雷达跟随功能使用与讲解

1. 功能简介

雷达跟随功能，主要利用激光雷达 360° 实时扫描周围环境，寻找周围可检测到的最近的物体，并对其进行跟随。若跟随途中扫描到新的距离更近的物体，小车则会丢弃当前跟随目标，选择跟随新出现的更近的目标物体。最终小车与被跟随物保持一段距离，车头朝向被跟随物。

2. 使用方法

使用前需要确保上位机连接小车 WiFi，并已经 SSH 远程登录到小车端。

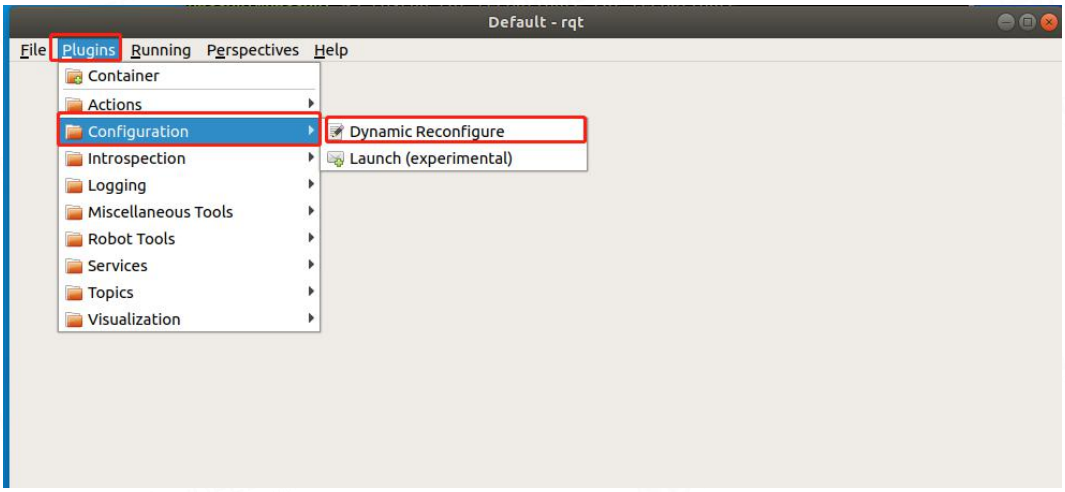
1) 打开终端输入启动雷达跟随功能的指令

```
roslaunch simple_follower laser_follower.launch
```

2) rqt 在线调参

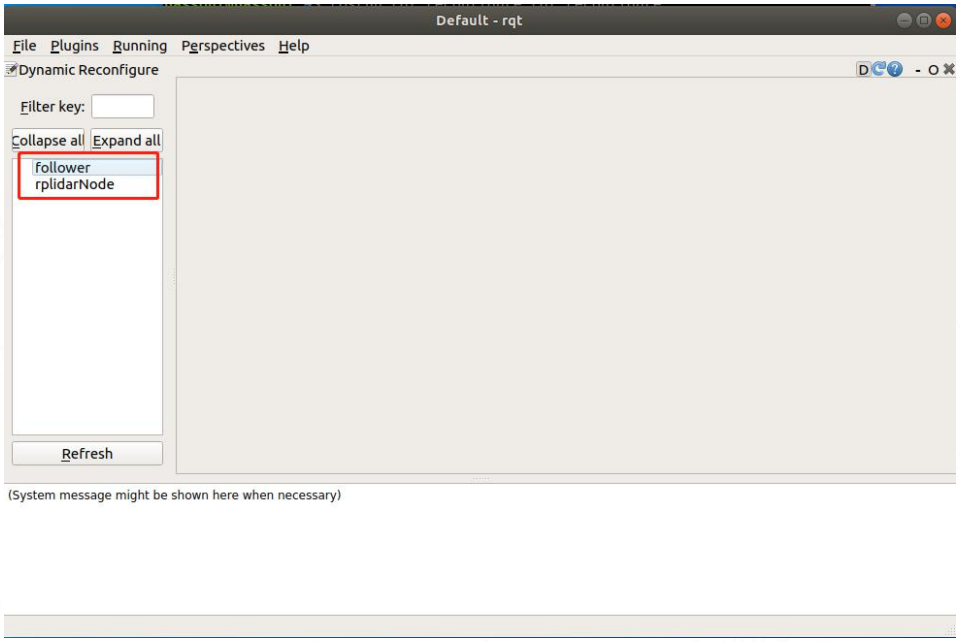
```
rqt
```

在成功打开 rqt 后，会弹出一个 Default -rqt 窗口，之后点击左上角的 Plugins →选择 Configuration→点击 Dynamic Reconfigure。



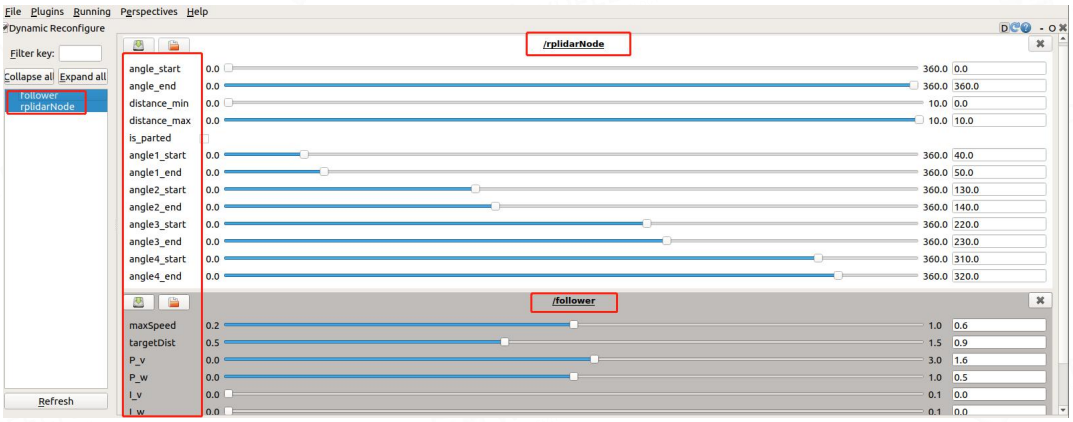
打开 Dynamic Reconfigure

打开 Dynamic Reconfigure 之后会出现如下界面，其中左侧任务栏中有两个可在线调参的节点。（若没有显示节点，可通过点击左下角的 Refresh 进行刷新）



Dynamic Reconfigure 界面

选中 follower 和 rplidarNode 两个节点（也可选择一个节点，对一个节点的参数进行调参），之后进入在线调参界面，共 21 个参数可以进行在线调参，可通过拖动滑动条上的滑块或直接在参数后端的输入框进行输入具体参数并按下回车键进行调参。



在线调参界面

调整参数之后，小车根据参数做出相应的反馈，相应参数的 INFO 日志也将会输出在运行 laser_follower.launch 的终端上。

3. 注意事项

① 运行雷达跟随功能

运行 laser_follower.launch 后若终端无红色报错则运行成功。

```
[WARN] [1625489905.525372]: 1
starting
[INFO] [1625489906.904107]: PID initialised with P:[1.6, 0.5], I:[0.0, 0.0], D:[0.0, 0.0]
[INFO] [1625489906.907751]: max_speed:0.6,targetDist:0.9
[INFO] [1625489906.927095]: PID initialised with P:[1.6, 0.5], I:[0.0, 0.0], D:[0.0, 0.0]
RPLIDAR S/N: C85399F6C9E59AD2C5E59CF734813412
[ INFO] [1625489907.206649599]: Firmware Ver: 1.29
[ INFO] [1625489907.206730849]: Hardware Rev: 7
[ INFO] [1625489907.208271422]: RPLidar health status : 0
[ INFO] [1625489907.753604546]: current scan mode: Sensitivity, max_distance: 12.0 m, Point number: 7.9K , angle_compensate: 2
[INFO] [1625489907.885021]: lost connection
[WARN] [1625489909.124291]: laser no object found
[WARN] [1625489909.128987]: laser:nothing found
```

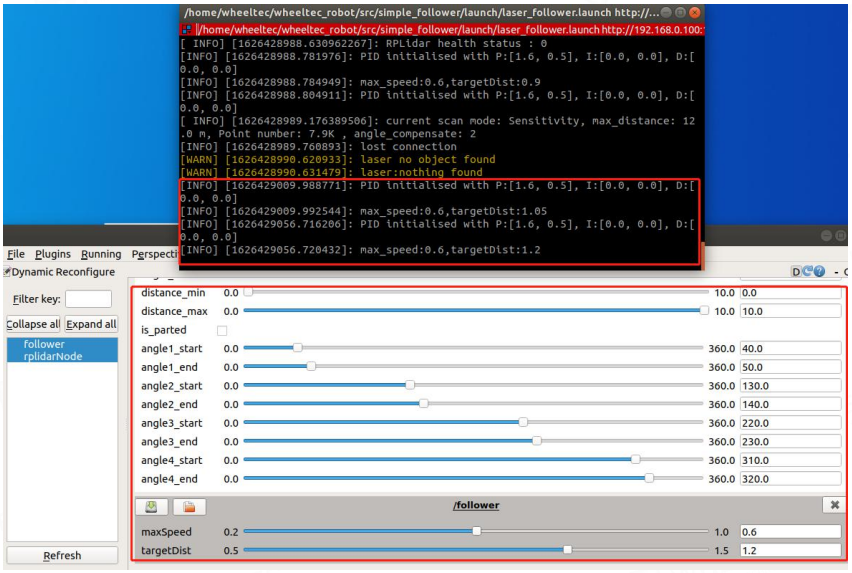
运行成功

雷达跟随启动后，小车会不断寻找雷达扫描范围内距离最近的目标，之后找到一个距离最近且合理的目标进行跟随，最终与被跟随物保持适当的距离（即中距值），车头正对被跟随物体（即小车与被跟随物体之间夹角为 0°）。

注意：开启雷达跟随时尽量保证空间不要过于狭小，物与物之间的距离最好保持 1m 及以上。同时应注意雷达高度，由于雷达只能扫描到与其处于同一水平面上的物体，因此不要将高度较低的物品放置在小车行经路线上，以免发生碰撞。

② rqt 调参

其中 targetDist 每次调整须在 0.1 及以上，否则可能被 PID 控制视作误差较小而忽略。distance_min 及 distance_max 的调参过程中，若出现 distance_min 比 distance_max 大，则会互换两个参数的值以保证 max 比 min 大。angle_start、angle_end 是扫描的角度，而 angle1_start~angle4~end 是屏蔽的角度。且屏蔽角度需设置的比实际需要屏蔽的角度略大一些。

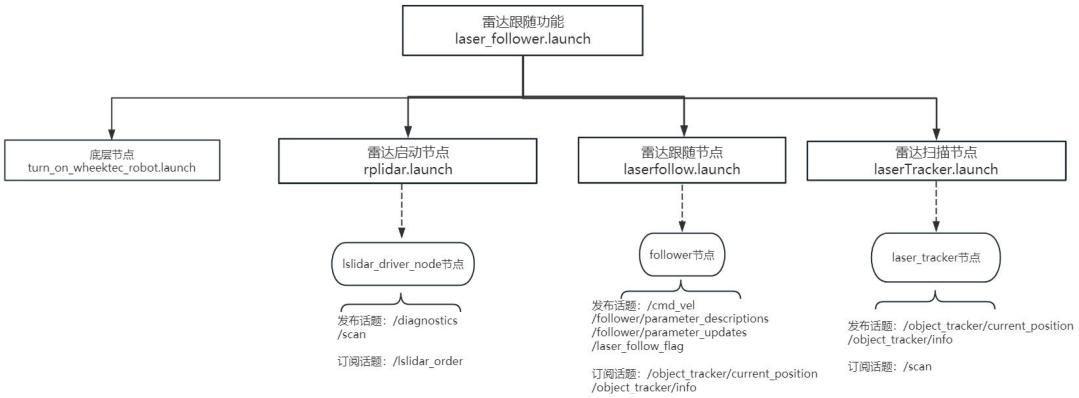


调参

4. 功能讲解

雷达跟随功能通过运行 laser_follower.launch 开启，在 laser_follower.launch 中包含了四个 launch 文件。其中 turn_on_wheeltec_robot.launch 用于开启小车的底层运动节点，另外三个 launch 文件中的 rplidarNode、laser_follow.py 及 laserTracker.py 负责主要功能的实现（其中 rplidarNode 为节点名，其对应源码文件为 rplidar_ros 中的 node.cpp 文件）。

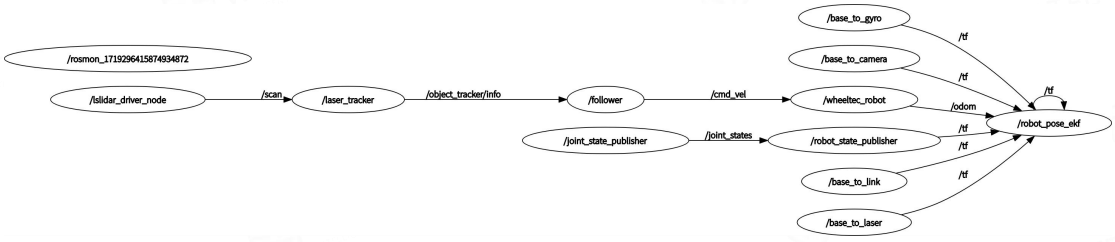
① 启动雷达跟随功能



雷达跟随功能启动框架

② 雷达跟随功能节点关系

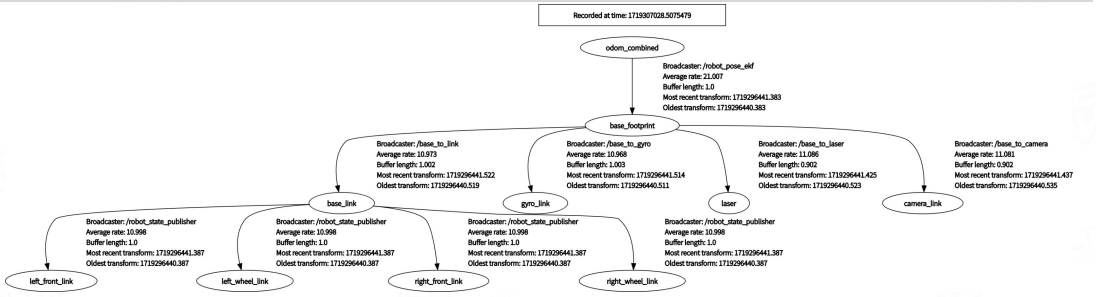
```
rqt_graph
```



雷达跟随功能节点关系图

③ 雷达跟随功能 TF 树

```
rosrun rqt_tf_tree rqt_tf_tree
```



建图功能 TF 树

④ 参数说明

1.雷达扫描范围

雷达扫描范围由 angle_start、angle_end、distance_min 及 distance_max 四个参数决定，其中两个 angle 参数决定扫描角度，两个 distance 参数决定扫描半径。即雷达只扫描 angle_start 与 angle_end 之间的角度，只扫描 distance_min 到 distance_max 之间的距离范围。

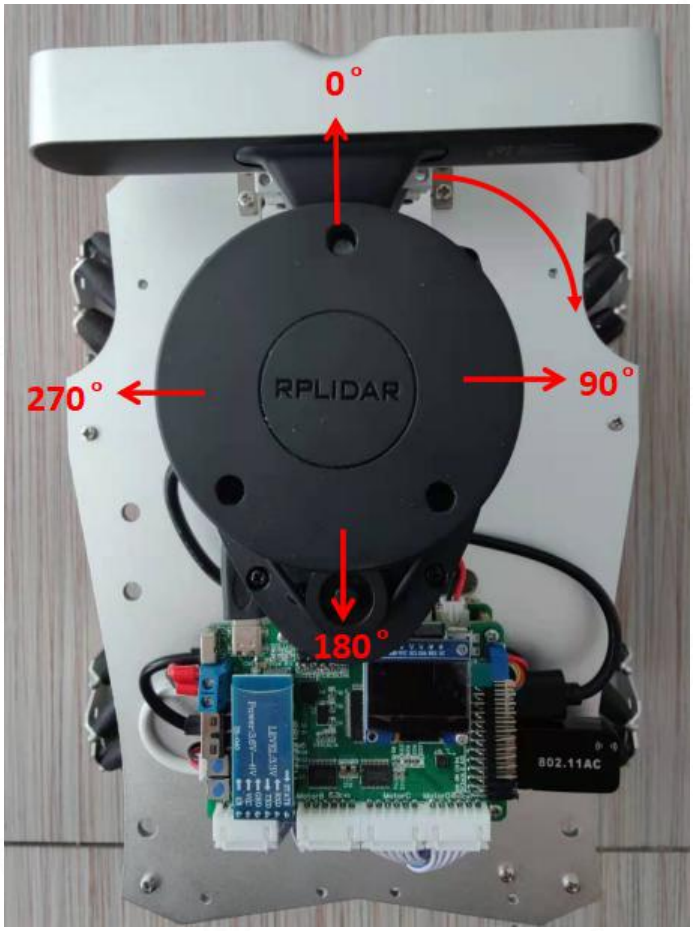
参数解释：

angle_start:雷达扫描的起始角度(扫描的角度以车头方向为 0°，顺时针转动，参考下图)

angle_end:雷达扫描的结束角度（angle_start 与 angle_end 并非限制雷达实际扫描范围，而是在程序中屏蔽了角度小于 angle_start 大于 angle_end 的扫描数据以达到限制角度的效果）

distance_min:雷达扫描的最小半径（单位：米/m）

distance_max:雷达扫描的最大半径



雷达扫描角度

2.雷达多角度屏蔽

多角度屏蔽共由 9 个参数组成，is_parted 及 angle1_start~angle4_end，其中 is_parted 为一个开关，当它为 true 时开启雷达的多角度屏蔽，上面雷达扫描范围中的 angle_start 及 angle_end 会失效，而 angle1_start~angle4_end 则会生效。当它为 false 的时候则 angle_start 及 angle_end 生效，angle1_start~angle4_end 失效。开启雷达多角屏蔽时，雷达会屏蔽 angle1_start~angle1_end、angle2_start~angle2_end、angle3_start~angle3_end 及 angle4_start~angle4_end 的扫描角度，当 start 和 end 的值相同的时候则不屏蔽角度，故可以通过将 start 与 end 置为相同的角度以控制屏蔽角度的数量。需注意的是要将雷达屏蔽的参数与雷达扫描范围的角度参数区分开，因为一个是屏蔽的角度，一个是要扫描的角度。且雷达屏蔽角度需要设置的比实际需要屏蔽的角度略大。

参数解释：

is_parted:是否开启雷达多角度屏蔽（开启后，angle_start 及 angle_end 参数失效，angle1_start~angle4_end 这 8 个参数生效）

angle1_start:第一个屏蔽角度的起始角

angle1_end:第一个屏蔽角度的结束角（后面的 angle2、3、4 与 angle1 含义一致，不一一解释）

3.中距值及最大速度

小车与目标物之间保持静止状态的距离中距值由 targetDist 决定，在进行雷达跟随时小车运动的最大速度则由 maxSpeed 参数决定。

参数解释：

maxSpeed:小车在雷达跟随时的最大速度（单位：m/s）

targetDist:小车与目标的中距值（即小车与目标之间达到该距离时小车保持静止状态，单位：m）

4.PID 值

PID 值由 P_v、P_w、I_v、I_w、D_v、D_w 六个参数共同决定。P_v 及 P_w 决定小车的响应速度；I_v 及 I_w 使小车与目标值更加接近，减小误差；D_v 及 D_w 减少小车在目标值附近来回振动的情况，减少超调及调节时间。

参数解释：

P_v:线速度比例

P_w:角速度比例

I_v:线速度积分

I_w:角速度积分

D_v:线速度微分

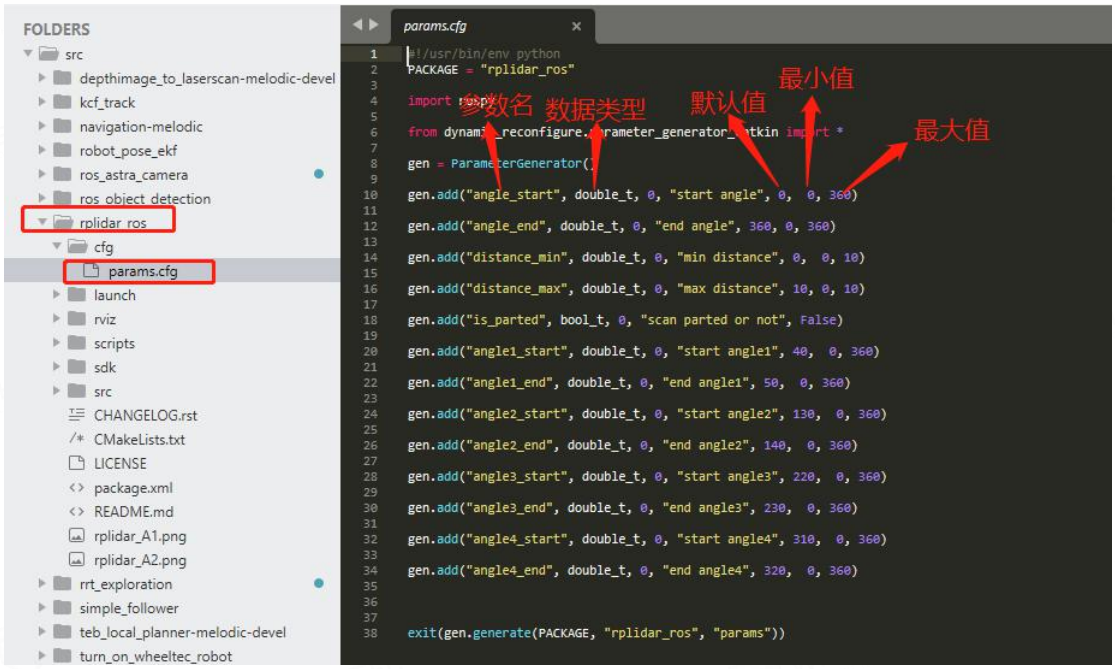
D_w:角速度微分

⑤ 参数修改

雷达跟随参数调整有两种方式。第一种是直接在源文件中修改参数值，在 launch 文件中修改参数值保存后即可生效不需要重新编译，cfg 文件需要重新编译方可生效。这种方式修改后长期有效，但需要重新开启节点才能生效。第二种便是 rqt 在线调参，其具有实时性，可在节点运行时进行参数调整，且立刻生效无需重启节点，但修改后的参数仅在此次节点运行时生效，重新开启节点后参数依然为参数服务器中的默认值。因此 rqt 在线调参可以作为一种调试工具，但要让修改的参数长期有效还是需要到源文件中进行参数修改。

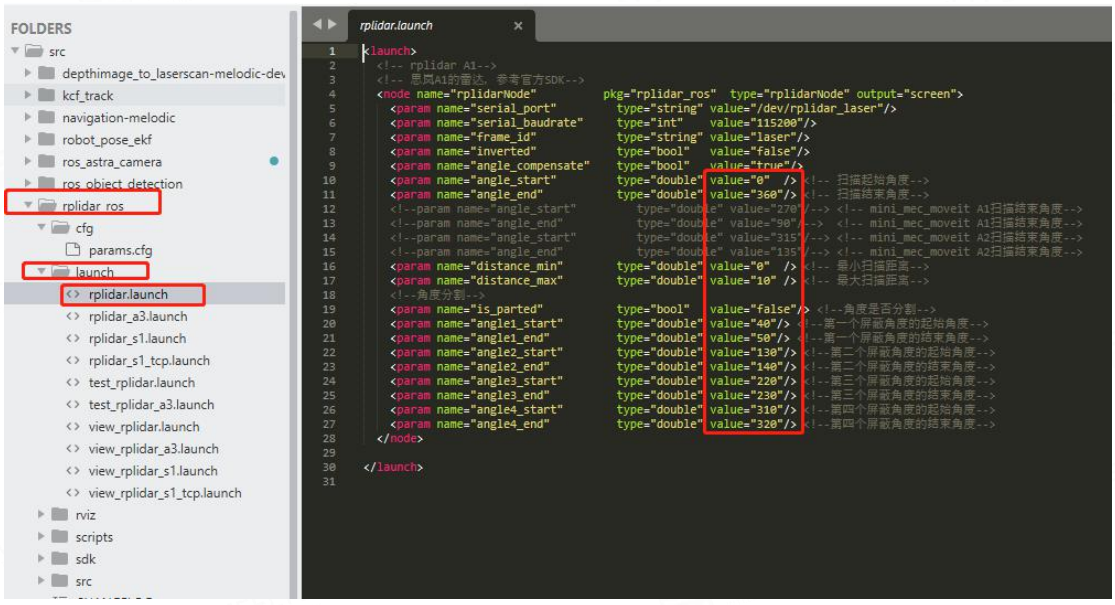
下图是 rqt 在线调参中雷达节点的 cfg 文件，里面的参数即在线调参时显示的参数。可在这个文件中修改最小值最大值来改变在线调参时参数的上下限。这里的默认值由于会被参数服务器中同名参数的值覆盖，所以在这修改默认值是无效的，修改参数默认值需要到相应的 launch 文件中进行操作（除了 PID 的六个参数，PID 的六个参数在参数服务器中不存在同名参数）。

注意：修改最小最大值需注意雷达或小车本身的性能上下限。cfg 文件的修改需要进行编译才可生效，编译前应将系统时间设置为当前时间。



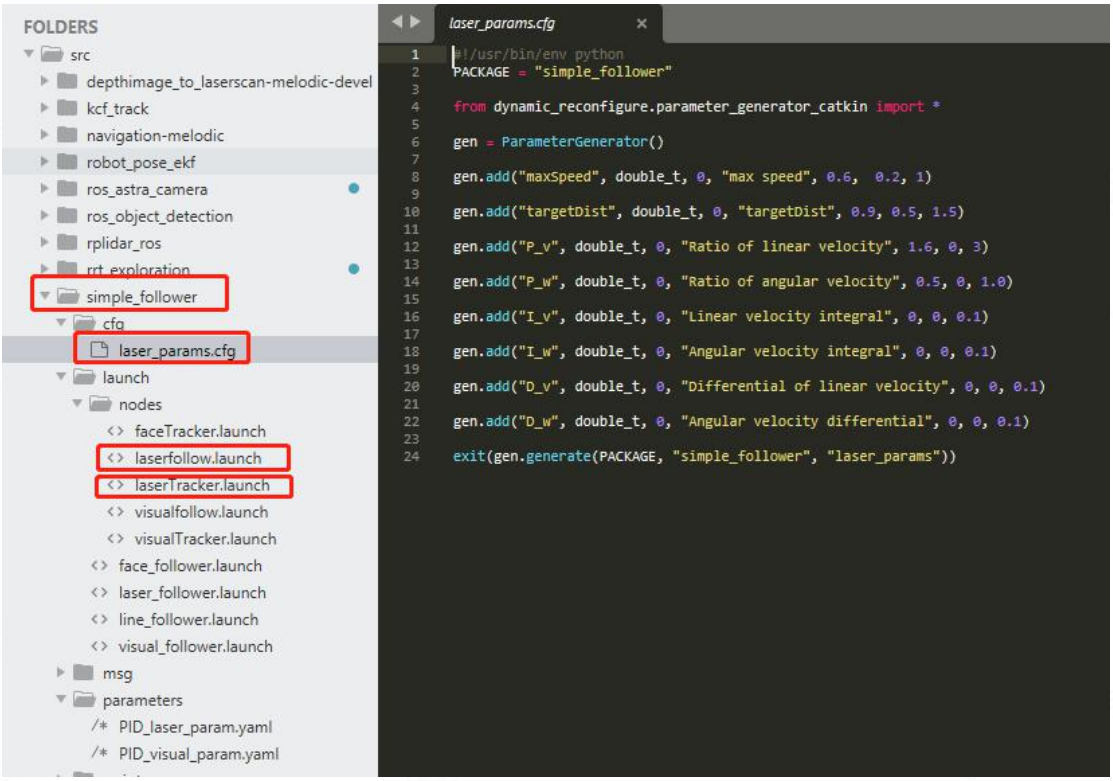
雷达节点 cfg 文件

在每个功能节点对应的 launch 文件中修改 param 中的 value 值即可修改相关参数的默认值。（launch 文件的修改保存后即可生效不需要重新编译）



雷达节点 launch 文件

下图则为雷达跟随的 cfg 文件及 launch 文件，主要修改最大速度、中距值及 PID 参数。修改方式和上面雷达节点的一样，故在这里不再细述。



雷达跟随 cfg 及 launch 文件

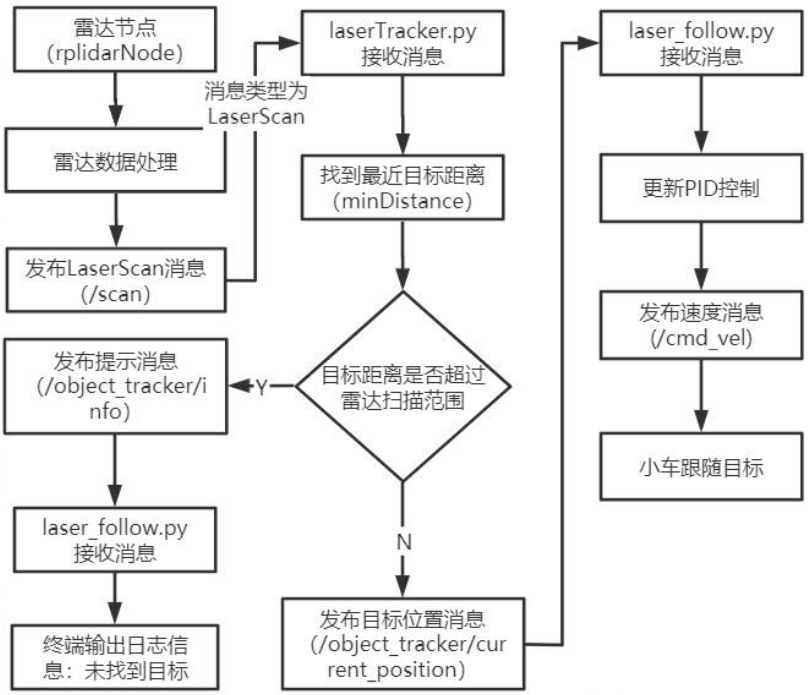
⑥ 雷达跟随功能流程

雷达 360°扫描获取数据后，雷达节点进行数据处理并在/scan 话题中发布 LaserScan 类型的雷达扫描数据消息。

laserTracker.py 中的 scanSubscriber 通过订阅/scan 话题接收到 LaserScan 消息后，得到周围 360°扫描点的数据，从中找出最近且有效的目标。当找到最近目标后，将其距离与雷达扫描范围进行对比。

若距离超过雷达扫描半径的最大值，则认为该目标不是一个正确的目标，并在/object_tracker/info 话题中发布 StringMsg 类型的提示消息，laser_follow.py 中的 trackerInfoSubscriber 通过订阅/object_tracker/info 话题接收到该话题的消息后在终端输出日志：“laser:nothing found”（未找到目标）。

若距离在雷达扫描范围内，则进行目标角度计算，并在/object_tracker/current_position 话题中发布 Position 类型的目标位置消息，Position 消息中包含目标的角度及距离信息，laser_follow.py 中的 positionSubscriber 通过订阅/object_tracker/current_position 话题接收到该消息后，更新 PID 控制，最后在/cmd_vel 话题中发布 Twist 类型的速度消息控制小车的速度，实现小车对目标的跟随（如果与目标距离已经达到中距值且角度为 0°则小车速度为 0）。



雷达跟随程序流程

⑦ 雷达跟随功能关键源码解析

```
#源文件：rplidar_ros/src/node.cpp
for (size_t i = 0; i < node_count; i++) {
                float read_value = (float) nodes[i].dist_mm_q2/4.0f/1000;
                 //angle_cur 为当前扫描角度
                angle_cur = i*360/node_count;
                if(angle_end>=angle_start)
                  {
                    if(angle_cur>angle_end ||
angle_cur<angle_start)read_value=0;
                  }
                else
                  {
                    if(angle_cur>angle_end &&
angle_cur<angle_start)read_value=0;
                  }
                // if (read_value == 0.0)
                if (read_value <= distance_min || read_value >= distance_max)
                    scan_msg.ranges[i] = std::numeric_limits<float>::infinity();
// 如果不在角度不在扫描范围内，将该角度对应的目标距离置为无穷大
                else
                    scan_msg.ranges[i] = read_value;
                scan_msg.intensities[i] = (float) (nodes[i].quality >> 2);
            }
```

这段源码主要进行雷达扫描范围的限制，angle_cur 为当前扫描的角度，

read_value 为该角度对应的障碍物距离信息。通过条件判断语句判断 angle_cur 是否在扫描范围 angle_start~angle_end 之间，不在这个范围内则将 read_value 值先置为 0。之后用 read_value 与雷达扫描半径 distance_min，distance_max 作比较，如果 read_value 小于等于最小半径或者大于等于最大半径（read_value 为 0 必然小于等于最小半径），就将 read_value 置为无穷大，read_value 为无穷大就意味着该角度对应的障碍物距离为无穷远相当于没有障碍物。我们可以发现，其实雷达依然是 360°进行扫描，只不过我们在进行数据处理时将我们手动设定的扫描范围之外的数据屏蔽了，从而达到了控制扫描范围的效果。雷达多角度屏蔽的实现原理也与此类似，在这就不再详述。

```
#源文件: laserTracker.py
    def registerScan(self, scan_data):
        ranges = np.array(scan_data.ranges)
        sortedIndices = np.argsort(ranges) # 根据距离进行排序
        minDistanceID = None
        minDistance   = float('inf')

        if(not(self.lastScan is None)):
            # 若我们已经有了上一次扫描用来比较
            for i in sortedIndices:
                # 从最近的目标点开始检查所有测量的范围
                tempMinDistance   = ranges[i]

                windowIndex = np.clip([i-self.winSize,
i+self.winSize+1],0,len(self.lastScan))
                window = self.lastScan[windowIndex[0]:windowIndex[1]]

                with np.errstate(invalid='ignore'):
                    # 检查窗口中任何扫描点（在上次扫描中）是否和当前的扫描点相近
                    if(np.any(abs(window-tempMinDistance)<=self.deltaDist)):
                        # 我们找到了合理的距离
                        minDistanceID = i
                        minDistance = ranges[minDistanceID]
                        break # 至少有一个点相当地接近
                        # 所以我们找到了一个有效的最小值，可以停止循环
        self.lastScan=ranges
```

这段源码主要实现筛选出距离最近的有效目标。通过 np.argsort()方法获取对 ranges 按值从小到大排序后的索引顺序 sortedIndices，并以此为顺序遍历 ranges 即按目标距离从近到远遍历（ranges 为数组，其中每个元素代表一个扫描点，每

个元素存储着对应扫描点的目标距离信息，每个元素的索引值（i）*扫描精度（angle_increment）+雷达起始角度（angle_min）即为该扫描点的在雷达中的角度）。window 为一个窗口可以理解为在雷达扫描范围中截取一段，之后拿这段位置的当前值与上一次扫描中该位置的值对比，若差值的绝对值小于 deltaDist 则当作有效。

```
#源文件: laserTracker.py
        if(minDistance > scan_data.range_max):
            # 我们没有真正找到一个合适的目标
            # 发布 warning 日志: 我们没有找到任何目标
            rospy.logwarn('laser no object found')
            self.infoPublisher.publish(StringMsg('laser:nothing found'))

        else:
            # calculate angle of the objects location. 0 is straight ahead
            # 计算物体位置的角度, 0 在正前方
            minDistanceAngle = scan_data.angle_min + minDistanceID *
scan_data.angle_increment
            # 这里只有一个 x 角, 所以 y 是任意设定的
            self.positionPublisher.publish(PositionMsg(minDistanceAngle, 42,
minDistance))
            #self.infoPublisher.publish(StringMsg('successful'))
```

这里是将筛选出的最近目标距离 minDistance 与雷达扫描范围的最大半径 range_max 做对比，若大于雷达扫描的最大半径，则说明目标存在问题，我们并未找到一个有效目标。若小于或等于雷达扫描的最大半径，则计算目标的角度 minDistanceAngle，minDistanceID 为该扫描点在 ranges 中的索引号，angle_increment 为扫描精度即每一个扫描点的角度间隔，minDistanceID 与 angle_increment 相乘后再加上雷达扫描的起始角度 angle_min 即为目标在雷达中的角度（范围为-3.14~3.14），但由于我们的雷达是反装的这使得雷达上的 0°与小车车头方向相反，所以后面会在 laser_follow.py 文件中对角度做出修正，使 0°与小车车头方向保持一致。在获取到目标角度距离信息后将这些信息封装成 PositionMsg 消息类型并发布消息。

```
#源文件: laser_follow.py
    # Dynamic Reconfigure Config
    def reconfigCB(self,config,level):
        self.max_speed = config.maxSpeed # 将最大速度赋值为动态调参后的值
        targetDist = config.targetDist # 将中距值赋值为动态调参后的值
        # 获取 PID 值
```

```
        PID_param['P'] = [config.P_v,config.P_w]
        PID_param['I'] = [config.I_v,config.I_w]
        PID_param['D'] = [config.D_v,config.D_w]
        # 创建 simplePID 对象
        self.PID_controller = simplePID([0, targetDist], PID_param['P'],
PID_param['I'], PID_param['D']) # 实例化 simplePID 以实现 PID 控制更新

    rospy.loginfo("max_speed:{},targetDist:{}".format(self.max_speed,targetDist)
) # 在屏幕上输出最大速度与中距值日志
        return config
```

这段代码主要实现 rqt 在线调参功能，并实例化 simplePID 类。

```
#源文件: laser_follow.py
    def positionUpdateCallback(self, position):
        angleX= position.angleX # 与目标之间的角度
        distance = position.distance # 与目标之间的距离

        if(angleX>0):
            angleX=angleX-3.1415
        else :
            angleX=angleX+3.1415

        # 调用 PID 控制器来更新它并获得新的速度
        [uncliped_ang_speed, uncliped_lin_speed] =
self.PID_controller.update([angleX, distance])

        # 将这些速度上面规定的最大速度以内
        angularSpeed = np.clip(-uncliped_ang_speed, -self.max_speed,
self.max_speed)
        linearSpeed  = np.clip(-uncliped_lin_speed, -self.max_speed,
self.max_speed)

        # 创建 Twist 消息发送到 cmd_vel 话题
        velocity = Twist()
        velocity.linear = Vector3(linearSpeed,0,0.) # 线速度
        velocity.angular= Vector3(0., 0.,angularSpeed) # 角速度
        self.cmdVelPublisher.publish(velocity)
```

获取到 PostionMsg 后，首先通过 if(angle>0)这段条件语句将角度翻转 180°进行修正，保证小车正前方为 0°且角度范围为-3.14~3.14，方便后面 PID 控制。之后调用 PID 控制更新控制信号，利用 np.clip()方法将角速度与线速度限制在我们事先设定好的 max_speed 内。最后在/cmd_vel 话题中发布消息控制小车移动。

```
#源文件: laser_follow.py
def update(self, current_value):
```

```
        current_value=np.array(current_value) # [angleX distance]
        if(np.size(current_value) != np.size(self.setPoint)):
            raise TypeError('current_value and target do not have the same shape')
        if(self.timeOfLastCall is None):
            # PID 第一次被调用，我们还不知道时间差
            # 没有控制信号被应用
            self.timeOfLastCall = time.clock()
            return np.zeros(np.size(current_value))

        error = self.setPoint - current_value # 偏差值

        # 偏差较小时停止移动
        if error[0]<0.1 and error[0]>-0.1: # error[0]为角度偏差
            error[0]=0
        if error[1]<0.1 and error[1]>-0.1: # error[1]为距离偏差
            error[1]=0

        # when target is little, amplify velocity by amplify error
        # 当目标很小时，通过放大误差来提高速度
        if error[1]>0 and self.setPoint[1]<1.3:
            error[1]=error[1]*(1.3/self.setPoint[1])
        P = error

        currentTime = time.clock()
        deltaT      = (currentTime-self.timeOfLastCall)

        # 误差的积分是 当前误差*时间差
        self.integrator = self.integrator + (error*deltaT)
        I = self.integrator

        # D 是误差的差值 / 时间差
        D = (error-self.last_error)/deltaT

        self.last_error = error
        self.timeOfLastCall = currentTime

        # 返回控制信号
        return self.Kp*P + self.Ki*I + self.Kd*D
```

这是一段更新 PID 控制信号的方法，定义在 simplePID 类中。current_value 是当前小车与目标之间的角度与距离，是一个数组，形式为[angleX distance]即[角度 距离]。error 为一个偏差值，当前角度距离与目标值角度距离的偏差，目标值是已经设定好的[0 targetdist]即[0 中距值]。if error[0]<0.1 and error[0]>-0.1 这段条

件判断语句是为了让小车在误差较小时停止移动，以免造成小车来回摆动。if error[1]>0 and self.setPoint[1]<1.3 当目标值设定较小时适当放大误差来调整小车位置。之后进行 PID 的计算。最后返回控制信号，实现对小车角速度线速度的控制。