**详细设计文档**

**仰望星空小组**

目录

[**1 项目技术栈简介** 1](#_Toc108183953)

[**2 视图层技术说明** 1](#_Toc108183954)

[**3 业务层技术说明** 5](#_Toc108183955)

[**3.1 Spring Cloud框架及其相关技术** 5](#_Toc108183956)

[**3.1.1 项目整体目录结构** 5](#_Toc108183957)

[**3.1.2 nacos使用及其配置** 6](#_Toc108183958)

[**3.1.3 gatway使用及其配置** 6](#_Toc108183959)

[**3.2 swagger接口文档及接口设计详细说明** 7](#_Toc108183960)

[**3.2.1 swagger相关配置** 7](#_Toc108183961)

[**3.2.2单次上传接口代码设计说明** 8](#_Toc108183962)

[**3.2.3批量上传接口代码设计说明** 10](#_Toc108183963)

[**3.3 数据库相关配置代码** 12](#_Toc108183964)

[**3.3.1 Java连接oss服务及其配置** 12](#_Toc108183965)

[**3.3.2 Jpa连接数据库及其配置** 13](#_Toc108183966)

[**4 模型层技术说明** 15](#_Toc108183967)

[**4.1 flask模型层接口的设计** 15](#_Toc108183968)

[**4.2 四种模型训练和推理的相关代码说明** 17](#_Toc108183969)

[**4.2.1 模型训练** 17](#_Toc108183970)

[**4.2.2 模型部署** 22](#_Toc108183971)

[**4.2.2.2 地物分类** 23](#_Toc108183972)

[**4.2.2.3 目标检测和目标提取** 23](#_Toc108183973)

**1 项目技术栈简介**

本系统可分为视图层、业务层和模型层，各个层次的相关的技术栈如下所示：

**视图层：**

* Vue框架：构建前端页面的主体框架。
* html/css/js语言：用于编写前端页面内容、风格，以及与后端的交互。
* Element UI组件库：用于简化前端交互组件的设计。
* brower.js 的oss连接接口：实现对于大文件分片上传与下载。

**业务层：**

* Spring cloud框架：用于构建后端微服务的主体框架。
* swagger：动态接口文档，便于前后端的接口交互。
* Spring cloud gateway：有关微服务框架的网关设计，实现端口统一、熔断过滤等功能。
* nacos：完成对于微服务的统一配置管理。
* Jpa：用于连接mysql数据库的中间件，实现ORM映射等功能。
* 关系型数据库mysql：实现对用户信息的持久化储存。
* 对象存储oss：实现对批处理大文件的持久化存储。

**模型层：**

* 动态模型训练：使用百度AI Studio进行模型训练。
* 静态模型部署与推理：采用python脚本推理出结果图片。
* flask工具：用户提供业务层所需要的推理接口。

有关系统整体架构以及各种技术的概要设计请详见《概要设计文档》，下面将对各个层次核心技术的设计进行详细说明。

**2 视图层技术说明**

视图层系统使用Vue框架并引入了Element UI 组件，并通过Brower.js 连接oss云服务，下面将详细介绍连接oss云服务中使用的技术。

首先，代码将创建oss客户端连接封装为一个接口，代码如下：

|  |
| --- |
| JavaScript import OSS from "ali-oss"  export default function Client(data = 'custom-data') {  // console.log('bucket',data)  return new OSS({  // yourRegion填写Bucket所在地域。以华东1（杭州）为例，Region填写为oss-cn-hangzhou。  region: "oss-cn-hangzhou",  // secure: true,  // 从STS服务获取的临时访问密钥（AccessKey ID和AccessKey Secret）。  accessKeyId: "LTAI5tFayF7LGkUjbqhPNDu8",  accessKeySecret: "rAHGulFmT45UXtiWFJ3QP1xiA33i4m",  // 从STS服务获取的安全令牌（SecurityToken）。  // stsToken: "yourSecurityToken",  // 填写Bucket名称，例如examplebucket。  bucket: "remote-s",  }) } |

而后通过JS代码对客户端实例化，进行分片上传和下载。

分片上传代码如下：

|  |
| --- |
| JavaScript async httpRequest(file) {  if (this.title == "" || this.title == null) {  this.$message({  message: "请输入标题",  type: "error",  });  return;  }  let client = Client();  let fname = `${new Date().getTime()}.zip`; //自定义文件名  // let fname = file.file.name;  let options = {  // 获取分片上传进度、断点和返回值。  progress: (p, cpt, res) => {  console.log(p);  },  // 设置并发上传的分片数量。  parallel: Math.ceil(  Math.floor((file.file.size / 1024 / 1024) \* 100) / 100  ),  // 设置分片大小。默认值为1 MB，最小值为100 KB。  partSize: 1024 \* 1024,  // headers,  };  try {  // object-name可以自定义为文件名（例如file.txt）或目录（例如abc/test/file.txt）的形式，  // 实现将文件上传至当前Bucket或Bucket下的指定目录。  const result = await client.multipartUpload(fname, file.file, {  ...options,  // progress,  });  let data = {  originName1: fname,  originName2: "",  size: Math.floor((file.file.size / 1024 / 1024) \* 100) / 100 + "M",  title: this.title,  type: "目标检测",  userId: Base64.decode(sessionStorage.getItem("userID")),  };  create(data)  .then((response) => {  console.log(response);  targetDetectionBatchAPI(response.data)  .then((res) => {  console.log(res);  this.filelist = [];  this.$message({  type: "success",  message: res.message,  });  })  .catch((err) => {  console.log(err);  });  })  .catch((error) => {  console.log(error);  this.$message({  type: "error",  message: "上传失败",  });  });  // console.log(data);  } catch (e) {  this.$message({  type: "error",  message: "上传失败",  });  console.log(e);  }  } |

分片下载代码如下：

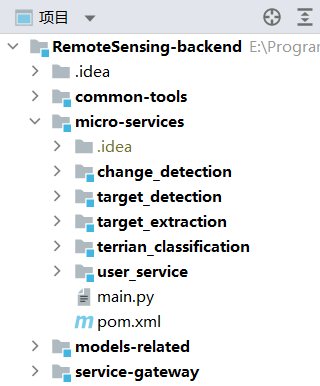
|  |
| --- |
| JavaScript download(row) {  if (row.status != "runEnd") {  this.$message({  type: "error",  message: "未运行结束",  });  return;  }  let client = Client();  let response = {  "content-disposition": `attachment; filename=${encodeURIComponent(  row.resultName  )}`,  };  // 填写Object完整路径。Object完整路径中不能包含Bucket名称。  let url = client.signatureUrl(row.resultName, { response });  // console.log(url);  fetch(url)  .then((res) => res.blob())  .then((blob) => {  // 将链接地址字符内容转变成blob地址  const a = document.createElement("a");  a.href = URL.createObjectURL(blob);  //测试链接console.log(a.href)  a.download = "fileName.xlsx"; // 下载文件的名字  document.body.appendChild(a);  a.click();  });  } |

**3 业务层技术说明**

**3.1 Spring Cloud框架及其相关技术**

**3.1.1 项目整体目录结构**

本项目后端整体目录结构如下图所示：



**common-tools模块**：用于存放相关配置类以及工具类（如swagger配置类、oss连接工具类等，以便每个微服务模块均可使用）的设计，如下图所示：

**micro-service模块：**包含五个微服务子模块，即：变化检测微服务、目标检测微服务、目标提取微服务、地物分类微服务、用户信息微服务。

**model-related模块：**用于存放mysql数据库的ORM映射模型。

**service-gatway模块：**用于存放有关网关的配置服务。

**3.1.2 nacos使用及其配置**

我们使用了nacos完成对微服务的集中配置与管理，并实现微服务的负载均衡，控制界面如下所示：



其中有关nacos的配置代码如下：

|  |
| --- |
| Properties *#nacos* spring.cloud.nacos.discovery.server-addr=112.124.12.171:8848 spring.cloud.nacos.config.server-addr=112.124.12.171:8848 spring.cloud.nacos.config.max-retry=10 spring.cloud.nacos.config.namespace=a20645fc-390a-44fb-b75b-dff21143620a spring.cloud.nacos.discovery.server-addr=112.124.12.171:8848 spring.cloud.nacos.discovery.namespace=a20645fc-390a-44fb-b75b-dff21143620a |

**3.1.3 gatway使用及其配置**

本系统中我们对微服务配置了网关，以实现端口统一、熔断过滤等功能。在熔断方面gatway将根据请求的超时、失败次数等情况对请求进行降级处理。有关的部分配置代码如下图所示：

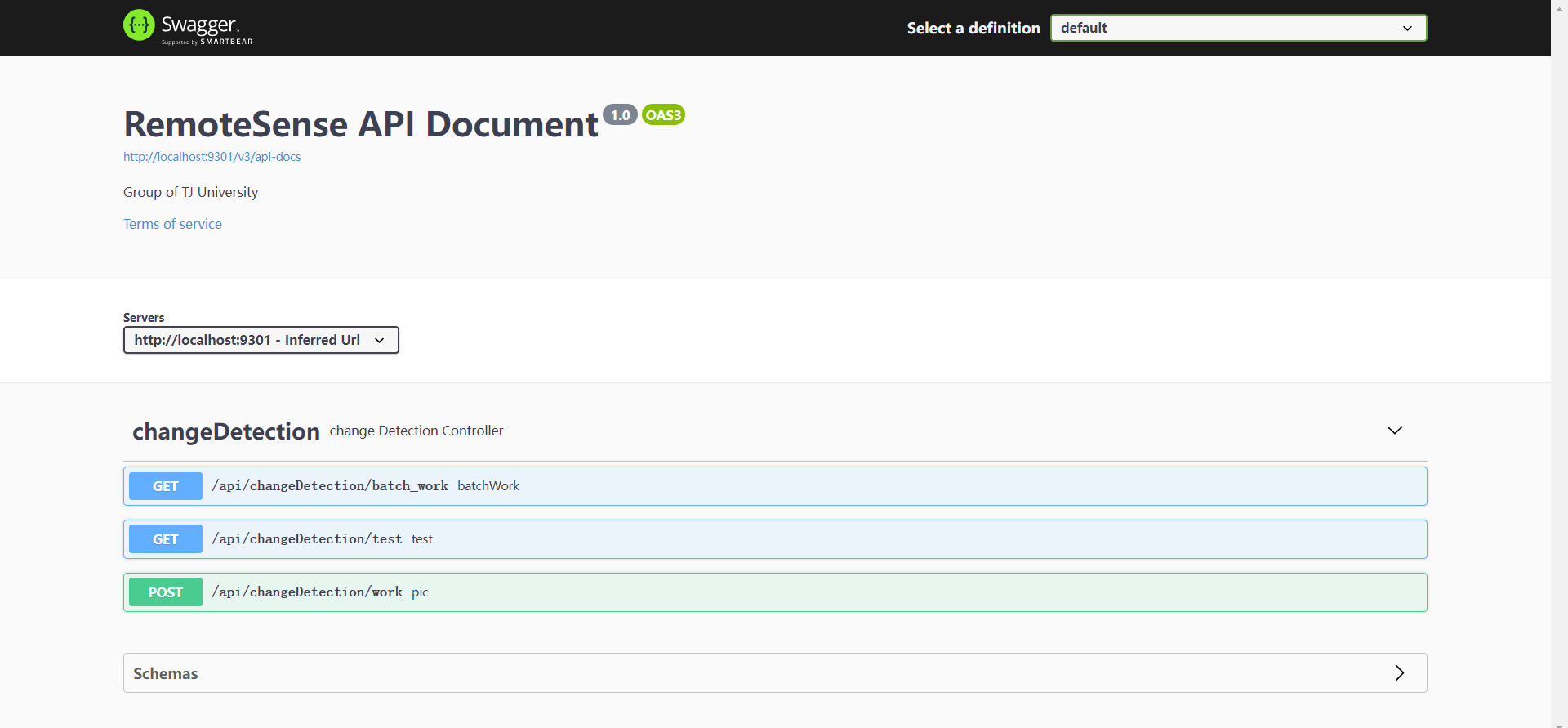
|  |
| --- |
| YAML server:  port: 9300 spring:  application:  name: service-gateway  redis:  host: 112.124.12.171  port: 9527  password: 339469604Lcy  database: 0  cloud:  gateway:  discovery:  locator:  enabled: true  lower-case-service-id: true  enabled: true  routes:  - id: change-detection  predicates: Path=/api/changeDetection/\*\*  uri: lb://change-detection  filters:  - name: Hystrix  args:  name: defaultfallback  fallback: forward:/defaultfallback  - name: RequestRateLimiter *#请求数限流，名字不能乱打* args:  redis-rate-limiter.replenishRate: 50 *#令牌填充速率* redis-rate-limiter.brustCapacity: 100 *#令牌捅容量* key-resolver: "#{@IPResolver}" *#SpEL表达式从Spring容器中获取Bean* |

其中在熔断方面还使用了内存数据库redis进行令牌填充。

**3.2 swagger接口文档及接口设计详细说明**

**3.2.1 swagger相关配置**

本项目使用了swagger3进行后端接口的统一化管理，以变化检测服务为例，效果如下所示：



相关swagger的配置代码如下：

|  |
| --- |
| Java @Configuration public class SwaggerConfig{  @Bean  public Docket createRestApi() {  *//返回文档摘要信息* return new Docket(DocumentationType.*OAS\_30*)  .apiInfo(apiInfo())  .select()  .apis(RequestHandlerSelectors.*basePackage*("com.remote"))  .paths(PathSelectors.*any*())  .build();  }   *//生成接口信息，包括标题、联系人等* private ApiInfo apiInfo() {  return new ApiInfoBuilder()  .title("RemoteSense API Document")  .description("Group of TJ University ")  .termsOfServiceUrl("https://www.tongji.edu.cn") *// 服务网址* .version("1.0")  .build();  } } |

**3.2.2单次上传接口代码设计说明**

单次处理后端先是接收前端图片传输的POST请求，然后将临时图片保存至本地文件夹，进而通过java的HttpClient工具调用模型层的flask接口完成单次推理，最终将结果图片以字节流的形式返回，并最后删除临时图片。相关核心代码如下（以目标检测为例）：

①保存图片至本地

|  |
| --- |
| Java *//保存至本地* String path="micro-services/target\_detection/src/main/resources"; String absolute=new File(path).getAbsolutePath(); File dest1 = new File(absolute+ "/input/" + a +".jpg");*//保存为jpg格式*  try {  picture.transferTo(dest1); *// 保存文件* } catch (Exception e) {  e.printStackTrace();  MyFile.*DeleteFolder*(absolute+ "/input/" + a +".jpg");  ls.remove(a);  return Result.*wrapErrorResult*("保存图片失败"); } |

②调用模型层的flask接口

|  |
| --- |
| Java String url="http://127.0.0.1:8300/TargetDetector"; Http http=new Http(); try {  *//传入文件名的参数* Map<String, Object> map=new HashMap<>();  map.put("a",a+".jpg");  map.put("r",r+".jpg");  map.put("dir",absolute+"/result/");  *//执行请求* String res=http.doGet(url,map);  System.*out*.println(res); } catch (Exception e) {  e.printStackTrace();  MyFile.*DeleteFolder*(absolute+ "/input/" + a +".jpg");  MyFile.*DeleteFolder*(absolute+ "/result/"+ r +".jpg");  ls.remove(a);ls.remove(r);  return Result.*wrapErrorResult*("目标url请求失败"); } |

③将结果图片以字节流的形式返回

|  |
| --- |
| Java *//读取python运行文件，并以字符流返还至前端* BufferedImage bufferedImage = null; try {  FileInputStream readPic=new FileInputStream(new File(absolute+"/result/"+ r +".jpg"));  bufferedImage = ImageIO.*read*(readPic);  readPic.close(); } catch (IOException e) {  e.printStackTrace();  return Result.*wrapErrorResult*("读取结果图片失败"); }finally {  *//删除图片* MyFile.*DeleteFolder*(absolute+ "/input/" + a +".jpg");  MyFile.*DeleteFolder*(absolute+ "/result/"+ r +".jpg");  ls.remove(a);ls.remove(r); } |

**3.2.3批量上传接口代码设计说明**

该接口是异步接口，分为主线程和异步线程。在主线程中程序根据传入的历史记录编号hisId获取待处理的上传记录，随后创建并启动异步线程，并在不阻塞的情况下返回“正在运行”的接口信息。在异步线程中则完成的是运行的主体逻辑：首先在mysql中修改该条历史状态为“运行中”，然后调用接口从oss中下载待处理zip文件，随后解压zip文件并遍历文件夹所有图片，对待处理图片依次使用flask模型层接口完成推理，最后将结果文件夹压缩并上传至oss并更新该记录的mysql数据库状态为“运行成功”（具体流程详见《概要说明文档》）。相关核心代码如下：

①获取并修改状态

|  |
| --- |
| Java *//获取目标记录* History history = historyService.getById(hisId); String fileName=history.getOriginName1(); *//修改状态* history.setStatus(HistoryStatus.*isRunning*.toString()); historyService.createOrUpdate(history); |

MySQL中对应结果即状态，以某次地物分类创建的结果为例，如下，创建成功后对应状态时uploadSuccess，说明上传成功，可以在历史记录中查询。



②创建异步线程运行脚本

|  |
| --- |
| JavaScript //异步线程，避免接口阻塞 Thread thread = new Thread(()->{  ...  //建立oss连接  OSSConnection oss = new OSSConnection();   //下载文件  if(!oss.downLoadMatipart("change-detection", fileName1)  ||!oss.downLoadMatipart("change-detection", fileName2)){  //更新数据库状态  history.setStatus(HistoryStatus.runError.toString());//失败的情况  historyService.createOrUpdate(history);  try {  throw new Exception("下载文件失败");  } catch (Exception e) {  e.printStackTrace();  }  }  //解压文件  //获取解压的目录名（须保证解压文件的内部目录和解压文件名一致）  ...  //遍历运行  for(int i=0;i<images1.length;i++){  ...  }  //压缩结果目录  ...  //更新数据库状态  history.setStatus(HistoryStatus.runEnd.toString());  history.setResultName(fName1+"\_result.zip");  historyService.createOrUpdate(history);   //删除tmp目录及文件  } //启动线程 thread.start(); |

运行结束前后MySQL的状态如下，在运行结束前，对应记录的状态时isRunning，运行成功后变为runEnd，如果运行失败，则会变为runError，这些都在枚举类中进行了封装。

|  |
| --- |
| Java public enum HistoryStatus {  uploadSuccess,//上传成功  isRunning,//正在运行中  runEnd,//运行成功  runError,//运行失败 } |





**3.3 数据库相关配置代码**

**3.3.1 Java连接oss服务及其配置**

通过参考OSS官方文档，我们自定义封装了一个适合本项目的OSS连接类，在类中封装了连接OSS服务器的方法以及分片上传、下载、删除文件等接口，供其他模块使用，引入的依赖和删除部分的代码如下：

|  |
| --- |
| XML <!--阿里OSS SDK--> <dependency>  <groupId>com.aliyun.oss</groupId>  <artifactId>aliyun-sdk-oss</artifactId>  <version>3.10.2</version> </dependency> |

|  |
| --- |
| Java public boolean removeFile(String objectName) {  // 创建OSSClient实例。  OSS ossClient = new OSSClientBuilder().build(endpoint, accessKeyId, accessKeySecret);  try {  if (objectName.isEmpty()) {  return false;  }  boolean found = ossClient.doesObjectExist(bucketName, objectName);  if (!found) {  return false;  }  // 删除文件或目录。如果要删除目录，目录必须为空。  ossClient.deleteObject(bucketName, objectName);  return true;  } catch (OSSException oe) {  return false;  } catch (ClientException ce) {  return false;  } finally {  if (ossClient != null) {  ossClient.shutdown();  }  } } |

**3.3.2 Jpa连接数据库及其配置**

本项目的数据持久层使用JPA，使用的Hibernate框架，首先导入数据库的配置，本项目中使用的是MySQL数据库，具体配置如下：

|  |
| --- |
| Properties #mysql spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver spring.datasource.url=jdbc:mysql://ip:port/remote?useUnicode=true&characterEncoding=utf-8&serverTimezone=UTC #name spring.datasource.username=xxxx #password spring.datasource.password=xxxxx |

然后导入JPA的依赖，如下：

|  |
| --- |
| XML <!--jpa--> <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId> </dependency> |

创建实体用于实现与数据库的映射，以用户表为例，与其形成映射的实体表如下：

|  |
| --- |
| Java @Entity @Table(name = "user") public class User {  @Id  @Column(name = "userId", nullable = false, length = 20)  private String id;   @Column(name = "name", nullable = false, length = 50)  private String name;   @Column(name = "password", nullable = false, length = 50)  private String password;   @Column(name = "email", nullable = false, length = 50)  private String email;   @Column(name = "salt", nullable = false, length = 50)  private String salt;   public String getSalt() {  return salt;  }   public void setSalt(String salt) {  this.salt = salt;  }   public String getEmail() {  return email;  }   public void setEmail(String email) {  this.email = email;  }   public String getPassword() {  return password;  }   public void setPassword(String password) {  this.password = password;  }   public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public String getId() {  return id;  }   public void setId(String id) {  this.id = id;  } } |

**4 模型层技术说明**

**4.1 flask模型层接口的设计**

将模型脚本的调用封装成接口，使得仅在系统开始运行时需要引用各库，而不需要每次执行python脚本时都对paddlers,paddlepaddle等库进行引用，提高了系统运行的效率，核心代码如下：

|  |
| --- |
| Python from flask import Flask, jsonify, request from flask\_httpauth import HTTPBasicAuth from flask\_cors import cross\_origin from change\_detection.src.main.resources.ChangeDetector import ChangeDetector from target\_detection.src.main.resources.TargetDetector import TargetDetector from target\_extraction.src.main.resources.TargetExtraction import TargetExtraction from terrian\_classification.src.main.resources.TerrianClassification import TerrianClassification  app = Flask(\_\_name\_\_) auth = HTTPBasicAuth()  @app.route('/ChangeDetector', methods=['GET']) @cross\_origin(supports\_credentials=True) def change\_detector():  args = request.args # 获取get参数  a = args.get("a")  b = args.get("b")  r = args.get("r")  dir = args.get("dir")  rate=ChangeDetector(a,b,r,dir)  res={"res":rate}  return jsonify(res)  @app.route('/TargetDetector', methods=['GET']) @cross\_origin(supports\_credentials=True) def target\_detector():  args = request.args # 获取get参数  a = args.get("a")  r = args.get("r")  dir = args.get("dir")  TargetDetector(a,r,dir)  res={"res":"success"}  return jsonify(res)  @app.route('/TargetExtraction', methods=['GET']) @cross\_origin(supports\_credentials=True) def target\_extraction():  args = request.args # 获取get参数  a = args.get("a")  r = args.get("r")  dir = args.get("dir")  TargetExtraction(a,r,dir)  res={"res":"success"}  return jsonify(res)  @app.route('/TerrianClassification', methods=['GET']) @cross\_origin(supports\_credentials=True) def terrian\_classification():  args = request.args # 获取get参数  a = args.get("a")  r = args.get("r")  dir = args.get("dir")  rate=TerrianClassification(a,r,dir)  res={"res":rate}  return jsonify(res)  if \_\_name\_\_ == '\_\_main\_\_':  app.run(debug=True, host='0.0.0.0',port=8300) |

**4.2 四种模型训练和推理的相关代码说明**

**4.2.1 模型训练**

**4.2.1.1 变化检测**

变化检测使用官方的BIT模型进行训练，为提高准确率采取了如下策略：

1. 数据增强：颜色抖动、随机裁剪、随机翻转；
2. 模型修改：采用hrnet18异构高级模型；
3. 超参数调整：修改起始学习率为0.0004、修改优化器为adamw；
4. 后处理调整：采取翻转预测的策略。

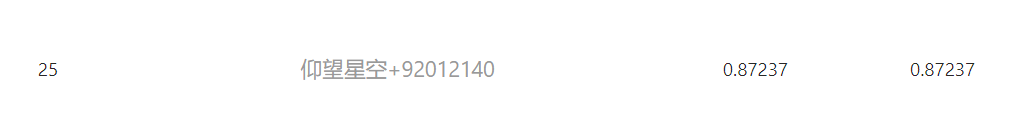
模型训练代码如下：

|  |
| --- |
| PowerShell !python /home/aistudio/PaddleCD/train.py \  --save\_dir /home/aistudio/data/output \  --config $COF\_PATH \  --do\_eval \  --use\_vdl \  --iters 3700 \  --save\_interval 370 \  --batch\_size=16 \  --log\_iters 50 \  --num\_workers 8 \  --seed 1919810 |

模型配置如下：

|  |
| --- |
| YAML batch\_size: 16 iters: 3700  model:  type: BIT  backb: hrnet18  in\_channels: 3  num\_classes: 2  loss:  types:  - type: CrossEntropyLoss  coef: [1]  optimizer:  type: AdamW  lr\_scheduler:  type: StepDecay  learning\_rate: 0.0004  step\_size: 1000  gamma: 0.5  train\_dataset:  type: RSCD  dataset\_root: /home/aistudio/data/src/  train\_path: /home/aistudio/data/src/train.txt  num\_classes: 2  transforms:  - type: RandomCrop  crop\_size: [384, 384]  aspect\_ratio: [1.0, 1.0]  scaling: [0.25, 1.0]  - type: RandomFlipOrRotation  probs: [0.6, 0.0]  probsf: [0.4, 0.4, 0.2, 0.0, 0.0]  probsr: [0.0,0.0,0.0]  - type: RandomDistort  - type: RandomBlur  - type: RandomSwap  - type: Normalize  mean: [0.5, 0.5, 0.5]  std: [0.5, 0.5, 0.5]  mode: train  val\_dataset:  type: RSCD  dataset\_root: /home/aistudio/data/src/  val\_path: /home/aistudio/data/src/val.txt  transforms:  - type: Normalize  mean: [0.5, 0.5, 0.5]  std: [0.5, 0.5, 0.5]  mode: val |

最终得分为0.87237，在复活赛榜单中排名第25名。



**4.2.1.2 目标检测**

目标检测中官方提供代码仅可以检测操场（playground）一类目标，修改代码扩展分类，使得模型可以识别数据集中的操场（playground），飞机（aircraft），oiltank（储油罐），overpass（立交桥）四种目标。

在模型方面依旧选用官方的PP-YOLO模型。

模型代码如下：

|  |
| --- |
| Python # 构建PP-YOLO模型 model = pdrs.tasks.PPYOLO(num\_classes=len(train\_dataset.labels)) model.net\_initialize(  pretrain\_weights='COCO',  save\_dir=osp.join(EXP\_DIR, 'pretrain'),  resume\_checkpoint=None,  is\_backbone\_weights=False ) |

训练代码如下：

|  |
| --- |
| Python # 执行模型训练 model.train(  num\_epochs=100,  train\_dataset=train\_dataset,  train\_batch\_size=8,  eval\_dataset=val\_dataset,  # 每多少个epoch存储一次检查点  save\_interval\_epochs=10,  # 每多少次迭代记录一次日志  log\_interval\_steps=10,  save\_dir=EXP\_DIR,  # 指定预训练权重  pretrain\_weights='COCO',  # 初始学习率大小  learning\_rate=0.0001,  # 学习率预热（learning rate warm-up）步数与初始值  warmup\_steps=0,  warmup\_start\_lr=0.0,  # 是否启用VisualDL日志功能  use\_vdl=True ) |

**4.2.1.3 地物分类和目标提取**

在官方提供算法基础上未作改变，使用DeepLab V3+模型。

地物分类模型代码如下：

|  |
| --- |
| Python # 构建DeepLab V3+模型，使用ResNet-50作为backbone model = pdrs.tasks.DeepLabV3P(  input\_channel=3,  num\_classes=len(train\_dataset.labels),  backbone='ResNet50\_vd' ) model.net\_initialize(  pretrain\_weights='CITYSCAPES',  save\_dir=osp.join(EXP\_DIR, 'pretrain'),  resume\_checkpoint=None,  is\_backbone\_weights=False ) |

地物分类训练代码如下：

|  |
| --- |
| Python # 执行模型训练 model.train(  num\_epochs=30,  train\_dataset=train\_dataset,  train\_batch\_size=16,  eval\_dataset=val\_dataset,  optimizer=optimizer,  save\_interval\_epochs=3,  # 每多少次迭代记录一次日志  log\_interval\_steps=100,  save\_dir=EXP\_DIR,  # 是否使用early stopping策略，当精度不再改善时提前终止训练  early\_stop=False,  # 是否启用VisualDL日志功能  use\_vdl=True,  # 指定从某个检查点继续训练  resume\_checkpoint=None ) |

目标提取模型代码如下：

|  |
| --- |
| Python # 构建DeepLab V3+模型，使用ResNet-50作为backbone model = pdrs.tasks.DeepLabV3P(  input\_channel=3,  num\_classes=len(train\_dataset.labels),  backbone='ResNet50\_vd' ) model.net\_initialize(  pretrain\_weights='CITYSCAPES',  save\_dir=osp.join(EXP\_DIR, 'pretrain'),  resume\_checkpoint=None,  is\_backbone\_weights=False ) |

目标提取训练代码如下：

|  |
| --- |
| Python # 执行模型训练 model.train(  num\_epochs=100,  train\_dataset=train\_dataset,  train\_batch\_size=8,  eval\_dataset=val\_dataset,  optimizer=optimizer,  save\_interval\_epochs=1,  # 每多少次迭代记录一次日志  log\_interval\_steps=30,  save\_dir=EXP\_DIR,  # 是否使用early stopping策略，当精度不再改善时提前终止训练  early\_stop=False,  # 是否启用VisualDL日志功能  use\_vdl=True,  # 指定从某个检查点继续训练  resume\_checkpoint=None ) |

**4.2.2 模型部署**

模型部署参考官方开源代码，先将模型训练得到的动态模型转为静态模型，再构建预测器进行推理。为提高推理精度，还对不同遥感功能进行了滑窗推理、翻转推理等不同尝试。最后对推理得到数据进行分析，以提供数据分析服务。

部署核心代码如下(以变化检测为例)：

|  |
| --- |
| Python  # 读入影像 im\_a = cv2.imread(A\_PATH) im\_b = cv2.imread(B\_PATH) ori\_size = im\_a.shape[:2]  # 把滑窗中的内容全部取出，存在一个列表中 patch\_pairs = [] for rows, cols in WindowGenerator(\*ori\_size, WINDOW\_SIZE, WINDOW\_SIZE, STRIDE, STRIDE):  patch\_pairs.append((im\_a[rows, cols], im\_b[rows, cols])) # 导出输入尺寸为滑窗大小的模型，--fixed\_input\_shape中的batch\_size等于len(patch\_pairs) run(  f"python ./PaddleRS/deploy/export/export\_model.py\  --model\_dir=./dynamic\_models/best\_model \  --save\_dir=./static\_models/{WINDOW\_SIZE}x{WINDOW\_SIZE} \  --fixed\_input\_shape=[{len(patch\_pairs)},3,{WINDOW\_SIZE},{WINDOW\_SIZE}]",  shell=True,  check=True ) # 构建预测器并执行推理 modelsPath=absolute+'/static\_models' predictor = Predictor(f"{modelsPath}/{WINDOW\_SIZE}x{WINDOW\_SIZE}", use\_gpu=True) res = predictor.predict(patch\_pairs) |

**4.2.2.1 变化检测**

为提高推理精度，选用翻转推理策略。对于推理结果，计算变化区域和未变化区域像素点个数比例，以计算变化率，计算变化率代码如下：

|  |
| --- |
| Python def get\_rate(image,n):  w,h = image.shape  size = w\*h  count = 0  for row in image:  for p in row:  if p == n:  count+=1  return count/size |

**4.2.2.2 地物分类**

计算大地区域和其他物体区域像素点个数比例，以计算地物比率，计算地物比率代码与计算变化率代码相同。

**4.2.2.3 目标检测和目标提取**

目前仅进行部署，未提供数据分析服务。