美团第三届
低空经济智能飞行管理挑战赛 性能赛
比赛平台使用说明

2025.08.25

目录

1. 平台使用流程概述

本届比赛平台基于 UE 与 Gazebo 构建，高保真还原地面小车搭载 RGB 相机、LiDAR 点云、里程计 的传感与运动特性。选手可通过自己的 key 提交调试任务，云端仿真环境将自动运行并返回 消息日志、状态反馈及得分数据，支持离线回放分析。

选手需基于组委会提供的 SDK 开发导航模块，接收地面小车的 RGB 图像、LiDAR 点云、里程计数据及语言指令，通过规划与控制算法解析环境信息并输出位姿/速度控制指令，驱动地面小车完成自主导航。

开发完成后，选手需将算法容器化部署并推送至腾讯云镜像仓库，通过提交工具指定镜像 URI 及任务文件发起任务。平台将按队列顺序运行选手镜像，执行导航任务并返回状态、得分及日志数据包，选手可下载日志进行回放分析与调试优化。

2. 拉取镜像

2.1 系统要求

本地推荐配置

操作系统：Ubuntu 20.04 （方便本地使用 ROS Noetic 进行调试）

CPU：8 核 主频 2.5GHz

GPU：英伟达 8GB 显存

内存：16GB

存储：100GB SSD

线上运行环境

线上环境无需选手配置，选手可根据线上环境配置选择规模合适的模型。

操作系统：Ubuntu 22.04

CPU：Intel(R) Core(TM) i9-14900K

GPU：NVIDIA GeForce RTX 4090

GPU 驱动版本：570.133.07

CUDA 版本：12.8

内存：48GB

存储：500GB

2.2 环境配置

安装 Docker

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# 安装后设置非 sudo 用户也能运行 docker
sudo groupadd docker

sudo gpasswd -a ${USER} docker

sudo systemctl restart docker

sudo chmod a+rw /var/run/docker.sock
```

验证 Docker 是否安装成功。

```
docker -version
```

如输出类似下面内容则证明安装成功。

```
Docker version 28.1.1, build 4eba377
```

安装 Nvidia Container Toolkit

Nvidia Container Toolkit 给 Docker 容器内使用 GPU 提供支持。

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o
/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
```

```
  && curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list | \
    sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \
    sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo sed -i -e '/experimental/ s/^#//g' /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo apt-get update

sudo apt-get install -y nvidia-container-toolkit

sudo nvidia-ctk runtime configure --runtime=docker

sudo systemctl restart docker
```

安装完成后可先进行 **2.3 镜像拉取**，使用 SDK 镜像验证 **Nvidia Container Toolkit** 是否安装成功。

```
docker run --gpus all uav-challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge:sdk nvidia-smi
```

如输出下面内容则证明安装成功。

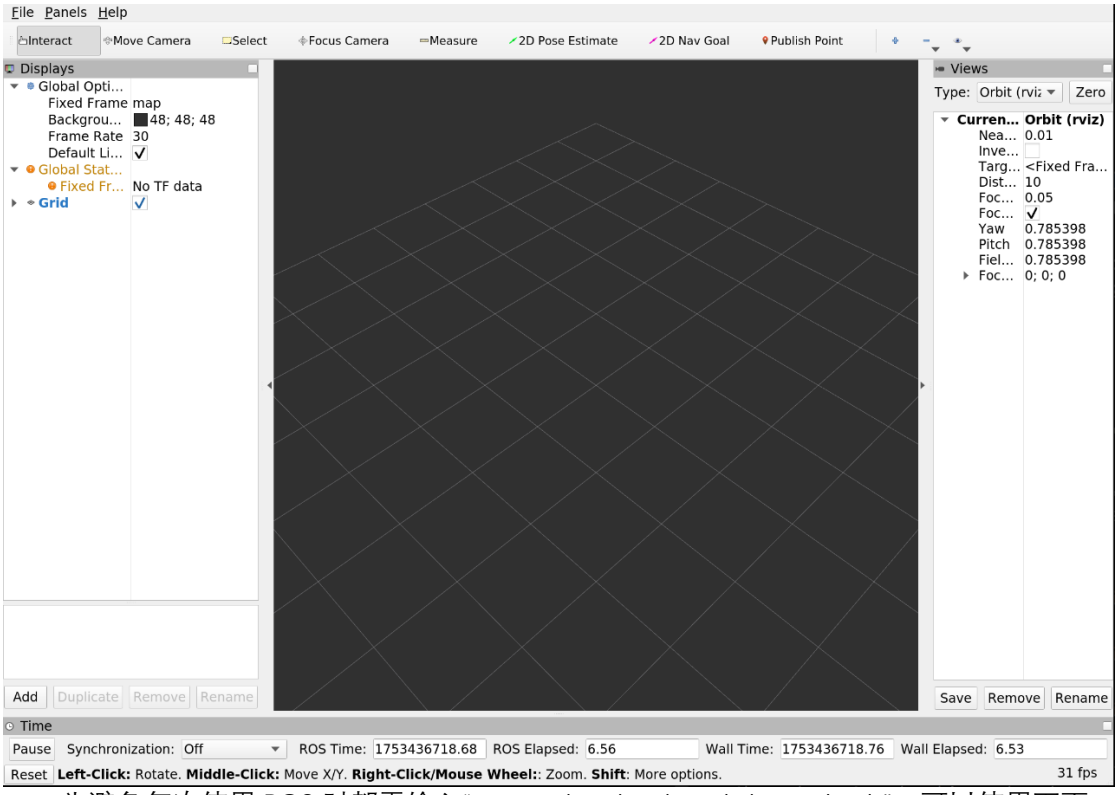
```
Thu Jul 31 02:56:17 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.247.01             Driver Version: 535.247.01   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA GeForce RTX 4080        Off | 00000000:01:00.0  On |                  N/A |
| 30%   28C    P8               9W / 320W |    531MiB / 16376MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

安装 ROS Noetic

按照官方说明 http://wiki.ros.org/noetic/Installation/Ubuntu 安装好 ros-noetic-desktop-full 版本，安装完成后可运行 rviz 验证是否安装成功。

```
source /opt/ros/noetic/setup.bash
roscore
## 新打开一个终端运行
source /opt/ros/noetic/setup.bash
rviz
```

如出现 rviz 的图形界面则证明安装成功。



为避免每次使用 ROS 时都需输入“source /opt/ros/noetic/setup.bash”，可以使用下面语句将“source /opt/ros/noetic/setup.bash”加入~/.bashrc 文件。

```
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
## 使修改后的~/.bashrc 文件生效
source ~/.bashrc
```

2.3 镜像拉取

登陆到腾讯云 Docker 服务

```
docker login uav-challenge.tencentcloudcr.com --username 'tcr$2025' --password
TAUAHrX8DU4ZXbQPbpC2w7H1SbQltDPc
```

拉取 SDK 镜像

```
docker pull uav-challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge:sdk
```

查看镜像

```
docker images
```

拉取成功输出

```
REPOSITORY                                                        TAG       IMAGE ID       CREATED      SIZE
uav-challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge sdk       bfaa3dbf8a0c   5 days ago   4.1GB
```

运行镜像

```
docker run -d --gpus all --name race_user_sdk_container uav-
challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge:sdk tail -f /dev/null
```

该命令会使镜像在后台运行且不会退出，以方便后续打包提交

进入容器

```
docker exec -it race_user_sdk_container bash
```

上传到云端的镜像会自动运行入口脚本/run.sh，SDK 镜像中 run.sh 的运行代码为：

```
python3 /catkin_ws/src/vln_mock/scripts/vln_demo_mock.py
```

vln_demo_mock.py 已包含接口样例，详情会在下一节进行说明

选手在熟悉完整的提交流程后，可自行修改入口脚本内容

3. 应用开发

3.1 协议与接口

3.1.1 地面小车配置

地面小车的机身是一个半径为 0.6m，高 0.63m 的圆柱体，系统会以此为基准进行碰撞检测。

控制器的各项限制如下：

速度	加速度	角速度	角加速度
1.5m/s	3m/s^2	6.28rad/s	6.28rad/s^2

车体坐标系定义

地面小车的车体坐标系的原点定义为机身圆柱体中心点在地面上的投影，xyz 三轴方向均与世界坐标系相同。

RGB 相机位姿

地面小车搭载的 RGB 相机在车体坐标系下的位置为[0.5 ,-0.04 ,0.57]，姿态为[0.000, 0.314, 0.000](分别代表滚转角 ROLL，PITCH，YAW，弧度制）。

雷达位姿

地面小车搭载的雷达型号为 Livox MID360，在车体坐标系下的位置为[-0.011, 0.023, 0.480]，姿态为[3.14, 0.000, 0.000]（分别代表滚转角 ROLL，PITCH，YAW，弧度制）。

3.1.2 数据接口

地面小车对选手开放的数据包含语言指令、RGB 图像、里程计和雷达点云数据，这些数据的详细说明见下表。其中，语言指令（包括测试过程中选手的自定义内容和比赛的官方语言指令）统一通过这个接口发送，该消息会暂存在 ROS 中，只需要选手在代码中订阅这个消息就会收到（仅一次）。请注意，雷达点云数据为 Livox MID360 雷达的原始数据（雷达坐标系）。

数据类型	**消息接口**	**消息定义**	**频率**
语言指令	/instruction	String	发送一次
RGB 图像	/magv/camera/image_compressed/compressed	sensor_msgs/CompressedImage	10Hz
里程计	/magv/odometry/gt	nav_msgs/Odometry	25Hz
雷达点云	/magv/scan/3d	sensor_msgs/PointCloud2	10Hz
TF 转换矩阵	/tf /tf_static	tf2_msgs/TFMessage	/tf:50Hz /tf_static: 发送一次

3.1.3 控制接口

地面小车包含 2 种控制接口，选手根据自己的习惯任选其一即可。请注意，混合使用可能导致车辆不可控。

位姿控制

允许选手发送一个车体坐标系下 xy 平面的目标位置和偏航角，接收到目标位姿后，地面小车会直线运动到目标位置，并在运动过程中调整自身姿态。

消息接口	/magv/planning/pos_cmd
消息类型	/magv_msgs/PositionCommand
消息定义	geometry_msgs/Point position geometry_msgs/Vector3 velocity

	float64 yaw float64 yaw_dot	
简单示例	cmd.position.x:1.0 cmd.postion.y:0.0 cmd.yaw:0.0	该示例为地面小车行驶至(1.0, 0.0)坐标点，并在目标点保持 0 度偏航角（弧度制）
	cmd.position.x:0.0 cmd.postion.y:0.0 cmd.yaw:1.57	该示例为地面小车行驶至(0.0, 0.0)坐标点，并在目标点保持 90 度偏航角（弧度制）

速度控制

地面小车自身可支持直接接收速度消息，消息说明如下

消息接口	/magv/omni_drive_controller/cmd_vel	
消息类型	/geometry_msgs/Twist	
简单示例	linear.x=1.0 linear.y=2.0 angular.z=0.3	上述示例为在车体坐标系下控制地面小车 x 轴方向速度为 1.0m/s，y 轴方向速度为 2.0m/s，绕 z 轴角速度为 0.3 rad/s（逆时针为正）

地面小车自带对速度控制接口的检测功能，若 2s 内没有收到速度消息，小车会自动停下。

3.1.4.任务状态接口

选手到达任务位置后，需向/status 接口发送消息：

消息接口	/status	
消息类型	std_msgs/int32	
简单示例	0 或 1	该消息只有在结束任务时才需要发送，在该消息接口发送任何消息都会导致当前任务结束。发送 0 代表选手认为当前任务成功完成；发送 1 代表选手认为当前任务无法完成，主动放弃。

3.2 示例代码

本节所有的示例代码均会包含在 demo 镜像的 /catkin_ws/src/vln_mock/scripts/vln_demo_mock.py 文件中，该文件是一个完整的、可运行的、仅用于演示各接口调用方式的 demo，选手可在镜像中自行查看。

3.2.1 创建 ROS NodeHandler 示例

```
    def __init__(self):
        # 初始化节点
        rospy.init_node('vln_demo_mock', anonymous=True)

        # 创建发布者，发布到/status 话题
        self.status_pub = rospy.Publisher('/status', Int32, queue_size=10)
        self.target_pub_discrete = rospy.Publisher('/magv/planning/pos_cmd',
PositionCommand, queue_size=10)
        self.target_pub_continuous =
rospy.Publisher('/magv/omni_drive_controller/cmd_vel', Twist, queue_size=10)

        # 创建订阅者，订阅各类话题
        rospy.Subscriber('/instruction', String, self.instruction_callback)
        rospy.Subscriber('/magv/odometry/gt', Odometry, self.odometry_callback)
        rospy.Subscriber('/magv/scan/3d', PointCloud2, self.pointcloud_callback)
```

```
        rospy.Subscriber('/magv/camera/image_compressed/compressed', CompressedImage,
self.image_callback)

        rospy.loginfo("VLN Demo Mock node started, waiting for instruction...")
```

3.2.2 消息接收示例

消息接收示例

语言指令消息

```
    def instruction_callback(self, msg):
        """
        Callback function for receiving instruction messages
        """
```

里程计消息

```
    def odometry_callback(self, msg):
        """
        Callback function for receiving odometry messages
        """
        #rospy.loginfo("Received odometry: %s", msg)
        #magv 当前的位置信息保存在 msg.pose.pose 下（注意有两层 pose）
        self.odom_x = msg.pose.pose.position.x
        self.odom_y = msg.pose.pose.position.y
        # 提取四元数
        orientation_q = msg.pose.pose.orientation
        quaternion = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
        # 转换为欧拉角
        (roll, pitch, yaw) = tf.transformations.euler_from_quaternion(quaternion)
        self.odom_yaw = yaw  # 提取偏航角

        #magv 当前的速度信息保存在 msg.twist.twist 下（注意有两层 twist）
        self.vel_x = msg.twist.twist.linear.x#世界坐标系 X 轴方向速度
        self.vel_y = msg.twist.twist.linear.y#世界坐标系 Y 轴方向速度
        self.vel_yaw = msg.twist.twist.angular.z#世界坐标系偏航角速度 s
```

雷达点云消息

```
    #这里是一个输出全部点云数据的示例，仅作参考
    #这里使用 senser_msgs 解析和操作 PointCloud2 数据的工具库，选手也可以自行选用其他库，如
ros_numpy
    def pointcloud_callback(self, msg):
        """
        Callback function for receiving pointcloud messages
        """
        # 读取点云中的点，并滤除异常值
        points = pc2.read_points(msg, field_names=("x", "y", "z"), skip_nans=True)
        # 计算点云数量
        pc_number = len(list(points))
        #rospy.loginfo("Received pointcloud of number: %d", pc_number)
        # 打印每个点
        for point in points:
            x, y, z = point
            # rospy.loginfo("Point: x=%f, y=%f, z=%f", x, y, z)
```

RGB 图像消息

```
    #这里是一个输出接收图像的示例，仅作参考
    #这里使用 numpy 和 Pillow 库处理压缩图像
    def image_callback(self, msg):
        """
        Callback function for receiving compressed image messages using Pillow
        """
        # 1. 用 Pillow 解码为 PIL.Image 对象
        image = Image.open(io.BytesIO(msg.data))
        # 2. 转为 numpy 数组（格式为 HWC，RGB）
        np_image = np.array(image)
```

```
        #rospy.loginfo("Received image of shape: %s", np_image.shape)
        # 例如：显示图片（调试时用，Pillow 自带 show 方法）
        # image.show()
```

3.2.3 消息发送示例

demo 示例中的位姿控制和速度控制混用仅作为用法展示，请勿在实际使用中将两者混用，只能选择一种作为控制模式。

位姿控制指令

```
    # 该示例发送目标消息，向车体坐标系 x 轴正方向移动 1m
    def example_moving_forward(self):
        #构建目标消息
        goal1 = PositionCommand()
        goal1.position.x = self.odom_x + 1.0 * math.cos(self.odom_yaw)  # 目标 x 坐标
        goal1.position.y = self.odom_y + 1.0 * math.sin(self.odom_yaw) # 目标 y 坐标
        goal1.position.z = 0.0
        goal1.velocity.x = 0.0
        goal1.velocity.y = 0.0
        goal1.velocity.z = 0.0
        goal1.yaw = self.odom_yaw + 0.0  #目标朝向
        goal1.yaw_dot = 0.0

        rospy.loginfo("Sending goal: x=%.2f, y=%.2f, theta=%.2f",
                      goal1.position.x, goal1.position.y, goal1.yaw)

        self.target_pub_discrete.publish(goal1)  # 发送目标消息

    # 该示例发送目标消息，偏航角向左转 90 度
    def example_turn_left(self):
        #构建目标消息
        goal2 = PositionCommand()
        goal2.position.x = self.odom_x  # 目标 x 坐标
        goal2.position.y = self.odom_y  # 目标 y 坐标
        goal2.position.z = 0.0
        goal2.velocity.x = 0.0
        goal2.velocity.y = 0.0
        goal2.velocity.z = 0.0
        goal2.yaw = self.odom_yaw + math.pi / 2  #目标朝向
        goal2.yaw_dot = 0.0

        rospy.loginfo("Sending goal: x=%.2f, y=%.2f, theta=%.2f",
                      goal2.position.x, goal2.position.y, goal2.yaw)

        self.target_pub_discrete.publish(goal2)  # 发送目标消息
```

速度控制指令

```
    # 该示例发送目标消息，设置 magv 速度为车体坐标系 x 方向 1m/s，由于控制器设定，实际会以该速度行驶 2s，2s 之后停下>，因此若需要持续行驶，需要周期性发送目标消息（不低于 0.5Hz）
    def example_moving_forward_continuous(self):
        #构建目标消息
        goal3 = Twist()
        goal3.linear.x = 1.0  # x 轴速度
        goal3.linear.y = 0.0  # y 轴速度
        goal3.linear.z = 0.0
        goal3.angular.x = 0.0
        goal3.angular.y = 0.0
        goal3.angular.z = 0.0  # 偏航角速度

        rospy.loginfo("Sending velocity: x=%.2f, y=%.2f, theta=%.2f",
                      goal3.linear.x, goal3.linear.y, goal3.angular.z)

        self.target_pub_continuous.publish(goal3)  # 发送目标消息

    # 该示例发送目标消息，设置 magv 速度为偏航角速度为 0.5rad/s，由于控制器设定，实际会以该速度行驶 2s，2s 之后停下
```

```
，因此若需要持续行驶，需要周期性发送目标消息（不低于 0.5Hz）
    def example_turn_left_continuous(self):
        #构建目标消息
        goal4 = Twist()
        goal4.linear.x = 0.0  # x 轴速度
        goal4.linear.y = 0.0  # y 轴速度
        goal4.linear.z = 0.0
        goal4.angular.x = 0.0
        goal4.angular.y = 0.0
        goal4.angular.z = 0.5  # 偏航角速度

        rospy.loginfo("Sending velocity: x=%.2f, y=%.2f, theta=%.2f",
                      goal4.linear.x, goal4.linear.y, goal4.angular.z)

        self.target_pub_continuous.publish(goal4)  # 发送目标消息
```

3.3 Base demo

选手可基于组委会提供的 SDK，熟悉后续的任务提交流程。

组委会同时也提供了一个完整的基础实现，可供选手参考，设计文档参见：性能赛 样例设计（可在大赛官网 https://uav-challenge.meituan.com/下载浏览）。

样例镜像拉取：

```
docker pull uav-challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge:demo
```

4. 任务提交

4.1 提交说明

4.1.1 Docker 镜像提交

镜像打包

确保 race_user_sdk_container 正在运行。

```
docker ps
```

输出：

```
CONTAINER ID   IMAGE                                                                COMMAND                  CREATED      STATUS      PORTS     NAMES
9af8224d0702   uav-challenge.tencentcloudcr.com/uav_challenge_2025/uav_challenge:sdk   "/ros_entrypoint.sh …"   3 days ago   Up 3 days             race_user_sdk_container
```

执行打包命令

```
# commit 后依次为容器名，镜像名，镜像 tag
# 建议使用 race_user，镜像 tag 选手可自定义
docker commit race_user_sdk_container race_user:xxxx
```

登陆腾讯云 Docker 服务

详情参见 2.3 镜像拉取。

提交镜像到 Docker Hub

```
# 给本地镜像打上新标签
# docker tag <源镜像>:<源标签> <目标仓库>/<命名空间>/<镜像名>:<目标标签>
docker tag race_user:xxxx uav-challenge.tencentcloudcr.com/uav_challenge_2025/appkey:tag

# 提交镜像
docker push uav-challenge.tencentcloudcr.com/uav_challenge_2025/appkey:tag
```

其中 appkey 需要从大赛官方通知邮件中获取，**请各个团队妥善保存 appkey（不可共享）**，tag 选手可自定义。

镜像成功上传到远程仓库后，选手需要使用提交工具提交自己的任务。

注意：镜像上传成功之后，方可使用提交工具进行任务提交。

4.1.2 任务提交

组委会为选手提供了任务提交与结果查询的工具和使用脚本 submit.sh，选手首先需要下载 submit.sh。

```
# 下载 submit.sh
wget https://s3plus.meituan.net/udss-sim-data/submit.sh
chmod +x submit.sh
```

用邮件中获取的 appkey 和 appsecret 替换 submit.sh 中的 app-key 和 app-secret。

在线调试阶段，选手需要创建一个任务配置文件，如 test.json，将下面的任务配置信息复制到 test.json。

```
{
        "scene_id": "Level01",
        "reference_length": 9,
        "timeout":100.0,
        "start_pose": [-4.50, 5.50, 0.0],
        "goal": [6.5, 5.5],
        "instruction": "move forward and stop at the tree"
}
```

任务字段说明:

字段	说明
scene_id	场景名称
reference_length	参考路径长度，用于计算得分
timeout	最长运行时间，大于该时间会处罚并强制退出，且不得分（该参数上限为 300s，设置大于 300s 会强制修正为 300s）
start_pose	移动地面小车的起始位置
goal	marker 板的摆放位置
instruction	自然语言指令，比赛使用英文用例

调用提交脚本，传入 docker uri，任务配置文件路径和任务类型

任务类型：1 代表在线竞赛，2 代表在线调试，若选手未指定则默认为 2。

```
# 提交任务请求示例
./submit.sh submit uav-challenge.tencentcloudcr.com/uav_challenge_2025/appkey:tag
test.json 2
```

注意：在线竞赛云端会运行指定的用例集合，无需任务参数文件，但提交命令仍需要传入一个空文件地址。

成功创建任务返回示例

```
[IMPORTANT] submited docker images, appkey: appkey, image name: uav-
challenge.tencentcloudcr.com/uav_challenge_2025/appkey, image tag: latest, taskID: 4955
```

包含 appkey、镜像信息、**taskID（全局唯一）**，请记住这个 taskID，会用于后续任务结果查询。

注意：

当已经提交过任务，在系统中排队时（WAITING 状态），新提交任务会替换排队的任务的镜像和参数

当有任务正在运行时（RUNNING 状态），新提交任务会失败

4.2 场景与用例设置

可以查看性能赛 性能赛 场景与用例说明（可在大赛官网 https://uav-challenge.meituan.com/下载浏览），里面包括了本次赛事提供的所有测试用例及其对应的 json 文件，选手通过提交不同的 json 文件，可以测试不同的 case。

如果选手觉得测试用例数量不够，可以自行定义测试用例，可以在用例库中选择基础场景（基础场景一旦选择，场景中物体的摆放将不能改变）选手可以通过调整移动地面小车的起始位姿、marker 板的摆放位置，以及自然语言指令等，构建不同的测试用例，测试自己算法的泛化性。

4.3 结果查看

结果查看仍然使用任务提交的脚本，如未下载，详见 4.1.2 任务提交。

调用提交脚本，标签为：query，传入创建任务时返回的全局唯一的 taskID

```
# 查询结果请求示例
./submit.sh query 4955
```

成功查询返回示例

```
[QUERY] user: appkey, task: 4955, docker: uav-
challenge.tencentcloudcr.com/uav_challenge_2025/appkey:latest, status: OVER, result:
{"score":0.0,"dataAddress":"https://s3plus.sankuai.com/udss-sim-
data/result_25/4955/result.tar"}
```

包含用户、taskID、镜像信息、任务状态和任务结果。

任务结果 result 是一个 JSON 结构，其中会包含这个任务的得分 score 和任务具体数据的地址 dataAddress，选手可以根据数据地址来下载任务执行过程的具体数据。

查询得到的任务状态包括:

WAITING：等待执行。选手提交任务后，任务会进入等待队列，此时任务状态为 WAITING，在任务处于 WAITING 状态期间，选手再次提交任务，新的任务会覆盖掉之前的任务，在队列中的位置不发生变化。

RUNNING：运行状态。当系统存在空闲资源可供任务运行时，系统会运行最先进入等待队列的任务，此时这个任务状态转变为 RUNNING。

OVER：结束状态。处于 RUNNING 状态的任务成功完成后，就会转为 OVER 状态。

ERROR：运行异常。当选手提交的镜像有问题，导致任务无法正常运行时，任务结束，转为 ERROR 状态。

4.4 最佳实践

Docker 镜像文件是一个按层级去构建的文件，无论用户在容器中进行了多少操作，在使用 docker commit 后构建的镜像，都是在原有镜像中增加了一层 Layer，docker hub 中已有的 layer 不会重复上传。

建议将静态内容（如依赖、模型文件等）放在前面，将动态内容（如代码、配置等）放在后面。选手在多次提交中如果没有修改模型，仅会增量上传动态内容的部分。

选手在确定好要使用的大模型后，使用 docker commit 导出一个能够运行大模型的 Docker 镜像。

```
docker commit race_user_sdk_container race_user:qwen2.5_7b
```

同时后续开发所需使用的容器都基于该镜像。

```
docker run -d --name race_user_sdk_container_v1 race_user:qwen2.5_7b tail -f /dev/null
```

开发完成后将新的容器导出并提交，可避免在多次提交中大模型层反复上传。

```
docker commit race_user_sdk_container_v1 race_user:qwen2.5_7b_v1
```

可以将容器中不必要的文件（如日志、缓存等）删除后再 commit 镜像，这样可进一步减少镜像的大小。

5. 数据回放

5.1 数据下载

选手使用脚本查询任务结果时，返回值包含 result 结构如下。

```
{
    "score": 0.0,
    "dataAddress": "https://s3plus.sankuai.com/udss-sim-data/result_25/4947/result.tar"
}
```

其中 score 是该任务的得分，其中 dataAddress 字段的值为任务运行的具体数据的 S3 地址，选手可以自行下载、查看任务执行的数据。

```
# 获取任务执行数据的示例
wget https://s3plus.sankuai.com/udss-sim-data/result_25/4955/result.tar
```

下载解压 tar 包会获得每一个 case 的具体结果：包括选手保存数据 user_log 目录、case 得分描述 task_0.json、case 参数 task.json。

```
├── case0_id #case 对应的 id
          ├── user_log #镜像中/user_log
          ├── task_0.json #case 得分情况
          ├── task.json #case 参数
├── case1_id
          ├── user_log
          ├── task_0.json
          ├── task.json
```

```
...
└── caseN_id
        ├── user_log
        ├── task_0.json
        └── task.json
```

其中 task_0.json 中的 case_status 字段描述任务情况，可参考下表

内容	描述
CASE_SUCCESS	任务成功完成
CASE_ABORT	地面小车主动发送放弃任务
CASE_ERROR_MISS	VLN 发送完成指令时地面小车不处于目标位置
CASE_ERROR_TIMEOUT	任务超时，VLN 未发送完成指令
CASE_ERROR_COLLISION	地面小车撞车

注意：

在线竞赛数据包中只包含 case 得分描述 task_0.json，其他结果不提供

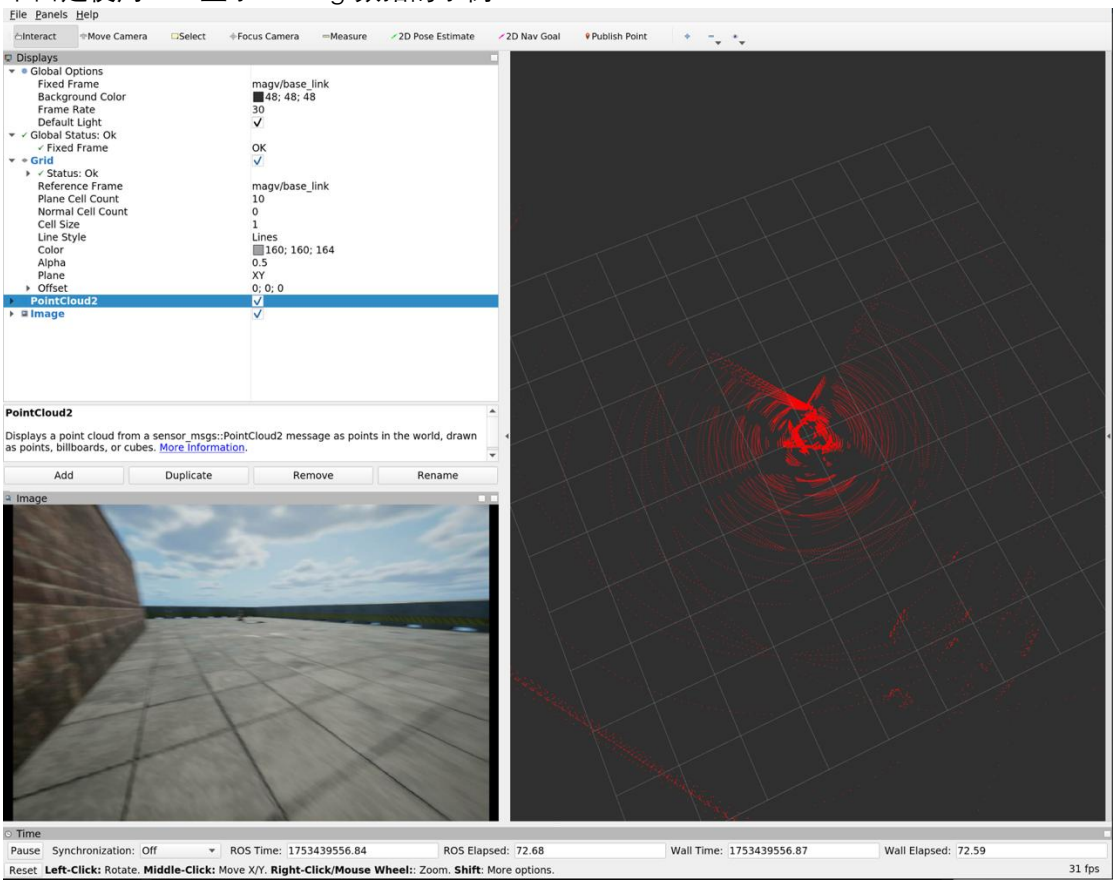
请勿放大量数据到/user_log 下，/user_log 数据量不能超过 500M

5.2 数据回放

在宿主机上使用 rosbag play 对 rosbag 数据进行回放，同时使用 rivz 等工具对数据进行可视化。

```
# 先打开一个终端运行 roscore
roscore

# 再打开一个终端运行 rviz
rviz

# 再打开一个窗口运行 rosbag play
rosbag play xxxxx.bag
```

下面是使用 rviz 显示 rosbag 数据的示例:



5.3 最佳实践

在镜像中的启动脚本/run.sh 已经预先定义好了 rosbag record 的使用命令，里面记录了一些重要话题，选手可以自行更改需要记录的话题。

强烈建议不要记录所有话题，这将导致 rosbag 的数据包巨大，选手下载数据的时间会数十倍地增加，同时也会给服务器存储造成巨大压力，可能导致数据无法提供。建议在/run.sh 中记录调试必要的数据。