

XDNAP 算法上传接口设置



西安电子科技大学
XIDIAN UNIVERSITY

文档状态

文件状态	项目名称	文档名	作者	时间	页数	密级
<input type="checkbox"/> 草稿 <input checked="" type="checkbox"/> 修改迭代 <input type="checkbox"/> 正式发布	6G 全场景按需服务 关键技术	算法上传接口设置	毛伟灏	2024.4	-	公开

版本控制

修改日期	修改类型*	作者	参与者	版本	联系方式
2023.6	A	毛伟灏	-	1.2	maowh@stu.xidian.edu.cn
2024.3	A	毛伟灏	-	1.3	maowh@stu.xidian.edu.cn
2024.5	A	吕嘉豪	毛伟灏、杜福林、申蕊鸣	2.0	s2352998514@163.com

* 修改类型为 A-Added M-Modified D- Deleted

维护说明

本文档维护说明旨在确保操作手册保持最新，内容准确无误，以指导用户正确、高效地使用系统。文档的更新和维护是为了反映系统的变化、用户反馈、技术进步以及政策调整。

1. 文档将定期进行审查，以确保所有的操作步骤、程序和策略与当前系统功能相匹配。审查频率为每季度一次，或者在系统更新后立即进行。
2. 用户和技术支持团队的反馈是文档维护的关键组成部分。所有的反馈都应当记录并定期审查，以确定是否需要文档进行更新。
3. 每次文档更新后，都应当创建一个新的版本，并保留旧版本的档案。新版本的文档应明确标注更新日期和版本号。

该文档仅供内部人员学习使用

目录

XDNAP 算法上传接口设置.....	1
维护说明.....	2
1 引言.....	5
2 本地算法上传标准.....	7
2.1 算法拆分标准.....	7
2.2 代码书写标准.....	8
3 网络算法标准.....	9
3.1 算法虚拟化封装标准.....	9
3.2 网络算法启动标准.....	10
3.3 网络算法接口标准.....	11
4 基于 C++开发的通信仿真平台	18
4.1 平台功能介绍.....	18
4.2 前端场景布设及使用.....	22
4.2.1 前端平台操作如下	22
4.2.2 数据库存储验证	22
4.2.3 通信平台操作	23
4.2.4 注意事项	23
5 算法环境部署标准.....	24
5.1 Python 环境部署标准.....	24
5.1.1 requirements.txt 标准	24
5.1.2 Python 版本标准.....	25
6 基于微服务技术的算法标准（Python）	26
6.1 Python 算法标准.....	26
6.1.1 Json 文件编写规范	26
6.1.2 主函数算法标准	28
6.2 基于微服务架构的算法样例.....	29
6.2.1 算法路径标准	30
6.2.2 读取 Json 文件	31
6.2.3 Send.py 脚本的使用	31
6.2.4 requirement.txt	33
6.2.5 声明项目名称	33
6.3 数据库展示.....	33

6.4	利用 Docker 实现微服务架构.....	36
6.5	移植镜像步骤.....	38
6.6	移植镜像挂载.....	38
6.7	平台效果展示.....	38
7	基于微服务技术的算法标准 (MATLAB)	39
7.1	Matlab 算法标准.....	39
7.1.1	Json 文件编写规范.....	40
7.1.2	主函数算法标准	40
7.2	基于微服务架构的算法样例.....	41
7.2.1	算法路径标准	41
7.2.2	主函数读取 Json 文件.....	42
7.2.3	Send.py 脚本的使用	42
7.2.4	主函数连接数据库	43
7.3	数据库展示.....	43
7.4	利用 Docker 实现微服务架构.....	46

1 引言

XDNAP 平台采用云原生设计理念,通过微服务架构将网络功能进行了拆分,使其与传统硬件设备相分离。平台整体架构包括服务器端的数据湖、智能算法库等模块,并将其他服务组件集成在微服务总线上,实现了服务设计模块之间的相互调用。同时,该平台还建立了内部模拟器和外部半实物模拟器,以虚实结合的方式实现了场景的全域模拟。在这一技术背景下,本文档旨在提供一套完善的接口设置,以便于对算法进行上传并有效部署。

首先对算法的各个组成部分进行拆分。将算法按照分布式平台的需求拆分为:场景构建、算法训练、数据存储等多个部分。算法将上线至 XDNAP 平台并进行校验, DRL 等算法(以下利用 DRL 算法举例)的核心部分在智能算法库中进行拆分存储。

其次,使用平台对算法的场景构建进行初步设定,在服务器前端输入算法需要的参数,并通过拖拽的方式,生成网络场景。场景数据实时生成,前端将场景信息,以及所输入的参数信息转发至后台,在数据湖模块进行存储,并等待数据总线模块进行转发调用。场景信息数据十分庞大,通过键值对形式存储于数据湖中,在算法调用之前,需先通过配置绑定模块进行相关场景数据配置绑定,并以 Json (或者 Yml,以下利用 Json 算法举例)的形式返回,为算法收集提取所需要的场景参数及状态数据。

接着,为了解决用户关联问题,平台调用智能算法库的 DRL 算法,并将实时生成的用户和基站状态信息作为相应参数传输给 DRL 算法,算法进行实时的策略输出。同时,状态信息以及生成的基站动作选择数据,将作为神经网络训练数据集存储于数据湖模块。

随后,场景生成模块从数据总线模块获取算法输出的基站动作选择策略,并实时的指导用户的接入策略。在策略执行的过程中,如果达到了记忆库所设计的上限,算法模块将同时从数据湖模块取出记忆库,并作为训练数据集对智能算法库中的深度神经网络进行训练。训练完成的神经网络将在下次策略生成中参与算法的基站动作选择流程,以加速算法的收敛。

最后,场景数据更新完毕,继续将基站用户状态及信道状态等信息传入后端进行算法迭代,并重复此过程。通过场景饱和和算法迭代进行判断后结束算法。智能体为动态连续的场景输出实时的用户关联策略,通过不断的训练、验证和优化,不断提高深度学习模型的预测性能和可靠性。

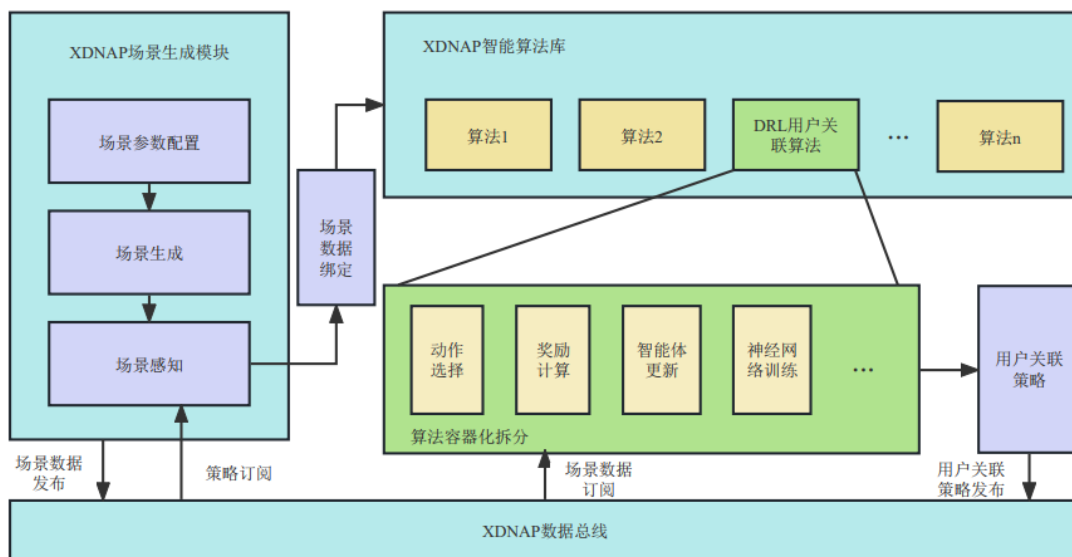


图 1.1 仿真数据交互流程图

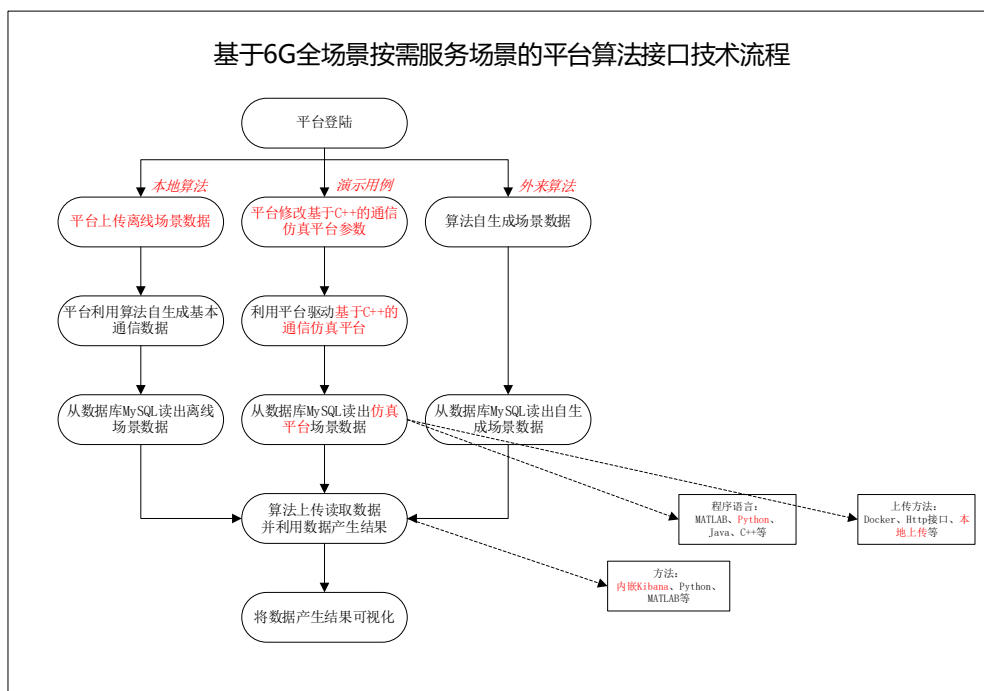


图 1.2 仿真平台优化方案

2 本地算法上传标准

在仿真流程中，算法上线主要包括代码打包、校验、上传三个步骤。如图 2.1 所示，代码包必须符合本平台所提出的代码规范。首先，网络参数设定需要合理，代码所需要的网络参数必须包含在平台场景库参数内。其次，算法函数命名要符合一定的标准，一些公用的机器学习或工具库的函数名称要进行统一，如果没有按照标准创建函数名，需要在平台前端界面进行函数关联定义。在代码包通过校验后，即可上传至平台供各个模块调度使用。打包上线过程参考管道的概念，使用户可以实时交付上传，不断更新迭代。

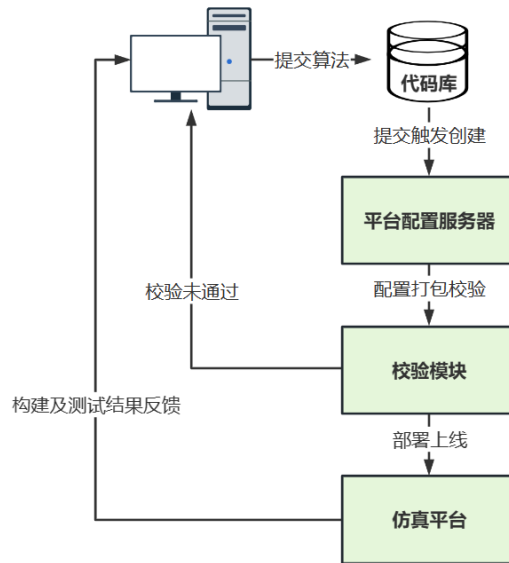


图 2.1 本地算法上传流程

2.1 算法拆分标准

由于展示模块的前后端的请求交互基于 http 协议完成，而算法智能体的训练虽然耗时较短但仍然需要一段时间，可能会导致 http 同步请求不到而断开连接。故本小节着重讲述了算法异步响应的设计，以解决该问题。

多数机器学习算法都是异步算法，在算法上线时需要对算法进行模块化拆分上线，所以制定了相应标准。如表 2.1 所示，本文针对算法结果信息的异步返回，对算法进行了函数化拆分，要求异步算法必须提供四部分文件，如表 2.1 所示，包括 Start、Stop、State、Config。对异步算法拆分进行解释，START 函数将使能用户关联算法，调用算法内部子函数，是算法的入口。START 方法启动算法后将读取 Json 文件中场景参数信息，并将不断向 CONFIG 文件中写入状态码信息。规定状态码 0 表示算法执行失败，1 表示算法执行中，2 表示算法执行成功并产生结果。STATE 用于前端实时获取 CONFIG 文件中的状态码信息，以确定算法的执行状态。若由于不合理的参数设置使得算法运行抛出异常或系统故障则执行 STOP 函数，强行终止算

法。

表 2.1 函数名称对应功能

函数名称	函数功能
START	启动算法
STOP	终止算法
STATE	实时从 CONFIG 文件中读取算法执行状态
CONFIG	包含算法的参数、STATE 状态码等信息

2.2 代码书写标准

- 1. 目前使用 java 进行前后端平台的开发，平台前端基于 Java-Spring 进行搭建，通过 SpringBoot 轻量化地集成各个模块，主要使用模块包括：Spring MVC、Spring MyBatis、Spring Security 等。其中，Spring MVC 用于解决前后端请求处理和交互的问题，Spring MyBatis 用于方便的访问数据库，并拉取相关数据在平台前端界面进行展示。而 Spring Security 用于管理平台的权限，处理例如异常报错等安全性问题。
- 2. 平台是 java 开发，所以上传 python 需要使用 java 和 python 专门的连接包。
main 函数格式需要写为方法，要存放在根目录下。
- 3. 整个算法代码执行过程不能中断，所以删除所有打印和生成图片的语句，后期代码书写最好使用标志位，作为参数手动调节区分开发模式和上线模式。开发模式下可以随意打印输出。
- 4. 参数不再受 5 个的限制，后期在场景构建模块通过 Json 传输键值对，匹配参数，传入算法。目前是只能是 5 个参数。
- 5. 参数格式限制为字符串，需要手动转化。比如你要 int 类型的，但是传输过来的是 String，需要在算法中自己转化。

3 网络算法标准

在面对算法安全性和保密性等敏感问题时,采用网络算法标准是至关重要的。网络算法标准提供了一系列针对算法上传、部署和运行的安全措施,以确保算法的安全性和保密性。这些标准涵盖了算法虚拟化封装、网络算法启动和接口标准等方面,为算法的安全性提供了全面的保障。

用户需要将自己的算法代码上传至云端仓库(如 Git、GitHub 等),XDNAP 的算法上线系统将从云端自动拉取代码。其次,平台进入代码单元测试阶段,系统会自动配置 SonarQube,找出静态代码可能存在的风险,对包括 C、C++、java、python、SQL 等二十几种编程语言的代码质量做出管理与检测。然后,对于检测通过的代码,平台将启动镜像打包程序,并将镜像推送到 Docker Hub。最后,平台会对当前网络状态进行感知和预测,然后自动拉取镜像,将镜像制成算法成品,剥离数据集,将算法部署到智能算法库模块,并在系统审核后正式部署到开发环境。算法共包含“senddata”、“datagen”、“mytrain”、“testfp”、“testhetgmn”五个部分,涉及中间件 kafka、zookeeper,如图 3.1 所示。

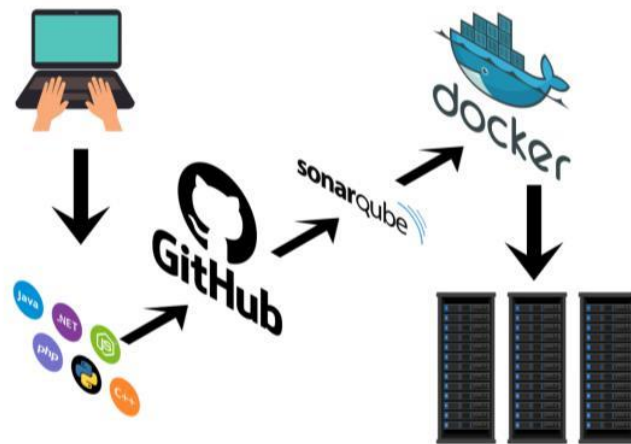


图 3.1 网络算法流程

3.1 算法虚拟化封装标准

算法的虚拟化基于 Docker,其底层原理是利用 Linux 系统的多种防护机制实现严格的进程隔离。也就是说,通过 Docker 可以将计算机上的物理资源分成若干个独立的区域,每个区域互不干涉,而且不会抢占资源。镜像就是 Docker 部署的基本单位,一个镜像打包了指定应用程序所需要的所有物理资源及运行环境。对内,镜像指定了划分计算机资源的范围;对外,镜像是一个独立的文件。镜像运行的实例被称为容器,容器是应用程序层的抽象。

算法的虚拟化过程可以由 XDNAP 算法上线系统自动完成,但为了展示该技术的实现原理及过程,本小节通过本地手动配置完成算法的虚拟化封

装。

如图 3.2 所示，通过 “`docker build -t <name> .`” 命令将程序打包成镜像，打包过程包括但不限于如下几步。

1. FROM: 指定基础镜像。例如算法基于 `python` 语言实现，因此在打包过程中需要指明 `python` 以及相应版本。
2. MAINTAINER: 镜像制作者的个人信息。
3. RUN: 构建容器时需要执行的命令。构建容器时经常涉及到一些脚本或命令的执行，例如使用 `pip` 指令下载 `requirements.txt` 中指定的第三方库。
4. EXPOSE: 指定端口号。容器启动后可能需要对外暴露端口以实现程序和外部的交互。
5. WORKDIR: 创建容器后终端默认的工作路径。
6. ENV: 构建镜像时需要的环境变量。
7. ADD: 将宿主机目录下的文件拷贝至镜像，也可以指定 URL，将云端文件拷贝至镜像。
8. COPY: 将宿主机本地的指定文件或目录拷贝进镜像。
9. VOLUME: 容器数据卷。容器启动后可能有一些需要持久化或者与宿主机共享的文件，可将其保存至数据卷。
10. CMD: 构建镜像时需要执行的命令。

```
[+] Building 53.5s (10/10) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 32B                                              0.0s
=> [internal] load .dockerignore                                                0.0s
=> => transferring context: 2B                                                  0.0s
=> [internal] load metadata for docker.io/bitnami/pytorch:1.13.1-debian-11-r23 0.0s
=> CACHED [stage-1 1/5] FROM docker.io/bitnami/pytorch:1.13.1-debian-11-r23    0.0s
=> [internal] load build context                                              0.0s
=> => transferring context: 283B                                              0.0s
=> [stage-1 2/5] ADD . /code                                                  0.1s
=> [stage-1 3/5] WORKDIR ./code                                              0.1s
=> [stage-1 4/5] COPY requirements.txt requirements.txt                      0.1s
=> [stage-1 5/5] RUN pip3 install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/ 52.3s
=> exporting to image                                                        1.0s
=> => exporting layers                                                        1.0s
=> => writing image sha256:894951afae6938187be42a3917a5c40bc8a26fbce7a2086341d08c4c35802c52 0.0s
=> => naming to docker.io/library/mytrain                                    0.0s

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

图 3.2 算法成功虚拟化封装

3.2 网络算法启动标准

首先将所有服务用 `docker` 启动，服务会持续监听端口，五个组件启动命定如下。

```
docker run -d --name senddata --publish 9996:9996 --network mynet
senddata
```

```
docker run -d --name datagen --publish 8887:8887 --network mynet -v
C:\Users\admin\Desktop\dockerdata\datagen:/app/datagen/data datagen
```

```
docker run -d --name mytrain --publish 6664:6664 --network mynet --
privileged=true -u=root -v
C:\Users\admin\Desktop\dockerdata\datagen:/app/mytrain/data -v
```

```
C:\Users\admin\Desktop\dockerdata\model:/app/mytrain/model mytrain
```

```
docker run -d --name testfp --publish 5552:5552 --network mynet --privileged=true -u=root -v
```

```
C:\Users\admin\Desktop\dockerdata\testdata:/testFP/test testfp
```

```
docker run -d --name testhetgnn --publish 3331:3331 --network mynet --privileged=true -u=root -v
```

```
C:\Users\admin\Desktop\dockerdata\testdata:/app/testHetGNN/test -v
```

```
C:\Users\admin\Desktop\dockerdata\model:/app/testHetGNN/model testhetgnn
```

需要注意的是，数据集需要利用宿主机做存储，涉及宿主机上的数据需要在启动容器时将数据卷挂载到对应路径，即上述指令中高亮部分需要根据宿主机路径自行更改，其他指令不得改动。

3.3 网络算法接口标准

上一小节已经通过 Docker 技术将算法封装到 XDNAP 平台，由于 XDNAP 组件间的通信是基于 RESTful 风格的接口，接下来本小节通过第三方接口测试工具 Postman 向算法发送超文本传输协议(Hyper Text Transfer Protocol, HTTP)请求并测试分布式算法各部分之间的接口响应。需要说明的是，算法通过 XDNAP 上线后会通过控制总线将服务自动注册至平台，彼时各服务占用端口为平台统一调度。但此处为方便测试，将端口固定为表 3.1。XDNAP 场景生成模块会通过前端输入将场景所需参数发布给数据总线，如图 3.3 所示，将该端口指定为 9999，通过 Postman 将场景参数以 json 文件发送至数据总线，前端获取发送成功的响应。如图 3.4 所示，进入数据总线对应的 Docker 容器内部，输出消费数据。该接口测试说明，前端输入的场景参数可以顺利进入数据总线，等待算法订阅。

表 3.1 函数名称对应功能

微服务功能	端口号
场景参数发布	9999
场景数据订阅	8888
算法 训练使能	6666
算法 测试结果发布	3333
其他算法使能	5555

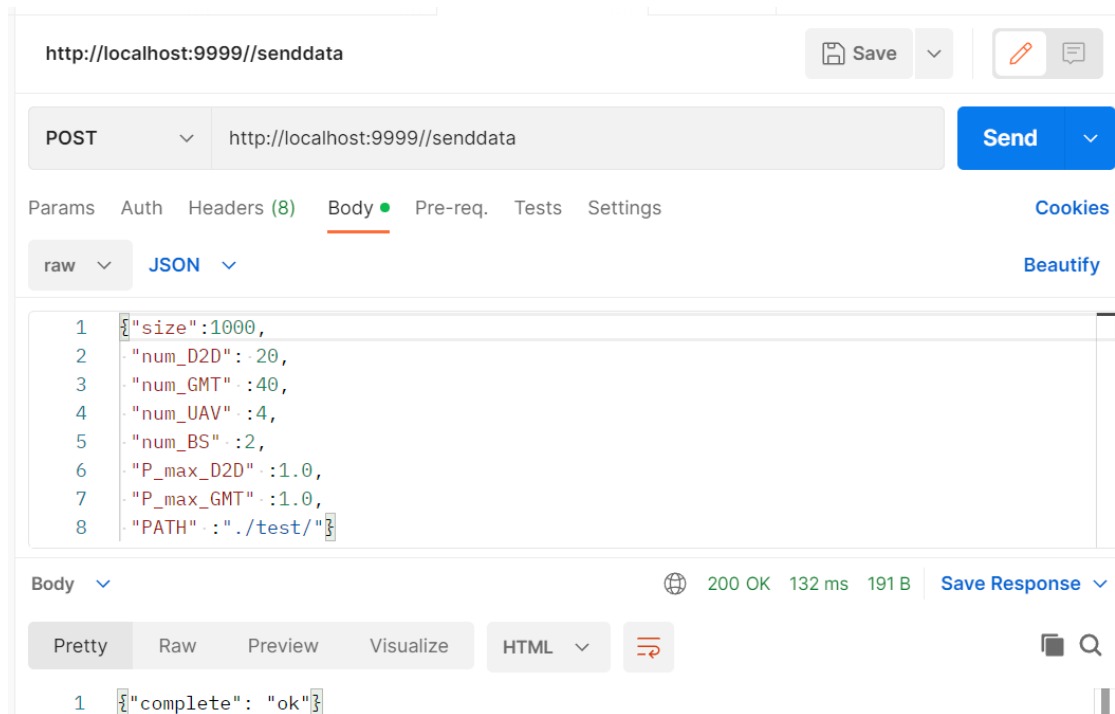


图 3.3 使用 Postman 发布场景参数

```
root@3d9a7350707e:/opt/kafka/bin# kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic HetGNN
{"size": 1000, "num_D2D": 20, "num_GMT": 40, "num_UAV": 4, "num_BS": 2, "P_max_
D2D": 1.0, "P_max_GMT": 1.0, "PATH": "./test/"}
```

图 3.4 场景参数发布后 kafka 容器内部响应

场景参数发布至数据总线后，只需要访问数据总线并指定主题便可以订阅使用。场景参数通过计算可以生成指定数据集，数据集的生成为 XDNAP 场景生成工作的一部分，当数据集生成后，由算法加载并使用。如图 3.5 所示，通过 8888 端口，算法可以加载包括该数据集并得到反馈。

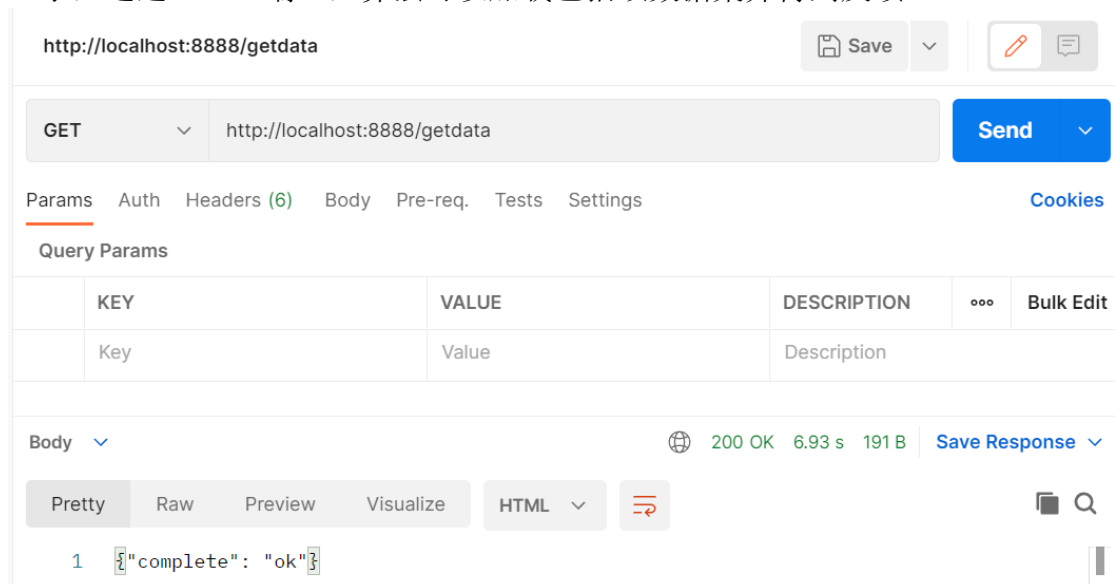


图 3.5 使用 Postman 为算法加载数据集

进入算法数据加载模块的 Docker 容器，可以看到算法正在加载数据集，

如图 3.6 所示。

```
production deployment. Use a production WSGI server
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8888
* Running on http://10.168.1.195:8888
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 100-565-333
66%|██████████      | 665/1000 [00:11<00:05, 58.40it/s]
```

图 3.6 加载数据集算法容器内部响应

向“getdata”的 8887 端口发送 get 请求如下。

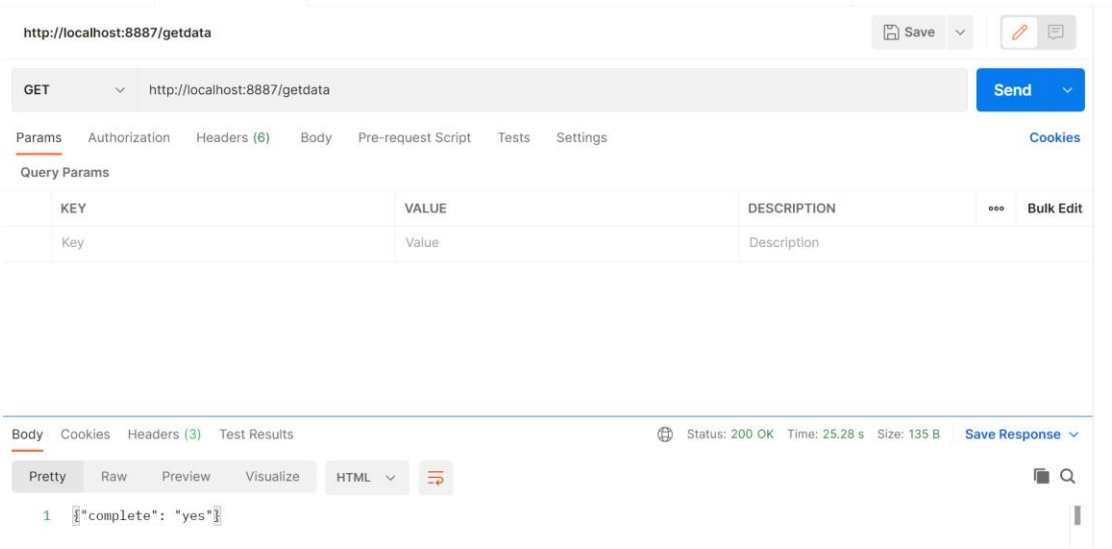


图 3.7 调试图-1

这部分模拟当程序启动时需要持续向 kafka 索要环境配置参数，该参数应由前端输入。

向“senddata”的 9996 端口发送 post 请求如下。

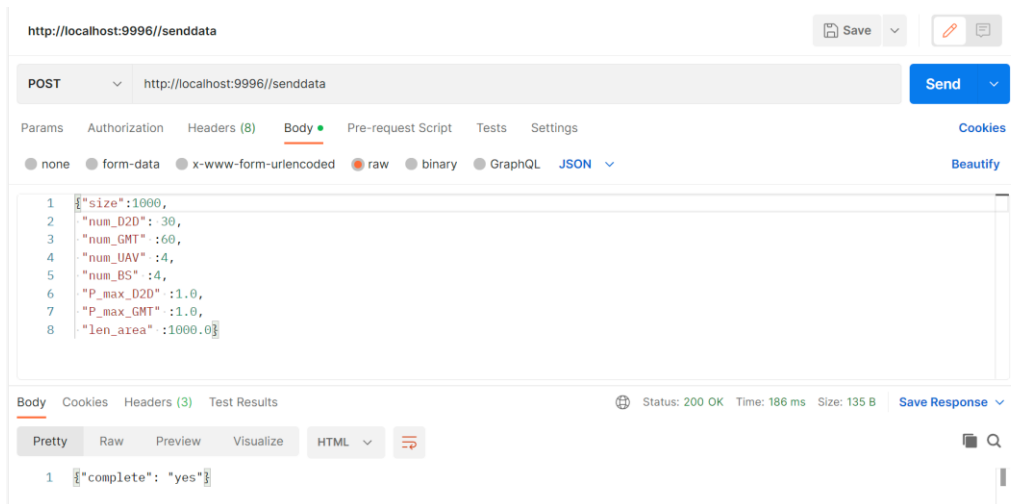


图 3.8 调试图-2

这部分模拟前端输入了一系列参数，包括需要产生的数据集文件大小、D2D 用户数、蜂窝用户数、无人机数、地面基站数、D2D 用户的最大功率限制、蜂窝用户最大功率限制、场景范围（方形区域边长）等，这里的 D2D 和蜂窝用户以及无人机平面坐标在方形区域内随机，地面基站位置和无人机飞行高度在代码中固定。

这里点击 **send** 按钮，这个参数表会写入 kafka，并被上述“getdata”的 8887 端口 get 到。

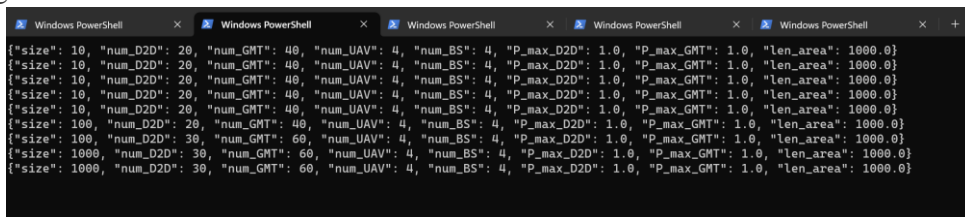


图 3.9 调试图-3

然后“getdata”开始生产数据，生成的数据集将保存到容器内的如下路径。

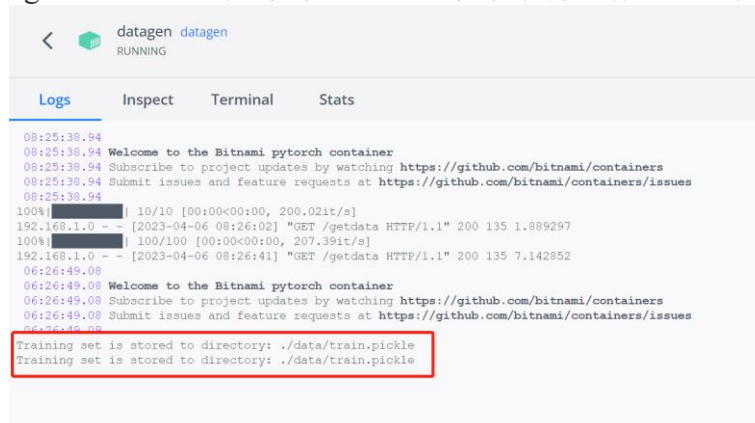


图 3.10 调试图-4

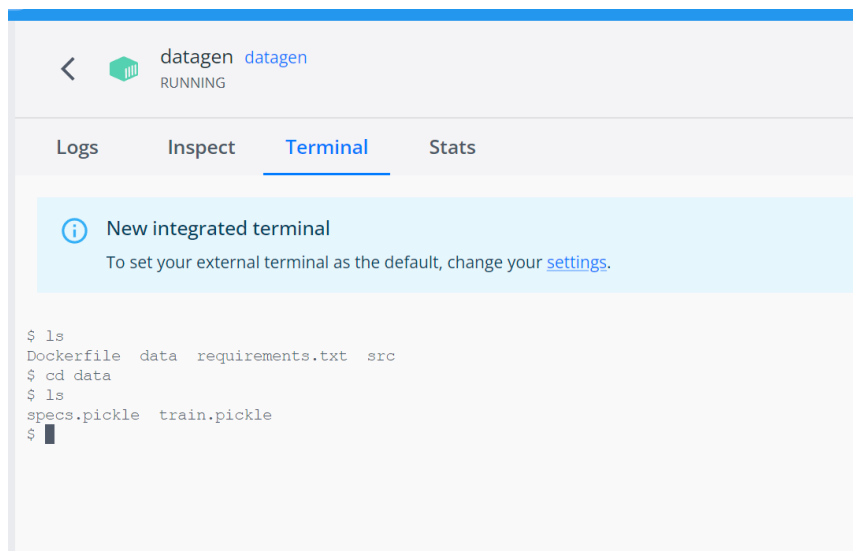


图 3.11 调试图-5

由于该路径已经挂载到宿主机，因此宿主主机上可以看到如图所示。

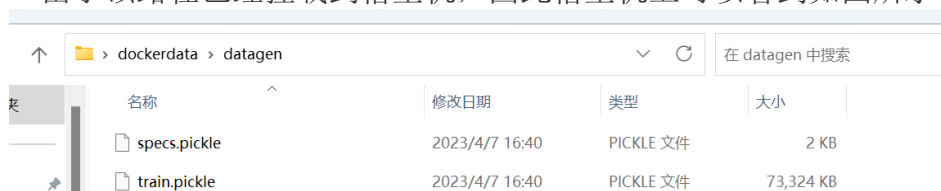


图 3.12 调试图-6

向“mytrain”的 6664 端口发送 post 请求如下。

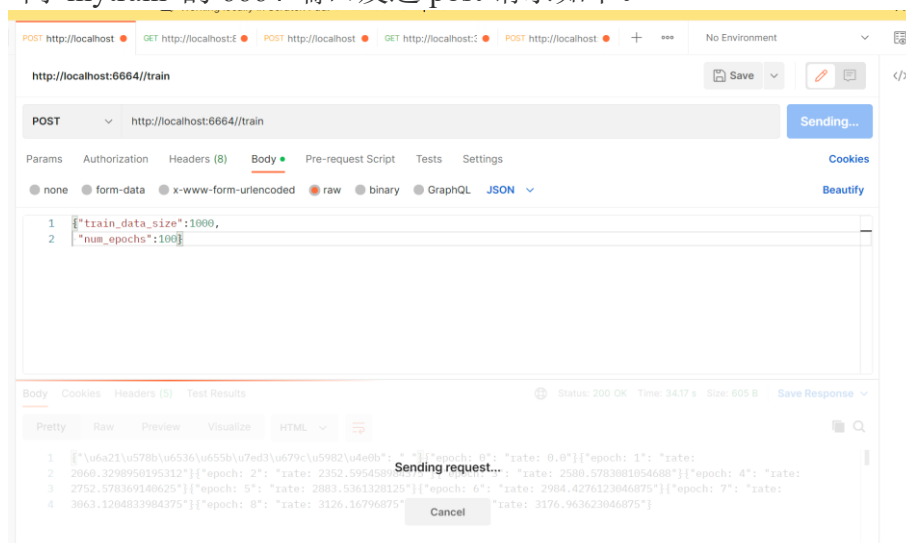


图 3.13 调试图-7

模型开始训练，所需时间较长，需要指定训练需要的数据集大小以及训练轮次。最后反馈给前端一个训练结果，训练结果暂时没有写入 kafka。模型训练所需的数据来自上述“getdata”的结果，“getdata”先写到宿主机，“mytrain”会将目标数据集读取路径挂载到宿主机相同路径，由于文件较大宿主机做了个转存，这也是之前讨论的数据集较大要不要用数据库存储。我这个目前只有 70 多 M。

同样的，模型文件也有对应的保存路径，也同样挂载到宿主机上。

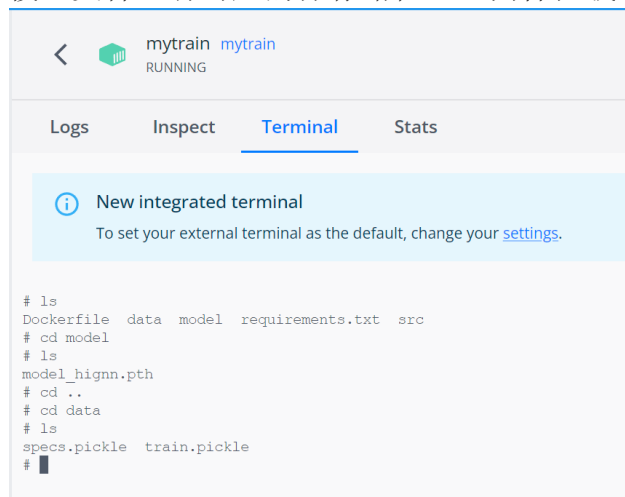


图 3.14 调试图-8



图 3.15 调试图-9

最后是模型测试（推理）阶段。

向“testfp”的 5552 端口发送 post 请求如下。

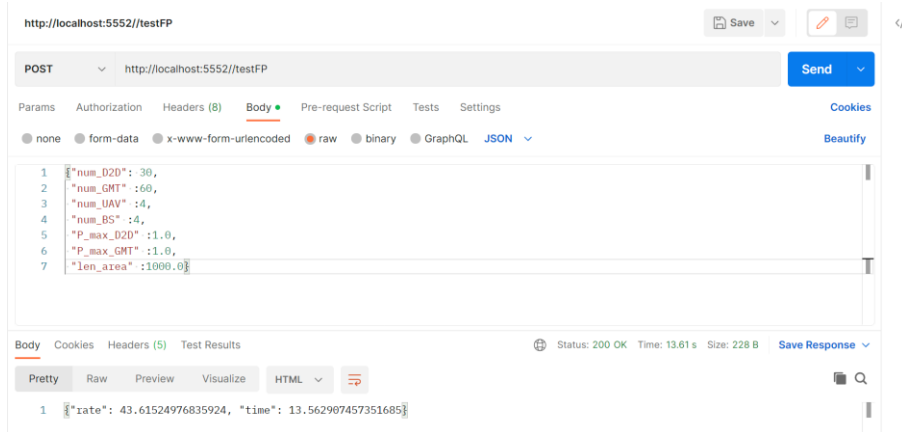


图 3.16 调试图-10

需要指定实时的场景数据，训练时的场景可能有变化，实时场景在使用 FP（对比算法）时会有一个计算结果，该计算结果以及计算时长会反馈回前端。

向“testthetgmn”的 3331 端口发送 get 请求如下。

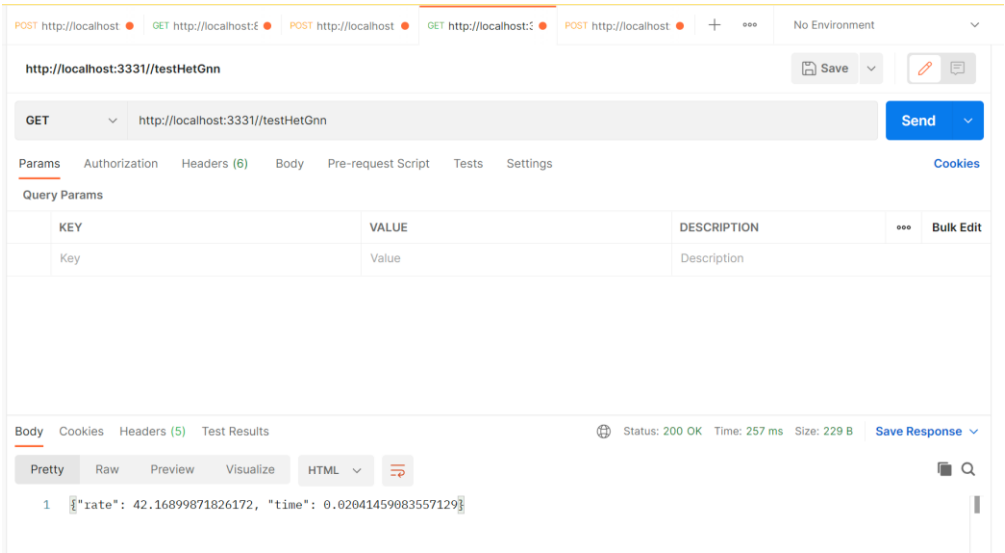


图 3.17 调试图-11

新的场景数据在测试 FP 时已经生成了，因此这里无需再定义实时场景参数。同样的场景使用 **hetgnn** 还会有一个计算结果以及计算时长，最后反馈给前端。

4 基于 C++开发的通信仿真平台

仿真平台利用对空中、太空以及地面等全域网络资源的高效编排与调度,实现了对“空中飞车”的全面资源接入。面对不同域间的交互挑战,平台采用了创新的跨域交互机制。这一机制的核心是数据调度模块,它能够灵活处理来自不同域的数据流,确保数据的有效传输和接收。通过我们精心设计的数据包调度系统,空中飞车得以实现全方位的网络连接,从而提供无缝、连续的服务。这不仅展示了技术的前沿进步,也为空中飞车的实用性和可靠性提供了坚实的保障。

在应对跨域资源访问的复杂性时,我们特别面对了模拟跨域物理层信道的重大挑战。为此,我们开创性地设计了一个解决方案:在同一项目框架内,通过高效的消息中间件实现不同域之间的连接,并与一个大规模模拟环境(Large-scale Engineering Simulation Environment, LSESE)进行互动。在 LSESE 中,我们精心构建了物理层与数据链路层的交互模式,而高于数据链路层的交互则在各自的专属域中完成。

这种创新的设计策略不仅极大地提升了跨域通信的效率和准确性,还促进了系统内各层面之间的紧密协作。借助这一机制,我们的“空中飞车”系统能够在确保高带宽和低延迟的同时,实现全球范围内的稳定通信覆盖。更重要的是,这种设计为紧急情况的应对提供了坚实的技术支持,特别是在灾难响应和临时事件覆盖方面。

综合而言,这种全场景连接架构不仅显著增强了通信技术的可靠性和适应性,也为智慧城市和未来物联网的发展打下了坚实的基础。通过这种多域整合和高效交互,我们的系统不只是优化了网络的整体性能,还为广泛的应用场景提供了强有力的支持。这一跨域通信机制尤其适用于复杂环境和高标准应用场景,例如智慧城市管理、远程医疗服务和灾难响应等。总体来说,“空中飞车”的全场景连接系统通过其创新设计和执行原则,已成为未来通信技术和物联网应用的一个强大而灵活的平台。

4.1 平台功能介绍

为了全面揭示“空中飞车”系统的卓越性能和创新技术,让我们详细阐述这一全场景连接技术的具体实现过程。该过程体现了先进的 6G 通信技术与多域集成的高效融合,具体实现过程如下:

第一步:“空中飞车”节点设计。

“空中飞车”节点有完整的 OSI 七层模型,在仿真运行,数据包传递到网络层后,会对该包的目的节点进行判断,如果是发送到当前域的节点,就直接向下传递,但发送到其他域节点时,到网络层会把数据传递到 LSESE 中的镜像节点的网络层,再向下模拟数据链路层和物理层。

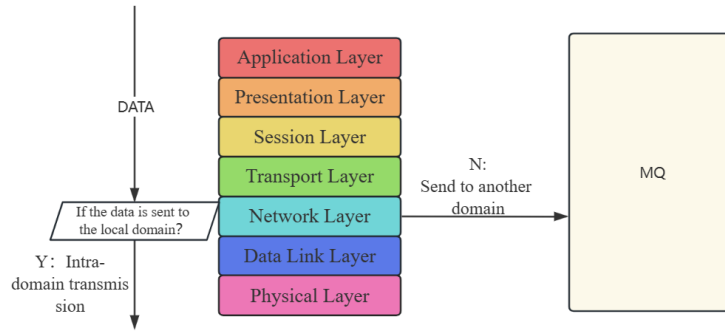


图 4.1 用户节点设计图

第二步：设计访问跨域资源仿真流程。

- a) 用户产生数据到达网络层
- b) 数据包转发到消息组件
- c) 经由消息组件到达大工程模拟环境的 GatewayStation
- d) 转发到镜像用户的网路层
- e) 经过镜像用户网络层向下经过数据链路层和物理层
- f) 在大工程模拟环境仿真用户与无人机之间的物理层信道
- g) 到达镜像接入节点的网络层
- h) 转发到消息组件
- i) 到达接入节点域的 GateWayStation
- j) 转发到接入节点的网络层

向上层传输到达接入节点

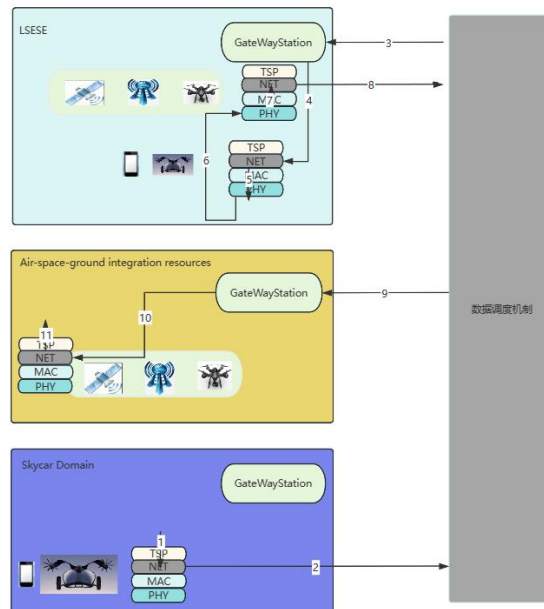


图 4.2 跨域仿真流程图

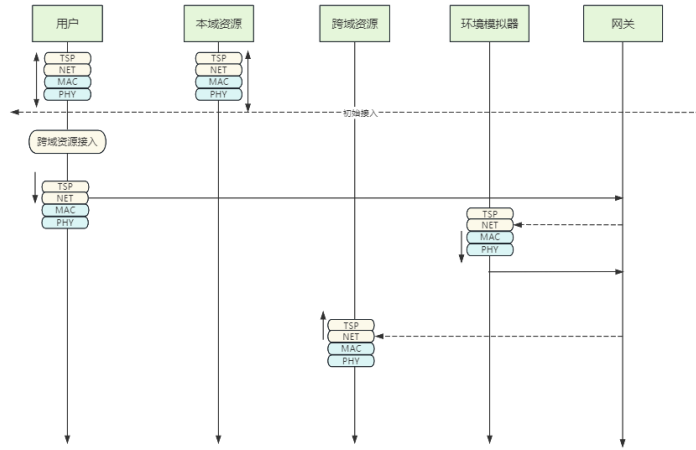


图 4.3 数据流向图

第三步：设计 LSESE 数据包流向。

当 LSESE 返回数据的时候，会通过 MQ 中间件中创建的通往三个域的消息队列，根据数据包信息发往不同的域。针对不同的场景创建了不同的消息队列，不同场景间可以通过消息队列完成 LSESE 场景中数据链路层和物理层的仿真。

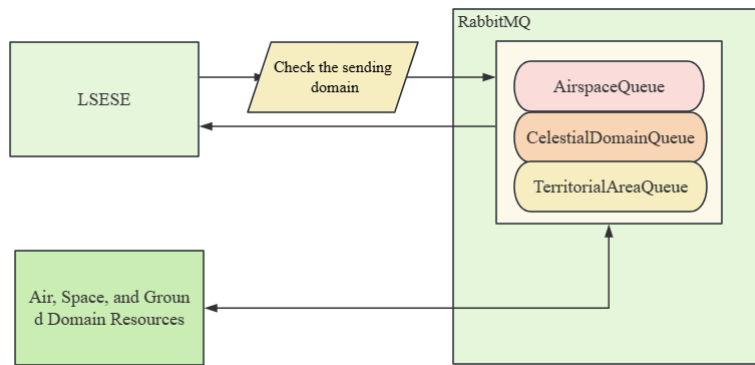


图 4.4 数据包发送示意图

接下来进行全场景连接功能测试用例演示。

第一步：创建无人机场景、卫星场景以及大工程模拟场景。无人机场景中包括用户以及无人机，基站。卫星场景包括低轨道卫星，中轨道卫星以及高轨道卫星。LSESE 场景中设置了用户、无人机、卫星的镜像节点来仿真数据链路层和物理层。

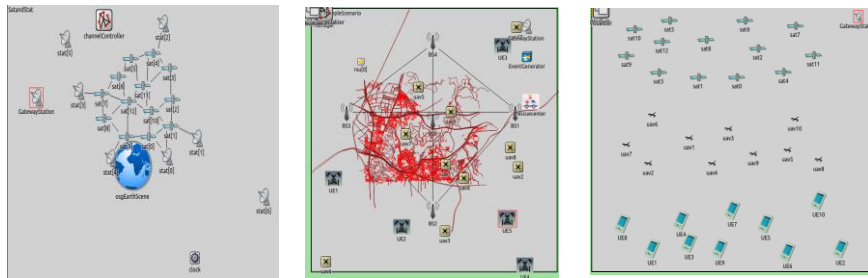


图 4.5 卫星、无人机、大工程模拟环境构建

第二步：当用户需要接入卫星时，首先数据包需要在网络层传入消息组件，

进而在 LSESE 工程中模拟数据链路层和物理层。

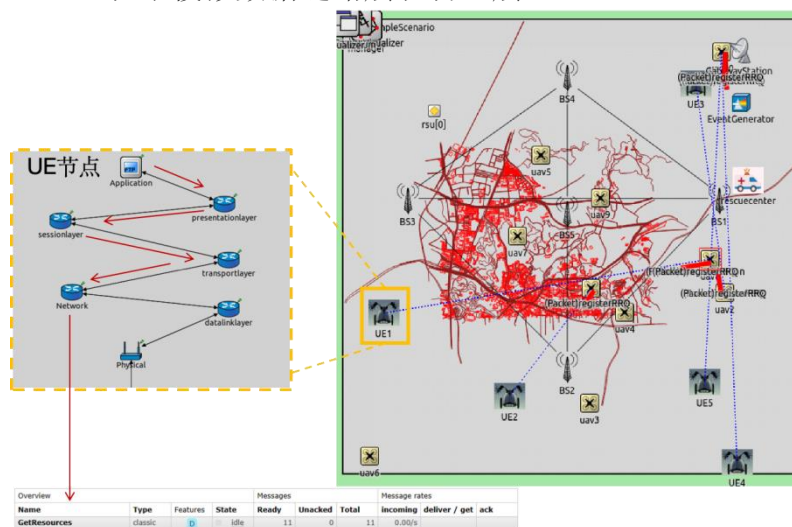


图 4.6 跨域数据包流向

第三步: LSESE 场景可以接收这个数据包, 并且把数据包传入需要接入的卫星, 在本场景中只需要模拟数据链路层和物理层, 并按照事先定义的消息队列把消息发送到不同的项目。

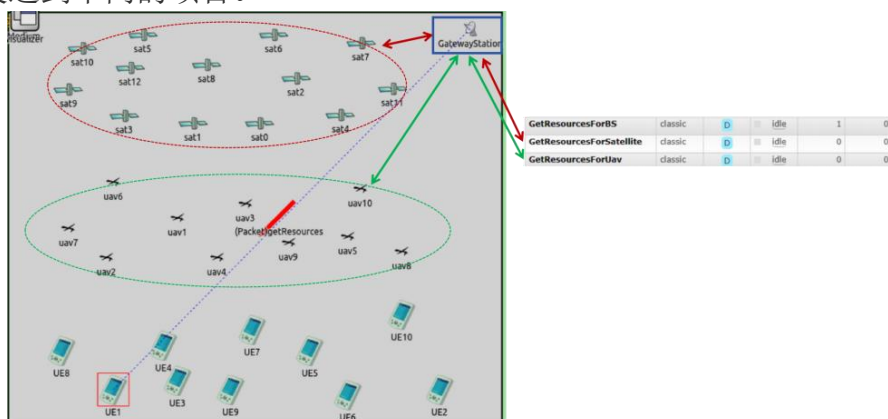


图 4.7 LSESE 数据包流向

第四步: 卫星场景可以接收这个数据包, 并根据数据请求内容来进行相应的操作。卫星场景回复的数据会通过原路径返回。

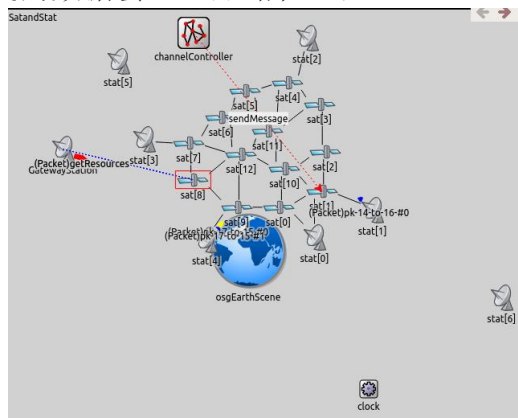


图 4.8 卫星场景接收数据

4.2 前端场景布设及使用

4.2.1 前端平台操作如下

- 步骤 1: 登录
 1. 在浏览器中输入前端平台的 URL。
 2. 输入您的用户名和密码。
 3. 点击“登录”按钮。
- 步骤 2: 场景生成
 1. 在主界面上选择“场景生成”模块。
 2. 在表格中输入以下信息（示例）：
id: 唯一标识符。
name: 实体名称。
x, y: 坐标位置。
type: 实体类型。
mobility: 移动方式。
speed: 移动速度。
angle: 移动角度。
 3. 点击“生成参数”按钮。



图 4.9 场景生成模块

- 步骤 3: 参数存储
 1. 系统会自动将生成的参数存储到数据库中。
 2. 验证参数是否已经存储。

4.2.2 数据库存储验证

1. 使用数据库管理工具连接到数据库。
2. 查询相关表格确认数据存储。

XDNAP 算法上传接口设置（2024.5）

id	name	x	y	type	mobility	speed	angle
1	BS1	10000	10000	BS			
2	Car1	10000	20000	Car	line	10	30
3	People	10000	30000	People	mass	1.2	60
4	che	33333	33333	Car	line	3	55

节点名称

x,y: 要设置对应的范围

节点类型: 只有三种固定类型, 基站就是BS, 汽车就是Car, 用户就是People

汽车和用户的移动速度

汽车和用户的移动方向

汽车和用户要设置移动类型, 可选两种, line (线性移动) 或mass (随机移动)

图 4.10 数据库存储

4.2.3 通信平台操作

- 步骤 1: 配置（正在维护）
 1. 打开 C++通信平台。
 2. 在配置文件中设置数据库连接参数。
- 步骤 2: 数据读取
 1. 执行读取脚本。
 2. 验证数据是否准确读取。

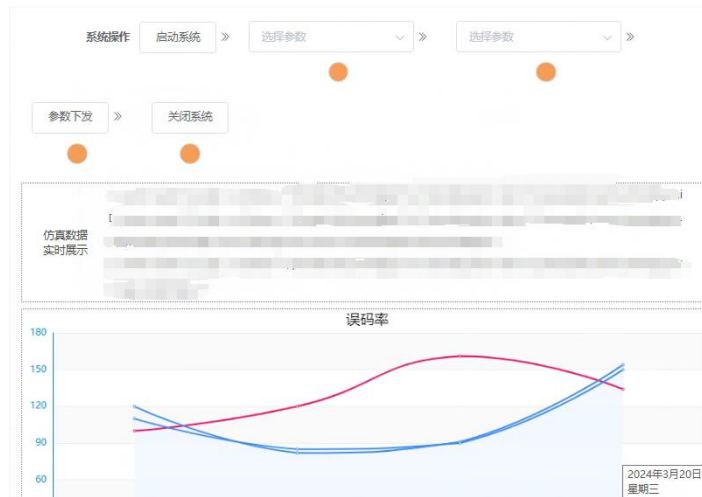


图 4.11 平台启动图

4.2.4 注意事项

1. 确保所有软件版本符合系统要求。
2. 维护密码的安全性。
3. 常见问题解答
 - Q1: 数据不一致怎么办?
 - A1: 请核对前端生成参数的准确性并重新生成。
 - Q2: 数据库连接失败如何解决?
 - A2: 检查网络设置, 确认数据库服务运行正常。

5 算法环境部署标准

在 6G 通信中，随着人工智能技术的广泛应用，Python 的 PyTorch 和 TensorFlow 等框架在算法环境部署中扮演着重要角色。因此，本小节将重点介绍 Python 算法环境部署的标准，以满足在 6G 通信中 AI 应用的需求。对于其他框架和语言的算法环境部署标准，将在后续开发中逐步完善。

Python 作为一种简洁、易读易写的高级编程语言，在人工智能领域得到了广泛应用。PyTorch 和 TensorFlow 作为 Python 中两个主要的深度学习框架，为开发者提供了丰富的工具和功能，使其能够快速构建、训练和部署各类神经网络模型。

在本小节中，将重点介绍 Python 算法环境部署的设计原则、标准化流程、环境配置以及依赖管理等方面。通过遵循这些标准，可以确保算法在不同环境中的顺利部署和运行，提高算法的可靠性和稳定性，从而更好地满足 6G 通信中对 AI 技术的需求。

在未来的开发过程中，将进一步完善其他框架和语言的算法环境部署标准，以确保在不同环境和场景下的算法应用都能够达到统一的质量和效率标准。

5.1 Python 环境部署标准

本平台基于 Anaconda 生成的虚拟环境，旨在提供一种便捷、可视化的方式，用于隔离不同的 Python 环境。通过平台端的控制，用户可以轻松地进行虚拟环境的管理，包括创建、克隆、导出和导入等操作，从而满足不同项目和任务的需求。

1. 创建虚拟环境：平台端提供了可视化界面，用户可以通过简单的操作，在平台上创建新的虚拟环境。用户可以选择所需的 Python 版本、环境名称和依赖库等配置选项，然后平台将自动创建一个独立的虚拟环境。
2. 克隆虚拟环境：平台还支持从已有的虚拟环境克隆出新的环境。用户可以选择已存在的虚拟环境作为模板，然后指定新环境的名称和配置，平台将复制并创建一个与原环境相同的新环境。
3. 导出和导入虚拟环境：平台提供了导出和导入虚拟环境的功能，使用户可以方便地共享和备份虚拟环境。用户可以将虚拟环境导出为一个文件，然后在需要时导入到其他机器或平台上使用。这样可以确保环境的一致性和可移植性。

5.1.1 requirements.txt 标准

当您在本地成功地运行了 Python 程序后，您可以按照以下步骤导出对

应的 requirements.txt 环境包。

1. 打开命令行界面（例如 cmd 或终端）。
2. 进入您的 Python 项目所在的目录。

```
cd path/to/your/python/project
```

3. 确保您已经安装了 pip 工具。如果没有安装，您可以通过以下命令安装 pip：
python -m ensurepip
4. 使用以下命令生成 requirements.txt 文件：

```
pip freeze > requirements.txt
```

这个命令会将当前 Python 环境中已安装的所有包及其版本信息输出到 requirements.txt 文件中。

5. 检查 requirements.txt 文件，确保它包含了您项目中所有需要的包及其对应的版本信息。
6. 如果有特殊的包库或需要指定特定版本，请确保在 requirements.txt 文件中进行相应的编辑。您可以手动添加需要的包及版本信息，格式为 package==version。

完成以上步骤后，您就成功地生成了 requirements.txt 文件，其中包含了您 Python 项目所需要的所有环境包及其对应的版本信息。您可以将这个文件分享给其他人或者在其他环境中使用，以确保能够成功地搭建相同的 Python 环境。如果您有特殊的包库或需要进一步的帮助，请随时联系我们。

5.1.2 Python 版本标准

对于用户在平台上创建虚拟环境，需要提供以下信息。

1. 环境名称：用户需要为新创建的虚拟环境指定一个名称，以便在平台上进行识别和管理。
2. requirements.txt 文件：用户需要提供一个包含项目所需的所有 Python 包及其对应版本的 requirements.txt 文件。这个文件可以包含在项目的代码仓库中，或者由用户提供。它会告诉平台在创建虚拟环境时需要安装哪些包及其版本。
3. Python 版本号：用户需要指定所需的 Python 版本号，以确保虚拟环境中使用的是正确的 Python 解释器。用户可以根据项目需求选择合适的 Python 版本，比如 Python 3.6、Python 3.7 等。

提供了以上信息后，用户就可以通过平台端的控制界面，可视化地创建、克隆、导出和导入虚拟环境，从而实现不同项目和任务之间 Python 环境的隔离和管理。

6 基于微服务技术的算法标准 (Python)

在本节中，我们将专注于 Python 算法内部标准，并将其他方面的说明留待与其他课题对接后进行。

6.1 Python 算法标准

从 2.2 代码书写标准可知，平台是 java 开发，所以上传 python 需要使用 java 和 python 专门的连接包。该链包后续称为 **Send-Plug** 接口。main 函数格式需要写为方法，要存放在根目录下。

Main 函数中的参数需要 Json 文件来进行控制，该样例中主函数名称为 Begin.py 如图 6.1 所示：

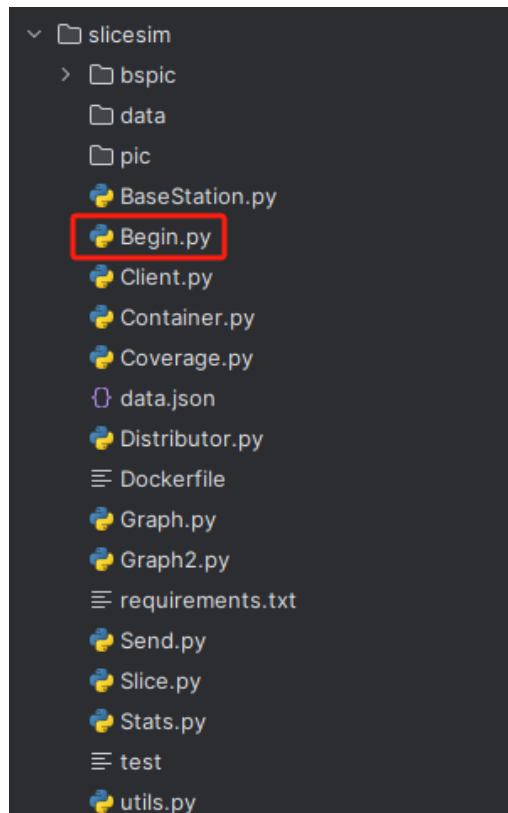


图 6.1 主函数位置图

6.1.1 Json 文件编写规范

图 6.1 中的 **data.json** 文件作为配置的文件，控制整个算法的参数，也作为整个函数的控制端口。Json 文件编写规范，如图 6.2 所示。

```
{
  "settings": {
    "simulation_time": 30,
    "num_clients": 100,
    "limit_closest_base_stations": 20,
    "statistics_params": {
      "warmup_ratio": 0.05,
      "cooldown_ratio": 0.05,
      "x": {
        "min": 0,
        "max": 2000
      },
      "y": {
        "min": 0,
        "max": 2000
      }
    },
    "logging": true,
    "log_file": "output.txt",
    "plotting_params": {
      "plotting": true,
      "plot_save": true,
      "plot_show": true,
      "plot_file": "output",
      "plot_file_dpi": 2000,
      "scatter_size": 1000000
    }
  },
  "slices": {
    "超级无线宽带": {
      "delay_tolerance": 2.5,
      "qos_class": 5,
      "bandwidth_guaranteed": 10000000000,
      "bandwidth_max": 1000000000000,
      "client_weight": 0.1,
      "threshold": 0,
      "usage_pattern": {
```

图 6.2 Json 文件编写规范

缩进:

使用两个空格进行缩进，以提高文件的可读性。

键值对格式:

键和值之间使用冒号和一个空格分隔。

注释:

JSON 本身不支持注释。如果必须添加注释，可以在不影响实际数据的地方（如示例中的注释）使用，但在实际的 JSON 解析中应去除。

键的命名:

键使用小写字母，并使用下划线 _ 分隔单词，以增加可读性。

值的格式:

值可以是字符串、整数、浮点数、布尔值、数组或对象等，根据需要选择合适的数据类型。

标点符号:

键和值都使用双引号括起来（这是 JSON 标准的要求）。

每个键值对之后使用逗号分隔，最后一个键值对不需要逗号。

注意事项：

JSON 文件中不能包含单行或多行注释。如果需要包含注释，通常可以通过额外的键值对来描述，如 "comment": "This is a comment"，但这不是标准做法，仅适用于特定场景。确保 JSON 文件始终是有效的 JSON 格式，否则解析会失败。遵循这些规范可以使 JSON 文件更易于理解和维护，并且提高文件的可读性和一致性。

6.1.2 主函数算法标准

1. 读取配置文件路径：程序首先构造了配置文件的路径。这里使用了 `os.path.dirname(__file__)` 获取当前脚本所在目录，然后使用 `sys.argv[1]` 获取命令行参数中传递的配置文件名，并将两者拼接成完整的文件路径。
2. 尝试打开配置文件：使用 `open()` 函数打开配置文件，该文件通过平台进行上传，指定位置。指定以只读模式打开，并指定编码为 UTF-8。文件流被赋值给 `stream` 变量，如图 6.3。
3. 加载配置数据：调用 `yaml.load(stream, Loader=yaml.FullLoader)` 函数，从打开的文件流中加载配置数据。`yaml.load()` 函数用于将 YAML 格式的数据加载为 Python 对象。加载后的数据保存在变量 `data` 中。
4. 处理文件不存在情况：如果配置文件不存在，捕获 `FileNotFoundError` 异常，并打印错误信息 "File Not Found"，然后程序退出。
5. 提取设置信息：从 json 文件加载的数据中提取出 `settings` 部分，并将其保存在 `SETTINGS` 变量中，如图 6.4。
6. 获取客户端数量：从 `SETTINGS` 中提取出键为 `num_clients` 的值，并将其保存在 `NUM_CLIENTS` 变量中。

```
# Read Json file 读取配置文件
CONF_FILENAME = os.path.join(os.path.dirname(__file__), "data.json")
try:
    with open(CONF_FILENAME, 'r', encoding='utf-8') as stream:
        data = json.load(stream)
except FileNotFoundError:
    print('File Not Found:', CONF_FILENAME)
    exit(0)
```

图 6.3 Json 文件读取标准

```

SETTINGS = data['settings']
NUM_CLIENTS = SETTINGS['num_clients']

```

图 6.4 函数调用标准

6.2 基于微服务架构的算法样例

微服务架构是一种软件架构风格，其中应用程序被构建为一组小型的、独立部署的服务，这些服务围绕着业务能力进行组织，每个服务都运行在自己的进程中，并通过轻量级的通信机制相互通信。在网络算法上传的内容中，我们可以进一步扩充微服务架构的讨论，详细介绍各个接口标准和实现方法的摘要。如图 6.5 所示，在基于微服务的平台架构中，利用微服务技术构建算法管理库是十分重要的，所以本节将描述合理的微服务算法架构。本节需要结合该程序和本文档进行逐步调试，程序下载路径：<https://github.com/Weihaomao/6G-Send-Plug>

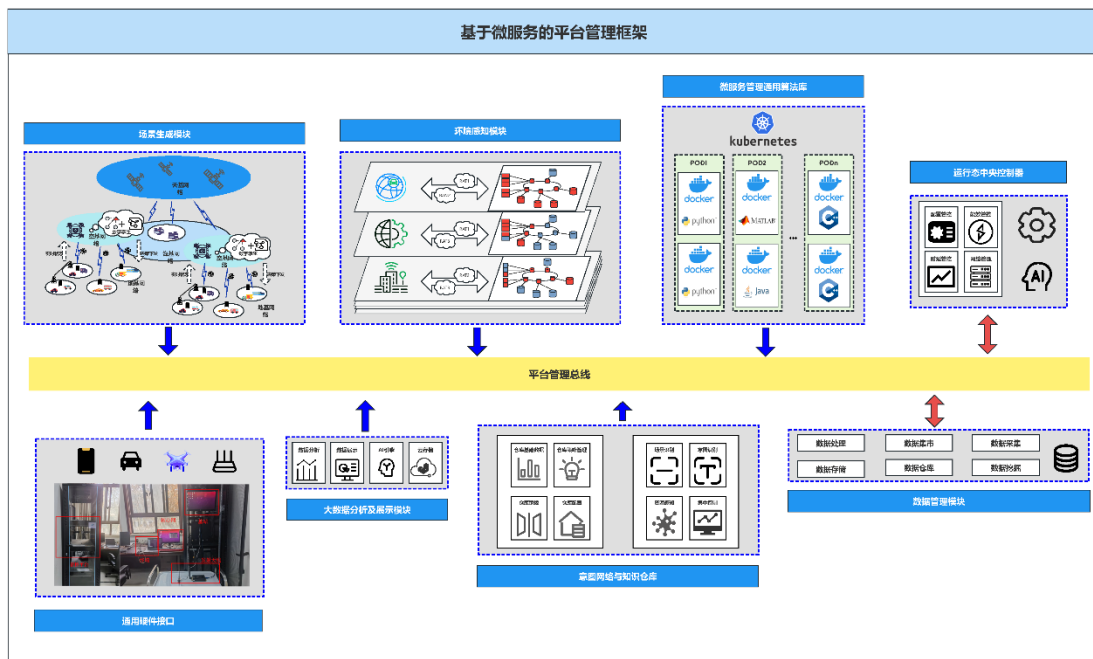


图 6.5 基于微服务的平台管理架构

结合第 3 节中网络算法上传的内容，本节利用微服务架构详细的讲解了各个接口标准和实现方法的摘要。

接收请求：Flask 应用程序定义了一个路由，例如/update_properties，用于接收 POST 请求。

解析请求数据：当客户端向该路由发送 POST 请求时，Flask 应用程序使用 `request.get_json()` 方法从请求中提取 JSON 数据。这个 JSON 数据可能包含有关于要调整的 Python 属性的信息。

处理请求：应用程序中的视图函数根据请求中的数据进行处理。可能的操作包括修改全局变量、调用相关函数或方法等。例如，根据请求中的数据更新 Python 属性的值。

响应请求：处理完请求后，应用程序向客户端发送适当的响应。这可能是一个简单的成功或失败消息，或者是返回更新后的属性数据。

应用更改：根据请求中的数据，应用程序对 Python 属性进行相应的更改。这确保了应用程序对请求做出了正确的响应，并使得 Python 属性的值与客户端期望的一致。

通过上述流程摘要接下来进行讲解详细的基于微服务架构的接口设置。

6.2.1 算法路径标准

在这个 Python 示例中，我们以图形化的方式展示了整个流程。基于上一版本的算法接口文档，本文档做出了更新。更新内容入如下：首先，必要路径中包含一个名为 **data** 的文件夹，用于存储生成的数据文件，不再是 **pic** 文件夹（更新后不仅有图片文件），**data** 文件夹可接收的文件后缀有：**.png\gif\text**，支持图片、动态图、**txt** 数据文档。这些数据文件可能是图片或文本，它们通过 base64 编解码进行处理，以便在需要时进行传输和存储。

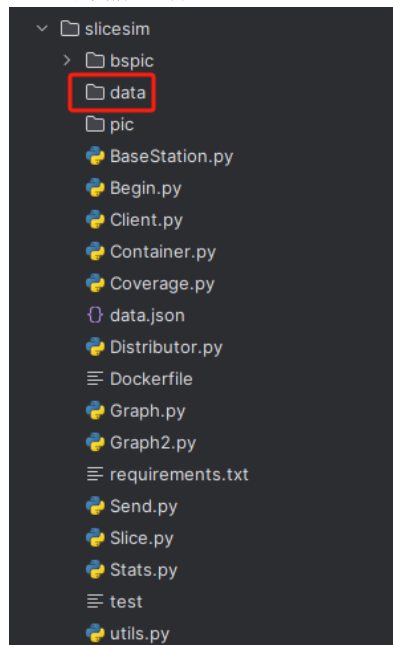


图 6.6 算法路径标准下的文件

外部环境通过 POST 请求传入的 JSON 数据存储于根目录(图 6.6 中为 slicesim)

下，作为配置文件的输入。这些 JSON 数据可以包含各种设置和参数（图 6.5 中的 data.json），用于控制程序的行为和输出。

我们提供了一个名为 Send.py 的脚本，该脚本实现了与数据库的交互和调用。它使用了 Flask 框架来实现一个简单的 Web 服务，接收 POST 请求并生成相应的配置文件。一旦接收到请求，脚本会将配置文件存储到数据库中，并且可以根据需要对数据库进行查询和修改。

整个流程通过 Python 脚本自动化和控制，使得数据的传输和处理变得更加高效和可靠。同时，Flask 框架的使用使得 Web 服务的实现变得简单而灵活，可以方便地扩展和定制功能。

6.2.2 读取 Json 文件

通过 6.1.2 主函数调用写法的内容，我们读取 Json 文件（Yaml 文件也同）中的内容，此处样例也给出 Json 文件的读取方法：

```
CONF_FILENAME = os.path.join(os.path.dirname(__file__), "data.json")

try:
    with open(CONF_FILENAME, 'r', encoding='utf-8') as stream:
        data = json.load(stream)
except FileNotFoundError:
    print('File Not Found:', CONF_FILENAME)
    exit(0)

SLICES_INFO = data['slices']
NUM_CLIENTS = SETTINGS['num_clients']
MOBILITY_PATTERNS = data['mobility_patterns']
BASE_STATIONS = data['base_stations']
CLIENTS = data['clients']
```

以上实现了对 data.json 文件的读取，具体参数名称要依靠各自算法中的内容而定。

6.2.3 Send.py 脚本的使用

首先，确保主函数和 Send.py 脚本在同一个目录中，并且在 Send.py 中引用主函数。然后，我们需要将 Send.py 修改为可执行脚本，并将其作为启动函数。

在 Send.py 脚本中，添加以下代码来引用主函数：

```
import Begin
```

```
from flask import Flask, request, jsonify
import json
import pymysql.cursors
import Begin
import os
import base64
```

图 6.7 引入模块

接下来，我们需要对数据库进行设置，例如：根据提供的信息，数据库的主机名是 `w6109r6604.goho.co`，端口是 `10195`，用户名是 `root`，密码是 `password`，数据库名是 `data`。该数据为测试数据库，供所有课题进行测试使用。

```
# MySQL 数据库连接配置
MYSQL_HOST = 'w6109r6604.goho.co'
MYSQL_PORT = 10195
MYSQL_USER = 'root'
MYSQL_PASSWORD = 'password'
MYSQL_DB = 'data'
```

图 6.8 MySQL 配置

```
MYSQL_HOST = 'w6109r6604.goho.co'
```

```
MYSQL_PORT = 10195
```

```
MYSQL_USER = 'root'
```

```
MYSQL_PASSWORD = 'password'
```

```
MYSQL_DB = 'data'
```

最后，再 80 行左右找到主函数调用位置，填入你的函数主函数启动：

```
#!/usr/bin/env python
# !! 调用主程序位置：
# Begin.main() 这个方法弃掉 容易出现内存泄漏！
subprocess.run(["python", "Begin.py"])
```

图 6.9 主程序位置

至此，我们已经成功设置了程序内微服务的大致架构。我们已经创建了一个 Flask 应用程序，定义了路由和视图函数来处理 POST 请求，并设置了与数据库的连接参数。此外，我们还指定了存储文件和配置文件的路径，并在需要时使用 `base64` 进行编解码。接下来，我们可以进一步完善代码，处理 POST 请求中的数据，并根据需要对 Python 属性进行调整和更新。

在数据库内存储的信息我们同样配备了 `base64` 的解码器，将提供给大家用于解码数据，验证是否正确使用脚本。

6.2.4 requirement.txt

在 5.1.1 中我们已经讲述了 requirement.txt 的具体内容, 这里我们提到 Send.py 需要的几个依赖包, 手动添加到你已经生成的文件中:

Flask==2.2.5

PyMySQL==1.1.0

(版本这里不进行限制, 这里的版本是参考, 如果 python 版本不统一可以生成其他版本的 Flask 和 PyMySQL)

6.2.5 声明项目名称

通过该名称使得项目之间不会重名、在结果数据中查询数据时不会出现混乱的情况。

修改名称位置为: Send.py 接口下的该位置, 本示例名为 slice2, **该名字需要和课题 5 负责人对接后使用。**

```
#创建自己的项目名称
my_project_name = "slice2"
```

图 6.10 项目名称

6.3 数据库展示

首先从检查数据库是否已生成对应的表单, 和 Send.py 脚本中一致, 此处创建表单分别为 program 和 program_output, 分别作为输入参数和输出参数作为记录。

```
# 将 JSON 数据写入 MySQL 数据库
with connection.cursor() as cursor:
    # 创建表
    create_table_query = """
    CREATE TABLE IF NOT EXISTS program (
        program_setting_time TEXT,
        data TEXT
    )
    """
    cursor.execute(create_table_query)

    from datetime import datetime

    # 获取当前时间
    current_time = datetime.now()

    # 插入数据
    insert_query = "INSERT INTO program (program_setting_time, data) VALUES (%s, %s)"
    cursor.execute(insert_query, args=(current_time, json.dumps(data)))
```

图 6.11 program 输入数据库程序

```

with connection.cursor() as cursor:
    # 创建表
    create_table_query = """
    CREATE TABLE IF NOT EXISTS program_output (
        program_setting_time TEXT,
        filename TEXT,
        content LONGTEXT
    )
    """
    cursor.execute(create_table_query)
    save_files_in_directory_to_db('pic')

```

图 6.12 program_output 输出数据库程序

program 表单和 program_output 表单有以下字段：

1. program_setting_time: 存储程序设置时间的文本字段。
2. data: 存储数据的文本字段。
3. filename: 存储文件名的文本字段。
4. content: 存储文件内容的长文本字段。

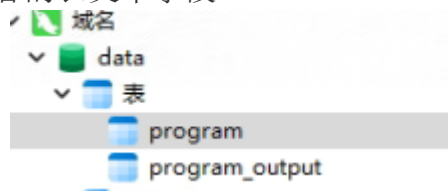


图 6.13 数据库表单内容

通过 6.4 利用 Docker 实现微服务架构的 Send.py 的介绍，将输入参数和输出数据存储于数据库，利用 Base64 进行解码，接下来利用配套内容加以验证：

对象	program @data (域名) - 表	program_output @data (域名) - 表
	开始事务	文本
	筛选	排序
	导入	导出
	数据生成	创建图表
program_setting_time	data	
2024-04-01 17:54:43.351271	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 17:55:20.799520	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 17:55:24.560662	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 21:49:24.509475	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 14:10:52.597830	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 14:11:02.716875	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 14:11:04.511904	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	
2024-04-01 14:15:01.353373	{'settings': {'simulation_time': 100, 'num_clients': 10, 'limit_closest_base_stati	

图 6.14 MySQL 中输入参数

对象	program @data (域名) - 表	program_output @data (域名) - 表
开始事务	文本	筛选 排序 导入 导出 数据生成 创建图表
program_setting_time	filename	content
2024-04-01 17:54:44.374398	output.png	iVBORw0KGgoAAAANSUHuEgAABkAAAAOECAyAAAD5Tf2iAAAAAHNC
2024-04-01 17:54:44.374398	output.txt	QINfMCAJIGNvdjpbYz0MTUwLCwgMTkwMCKsiHl9MTUwMF0JHdpc
2024-04-01 17:55:25.714842	output.png	iVBORw0KGgoAAAANSUHuEgAABkAAAAOECAyAAAD5Tf2iAAAAAHNC
2024-04-01 17:55:25.714842	output.txt	QINfMCAJIGNvdjpbYz0MTUwLCwgMTkwMCKsiHl9MTUwMF0JHdpc
2024-04-01 21:49:25.735192	output.png	iVBORw0KGgoAAAANSUHuEgAABkAAAAOECAyAAAD5Tf2iAAAAAHNC
2024-04-01 21:49:25.735192	output.txt	QINfMCAJIGNvdjpbYz0MTUwLCwgMTkwMCKsiHl9MTUwMF0JHdpc
2024-04-01 14:15:02.795239	output.txt	QINfMCAJIGNvdjpbYz0MTUwLCwgMTkwMCKsiHl9MTUwMF0JHdpc
2024-04-01 14:15:02.795239	output.png	iVBORw0KGgoAAAANSUHuEgAABkAAAAOECAyAAAD5Tf2iAAAAAHNC

图 6.15 MySQL 中输出参数

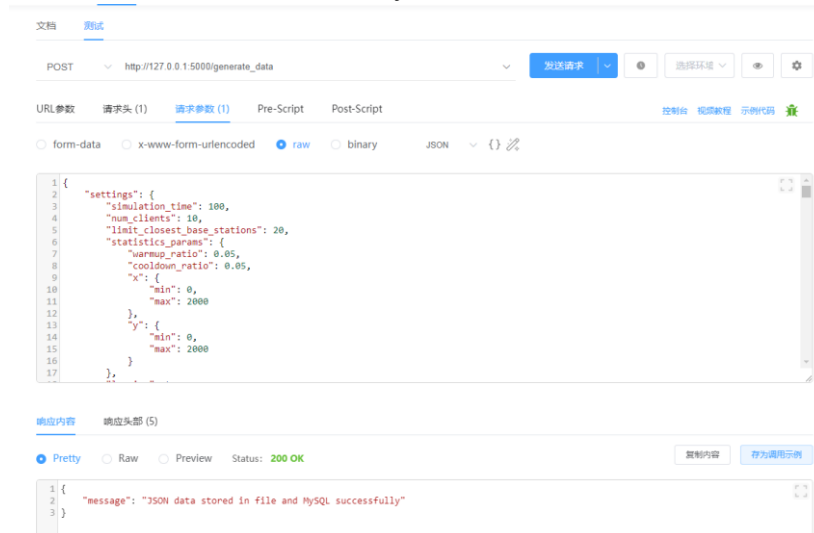


图 6.16 发送 POST 请求

通过图 6.18 发送 POST 请求，得到图 6.16 和图 6.17 中的参数，可以验证数据内容可以通过持续的 POST 请求进行上传和更新。这种持续更新可以帮助优化新的结果，并将它们存储到平台中。随着数据的不断更新，可以进行可视化展示，并验证按需服务的效果。

利用本平台提供的编解码脚本验证是否传输成功:

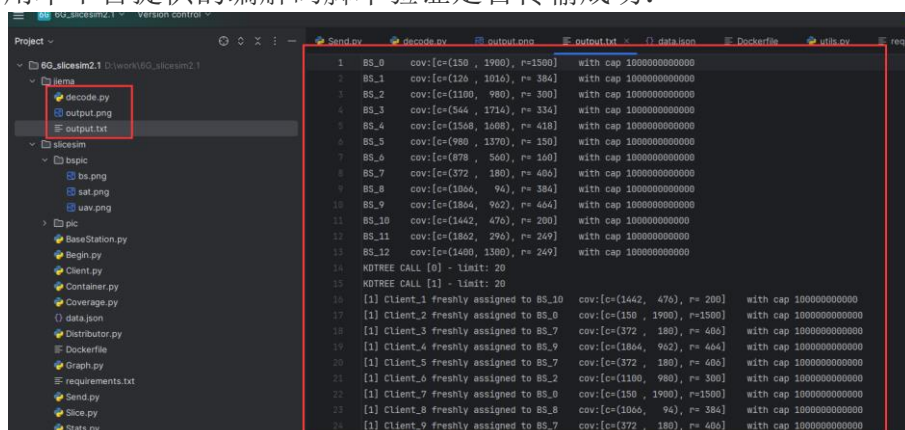


图 6.17 编解码器验证脚本

通过如图 6.17 所示的结果，该解码器可适配任何格式的文件，包括 png、txt 等，利用 base64 位进行编解码。同理，Send.py 脚本中也内置了多样化的编码器，

实现各种文件的编码。

至此，微服务架构的算法上传 python 模块已经完成。

6.4 利用 Docker 实现微服务架构

完成以上请先联系课题 5 检查是否正确，正确后修改平台数据库后再操作以下内容。

首先确保你本地环境已经拥有一个 Docker，在根目录下写一个 DockerFile 编译整个程序。并且，检查所有的函数，不得使用绝对路径！

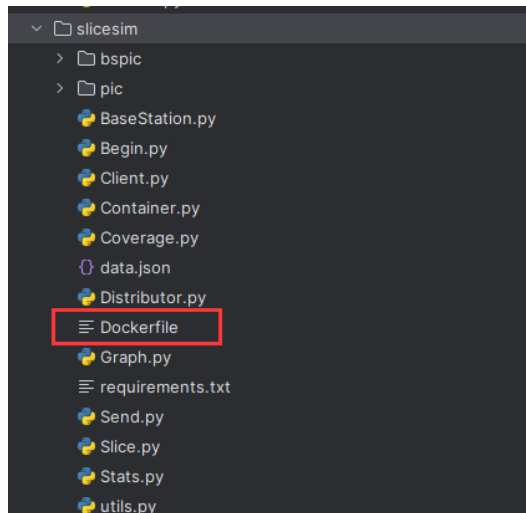


图 6.18 DockerFile 位置

编写一个 DockerFile，下面是一个简单的 DockerFile 示例，用于构建一个基于 Python 的 Web 应用程序：

<https://docs.docker.com/engine/reference/builder/#run>

使用官方 Python 基础镜像

FROM python:3.9-slim

设置工作目录

WORKDIR /app

将当前目录中的所有内容复制到工作目录中

COPY . .

安装依赖项

RUN pip install --no-cache-dir -r requirements.txt

暴露端口

EXPOSE 5000

定义环境变量

ENV FLASK_APP=app.py

启动应用

CMD ["flask", "run", "--host=0.0.0.0"]

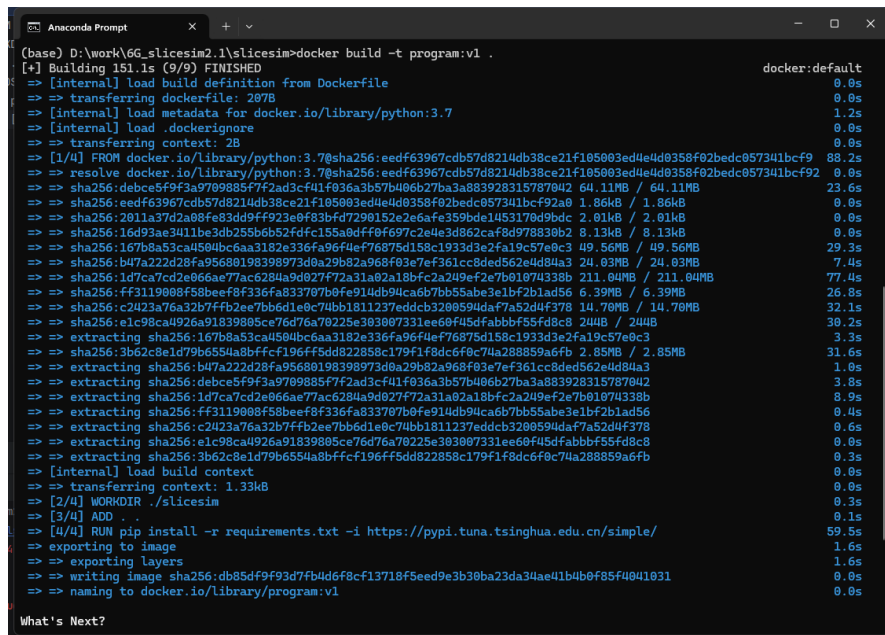
```
1 FROM python:3.7
2 WORKDIR ./slicesim
3 ADD . .
4 EXPOSE 5000
5 RUN pip install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
6 CMD ["python", "./Send.py"]
```

图 6.19 DockerFile 内容

在 cmd 中编译 Docker 镜像:

参考文档: <https://docs.docker.com/engine/reference/commandline/build/>

docker build -t test:v1 .



```
(base) D:\work\6G_slicesim2.1\slicesim>docker build -t program:v1 .
[+] Building 151.1s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 207B
=> [internal] load metadata for docker.io/library/python:3.7
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/4] FROM docker.io/library/python:3.7@sha256:eedf63967c5b57d8214db38ce21f105003ed4e4d0358f02bedc057341bcf9
=> => resolve docker.io/library/python:3.7@sha256:eedf63967c5b57d8214db38ce21f105003ed4e4d0358f02bedc057341bcf9
=> => sha256:debc5f9f3a9709885f7f2ad3cf41f036a3b57b406b27ba3a833928315787042 64.11MB / 64.11MB
=> => sha256:eedf63967c5b57d8214db38ce21f105003ed4e4d0358f02bedc057341bcf9 1.86kB / 1.86kB
=> => sha256:2011a37d2a08fe83d4d9ff923e8f83bf7290152e2a6afe35b0de145317049bdc 2.01kB / 2.01kB
=> => sha256:16d93ae3411be3db255b6b52fdcf155a0d4ff0f697c24e43d862caf8d978830b2 8.13kB / 8.13kB
=> => sha256:167b8a53ca4504bc6aa3182e336fa96f4ef76875d158c1933d3e2fa19c57e0c3 49.56MB / 49.56MB
=> => sha256:b47a22d28fa95680198398973d0a29b82a968f03e7ef361cc8ded562e4d84a3 24.03MB / 24.03MB
=> => sha256:1d7ca7cd2e066ae77ac6284a9d027f72a31a02a18bfc2a249ef2e7b01074338b 211.04MB / 211.04MB
=> => sha256:ff3119008f58bee8f336fa833707b0fe914db94ca6b7bb55abe3e1bf2b1ad56 6.39MB / 6.39MB
=> => sha256:c2423a76a32b7fb2ee7bb6d1e0c74bb1811237eddc3200594da7a52d4f378 14.70MB / 14.70MB
=> => sha256:elc98ca4926a91839805ce76d76a70225e303007331ee60f45dfabbbf55fd8c8 244B / 244B
=> => extracting sha256:167b8a53ca4504bc6aa3182e336fa96f4ef76875d158c1933d3e2fa19c57e0c3
=> => sha256:3b62c8e1d79b6554a8bffc196ff5dd822858c179f1f8dc6f0c74a288859a6fb 2.85MB / 2.85MB
=> => extracting sha256:b47a22d28fa95680198398973d0a29b82a968f03e7ef361cc8ded562e4d84a3
=> => extracting sha256:debc5f9f3a9709885f7f2ad3cf41f036a3b57b406b27ba3a833928315787042
=> => extracting sha256:1d7ca7cd2e066ae77ac6284a9d027f72a31a02a18bfc2a249ef2e7b01074338b
=> => extracting sha256:ff3119008f58bee8f336fa833707b0fe914db94ca6b7bb55abe3e1bf2b1ad56
=> => extracting sha256:c2423a76a32b7fb2ee7bb6d1e0c74bb1811237eddc3200594da7a52d4f378
=> => extracting sha256:elc98ca4926a91839805ce76d76a70225e303007331ee60f45dfabbbf55fd8c8
=> => extracting sha256:3b62c8e1d79b6554a8bffc196ff5dd822858c179f1f8dc6f0c74a288859a6fb
=> [internal] load build context
=> => transferring context: 1.33kB
=> [2/4] WORKDIR ./slicesim
=> [3/4] ADD . .
=> [4/4] RUN pip install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
=> => exporting to image
=> => exporting layers
=> => writing image sha256:db85df9f93d7fb4d6f8cf13718f5eed9e3b30ba23da34ae41b4b0f85f4041031
=> => naming to docker.io/library/program:v1
What's Next?
```

图 6.20 Docker 编译示意图

运行容器:

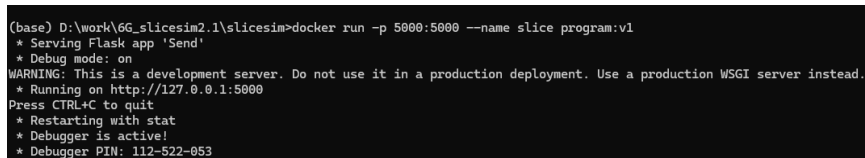
参考文档: <https://docs.docker.com/engine/reference/run/>

docker run -d -p 5000:5000 --name slice program:v1

-p 映射容器内端口到宿主机

--name 容器名字

-d 后台运行



```
(base) D:\work\6G_slicesim2.1\slicesim>docker run -p 5000:5000 --name slice program:v1
* Serving Flask app 'Send'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 112-522-053
```

图 6.21 Docker 容器启动示意图

6.5 移植镜像步骤

1. 在本地电脑上打开终端，查看镜像列表，找到需要的镜像

`docker images`

2. 将需要的镜像保存（保存的文件名就是将镜像保存到本地后起的名称需要加后缀的如 xxx.tar）

`docker save -o 保存的文件名 保存的镜像`

`docker save -o /data/export/test.tar test:4.0`

3. 得到 xxx.tar 文件上传至服务器，等待加载完成后，该算法已经成功在服务端运行

`docker load -i /data/export/test.tar`

6.6 移植镜像挂载

由于修改程序后每次重新打包生成镜像十分麻烦，故节讲述如何在生成的容器中实现挂载本地路径。

`docker run -p 5000:5000 --name slicegif7 -v D:\work\6G_slicesim3\slicesim:/slicesim -d slicegif:v7`

此处-v 实现路径挂载，D:\work\6G_slicesim3\slicesim 为本地路径，/slicesim 为 docker 内的路径，其将路径进行挂载，实现每次代码无需重新生成镜像。

6.7 平台效果展示

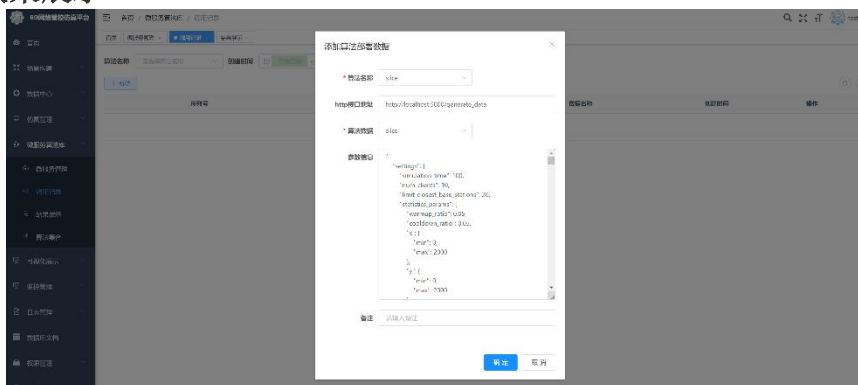
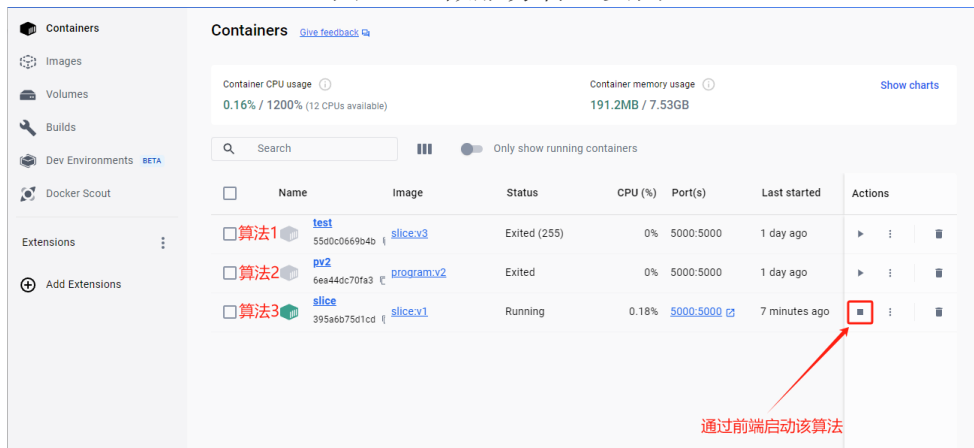


图 6.22 添加场景数据

ID	场景名称	场景API地址	状态	备注	创建时间	操作
1	slice	http://localhost:5000/generate_data	ON	测试数据	2024-04-04 02:25:17	编辑 删除
2	test	http://localhost:5000/test	ON	测试	2024-04-04 02:26:55	编辑 删除

图 6.23 微服务管理页面



通过前端启动该算法

图 6.24 Docker 显示页面

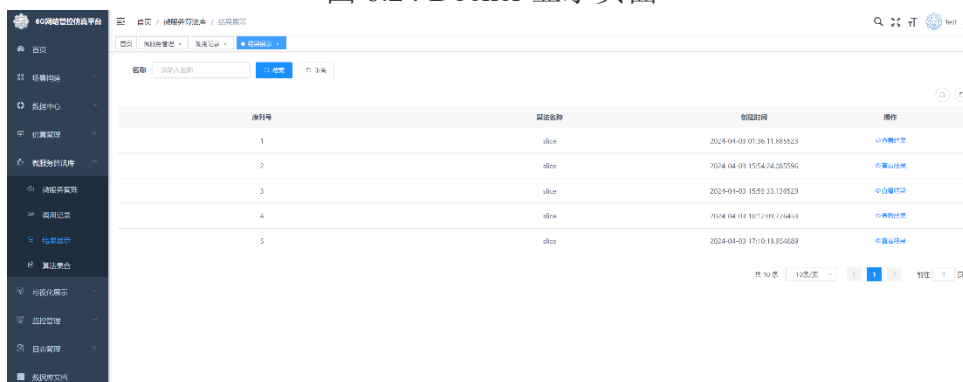


图 6.25 结果数据回滚

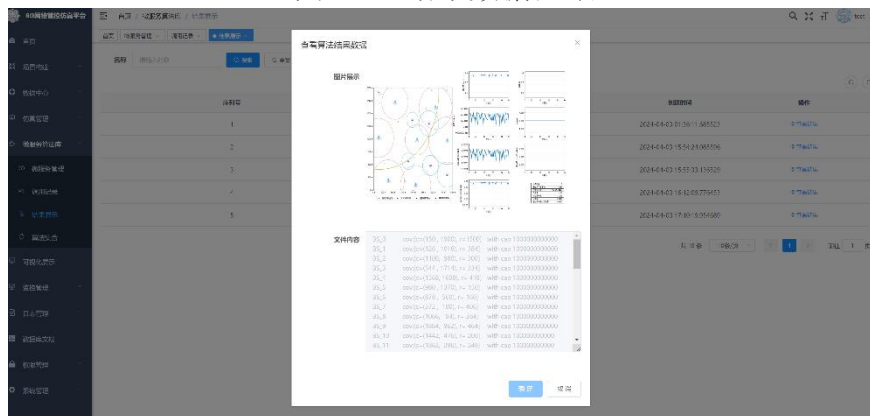


图 6.26 结果数据展示

7 基于微服务技术的算法标准（MATLAB）

在本节中，我们将专注于 Matlab 算法内部标准，并将其他方面的说明留待与其他课题对接后进行。

7.1 Matlab 算法标准

上传 Matlab 需要使用 java 和 python 的连接包。即第六章的 Send-Plug 接口。main 函数格式需要写为方法，要存放在根目录下。

Main 函数中的参数需要 Json 文件来进行控制，该样例中主函数名称为 PL_free.m，如图 7.1 所示：

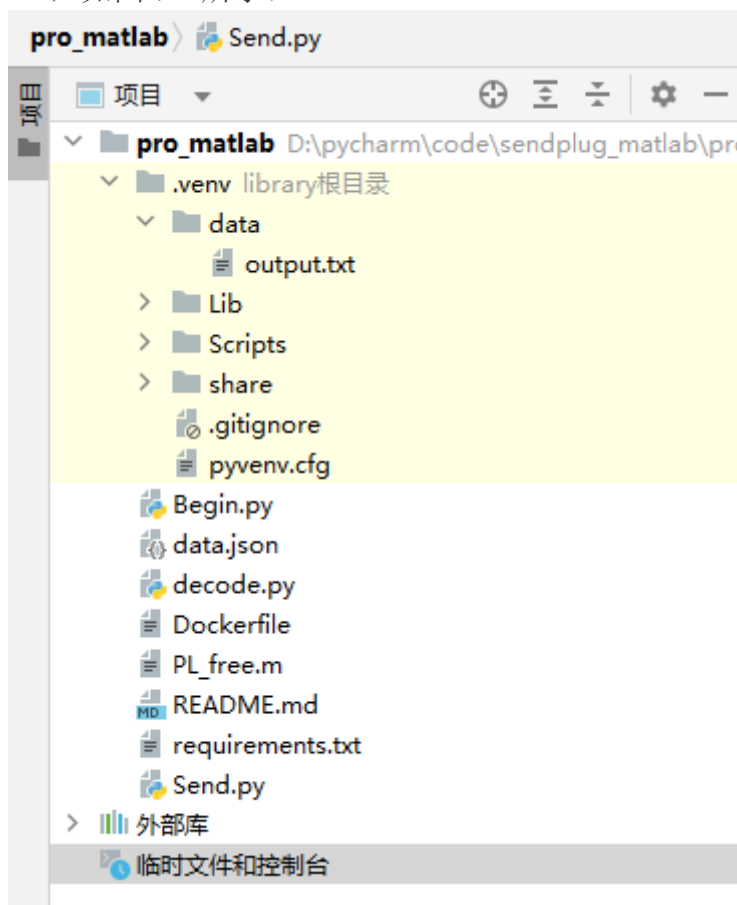


图 7.1 主函数位置图

7.1.1 Json 文件编写规范

图 7.1 中的 data.json 文件作为配置的文件，控制整个算法的参数，也作为整个函数的控制端口。Json 文件编写规范，具体参考 6.1.1。

7.1.2 主函数算法标准

1. 读取 json 文件内容：程序首先构造了配置文件的路径。这里使用了 fileread 函数读取 json 文件的内容，并将其存储在 jsonstr 变量中。
2. 字符串解码：这里使用 Matlab 自带的 jsondecode 函数将 json 字符串解码为 Matlab 结构体，存储在 jsonData 变量中，如图 7.2。
3. 提取字段信息：使用 jsonData. 函数从 jsonData 中提取数据，并将其保存在 fc、dist、Gt、Gr 中，如图 7.2。


```

jsonStr = fileread('D:\pycharm\code\sendplug_matlab\pro_matlab\data.json');
jsonData = jsondecode(jsonStr);
fc=jsonData.fc;
dist=jsonData.dist;
Gt=jsonData.Gt;
Gr=jsonData.Gr;

```

图 7.2 Json 文件读取标准

7.2 基于微服务架构的算法样例

本节需要结合第六章的程序和本文档进行逐步调试。利用微服务架构详细的讲解了各个接口标准和实现方法。

7.2.1 算法路径标准

在这个 Matlab 示例中，我们以图形化的方式展示了整个流程。首先，必要路径中包含一个名为 data 的文件夹，用于存储生成的数据文件，data 文件夹可接收的文件后缀有：.png\gif\txt，支持图片、动态图、txt 数据文档。这些数据文件可能是图片或文本，它们通过 base64 编解码进行处理，以便在需要时进行传输和存储。

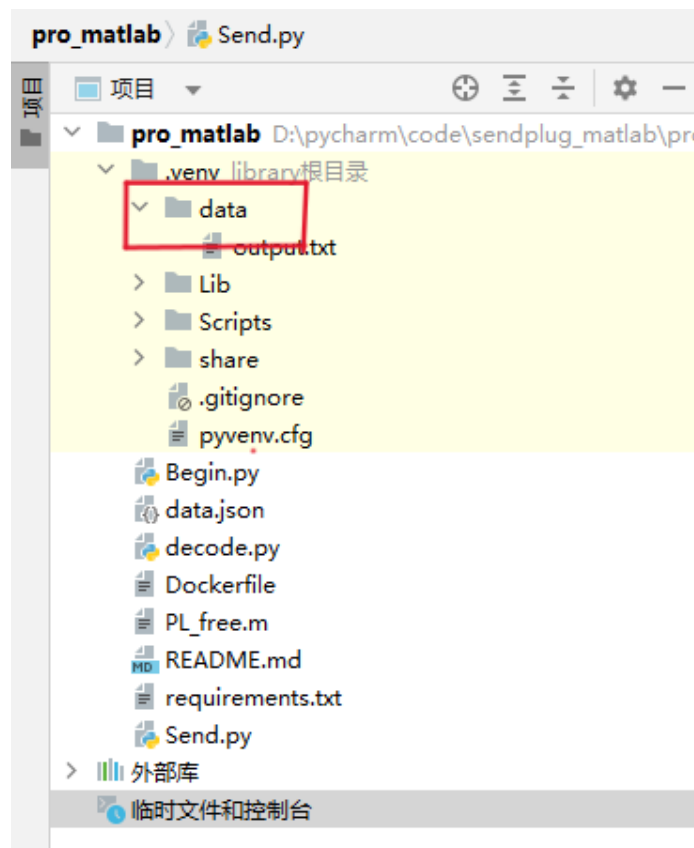


图 7.3 算法路径标准下的文件

外部环境通过 POST 请求传入的 JSON 数据存储在根目录(图 7.3 中为

pro_matlab 下), 作为配置文件的输入。这些 JSON 数据可以包含各种设置和参数 (图 7.6 中的 data.json), 用于控制程序的行为和输出。

使用 Send.py 的脚本, 实现外部环境 with json 文件的交互。它接收 POST 请求, 生成相应的配置文件, 并调用主函数。一旦接收到请求, 脚本会将外部数据写入 json 文件, 并将配置文件存储到数据库中, 然后调用主函数, 主函数会读取 json 文件的内容, 并将最后的算法结果存储到数据库中。

整个流程通过 Python 脚本自动化和控制, 使得数据的传输和处理变得更加高效和可靠。同时, Flask 框架的使用使得 Web 服务的实现变得简单而灵活, 可以方便地扩展和定制功能。

7.2.2 主函数读取 Json 文件

通过 7.1.2 主函数调用写法的内容, 读取 Json 文件中的内容, 此处样例给出 Json 文件的读取方法:

```
jsonStr = fileread('D:\pycharm\code\sendplug_matlab\pro_matlab\data.json');
jsonData = jsondecode(jsonStr);
```

以上实现了对 data.json 文件的读取, 具体参数名称要依靠各自算法中的内容而定。

7.2.3 Send.py 脚本的使用

首先, 确保主函数和 Send.py 脚本在同一个目录中, 并且在 Send.py 中引用主函数。然后, 我们需要将 Send.py 修改为可执行脚本, 并将其作为启动函数。

在 Send.py 脚本中, 添加以下代码来引用主函数:

```
import matlab
import matlab.engine

from flask import Flask, request, jsonify
import json
import pymysql.cursors
import base64
import matlab
import matlab.engine
```

图 7.4 引入模块

最后, 找到主函数调用位置, 填入 Matlab 函数主函数启动:

```

!!! 调用主程序位置:
# Begin.main() 这个方法弃掉 容易出现内存泄漏!
# subprocess.run(["matlab", "PL_free.m"])
eng = matlab.engine.start_matlab()
eng.eval("run PL_free.m", nargout=0)
eng.quit()

```

图 7.5 主程序位置

7.2.4 主函数连接数据库

我们需要在主函数中对数据库进行设置，例如：根据提供的信息，数据库的主机名是 `w6109r6604.goho.co`，端口是 `10195`，用户名是 `root`，密码是 `password`，数据库名是 `my_database`。该数据为测试数据库，供所有课题进行测试使用。

```

# MySQL 数据库连接配置
MYSQL_HOST = 'w6109r6604.goho.co'
MYSQL_PORT = 10195
MYSQL_USER = 'root'
MYSQL_PASSWORD = 'password'
MYSQL_DB = 'my_database'

# 创建自己的项目名称
my_project_name = "plug_matlab"

# 建立 MySQL 数据库连接
connection = pymysql.connect(host=MYSQL_HOST,
                             port=MYSQL_PORT,
                             user=MYSQL_USER,
                             password=MYSQL_PASSWORD,
                             db=MYSQL_DB,
                             charset='utf8mb4',
                             cursorclass=pymysql.cursors.DictCursor)

```

图 7.6 MySQL 配置

```
MYSQL_HOST = 'w6109r6604.goho.co'
```

```
MYSQL_PORT = 10195
```

```
MYSQL_USER = 'root'
```

```
MYSQL_PASSWORD = 'password'
```

```
MYSQL_DB = 'my_database'
```

至此，我们已经成功设置了程序内微服务的大致架构。我们已经创建了一个 Flask 应用程序，定义了路由和视图函数来处理 POST 请求，并设置了与数据库的连接参数。此外，我们还指定了存储文件和配置文件的路径，并在需要时使用 base64 进行编解码。

7.3 数据库展示

首先从检查数据库是否已生成对应的表单，和 `Send.py` 脚本中一致，此处创

建表单分别为 `program_input` 和 `program_output`，分别作为输入参数和输出参数作为记录。

```
# 将 JSON 数据写入 MySQL 数据库
with connection.cursor() as cursor:
    # 创建表
    create_table_query = """
    CREATE TABLE IF NOT EXISTS program_input (
        program_setting_time TEXT,
        data TEXT,
        name VARCHAR(255)
    )
    """
```

图 7.7 `program_input` 输入数据库程序

```
#将data中的output.png和txt存入数据库:
with connection.cursor() as cursor:
    # 创建表
    create_table_query = """
    CREATE TABLE IF NOT EXISTS program_output (
        program_begin_time TEXT,
        program_end_time TEXT,
        filename TEXT,
        content LONGTEXT,
        name VARCHAR(255)
    )
    """
```

图 7.8 `program_output` 输出数据库程序

`Program_input` 表单有以下字段：

- 1 `program_setting_time`：存储程序设置时间的文本字段。
- 2 `data`：存储数据的文本字段。。

`program_output` 表单有以下字段：

- 1 `program_begin_time`：存储程序开始时间的文本字段。
- 2 `program_end_time`：存储程序结束时间的文本字段。
- 3 `filename`：存储文件名的文本字段。
- 4 `content`：存储文件内容的长文本字段。

program_setting_time	data	name
2024-05-24 20:07:48.2996	1500000000.0	"dist" plug_matlab
2024-05-24 20:18:15.5492	1500000000.0	"dist" plug_matlab
2024-05-24 21:15:29.2334	1500000000.0	"dist" plug_matlab
2024-05-24 21:17:38.9427	1500000000.0	"dist" plug_matlab
2024-05-24 21:20:58.2120	1500000000.0	"dist" plug_matlab
2024-05-24 21:24:27.5544	1500000000.0	"dist" plug_matlab
2024-05-24 21:33:59.0091	1500000000.0	"dist" plug_matlab

图 7.9 数据库表单内容

通过 7.4 利用 Docker 实现微服务架构的 Send.py 的介绍，将输入参数和输出数据存储于数据库，利用 Base64 进行解码，接下来利用配套内容加以验证：

program_setting_time	data	name
2024-05-24 20:07:48.2996	1500000000.0	"dist" plug_matlab
2024-05-24 20:18:15.5492	1500000000.0	"dist" plug_matlab
2024-05-24 21:15:29.2334	1500000000.0	"dist" plug_matlab
2024-05-24 21:17:38.9427	1500000000.0	"dist" plug_matlab
2024-05-24 21:20:58.2120	1500000000.0	"dist" plug_matlab
2024-05-24 21:24:27.5544	1500000000.0	"dist" plug_matlab
2024-05-24 21:33:59.0091	1500000000.0	"dist" plug_matlab

图 7.10 MySQL 中输入参数

program_begin_time	program_end_time	filename	content	name
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	Begin.py	ZGVmIG1ha	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	data.json	ew0KICAgIC	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	decode.py	lyAvL+eUqC	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	Dockerfile	RIJPTSbweXl	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	output.txt		plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	PL_free.m	JeacrOWHve	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	README.md	lyA2Ry1TZW	plug_matlab
2024-05-24 20:07:48.2996	2024-05-24 20:07:48.48	Send.py	ZnJvbSBmbC	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	Begin.py	ZGVmIG1ha	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	data.json	ew0KICAgIC	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	decode.py	lyAvL+eUqC	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	Dockerfile	RIJPTSbweXl	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	PL_free.m	JeacrOWHve	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	README.md	lyA2Ry1TZW	plug_matlab
2024-05-24 21:20:58.2120	2024-05-24 21:21:09.59	Send.py	ZnJvbSBmbC	plug_matlab
2024-05-24 21:24:27.5544	2024-05-24 21:24:40.35	output.txt	MzUuOTYzN	plug_matlab
2024-05-24 21:33:59.0091	2024-05-24 21:34:10.61	output.txt	MzUuOTYzN	plug_matlab
2024-05-26 10:30:40.3745	2024-05-26 10:34:51.02	output.txt	MzUuOTYzN	plug_matlab

图 7.11 MySQL 中输出参数

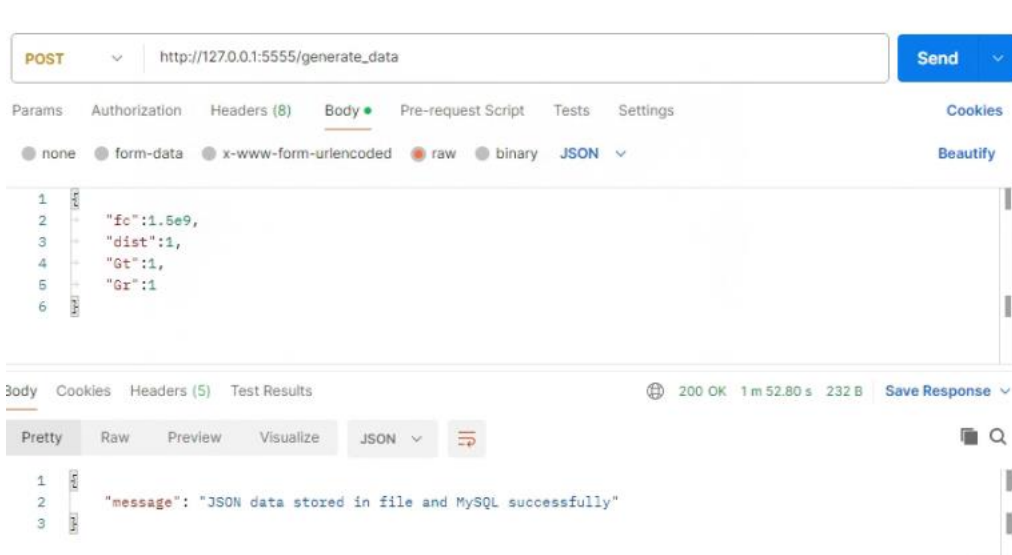


图 7.12 发送 POST 请求

通过图 7.12 发送 POST 请求，得到图 7.10 和图 7.11 中的参数，可以验证数据内容可以通过持续的 POST 请求进行上传和更新。这种持续更新可以帮助优化新的结果，并将它们存储到平台中。随着数据的不断更新，可以进行可视化展示，并验证按需服务的效果。

至此，微服务架构的算法上传 Matlab 模块已经完成。

7.4 利用 Docker 实现微服务架构

完成以上请先联系课题 5 检查是否正确，正确后修改平台数据库后再操作以下内容。

首先确保你本地环境已经拥有一个 Docker，在根目录下写一个 DockerFile 编译整个程序。**并且，检查所有的函数，不得使用绝对路径！**

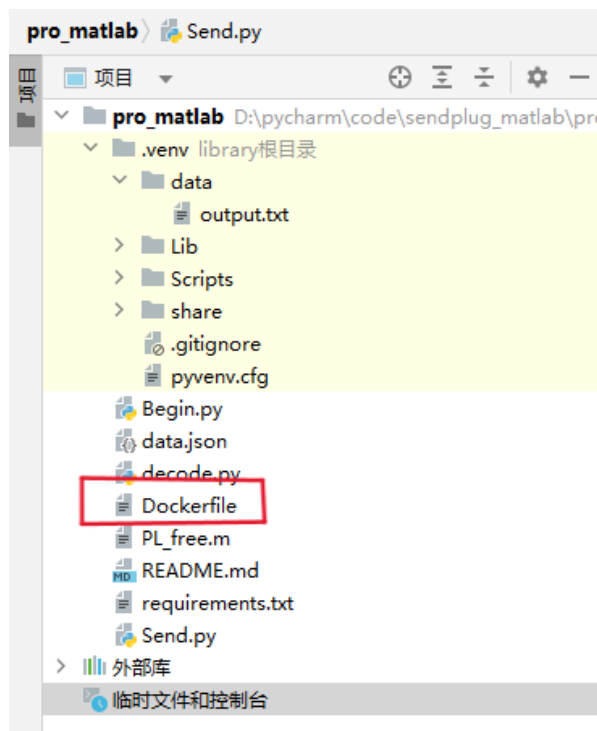


图 7.13 DockerFile 位置

编写一个 DockerFile，下面是一个简单的 DockerFile 示例：

<https://docs.docker.com/engine/reference/builder/#run>

使用官方 Python 基础镜像

FROM python:3.9

设置工作目录

WORKDIR /matlab_pro

将当前目录中的所有内容复制到工作目录中

ADD ./matlab_pro

安装依赖项

RUN pip install -r requirements.txt -I https://pypi.tuna.tsinghua.edu.cn/simple/t

暴露端口

EXPOSE 5000

定义环境变量

ENV FLASK_APP=app.py

启动应用

CMD ["python", "./send.py"]

```
FROM python:3.9
ADD . /matlab_pro
WORKDIR /matlab_pro
EXPOSE 5555
RUN pip install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
CMD ["python", "./Send.py"]
```

图 7.14 DockerFile 内容

在 cmd 中编译 Docker 镜像:

参考文档: <https://docs.docker.com/engine/reference/commandline/build/>

`docker build -t pro_matlab:v1 .`

```
D:\pycharm\code\sendplug_matlab\pro_matlab>docker build -t pro_matlab:v1 .
[+] Building 292.3s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load .dockerignore
=> [internal] load metadata for docker.io/library/python:3.9
=> [internal] load build context
=> [internal] load metadata for docker.io/library/python:3.9
[1/4] FROM docker.io/library/python:3.9@sha256:c0d0c146710fed0a6d62cb55b92f00bfbfc3b931fff6218f4953bab58333c3
=> resolve docker.io/library/python:3.9@sha256:c0d0c146710fed0a6d62cb55b92f00bfbfc3b931fff6218f4953bab58333c3
=> sha256:c0d0c146710fed0a6d62cb55b92f00bfbfc3b931fff6218f4953bab58333c3 2.60kB / 2.60kB
=> sha256:45e88e32518da8b8c551443510c7859e49938ee4d416fced6fb87f5b0a238ac1 2.22kB / 2.22kB
=> sha256:81f391f1a7d79bc72276ad4bf3ba880646124b37bb773411c7b7361507565e6 8.60kB / 8.60kB
=> sha256:9b829c73b52b92b97d5c07a54fb0f3e921995a296c714b53a32ae67d19231fcd 5.15MB / 5.15MB
=> sha256:cb567ae381722f070eca53f35823ed21baa85d61d5d95cd5a95ab53d740cdd56 10.87MB / 10.87MB
=> sha256:0c984e541cdcd809221d2173bd1db736651b95b74f32a009a0b77a6e1e3 54.92MB / 54.92MB
=> sha256:a494e4811622b31c027ccac322ca463937fd805f569a93e6f15c01aade718793 54.57MB / 54.57MB
=> sha256:6f9f74896d8a93fe0172f594fab85e0b4e8a0481a0ref9d112erc7e4d3c78f7 196.51MB / 196.51MB
=> extracting sha256:0e29546d541cd4309221d2173bd1db736651b95b74f32a009a0b77a6e1e3 2.6s
=> sha256:fc6bd5f7c986044761da91fe5f61be5b561dc398814fb15f7aa998f53023e774 6.29MB / 6.29MB
=> extracting sha256:9b829c73b52b92b97d5c07a54fb0f3e921995a296c714b53a32ae67d19231fcd 0.3s
=> sha256:ccba14545fc249e01a059ed19c9da742925e63ee83b3f25a870f2a3b6e7fec0d 234B / 234B
=> sha256:12abdd900563d02f06e465f3eff36c0923b01b7b1765c9543c4a971d476fe094 17.71MB / 17.71MB
=> extracting sha256:cb567ae381722f070eca53f35823ed21baa85d61d5d95cd5a95ab53d740cdd56 0.4s
=> sha256:cfadf9830bc25d1742a2df5ca43f674d18f762e923e97b900127ccde903fe233 2.35MB / 2.35MB
=> extracting sha256:6494e4811622b31c027ccac322ca463937fd805f569a93e6f15c01aade718793 2.7s
=> extracting sha256:6f9f74896d8a93fe0172f594fab85e0b4e8a0481a0ref9d112erc7e4d3c78f7 4.9s
=> extracting sha256:fc6bd5f7c986044761da91fe5f61be5b561dc398814fb15f7aa998f53023e774 0.3s
=> extracting sha256:12abdd900563d02f06e465f3eff36c0923b01b7b1765c9543c4a971d476fe094 0.6s
=> extracting sha256:ccba14545fc249e01a059ed19c9da742925e63ee83b3f25a870f2a3b6e7fec0d 0.0s
=> extracting sha256:cfadf9830bc25d1742a2df5ca43f674d18f762e923e97b900127ccde903fe233 0.2s
[2/4] WORKDIR /pro_matlab
[3/4] ADD .
[4/4] RUN pip install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple/
=> exporting to image
=> exporting layers
=> writing image sha256:d0f73ec463c985d9b5828f64d0c508425ef0e5417d88c9d6fd0ebb4b6351d205
=> naming to docker.io/library/pro_matlab:v1
```

图 7.15 Docker 编译示意图

运行容器:

参考文档: <https://docs.docker.com/engine/reference/run/>

`docker run -d -p 5000:5000 --name matlab pro_matlab:v1`

-p 映射容器内端口到宿主机

--name 容器名字

-d 后台运行

```
D:\pycharm\code\sendplug_matlab\pro_matlab>docker run -d -p 5555:5555 --name matlab pro_matlab:v1
72035a1a3ccd835117212deaec5b4d37119a552da79d851e71c3745eb224465
```

图 7.16 Docker 容器启动示意图

