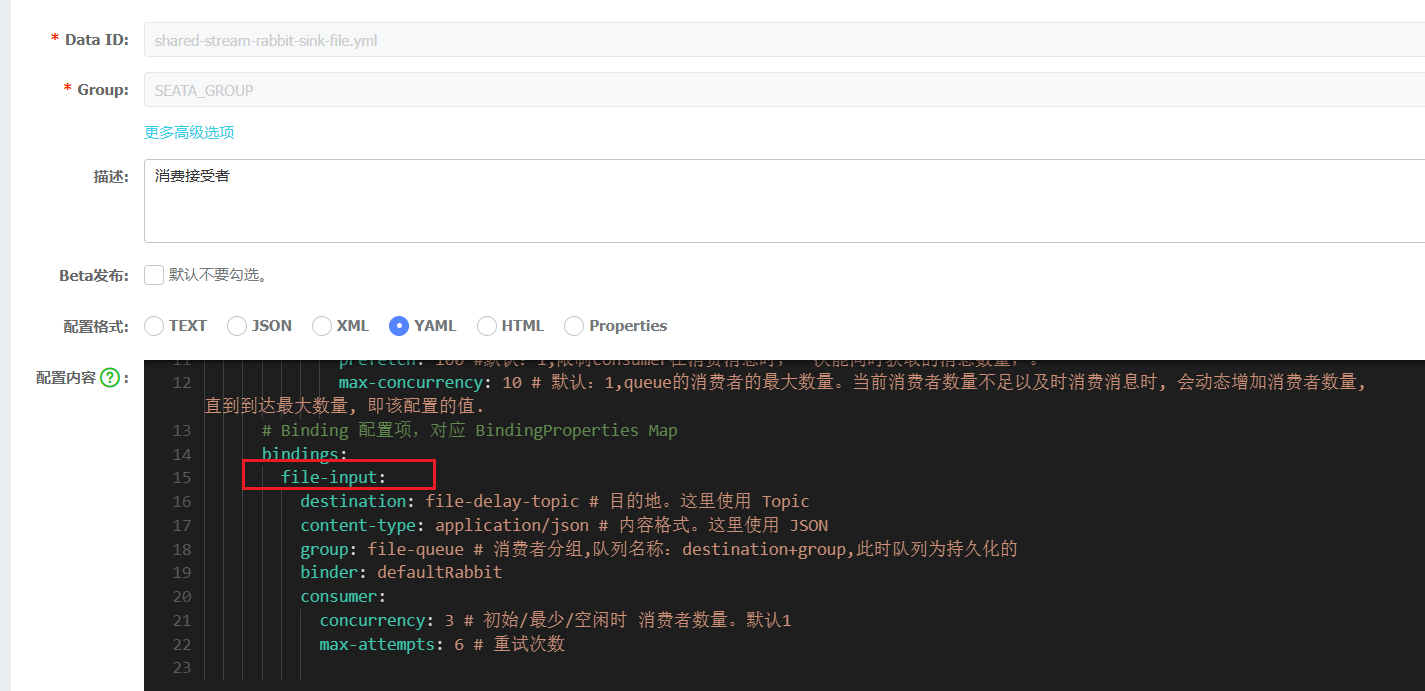
**1）配置Stream**

在task-listener模块的bootstrap.yml中引入关于Spring Cloud Stream配置。



**2）定义Sink**

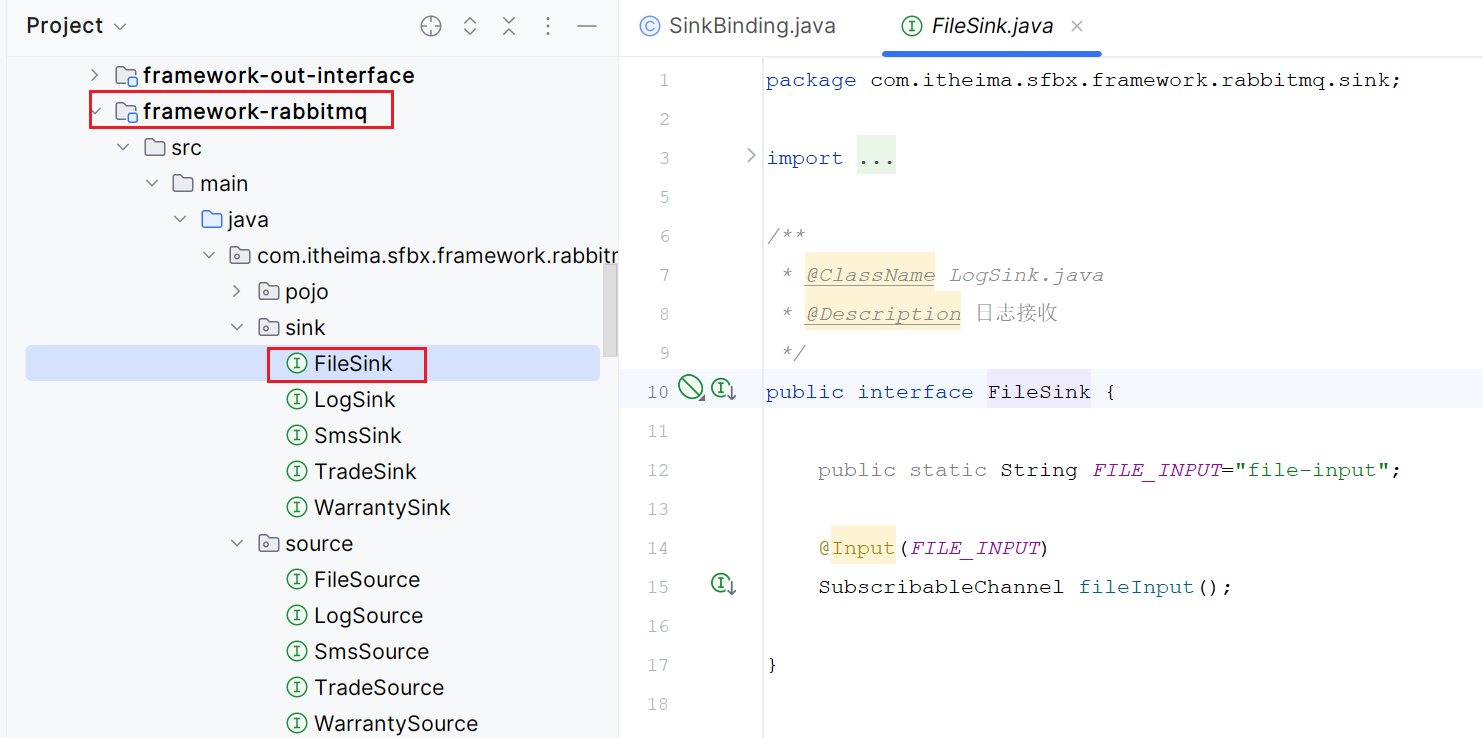
根据nacos中配置文件shared-stream-rabbit-sink-file.yml指定的input；



编写对应的Sink绑定类和Sink类如下：



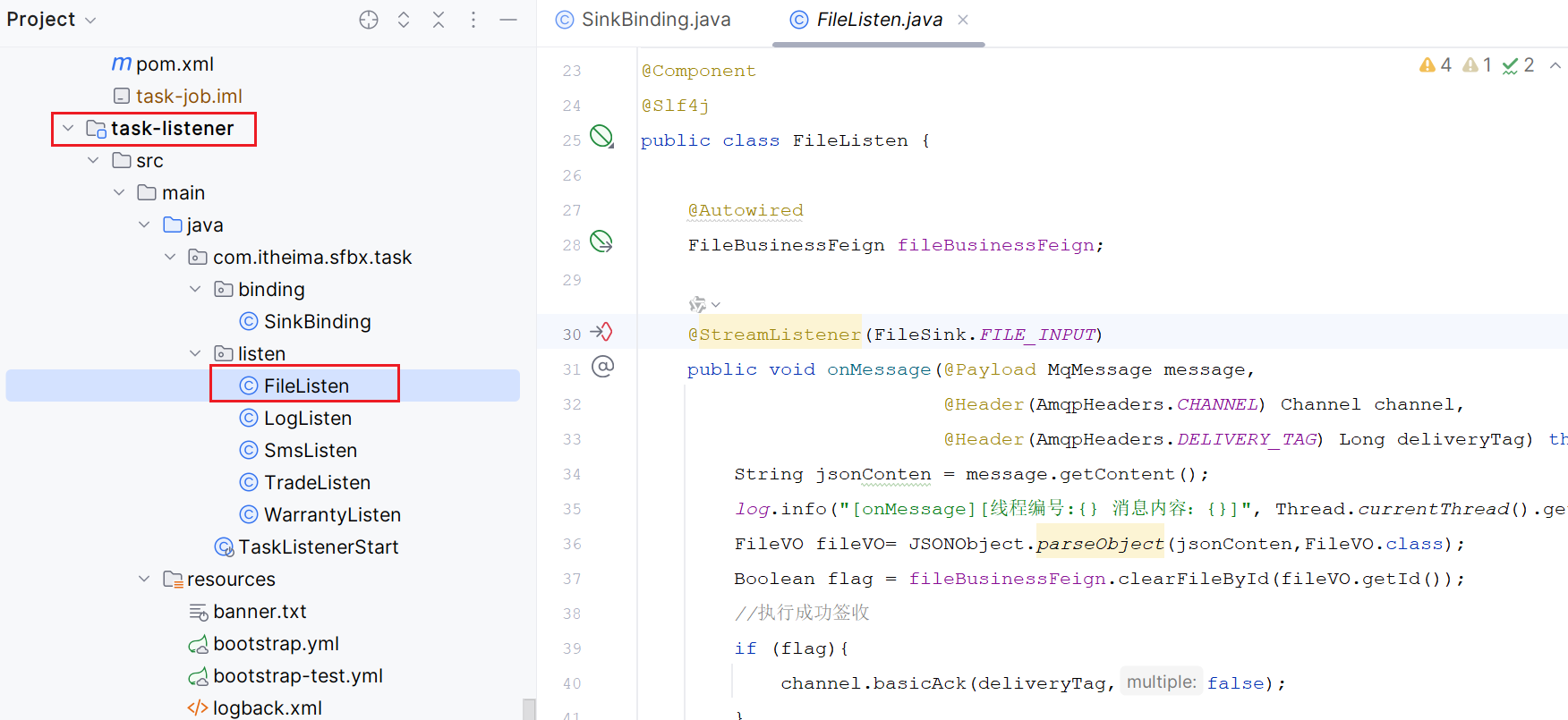
FileSink类如下：



**3）接收消息**

消息的接收处理统一在 task-listener 中处理：

sfbx-cloud\sfbx-task\task-listener\src\main\java\com\itheima\sfbx\task\listen\FileListen.java



**5、作业&问题**

针对今天学习和应用的 对象存储 技术；需要回答一些关于这一方面的常见面试问题：

* **作业**：自己的在线文档中；总结分片上传的流程、步骤，晚上找同学说出来（组长检查）
* **作业**：站在你们在哪里使用到了XXL-JOB的面试问题的角度；总结分析如何利用定时任务清理垃圾文件
* **作业**：站在你们在哪里使用到了RabbitMQ的面试问题的角度；总结分析如何利用MQ的延迟消息清理垃圾文件
* 为什么分片文件上传的时候需要将分片文件上次每次经过 后端微服务？而不是直接上传到oss？
* 说出分片上传的流程
* 如何判断是否重复上传同一个文件，也就是说怎么判断两个文件是否相同？
* 怎么校验文件是否为恶意文件？
* 说出保险产品附件上传的文件为什么需要清理，如何清理
* XXL-JOB 的阻塞处理策略说一下
* Cron表达式格式说明一下；一般你们的定时任务是什么时候执行
* Spring Cloud Stream怎么使用（包括发送和接收消息）
* **简历职责-候选**：解决保险产品附件保险协议、合同文件过大上传不稳定的问题；
* **简历职责-候选**：引入并搭建任务调度组件和新建任务调度微服务实现产品附件上传到OSS后未关联保险产品的无效附件问题；