



目录 CONTENTS

第一节

编写ROS工程
使机器人移动

第二节

gmapping建图
与自主导航

第三节

使用摄像头识别
颜色并执行动作

第四节

gmapping建图
与自主导航

第五节

奥比中光摄像头
测试与使用方法



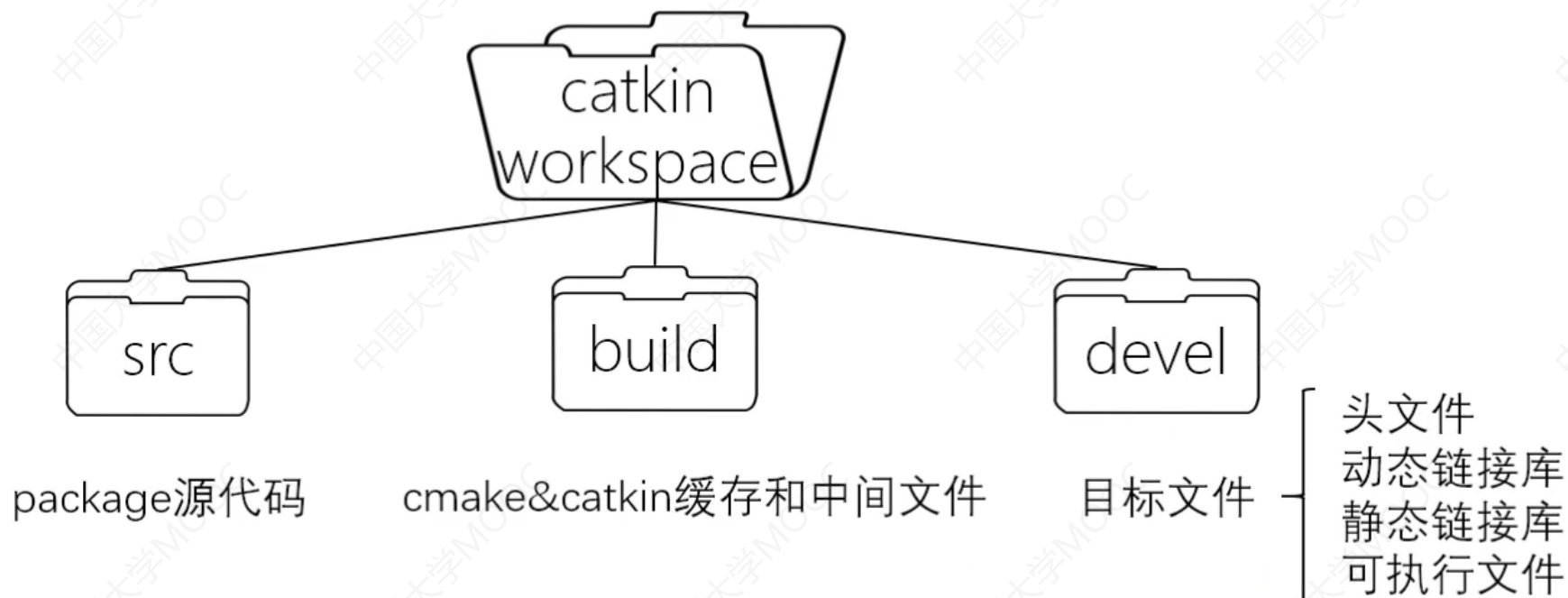
第一节

编写ROS工程使机器人移动



1 ROS工程的基本结构

ROS工程的基本结构，如下图：

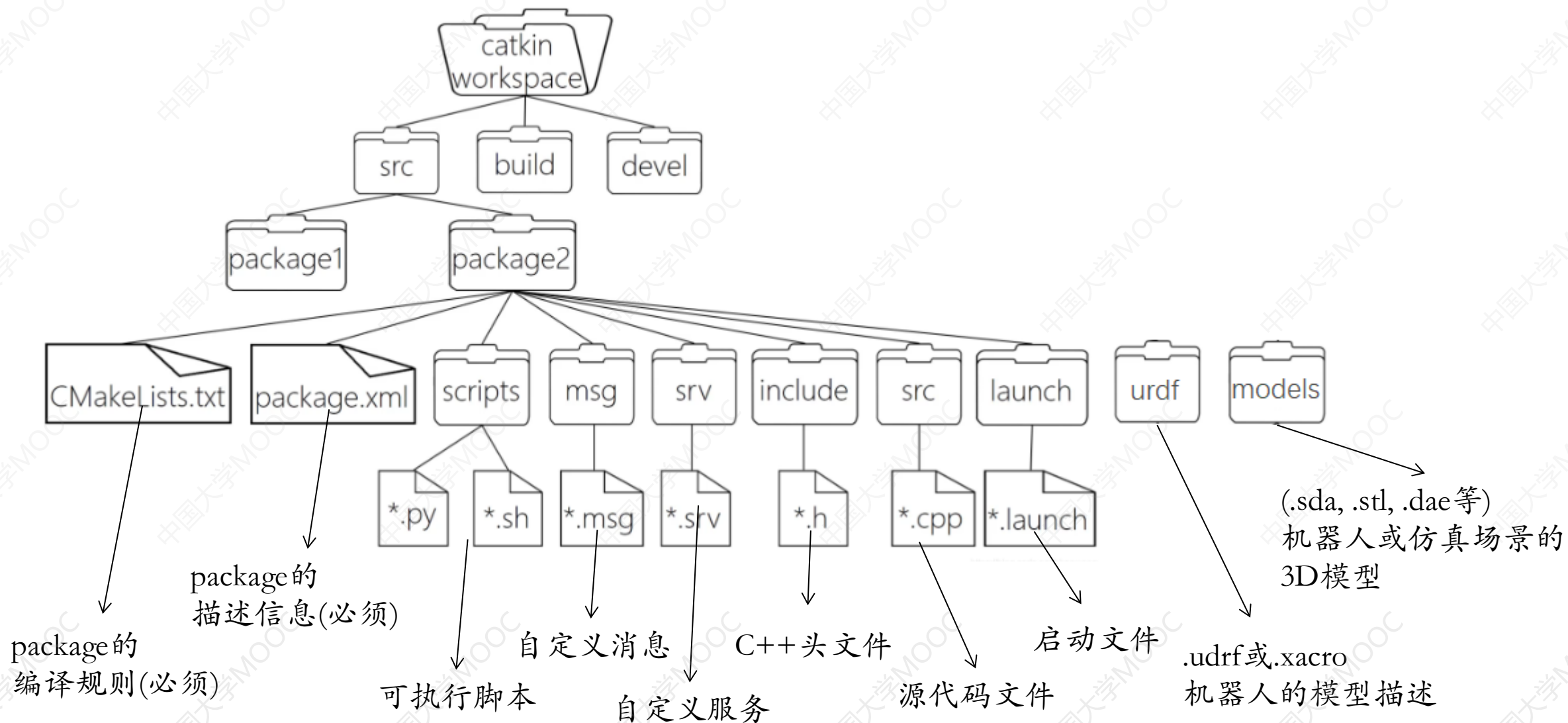


- 1、catkin_workspace是用户创建的工程文件目录（可以任意命名），包含整个工程。
- 2、src目录是用户创建的功能包集合（不能修改名称），包含一个或多个功能包。
- 3、使用catkin_make指令编译之后，将生成build目录和devel目录。



2 ROS功能包的基本结构

ROS功能包的基本结构，如下图：





3 package.xml文件介绍

新建一个ros工程与ros功能包，新建功能包的指令：

catkin_create_pkg 包名 ros库1 ros库2...

```
cd
mkdir test_ws/src -p
cd test_ws/src
catkin_create_pkg test1 roscpp
cd test1
```

可以看到，在test1包下自动生成了CMakeLists.txt文件和package.xml文件，其中package.xml去掉注释后如右图所示。

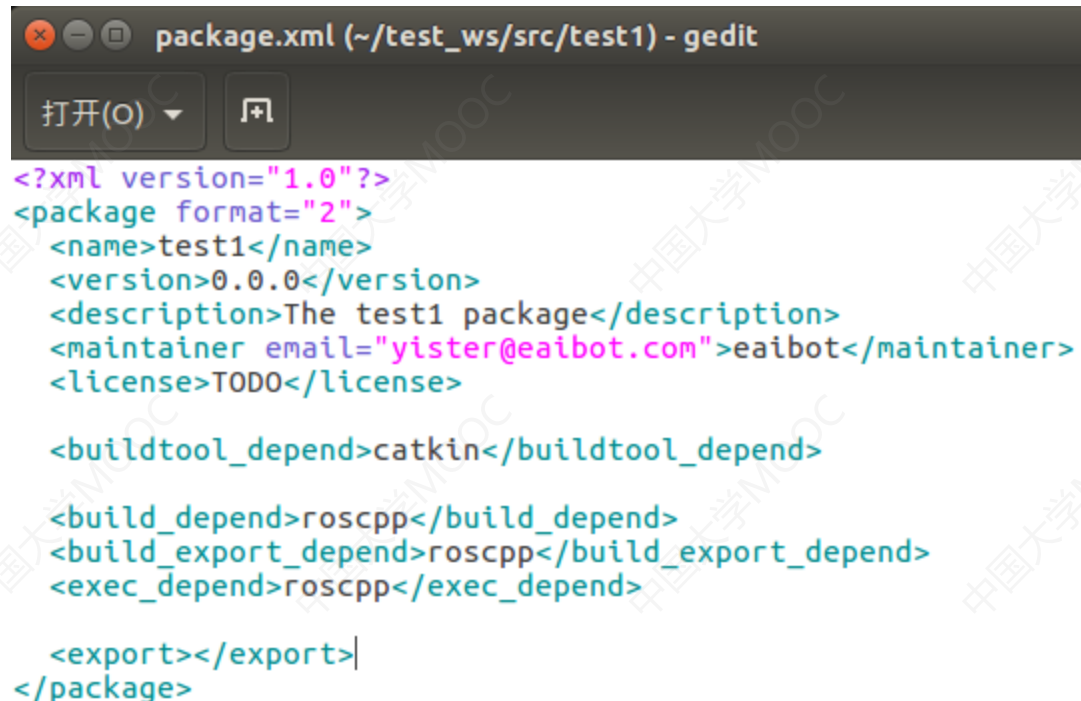
如右上图，package.xml文件**必须**要包含name, version, description, maintainer, license 5个属性

由于编译代码要用到catkin_make工具，所以其中还需要加入对catkin工具的依赖：

buildtool_depend 编译构建工具,通常为catkin。

由于代码中使用到了roscpp的头文件，所以还需要加入对roscpp的依赖：

build_depend 编译依赖项、build_export_depend 导出依赖项、exec_depend 运行依赖项。



```
package.xml (~/test_ws/src/test1) - gedit
打开(O)  [icon]
<?xml version="1.0"?>
<package format="2">
  <name>test1</name>
  <version>0.0.0</version>
  <description>The test1 package</description>
  <maintainer email="yister@eaibot.com">eaibot</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <exec_depend>roscpp</exec_depend>

  <export></export>
</package>
```



4.1 CMakeLists.txt文件介绍

如右下图，CMakeLists.txt文件必须包含：cmake_minimum_required(版本号), project(包名), find_package(catkin REQUIRED), catkin_package() 结构。

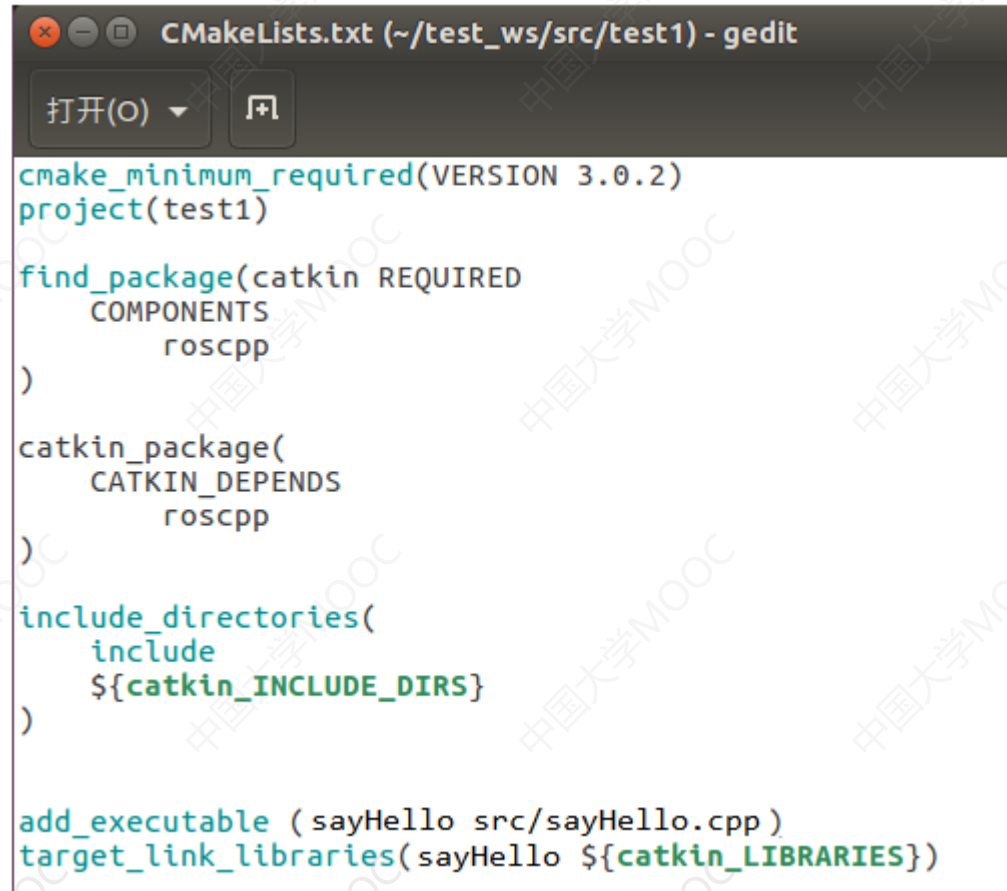
①.用find_package指令来指定在构建项目过程中依赖了哪些其他的包，在ROS中，catkin必备依赖，所以雷打不打地我们写上catkin REQUIRED，在此基础上，如果我们还需要依赖其他包或者组件，我们就在 上述的包后面继续添加包或组件名即可：

```
find_package(catkin REQUIRED COMPONENTS nodelet)
```

如果CMake通过find_package找到一个包，则会自动生成有关包所在路径的CMake环境变量，环境变量描述了包中头文件的位置，源文件的位置，包所依赖的库以及这些库的路径。

由于后文的代码中用到了ros的头文件，所以我们需要在find_package中加入COMPONENTS roscpp。

并添加include_directories(include \${catkin_INCLUDE_DIRS})



```
cmake_minimum_required(VERSION 3.0.2)
project(test1)

find_package(catkin REQUIRED
             COMPONENTS
               roscpp
)

catkin_package(
  CATKIN_DEPENDS
    roscpp
)

include_directories(
  include
    ${catkin_INCLUDE_DIRS}
)

add_executable (sayHello src/sayHello.cpp)
target_link_libraries(sayHello ${catkin_LIBRARIES})
```



4.2 CMakeLists.txt 文件介绍

②. 用 `catkin_package` 指令将 catkin 特定的信息输出到构建系统上，用于生成 pkg 配置文件以及 CMake 文件。一个简单的 `catkin_package` 通常包含 `CATKIN_DEPENDS` 该项目依赖的其他 catkin 项目，比如

```
catkin_package( CATKIN_DEPENDS roscpp )
```

③. 如果需要编译某个 c++ 文件作为可执行文件，则需使用 `add_executable` 和 `target_link_libraries` 指定。

使用 `add_executable` 指定要构建的可执行目标；`target_link_libraries()` 来指定可执行目标链接的库，通常在 `add_executable()` 调用之后完成，如果需要找到 ros 库，则添加 `${catkin_LIBRARIES}`。例如：

```
add_executable (sayHello src/sayHello.cpp)
```

```
target_link_libraries (sayHello ${catkin_LIBRARIES})
```




5 编写简单的代码并编译

在test1/src目录下创建文件sayHello.cpp，写如下代码（其中使用到了ROS库的日志打印函数）

```
#include "ros/ros.h"
int main(){
    ROS_WARN("Hello");
}
```

由于要编译这个源码文件成为可运行的程序，所以需要如上一节所说，配置add_executable和target_link_libraries。然后在test_ws工程的目录下，编译整个工程包。

```
cd ~/test_ws
sudo chmod 777 src -R          #一键给文件读写、执行权限
catkin_make                    #编译

#配置工作空间，刷新环境变量。也可以将配置直接加到~/.bashrc文件末尾
source devel/setup.bash

#运行程序
roslaunch test1 sayHello
```

正常编译的node，都会在对
应的工程目录的devel/lib/对
应package/下生成node文件



6.2 编写简单程序控制底盘运动

接上节

```
geometry_msgs::Twist twist;
geometry_msgs::Vector3 linear;
linear.x = 0;
linear.y = 0;
linear.z = 0;
geometry_msgs::Vector3 angular;
angular.x = 0;
angular.y = 0;
angular.z = 0.1;
twist.linear = linear;
twist.angular = angular;
ros::Rate rate(10); //发布频率，一秒10次。
while(ros::ok())
{
    pub.publish(twist);
    rate.sleep(); //根据上面的设置，这里就是睡眠100ms的意思
}
```



6.3 编写简单程序控制底盘运动

修改CMakeLists.txt文件，find_package和catkin_package新增geometry_msgs依赖，新增指定编译文件，如下所示：

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  geometry_msgs
)

catkin_package(
  CATKIN_DEPENDS roscpp geometry_msgs
)

#...

add_executable (cmd_test src/cmd_test.cpp)
target_link_libraries (cmd_test ${catkin_LIBRARIES})
```

修改package.xml文件，新增geometry_msgs依赖，如下所示：

```
<build_depend>geometry_msgs</build_depend>
<build_export_depend>geometry_msgs</build_export_depend>
<exec_depend>geometry_msgs</exec_depend>
```



6.4 编写简单程序控制底盘运动

编译(并source)，并运行，可以看到底盘开始原地旋转。

```
#先打开一个终端，将driver驱动程序启动
roslaunch dashgo_driver driver_imu.launch
#再打开一个终端，编译并运行程序
cd ~/test_ws
catkin_make
source devel/setup.bash
roslaunch test1 cmd_test
```



第二节

gmapping建图与自主导航



1.1 建图用到的功能包

首先从建图的launch文件开始分析。

```
<launch>
  <include file="$(find dashgo_driver)/launch/driver_imu.launch"/>
  <include file="$(find ydlidar)/launch/ydlidar1_up.launch"/>
  <include file="$(find dashgo_description)/launch/dashgo_description.launch"/>
  <include file="$(find dashgo_nav)/launch/include/imu/gmapping_base.launch"/>
  <include file="$(find dashgo_nav)/launch/include/imu/teb_move_base.launch"/>
  <node name="robot_pose_publisher" pkg="robot_pose_publisher" type="robot_pose_publisher" />
  <node name="robot_data" pkg="dashgo_tools" type="robot_data.py" respawn="true" />
  <node name="scan_cloud" pkg="dashgo_tools" type="get_scan_data" respawn="true" />
</launch>
```

1. driver_imu.launch (在dashgo_driver功能包中)中启动了dashgo_driver.py (和底层STM32控制板通信)，几个tf关系 (imu坐标系、超声波坐标系与机器人坐标的相对关系)，robot_pose_ekf卡尔曼滤波算法包 (融合编码器数据和陀螺仪数据得到融合后的里程计数据)，yocs_velocity_smoother速度平滑包，nodelet_manager进程 (实现零拷贝)。



1.2 建图用到的功能包

2. `ydliidar1_up.launch`（在`ydliidar`功能包中）中启动了`ydliidar_node.cpp`（从串口解析雷达数据），`tf`关系（雷达坐标系与机器人坐标系的相对关系）。
3. `dashgo_description.launch`（在`dashgo_description`功能包中）中启动了机器人的模型文件，确定了机器人的基础坐标系。
4. `gmapping_base.launch`（在`dashgo_nav`功能包中）中启动了`gmapping`算法包。
5. `teb_move_base.launch`（在`dashgo_nav`功能包中）中启动导航相关功能包（后面再讲），以支持边建图边导航。
6. `robot_pose_publisher`功能包，仅发布机器人基于地图的坐标信息，和建图没有什么关系，无需关注。
7. `robot_data.py` 和建图没有什么关系，无需关注，是服务端用来记录状态的。
8. `get_scan_data.cpp`（在`dashgo_tools`功能包中）启动了一个雷达数据转换，和建图也没有关系，给客户端显示用的。



2.1 建图的步骤

可以查看是否有ros进程，有冲突的进程，可以用kill 进程id 杀掉进程。

```
ps -aux | grep ros
```

#确保没有冲突的进程后，启动建图

```
roslaunch dashgo_nav gmapping_imu.launch
```

3.再打开一个终端窗口，启动rviz工具

```
export ROS_MASTER_URI=http://localhost:11311 #可以添加到~/.bashrc文件中以避免手动输入
```

```
roslaunch dashgo_rviz view_navigation.launch
```



2.2 建图的步骤

4.再打开一个终端窗口，启动键盘控制脚本。

```
roslaunch dashgo_tools teleop_twist_keyboard.py
```

#用键盘控制底盘移动，i前进，j后退，l左转，r右转。

#可以在步骤3的rviz工具中观察建图的实况。

#建图完成后，我们在本窗口调用以下指令保存地图

```
roscd dashgo_nav/maps      #进入存放地图的目录
```

```
roslaunch map_server map_saver -f eai_map_imu      #保存地图
```



3.1 导航用到的功能包

首先从导航的launch文件开始分析。

```
<launch>
  <include file="$(find dashgo_driver)/launch/driver_imu.launch"/>
  <include file="$(find ydlidar)/launch/ydlidar1_up.launch"/>
  <include file="$(find dashgo_description)/launch/dashgo_description.launch"/>
  <arg name="map_file" default="$(find dashgo_nav)/maps/eai_map_imu.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />
  <roscparam file="$(arg map_file)" command="load" ns="map" />
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find dashgo_nav)/launch/include/imu/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>
```



3.2 导航用到的功能包

接上节

```
<include file="$(find dashgo_nav)/launch/include/imu/teb_move_base.launch"/>
<node name="robot_pose_publisher" pkg="robot_pose_publisher" type="robot_pose_publisher" />

<node pkg="laser_filters" type="scan_to_scan_filter_chain" output="screen" name="laser_filter">
  <rosparam command="load" file="$(find dashgo_tools)/conf/box_filter.yaml" />
</node>
<node pkg="laser_filters" type="scan_to_scan_filter_chain" output="screen" name="laser_filter_2">
  <rosparam command="load" file="$(find dashgo_tools)/conf/box_filter_2.yaml" />
  <remap from="scan" to="scan_2" />
  <remap from="is_passed" to="is_passed_2" />
  <remap from="scan_filtered" to="scan_filtered_2" />
</node>
<node name="multiGoals" pkg="dashgo_tools" type="multiGoals.py" respawn="true" />
<node name="scan_cloud" pkg="dashgo_tools" type="get_scan_data" respawn="true" />
</launch>
```



3.3 导航用到的功能包

1. driver_imu.launch 同建图
2. ydlidar1_up.launch 同建图
3. dashgo_description.launch 同建图。
4. map_server 功能包，加载地图到内存中以供导航使用。
5. amcl.launch.xml（在dashgo_nav包中）中启动了amcl自适应蒙特卡洛定位 算法包。
6. teb_move_base.launch（在dashgo_nav包中）中启动了move_base功能包，这个是核心的导航算法包，它将规划出前往特定坐标点的全局路径和局部路径。
7. robot_pose_publisher 同建图。
8. scan_to_scan_filter_chain 节点（在laser_filters功能包中），用来规定一个特定的矩形范围，判断雷达数据是否出现在该矩形范围内，从而决定是否需要紧急刹车。
9. multiGoals.py（在dashgo_tools包中）和导航算法没有什么关系，是服务端用来记录状态、目标点、任务的，主要是用来执行客户端创建的导航任务的，无需关注。
10. get_scan_data.cpp 同建图。



4 导航的步骤

1.启动导航。

#可以查看是否有ros进程，有冲突的进程，可以用kill 进程id 杀掉进程。

```
ps -aux | grep ros
```

#确保没有冲突的进程后，启动导航

```
roslaunch dashgo_nav navigation_imu.launch
```

2.再打开一个终端窗口，启动rviz工具

```
export ROS_MASTER_URI=http://localhost:11311 #可以添加到~/.bashrc文件中以避免手动输入
```

```
roslaunch dashgo_rviz view_navigation.launch
```

在rviz界面中，点击2DPose Estimate，然后在地图上设置底盘当前位置，观察激光点云和地图轮廓大致匹配（位置正确）；然后点击2D Nav Goal，在地图上点击（并拖拽选定方向）设置目标点，底盘就会自主规划出合理的全局路径和局部路径，并移动到目标位置。



第三节

使用摄像头识别颜色并执行动作



1 使用摄像头识别颜色并执行动作

Demo代码分析具体看代码中的注释。

测试步骤如下：

#先打开一个终端，将driver驱动程序启动

```
roslaunch dashgo_driver driver_imu.launch
```

#再打开一个终端，编译并运行程序

```
cd ~/test_ws
```

```
catkin_make
```

```
source devel/setup.bash
```

```
roslaunch test1 color_test
```

#红后退，绿前进，黄左转，蓝右转。拿打印好的各种颜色的纸放到Zed摄像头前方（拿近一点），底盘根据颜色执行动作。

#ps：注意不要在红/绿色衣裤同学旁边测试



第四节

ZED摄像头测试与使用方法



1.1 ZED摄像头测试与使用方法

1.编译好test_ws工程，运行测试程序。

启动一个rosmaster

roscore

#再打开一个终端，运行测试程序

roslaunch test_ws zed_test

#再打开一个终端，运行rqt_image_view工具，选择/zed_img节点（不查看图像也可以直接用rostopic看节点是否有数据），查看图像

roslaunch rqt_image_view rqt_image_view

2.Demo代码分析

```
#include "ros/ros.h"
```

```
#include <sensor_msgs/Image.h>
```

```
#include <cv_bridge/cv_bridge.h> //需要在CMakeLists.txt文件中find_package中添加cv_bridge
```

```
#include <opencv2/opencv.hpp>
```

```
using namespace cv; //可以直接引用cv的函数和变量
```



1.2 ZED摄像头测试与使用方法

接上页

```
int main(int argc, char** argv)
{
    ros::init(argc , argv , "zed_test"); //进行节点初始化
    ros::NodeHandle nh; //创建节点句柄
    ros::Publisher pub = nh.advertise<sensor_msgs::Image>("zed_img", 10);
    VideoCapture capture;
    capture.open(0); //打开 zed 相机
    if (!capture.isOpened())
    {
        printf("摄像头没有正常打开, 重新插拔工控机上当摄像头\n");
        return 0;
    }
    Mat frame; //当前帧图片
```



1.3 ZED摄像头测试与使用方法

接上页

```
while (ros::ok())
{
    capture.read(frame); //读取摄像头图像到frame中
    if(frame.empty())
    {
        break;
    }
    sensor_msgs::Image img;
    //将cv::Mat类型数据转成sensor_msgs::Image类型数据
    cv_bridge::CvImage(std_msgs::Header(), "bgr8", frame).toImageMsg(img);
    pub.publish(img); //发布数据
}
}
```

这样我们就将实现了读取zed摄像头数据发送到ros话题中的功能。



第五节

奥比中光摄像头测试与使用方法



1 奥比中光摄像头测试与使用方法

1.运行奥比中光摄像头launch文件，观察rgb图像数据和深度点云数据。

```
roslaunch astra_launch astra.launch  
#观察无红色报错信息  
rostopic echo /camera/rgb/image_raw #观察rgb图像数据是否有变化  
rostopic echo /camera/depth/points #观察深度点云数据是否有变化
```

如果rgb图像数据和深度点云数据有变化，则说明奥比中光摄像头的拍摄和深度测距功能能够正常使用。

如果要进一步观察图像质量，可以在电脑上执行如下指令观察图像。

```
export ROS_MASTER_URI=http://localhost:11311 #可以添加到~/.bashrc文件中以避免手动输入  
roslaunch rqt_image_view rqt_image_view #在界面中数据列表中选择/camera/rgb/image_raw，即可看到图像
```

2.奥比中光的使用方法，如上，启动astra.launch文件后，rgb图像或者深度点云数据会被发布到ros的话题中，我们可以订阅对应话题的数据，供自己的程序使用。