7、Cartographer graph building algorithm

Cartographer：https://google-cartographer.readthedocs.io/en/latest/

Cartographer ROS2：https://github.com/ros2/cartographer_ros

The operating environment and software and hardware reference configurations are as follows:

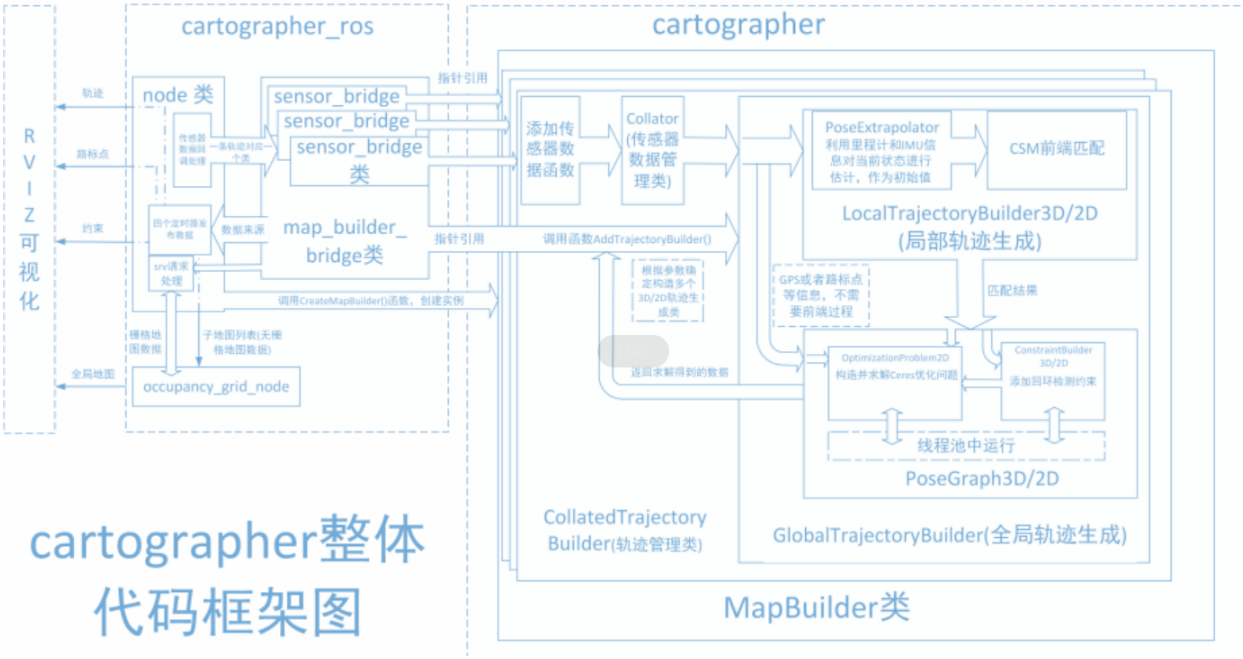
- REFERENCE MODEL: ROSMASTER X3
- Robot hardware configuration: Arm series main control, Silan A1 lidar, AstraPro Plus depth camera
- Robot system: Ubuntu (version not required) + docker (version 20.10.21 and above)
- PC Virtual Machine: Ubuntu (20.04) + ROS2 (Foxy)
- Usage scenario: Use on a relatively clean 2D plane

7.1、Introduction

- Cartographer

Cartographer is a 2D and 3D SLAM (simultaneous localization and mapping) library supported by Google's open source ROS system. Method graph construction algorithm based on graph optimization (multi-threaded backend optimization, problem optimization built by CERE). Data from multiple sensors, such as LIDAR, IMUs, and cameras, can be combined to simultaneously calculate the sensor's position and map the environment around the sensor.

The source code of Cartographer mainly includes three parts: Cartographer, cartographer_ros, and Cers-Solver (back-end optimization).



Cartographer adopts the mainstream SLAM framework, that is, the three-stage of feature extraction, closed-loop detection, and back-end optimization. A certain number of LaserScan forms a submap submap, and a series of submap submaps make up the global map. The cumulative error of the short-term process of building a submap with LaserScan is not large, but the long-term process of building a global map with a submap will have a large cumulative error, so it is necessary to use closed-loop detection to correct the position of these submaps, the basic unit of closed-loop detection is submap, and closed-loop detection adopts a scan_match strategy. The focus of cartographer is the creation of submap submaps that fuse multi-sensor data (odometry, IMU, LaserScan, etc.) and the implementation of scan_match strategies for closed-loop inspection.

- cartographer_ros

cartographer_ros is running under ROS and can accept various sensor data in the form of ROS messages

After processing, it is published in the form of a message for easy debugging and visualization.

7.2、use

7.2.1、pre-use configuration

Note: Since ROSMASTER series robots are divided into multiple robots and multiple devices, the factory system has been configured with routines for multiple devices, but because the product cannot be automatically identified, it is necessary to manually set the machine type and radar model.

After entering the container: according to the model of the car, the type of radar and the type of camera, make the following modifications:

```
root@ubuntu:/# cd
root@ubuntu:~# vim .bashrc
```

```
fi

# env
alias python=python3
export ROS_DOMAIN_ID=12

export ROBOT_TYPE=x3   # r2, x1, x3
export RPLIDAR_TYPE=a1  # a1, s2
export CAMERA_TYPE=astraplus  #astrapro, astraplus
echo "------------------------------------------------------"
echo -e "ROS_DOMAIN_ID: \033[32m$ROS_DOMAIN_ID\033[0m"
echo -e "my_robot_type: \033[32m$ROBOT_TYPE\033[0m | my_lidar: \033[32m$RPLIDAR_TYPE\033[0m | my_camera: \033[32m$CAMERA_TYPE\033[0m"
echo "------------------------------------------------------"

#colcon_cd
source /usr/share/colcon_cd/function/colcon_cd.sh
export _colcon_cd_root=/root/yahboomcar_ros2_ws/yahboomcar_ws
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash

#ros2
source /opt/ros/foxy/setup.bash
source /root/yahboomcar_ros2_ws/yahboomcar_ws/install/setup.bash
source /root/yahboomcar_ros2_ws/software/library_ws/install/setup.bash
```

After the modification is complete, save and exit vim, and then execute:

```
root@ubuntu:~# source .bashrc
------------------------------------------------------
ROS_DOMAIN_ID: 12
my_robot_type: x3 | my_lidar: a1 | my_camera: astraplus
------------------------------------------------------
root@ubuntu:~#
```

You can see the model of the currently modified car, the type of radar and the type of camera.

7.2.2、specific use

Note: When building the image, the slower the effect, the better (note if the rotation speed is slower), too fast the effect, the effect will be poor.

First of all, you need to do port binding operations in the host [that is, on the jetson of the car] [see the port binding tutorial chapter], which mainly uses two devices: radar and serial port;

Then check whether the radar and serial device are in the port binding state: on the host [that is, the jetson of the car], refer to the following command to view, and the successful binding is the following state:

```
jetson@ubuntu:~$ ll /dev | grep ttyUSB*
lrwxrwxrwx   1 root   root           7 Apr 21 14:52 myserial → ttyUSB0
lrwxrwxrwx   1 root   root           7 Apr 21 14:52 rplidar → ttyUSB1
crwxrwxrwx   1 root   dialout 188,   0 Apr 21 14:52 ttyUSB0
crwxrwxrwx   1 root   dialout 188,   1 Apr 21 14:52 ttyUSB1
```

If the radar or serial device is not bound to the display, you can plug and unplug the USB cable to view it again

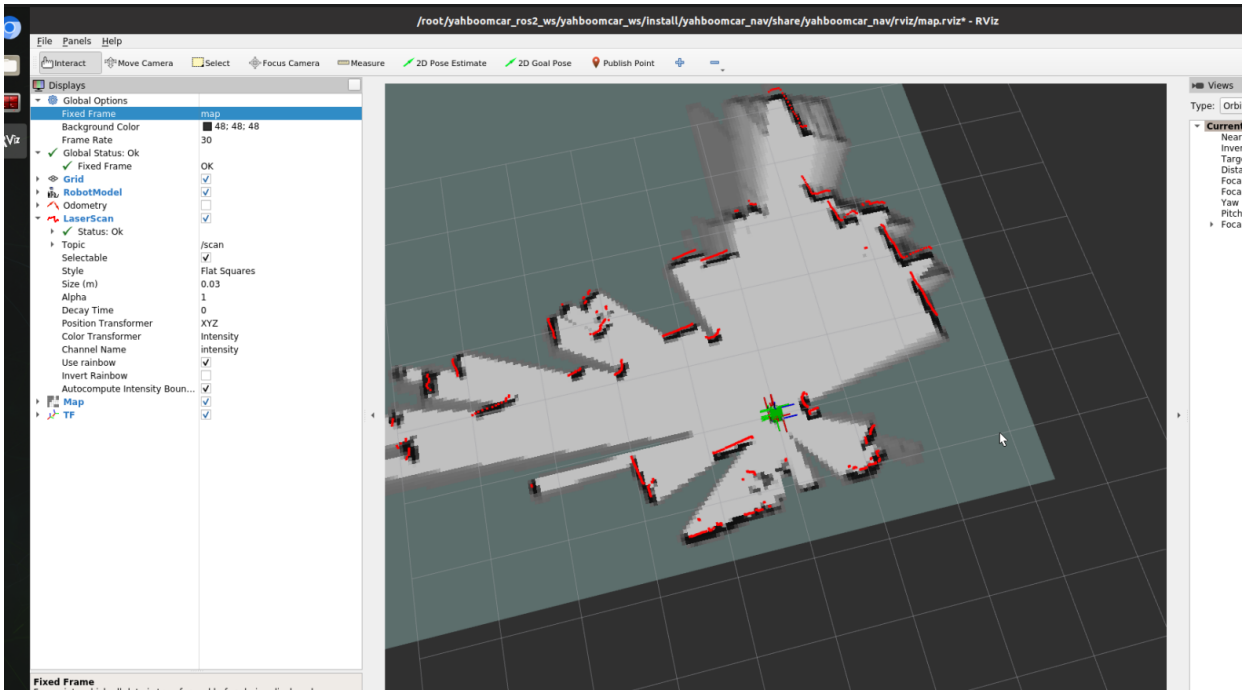
Enter the docker container, see [Docker course ----- 5. Enter the robot's docker container], and execute the following launch file in the terminal:

1. Start mapping

```
ros2 launch yahboomcar_nav map_cartographer_launch.py
```

2. Start rviz to display the map, this step is recommended to be executed in the virtual machine, and multi-machine communication needs to be configured in the virtual machine

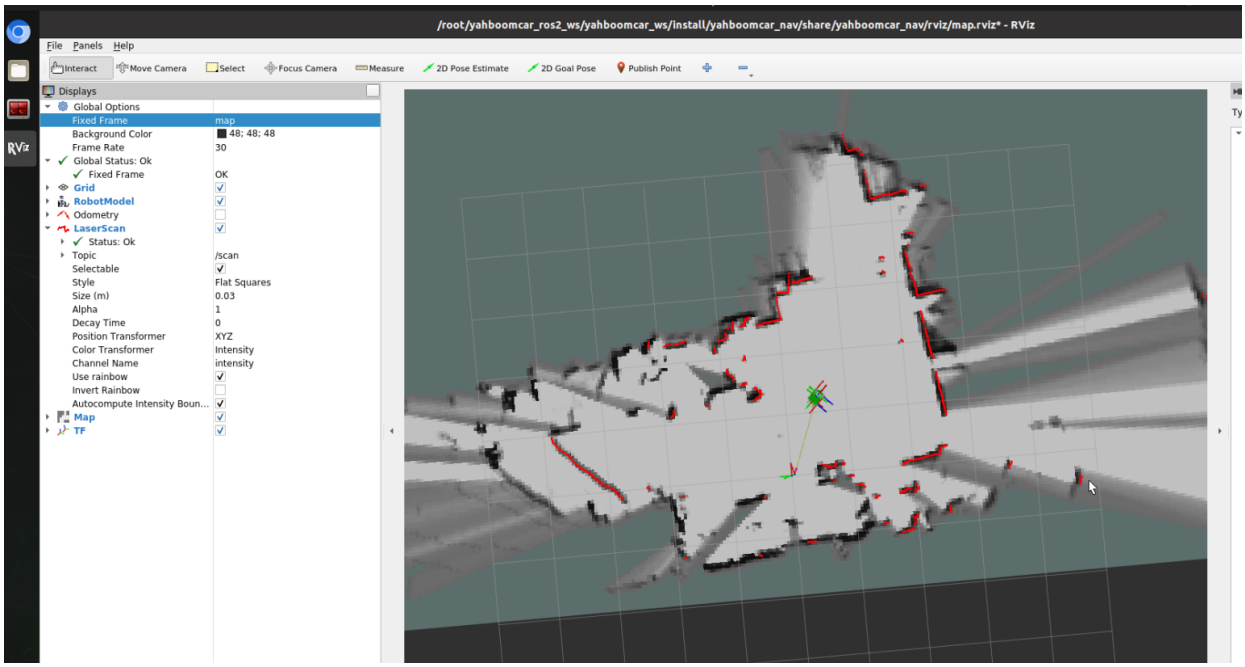
```
ros2 launch yahboomcar_nav display_map_launch.py
```



3. Start the keyboard control node, this step is recommended to be executed in the virtual machine, and multi-machine communication needs to be configured in the virtual machine

   Or use the remote control [Move the trolley slowly] to start building the map until the complete map is built

```
ros2 run yahboomcar_ctrl yahboom_keyboard
```

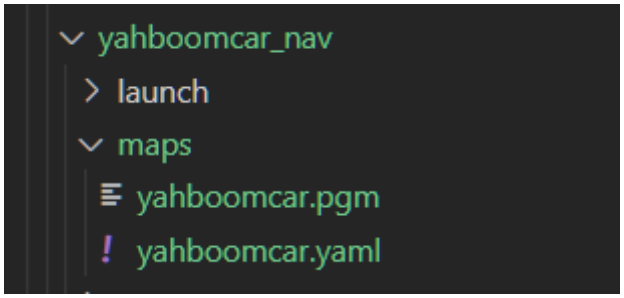


4. Save the map, the path is as follows:

```
ros2 launch yahboomcar_nav save_map_launch.py
```

The save path is as follows:

```
/root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/
```



A PGM image, a YAML file yahboomcar.yaml

```
image: /root/yahboomcar_ros2_ws/yahboomcar_ws/src/yahboomcar_nav/maps/yahboomcar.pgm
mode: trinary
resolution: 0.05
origin: [-10, -10, 0]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.25
```

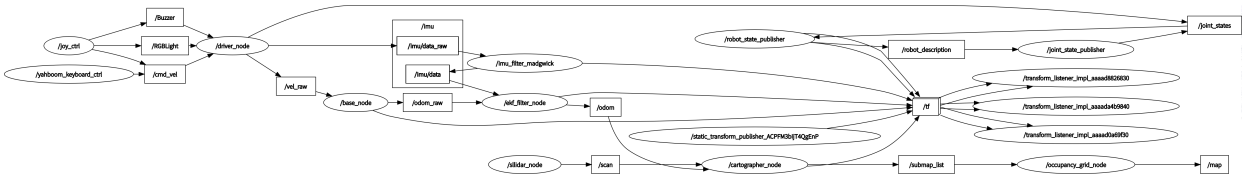
Parameter parsing:

- image: The path to the map file, which can be absolute or relative
- mode: This attribute can be one of trinary, scale, or raw, depending on the mode selected, trinary mode is the default mode
- Resolution: The resolution of the map, meters per pixel
- origin: 2D pose (x,y,yaw) in the lower left corner of the map, where yaw is rotated counterclockwise (yaw=0 means no rotation). Many parts of the system today ignore the YAW value.
- negate: whether to reverse the meaning of white/black, free/occupy (the interpretation of the threshold is not affected)
- occupied_thresh: Pixels with a probability of occupancy greater than this threshold are considered fully occupied.
- free_thresh: Pixels with a probability of occupancy less than this threshold are considered completely free.

7.3、node parsing

7.3.1、display the computational graph

```
rqt_graph
```



7.3.2、cartographer_node details

```
rr@rr-pc:~/rviz$ ros2 node info /cartographer_node
/cartographer_node
  Subscribers:
    /odom: nav_msgs/msg/Odometry
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /scan: sensor_msgs/msg/LaserScan
  Publishers:
    /constraint_list: visualization_msgs/msg/MarkerArray
    /landmark_poses_list: visualization_msgs/msg/MarkerArray
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /scan_matched_points2: sensor_msgs/msg/PointCloud2
    /submap_list: cartographer_ros_msgs/msg/SubmapList
    /tf: tf2_msgs/msg/TFMessage
    /trajectory_node_list: visualization_msgs/msg/MarkerArray
  Service Servers:
    /cartographer_node/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /cartographer_node/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /cartographer_node/get_parameters: rcl_interfaces/srv/GetParameters
    /cartographer_node/list_parameters: rcl_interfaces/srv/ListParameters
    /cartographer_node/set_parameters: rcl_interfaces/srv/SetParameters
    /cartographer_node/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
    /finish_trajectory: cartographer_ros_msgs/srv/FinishTrajectory
    /start_trajectory: cartographer_ros_msgs/srv/StartTrajectory
    /submap_query: cartographer_ros_msgs/srv/SubmapQuery
    /write_state: cartographer_ros_msgs/srv/WriteState
  Service Clients:

  Action Servers:

  Action Clients:
```

```
rr@rr-pc:~/rviz$ ros2 node info /occupancy_grid_node
/occupancy_grid_node
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /submap_list: cartographer_ros_msgs/msg/SubmapList
  Publishers:
    /map: nav_msgs/msg/OccupancyGrid
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /occupancy_grid_node/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /occupancy_grid_node/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /occupancy_grid_node/get_parameters: rcl_interfaces/srv/GetParameters
    /occupancy_grid_node/list_parameters: rcl_interfaces/srv/ListParameters
    /occupancy_grid_node/set_parameters: rcl_interfaces/srv/SetParameters
    /occupancy_grid_node/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:
    /submap_query: cartographer_ros_msgs/srv/SubmapQuery
  Action Servers:

  Action Clients:
```

7.3.3、TF transform

```
ros2 run tf2_tools view_frames.py
```