视觉跟随功能使用与讲解

1. 功能简介

视觉跟随功能是实现小车对目标颜色的物体进行跟随。该功能基于WHEELTEC 机器人的摄像头硬件，以 ROS 中的 cv_bridge 功能包作为端口将图像信息传输给 OpenCV，OpenCV 赋予了 ROS 十分强大的图像处理能力，可以实现多种视觉处理功能。

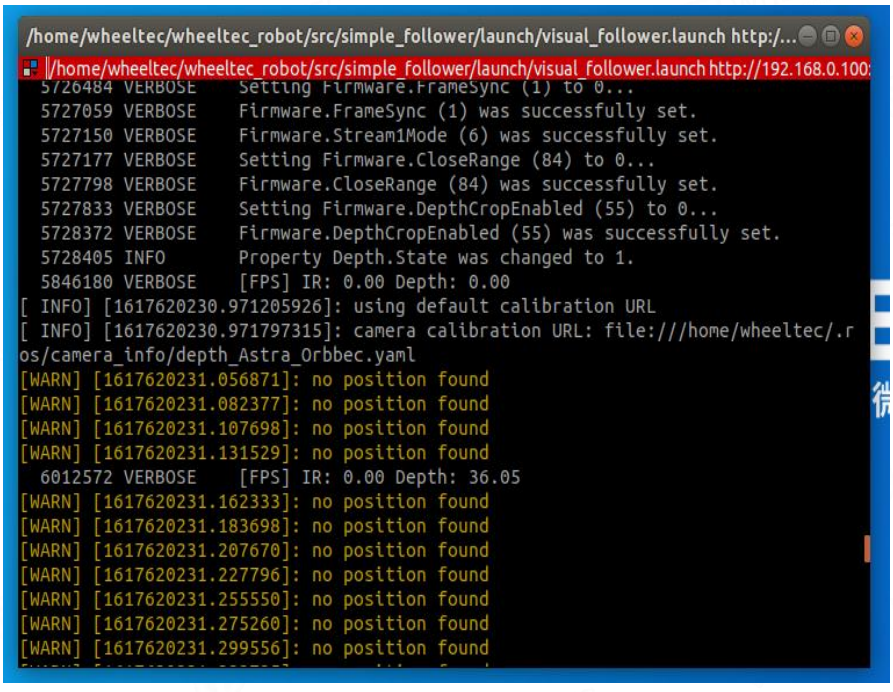
2. 使用方法

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 启动视觉跟随功能

使用 ssh 命令远程登录小车后，运行 simple_follower 功能包中的 visual_follower.launch ,若终端无红色报错则运行成功:

```
roslaunch simple_follower visual_follower.launch
```

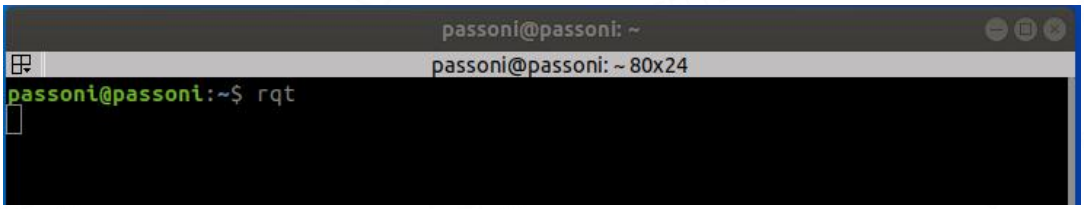


启动节点后的终端信息：小车开始寻找目标物体

执行完命令之后，小车开始寻找目标物体。默认情况下，小车视觉跟随目标为红色物体，小车会通过目标物体的位置进行相关计算获得运动速度，从而实现对目标物体的跟随。

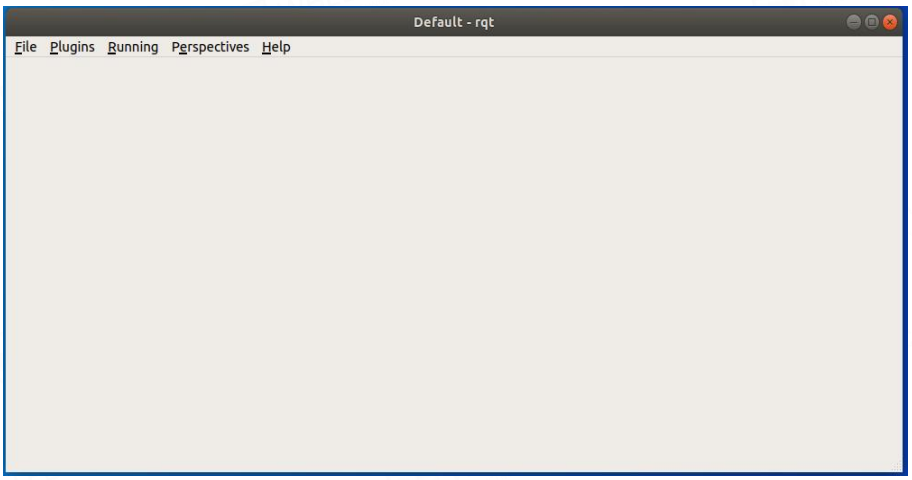
② rqt 在线调参

在成功运行 visual_follower.launch 后，另打开一个新的终端，输入 rqt 命令。（注意这里无需 ssh 远程登录至小车端）



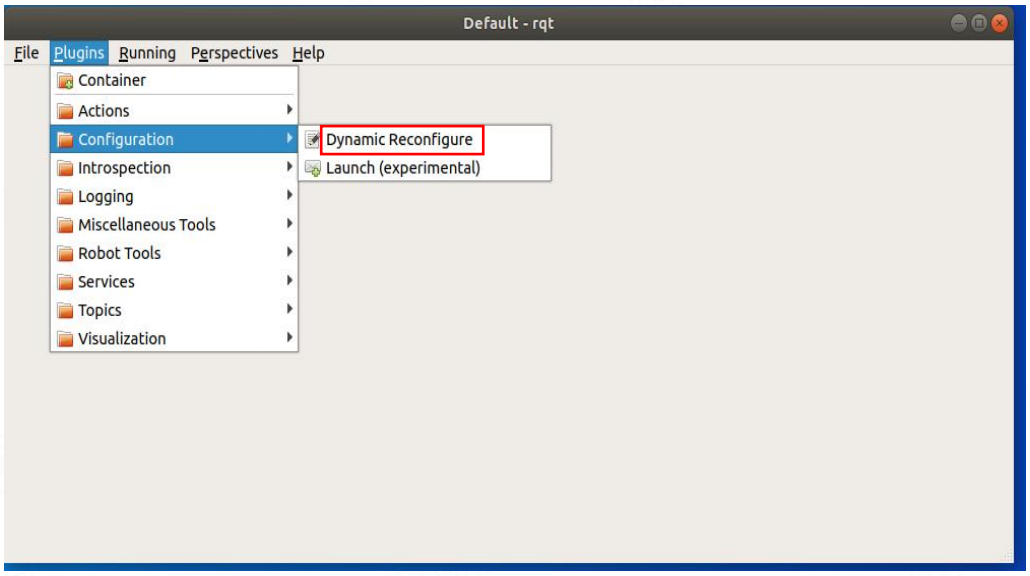
在新的终端输入 rqt

打开 rqt 默认会出现以下界面：



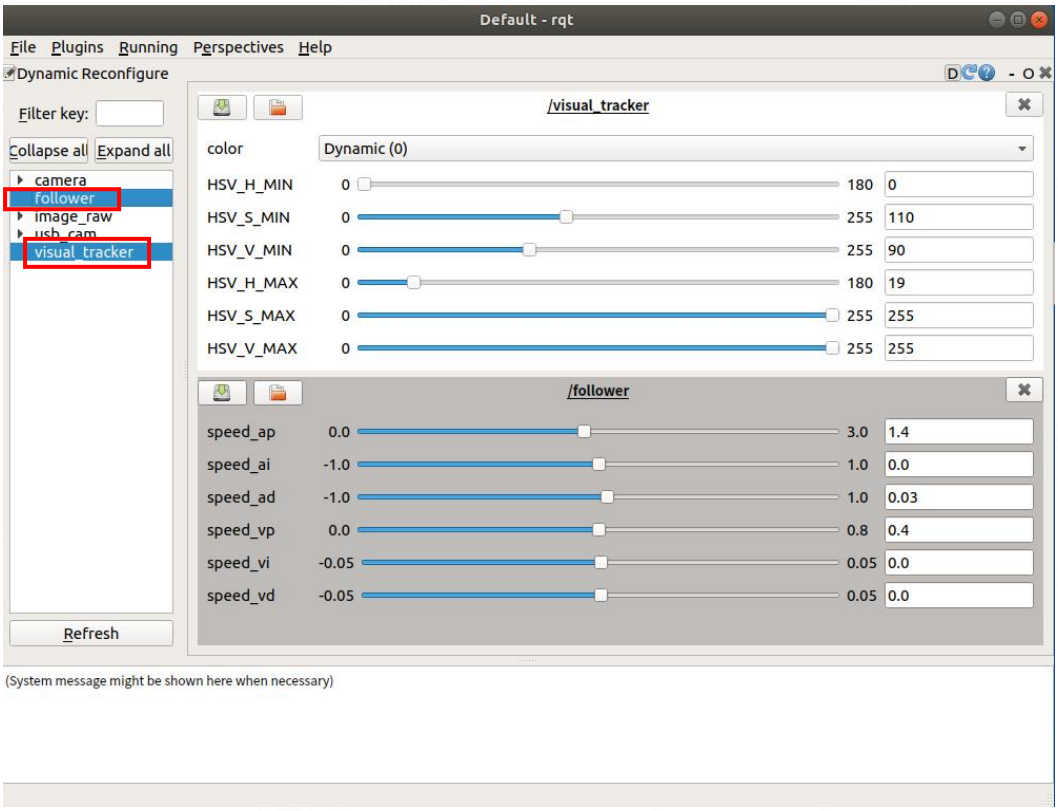
默认状态下的 rqt

之后我们再点击 Plugins 选项，找到 Configuration 选项下的 Dynamic Reconfigure 选项并打开。



打开参数动态配置工具 Dynamic Reconfigure

注意选中 follower 和 visual_tracker 选项，这两个选项打开的界面分别是 follower 节点和 visual_tracker 节点的动态调参，然后我们就可以看到该界面：



动态调参界面

接下来可以根据自己的需要对小车视觉跟随的目标颜色或者速度 PID 进行调整，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。

Follower 节点中需要用到的参数说明如下：

表 1　参数 speed 相关说明

参数名	说明
speed_ap	角速度参数 P
speed_ai	角速度参数 I
speed_ad	角速度参数 D
speed_vp	线速度参数 P
speed_vi	线速度参数 I
speed_vd	线速度参数 D

Visual_tracker 中需要用到的参数说明如下：

表 2　参数 color 对应说明

参数值	0	1	2	3	4
颜色	动态可调	红色	蓝色	绿色	黄色

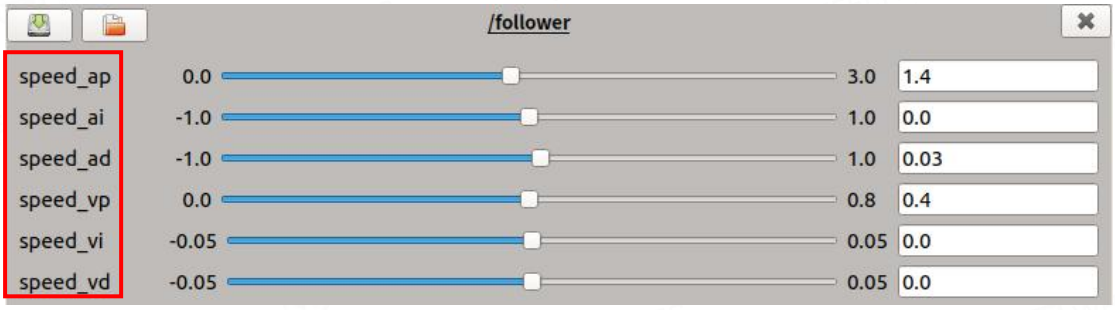
表 3 参数 HSV 相关说明

参数名	说明
HSV_H_MIN	目标物体颜色色调（H）的最小值
HSV_H_MAX	目标物体颜色色调（H）的最大值
HSV_S_MIN	目标物体颜色饱和度（S）的最小值
HSV_S_MAX	目标物体颜色饱和度（S）的最大值
HSV_V_MIN	目标物体颜色明度（V）的最小值
HSV_V_MAX	目标物体颜色明度（V）的最大值

3. 注意事项

① 小车行走问题

如果小车有反复前后行走的情况，可能是因为 PID 参数的大小不够合理，在加入参数动态配置工具（rqt_reconfigure）之后，我们就可以实时对速度 PID 的参数进行调整。在调整速度 PID 参数时，应避免参数设置过大导致小车极度不稳定，从而造成碰撞。同时，开启视觉跟随时，小车并无自主避障的功能，小车活动范围不要有太多杂物，避免碰撞。

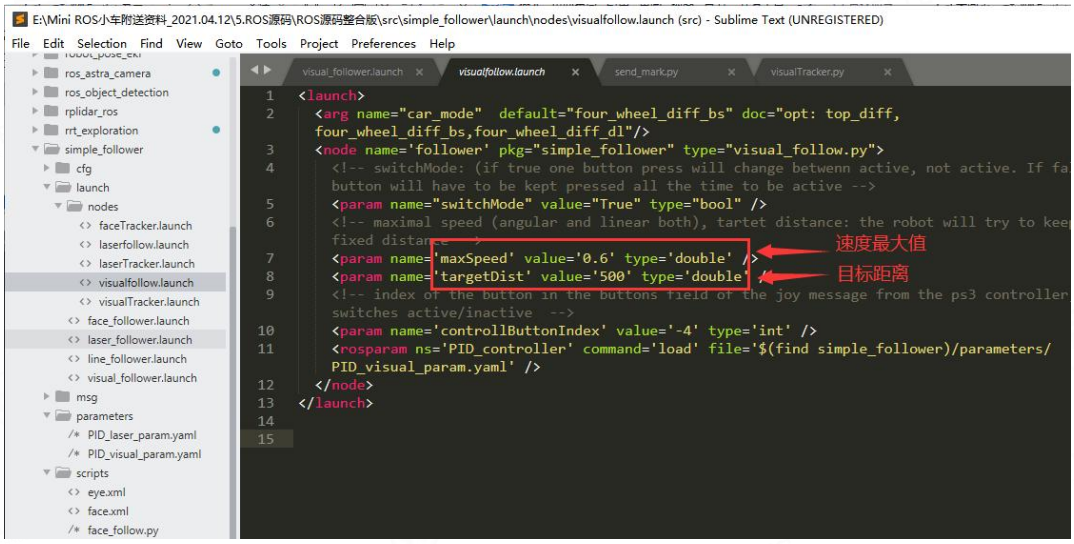


调整速度 PID 参数

② 视觉跟随参数调整

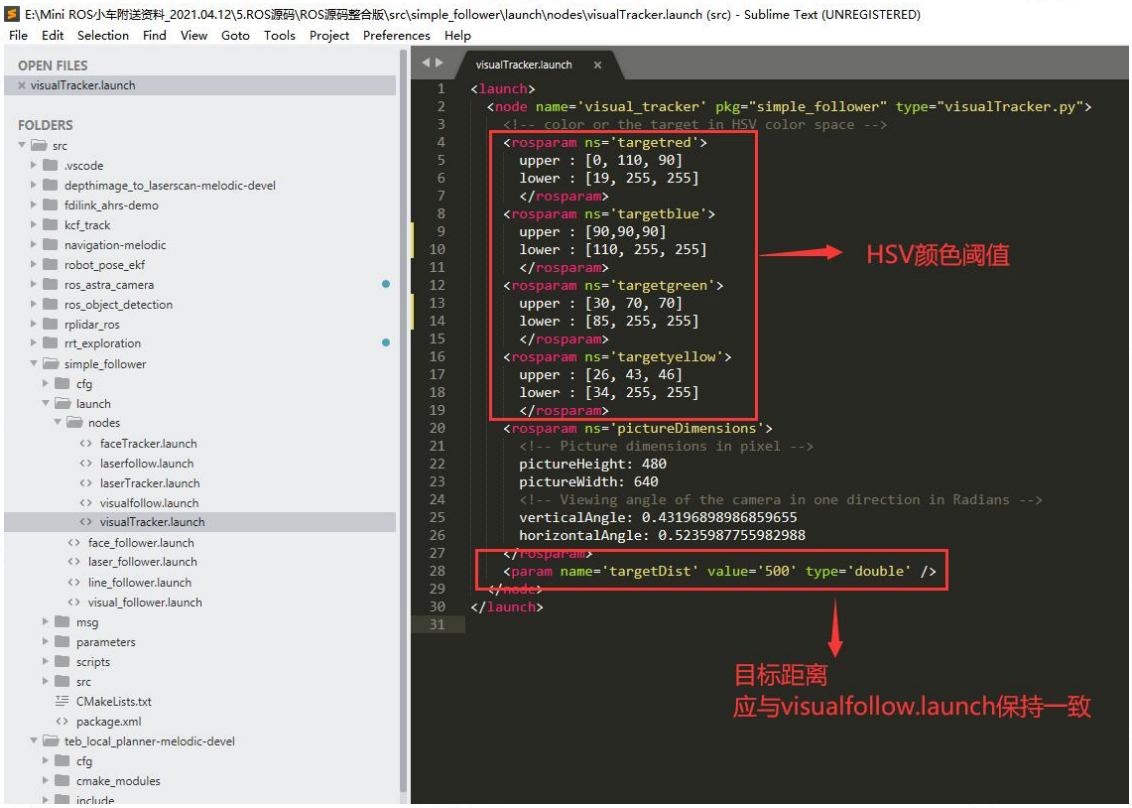
视觉跟随参数调整有两种方式。第一种是直接在节点的 launch 文件中修改 param 即修改参数服务器中的参数，这种方式修改参数后不需要编译，重启节点

后生效。在 visualfollow.launch 文件中，我们可以更改速度最大值 maxSpeed 和目标距离 targetDist 这两个参数，如下图所示。请注意如果不是必要的话请不要随意更改速度最大值和目标距离，在上述代码中已经对速度和目标距离做了严格限制。



follower 节点 launch 文件

在 visualTracker.launch 文件中，我们可以更改不同颜色的颜色阈值参数，如下图所示。



visual_tracker 节点 launch 文件

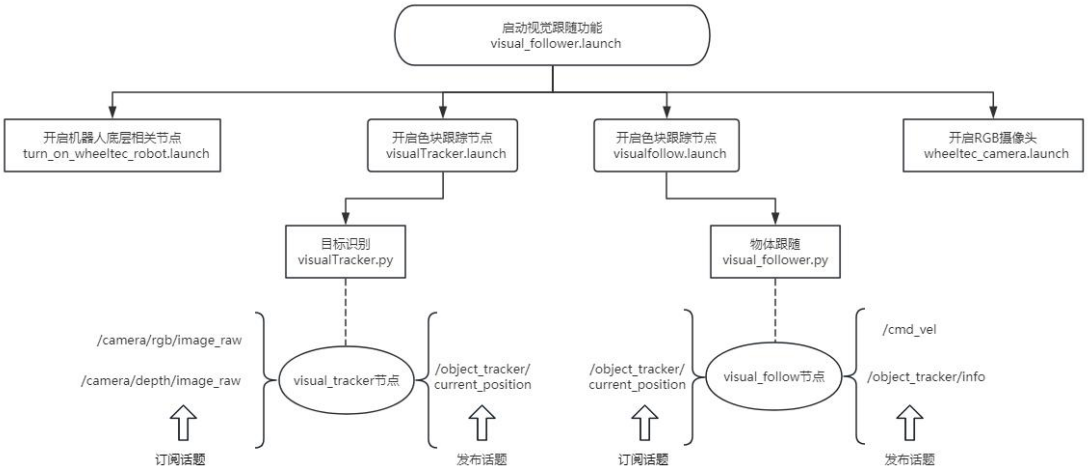
第二种是 rqt 在线调参，可以在节点运行时进行参数调整，修改后的参数只在本次节点运行时生效，因此 rqt 在线调参可以作为一种调试工具。

dynamic_reconfigure 例程：http://wiki.ros.org/dynamic_reconfigure/Tutorials

对于目标物体的颜色设置，我们使用 HSV 模型来对颜色阈值进行设置，一般对于颜色空间的图像进行有效处理都是在 HSV 空间进行的，基本色对应的 HSV 分量需要给定一个严格的范围。

4. 功能讲解

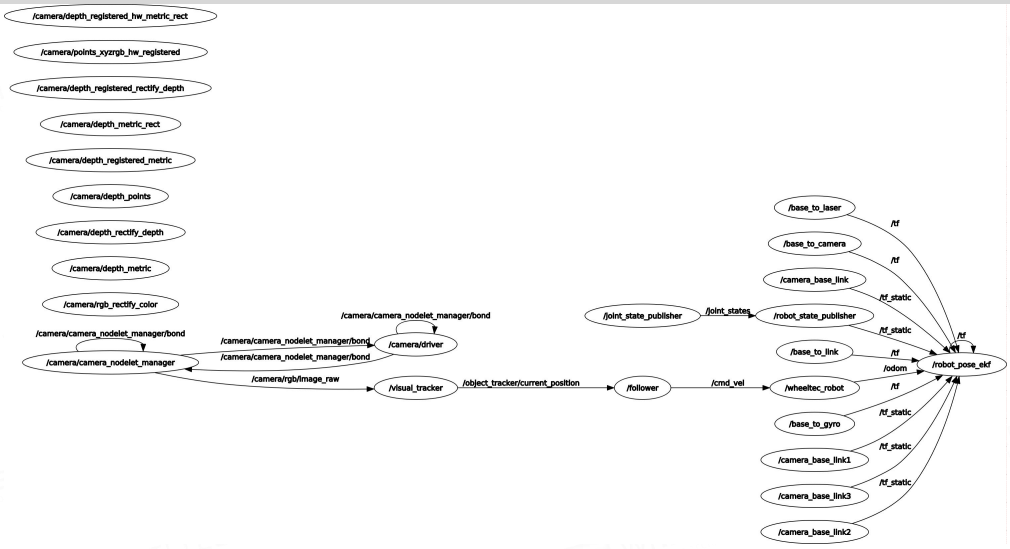
① 视觉跟随功能启动框架



视觉跟随功能启动框架

② 视觉跟随功能节点关系

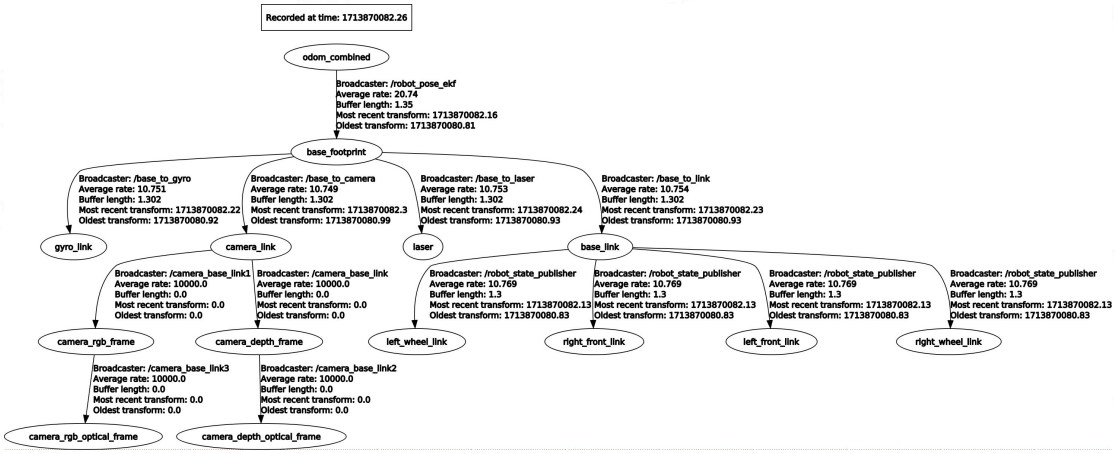
```
rqt_graph
```



视觉跟随功能节点关系图

③ 视觉跟随功能 TF 树

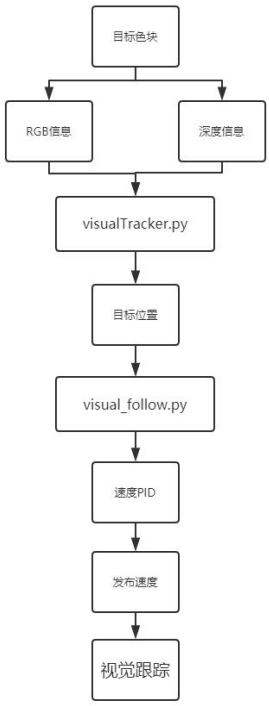
```
rosrun rqt_tf_tree rqt_tf_tree
```



视觉跟随功能 TF 树

④ 视觉跟随功能流程解析

视觉跟随功能通过运行 visual_follower.launch 开启，在 visual_follower.launch 中包含了 5 个 launch 文件。其中 turn_on_wheeltec_robot.launch 用于开启小车的底层运动节点，wheeltec_camera.launch 负责打开 RGB 摄像头和深度摄像头，visualTracker.py 和 visualfollow.py 负责主要功能的实现。其中的 visualTracker.py 负责目标识别，visual_follow.py 负责物体跟随。视觉跟随功能程序流程如下：



视觉跟随流程解析

⑤ 视觉跟随功能 visualTracker.py 文件关键源码解析

我们先从目标识别 visualTracker.py 文件的主要部分讲起：

visual_tracker 节点同时订阅了 RGB 摄像头和深度摄像头的话题，并将两种图像信息传输给回调函数 trackObject。

```
        # message_filters 订阅 RGB 摄像头话题
    im_sub = message_filters.Subscriber('/camera/rgb/image_raw', Image)
    # 订阅过滤器订阅深度摄像头话题
    dep_sub = message_filters.Subscriber('/camera/depth/image_raw', Image)
    # 将 RGB 图像和深度图像信息放入消息过滤器做近似时间同步
    self.timeSynchronizer = message_filters.ApproximateTimeSynchronizer([im_sub,
dep_sub], 10, 0.5)

        #将时间同步后的图像传输给回调函数 trackObject 做图像处理
    self.timeSynchronizer.registerCallback(self.trackObject)
    #发布视觉跟随目标所在位置
    self.positionPublisher = rospy.Publisher('/object_tracker/current_position',
PositionMsg, queue_size=3)
    #动态参数服务器的调用
    self.color_obj = Server(Params_colorConfig,self.colorreconfigure)
```

关于 message_filters 的官方使用例程：https://wiki.ros.org/message_filters

message_filters.Subscriber 是对 ROS 订阅的封装，为其他过滤器提供源代码，message_filters.Subscriber 无法将另一个过滤器的输出作为其输入，而是使用 ROS 话题作为其输入。

TimeSynchronizer 过滤器通过包含在其中的时间戳来同步输入通道，并以单个回调的形式输出它们，而 ApproximateTimeSynchronizer 与 TimeSynchronizer 类似只多了一个延时参数。

将图像信息传输给回调函数 trackObject 之后，回调函数会做具体的图像处理并且找到跟随目标所在的位置，下面就让我们来看看回调函数 trackObject 中的主要内容：

```
        #将 RGB 图像信息传输到 OpenCV 中
    frame = self.bridge.imgmsg_to_cv2(image_data, desired_encoding='rgb8')
    #将深度图像信息传输到 OpenCV 中
    depthFrame = self.bridge.imgmsg_to_cv2(depth_data,
desired_encoding='passthrough')
    #将 RGB 图像信息转换为 hsv 图像信息
    hsv = cv2.cvtColor(frame, cv2.COLOR_RGB2HSV)
```

```
    #然后利用 cv2.inRange 函数设阈值，去除背景部分，只保留目标颜色部分
    org_mask = cv2.inRange(hsv, self.targetUpper, self.targetLower)
    #将得到的目标颜色部分再进行形态学腐蚀，去除图像噪点和干扰
    mask = cv2.erode(org_mask, None, iterations=4)
    #检测图像轮廓，只取轮廓的返回值
        contours = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)[-2]
    newPos = None
    try:
        # 将所有轮廓进行排序并获得面积最大的轮廓作为目标
        contour = sorted(contours, key=cv2.contourArea, reverse=True)[0]
            # 获取目标轮廓的中心位置和距离
        pos = self.analyseContour(contour, depthFrame)
        if newPos is None:
            newPos = pos
            self.lastPosition = pos
        self.publishPosition(pos)
        self.lastPosition = newPos
    # 如果数组下标出现错误打印警告信息
    except IndexError:
        rospy.logwarn('no position found')
            # 小车停止移动
        posMsg = PositionMsg(0, 0,self.targetDist)
        self.positionPublisher.publish(posMsg)
```

最后回调函数 trackObject 将得到的目标位置发送到当前位置的话题中，之后就由 visual_follow 来进行目标物体跟随。如果没有找到目标轮廓的话，就发布目标位置来替代当前位置，使我们的机器人能够在找不到目标的情况下停止移动。

⑥ 视觉跟随功能 visual_follow.py 文件关键源码解析

visual_follow 节点主要是订阅 visual_tracker 处理图像所得到的目标位置话题，再经过 PID 速度控制后，发布小车速度话题的命令。

让我们来看看物体跟随 visual_follow.py 文件的主要部分：

```
    # 发布速度话题
        self.cmdVelPublisher = rospy.Publisher('/cmd_vel', Twist, queue_size =3)
    # 订阅目标跟随的位置话题，它为我们提供了所跟随对象的当前位置
    self.positionSubscriber =
rospy.Subscriber('/object_tracker/current_position', PositionMsg,
self.positionUpdateCallback)
    # 订阅目标跟随的位置消息提示
    self.trackerInfoSubscriber = rospy.Subscriber('/object_tracker/info',
StringMsg, self.trackerInfoCallback)
    # 调用动态参数服务器
```

```
self.speed_PID = Server(Params_PIDConfig,self.followreconfigure)
```

visual_follow 节点首先订阅了目标跟随的当前位置话题，并且将数据消息传输给回调函数 PositionMsg，在回调函数中通过对速度 PID 控制函数 update 的调用来处理位置信息，并且返回我们所需要的速度值。

下面我们再来看一下速度 PID 控制的分析：

```
#误差计算
    error = self.setPoint - current_value
    #当误差较小时，停止运动；这里可以根据实际情况调节
    if error[0]<0.1 and error[0]>-0.1:
        error[0]=0
if error[1]<100 and error[1]>-100:
        error[1]=0
    # 当距离值较小时，通过放大误差来调节速度，放大系数可根据需要进行调节
if (error[1]>0 and self.setPoint[1]<1200):
        error[1]=error[1]*(1200/self.setPoint[1])*0.7
    # 计算 PID 参数，并返回计算的速度值
P =  error
    currentTime = time.clock()
    deltaT      = (currentTime-self.timeOfLastCall)
    self.integrator = self.integrator + (error*deltaT)
I = self.integrator
    D = (error-self.last_error)/deltaT
    self.last_error = error
    self.timeOfLastCall = currentTime
return  self.Kp*P + self.Ki*I + self.Kd*D
```

这里的 PID 控制过程是非常经典的位置 PID 控制，如果有不懂的同学可以去看我们的 STM32 电机控制例程。PID 控制函数的返回值会进入到目标位置话题的返回函数 positionUpdateCallback 中，返回函数将速度消息处理后再发布到速度话题中，从而使的小车接收速度消息。最后让我们看一下返回函数 positionUpdateCallback 的主要内容：

```
    angleX= position.angleX
distance = position.distance
# 将位置信息传输给 update 函数做 PID 控制，并且返回速度值
[uncliped_ang_speed, uncliped_lin_speed] =self.update([angleX, distance])
    # 将这些速度调整到指定的速度范围内
angularSpeed = np.clip(-uncliped_ang_speed, -self.max_speed, self.max_speed)
linearSpeed  = np.clip(-uncliped_lin_speed, -self.max_speed, self.max_speed)
# 发送速度消息
velocity = Twist()
velocity.linear = Vector3(linearSpeed,0,0.)
```

```
    velocity.angular= Vector3(0., 0.,angularSpeed)
    self.cmdVelPublisher.publish(velocity)
```

最终，小车接收速度消息后，就可以不断接近跟随目标，并与目标物体保持设置的距离，以达到想要的效果。