apollo各模块启动指导文档

0 文档说明

本文档将以record数据包及线下课程实际传感器输出作为测试数据，指导对各个模块的启动。**各个模块的启动存在依赖关系**，启动顺序应当为：

1. **启动各个传感器部件和控制模块**：`Transform`、`lidar`、`camera`、`canbus`、`control`及`GPS`等。**注意，播放cyber包时可以跳过该步骤**。
2. **启动定位模块**：包括三种算法（基于RTK的定位，基于NDT的定位，基于MSF的定位），启动后两种需要额外制作地图。
3. **启动感知模块**：分为给予视觉感知、激光感知、雷达感知和融合感知多个部分，取决于传感器输入。
4. **启动Planning、Routing模块**：依赖于定位模块、虚拟车道线或者地图。
5. **启动预测模块**：依赖与定位模块、感知模块

- **注意**：部分模块的启动不依靠`Dreamview`的可视化界面，以方便输出各种信息以供调试。如果想不采用launch启动，可在`/apollo/modules/dreamview/conf/hmi_modes/mkz_standard_debug.pb.txt`中进行修改。

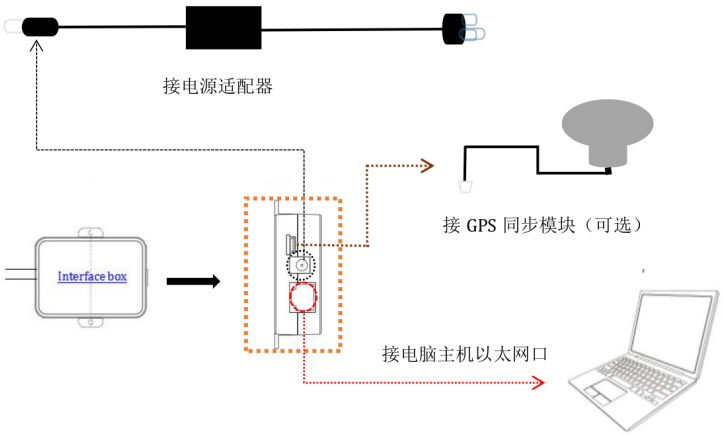
1 传感器连接与驱动配置

该模块将展示速腾16线雷达、工业相机、组合导航模块，Canbus四种驱动的运行方式。

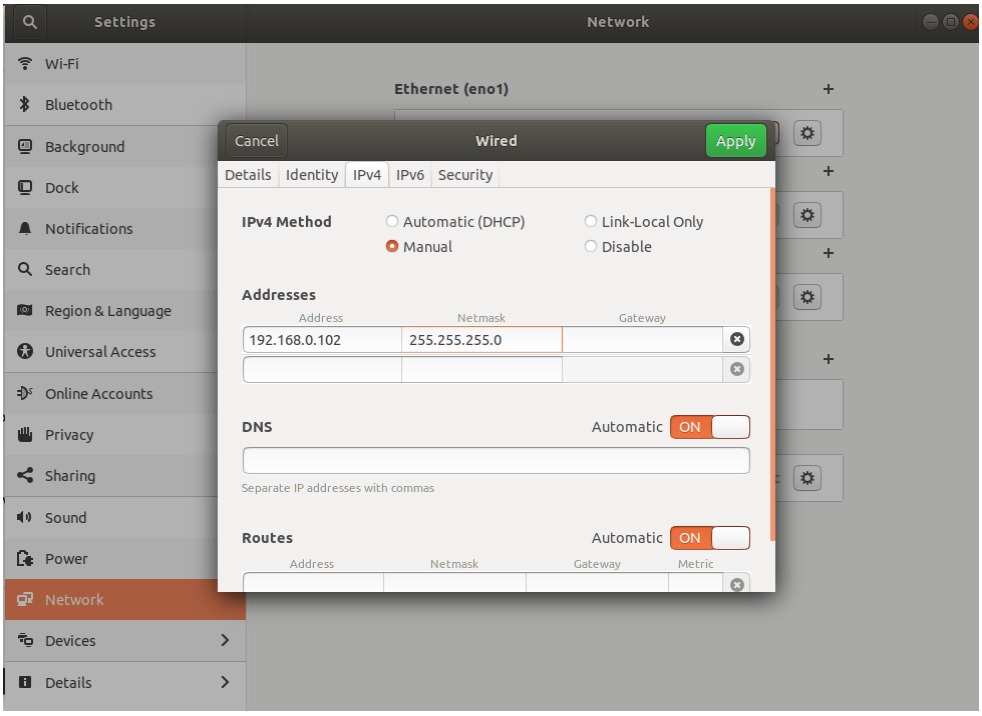
1.1 启动速腾16线激光传感器

1.1.1 驱动配置

1. *组合并连线*。电气接口详见`supplement/instruction`中材料`速腾16线.pdf`
   - **注意**：其中 GPS-激光授时部分详见`1.4 启动华测组合导航CGI-410`部分。



2. *修改本地IP和端口*。
   - 修改本地静态 IP 为`192.168.1.102`，使其保持在同一个IP端下，保持通讯。
   - 检查：查看 IP 修改是否生效时，可以新建终端，输入`ipconfig -a`，查看当前 IP 是否变化



3. （可选）*RS-LiDAR-ROS-Package安装测试*

   详见官方中文指导说明：README_CN.md

1.1.2 Apollo驱动启动

1. *修改配置文档*: `modules/drivers/lidar/conf/lidar_config.pb.txt`。注意，这里**是否启动雷达自身时钟**与时间同步问题有关，将会在后面详细说明。

```
....
# 以上略
robosense {
    model: "RS16"
    frame_id: "lidar16"
    ip: "192.168.1.200"    # 雷达默认的ip
    msop_port: 6699        # 雷达默认的两个端口号1
    difop_port: 7788       # 雷达默认的两个端口号2
    echo_mode: 1
    start_angle: 0
    end_angle: 360
    min_distance: 0
    max_distance: 200
    cut_angle: 0
    pointcloud_channel: "/apollo/sensor/lidar16/PointCloud2" # 点云通道名称
    scan_channel: "/apollo/sensor/lidar16/Scan"              # 点云通道名称
    use_lidar_clock: false # 是否使用雷达自身的时钟
}
```

2. *启动雷达驱动命令*

```
# 实际上启动的是 lidar.dag
cyber_launch start modules/drivers/lidar/launch/driver.launch
```

   - 注意：雷达选型不同导致最终的dag文件有一定差异，这里给出`lidar.dag`的全部内容以供参考

```
module_config {
    module_library : "/apollo/bazel-
bin/modules/drivers/lidar/robosense/librobosense_driver_component.so"
    components {
        class_name : "RobosenseComponent"
        config {
          name : "RS16_Driver"
          config_file_path :
"/apollo/modules/drivers/lidar/conf/rs16.pb.txt"
        }
    }
}

module_config {
    module_library : "/apollo/bazel-
bin/modules/drivers/lidar/velodyne/compensator/libvelodyne_compensator_c
omponent.so"
    components {
      class_name : "CompensatorComponent"
      config {
        name : "RS16_Compensator"
        config_file_path :
"/apollo/modules/drivers/lidar/conf/rs16_compensator.pb.txt"
        readers {channel: "/apollo/sensor/lidar16/PointCloud2"}
```

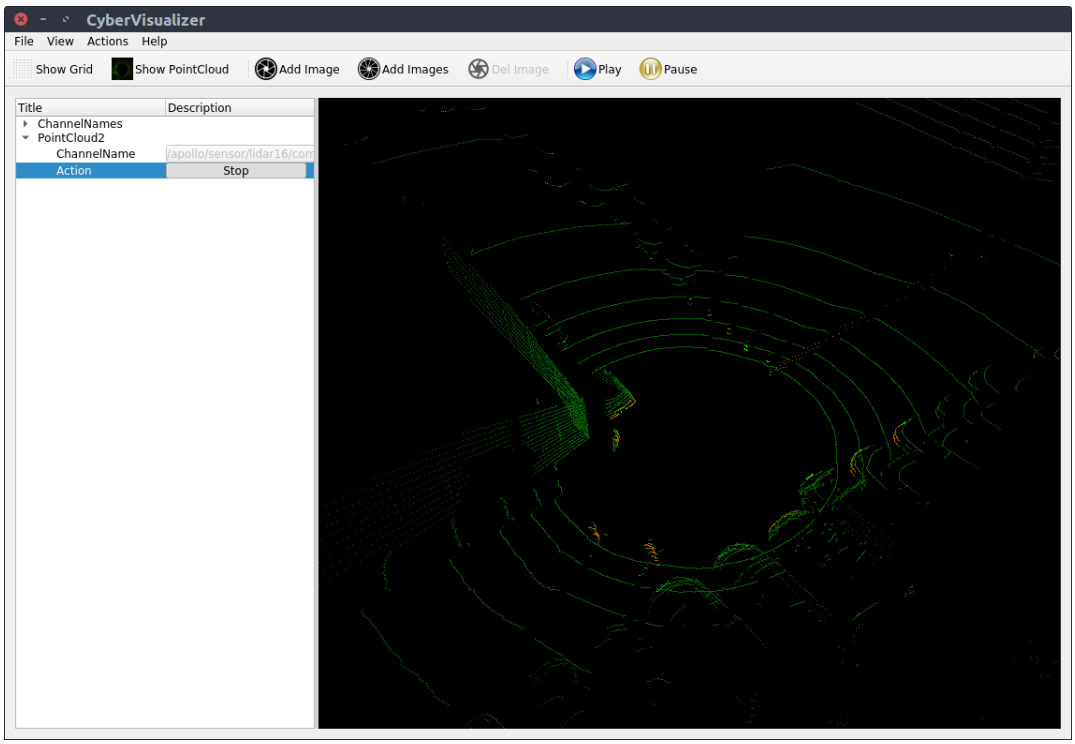
```
      }
    }
}
```

3. *验证雷达驱动是否成功*

   - 终端输入：`cyber_monitor`，查看是否存在以下三个通道：

```
/apollo/sensor/lidar16/PointCloud2                    10.00
/apollo/sensor/lidar16/Scan                           10.00
/apollo/sensor/lidar16/compensator/PointCloud2        10.00
```

   - 终端输入：`cyber_visualizer`，订阅`/apollo/sensor/lidar16/compensator/PointCloud2`话题，点击`Play`按钮后窗口出现点云成像。



1.2 启动工业相机`LI-USB3.0-AR023ZWDR CS-6mm`

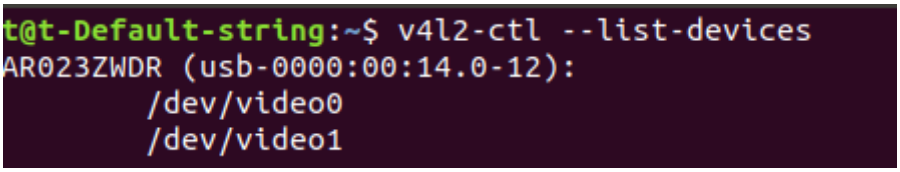
1.2.1 驱动配置

1. *记录相机接入端口*

   当相机接入我们的电脑中，在`/dev`这个目录下会显示我们接入的相机设备，作为我们访问外部设备的端口。首先确定外接相机对应的端口：

```
v4l2-ctl --list-devices
```

   出现如下界面时正确：（其中`AR023ZWDR`为自带相机，其他的可以忽略）

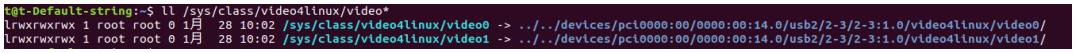


2. *建立软连接*

界面显示`AR023ZWDR`相机设备端口为`/dev/video0`和`/dev/video1`。 apollo 在程序中的设定我们的相机的端口为`/dev/camera/6mm`，因此需要通过建立规则文件配置的方法将这两者建立软连接，方法如下：

- 查看摄像头所连接的USB端口对应的端口号：

```
ll /sys/class/video4linux/video*
```



端口号为诸如`2-3:1.0`的字段，注意：后续操作每次要插入相同的USB口才能保持生效。

- 编辑规则文件：

```
sudo vim /etc/udev/rules.d/99-webcam.rules
```

添加以下内容：（注意`KERNELS`对应上一个步骤的字段）

```
SUBSYSTEM=="video4linux", SUBSYSTEMS=="usb", KERNELS=="2-3:1.0",
ATTR{index}=="0", MODE="0666", SYMLINK+="camera/front_6mm",
OWNER="apollo", GROUP="apollo"
```

- 执行如下命令，使配置的规则文件在本地系统生效：

```
bash ~/apollo/docker/setup_host/setup_host.sh  # 根据apollo位置确定对应的目录及文件
sudo reboot  # 重启工控机
```

- 开机后检查规则文件是否生效：

```
 ls /dev/camera*
```

出现`front_6mm`，我们已经将`/dev/camera/front_6mm`链接到`/dev/video0`下了。

1.2.2 Apollo驱动启动

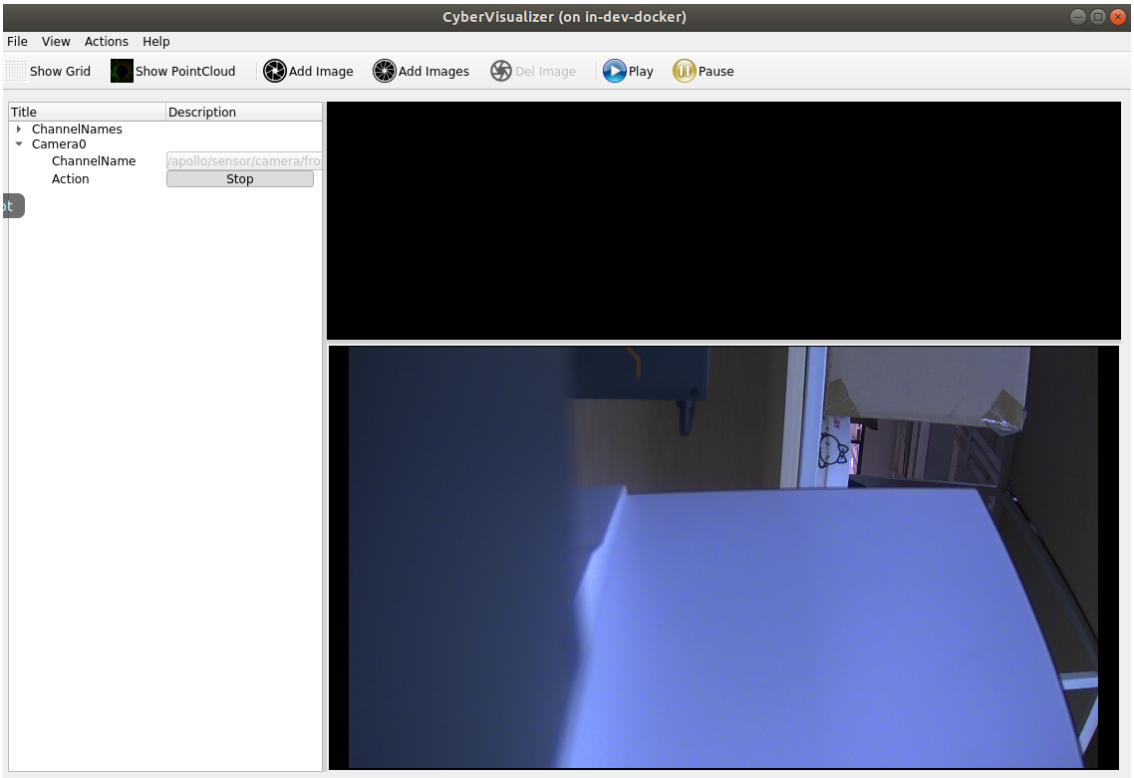
1. *启动camera驱动*

```
cyber_launch start modules/drivers/camera/launch/camera.launch
```

注意：仅使用一个相机时会出现`Cannot identify '/dev/camera/front_12mm': 2, No such file or directory`的错误，但是不影响正常使用。

2. *对Apollo输出通道进行检查*

- 启动`cyber_visualizer`，点击`Add Image`并订阅相关话题，观测效果图像或者数据信息



- 启动`cyber_monitor`：查看是否存在以下两个个通道：

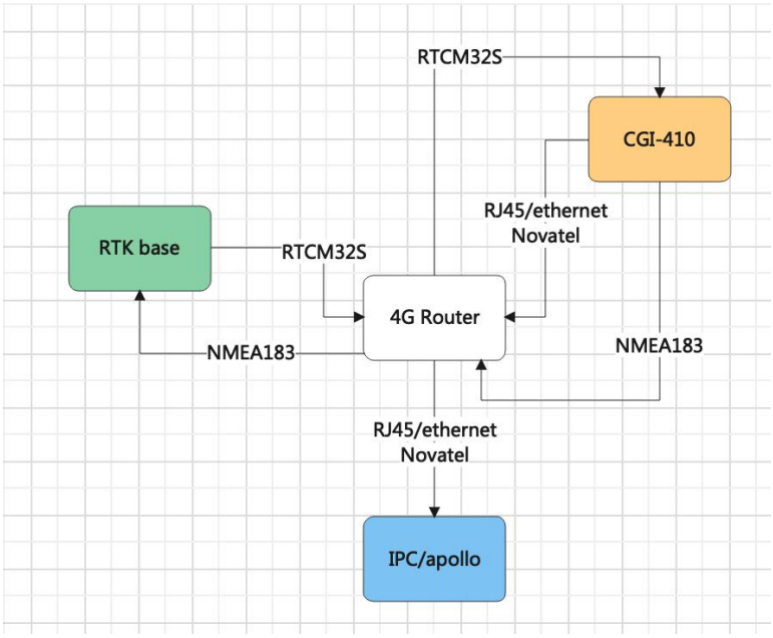
```
/apollo/sensor/camera/front_6mm/image                    15.00
/apollo/sensor/camera/front_6mm/image/compressed         15.00
```

1.3 启动华测组合导航CGI-410

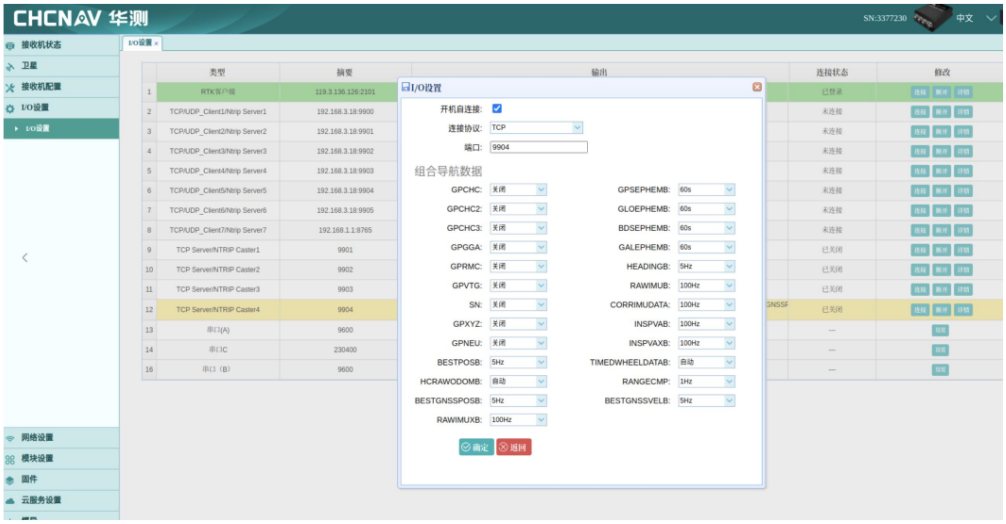
1.3.1 驱动配置

1. *拼装组合并连线*。详见指导手册：[Apollo适配CGI-410]

   - **连接组合导航系统**。
     - 连线时必要连接为：电源线（12 V，2 A，与车载相连接）、网口线（与工控机相连接）和授时线（可选，与激光相连接）；其余相关线为串口调试使用，可以不连接。
     - 连接时注意两个天线位置：主天线（又称为**定位天线**，由`GNSS1`接出）位于车辆后方；次天线（又称**定向天线**，由`GNSS2`接出）位于车辆的前方。天线进行布置和固定时要求记录杆臂值（主天线到IMU中心的距离），存放在矫正车辆的`gnss_params/ant_imu_leverarm.yaml`中。



- **配置`CGI-410`及相关输出**：
  - 插入SIM卡，打开电脑 WiFi， 搜索名为 `GNSS-XXXXXXX` 的无线网络（其中 XXXXXXX 代表你的接收器的 SN 号），然后建立连接，密码是 `12345678`； 打开浏览器，在地址栏输入 `192.168.200.1`， 弹出登陆界面， 账号： `admin`， 密码： `password`；
  - （可选）开启移动网络：点击 `WIFI 设置`， 可以开启Internet， 连接接收机 WiFi 的载体就可以使用接收机的网络进行上网， 可以关闭 Internet 以免流量用超；
  - 设置输出IP：在CGI-410的网络设置-有线网络中把组合导航 IP 地址更改为 `192.168.1.110`
  - 配置IO输出：在 `TCP Serve/NTRIP Caster4` 配置输出 Novatel 协议数据， 端口 `9904`；在 `串口A` 配置中设置波特率为9600 bps，输出协议为GPRMC，输出频率为1 Hz；本课程阶段内暂时不开通RTK账户，不影响正常使用，如果使用，则在第一行进行配置。



  - 融合数据设置：输出参考点位修正为 IMU，测量定位天线到后轮中心杆臂并填入。
- **本地端口修改**。在指导手册中建议修改静态IP为 `192.168.1.102`，但是**该IP与激光雷达端口冲突**，可以修改为 `192.168.1.103`。

2. *（可选）激光授时线连接*
   - 详见补充文件：**`Apollo补充说明`中第一章节：`时序闭环`**

1.3.2 Apollo驱动启动

1. *修改gnss配置*

   在`modules/drivers/gnss/gnss_conf.pb.txt`中，除了`data`数据结构，还应当有其他配置。

   当选择车型为`dev_kit_pix_hooke`时，还需要修改`modules/calibration/data/dev_kit_pix_hooke/gnss_conf/gnss_conf.pb.txt`。该文件内容展示如下：

```
data {
    format: NOVATEL_BINARY
    tcp {
        address: "192.168.1.110" # gnss ip
        port: 9904               # gnss 端口号
    }
}

rtk_solution_type: RTK_RECEIVER_SOLUTION
imu_type: CPT_XW5651
proj4_text: "+proj=utm +zone=50 +ellps=WGS84 +towgs84=0,0,0,0,0,0,0 +units=m
+no_defs"

tf {
    frame_id: "world"
    child_frame_id: "novatel"
}


# If given, the driver will send velocity info into novatel one time per
second
wheel_parameters: "SETWHEELPARAMETERS 100 1 1\r\n"

gpsbin_folder: "/apollo/data/gpsbin"
```

   其中关于文件地理坐标系的解析详见：**`Apollo补充说明`中的第二章节：`Apollo中的各个坐标系`**

2. *启动与监控*

   - 启动`cyber_monitor`、`GPS`模块
   - 检查如下`channel`：

```
/apollo/sensor/gnss/corrected_imu    100.00
/apollo/sensor/gnss/odometry         100.00
/apollo/sensor/gnss/best_pose         10.00
```

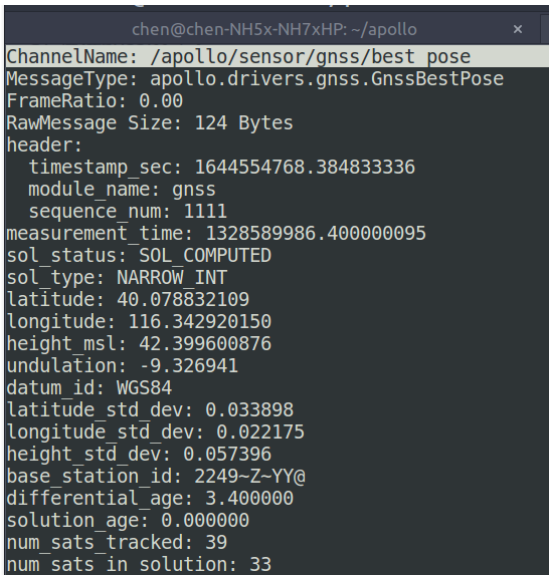
3. *验证gnss质量*:

   - `/apollo/sensor/gnss/ins_stat`中`pos_type: 56`
   - `/apollo/sensor/gnss/ins_status`中`type: GOOD`
   - `/apollo/sensor/gnss/best_pose`中`sol_status: SOL_COMPUTED`

1.3.3 GNSS系统时间说明

1. 关于`header.timestamp`与`measurement_time`：

   `measurement_time`表示gps测量的时间，`header.timestamp`表示gps信息达到系统的时间。可以看出测量时间的打头为`13xxxx`，但是系统时间却是`16xxxxx`打头，这是因为测量时间是采用了gps时间，系统时间采用了北京当地的`UTC`时间。



2. 关于`/apollo/sensor/gnss/odometry`的时间戳：

   尽管该通道下的`header.timestamp`也是`16xxxxx`开头，但是该时间戳实际上是由gps测量时间转换到北京当地`UTC`时间后得到的，因此每次运行时需要检查该时间是否与系统时间相差过大。

1.4 启动 canbus

1.4.1 驱动配置

1. 连接工控机与can口（连接can1口）
2. 启动can卡：

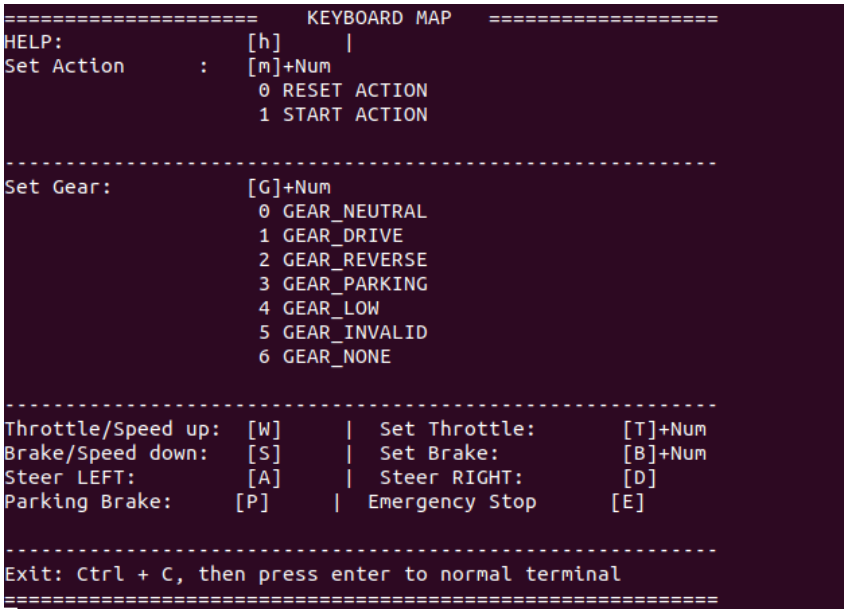
```
cd /home/EMUC_B202_SocketCAN_driver_v3.2_utility_v3.1_20210302/
sudo ./start.sh
```

1.4.2 （实车相关）控制与连接测试

1. 将车辆的四个轮子使用千斤顶抬起，**使车轮悬空**，便于观察控制情形；
2. 将遥控器的手柄切换至**自动驾驶模式**；
3. 进入Apollo系统，执行：

```
bash scripts/canbus.sh  # 启动canbus模块
bash scripts/canbus_teleop.sh # 启动键盘控制界面
```

   出现如下界面：



4. 对应的指令如下，注意此时轻按，**避免一次性加太多**：

```
m+0: 重启
m+1: 启动
g+1: 挂前进当
a: 车轮左转 # 按几次a，看看车轮是否转动
d: 车轮右转 # 按几次d，看看车轮是否转动
w: 油门增加一档 # 按几次w，看车辆是否前进
s: 刹车增加一档 # 按几次s，看车辆是否停下来
```

   - **注意**：在低速模式下，各车轮转速并不相同，这属于正常现象。

1.4.3 Apollo驱动启动

1. Apollo启动并检查通讯：
   - 执行命令：`bash /apollo/scripts/canbus.sh`
   - 检查`/apollo/canbus/chassis`和`/apollo/canbus/chassis_detail`通道是否由正常输出。
2. 注意事项：
   - `/apollo/canbus/chassis`通道中`driving_mode`表示车辆状态，当处于`EMERGENCY_MODE`时需要检查是否存在故障等问题，修复问题后重新启动程序即可。

```
ChannelName: /apollo/canbus/chassis
MessageType: apollo.canbus.Chassis
FrameRatio: 99.98
RawMessage Size: 111 Bytes
engine_started: 1
speed_mps: 0.822000
throttle_percentage: 3.700000
brake_percentage: 0.000000
steering_percentage: -3.800064
parking_brake: 0
driving_mode: EMERGENCY_MODE
error_code: NO_ERROR
gear_location: GEAR_DRIVE
header:
  timestamp_sec: 1643794701.751904249
  module_name: canbus
  sequence_num: 616192
wheel_speed:
  wheel_spd_rr: 0.838000000
  wheel_spd_rl: 0.838000000
  wheel_spd_fr: 0.806000000
  wheel_spd_fl: 0.806000000
surround:
  sonar01: 0.000000000
battery_soc_percentage: 78
```

2 启动定位模块

2.1 测试数据集与地图创建

2.1.1 本章节所使用的数据集

1. 带有定位数据的数据集：`data/bag/202211_local`
2. 不带有定位数据的数据集：`data/bag/202211_no_local`

2.1.2 车道线地图与定位地图创建

该部分详见`Apollo补充说明`中的第三章节：`Apollo中的各种地图`

2.1 基于RTK定位模块

1. 修改dag文件

   将`modules/localization/dag/dag_streaming_rtk_localization.dag`修改为：

```
# Define all coms in DAG streaming.
module_config {
    module_library : "/apollo/bazel-
bin/modules/localization/rtk/librtk_localization_component.so"
    components {
      class_name : "RTKLocalizationComponent"
      config {
        name : "rtk_localization"
        config_file_path :
"/apollo/modules/localization/conf/rtk_localization.pb.txt"
        readers: [
          {
```

```
            channel: "/apollo/sensor/gnss/odometry"
            qos_profile: {
              depth : 10
            }
            pending_queue_size: 50
          }
        ]
      }
    }
}
```

2. 启动文件

```
cyber_launch start modules/localization/launch/rtk_localization.launch
```

- **注意**：受限于法律法规等相关问题，部分数据包**不提供`/apollo/sensor/gnss/odometry`、`/apollo/sensor/gnss/ins_stat`这两个`channel`**，而直接提供`/apollo/localization/pose`数据。此时需要借助`/apollo/modules/tools/sensor_calibration/`下的两个脚本工具（本质上时py脚本，但是在Apollo 6.0后也被统一编译成了可执行文件）。

  开启两个不同终端进入docker后在/apollo根目录下分别执行：

  ```
  ./bazel-bin/module/tools/sensor_calibration/ins_stat_publisher
  ./bazel-bin/module/tools/sensor_calibration/odom_publisher
  ```

  这两个脚本便可以产生`/apollo/sensor/gnss/ins_stat`、`/apollo/sensor/gnss/odometry`这两个`channel`，之后用`cyber_recorder`工具重新生成一个数据包。如果上述任一脚本找不到，请执行`./apollo.sh build tools`来生成它们。

2.2 基于NDT定位

1. 修改源码中因为Eigen内存没对齐导致的相关错误：（`ndt_localization.h`第136行）

```
# std::list<TimeStampPose> odometry_buffer_;
std::list<TimeStampPose,Eigen::aligned_allocator<TimeStampPose>>
odometry_buffer_;
```

2. 重新编译文件：

```
bash apollo.sh build_opt localization # bash apollo.sh build
```

3. 修改配置文件：`modules/localization/conf/localization.conf`

```
# 5行
--map_dir=/apollo/modules/map/data/map_test
# 15行
--enable_lidar_localization=false
# 115行
--local_utm_zone_id=50
# 130行
--lidar_topic=/apollo/sensor/lidar16/compensator/PointCloud2
# 135行
--
lidar_extrinsics_file=/apollo/modules/calibration/data/dev_kit_pix_hooke/lida
r_params/lidar16_novatel_extrinsics.yaml
```

4. 启动ndt定位模块
   - 启动数据集，确保`/apollo/localization/pose`通道没有输出：

     ```
     cyber_recorder play -f
     data/bag/202211_no_local/20220211123701.record.000* -l
     ```

   - 启动程序，当出现`/apollo/localization/ndt_lidar`时认为成功：

     ```
     cyber_launch start modules/localization/launch/ndt_localization.launch
     ```

   - 启动可视化程序：

     ```
     cyber_launch start modules/localization/launch/msf_visualizer.launch
     ```

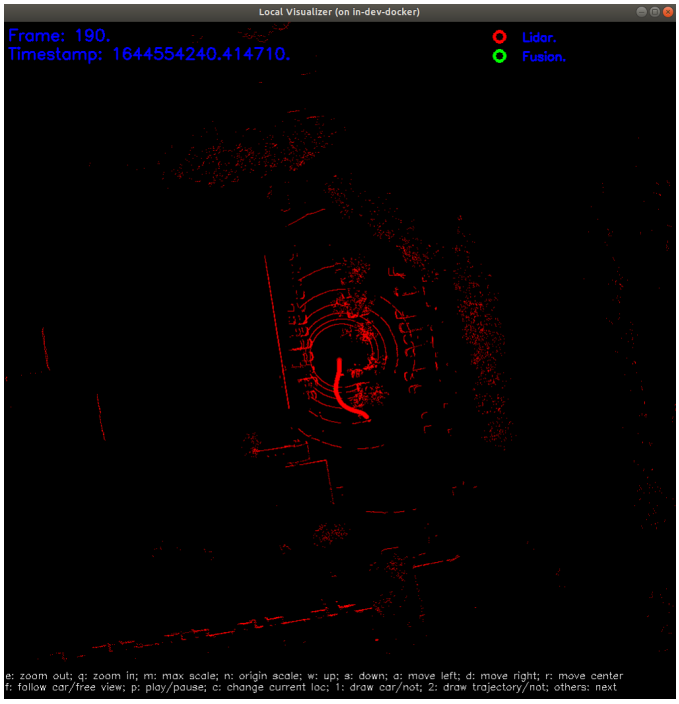
     - 关于可视化：尽管可视化程序在名称上归属于msf，但是它在**所有定位方式中**均可以使用。使用时需要确认：

       (1) 检查`dag_streaming_msf_visualizer.dag`的`channel`，**保证lidar名称正确**；

       (2) 定位方式的地图依赖于msf地图，**需要预先建立msf的`local map`地图**；

       (3) 定位策略依赖于`localization.conf`文件的配置，特别是地图所在位置，需要仔细审查。

     - **注意**：当地图不可显示时，如上图所示，删除缓存文件：`rm -rf cyber/data/map_visual`，并将`--local_map_name`设置为`local_map`后重新启动



3 启动感知、预测模块

3.1 基于激光点云的感知

1. 确保存在定位模块/数据集有下列信息正确输出：
   - `/apollo/sensor/lidar16/compensator/PointCloud2`
   - `/apollo/localization/pose`
   - `/tf`及`/tf_static`
2. 调整`/apollo/modules/perception/production/dag/`文件夹下的`dag_streaming_perception_dev_kit_lidar.dag`：

```
components {
    class_name : "RecognitionComponent"
    config {
      name: "RecognitionComponent"
      config_file_path:
"/apollo/modules/perception/production/conf/perception/lidar/recognition_conf
.pb.txt"
      readers {
          channel: "/perception/inner/SegmentationObjects"
        }
    }
}
```

3. 启动感知模块，等待待显存稳定（一般在1-2分钟左右）

```
cyber_launch start
/apollo/modules/perception/production/launch/dev_kit_perception_lidar.launch
```

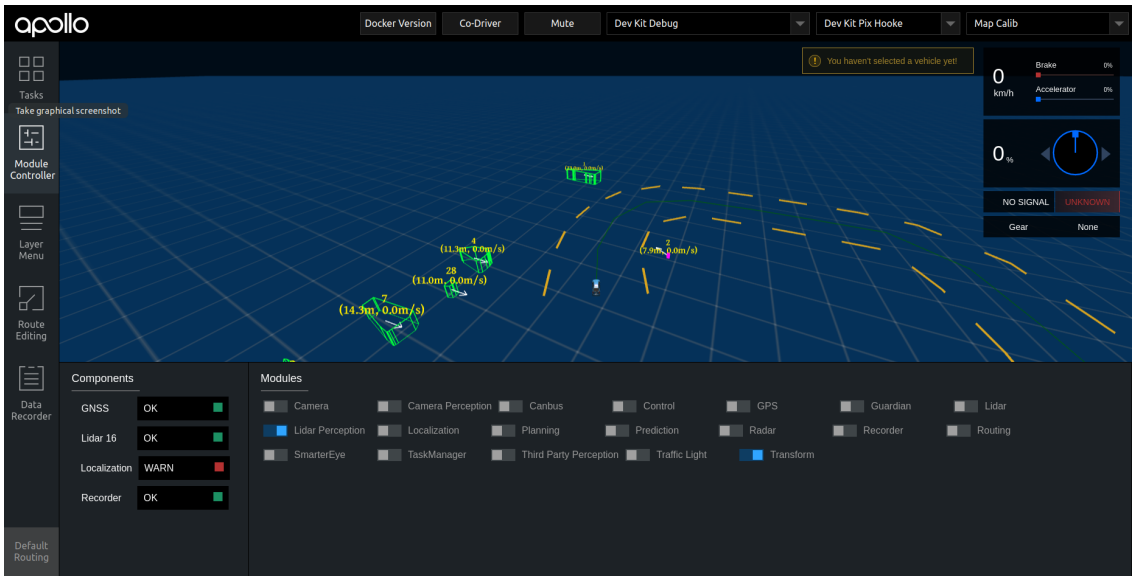
4. 待**显存稳定后**启动数据集

```
cyber_recorder play -f data/bag/20220203/20220203100813.record.0000*
```

5. 查看`/apollo/perception/obstacles`中是否由数据输出（激光检测输出为多边形）

```
cyber_monitor
```

6. 在`dreamviewer`最终效果如下：可以看出，雷达视野



3.2 基于图像的感知模块

1. 确保存在定位模块/数据集有下列信息正确输出：
   - `/apollo/sensor/lidar16/compensator/PointCloud2`
   - `/apollo/localization/pose`
   - `/tf`及`/tf_static`
2. 修改配置文件，确保检测到的障碍物信息向指定channel输出：
   - 修改文件位置为：`modules/perception/production/conf/perception/camera/fusion_camera_detection_component.pb.txt`
   - 修改文件内容为：

     ```
     output_final_obstacles : true
     output_obstacles_channel_name : "/apollo/perception/obstacles"
     ```

3. 启动图像模块，等待待显存稳定（一般在1-2分钟左右）。

```
cyber_launch start
/apollo/modules/perception/production/launch/dev_kit_perception_camera.launch
```

4. **待显存稳定后**启动数据集：

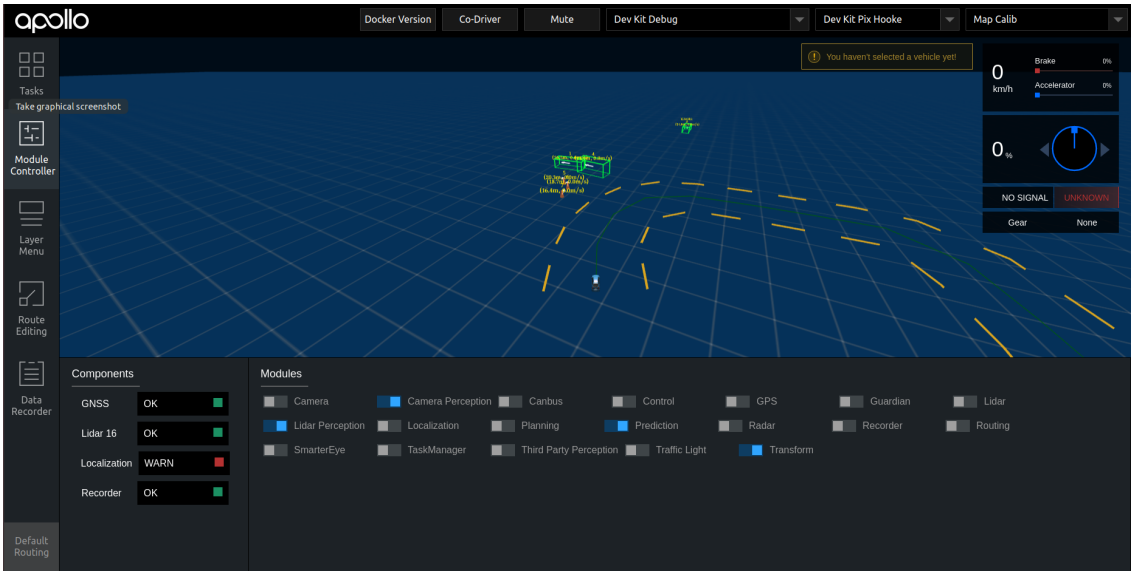
```
cyber_recorder play -f data/bag/20220203/20220203100813.record.0000*
```

5. 查看输出结果：（camera检测输出为三维目标框）

```
cyber_monitor
```

观察到`/apollo/perception/obstacles`中有数据输出。

6. 在`dreamviewer`最终效果如下：可以看出，**相机的视野较远**，还能有效的检测到**小物体**，但是**视野受限**。



3.3 基于相机和激光融合的感知模块

1. 确保存在定位模块/数据集有下列信息正确输出：
   - `/apollo/sensor/lidar16/compensator/PointCloud2`
   - `/apollo/localization/pose`
   - `/tf`及`/tf_static`
2. 修改相关配置：

   与单传感器感知不同，融合感知需要将点云、相机的感知结果进行后融合。为了保证输出通道不被占用，首先需要将相机感知的通道做一定调整，将结果传给位于点云感知的融合模块：

   - 修改文件：`modules/perception/production/conf/perception/camera/fusion_camera_detection_component.pb.txt`

     ```
     output_final_obstacles : true
     output_obstacles_channel_name : "/perception/obstacles"
     ```

   选择需要融合的主传感器，并设置输出通道：

   - 修改文件：`modules/perception/production/conf/perception/fusion/fusion_component_conf.pb.txt`

     ```
     fusion_method: "ProbabilisticFusion"
     fusion_main_sensors: "velodyne16"
     fusion_main_sensors: "front_6mm"
     object_in_roi_check: true
     radius_for_roi_object_check: 120
     output_obstacles_channel_name: "/apollo/perception/obstacles"
     output_viz_fused_content_channel_name:
     "/perception/inner/visualization/FusedObjects"
     ```

3. 设置启动文件：修改`modules/perception/production/dag/dag_streaming_perception.dag`，内容如下：

   ```
   module_config {
     module_library : "/apollo/bazel-
   bin/modules/perception/onboard/component/libperception_component_camera.so"
   ```

```
  components {
    class_name : "FusionCameraDetectionComponent"
    config {
      name: "FusionCameraComponent"
      config_file_path:
"/apollo/modules/perception/production/conf/perception/camera/fusion_camera_d
etection_component.pb.txt"
      flag_file_path:
"/apollo/modules/perception/production/conf/perception/perception_common.flag
"
    }
  }
}


module_config {
  module_library : "/apollo/bazel-
bin/modules/perception/onboard/component/libperception_component_lidar.so"

 components {
    class_name : "SegmentationComponent"
    config {
      name: "Velodyne16Segmentation"
      config_file_path:
"/apollo/modules/perception/production/conf/perception/lidar/velodyne16_segme
ntation_conf.pb.txt"
      flag_file_path:
"/apollo/modules/perception/production/conf/perception/perception_common.flag
"
      readers {
          channel: "/apollo/sensor/lidar16/compensator/PointCloud2"
        }
    }
  }

  components {
    class_name : "RecognitionComponent"
    config {
      name: "RecognitionComponent"
      config_file_path:
"/apollo/modules/perception/production/conf/perception/lidar/recognition_conf
.pb.txt"
      readers {
          channel: "/perception/inner/SegmentationObjects"
        }
    }
  }


  components {
    class_name: "FusionComponent"
    config {
      name: "SensorFusion"
      config_file_path:
"/apollo/modules/perception/production/conf/perception/fusion/fusion_componen
t_conf.pb.txt"
      readers {
          channel: "/perception/inner/PrefusedObjects"
```

```
        }
      }
    }

}
```

4. 启动融合感知模块，等待待显存稳定（一般在1-2分钟左右）。
5. **待显存稳定后**启动数据集：

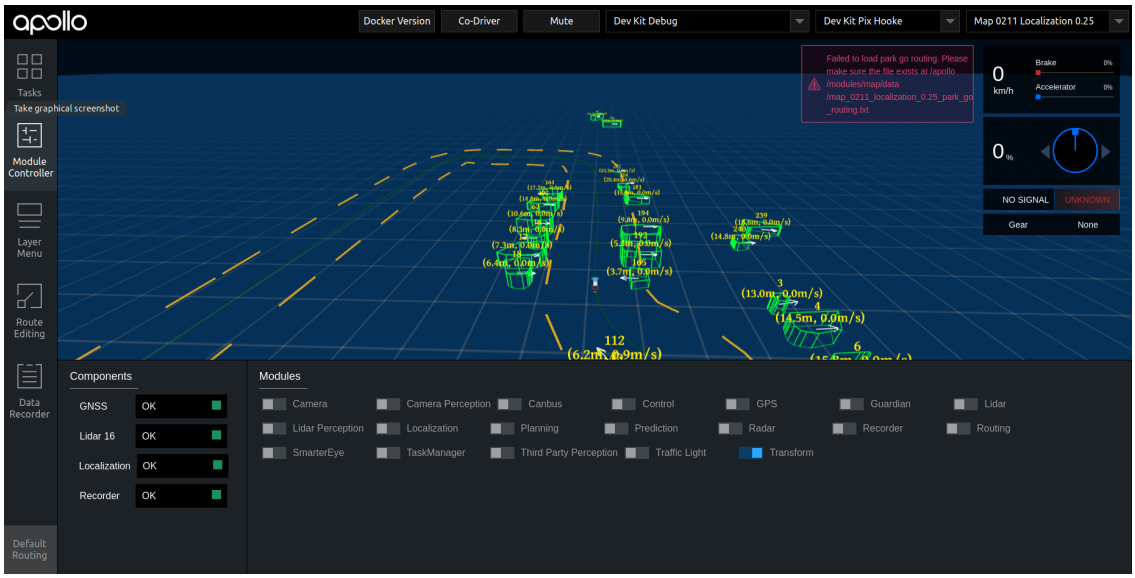
```
cyber_recorder play -f data/bag/20220203/20220203100813.record.0000*
```

6. 查看输出结果：（三维目标框和多边形框）

```
cyber_monitor
```

   观察到`/apollo/perception/obstacles`中有数据输出。

7. 在`dreamviewer`最终效果如下：可以看出，**近处时为点云检测为主的多边形目标，远处时为相机检测到的矩形框**，兼具了两者的优势。



3.4 启动预测模块

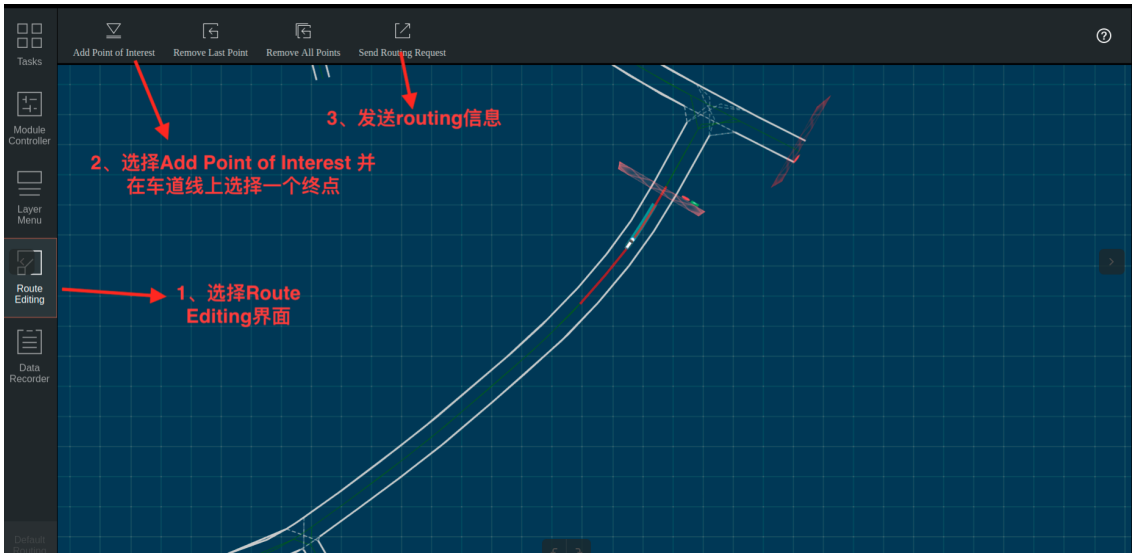
1. 确保感知、定位模块启动，车型、地图加载完毕
2. 启动`cyber_launch start modules/prediction/launch/prediction.launch`
3. 观测`cyber_monitor`的`/apollo/prediction`通道是否正常工作输出

```
ChannelName: /apollo/prediction
MessageType: apollo.prediction.PredictionObstacles
FrameRatio: 0.00
RawMessage Size: 3199 Bytes (3.12 KB)
header:
  timestamp_sec: 1644114068.160332918
  module_name: prediction
  sequence_num: 218
  lidar_timestamp: 1643854111485752064
  camera_timestamp: 1643854111485752064
  radar_timestamp: 0
pr  Take graphical screenshot  [6 items]
perception_error_code: OK
start_timestamp: 1644114068.156162262
end_timestamp: 1644114068.160304546
```

4 启动Planning、Routing模块

4.2 启动Routing模块

1. 确保地图中有车道线或者虚拟车道线，所有规划都在车道线内进行。虚拟车道线的制作详见地图制作的部分。
2. 在`Routing Editor`标签中点击`Add Point of Interest`按钮添加一个point, 然后选择`Send Routing Request`按钮发送添加的`routing`点



3. 启动：在dreamview中直接启动`Routing`模块即可。
4. **注意：制作出来的虚拟车道线是单行线，`routing`时不能反向通行。**

4.3 启动Planning模块

1. 确保完成：`油门刹车标定`及`PID调试`任务
2. 配置文件：`/apollo/modules/planning/conf/planning.conf`
   和`/apollo/modules/planning/conf/planning_config.pb.txt`两个配置文件

修改文件名称	修改内容	对应的gflag参数	单位	作用
`planning.conf`	修改`default_cruise_speed`数值	比如1.5	m/s	默认巡航速度
`planning.conf`	修改`planning_upper_speed_limit`数值	比如1.5	m/s	车planning最大速度
`planning.conf`	添加`planning_lower_speed_limit`数值	比如0.5	m/s	车planning最小速度
`planning.conf`	添加`speed_upper_bound`数值	比如1.5	m/s	车最大速度
`planning.conf`	添加`max_stop_distance_obstacle`数值	比如10	m	障碍物最大停止距离
`planning.conf`	添加`min_stop_distance_obstacle`数值	比如5	m	障碍物最小停止距离
`planning.conf`	添加`destination_check_distance`数值	比如1.0	m	认为车已经到达目的地时，车与目的地距离
`planning.conf`	添加`lon_collision_buffer`数值	比如0.3	m	车与障碍物的默认碰撞距离
`planning.conf`	添加`enable_scenario_park_and_go`配置项	false		使起步停车场景失效
`planning_config.pb.txt`	修改`total_time`数值	比如15.0	s	planning规划多长时间的路线
`planning_config.pb.txt`	修改`max_acceleration`数值	比如1.0	m/s^2	车辆最大加速度
`planning_config.pb.txt`	修改`lowest_speed`数值	比如0.5	m/s	planning时车的最低速度
`planning_config.pb.txt`	修改`max_speed_forward`数值	比如1.5	m/s	车前进的最大速度
`planning_config.pb.txt`	修改`max_acceleration_forward`数值	比如1.0	m/s^2	车前进的最大加速度

3. 在dreamview中直接启动`Planning`模块即可
4. 在docker环境中输入命令`cyber_monitor`并查看planning channel信息：有数据输出即正确