# ROS概述与环境搭建

## ROS安装

<http://wiki.ros.org/cn/ROS/Installation>

构建依赖rosdep失败：

六部工坊：https://www.bilibili.com/video/BV1zv411E7G6?p=5

rosdepc：<https://zhuanlan.zhihu.com/p/398754989>

安装

sudo pip install rosdepc

如果显示没有pip可以试试pip3

sudo pip3 install rosdepc

如果pip3还没有

sudo apt-get install python3-pip

sudo pip install rosdepc

使用

sudo rosdepc init

rosdepc update

## 测试 ROS

1首先启动三个命令行(ctrl + alt + T)

2命令行1键入:roscore

3命令行2键入:rosrun turtlesim turtlesim\_node(此时会弹出图形化界面)

4命令行3键入:rosrun turtlesim turtle\_teleop\_key

## ROS快速体验

### 1.HelloWorld实现简介

**1.创建工作空间并初始化**

mkdir -p 自定义空间名称/src

cd 自定义空间名称

catkin\_make

**2.进入 src 创建 ros 包并添加依赖**

cd src

catkin\_create\_pkg 自定义ROS包名 roscpp rospy std\_msgs

**catkin\_make 只编译一个包：**

**catkin\_make -DCATKIN\_WHITELIST\_PACKAGES="package1;package2"**

**使用后需catkin\_make -DCATKIN\_WHITELIST\_PACKAGES=""才能编译所有功能包**

### 2.HelloWorld(C++版)

1. **进入 ros 包的 src 目录编辑源文件**

cd 自定义的包/src

1. **C++源码实现(文件名自定义gedit hello.cpp)**

#include "ros/ros.h"

//#include <iostream>

int main(int argc, char \*argv[])

{

//执行 ros 节点初始化

ros::init(argc,argv,"hello");

//创建 ros 节点句柄(非必须)

ros::NodeHandle n;

//控制台输出 hello world

ROS\_INFO("hello world!");//std::cout << "hello ros" << std::endl;

return 0;

}

1. **编辑 ros 包下的 Cmakelist.txt文件**

#生成可执行目标文件

add\_executable(上一步骤的源文件名（hello） src/上一步骤的源文件名.cpp)

#链接

target\_link\_libraries(上一步骤的源文件名 ${catkin\_LIBRARIES})

1. **进入工作空间目录并编译**

cd 自定义空间名称

catkin\_make

#生成 build devel ....

1. **执行**

先启动命令行1：roscore

**再启动命令行2：**

**cd 工作空间**

**source ./devel/setup.bash（source命令用于执行文本文件中的一连串指令）**

**rosrun 包名 C++节点（hello）**

**为了方便可以：echo "source ~/工作空间/devel/setup.bash" >> ~/.bashrc**

**~/.bashrc为终端程序的初始化脚本**

### 3.HelloWorld(Python版)

1. **进入 ros 包添加 scripts 目录并编辑 python 文件**

cd ros包

mkdir scripts

1. **新建 python 文件: (文件名自定义)**

#! /usr/bin/env python

import rospy

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("Hello")

rospy.loginfo("Hello World!!!!")

1. **为 python 文件添加可执行权限**

chmod +x 自定义文件名.py

1. **编辑 ros 包下的 CamkeList.txt 文件**

catkin\_install\_python(PROGRAMS scripts/自定义文件名.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

1. **进入工作空间目录并编译**

cd 自定义空间名称

catkin\_make

1. **进入工作空间目录并执行**

先启动命令行1：roscore

再启动命令行2：cd 工作空间

source ./devel/setup.bash

rosrun 包名 自定义文件名.py

## ROS集成开发环境搭建

1. **vscode安装与卸载**

下载安装：百度

卸载：sudo dpkg --purge code

1. **vscode插件**

c/c++,Chinese,Cmake,Cmake tools,python,pylance,ROS，Bracket Pair Colorizer 2

1. **vscode使用的基本配置**
2. **创建 ROS 工作空间**

mkdir -p xxx\_ws/src

cd xxx\_ws

catkin\_make

1. **启动 vscode**

cd xxx\_ws

code .

1. **vscode 中编译 ros**

快捷键 ctrl + shift + B 调用编译，选择:catkin\_make:build

可以点击配置设置为默认，修改.vscode/tasks.json 文件

{

"version": "2.0.0",

"tasks": [

{

"label": "catkin\_make:debug", //代表提示的描述性信息

"type": "shell", //可以选择shell或者process,如果是shell代码是在shell里面运行一个命令，如果是process代表作为一个进程来运行

"command": "catkin\_make",//这个是我们需要运行的命令

"args": [],//如果需要在命令后面加一些后缀，可以写在这里，比如-DCATKIN\_WHITELIST\_PACKAGES=“pac1;pac2”

"group": {"kind":"build","isDefault":true},

"presentation": {

"reveal": "always"//可选always或者silence，代表是否输出信息

},

"problemMatcher": "$msCompile"

}

]

}

1. **创建 ROS 功能包**

选定 src 右击 ---> create catkin package

设置包名 添加依赖

1. **C++ 实现**

代码同上

当ROS\_\_INFO 终端输出有中文时，会出现乱码

解决办法：在函数开头加入下面代码的任意一句

setlocale(LC\_CTYPE, "zh\_CN.utf8");

setlocale(LC\_ALL, "");

1. **python 实现**

在功能包下新建scripts文件夹，添加python文件，并添加可执行权限

代码同上: 1.导包2.初始化 ROS 节点3.日志输出 HelloWorld

1. **配置 CMakeLists.txt**

同上

1. **编译执行**

编译: ctrl + shift + B

执行: 和之前一致，只是可以在 VScode 中添加终端，首先执行:source ./devel/setup.bash

1. **Roboware**

<https://max.book118.com/html/2018/0127/150769793.shtm>

1. **launch文件演示**

1选定功能包右击 ---> 添加 launch 文件夹

2选定 launch 文件夹右击 ---> 添加 launch 文件

3编辑 launch 文件内容

<launch>

<node pkg="HelloWorld" type="hello" name="hello" output="screen" />

<node pkg="turtlesim" type="turtlesim\_node" name="t1"/>

<node pkg="turtlesim" type="turtle\_teleop\_key" name="key1" />

</launch>

node ---> 包含的某个节点

pkg -----> 功能包

type ----> 被运行的节点文件(要与CMakeList中可执行文件名相同)

name --> 为节点命名（应与cpp文件中节点名相同，但好像不同也无所谓）

output-> 设置日志的输出目标

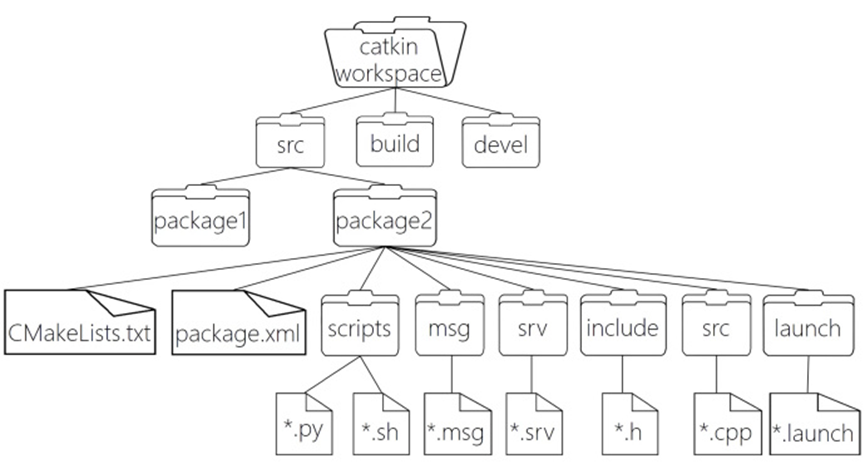
4运行 launch 文件：roslaunch 包名 launch文件名

1. **VS快捷键**

**https://blog.csdn.net/weixin\_46655235/article/details/121788623**

## ROS架构

1. **ROS文件系统**

****

WorkSpace --- 自定义的工作空间

|--- build:编译空间，用于存放CMake和catkin的缓存信息、配置信息和其他中间文件。

|--- devel:开发空间，用于存放编译后生成的目标文件，包括头文件、动态&静态链接库、可执行文件等。

|--- src: 源码

|-- package：功能包(ROS基本单元)包含多个节点、库与配置文件，包名所有字母小写，只能由字母、数字与下划线组成

|-- CMakeLists.txt 配置编译规则，比如源文件、依赖项、目标文件

|-- package.xml 包信息，比如:包名、版本、作者、依赖项...(以前版本是 manifest.xml)

|-- scripts 存储python文件

|-- src 存储C++源文件

|-- include 头文件

|-- msg 消息通信格式文件

|-- srv 服务通信格式文件

|-- action 动作格式文件

|-- launch 可一次性运行多个节点

|-- config 配置信息

|-- CMakeLists.txt: 编译的基本配置

1. **package.xml**

该文件定义有关软件包的属性，例如软件包名称，版本号，作者，维护者以及对其他catkin软件包的依赖性。请注意，该概念类似于旧版 rosbuild 构建系统中使用的manifest.xml文件。

<?xml version="1.0"?>

<!-- 格式: 以前是 1，推荐使用格式 2 -->

<package format="2">

<!-- 包名 -->

<name>HelloWorld</name>

<!-- 版本 -->

<version>0.0.0</version>

<!-- 描述信息 -->

<description>The demo\_hello\_vscode package</description>

<!-- One maintainer tag required, multiple allowed, one person per tag -->

<!-- Example: -->

<!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->

<!-- 维护人员 -->

<maintainer email="xuwenyu@todo.todo">xuwenyu</maintainer>

<!-- One license tag required, multiple allowed, one license per tag -->

<!-- Commonly used license strings: -->

<!-- BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->

<!-- 许可证信息，ROS核心组件默认 BSD -->

<license>TODO</license>

<!-- Url tags are optional, but multiple are allowed, one per tag -->

<!-- Optional attribute type can be: website, bugtracker, or repository -->

<!-- Example: -->

<!-- <url type="website">http://wiki.ros.org/demo01\_hello\_vscode</url> →

<!-- Author tags are optional, multiple are allowed, one per tag -->

<!-- Authors do not have to be maintainers, but could be -->

<!-- Example: -->

<!-- <author email="jane.doe@example.com">Jane Doe</author> -->

<!-- The \*depend tags are used to specify dependencies -->

<!-- Dependencies can be catkin packages or system dependencies -->

<!-- Examples: -->

<!-- Use depend as a shortcut for packages that are both build and exec dependencies -->

<!-- <depend>roscpp</depend> -->

<!-- Note that this is equivalent to the following: -->

<!-- <build\_depend>roscpp</build\_depend> -->

<!-- <exec\_depend>roscpp</exec\_depend> -->

<!-- Use build\_depend for packages you need at compile time: -->

<!-- <build\_depend>message\_generation</build\_depend> -->

<!-- Use build\_export\_depend for packages you need in order to build against this package: -->

<!-- <build\_export\_depend>message\_generation</build\_export\_depend> -->

<!-- Use buildtool\_depend for build tool packages: -->

<!-- <buildtool\_depend>catkin</buildtool\_depend> -->

<!-- Use exec\_depend for packages you need at runtime: -->

<!-- <exec\_depend>message\_runtime</exec\_depend> -->

<!-- Use test\_depend for packages you need only for testing: -->

<!-- <test\_depend>gtest</test\_depend> -->

<!-- Use doc\_depend for packages you need only for building documentation: -->

<!-- <doc\_depend>doxygen</doc\_depend> →

<!-- 依赖的构建工具，这是必须的 -->

<buildtool\_depend>catkin</buildtool\_depend>

<!-- 指定构建此软件包所需的软件包 -->

<build\_depend>roscpp</build\_depend>

<build\_depend>rospy</build\_depend>

<build\_depend>std\_msgs</build\_depend>

<!-- 指定根据这个包构建库所需要的包 -->

<build\_export\_depend>roscpp</build\_export\_depend>

<build\_export\_depend>rospy</build\_export\_depend>

<build\_export\_depend>std\_msgs</build\_export\_depend>

<!-- 运行该程序包中的代码所需的程序包 -->

<exec\_depend>roscpp</exec\_depend>

<exec\_depend>rospy</exec\_depend>

<exec\_depend>std\_msgs</exec\_depend>

<!-- The export tag contains other, unspecified, tags -->

<export>

<!-- Other tools can request additional information be placed here -->

</export>

</package>

1. **Cmakelists.txt**

文件CMakeLists.txt是CMake构建系统的输入，用于构建软件包。任何兼容CMake的软件包都包含一个或多个CMakeLists.txt文件，这些文件描述了如何构建代码以及将代码安装到何处。

cmake\_minimum\_required(VERSION 3.0.2) #所需 cmake 版本

project(HelloWorld) #包名称，会被 ${PROJECT\_NAME} 的方式调用

## Compile as C++11, supported in ROS Kinetic and newer

# add\_compile\_options(-std=c++11)

## Find catkin macros and libraries

## if COMPONENTS list like find\_package(catkin REQUIRED COMPONENTS xyz)

## is used, also find other catkin packages

#设置构建所需要的软件包

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

)

## System dependencies are found with CMake's conventions

#默认添加系统依赖

# find\_package(Boost REQUIRED COMPONENTS system)

## Uncomment this if the package has a setup.py. This macro ensures

## modules and global scripts declared therein get installed

## See http://ros.org/doc/api/catkin/html/user\_guide/setup\_dot\_py.html

# 启动 python 模块支持

# catkin\_python\_setup()

################################################

## Declare ROS messages, services and actions ##

## 声明 ROS 消息、服务、动作... ##

################################################

## To declare and build messages, services or actions from within this

## package, follow these steps:

## \* Let MSG\_DEP\_SET be the set of packages whose message types you use in

## your messages/services/actions (e.g. std\_msgs, actionlib\_msgs, ...).

## \* In the file package.xml:

## \* add a build\_depend tag for "message\_generation"

## \* add a build\_depend and a exec\_depend tag for each package in MSG\_DEP\_SET

## \* If MSG\_DEP\_SET isn't empty the following dependency has been pulled in

## but can be declared for certainty nonetheless:

## \* add a exec\_depend tag for "message\_runtime"

## \* In this file (CMakeLists.txt):

## \* add "message\_generation" and every package in MSG\_DEP\_SET to

## find\_package(catkin REQUIRED COMPONENTS ...)

## \* add "message\_runtime" and every package in MSG\_DEP\_SET to

## catkin\_package(CATKIN\_DEPENDS ...)

## \* uncomment the add\_\*\_files sections below as needed

## and list every .msg/.srv/.action file to be processed

## \* uncomment the generate\_messages entry below

## \* add every package in MSG\_DEP\_SET to generate\_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder

# add\_message\_files(

# FILES

# Message1.msg

# Message2.msg

# )

## Generate services in the 'srv' folder

# add\_service\_files(

# FILES

# Service1.srv

# Service2.srv

# )

## Generate actions in the 'action' folder

# add\_action\_files(

# FILES

# Action1.action

# Action2.action

# )

## Generate added messages and services with any dependencies listed here

# 生成消息、服务时的依赖包

# generate\_messages(

# DEPENDENCIES

# std\_msgs

# )

################################################

## Declare ROS dynamic reconfigure parameters ##

## 声明 ROS 动态参数配置 ##

################################################

## To declare and build dynamic reconfigure parameters within this

## package, follow these steps:

## \* In the file package.xml:

## \* add a build\_depend and a exec\_depend tag for "dynamic\_reconfigure"

## \* In this file (CMakeLists.txt):

## \* add "dynamic\_reconfigure" to

## find\_package(catkin REQUIRED COMPONENTS ...)

## \* uncomment the "generate\_dynamic\_reconfigure\_options" section below

## and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder

# generate\_dynamic\_reconfigure\_options(

# cfg/DynReconf1.cfg

# cfg/DynReconf2.cfg

# )

###################################

## catkin specific configuration ##

## catkin 特定配置##

###################################

## The catkin\_package macro generates cmake config files for your package

## Declare things to be passed to dependent projects

## INCLUDE\_DIRS: uncomment this if your package contains header files

## LIBRARIES: libraries you create in this project that dependent projects also need

## CATKIN\_DEPENDS: catkin\_packages dependent projects also need

## DEPENDS: system dependencies of this project that dependent projects also need

# 运行时依赖

catkin\_package(

# INCLUDE\_DIRS include

# LIBRARIES HelloWorld

# CATKIN\_DEPENDS roscpp rospy std\_msgs

# DEPENDS system\_lib

)

###########

## Build ##

###########

## Specify additional locations of header files

## Your package locations should be listed before other locations

# 添加头文件路径，当前程序包的头文件路径位于其他文件路径之前

include\_directories(

# include

${catkin\_INCLUDE\_DIRS}

)

## Declare a C++ library

# 声明 C++ 库

# add\_library(${PROJECT\_NAME}

# src/${PROJECT\_NAME}/hello.cpp

# )

## Add cmake target dependencies of the library

## as an example, code may need to be generated before libraries

## either from message generation or dynamic reconfigure

# 添加库的 cmake 目标依赖

# add\_dependencies(${PROJECT\_NAME} ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

## Declare a C++ executable

## With catkin\_make all packages are built within a single CMake context

## The recommended prefix ensures that target names across packages don't collide

# 声明 C++ 可执行文件

add\_executable(hello src/hello.cpp)

## Rename C++ executable without prefix

## The above recommended prefix causes long target names, the following renames the

## target back to the shorter version for ease of user use

## e.g. "rosrun someones\_pkg node" instead of "rosrun someones\_pkg someones\_pkg\_node"

#重命名c++可执行文件

# set\_target\_properties(${PROJECT\_NAME}\_node PROPERTIES OUTPUT\_NAME node PREFIX "")

## Add cmake target dependencies of the executable

## same as for the library above

#添加可执行文件的 cmake 目标依赖

add\_dependencies(hello ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

## Specify libraries to link a library or executable target against

#指定库、可执行文件的链接库

target\_link\_libraries(hello

${catkin\_LIBRARIES})

#############

## Install ##

## 安装 ##

#############

# all install targets should use catkin DESTINATION variables

# See http://ros.org/doc/api/catkin/html/adv\_user\_guide/variables.html

## Mark executable scripts (Python etc.) for installation

## in contrast to setup.py, you can choose the destination

#设置用于安装的可执行脚本

catkin\_install\_python(PROGRAMS

scripts/hello.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

## Mark executables for installation

## See http://docs.ros.org/melodic/api/catkin/html/howto/format1/building\_executables.html

# install(TARGETS ${PROJECT\_NAME}\_node

# RUNTIME DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

# )

## Mark libraries for installation

## See http://docs.ros.org/melodic/api/catkin/html/howto/format1/building\_libraries.html

# install(TARGETS ${PROJECT\_NAME}

# ARCHIVE DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

# LIBRARY DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

# RUNTIME DESTINATION ${CATKIN\_GLOBAL\_BIN\_DESTINATION}

# )

## Mark cpp header files for installation

# install(DIRECTORY include/${PROJECT\_NAME}/

# DESTINATION ${CATKIN\_PACKAGE\_INCLUDE\_DESTINATION}

# FILES\_MATCHING PATTERN "\*.h"

# PATTERN ".svn" EXCLUDE

# )

## Mark other files for installation (e.g. launch and bag files, etc.)

# install(FILES

# # myfile1

# # myfile2

# DESTINATION ${CATKIN\_PACKAGE\_SHARE\_DESTINATION}

# )

#############

## Testing ##

#############

## Add gtest based cpp test target and link libraries

# catkin\_add\_gtest(${PROJECT\_NAME}-test test/test\_demo01\_hello\_vscode.cpp)

# if(TARGET ${PROJECT\_NAME}-test)

# target\_link\_libraries(${PROJECT\_NAME}-test ${PROJECT\_NAME})

# endif()

## Add folders to be run by python nosetests

# catkin\_add\_nosetests(test)

## ROS文件系统相关命令

**1.增**

catkin\_create\_pkg 自定义包名 依赖包 === 创建新的ROS功能包

sudo apt install xxx === 安装 ROS功能包

**2.删**

sudo apt purge xxx ==== 删除某个功能包

**3.查**

rospack list === 列出所有功能包

rospack find 包名 === 查找某个功能包是否存在，如果存在返回安装路径

roscd 包名 === 进入某个功能包

rosls 包名 === 列出某个包下的文件

apt search xxx === 搜索某个功能包

**4.改**

rosed 包名 文件名 === 修改功能包文件

需要安装 vim

比如:rosed turtlesim Color.msg

**5.执行**

**5.1 roscore或(指定端口号)roscore -p xxxx**

roscore === 是 ROS 的系统先决条件节点和程序的集合， 必须运行 roscore 才能使 ROS 节点进行通信。

roscore 将启动:

ros master

ros 参数服务器

rosout 日志节点

**5.2 rosrun**

rosrun 包名 可执行文件名 === 运行指定的ROS节点

**5.3 roslaunch**

roslaunch 包名 launch文件名 === 执行某个包下的 launch 文件

## ROS计算图

前面介绍的是ROS文件结构，是磁盘上 ROS 程序的存储结构，是静态的，而 ros 程序运行之后，不同的节点之间是错综复杂的，ROS 中提供了一个实用的工具:rqt\_graph。

rqt\_graph能够创建一个显示当前系统运行情况的动态图形。ROS 分布式系统中不同进程需要进行数据交互，计算图可以以点对点的网络形式表现数据交互过程。rqt\_graph是rqt程序包中的一部分。

计算图安装

如果前期把所有的功能包（package）都已经安装完成，则直接在终端窗口中输入：rosrun rqt\_graph rqt\_graph或rqt\_graph

如果未安装则在终端（terminal）中输入

sudo apt install ros-<distro>-rqt

sudo apt install ros-<distro>-rqt-common-plugins

# ROS通信机制

ROS是进程（也称为节点/Nodes）的分布式框架。ROS 中的基本通信机制主要有如下三种实现策略:话题通信(发布订阅模式),服务通信(请求响应模式),参数服务器(参数共享模式)

## 话题通信

话题通信是ROS中使用频率最高的一种通信模式，话题通信是基于发布订阅模式的，也即:一个节点发布消息，另一个节点订阅该消息。话题通信的应用场景也极其广泛，比如下面一个常见场景:

机器人在执行导航功能，使用的传感器是激光雷达，机器人会采集激光雷达感知到的信息并计算，然后生成运动控制信息驱动机器人底盘运动。

在上述场景中:以激光雷达信息的采集处理为例，在 ROS 中有一个节点需要时时的发布当前雷达采集到的数据，导航模块中也有节点会订阅并解析雷达数据。

再以运动消息的发布为例，导航模块会根据传感器采集的数据时时的计算出运动控制信息并发布给底盘，底盘也可以有一个节点订阅运动信息并最终转换成控制电机的脉冲信号。

**作用:用于不断更新的、少逻辑处理的数据传输场景。**

### 1.理论模型

ROS Master (管理者)负责保管 Talker (发布者)和 Listener (订阅者)注册的信息，并匹配话题相同的 Talker 与 Listener，帮助 Talker 与 Listener 建立连接，连接建立后，Talker 可以发布消息，且发布的消息会被 Listener 订阅。

整个流程由以下步骤实现:

（1）Talker注册（2）Listener注册（3）ROS Master实现信息匹配（4）Listener向Talker发送请求（5）Talker确认请求（6）Listener与Talker件里连接（7）Talker向Listener发送消息

注意1:上述实现流程中，前五步使用的 RPC协议，最后两步使用的是 TCP 协议

注意2: Talker 与 Listener 的启动无先后顺序要求

注意3: Talker 与 Listener 都可以有多个

注意4: Talker 与 Listener 连接建立后，不再需要 ROS Master。即关闭ROS Master，Talker 与 Listern 照常通信。

### 2.话题通信基本操作(C++)

需求:要求发布方以10HZ(每秒10次)的频率发布文本消息，订阅方订阅消息并将消息内容打印输出

**发布方**

/\*

需求: 实现基本的话题通信，一方发布数据，一方接收数据，

实现的关键点:

1.发送方

2.接收方

3.数据(此处为普通文本)

PS: 二者需要设置相同的话题

消息发布方:

循环发布信息:HelloWorld 后缀数字编号

实现流程:

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "std\_msgs/String.h" //普通文本类型的消息,消息/服务的类型（type）为功能包（package）名/（.msg/.srv）文件名

#include <sstream>

int main(int argc, char \*argv[])

{

//设置中文编码

setlocale(LC\_ALL,"");

//2.初始化 ROS 节点:命名(唯一)

// 参数1和参数2 后期为节点传值会使用

// 参数3 是节点名称，是一个标识符，需要保证运行后，在 ROS 网络拓扑中唯一

ros::init(argc,argv,"talker");

//3.实例化 ROS 句柄

ros::NodeHandle nh;//该类封装了 ROS 中的一些常用功能，向ROS master注册节点信息

//4.实例化 发布者 对象

//泛型: 发布的消息类型

//参数1: 要发布到的话题

//参数2: 队列中最大保存的消息数，超出此阀值时，先进的先销毁(时间早的先销毁)

ros::Publisher pub = nh.advertise<std\_msgs::String>("chatter",10);

//5.组织被发布的数据，并编写逻辑发布数据

//数据(动态组织)

std\_msgs::String msg;

// msg.data = "你好啊！！！";

std::string msg\_front = "Hello 你好！"; //消息前缀

int count = 0; //消息计数器

//逻辑(一秒10次)

ros::Rate r(1);

//节点不死

while (ros::ok())

{

//使用 stringstream 拼接字符串与编号

std::stringstream ss;

ss << msg\_front << count;

msg.data = ss.str();

//发布消息

pub.publish(msg);

//加入调试，打印发送的消息

ROS\_INFO("发送的消息:%s",msg.data.c\_str());

// c\_str() 函数可以将 const string\* 类型 转化为 cons char\* 类型, 就是将C++的string转化为C的字符串数组，c\_str()生成一个const char \*指针，指向字符串的首地址

//根据前面制定的发送贫频率自动休眠 休眠时间 = 1/频率；

r.sleep();

count++;//循环结束前，让 count 自增

//暂无应用

ros::spinOnce();

}

return 0;

}

**订阅方**

\*消息订阅方:订阅话题并打印接收到的消息

实现流程:

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "std\_msgs/String.h"

//5.处理订阅的消息(回调函数)

void doMsg(const std\_msgs::String::ConstPtr& msg\_p){

ROS\_INFO("我听见:%s",msg\_p->data.c\_str());

// ROS\_INFO("我听见:%s",(\*msg\_p).data.c\_str());

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

//2.初始化 ROS 节点:命名(唯一)

ros::init(argc,argv,"listener");

//3.实例化 ROS 句柄

ros::NodeHandle nh;

//4.实例化 订阅者 对象(订阅者可不加<>因为可用回调函数判断类型)

ros::Subscriber sub = nh.subscribe<std\_msgs::String>("chatter",10,doMsg);

//6.设置循环调用回调函数

ros::spin();//循环读取接收的数据，并调用回调函数处理

return 0;

}

**配置 CmakeLists.txt**

add\_executable(talker src/talker.cpp)

add\_executable(listener src/listener.cpp)

target\_link\_libraries(talker ${catkin\_LIBRARIES})

target\_link\_libraries(listener ${catkin\_LIBRARIES})

**执行:**1.启动 roscore;2.启动发布节点;3.启动订阅节点。

补充:

vscode 中的 main 函数 声明 int main(int argc, char const \*argv[]){}，默认生成 argv 被 const 修饰，需要去除该修饰符

ros/ros.h No such file or directory .....检查 CMakeList.txt find\_package 出现重复,删除内容少的即可

订阅时，第一条数据丢失

原因: 发送第一条数据时， publisher 还未在 roscore 注册完毕

解决: 注册后，加入休眠 ros::Duration(3.0).sleep(); 延迟第一条数据的发送

可以使用 rqt\_graph 查看节点关系。

### 3.话题通信基本操作(Python)

**发布方**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

#上面的代码是为了能支持中文编码

#1.导包

import rospy

from std\_msgs.msg import String

if \_\_name\_\_ == "\_\_main\_\_":

#2.初始化 ROS 节点:命名(唯一)

rospy.init\_node("talker\_p")

#3.实例化 发布者 对象

pub = rospy.Publisher("chatter",String,queue\_size=10)

#4.组织被发布的数据，并编写逻辑发布数据

msg = String() #创建 msg 对象

msg\_front = "hello 你好"

count = 0 #计数器

# 设置循环频率

rate = rospy.Rate(1)

while not rospy.is\_shutdown():

#拼接字符串

msg.data = msg\_front + str(count)

pub.publish(msg)

rate.sleep()

rospy.loginfo("写出的数据:%s",msg.data)

count += 1

**订阅方**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

#1.导包

import rospy

from std\_msgs.msg import String

#4.处理订阅的消息(回调函数)

def doMsg(msg):

rospy.loginfo("I heard:%s",msg.data)

if \_\_name\_\_ == "\_\_main\_\_":

#2.初始化 ROS 节点:命名(唯一)

rospy.init\_node("listener\_p")

#3.实例化 订阅者 对象

sub = rospy.Subscriber("chatter",String,doMsg,queue\_size=10)

#5.设置循环调用回调函数

rospy.spin()

**添加可执行权限：**终端下进入 scripts 执行:chmod +x \*.py

**配置 CmakeLists.txt**

catkin\_install\_python(PROGRAMS

scripts/talker.py

scripts/listener.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

### 4.话题通信自定义msg

ROS 中通过 std\_msgs 封装了一些原生的数据类型,比如:String、Int32、Int64、Char、Bool、Empty.... 但是，这些数据一般只包含一个 data 字段，结构的单一意味着功能上的局限性，当传输一些复杂的数据，比如: 激光雷达的信息... std\_msgs 由于描述性较差而显得力不从心，这种场景下可以使用自定义的**消息类型（rosmsg type）**

msgs只是简单的文本文件，每行具有字段类型和字段名称，可以使用的字段类型有：

int8, int16, int32, int64 (或者无符号类型: uint\*)

float32, float64

string

time, duration

other msg files

variable-length array[] and fixed-length array[C]

ROS中还有一种特殊类型：Header，标头包含时间戳和ROS中常用的坐标帧信息。会经常看到msg文件的第一行具有Header标头。

需求:创建自定义消息，该消息包含人的信息:姓名、身高、年龄等。

**流程: 1.定义msg文件：**

功能包下新建 msg 目录，添加文件 Person.msg

string name

uint16 age

float64 height

**2.编辑配置文件：**

package.xml中添加编译依赖与执行依赖：

<build\_depend>message\_generation</build\_depend>

<exec\_depend>message\_runtime</exec\_depend>

CMakeLists.txt编辑 msg 相关配置（在原CmakeList文件中有需要自己找到修改）

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

message\_generation

)

# 需要加入 message\_generation,必须有 std\_msgs

## 配置 msg 源文件

add\_message\_files(

FILES

Person.msg

)

# 生成消息时依赖于 std\_msgs

generate\_messages(

DEPENDENCIES

std\_msgs

)

#执行时依赖(自定义msg必须加，srv可加可不加)

catkin\_package(

# INCLUDE\_DIRS include

# LIBRARIES demo1\_msg

CATKIN\_DEPENDS roscpp rospy std\_msgs message\_runtime

# DEPENDS system\_lib

)

**3.编译**

编译后的中间文件查看:C++ 需要调用的中间文件(.../工作空间/devel/include/包名/xxx.h)

Python 需要调用的中间文件(.../工作空间/devel/lib/python2.7/dist-packages/包名/msg)

### 5.话题通信自定义msg调用(C++)

需求:编写发布订阅实现，要求发布方以10HZ(每秒10次)的频率发布自定义消息，订阅方订阅自定义消息并将消息内容打印输出。

流程:  **0.vscode 配置**

为了方便代码提示以及避免误抛异常，需要先配置 vscode，将前面生成的 head 文件路径配置进 c\_cpp\_properties.json 的 includepath属性:

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"home/xxx/yyy工作空间/devel/include/" //配置 head 文件的路径

"${workspaceFolder}/devel/include"//或者

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

**1.发布方**

/\*

需求: 循环发布人的信息

\*/

#include "ros/ros.h"

#include "demo1\_msg/Person.h"

//消息/服务的类型名（type）为功能包（package）名/（.msg/.srv）文件名

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"talker\_person");

ros::NodeHandle nh;

//创建发布者对象

ros::Publisher pub = nh.advertise<demo1\_msg::Person>("chatter",1000);//cpp发送方需要加类型，接收方不需要

//组织被发布的消息，编写发布逻辑并发布消息

demo1\_msg::Person p;

p.name = "sunwukong";

p.age = 2000;

p.height = 1.45;

ros::Rate r(1);

while (ros::ok())

{

pub.publish(p);

p.age += 1;

ROS\_INFO("我叫:%s,今年%d岁,高%f米", p.name.c\_str(), p.age, p.height);

r.sleep();

ros::spinOnce();//没用到

}

return 0;

}

**2.订阅方**

/\*

需求: 订阅人的信息

\*/

#include "ros/ros.h"

#include "demo1\_msg/Person.h"

//回调函数中处理 person

void doPerson(const demo1\_msg::Person::ConstPtr& person\_p){

ROS\_INFO("订阅的人信息:%s, %d, %.2f", person\_p->name.c\_str(), person\_p->age, person\_p->height);//%2f是float的所有位数输出2位,%.2f是float后的小数只输出两位

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"listener\_person");

ros::NodeHandle nh;

// 订阅者可不加<>因为可用回调函数判断类型

ros::Subscriber sub = nh.subscribe<demo1\_msg::Person>("chatter",10,doPerson);

ros::spin();

return 0;

}

**3.配置 CMakeLists.txt**

需要添加 add\_dependencies 用以设置所依赖的消息相关的中间文件。

add\_executable(p\_talker src/p\_talker.cpp)

add\_executable(p\_person src/p\_listener.cpp)

#\_generate\_messages\_cpp与\_gencpp一样

add\_dependencies(p\_talker ${PROJECT\_NAME}\_generate\_messages\_cpp)

add\_dependencies(p\_listener ${PROJECT\_NAME}\_generate\_messages\_cpp)

target\_link\_libraries(p\_talker ${catkin\_LIBRARIES})

target\_link\_libraries(p\_listener ${catkin\_LIBRARIES})

**4.执行:**1.启动 roscore;2.启动发布节点;3.启动订阅节点。

### 6.话题通信自定义msg调用(Python)

**0.vscode配置**

为了方便代码提示以及误抛异常，需要先配置 vscode，将前面生成的 python 文件路径配置进 settings.json

{

"python.autoComplete.extraPaths": [

"/opt/ros/melodic/lib/python2.7/dist-packages",

"home/xxx/yyy工作空间/devel/lib/python2.7/dist-packages"

]

}

**1.发布方**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

发布方:循环发送消息

"""

import rospy

from demo1\_msg.msg import Person

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("talker\_person\_p")

pub = rospy.Publisher("chatter ",Person,queue\_size=10)

#组织消息

p = Person()

p.name = "葫芦瓦"

p.age = 18

p.height = 0.75

#编写消息发布逻辑

rate = rospy.Rate(1)

while not rospy.is\_shutdown():

pub.publish(p) #发布消息

rate.sleep() #休眠

rospy.loginfo("姓名:%s, 年龄:%d, 身高:%.2f",p.name, p.age, p.height)

**2.订阅方**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

订阅方:订阅消息

"""

import rospy

from demo1\_msg.msg import Person

def doPerson(p):

rospy.loginfo("接收到的人的信息:%s, %d, %.2f",p.name, p.age, p.height)

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("listener\_person\_p")

sub = rospy.Subscriber("chatter ",Person,doPerson,queue\_size=10)

rospy.spin()

**3.权限设置:** 终端下进入 scripts 执行:chmod +x \*.py

**4.配置 CMakeLists.txt**

catkin\_install\_python(PROGRAMS

scripts/talker.py

scripts/listener.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

## 服务通信

服务通信也是ROS中一种极其常用的通信模式，服务通信是基于请求响应模式的，是一种应答机制。也即: 一个节点A向另一个节点B发送请求，B接收处理请求并产生响应结果返回给A。比如如下场景:

机器人巡逻过程中，控制系统分析传感器数据发现可疑物体或人... 此时需要拍摄照片并留存。在上述场景中，就使用到了服务通信。

一个节点需要向相机节点发送拍照请求，相机节点处理请求，并返回处理结果。

作用：用于偶然的、对时时性有要求、有一定逻辑处理需求的数据传输场景。

案例：实现两个数字的求和，客户端节点，运行会向服务器发送两个数字，服务器端节点接收两个数字求和并将结果响应回客户端。

### 1.服务通信理论模型

ROS Master (管理者)负责保管 Server (服务端)和 Client (客户端)注册的信息，并匹配话题相同的 Server 与 Client ，帮助 Server 与 Client 建立连接，连接建立后，Client 发送请求信息，Server 返回响应信息。

注意:

1.客户端请求被处理时，需要保证服务器已经启动；

2.服务端和客户端都可以存在多个。

### 2.服务通信自定义srv

需求:服务通信中，客户端提交两个整数至服务端，服务端求和并响应结果到客户端，请创建服务器与客户端通信的数据载体。

流程:srv 文件内的可用数据类型与 msg 文件一致，且定义 srv 实现流程与自定义 msg 实现流程类似:

**1.定义srv文件**

服务通信中，数据分成两部分，请求与响应，在 srv 文件中请求和响应使用---分割，具体实现如下:

功能包下新建 srv 目录，添加 xxx.srv 文件，内容:

# 客户端请求时发送的两个数字

int32 num1

int32 num2

---

# 服务器响应发送的数据

int32 sum

**2.编辑配置文件**

package.xml中添加编译依赖与执行依赖

<build\_depend>message\_generation</build\_depend>

<exec\_depend>message\_runtime</exec\_depend>

CMakeLists.txt编辑 srv 相关配置

find\_package(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

message\_generation

)

# 需要加入 message\_generation,必须有 std\_msgs

add\_service\_files(

FILES

num.srv

)

generate\_messages(

DEPENDENCIES

std\_msgs

)

注意: 官网没有在 catkin\_package 中配置 message\_runtime,经测试配置也可以

**3.编译：**编译后的中间文件查看:C++ 需要调用的中间文件(.../工作空间/devel/include/包名/xxx.h)

Python 需要调用的中间文件(.../工作空间/devel/lib/python2.7/dist-packages/包名/srv)

### 3.服务通信自定义srv调用(C++)

需求:编写服务通信，客户端提交两个整数至服务端，服务端求和并响应结果到客户端。

流程: **0.vscode配置**

需要像之前自定义 msg 实现一样配置c\_cpp\_properies.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同。

**1.服务端**

/\*

需求: 编写两个节点实现服务通信，客户端节点需要提交两个整数到服务器，服务器需要解析客户端提交的数据，相加后，将结果响应回客户端，客户端再解析

服务器实现:

1.包含头文件

2.初始化 ROS 节点

3.创建 ROS 句柄

4.创建 服务 对象

5.回调函数处理请求并产生响应

6.由于请求有多个，需要调用 ros::spin()

\*/

#include "ros/ros.h"

#include "demo\_srv/num.h"

//消息/服务的类型名（type）为功能包（package）名/（.msg/.srv）文件名

// bool 返回值由于标志是否处理成功

bool doReq(demo\_server::num::Request& req,

demo\_server::num::Response& resp){

int num1 = req.num1;

int num2 = req.num2;

ROS\_INFO("服务器接收到的请求数据为:num1 = %d, num2 = %d",num1, num2);

//逻辑处理

if (num1 < 0 || num2 < 0)

{

ROS\_ERROR("提交的数据异常:数据不可以为负数");

return false;

}

//如果没有异常，那么相加并将结果赋值给 resp

resp.sum = num1 + num2;

return true;

}//回调函数处理请求并产生响应

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"Add\_server");

ros::NodeHandle nh;

//创建 服务 对象

ros::ServiceServer server = nh.advertiseService("Add",doReq);

ROS\_INFO("服务已经启动");

//由于请求有多个，需要调用 ros::spin()

ros::spin();

return 0;

}

**2.客户端**

/\* 服务器实现:

1.包含头文件

2.初始化 ROS 节点

3.创建 ROS 句柄

4.创建 客户端 对象

5.请求服务，接收响应

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "demo\_srv /num.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 调用时动态传值,如果通过 launch 的 args 传参，需要传递的参数个数 +3

if (argc != 3)

// if (argc != 5)//launch 传参(0-文件路径 1传入的参数 2传入的参数 3节点名称 4日志路径)

{

ROS\_ERROR("请提交两个整数");

return 1;

}

// 2.初始化 ROS 节点

ros::init(argc,argv,"Add\_client");

// 3.创建 ROS 句柄

ros::NodeHandle nh;

// 4.创建 客户端 对象

ros::ServiceClient client = nh.serviceClient<demo\_srv::num>("Add");

//等待服务启动成功

//方式1

ros::service::waitForService("Add");

//方式2

// client.waitForExistence();

// 5.组织请求数据

//方法1

//demo\_srv::num::Request a;

//demo\_srv::num::Response b;

//a.num1 = atoi(argv[1]);//atoi函数将字符串转为整数

//a.num2 = atoi(argv[2]);

//方法2

demo\_srv::Add ai;

ai.request.num1 = atoi(argv[1]);

ai.request.num2 = atoi(argv[2]);

// 6.发送请求,返回 bool 值，标记是否成功

bool flag = client.call(ai);//没有bool flag也行

// 7.处理响应

if (flag)

{

ROS\_INFO("请求正常处理,响应结果:%d",ai.response.sum);

}

else

{

ROS\_ERROR("请求处理失败....");

return 1;

}

return 0;

}

**3.****配置 CMakeLists.txt**

add\_executable(Add\_server src/Add\_server.cpp)

add\_executable(Add\_client src/Add\_client.cpp)

#\_generate\_messages\_cpp与\_gencpp一样

add\_dependencies(Add\_server ${PROJECT\_NAME}\_gencpp)

add\_dependencies(Add\_client ${PROJECT\_NAME}\_gencpp)

target\_link\_libraries(Add\_server ${catkin\_LIBRARIES})

target\_link\_libraries(Add\_client ${catkin\_LIBRARIES})

**4.执行：**需要先启动服务:rosrun 包名 服务

然后再调用客户端 :rosrun 包名 客户端 参数1 参数2

### 4.服务通信自定义srv调用(Python)

流程: **0.vscode配置**

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"python.autoComplete.extraPaths": [

"/opt/ros/melodic/lib/python2.7/dist-packages",

"home/xxx/yyy工作空间/devel/lib/python2.7/dist-packages"

]

}

1. **服务端**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

需求:

编写两个节点实现服务通信，客户端节点需要提交两个整数到服务器

服务器需要解析客户端提交的数据，相加后，将结果响应回客户端，

客户端再解析

服务器端实现:

1.导包

2.初始化 ROS 节点

3.创建服务对象

4.回调函数处理请求并产生响应

5.spin 函数

"""

# 1.导包

import rospy

from demo\_srv.srv import num,numRequest,numResponse#根据srv文件名来定

# 回调函数的参数是请求对象，返回值是响应对象

def doReq(req):

# 解析提交的数据

sum = req.num1 + req.num2

rospy.loginfo("提交的数据:num1 = %d, num2 = %d, sum = %d",req.num1, req.num2, sum)

# 创建响应对象，赋值并返回

# resp = numResponse()

# resp.sum = sum

resp = numResponse(sum)

return resp

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("add\_server\_p")

# 3.创建服务对象

server = rospy.Service("Add",Add,doReq)

# 4.回调函数处理请求并产生响应

# 5.spin 函数

rospy.spin()

1. **客户端**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

客户端实现:

1.导包

2.初始化 ROS 节点

3.创建请求对象

4.发送请求

5.接收并处理响应

优化: 加入数据的动态获取

"""

#1.导包

import rospy

from demo\_srv.srv import \*

import sys

if \_\_name\_\_ == "\_\_main\_\_":

#优化实现

if len(sys.argv) != 3:

rospy.logerr("请正确提交参数")

sys.exit(1)

# 2.初始化 ROS 节点

rospy.init\_node("Add\_Client\_p")

# 3.创建请求对象

client = rospy.ServiceProxy("Add",Add)

# 请求前，等待服务已经就绪

# 方式1:

# rospy.wait\_for\_service("Add")

# 方式2

client.wait\_for\_service()

# 4.发送请求,接收并处理响应

# 方式1

# resp = client(3,4)

# 方式2

# resp = client(AddRequest(1,5))

# 方式3

req = AddRequest()

# req.num1 = 100

# req.num2 = 200

#优化

req.num1 = int(sys.argv[1])

req.num2 = int(sys.argv[2])

resp = client.call(req)

rospy.loginfo("响应结果:%d",resp.sum)

1. **设置权限:** 终端下进入 scripts 执行:chmod +x \*.py
2. **配置 CMakeLists.txt**

catkin\_install\_python(PROGRAMS

scripts/Add\_Server\_p.py

scripts/Add\_Client\_p.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

## 参数服务器

参数服务器在ROS中主要用于实现不同节点之间的数据共享。参数服务器相当于是独立于所有节点的一个公共容器(全局字典)，可以将数据存储在该容器中，被不同的节点调用，当然不同的节点也可以往其中存储数据，关于参数服务器的典型应用场景如下:

全局路径规划和本地路径规划时，就会使用到参数服务器：路径规划时，需要参考小车的尺寸，我们可以将这些尺寸信息存储到参数服务器，全局路径规划节点与本地路径规划节点都可以从参数服务器中调用这些参数

**作用:**存储一些多节点共享的数据，类似于全局变量。

**案例:**实现参数增删改查操作。

### 1.参数服务器理论模型

ROS Master (管理者)作为一个公共容器保存参数，Talker (参数设置者)可以向容器中设置参数(包括参数名与参数值)，Listener (参数调用者)可以获取参数(请求中包含要查找的参数名)。

参数可使用数据类型:32-bit integers(4个字节int),booleans,strings,doubles,iso8601 dates（时间,例：日历表示2004-05-03或20040503，时间表示（北京时间）22:30:05+08:00或223005+0800或223005+08）,lists（单列集合）,base64-encoded binary data,字典（双列集合键值对）

### 2.参数操作(C++)

**1.参数服务器新增(修改)参数**

/\*

参数服务器操作之新增与修改(二者API一样)\_C++实现:

在 roscpp 中提供了两套 API 实现参数操作

ros::NodeHandle

setParam("键",值)

ros::param

set("键","值")

示例:分别设置整形、浮点、字符串、bool、列表、字典等类型参数

修改(相同的键，不同的值)

\*/

#include "ros/ros.h"

int main(int argc, char \*argv[])

{

ros::init(argc,argv,"set\_update\_param");

std::vector<std::string> stus;

stus.push\_back("zhangsan");

stus.push\_back("李四");

stus.push\_back("王五");

stus.push\_back("孙大脑袋");

std::map<std::string,std::string> friends;

friends["guo"] = "huang";

friends["yuang"] = "xiao";

//NodeHandle--------------------------------------------------------

ros::NodeHandle nh;

nh.setParam("nh\_int",10); //整型

nh.setParam("nh\_double",3.14); //浮点型

nh.setParam("nh\_bool",true); //bool

nh.setParam("nh\_string","hello NodeHandle"); //字符串

nh.setParam("nh\_vector",stus); // vector

nh.setParam("nh\_map",friends); // map

//修改演示(相同的键，不同的值)

nh.setParam("nh\_int",10000);

//param--------------------------------------------------------

ros::param::set("param\_int",20);

ros::param::set("param\_double",3.14);

ros::param::set("param\_string","Hello Param");

ros::param::set("param\_bool",false);

ros::param::set("param\_vector",stus);

ros::param::set("param\_map",friends);

//修改演示(相同的键，不同的值)

ros::param::set("param\_int",20000);

return 0;

}

**2.参数服务器获取参数**

/\*

参数服务器操作之查询\_C++实现:

在 roscpp 中提供了两套 API 实现参数操作

ros::NodeHandle

param(键,默认值)

存在，返回对应结果，否则返回默认值

getParam(键,存储结果的变量)

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamCached(键,存储结果的变量)--提高变量获取效率

存在,返回 true,且将值赋值给参数2

若果键不存在，那么返回值为 false，且不为参数2赋值

getParamNames(std::vector<std::string>)

获取所有的键,并存储在参数 vector 中

hasParam(键)

是否包含某个键，存在返回 true，否则返回 false

searchParam(参数1，参数2)

搜索键，参数1是被搜索的键，参数2存储搜索结果的变量

ros::param ----- 与 NodeHandle 类似

ros::param::param(键,默认值)

ros::param::get(键,存储结果的变量)

ros::param::getCached(键,存储结果的变量)

ros::param::getParamNames(std::vector<std::string>)

ros::param::has(键)

ros::param::search(键，参数2)

\*/

#include "ros/ros.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"get\_param");

//NodeHandle--------------------------------------------------------

/\*

ros::NodeHandle nh;

// param 函数

int res1 = nh.param("nh\_int",100); // 键存在

int res2 = nh.param("nh\_int2",100); // 键不存在

int nh\_int3;

int res3 = nh.param<int>("nh\_int3",nh\_int3,100);//若参数服务器中参数" nh\_int3"存在， 把" nh\_int3"的值传递给变量nh\_int3；如果" nh\_int3"不存在，就传递默认值100

ROS\_INFO("param获取结果:%d,%d",res1,res2);

// getParam 函数

int nh\_int\_value;

double nh\_double\_value;

bool nh\_bool\_value;

std::string nh\_string\_value;

std::vector<std::string> stus;

std::map<std::string, std::string> friends;

nh.getParam("nh\_int",nh\_int\_value);

nh.getParam("nh\_double",nh\_double\_value);

nh.getParam("nh\_bool",nh\_bool\_value);

nh.getParam("nh\_string",nh\_string\_value);

nh.getParam("nh\_vector",stus);

nh.getParam("nh\_map",friends);

ROS\_INFO("getParam获取的结果:%d,%.2f,%s,%d",

nh\_int\_value,

nh\_double\_value,

nh\_string\_value.c\_str(),

nh\_bool\_value

);

for (auto &&stu : stus)//左右值通用引用且保持其常量性（可读写）

{

ROS\_INFO("stus 元素:%s",stu.c\_str());//c中无string类型，通过c\_str()把string对象转换成c中的字符串样式

}

for (auto &&f : friends)

{

ROS\_INFO("map 元素:%s = %s",f.first.c\_str(), f.second.c\_str());

}

// getParamCached()

nh.getParamCached("nh\_int",nh\_int\_value);

ROS\_INFO("通过缓存获取数据:%d",nh\_int\_value);

//getParamNames()

std::vector<std::string> param\_names1;

nh.getParamNames(param\_names1);

for (auto &&name : param\_names1)

{

ROS\_INFO("名称解析name = %s",name.c\_str());

}

ROS\_INFO("----------------------------");

ROS\_INFO("存在 nh\_int 吗? %d",nh.hasParam("nh\_int"));

ROS\_INFO("存在 nh\_intttt 吗? %d",nh.hasParam("nh\_intttt"));

std::string key;

nh.searchParam("nh\_int",key);

ROS\_INFO("搜索键:%s",key.c\_str());

\*/

//param--------------------------------------------------------

ROS\_INFO("++++++++++++++++++++++++++++++++++++++++");

int res3 = ros::param::param("param\_int",20); //存在

int res4 = ros::param::param("param\_int2",20); // 不存在返回默认

ROS\_INFO("param获取结果:%d,%d",res3,res4);

// getParam 函数

int param\_int\_value;

double param\_double\_value;

bool param\_bool\_value;

std::string param\_string\_value;

std::vector<std::string> param\_stus;

std::map<std::string, std::string> param\_friends;

ros::param::get("param\_int",param\_int\_value);

ros::param::get("param\_double",param\_double\_value);

ros::param::get("param\_bool",param\_bool\_value);

ros::param::get("param\_string",param\_string\_value);

ros::param::get("param\_vector",param\_stus);

ros::param::get("param\_map",param\_friends);

ROS\_INFO("getParam获取的结果:%d,%.2f,%s,%d",

param\_int\_value,

param\_double\_value,

param\_string\_value.c\_str(),

param\_bool\_value

);

for (auto &&stu : param\_stus)

{

ROS\_INFO("stus 元素:%s",stu.c\_str());

}

for (auto &&f : param\_friends)

{

ROS\_INFO("map 元素:%s = %s",f.first.c\_str(), f.second.c\_str());

}

// getParamCached()

ros::param::getCached("param\_int",param\_int\_value);

ROS\_INFO("通过缓存获取数据:%d",param\_int\_value);

//getParamNames()

std::vector<std::string> param\_names2;

ros::param::getParamNames(param\_names2);

for (auto &&name : param\_names2)

{

ROS\_INFO("名称解析name = %s",name.c\_str());

}

ROS\_INFO("----------------------------");

ROS\_INFO("存在 param\_int 吗? %d",ros::param::has("param\_int"));

ROS\_INFO("存在 param\_intttt 吗? %d",ros::param::has("param\_intttt"));

std::string key;

ros::param::search("param\_int",key);

ROS\_INFO("搜索键:%s",key.c\_str());

return 0;

}

**3.参数服务器删除参数**

/\*

参数服务器操作之删除\_C++实现:

ros::NodeHandle

deleteParam("键")

根据键删除参数，删除成功，返回 true，否则(参数不存在)，返回 false

ros::param

del("键")

\*/

#include "ros/ros.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"delete\_param");

ros::NodeHandle nh;

bool r1 = nh.deleteParam("nh\_int");

ROS\_INFO("nh 删除结果:%d",r1);

bool r2 = ros::param::del("param\_int");

ROS\_INFO("param 删除结果:%d",r2);

return 0;

}

### 3.参数操作(Python)

**1.参数服务器新增(修改)参数**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

参数服务器操作之新增与修改(二者API一样)\_Python实现:

"""

import rospy

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("set\_update\_paramter\_p")

# 设置各种类型参数

rospy.set\_param("p\_int",10)

rospy.set\_param("p\_double",3.14)

rospy.set\_param("p\_bool",True)

rospy.set\_param("p\_string","hello python")

rospy.set\_param("p\_list",["hello","haha","xixi"])

rospy.set\_param("p\_dict",{"name":"hulu","age":8})

# 修改

rospy.set\_param("p\_int",100)

**2.参数服务器获取参数**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

参数服务器操作之查询\_Python实现:

get\_param(键,默认值)

当键存在时，返回对应的值，如果不存在返回默认值

get\_param\_cached

get\_param\_names

has\_param

search\_param

"""

import rospy

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("get\_param\_p")

#获取参数

int\_value = rospy.get\_param("p\_int",10000)

double\_value = rospy.get\_param("p\_double")

bool\_value = rospy.get\_param("p\_bool")

string\_value = rospy.get\_param("p\_string")

p\_list = rospy.get\_param("p\_list")

p\_dict = rospy.get\_param("p\_dict")

rospy.loginfo("获取的数据:%d,%.2f,%d,%s",

int\_value,

double\_value,

bool\_value,

string\_value)

for ele in p\_list:

rospy.loginfo("ele = %s", ele)

rospy.loginfo("name = %s, age = %d",p\_dict["name"],p\_dict["age"])

# get\_param\_cached

int\_cached = rospy.get\_param\_cached("p\_int")

rospy.loginfo("缓存数据:%d",int\_cached)

# get\_param\_names

names = rospy.get\_param\_names()

for name in names:

rospy.loginfo("name = %s",name)

rospy.loginfo("-"\*80)

# has\_param

flag = rospy.has\_param("p\_int")

rospy.loginfo("包含p\_int吗？%d",flag)

# search\_param

key = rospy.search\_param("p\_int")

rospy.loginfo("搜索的键 = %s",key)

**3.参数服务器删除参数**

#! /usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

参数服务器操作之删除\_Python实现:

rospy.delete\_param("键")

键存在时，可以删除成功，键不存在时，会抛出异常

"""

import rospy

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("delete\_param\_p")

try:

rospy.delete\_param("p\_int")

except Exception as e:

rospy.loginfo("删除失败")

**4.设置权限:** 终端下进入 scripts 执行:chmod +x \*.py

**5.配置 CMakeLists.txt**

add\_executable(set\_update\_param src/set\_update\_param.cpp)

add\_executable(get\_param src/ get\_param.cpp)

add\_executable(delete\_param src/ delete\_param.cpp)

target\_link\_libraries(set\_update\_param ${catkin\_LIBRARIES})

target\_link\_libraries(get\_param ${catkin\_LIBRARIES})

target\_link\_libraries(delete\_param ${catkin\_LIBRARIES})

catkin\_install\_python(PROGRAMS

scripts/set\_update\_param\_p.py

scripts/get\_param \_p.py

scripts/delete\_param \_p.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

## 常用命令

作用:和之前介绍的文件系统操作命令比较，文件操作命令是静态的，操作的是磁盘上的文件，而上述命令是动态的，在ROS程序启动后，可以动态的获取运行中的节点或参数的相关信息。

案例:本节将借助于2.1、2.2和2.3的通信实现介绍相关命令的基本使用，并通过练习ROS内置的小海龟例程来强化命令的应用。

### 1.rosnode

rosnode 是用于获取节点信息的命令

rosnode ping 测试到节点的连接状态

rosnode list 列出活动节点

rosnode info 打印节点信息

rosnode machine 列出指定设备上节点

rosnode kill 杀死某个节点

rosnode cleanup 清除不可连接的节点, 启动乌龟节点，然后 ctrl + c 关闭，该节点并没被彻底清除，可以使用 cleanup 清除节点

### 2.rostopic

rostopic用于显示有关ROS 主题（话题）的调试信息，包括发布者，订阅者，发布频率和ROS消息。它还包含一个实验性Python库，用于动态获取有关主题的信息并与之交互。

rostopic bw 显示主题使用的带宽

rostopic delay 显示带有 header 的主题延迟

rostopic echo 打印消息到屏幕(---表示一个消息的结束及另一个的开始)

rostopic find 根据类型查找主题

rostopic hz 显示当前主题的消息发布频率

rostopic info 显示主题相关信息（**消息/服务的类型名（type）为功能包（package）名/（.msg/.srv）文件名**），发布者信息，订阅者信息）

rostopic list 显示所有活动状态下的主题

rostopic list -v : 获取话题详情(比如列出：发布者和订阅者个数...)

rostopic type 打印主题类型

rostopic pub /话题名称 消息类型 消息内容 可以直接调用命令向订阅者发布消息

为 小乌龟案例的 订阅者 发布一条运动信息：

rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist

"linear:

x: 1.0

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 2.0"

//只发布一次运动信息（这种长命令可用tab补全）

rostopic pub -r 10 /turtle1/cmd\_vel geometry\_msgs/Twist

"linear:

x: 1.0

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 2.0"

// 以 10HZ 的频率循环发送运动信息

### 3.rosmsg

rosmsg show 消息名称 显示消息描述

例：rosmsg show turtlesim/Pose

结果:

float32 x

float32 y

float32 theta

float32 linear\_velocity

float32 angular\_velocity

rosmsg info 显示消息信息,作用与 rosmsg show 一样

rosmsg list 列出所有消息

rosmsg md5 显示 md5 加密后的消息,一种校验算法，保证数据传输的一致性

rosmsg package 包名 显示某个功能包下的所有消息

rosmsg packages 列出包含消息的所有功能包

### 4.rosservice

调用部分服务时，如果对相关工作空间没有配置 path，需要进入工作空间调用 source ./devel/setup.bash

rosservice args 服务名 打印服务参数

rosservice call 使用提供的参数调用服务

为小乌龟的案例生成一只新的乌龟：

rosservice call /spawn "x: 1.0

y: 2.0

theta: 0.0

name: 'xxx'"

name: "xxx"

//生成一只叫 xxx 的乌龟

rosservice find 按照服务类型查找服务

rosservice info 打印有关服务的信息（node：提供它的节点，uri，type：srv类型（srv文件名），args：参数）

rosservice list 列出所有活动的服务

rosservice type 打印服务类型

rosservice uri 打印服务的 ROSRPC uri

### 5.rossrv

rossrv是用于显示有关ROS服务类型的信息的命令行工具，与 rosmsg 使用语法高度雷同。

rossrv show srv类型 显示服务详情

rossrv show turtlesim/Spawn

结果:

float32 x

float32 y

float32 theta

string name

---

string name

rossrv info 显示服务相关信息，作用与 rossrv show 一致

rossrv list 列出所有服务信息

rossrv md5 显示 md5 加密后的服务消息

rossrv package 显示某个包下所有服务消息

rossrv packages 显示包含服务消息的所有包

### 6.rosparam

用于使用YAML编码文件在参数服务器上获取和设置ROS参数。

rosparam set 键 值 设置参数

rosparam get 键 获取参数

rosparam load xxx.yaml (先准备 yaml 文件)从外部文件加载参数

rosparam dump yyy.yaml 将参数写出到外部文件

rosparam delete 键 删除参数

rosparam list 列出所有参数

## 通信机制实操

### 1.实操\_话题发布

需求描述:编码实现乌龟运动控制，让小乌龟做圆周运动。

实现分析:

乌龟运动控制实现，关键节点有两个，一个是乌龟运动显示节点 turtlesim\_node，另一个是控制节点，二者是订阅发布模式实现通信的，乌龟运动显示节点直接调用即可，运动控制节点之前是使用的 turtle\_teleop\_key通过键盘控制，现在需要自定义控制节点。

控制节点自实现时，首先需要了解控制节点与显示节点通信使用的话题与消息，可以使用ros命令结合计算图来获取。

了解了话题与消息之后，通过 C++ 或 Python 编写运动控制节点，通过指定的话题，按照一定的逻辑发布消息即可。

实现流程: **1.话题与消息获取**

**准备:** 先启动键盘控制乌龟运动案例。

**1.1话题获取**

通过计算图查看话题，启动计算图:rqt\_graph

或者通过 rostopic 列出话题: rostopic list

**1.2消息获取**

获取消息类型: rostopic info/type /turtle1/cmd\_vel

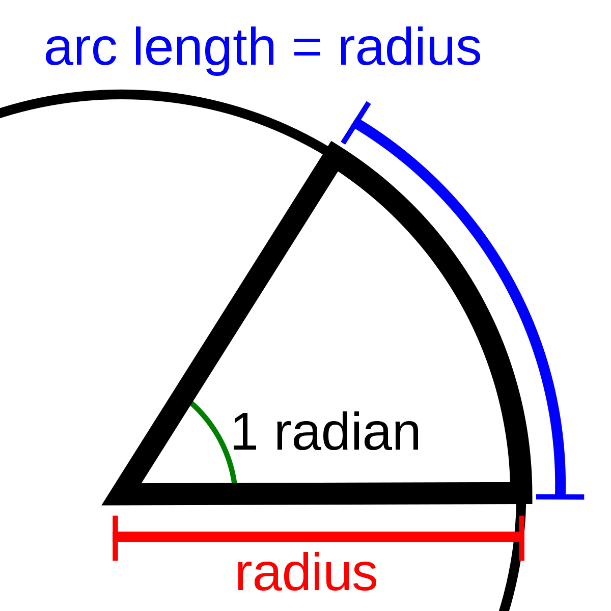
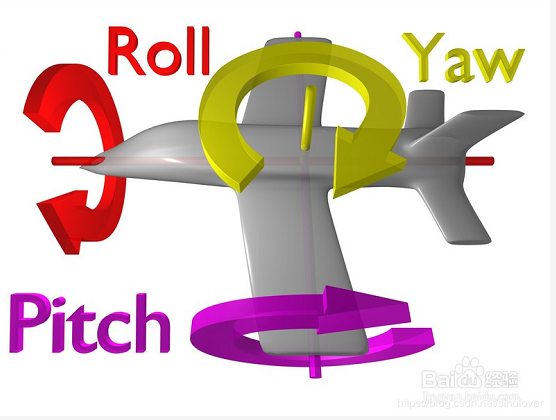
获取消息格式: rosmsg show/info geometry\_msgs/Twist

linear(线速度) 下的xyz分别对应在x、y和z方向上的速度(单位是 m/s)；

angular(角速度)下的xyz分别对应x轴上的翻滚、y轴上俯仰和z轴上偏航的速度(单位是rad/s)。(逆时针)

**补充:** **弧度: 单位弧度定义为圆弧长度等于半径时的圆心角。**

偏航、翻滚与俯仰坐标系图解:

****

**2.实现发布节点**

创建功能包需要依赖的功能包: roscpp rospy std\_msgs geometry\_msgs

**实现方案A: C++**

/\*

编写 ROS 节点，控制小乌龟画圆

准备工作:

1.获取topic(已知: /turtle1/cmd\_vel)

2.获取消息类型(已知: geometry\_msgs/Twist)

3.运行前，注意先启动 turtlesim\_node 节点

实现流程:

1.包含头文件

2.初始化 ROS 节点

3.创建发布者对象

4.循环发布运动控制消息

\*/

#include "ros/ros.h"

#include "geometry\_msgs/Twist.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"control");

ros::NodeHandle nh;

ros::Publisher pub = nh.advertise<geometry\_msgs::Twist>("/turtle1/cmd\_vel",1000);

// 循环发布运动控制消息

//组织消息

geometry\_msgs::Twist msg;

msg.linear.x = 1.0;

msg.linear.y = 0.0;

msg.linear.z = 0.0;

msg.angular.x = 0.0;

msg.angular.y = 0.0;

msg.angular.z = 2.0;

//设置发送频率

ros::Rate r(10);

//循环发送

while (ros::ok())

{

pub.publish(msg);

//ros::spinOnce();

}

return 0;

}

配置文件此处略

**实现方案B: Python**

#! /usr/bin/env python

import rospy

from geometry\_msgs.msg import Twist

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("control\_circle\_p")

pub = rospy.Publisher("/turtle1/cmd\_vel",Twist,queue\_size=1000)

rate = rospy.Rate(10)

msg = Twist()

msg.linear.x = 1.0

msg.linear.y = 0.0

msg.linear.z = 0.0

msg.angular.x = 0.0

msg.angular.y = 0.0

msg.angular.z = 0.5

while not rospy.is\_shutdown():

pub.publish(msg)

rate.sleep()

权限设置以及配置文件此处略

**3.运行**

首先，启动 roscore；

然后启动乌龟显示节点；rosrun turtlesim turtlesim\_node

最后执行运动控制节点；

或者写个launch文件

### 2.实操\_话题订阅

需求描述: 已知turtlesim中的乌龟显示节点，会发布当前乌龟的位姿(窗体中乌龟的坐标以及朝向)，要求控制乌龟运动，并时时打印当前乌龟的位姿。

实现流程: **1.话题与消息获取**

首先，需要启动乌龟显示以及运动控制节点并控制乌龟运动

获取话题: rostopic list

获取消息类型: rostopic type /turtle1/pose

获取消息格式: rosmsg info turtlesim/Pose

**2.实现订阅节点**

创建功能包需要依赖的功能包: roscpp rospy std\_msgs turtlesim

**实现方案A: C++**

/\*

订阅小乌龟的位姿: 时时获取小乌龟在窗体中的坐标并打印

准备工作:

1.获取话题名称 /turtle1/pose

2.获取消息类型 turtlesim/Pose

3.运行前启动 turtlesim\_node 与 turtle\_teleop\_key 节点

实现流程:

1.包含头文件

2.初始化 ROS 节点

3.创建 ROS 句柄

4.创建订阅者对象

5.回调函数处理订阅的数据

6.spin

\*/

#include "ros/ros.h"

#include "turtlesim/Pose.h"

void doPose(const turtlesim::Pose::ConstPtr& p){

ROS\_INFO("乌龟位姿信息:x=%.2f,y=%.2f,theta=%.2f,lv=%.2f,av=%.2f",

p->x,p->y,p->theta,p->linear\_velocity,p->angular\_velocity);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"sub\_pose");

ros::NodeHandle nh;

//创建订阅者对象，可不用加<>因为回调会判断类型

ros::Subscriber sub = nh.subscribe<turtlesim::Pose>("/turtle1/pose",1000,doPose);

// 回调函数处理订阅的数据

// spin

ros::spin();

return 0;

}

**实现方案B: Python**

#! /usr/bin/env python

import rospy

from turtlesim.msg import Pose

def doPose(data):

rospy.loginfo("乌龟坐标:x=%.2f, y=%.2f,theta=%.2f",data.x,data.y,data.theta)

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("sub\_pose\_p")

sub = rospy.Subscriber("/turtle1/pose",Pose,doPose,queue\_size=1000)

rospy.spin()

### 3.实操\_服务调用

需求描述:编码实现向 turtlesim 发送请求，在乌龟显示节点的窗体指定位置生成一乌龟，这是一个服务请求操作。

实现流程: **1.服务名称与服务消息获取**

首先，需要启动乌龟显示节点。

获取话题: rosservice list

获取消息类型: rosservice type /spawn

获取消息格式: rossrv info turtlesim/Spawn

**2.服务客户端实现**

创建功能包需要依赖的功能包: roscpp rospy std\_msgs turtlesim

**实现方案A:C++**

/\*

生成一只小乌龟

准备工作:

1.服务话题 /spawn

2.服务消息类型 turtlesim/Spawn

3.运行前先启动 turtlesim\_node 节点

实现流程:

1.包含头文件

需要包含 turtlesim 包下资源，注意在 package.xml 配置

2.初始化 ros 节点

3.创建 ros 句柄

4.创建 service 客户端

5.等待服务启动

6.发送请求

7.处理响应

\*/

#include "ros/ros.h"

#include "turtlesim/Spawn.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"set\_turtle");

ros::NodeHandle nh;

ros::ServiceClient client = nh.serviceClient<turtlesim::Spawn>("/spawn");

// 等待服务启动

// client.waitForExistence();

ros::service::waitForService("/spawn");

// 发送请求

turtlesim::Spawn spawn;

spawn.request.x = 1.0;

spawn.request.y = 1.0;

spawn.request.theta = 1.57;//逆时针为正方向

spawn.request.name = "my\_turtle";

bool flag = client.call(spawn);

// 处理响应结果

if (flag)

{

ROS\_INFO("新的乌龟生成,名字:%s",spawn.response.name.c\_str());//ROS\_INFO输出字符串必须.c\_str()

} else {

ROS\_INFO("乌龟生成失败！！！");

}

return 0;

}

配置文件此处略

**实现方案B:Python**

#! /usr/bin/env python

import rospy

from turtlesim.srv import Spawn,SpawnRequest,SpawnResponse

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("set\_turtle\_p")

client = rospy.ServiceProxy("/spawn",Spawn)

# 等待服务启动

client.wait\_for\_service()

# 发送请求

req = SpawnRequest()

req.x = 2.0

req.y = 2.0

req.theta = -1.57

req.name = "my\_turtle\_p"

try:

response = client.call(req)

# 处理响应

rospy.loginfo("乌龟创建成功!，叫:%s",response.name)

except expression as identifier:

rospy.loginfo("服务调用失败")

### 4.实操\_参数设置

需求描述: 修改turtlesim乌龟显示节点窗体的背景色，已知背景色是通过参数服务器的方式以 rgb 方式设置的。

实现流程: **1.服务名称与服务消息获取**

首先，需要启动乌龟显示节点。

获取参数列表: rosparam list

**2.参数修改**

**实现方案A:C++**

/\*

注意命名空间的使用。

\*/

#include "ros/ros.h"

int main(int argc, char \*argv[])

{

ros::init(argc,argv,"haha");

//1.

//ros::NodeHandle nh("turtlesim");

// ros::param::set("background\_r",0);

// ros::param::set("background\_g",0);

// ros::param::set("background\_b",0);

//2.

//ros::NodeHandle nh;

// ros::param::set("/turtlesim/background\_r",0);

// ros::param::set("/turtlesim/background\_g",0);

// ros::param::set("/turtlesim/background\_b",0);

//3.

ros::NodeHandle nh;

nh.setParam("/turtlesim/background\_r",0);

nh.setParam("/turtlesim/background\_g",0);

nh.setParam("/turtlesim/background\_b",0);

return 0;

}

配置文件此处略

**实现方案B:Python**

#! /usr/bin/env python

import rospy

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("hehe")

# rospy.set\_param("/turtlesim/background\_r",255)

# rospy.set\_param("/turtlesim/background\_g",255)

# rospy.set\_param("/turtlesim/background\_b",255)

rospy.set\_param("background\_r",255)

rospy.set\_param("background\_g",255)

rospy.set\_param("background\_b",255) # 调用时，需要传入 \_\_ns:=turtlesim

权限设置以及配置文件此处略

**3.运行**

首先，启动 roscore；

然后启动背景色设置节点；

最后启动乌龟显示节点；

**4.其他设置方式**

**方式1:修改小乌龟节点的背景色(命令行实现)**

rosparam set /turtlesim/background\_b 自定义数值

rosparam set /turtlesim/background\_g 自定义数值

rosparam set /turtlesim/background\_r 自定义数值

修改相关参数后，重启 turtlesim\_node 节点，背景色就会发生改变了

**方式2:启动节点时，直接设置参数**

rosrun turtlesim turtlesim\_node \_background\_r:=100 \_background\_g:=0 \_background\_b:=0

**方式3:通过launch文件传参**

<launch>

<node pkg="turtlesim" type="turtlesim\_node" name="set\_bg" output="screen">

<!-- launch 传参策略1 -->

<!-- <param name="background\_b" value="0" type="int" />

<param name="background\_g" value="0" type="int" />

<param name="background\_r" value="0" type="int" /> -->

<!-- launch 传参策略2 -->

<rosparam command="load" file="$(find demo\_turtle)/launch/color.yaml" />

</node>

</launch>

## 通信机制比较

三种通信机制中，参数服务器是一种数据共享机制，可以在不同的节点之间共享数据，话题通信与服务通信是在不同的节点之间传递数据的，三者是ROS中最基础也是应用最为广泛的通信机制。



# ROS通信机制进阶

## 常用API

### 1.初始化

**C++初始化**

/\*\* @brief ROS初始化函数。

\* 该函数可以解析并使用节点启动时传入的参数(通过参数设置节点名称、命名空间...)

\*

\* 该函数有多个重载版本，如果使用NodeHandle建议调用该版本。

\* \param argc 参数个数

\* \param argv 参数列表

\* \param name 节点名称，需要保证其唯一性，不允许包含命名空间

\* \param options 节点启动选项，被封装进了ros::init\_options

\*

void init(int &argc, char \*\*argv, const std::string& name, uint32\_t options = 0);

**Python初始化**

def init\_node(name, argv=None, anonymous=False, log\_level=None, disable\_rostime=False, disable\_rosout=False, disable\_signals=False, xmlrpc\_port=0, tcpros\_port=0):

"""

在ROS msater中注册节点

@param name: 节点名称，必须保证节点名称唯一，节点名称中不能使用命名空间(不能包含 '/')

@type name: str

@param anonymous: 取值为 true 时，为节点名称后缀随机编号

@type anonymous: bool

"""

### 2.话题与服务相关对象

**C++**

在 roscpp 中，话题和服务的相关对象一般由 NodeHandle 创建。

NodeHandle有一个重要作用是可以用于设置命名空间，这是后期的重点，但是本章暂不介绍。

**1.发布对象**

**对象获取:**

/\*\*

\* \brief 根据话题生成发布对象

\* 在 ROS master 注册并返回一个发布者对象，该对象可以发布消息

\* 使用示例如下:

\* ros::Publisher pub = handle.advertise<std\_msgs::Empty>("my\_topic", 1);

\*

\* \param topic 发布消息使用的话题

\* \param queue\_size 等待发送给订阅者的最大消息数量

\* \param latch (optional) 如果为 true,该话题发布的最后一条消息将被保存，并且后期当有订阅者连接时会将该消息发送给订阅者

\* \return 调用成功时，会返回一个发布对象

\*

template <class M>

Publisher advertise(const std::string& topic, uint32\_t queue\_size, bool latch = false)

**消息发布函数:**

template <typename M>

void publish(const M& message) const

**2.订阅对象**

**对象获取:**

/\*\*

\* \brief 生成某个话题的订阅对象

\* 该函数将根据给定的话题在ROS master 注册，并自动连接相同主题的发布方，每接收到一条消息，都会调用回调函数，并且传入该消息的共享指针，该消息不能被修改，因为可能其他订阅对象也会使用该消息。

\* 使用示例如下:

void callback(**const std\_msgs::Empty::ConstPtr &** message)

{

}

ros::Subscriber sub = handle.subscribe("my\_topic", 1, callback);

\* \param M [template] M 是指消息类型

\* \param topic 订阅的话题

\* \param queue\_size 消息队列长度，超出长度时，头部的消息将被弃用

\* \param fp 当订阅到一条消息时，需要执行的回调函数

\* \return 调用成功时，返回一个订阅者对象，失败时，返回空对象

\*

void callback(c**onst std\_msgs::Empty::ConstPtr &** message){...}

ros::NodeHandle nodeHandle;

ros::Subscriber sub = nodeHandle.subscribe("my\_topic", 1, callback);

if (sub) // Enter if subscriber is valid

{

...

}

\*/

template<class M>

Subscriber subscribe(const std::string& topic, uint32\_t queue\_size, void(\*fp)(const boost::shared\_ptr<M const>&), const TransportHints& transport\_hints = TransportHints())

**3.服务对象**

**对象获取:**

/\*\*

\* \brief 生成服务端对象

\* 该函数可以连接到 ROS master，并提供一个具有给定名称的服务对象。

\* 使用示例如下:

\verbatim

bool callback(**std\_srvs::Empty& request, std\_srvs::Empty& response**)

{

return true;

}

ros::ServiceServer service = handle.advertiseService("my\_service", callback);

\endverbatim

\*

\* \param service 服务的主题名称

\* \param srv\_func 接收到请求时，需要处理请求的回调函数

\* \return 请求成功时返回服务对象，否则返回空对象:

\verbatim

bool Foo::callback(std\_srvs::Empty& request, std\_srvs::Empty& response)

{

return true;

}

ros::NodeHandle nodeHandle;

Foo foo\_object;

ros::ServiceServer service = nodeHandle.advertiseService("my\_service", callback);

if (service) // Enter if advertised service is valid

{

...

}

\endverbatim

\*/

template<class MReq, class MRes>

ServiceServer advertiseService(const std::string& service, bool(\*srv\_func)(MReq&, MRes&))

**4.客户端对象**

**对象获取:**

/\*\*

\* @brief 创建一个服务客户端对象

\* 当清除最后一个连接的引用句柄时，连接将被关闭。

\* @param service\_name 服务主题名称

\*/

template<class Service>

ServiceClient serviceClient(const std::string& service\_name, bool persistent = false,

const M\_string& header\_values = M\_string())

**等待服务函数1:**

/\*\*

\* ros::service::waitForService("addInts");

\* \brief 等待服务可用，否则一致处于阻塞状态

\* \param service\_name 被"等待"的服务的话题名称

\* \param timeout 等待最大时常，默认为 -1，可以永久等待直至节点关闭

\* \return 成功返回 true，否则返回 false。

\*/

ROSCPP\_DECL bool waitForService(const std::string& service\_name, ros::Duration timeout = ros::Duration(-1));

**等待服务函数2:**

/\*\*

\* client.waitForExistence();

\* \brief 等待服务可用，否则一致处于阻塞状态

\* \param timeout 等待最大时常，默认为 -1，可以永久等待直至节点关闭

\* \return 成功返回 true，否则返回 false。

\*/

bool waitForExistence(ros::Duration timeout = ros::Duration(-1));

**请求发送函数:**

/\*\*

\* @brief 发送请求

\* 返回值为 bool 类型，true，请求处理成功，false，处理失败。

\*/

template<class Service>

bool call(Service& service)

**Python**

**1.发布对象**

**对象获取:**

class Publisher(Topic):

"""

在ROS master注册为相关话题的发布方

"""

def \_\_init\_\_(self, name, data\_class, subscriber\_listener=None, tcp\_nodelay=False, latch=False, headers=None, queue\_size=None):

"""

Constructor

@param name: 话题名称

@type name: str

@param data\_class: 消息类型

@param latch: 如果为 true,该话题发布的最后一条消息将被保存，并且后期当有订阅者连接时会将该消息发送给订阅者

@type latch: bool

@param queue\_size: 等待发送给订阅者的最大消息数量

@type queue\_size: int

"""

**消息发布函数:**

def publish(self, \*args, \*\*kwds):

**2.订阅对象**

**对象获取:**

class Subscriber(Topic):

"""

类注册为指定主题的订阅者，其中消息是给定类型的。

"""

def \_\_init\_\_(self, name, data\_class, callback=None, callback\_args=None,

queue\_size=None, buff\_size=DEFAULT\_BUFF\_SIZE, tcp\_nodelay=False):

"""

Constructor.

@param name: 话题名称

@type name: str

@param data\_class: 消息类型

@type data\_class: L{Message} class

@param callback: 处理订阅到的消息的回调函数

@type callback: fn(msg, cb\_args)

@param queue\_size: 消息队列长度，超出长度时，头部的消息将被弃用

"""

**3.服务对象**

**对象获取:**

class Service(ServiceImpl):

"""

声明一个ROS服务

使用示例::

s = Service('getmapservice', GetMap, get\_map\_handler)

"""

def \_\_init\_\_(self, name, service\_class, handler,

buff\_size=DEFAULT\_BUFF\_SIZE, error\_handler=None):

"""

@param name: 服务主题名称 ``str``

@param service\_class:服务消息类型

@param handler: 回调函数，处理请求数据，并返回响应数据

@type handler: fn(req)->resp

"""

**4.客户端对象**

**对象获取:**

class ServiceProxy(\_Service):

"""

创建一个ROS服务的句柄

示例用法::

add\_two\_ints = ServiceProxy('add\_two\_ints', AddTwoInts)

resp = add\_two\_ints(1, 2)

"""

def \_\_init\_\_(self, name, service\_class, persistent=False, headers=None):

"""

ctor.

@param name: 服务主题名称

@type name: str

@param service\_class: 服务消息类型

@type service\_class: Service class

"""

**请求发送函数:**

def call(self, \*args, \*\*kwds):

**等待服务函数:**

def wait\_for\_service(service, timeout=None):

"""

调用该函数时，程序会处于阻塞状态直到服务可用

@param service: 被等待的服务话题名称

@type service: str

@param timeout: 超时时间

@type timeout: double|rospy.Duration

"""

### 3.回旋函数

**C++**

**1.spinOnce()**

/\*\*

\* \brief 处理一轮回调

\* 一般应用场景:

\* 在循环体内，处理所有可用的回调函数

\*/

ROSCPP\_DECL void spinOnce();

**2.spin()**

/\*\*

\* \brief 进入循环处理回调

\*/

ROSCPP\_DECL void spin();

**3.二者比较**

相同点:二者都用于处理回调函数；

不同点:ros::spin() 是进入了循环执行回调函数，而 ros::spinOnce() 只会执行一次回调函数(没有循环)，在 ros::spin() 后的语句不会执行到，而 ros::spinOnce() 后的语句可以执行。

**Python**

def spin():

### 4.时间

**C++**

**1.时刻**

获取时刻，或是设置指定时刻:

ros::init(argc,argv,"hello\_time");

ros::NodeHandle nh;//必须创建句柄，否则时间没有初始化，导致后续API调用失败

ros::Time right\_now = ros::Time::now();//将当前时刻封装成对象

ROS\_INFO("当前时刻:%.2f",right\_now.toSec());//获取距离 1970年01月01日 00:00:00 的秒数

ROS\_INFO("当前时刻:%d",right\_now.sec);//**sec是字段，返回整型%d**，获取距离 1970年01月01日 00:00:00 的秒数

ros::Time someTime(100,100000000);// 参数1:秒数 参数2:纳秒

ROS\_INFO("时刻:%.2f",someTime.toSec()); //100.10

ros::Time someTime2(100.3);//直接传入 double 类型的秒数

ROS\_INFO("时刻:%.2f",someTime2.toSec()); //100.30

**2.持续时间**

设置一个时间区间(间隔):

ROS\_INFO("当前时刻:%.2f",ros::Time::now().toSec());

ros::Duration du(10);//持续10秒钟,参数是double类型的，以秒为单位

du.sleep();//按照指定的持续时间休眠

ROS\_INFO("持续时间:%.2f",du.toSec());//将持续时间换算成秒

ROS\_INFO("当前时刻:%.2f",ros::Time::now().toSec());

**3.持续时间与时刻运算**

为了方便使用，ROS中提供了时间与时刻的运算:

ROS\_INFO("时间运算");

ros::Time now = ros::Time::now();

ros::Duration du1(10);

ros::Duration du2(20);

ROS\_INFO("当前时刻:%.2f",now.toSec());

//1.time 与 duration 运算

ros::Time after\_now = now + du1;

ros::Time before\_now = now - du1;

ROS\_INFO("当前时刻之后:%.2f",after\_now.toSec());

ROS\_INFO("当前时刻之前:%.2f",before\_now.toSec());

//2.duration 之间相互运算

ros::Duration du3 = du1 + du2;

ros::Duration du4 = du1 - du2;

ROS\_INFO("du3 = %.2f",du3.toSec());

ROS\_INFO("du4 = %.2f",du4.toSec());

//PS: time 与 time 不可以运算

// ros::Time nn = now + before\_now;//异常

**4.设置运行频率**

ros::Rate rate(1);//指定频率

while (true)

{

ROS\_INFO("-----------code----------");

rate.sleep();//休眠，休眠时间 = 1 / 频率。

}

**5.定时器**

ROS 中内置了专门的定时器，可以实现与 ros::Rate 类似的效果:

ros::NodeHandle nh;//必须创建句柄，否则时间没有初始化，导致后续API调用失败

// ROS 定时器

/\*\*

\* \brief 创建一个定时器，按照指定频率调用回调函数。

\*

\* \param period 时间间隔

\* \param callback 回调函数

\* \param oneshot 如果设置为 true,只执行一次回调函数，设置为 false,就循环执行。

\* \param autostart 如果为true，返回已经启动的定时器,设置为 false，需要手动启动。

\*/

//Timer createTimer(Duration period, const TimerCallback& callback, bool oneshot = false,

// bool autostart = true) const;

// ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing);

ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing,true);//只执行一次

// ros::Timer timer = nh.createTimer(ros::Duration(0.5),doSomeThing,false,false);//需要手动启动

// timer.start();

ros::spin(); //必须 spin

定时器的回调函数:

void doSomeThing(**const ros::TimerEvent &event**){

ROS\_INFO("-------------");

//to\_string()函数将数字常量转化为字符串

ROS\_INFO("event:%s",std::to\_string(event.current\_real.toSec()).c\_str());

ROS\_INFO("event:%.2f", event.current\_real.toSec());//这个也行

}

**补充：**

struct TimerEvent

{

Time last\_expected; //上一回调函数应该发生的时刻

Time last\_real; //上一回调函数实际发生的时刻

Time current\_expected; //当前回调函数应该发生的时刻

Time current\_real; //当前回调函数实际发生的时刻

struct

{

WallDuration last\_duration; //包含上一回调的时间间隔

} profile;

};

**Python**

**1.时刻**

# 获取当前时刻

right\_now = rospy.Time.now()

rospy.loginfo("当前时刻:%.2f",right\_now.to\_sec())

rospy.loginfo("当前时刻:%.2f",right\_now.to\_nsec())#纳秒

# 自定义时刻

some\_time1 = rospy.Time(1234.567891011)

rospy.loginfo("设置时刻1:%.2f",some\_time1.to\_sec())

# 从时间创建对象

# some\_time3 = rospy.Time.from\_seconds(543.21)

some\_time3 = rospy.Time.from\_sec(543.21) # from\_sec 替换了 from\_seconds

rospy.loginfo("设置时刻3:%.2f",some\_time3.to\_sec())

**2.持续时间**

# 持续时间相关API

rospy.loginfo("持续时间测试开始.....")

du = rospy.Duration(3.3)

rospy.loginfo("du1 持续时间:%.2f",du.to\_sec())

rospy.sleep(du) #休眠函数

rospy.loginfo("持续时间测试结束.....")

**3.持续时间与时刻运算**

rospy.loginfo("时间运算")

now = rospy.Time.now()

du1 = rospy.Duration(10)

du2 = rospy.Duration(20)

rospy.loginfo("当前时刻:%.2f",now.to\_sec())

before\_now = now - du1

after\_now = now + du1

dd = du1 + du2

# now = now + now #非法

rospy.loginfo("之前时刻:%.2f",before\_now.to\_sec())

rospy.loginfo("之后时刻:%.2f",after\_now.to\_sec())

rospy.loginfo("持续时间相加:%.2f",dd.to\_sec())

**4.设置运行频率**

rate = rospy.Rate(0.5)

while not rospy.is\_shutdown():

rate.sleep() #休眠

rospy.loginfo("+++++++++++++++")

**5.定时器**

"""

def \_\_init\_\_(self, period, callback, oneshot=False, reset=False):

Constructor.

@param period: 回调函数的时间间隔

@type period: rospy.Duration

@param callback: 回调函数

@type callback: function taking rospy.TimerEvent

@param oneshot: 设置为True，就只执行一次，否则循环执行

@type oneshot: bool

@param reset: if True, timer is reset when rostime moved backward. [default: False]

@type reset: bool

"""

rospy.Timer(rospy.Duration(1),doMsg)

# rospy.Timer(rospy.Duration(1),doMsg,True) # 只执行一次

rospy.spin()

回调函数:

def doMsg(event):

rospy.loginfo("+++++++++++")

rospy.loginfo("当前时刻:%s",str(event.current\_real))

### 5.其他函数

在发布实现时，一般会循环发布消息，循环的判断条件一般由节点状态来控制，C++中可以通过 ros::ok() 来判断节点状态是否正常，而 python 中则通过 rospy.is\_shutdown() 来实现判断，导致节点退出的原因主要有如下几种:

(1)节点接收到了关闭信息，比如常用的 ctrl + c 快捷键就是关闭节点的信号；

(2)同名节点启动，导致现有节点退出；

(3)程序中的其他部分调用了节点关闭相关的API(C++中是ros::shutdown()，python中是rospy.signal\_shutdown())

另外，日志相关的函数也是极其常用的，在ROS中日志被划分成如下级别:

DEBUG(调试):只在调试时使用，此类消息不会输出到控制台；

INFO(信息):标准消息，一般用于说明系统内正在执行的操作；

WARN(警告):提醒一些异常情况，但程序仍然可以执行；

ERROR(错误):提示错误信息，此类错误会影响程序运行；

FATAL(严重错误):此类错误将阻止节点继续运行。

**C++**

**1.节点状态判断**

/\*\* \brief 检查节点是否已经退出

\* ros::shutdown() 被调用且执行完毕后，该函数将会返回 false

\* \return true 如果节点还健在, false 如果节点已经火化了。

\*/

bool ok();

**2.节点关闭函数**

void shutdown();

**3.日志函数**

ROS\_DEBUG("hello,DEBUG"); //不会输出

ROS\_INFO("hello,INFO"); //默认白色字体

ROS\_WARN("Hello,WARN"); //默认黄色字体

ROS\_ERROR("hello,ERROR");//默认红色字体

ROS\_FATAL("hello,FATAL");//默认红色字体

**Python**

**1.节点状态判断**

def is\_shutdown():

"""

@return: True 如果节点已经被关闭

@rtype: bool

"""

**2.节点关闭函数**

def signal\_shutdown(reason):

"""

关闭节点

@param reason: 节点关闭的原因，是一个字符串

@type reason: str

"""

def on\_shutdown(h):

"""

节点被关闭时调用的函数

@param h: 关闭时调用的回调函数，此函数无参

@type h: fn()

"""

**3.日志函数**

rospy.logdebug("hello,debug") #不会输出

rospy.loginfo("hello,info") #默认白色字体

rospy.logwarn("hello,warn") #默认黄色字体

rospy.logerr("hello,error") #默认红色字体

rospy.logfatal("hello,fatal") #默认红色字体

## ROS中的头文件与源文件

### 1.自定义头文件调用

需求:设计头文件，可执行文件本身作为源文件。

流程: **1.头文件**

在功能包下的 include/功能包名 目录下新建头文件: hello.h，示例内容如下:

//头文件保护，防止重定义（多个cpp文件包含一个h文件），与#pragma once类似，当第一次包含h时，由于未定义\_HELLO\_H条件为真，编译（执行）#endif前代码，若已经定义则忽略这一段代码

#ifndef \_HELLO\_H

#define \_HELLO\_H

namespace hello\_ns{

class HelloPub {

public:

void run();

};

}

#endif

**注意:** 在 VScode 中，为了后续包含头文件时不抛出异常，请配置 .vscode 下 c\_cpp\_properties.json 的 includepath属性: "/home/用户/工作空间/src/功能包/include/\*\*"(可以在include下打开终端输入pwd复制)

**2.可执行文件**

在 src 目录下新建文件:hello.cpp，示例内容如下:

#include "ros/ros.h"

#include "demo\_head\_source/hello.h"

namespace hello\_ns {

void HelloPub::run(){

ROS\_INFO("自定义头文件的使用....");

}

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ros::init(argc,argv,"test\_head");

hello\_ns::HelloPub helloPub;

helloPub.run();

return 0;

}

**3.配置文件**

配置CMakeLists.txt文件，头文件相关配置如下:

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

可执行配置文件配置方式与之前一致:

add\_executable(hello src/test\_head.cpp)

add\_dependencies(hello ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(hello

${catkin\_LIBRARIES}

)

最后，编译并执行，控制台可以输出自定义的文本信息。

### 2.自定义源文件调用

需求:设计头文件与源文件，在可执行文件中包含头文件。

流程: **1.头文件**

头文件设置于类似前面，在功能包下的 include/功能包名 目录下新建头文件: haha.h，示例内容如下:

#ifndef \_HAHA\_H

#define \_HAHA\_H

namespace hello\_ns {

class My {

public:

void run();

};

}

#endif

注意:在 VScode 中，为了后续包含头文件时不抛出异常，请配置 .vscode 下 c\_cpp\_properties.json 的 includepath属性: "/home/用户/工作空间/src/功能包/include/\*\*"(一个pkg包含多个h每次重新写一下includepath相应项)

**2.源文件**

在 src 目录下新建文件:source.cpp，示例内容如下:

#include "demo\_head\_source/haha.h"

#include "ros/ros.h"

namespace hello\_ns{

void My::run(){

ROS\_INFO("自定义源文件的使用....");

}

}

**3.可执行文件**

在 src 目录下新建文件: test\_source.cpp，示例内容如下:

#include "ros/ros.h"

#include "demo\_head\_source/haha.h"

int main(int argc, char \*argv[])

{

ros::init(argc,argv,"test\_source");

hello\_ns::My my;

my.run();

return 0;

}

**4.配置文件**

头文件与源文件相关配置:

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

## 声明C++库

add\_library(head

include/demo\_head\_source/haha.h

#include/${PROJECT\_NAME}/haha.h

src/source.cpp)

add\_dependencies(head ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

target\_link\_libraries(head

${catkin\_LIBRARIES}

)

可执行文件配置:

add\_executable(source src/test\_source.cpp)

add\_dependencies(source ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

#此处需要添加之前设置的 head 库

target\_link\_libraries(source

head

${catkin\_LIBRARIES}

)

### 3.Python模块导入

需求:首先新建一个Python文件A，再创建Python文件UseA，在UseA中导入A并调用A的实现。

**1.新建两个Python文件并使用import导入**

文件A实现(包含一个变量):

#! /usr/bin/env python

num = 1000

文件use\_A实现:

#! /usr/bin/env python

# coding=utf-8

import rospy

import os

import sys

import A

# rosrun的执行路径是工作空间的路径,在工作空间下无法找到依赖的模块

path = os.path.abspath(".")

# 核心

sys.path.insert(0,path + "src/demo\_head\_source/scripts")

#sys.path.insert(0,"/home/xuwenyu/test\_ws/src/demo\_head\_source/scripts")

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("usa\_A")

rospy.loginfo("执行时的参考路径:%s",path)

rospy.loginfo("num = %d",A.num)

**2.添加可执行权限，编辑配置文件并执行**

# ROS运行管理

## ROS元功能包

概念:MetaPackage是Linux的一个文件管理系统的概念。是ROS中的一个虚包，里面没有实质性的内容，但是它依赖了其他的软件包，通过这种方法可以把其他包组合起来，我们可以认为它是一本书的目录索引，告诉我们这个包集合中有哪些子包，并且该去哪里下载。

例如：sudo apt install ros-melodic-desktop-full 命令安装ros时就使用了元功能包，该元功能包依赖于ROS中的其他一些功能包，安装该包时会一并安装依赖。

还有一些常见的MetaPackage：navigation moveit! turtlebot3 ....

**实现**

首先:新建一个功能包

然后:修改package.xml ,内容如下:

<exec\_depend>被集成的功能包</exec\_depend>

.....

<export>

<metapackage />

</export>

最后:修改 CMakeLists.txt,内容如下:

cmake\_minimum\_required(VERSION 3.0.2)

project(demo)

find\_package(catkin REQUIRED)

catkin\_metapackage()

PS:CMakeLists.txt 中不可以有空行。

## ROS节点运行管理launch文件

概念：launch 文件是一个 XML 格式的文件，可以启动本地和远程的多个节点，还可以在参数服务器中设置参数。

### 1.launch文件标签之launch

<launch>标签是所有 launch 文件的根标签，充当其他标签的容器

**1.属性**

deprecated = "弃用声明"

告知用户当前 launch 文件已经弃用，会输出warn

**2.子级标签**

所有其它标签都是launch的子级

**3.演示**

<launch deprecated = "弃用声明">

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" />

<node pkg="turtlesim" type="turtle\_teleop\_key" name="my\_key" output="screen" />

</launch>

### 2.launch文件标签之node

<node>标签用于指定 ROS 节点，是最常见的标签，需要注意的是: roslaunch 命令不能保证按照 node 的声明顺序来启动节点(节点的启动是多进程的)

**1.属性**

pkg="包名"

节点所属的包

type="nodeType"

节点类型(与之相同名称的可执行文件名称)

name="nodeName"

节点名称(在 ROS 网络拓扑中节点的名称)

args="xxx xxx xxx" (可选)

将参数传递给节点

machine="机器名"

在指定机器上启动节点

respawn="true | false" (可选)

如果节点退出，是否自动重启

respawn\_delay=" N" (可选)

如果 respawn 为 true, 那么延迟 N 秒后启动节点

required="true | false" (可选)

该节点是否必须，如果为 true,那么如果该节点退出，将杀死整个 roslaunch

ns="xxx" (可选)

在指定命名空间 xxx 中启动节点

clear\_params="true | false" (可选)

在启动前，删除节点的私有空间的所有参数

output="log | screen" (可选)

日志发送目标，可以设置为 log 日志文件，或 screen 屏幕,默认是 log

**2.子级标签**

env 环境变量设置

remap 重映射节点名称

rosparam 参数设置

param 参数设置

**3.演示**

<launch>

<!-- respawn="true | false" true:如果节点退出,是否自动重启,respawn\_delay="N" 如果 respawn 为 true, 那么延迟 N 秒后启动节点 -->

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" respawn="true" respawn\_delay="5"/>

<!-- required="true | false"该节点是否必须，如果为 true,那么如果该节点退出，将杀死整个 -->

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" required="true" />

<!-- ns="xxx" 在指定命名空间 xxx 中启动节点,可使用rosnode list查看加与不加区别 -->

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" ns="hello"/>

<node pkg="turtlesim" type="turtle\_teleop\_key" name="my\_key" output="screen" />

</launch>

### 3.launch文件标签之include

include标签用于将另一个 xml 格式的 launch 文件导入到当前文件

**1.属性**

file="$(find 包名)/xxx.launch"

要包含的文件路径

ns="xxx" (可选)

在指定命名空间导入文件

**2.子级标签**

env 环境变量设置

arg 将参数传递给被包含的文件

**3.演示**

<launch>

<include file="$(find demo\_launch)/launch/turtle.launch" />

</launch>

### 4.launch文件标签之remap

用于话题重命名

**1.属性**

from="xxx"

原始话题名称

to="yyy"

目标名称

**2.演示**

<!-- 键盘控制:rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py 安装:sudo apt-get install ros-melodic-teleop-twist-keyboard

这里remap改的是订阅的话题 -->

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen">

<remap from="turtle1/cmd\_vel" to="cmd\_vel"/>

</node>

### 5.launch文件标签之param

<param>标签主要用于在参数服务器上设置参数，参数源可以在标签中通过 value 指定，也可以通过外部文件加载，在<node>标签中时，相当于私有命名空间。

**1.属性**

name="命名空间/参数名"

参数名称，可以包含命名空间

value="xxx" (可选)

定义参数值，如果此处省略，必须指定外部文件作为参数源

type="str | int | double | bool | yaml" (可选)

指定参数类型，如果未指定，roslaunch 会尝试确定参数类型，规则如下:

如果包含 '.' 的数字解析未浮点型，否则为整型

"true" 和 "false" 是 bool 值(不区分大小写)

其他是字符串

**2.演示**

<!-- 1.param:node外

<param name="A" type="int" value="100" />

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" >

2.param:node内(会包含节点名称my\_turtle)

<param name="B" type="double" value="3.14" />

</node> -->

### 6.launch文件标签之rosparam

<rosparam>标签可以从 YAML 文件导入参数，或将参数导出到 YAML 文件，也可以用来删除参数，<rosparam>标签在<node>标签中时被视为私有。

**1.属性**

command="load | dump | delete" (可选，默认 load)

加载、导出或删除参数

file="$(find xxxxx)/xxx/yyy...."

加载或导出到的 yaml 文件

param="参数名称"

ns="命名空间" (可选)

**2.演示**

<!-- 1.rosparam:node外，yaml文件每个参数:后面要有空格

<rosparam command="load" file="$(find demo\_launch)/launch/params.yaml" />

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" >

2.rosparam:node内(会包含节点名称my\_turtle)

<rosparam command="load" file="$(find demo\_launch)/launch/params.yaml" />

</node> -->

<launch>

<!-- 为了导出全部参数需要另建一个launch文件并后启动 -->

<rosparam command="dump" file="$(find demo\_launch)/launch/params\_out.yaml" />

<!-- 删除某参数 -->

<rosparam command="delete" param="bg\_R" />

</launch>

### 7.launch文件标签之group

<group>标签可以对节点分组，具有 ns 属性，可以让节点归属某个命名空间

**1.属性**

ns="名称空间" (可选)

clear\_params="true | false" (可选)

启动前，是否删除组名称空间的所有参数(慎用....此功能危险)

**2.子级标签**

除了launch 标签外的其他标签

**3.演示**

<launch>

<group ns="first">

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" />

<node pkg="turtlesim" type="turtle\_teleop\_key" name="my\_key" output="screen"/>

</group>

<group ns="second">

<node pkg="turtlesim" type="turtlesim\_node" name="my\_turtle" output="screen" />

<node pkg="turtlesim" type="turtle\_teleop\_key" name="my\_key" output="screen"/>

</group>

</launch>

### 8.launch文件标签之arg

<arg>标签是用于动态传参，类似于函数的参数，可以增强launch文件的灵活性

**1.属性**

name="参数名称"

default="默认值" (可选)

value="数值" (可选)

不可以与 default 并存

doc="描述"

参数说明

**2.演示**

launch文件传参语法实现：

<launch>

<!-- 设置多个参数,这些参数使用同一值 -->

<!-- 1 -->

<param name=“A” value="11" />

<param name=“B” value="11" />

<param name=“C” value="11" />

<!-- 2 -->

<arg name="car\_length" default="11"/>

<param name=“A” value="$(arg car\_length)" />

<param name=“B” value="$(arg car\_length)" />

<param name=“C” value="$(arg car\_length)" />

<node pkg=“包名” type=“可执行文件名” name=“节点名” args=“$(arg arg-name（例如上面的car\_length）)” />

<!—或者 -->

<node pkg=“包名” type=“可执行文件名” name=“节点名” args=“xxx（对应节点中的argv[1]）,程序中有yyy=’”argv[1]” />

</launch>

命令行调用launch传参：roslaunch arg.launch xxx:=值

## ROS工作空间覆盖

所谓工作空间覆盖，是指不同工作空间中，存在重名的功能包的情形。

**实现：**

0.新建工作空间A与工作空间B，两个工作空间中都创建功能包: turtlesim。

1.在 ~/.bashrc 文件下追加当前工作空间的 bash 格式如下:

source /home/用户/路径/工作空间A/devel/setup.bash

source /home/用户/路径/工作空间B/devel/setup.bash

2.新开命令行:source .bashrc加载环境变量

3.查看ROS环境环境变量echo $ROS\_PACKAGE\_PATH

结果:自定义工作空间B:自定义空间A:系统内置空间

4.调用命令:roscd turtlesim会进入自定义工作空间B

**原因：**ROS 会解析 .bashrc 文件，并生成 ROS\_PACKAGE\_PATH ROS包路径，该变量中按照 .bashrc 中配置设置工作空间优先级，在设置时需要遵循一定的原则:ROS\_PACKAGE\_PATH 中的值，和 .bashrc 的配置顺序相反--->后配置的优先级更高，如果更改自定义空间A与自定义空间B的source顺序，那么调用时，将进入工作空间A。

隐患：存在安全隐患，比如当前工作空间B优先级更高，意味着当程序调用 turtlesim 时，不会调用工作空间A也不会调用系统内置的 turtlesim，如果工作空间A在实现时有其他功能包依赖于自身的 turtlesim，而按照ROS工作空间覆盖的涉及原则，那么实际执行时将会调用工作空间B的turtlesim，从而导致执行异常，出现安全隐患。

**BUG 说明:** 当在 .bashrc 文件中 source 多个工作空间后，可能出现的情况，在 ROS PACKAGE PATH 中只包含两个工作空间，可以删除自定义工作空间的 build 与 devel 目录，重新 catkin\_make，然后重新载入 .bashrc 文件，问题解决。

## ROS节点名称重名

场景:ROS 中创建的节点是有名称的，C++初始化节点时通过API:ros::init(argc,argv,"xxxx");来定义节点名称，在Python中初始化节点则通过 rospy.init\_node("yyyy") 来定义节点名称。在ROS的网络拓扑中，是不可以出现重名的节点的，因为假设可以重名存在，那么调用时会产生混淆，这也就意味着，不可以启动重名节点或者同一个节点启动多次，的确，在ROS中如果启动重名节点的话，之前已经存在的节点会被直接关闭，但是如果有这种需求的话，在ROS中给出的解决策略是使用命名空间或名称重映射。

命名空间就是为名称添加前缀，名称重映射是为名称起别名。这两种策略都可以解决节点重名问题，两种策略的实现途径有多种:rosrun 命令，launch 文件，编码实现

**案例**

启动两个 turtlesim\_node 节点，当然如果直接打开两个终端，直接启动，那么第一次启动的节点会关闭

### 1.rosrun设置命名空间与重映射

**1.rosrun设置命名空间**

**1.1设置命名空间演示**

语法: rosrun 包名 节点名 \_\_ns:=新名称

rosrun turtlesim turtlesim\_node \_\_ns:=/xxx

rosrun turtlesim turtlesim\_node \_\_ns:=/yyy

**1.2运行结果**

rosnode list查看节点信息,显示结果

**2.rosrun名称重映射**

**2.1为节点起别名**

语法: rosrun 包名 节点名 \_\_name:=新名称

rosrun turtlesim turtlesim\_node \_\_name:=t1 | rosrun turtlesim turtlesim\_node /turtlesim:=t1(不适用于python)

rosrun turtlesim turtlesim\_node \_\_name:=t2 | rosrun turtlesim turtlesim\_node /turtlesim:=t2(不适用于python)

**2.2运行结果**

rosnode list查看节点信息,显示结果

**3.rosrun命名空间与名称重映射叠加**

**3.1设置命名空间同时名称重映射**

语法: rosrun 包名 节点名 \_\_ns:=新名称 \_\_name:=新名称

rosrun turtlesim turtlesim\_node \_\_ns:=/xxx \_\_name:=tn

**3.2运行结果**

rosnode list查看节点信息,显示结果

使用环境变量也可以设置命名空间,启动节点前在终端键入如下命令:

export ROS\_NAMESPACE=xxxx

### 2.launch文件设置命名空间与重映射

介绍 launch 文件的使用语法时，在 node 标签中有两个属性: name 和 ns，二者分别是用于实现名称重映射与命名空间设置的。

**1.launch文件**

<launch>

<node pkg="turtlesim" type="turtlesim\_node" name="t1" />

<node pkg="turtlesim" type="turtlesim\_node" name="t2" />

<node pkg="turtlesim" type="turtlesim\_node" name="t1" ns="hello"/>

</launch>

**在 node 标签中，name 属性是必须的，ns 可选。**

**2.运行**

rosnode list查看节点信息,显示结果

### 3.编码设置命名空间与重映射

如果自定义节点实现，那么可以更灵活的设置命名空间与重映射实现。

**1.C++ 实现:重映射**

**1.1名称别名设置**

核心代码:ros::init(argc,argv,"zhangsan",ros::init\_options::AnonymousName);

**1.2执行**

会在名称后面添加时间戳。

**2.C++ 实现:命名空间**

**2.1命名空间设置**

std::map<std::string, std::string> map;

map["\_\_ns"] = "xxxx";

ros::init(map,"wangqiang");

**2.2执行**

节点名称设置了命名空间。

**3.Python 实现:重映射**

**3.1名称别名设置**

核心代码:rospy.init\_node("lisi",anonymous=True)

**3.2执行**

会在节点名称后缀时间戳。

## ROS话题名称重名

在 ROS 中节点终端，不同的节点之间通信都依赖于话题，话题名称也可能出现重复的情况，这种情况下，系统虽然不会抛出异常，但是可能导致订阅的消息非预期的，从而导致节点运行异常。这种情况下需要将两个节点的话题名称由相同修改为不同。

又或者，两个节点是可以通信的，两个节点之间使用了相同的消息类型，但是由于，话题名称不同，导致通信失败。这种情况下需要将两个节点的话题名称由不同修改为相同。

在ROS中给出的解决策略与节点名称重命名类似，也是使用名称重映射或为名称添加前缀。根据前缀不同，有全局、相对、和私有三种类型之分。

全局(参数名称直接参考ROS系统，与节点命名空间平级)

相对(参数名称参考的是节点的命名空间，与节点名称平级)

私有(参数名称参考节点名称，是节点名称的子级)

名称重映射是为名称起别名，为名称添加前缀，该实现比节点重名更复杂些，不单是使用命名空间作为前缀、还可以使用节点名称作为前缀。两种策略的实现途径有多种:rosrun 命令，launch 文件，编码实现

案例：在ROS中提供了一个比较好用的键盘控制功能包: ros-noetic-teleop-twist-keyboard，该功能包，可以控制机器人的运动，作用类似于乌龟的键盘控制节点，可以使用 sudo apt install ros-melodic-teleop-twist-keyboard 来安装该功能包，然后执行: rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py，在启动乌龟显示节点，不过此时前者不能控制乌龟运动，因为，二者使用的话题名称不同，前者使用的是 cmd\_vel话题，后者使用的是 /turtle1/cmd\_vel话题。需要将话题名称修改为一致，才能使用

### 1.rosrun设置话题重映射

rosrun名称重映射语法: rorun 包名 节点名 话题名:=新话题名称

**1.方案1**

将 teleop\_twist\_keyboard 节点的话题设置为/turtle1/cmd\_vel

启动键盘控制节点:rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py /cmd\_vel:=/turtle1/cmd\_vel

启动乌龟显示节点: rosrun turtlesim turtlesim\_node

**2.方案2**

将乌龟显示节点的话题设置为 /cmd\_vel

启动键盘控制节点:rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

启动乌龟显示节点: rosrun turtlesim turtlesim\_node /turtle1/cmd\_vel:=/cmd\_vel

### 2.launch文件设置话题重映射

launch 文件设置话题重映射语法:

<node pkg="xxx" type="xxx" name="xxx">

<remap from="原话题" to="新话题" />

</node>

### 3.编码设置话题名称

话题的名称与节点的命名空间、节点的名称是有一定关系的，话题名称大致可以分为三种类型:

全局(话题参考ROS系统，与节点命名空间平级)

相对(话题参考的是节点的命名空间，与节点名称平级)

私有(话题参考节点名称，是节点名称的子级)

**1.C++ 实现**

准备:

1.初始化节点设置一个节点名称

ros::init(argc,argv,"hello")

2.设置不同类型的话题

3.启动节点时，传递一个 \_\_ns:= xxx

4.节点启动后，使用 rostopic 查看话题信息

**1.1全局名称**

格式:以/开头的名称，和节点名称无关(话题参考ROS系统，与节点命名空间平级)

比如:/xxx/yyy/zzz

示例1:ros::Publisher pub = nh.advertise<std\_msgs::String>("/chatter",1000);

结果1:/chatter

示例2:ros::Publisher pub = nh.advertise<std\_msgs::String>("/chatter/money",1000);

结果2:/chatter/money

**1.2相对名称**

格式:非/开头的名称, (话题参考的是节点的命名空间，与节点名称平级)

示例1:ros::Publisher pub = nh.advertise<std\_msgs::String>("chatter",1000);

结果1:xxx/chatter

示例2:ros::Publisher pub = nh.advertise<std\_msgs::String>("chatter/money",1000);

结果2:xxx/chatter/money

**1.3私有名称**

格式:以~开头的名称(话题参考节点名称，是节点名称的子级)

示例1:

ros::NodeHandle nh("~");

ros::Publisher pub = nh.advertise<std\_msgs::String>("chatter",1000);

结果1:/xxx/hello/chatter

示例2:

ros::NodeHandle nh("~");

ros::Publisher pub = nh.advertise<std\_msgs::String>("chatter/money",1000);

结果2:/xxx/hello/chatter/money

PS:当使用~,而话题名称有时/开头时，那么话题名称是绝对的

示例3:

ros::NodeHandle nh("~");

ros::Publisher pub = nh.advertise<std\_msgs::String>("/chatter/money",1000);

结果3:/chatter/money

**2.Python 实现**

准备:

1.初始化节点设置一个节点名称

rospy.init\_node("hello")

2.设置不同类型的话题

3.启动节点时，传递一个 \_\_ns:= xxx

4.节点启动后，使用 rostopic 查看话题信息

**2.1全局名称**

格式:以/开头的名称，和节点名称无关

示例1:pub = rospy.Publisher("/chatter",String,queue\_size=1000)

结果1:/chatter

示例2:pub = rospy.Publisher("/chatter/money",String,queue\_size=1000)

结果2:/chatter/money

**2.2相对名称**

格式:非/开头的名称,参考命名空间(与节点名称平级)来确定话题名称

示例1:pub = rospy.Publisher("chatter",String,queue\_size=1000)

结果1:xxx/chatter

示例2:pub = rospy.Publisher("chatter/money",String,queue\_size=1000)

结果2:xxx/chatter/money

**2.3私有名称**

格式:以~开头的名称

示例1:pub = rospy.Publisher("~chatter",String,queue\_size=1000)

结果1:/xxx/hello/chatter

示例2:pub = rospy.Publisher("~chatter/money",String,queue\_size=1000)

结果2:/xxx/hello/chatter/money

## ROS参数名称设置

关于参数重名的处理，没有重映射实现，为了尽量的避免参数重名，都是使用为参数名添加前缀的方式，实现类似于话题名称，有全局、相对、和私有三种类型之分。

案例：启动节点时，为参数服务器添加参数(需要注意参数名称设置)。

### 1.rosrun设置参数

rosrun 在启动节点时，也可以设置参数:

语法: rosrun 包名 节点名称 \_参数名:=参数值

1.设置参数

启动乌龟显示节点，并设置参数 A = 100

rosrun turtlesim turtlesim\_node \_A:=100

2.运行：rosparam list查看节点信息,显示结果

结果显示，参数A前缀节点名称，也就是说rosrun执行设置参数参数名使用的是私有模式

### 2.launch文件设置参数

通过 launch 文件设置参数的方式可以在 node 标签外，或 node 标签中通过 param 或 rosparam 来设置参数。在 node 标签外设置的参数是全局性质的，参考的是 / ，在 node 标签中设置的参数是私有性质的，参考的是 /命名空间/节点名称。

1.设置参数：以 param 标签为例，设置参数

<launch>

<param name="p1" value="100" />

<node pkg="turtlesim" type="turtlesim\_node" name="t1">

<param name="p2" value="100" />

</node>

</launch>

2.运行：rosparam list查看节点信息,显示结果

### 3.编码设置参数

**1.C++实现**

在 C++ 中，可以使用 ros::param 或者 ros::NodeHandle 来设置参数。

**1.1ros::param设置参数**

设置参数调用API是ros::param::set，该函数中，参数1传入参数名称，参数2是传入参数值，参数1中参数名称设置时，如果以 / 开头，那么就是全局参数，如果以 ~ 开头，那么就是私有参数，既不以 / 也不以 ~ 开头，那么就是相对参数。

ros::param::set("/set\_A",100); //全局,和命名空间以及节点名称无关

ros::param::set("set\_B",100); //相对,参考命名空间

ros::param::set("~set\_C",100); //私有,参考命名空间与节点名称

运行时，假设设置的 namespace 为 xxx，节点名称为 yyy，使用 rosparam list 查看

**1.2ros::NodeHandle设置参数**

设置参数时，首先需要创建 NodeHandle 对象，然后调用该对象的 setParam 函数。

ros::NodeHandle nh;

nh.setParam("/nh\_A",100); //全局,和命名空间以及节点名称无关

nh.setParam("nh\_B",100); //相对,参考命名空间

ros::NodeHandle nh\_private("~");

nh\_private.setParam("nh\_C",100);//私有,参考命名空间与节点名称

运行时，假设设置的 namespace 为 xxx，节点名称为 yyy，使用 rosparam list 查看

**2.python实现**

rospy.set\_param("/py\_A",100) #全局,和命名空间以及节点名称无关

rospy.set\_param("py\_B",100) #相对,参考命名空间

rospy.set\_param("~py\_C",100) #私有,参考命名空间与节点名称

## ROS分布式通信

ROS对网络配置有某些要求：

所有端口上的所有机器之间必须有完整的双向连接。

每台计算机必须通过所有其他计算机都可以解析的名称来公告自己。

**实现**

**1.准备**

先要保证不同计算机处于同一网络中，最好分别设置固定IP，如果为虚拟机，需要将网络适配器改为桥接模式；

**2.配置文件修改**

分别修改不同计算机的 /etc/hosts 文件，在该文件中加入对方的IP地址和计算机名:

主机端:

从机的IP 从机计算机名

从机端:

主机的IP 主机计算机名

设置完毕，可以通过 ping 命令测试网络通信是否正常。

IP地址查看名: ifconfig(若未安装先根据提示安装，windows下是ipconfig)

计算机名称查看: hostname

**3.配置主机IP**

配置主机的 IP 地址

~/.bashrc 追加

export ROS\_MASTER\_URI=http://主机IP:11311

export ROS\_HOSTNAME=主机IP

**4.配置从机IP**

配置从机的 IP 地址，从机可以有多台，每台都做如下设置:

~/.bashrc 追加

export ROS\_MASTER\_URI=http://主机IP:11311

export ROS\_HOSTNAME=从机IP

**测试**

1.主机启动 roscore(必须)

2.主机启动订阅节点，从机启动发布节点，测试通信是否正常

3.反向测试，主机启动发布节点，从机启动订阅节点，测试通信是否正常

# ROS常用组件

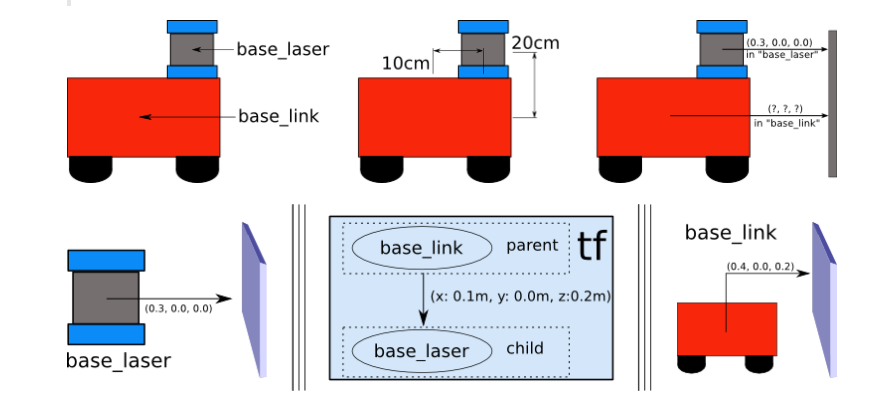
案例演示: 小乌龟跟随实现，该案例是ros中内置案例，终端下键入启动命令

roslaunch turtle\_tf2 turtle\_tf2\_demo\_cpp.launch或roslaunch turtle\_tf2 turtle\_tf2\_demo.launch

## TF坐标变换

场景1:雷达与小车

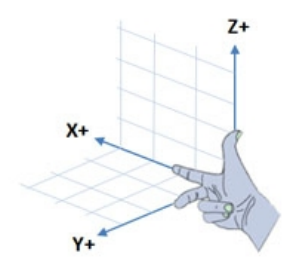
现有一移动式机器人底盘，在底盘上安装了一雷达，雷达相对于底盘的偏移量已知，现雷达检测到一障碍物信息，获取到坐标分别为(x,y,z)，该坐标是以雷达为参考系的，如何将这个坐标转换成以小车为参考系的坐标呢？



**概念**

tf:TransForm Frame,坐标变换

坐标系:ROS 中是通过坐标系统开标定物体的，确切的将是通过右手坐标系来标定的。



说明：在ROS中坐标变换最初对应的是tf，不过在 hydro 版本开始, tf 被弃用，迁移到 tf2,后者更为简洁高效，tf2对应的常用功能包有:

**tf2\_geometry\_msgs:可以将ROS消息转换成tf2消息。**

**tf2: 封装了坐标变换的常用消息。**

**tf2\_ros:为tf2提供了roscpp和rospy绑定，封装了坐标变换常用的API。**

### 1.坐标msg消息

订阅发布模型中数据载体 msg 是一个重要实现，首先需要了解一下，在坐标转换实现中常用的 msg:geometry\_msgs/TransformStamped和geometry\_msgs/PointStamped

前者用于传输坐标系相关位置信息，后者用于传输某个坐标系内坐标点的信息。

**1.geometry\_msgs/TransformStamped**

命令行键入:rosmsg info geometry\_msgs/TransformStamped

std\_msgs/Header header #头信息

uint32 seq #|-- 序列号

time stamp #|-- 时间戳

string frame\_id #|-- 坐标 ID

string child\_frame\_id #子坐标系的 id

geometry\_msgs/Transform transform #坐标信息

geometry\_msgs/Vector3 translation #偏移量

float64 x #|-- X 方向的偏移量

float64 y #|-- Y 方向的偏移量

float64 z #|-- Z 方向上的偏移量

geometry\_msgs/Quaternion rotation #四元数（用于表示坐标的相对姿态）

float64 x

float64 y

float64 z

float64 w

**2.geometry\_msgs/PointStamped**

命令行键入:rosmsg info geometry\_msgs/PointStamped

std\_msgs/Header header #头

uint32 seq #|-- 序号

time stamp #|-- 时间戳

string frame\_id #|-- 所属坐标系的 id

geometry\_msgs/Point point #点坐标

float64 x #|-- x y z 坐标

float64 y

float64 z

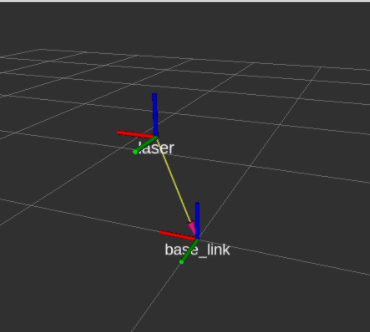
### 2.静态坐标变换

所谓静态坐标变换，是指两个坐标系之间的相对位置是固定的。

需求描述:现有一机器人模型，核心构成包含主体与雷达，各对应一坐标系，坐标系的原点分别位于主体与雷达的物理中心，已知雷达原点相对于主体原点位移关系如下: x 0.2 y0.0 z0.5。当前雷达检测到一障碍物，在雷达坐标系中障碍物的坐标为 (2.0 3.0 5.0),请问，该障碍物相对于主体的坐标是多少？

**结果演示:**





实现分析:

坐标系相对关系，可以通过发布方发布

订阅方，订阅到发布的坐标系相对关系，再传入坐标点信息(可以写死)，然后借助于 tf 实现坐标变换，并将结果输出

实现流程: **方案A:C++实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs

**2.发布方**

/\* 静态坐标变换发布方:发布关于 laser 坐标系的位置信息

实现流程:

1.包含头文件

2.初始化 ROS 节点

3.创建静态坐标转换广播器

4.创建坐标系信息

5.广播器发布坐标系信息

6.spin()

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "tf2\_ros/static\_transform\_broadcaster.h"

#include "geometry\_msgs/TransformStamped.h"

#include "tf2/LinearMath/Quaternion.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ROS 节点

ros::init(argc,argv,"static\_pub");

// 3.创建静态坐标转换广播器

tf2\_ros::StaticTransformBroadcaster broadcaster;

// 4.创建坐标系信息

geometry\_msgs::TransformStamped ts;

//----设置头信息

ts.header.seq = 100;

ts.header.stamp = ros::Time::now();

ts.header.frame\_id = "base\_link";

//----设置子级坐标系

ts.child\_frame\_id = "laser";

//----设置子级相对于父级的偏移量

ts.transform.translation.x = 0.2;

ts.transform.translation.y = 0.0;

ts.transform.translation.z = 0.5;

//----设置四元数:将 欧拉角数据转换成四元数

tf2::Quaternion qtn;

qtn.setRPY(0,0,0);

ts.transform.rotation.x = qtn.getX();

ts.transform.rotation.y = qtn.getY();

ts.transform.rotation.z = qtn.getZ();

ts.transform.rotation.w = qtn.getW();

// 5.广播器发布坐标系信息

broadcaster.sendTransform(ts);

ros::spin();

return 0;

}

**3.订阅方**

/\* 订阅坐标系信息，生成一个相对于子级坐标系的坐标点数据，转换成父级坐标系中的坐标点

实现流程:

1.包含头文件

2.初始化 ROS 节点

3.创建 TF 订阅节点

4.生成一个坐标点(相对于子级坐标系)

5.转换坐标点(相对于父级坐标系)

6.spin()

\*/

//1.包含头文件

#include "ros/ros.h"

#include "tf2\_ros/transform\_listener.h"

#include "tf2\_ros/buffer.h"

#include "geometry\_msgs/PointStamped.h"

#include "tf2\_geometry\_msgs/tf2\_geometry\_msgs.h" //注意: 调用 transform 必须包含该头文件

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ROS 节点

ros::init(argc,argv,"tf\_sub");

ros::NodeHandle nh;

// 3.创建 TF 订阅节点

**tf2\_ros::Buffer buffer;**

**tf2\_ros::TransformListener listener(buffer);**

ros::Rate r(1);

while (ros::ok())

{

// 4.生成一个坐标点(相对于子级坐标系)

geometry\_msgs::PointStamped point\_laser;

point\_laser.header.frame\_id = "laser";

point\_laser.header.stamp = ros::Time::now();

point\_laser.point.x = 1;

point\_laser.point.y = 2;

point\_laser.point.z = 7.3;

// 5.转换坐标点(相对于父级坐标系)

//新建一个坐标点，用于接收转换结果

//使用 try 语句或休眠，否则可能由于缓存接收延迟而导致坐标转换失败

try

{

geometry\_msgs::PointStamped point\_base;

point\_base = buffer.transform(point\_laser,"base\_link");

ROS\_INFO("转换后的数据:(%.2f,%.2f,%.2f),参考的坐标系是:%s",point\_base.point.x,point\_base.point.y,point\_base.point.z,point\_base.header.frame\_id.c\_str());

}

catch(const std::exception& e)

{

// std::cerr << e.what() << '\n';

ROS\_INFO("程序异常.....");

}

r.sleep();

ros::spinOnce();

}

return 0;

}

配置文件此处略。

**4.执行**

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，控制台将输出，坐标转换后的结果。

**方案B:Python实现**

**1.创建功能包**

**2.发布方**

#! /usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

import tf

from geometry\_msgs.msg import TransformStamped

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("static\_pub\_p")

# 3.创建 静态坐标广播器

broadcaster = tf2\_ros.StaticTransformBroadcaster()

# 4.创建并组织被广播的消息

tfs = TransformStamped()

# --- 头信息

tfs.header.frame\_id = "world"

tfs.header.stamp = rospy.Time.now()

tfs.header.seq = 101

# --- 子坐标系

tfs.child\_frame\_id = "radar"

# --- 坐标系相对信息

# ------ 偏移量

tfs.transform.translation.x = 0.2

tfs.transform.translation.y = 0.0

tfs.transform.translation.z = 0.5

# ------ 四元数

qtn = tf.transformations.quaternion\_from\_euler(0,0,0)

tfs.transform.rotation.x = qtn[0]

tfs.transform.rotation.y = qtn[1]

tfs.transform.rotation.z = qtn[2]

tfs.transform.rotation.w = qtn[3]

# 5.广播器发送消息

broadcaster.sendTransform(tfs)

# 6.spin

rospy.spin()

**3.订阅方**

#! /usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

# 不要使用 geometry\_msgs,需要使用 tf2 内置的消息类型

from tf2\_geometry\_msgs import PointStamped

# from geometry\_msgs.msg import PointStamped

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("static\_sub\_p")

# 3.创建 TF 订阅对象

buffer = tf2\_ros.Buffer()

listener = tf2\_ros.TransformListener(buffer)

rate = rospy.Rate(1)

while not rospy.is\_shutdown():

# 4.创建一个 radar 坐标系中的坐标点

point\_source = PointStamped()

point\_source.header.frame\_id = "radar"

point\_source.header.stamp = rospy.Time.now()

point\_source.point.x = 10

point\_source.point.y = 2

point\_source.point.z = 3

try:

# 5.调研订阅对象的 API 将 4 中的点坐标转换成相对于 world 的坐标

point\_target = buffer.transform(point\_source,"world")

rospy.loginfo("转换结果:x = %.2f, y = %.2f, z = %.2f",

point\_target.point.x,

point\_target.point.y,

point\_target.point.z)

except Exception as e:

rospy.logerr("异常:%s",e)

# 6.spin

rate.sleep()

权限设置以及配置文件此处略。

PS: 在 tf2 的 python 实现中，tf2 已经封装了一些消息类型，不可以使用 geometry\_msgs.msg 中的类型

补充1: 当坐标系之间的相对位置固定时，那么所需参数也是固定的: 父系坐标名称、子级坐标系名称、x偏移量、y偏移量、z偏移量、x 翻滚角度、y俯仰角度、z偏航角度，实现逻辑相同，参数不同，那么 ROS 系统就已经封装好了专门的节点，使用方式如下:

rosrun tf2\_ros static\_transform\_publisher x偏移量 y偏移量 z偏移量 z偏航角度 y俯仰角度 x翻滚角度 父级坐标系 子级坐标系

示例:rosrun tf2\_ros static\_transform\_publisher 0.2 0 0.5 0 0 0 /baselink /laser

也建议使用该种方式直接实现静态坐标系相对信息发布。

补充2:可以借助于rviz显示坐标系关系，具体操作:

新建窗口输入命令:rviz;

在启动的 rviz 中设置Fixed Frame 为 base\_link;

点击左下的 add 按钮，在弹出的窗口中选择 TF 组件，即可显示坐标关系。

### 3.动态坐标变换

所谓动态坐标变换，是指两个坐标系之间的相对位置是变化的。

需求描述:启动 turtlesim\_node,该节点中窗体有一个世界坐标系(左下角为坐标系原点)，乌龟是另一个坐标系，键盘控制乌龟运动，将两个坐标系的相对位置动态发布。

实现流程: **方案A:C++实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs、turtlesim

**2.发布方**

/\* 动态的坐标系相对姿态发布(一个坐标系相对于另一个坐标系的相对姿态是不断变动的)

需求: 启动 turtlesim\_node,该节点中窗体有一个世界坐标系(左下角为坐标系原点)，乌龟是另一个坐标系，键盘

控制乌龟运动，将两个坐标系的相对位置动态发布

实现分析: 1.乌龟本身不但可以看作坐标系，也是世界坐标系中的一个坐标点

2.订阅 turtle1/pose,可以获取乌龟在世界坐标系的 x坐标、y坐标、偏移量以及线速度和角速度

3.将 pose 信息转换成 坐标系相对信息并发布

实现流程:

1.包含头文件

2.初始化 ROS 节点

3.创建 ROS 句柄

4.创建订阅对象

5.回调函数处理订阅到的数据(实现TF广播)

5-1.创建 TF 广播器

5-2.创建 广播的数据(通过 pose 设置)

5-3.广播器发布数据

6.spin

\*/

// 1.包含头文件

#include "ros/ros.h"

#include "turtlesim/Pose.h"

#include "tf2\_ros/transform\_broadcaster.h"

#include "geometry\_msgs/TransformStamped.h"

#include "tf2/LinearMath/Quaternion.h"

void doPose(const turtlesim::Pose::ConstPtr& pose){

// 5-1.创建 TF 广播器

static tf2\_ros::TransformBroadcaster broadcaster;

// 5-2.创建广播的数据(通过 pose 设置)

geometry\_msgs::TransformStamped tfs;

// |----头设置

tfs.header.frame\_id = "world";

tfs.header.stamp = ros::Time::now();

// |----子坐标系 ID

tfs.child\_frame\_id = "turtle1";

// |----坐标系相对信息设置

tfs.transform.translation.x = pose->x;

tfs.transform.translation.y = pose->y;

tfs.transform.translation.z = 0.0; // 二维实现，pose 中没有z，z 是 0

// |--------- 四元数设置

tf2::Quaternion qtn;

qtn.setRPY(0,0,pose->theta);

tfs.transform.rotation.x = qtn.getX();

tfs.transform.rotation.y = qtn.getY();

tfs.transform.rotation.z = qtn.getZ();

tfs.transform.rotation.w = qtn.getW();

// 5-3.广播器发布数据

broadcaster.sendTransform(tfs);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ROS 节点

ros::init(argc,argv,"dynamic\_pub");

// 3.创建 ROS 句柄

ros::NodeHandle nh;

// 4.创建订阅对象

ros::Subscriber sub = nh.subscribe<turtlesim::Pose>("/turtle1/pose",1000,doPose);

// 5.回调函数处理订阅到的数据(实现TF广播)

// 6.spin

ros::spin();

return 0;

}

**3.订阅方**

//1.包含头文件

#include "ros/ros.h"

#include "tf2\_ros/transform\_listener.h"

#include "tf2\_ros/buffer.h"//可以不用

#include "geometry\_msgs/PointStamped.h"

#include "tf2\_geometry\_msgs/tf2\_geometry\_msgs.h" //注意: 调用 transform 必须包含该头文件

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ROS 节点

ros::init(argc,argv,"dynamic\_tf\_sub");

ros::NodeHandle nh;

// 3.创建 TF 订阅节点

tf2\_ros::Buffer buffer;

tf2\_ros::TransformListener listener(buffer);

ros::Rate r(1);

while (ros::ok())

{

// 4.生成一个坐标点(相对于子级坐标系)

geometry\_msgs::PointStamped point\_laser;

point\_laser.header.frame\_id = "turtle1";

point\_laser.header.stamp = ros::Time();//或ros::Time(0.0)表示此时无时间戳（时间戳无值），不能用ros::Time::now()，buffer里有一堆值每个都有时间戳，ros会将坐标点的时间戳与坐标转换的时间戳比较，时间接近才能转换

point\_laser.point.x = 1;

point\_laser.point.y = 1;

point\_laser.point.z = 0;

// 5.转换坐标点(相对于父级坐标系)

//新建一个坐标点，用于接收转换结果

//使用 try 语句或休眠，否则可能由于缓存接收延迟而导致坐标转换失败

try

{

geometry\_msgs::PointStamped point\_base;

point\_base = buffer.transform(point\_laser,"world");

ROS\_INFO("坐标点相对于 world 的坐标为:(%.2f,%.2f,%.2f)",point\_base.point.x,point\_base.point.y,point\_base.point.z);

}

catch(const std::exception& e)

{

// std::cerr << e.what() << '\n';

ROS\_INFO("程序异常:%s",e.what());

}

r.sleep();

ros::spinOnce();

}

return 0;

}

配置文件此处略。

**4.执行**

可以使用命令行或launch文件的方式分别启动发布节点与订阅节点，如果程序无异常，与演示结果类似。

可以使用 rviz 查看坐标系相对关系。

**方案B:Python实现**

**1.创建功能包**

**2.发布方**

#! /usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

import tf

from turtlesim.msg import Pose

from geometry\_msgs.msg import TransformStamped

# 4.回调函数处理

def doPose(pose):

# 4-1.创建 TF 广播器

broadcaster = tf2\_ros.TransformBroadcaster()

# 4-2.创建 广播的数据(通过 pose 设置)

tfs = TransformStamped()

tfs.header.frame\_id = "world"

tfs.header.stamp = rospy.Time.now()

tfs.child\_frame\_id = "turtle1"

tfs.transform.translation.x = pose.x

tfs.transform.translation.y = pose.y

tfs.transform.translation.z = 0.0

qtn = tf.transformations.quaternion\_from\_euler(0,0,pose.theta)

tfs.transform.rotation.x = qtn[0]

tfs.transform.rotation.y = qtn[1]

tfs.transform.rotation.z = qtn[2]

tfs.transform.rotation.w = qtn[3]

# 4-3.广播器发布数据

broadcaster.sendTransform(tfs)

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("dynamic\_tf\_pub\_p")

# 3.订阅 /turtle1/pose 话题消息

sub = rospy.Subscriber("/turtle1/pose",Pose,doPose)

# 4.回调函数处理

# 4-1.创建 TF 广播器

# 4-2.创建 广播的数据(通过 pose 设置)

# 4-3.广播器发布数据

# 5.spin

rospy.spin()

**3.订阅方**

#! /usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

# 不要使用 geometry\_msgs,需要使用 tf2 内置的消息类型

from tf2\_geometry\_msgs import PointStamped

# from geometry\_msgs.msg import PointStamped

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("static\_sub\_tf\_p")

# 3.创建 TF 订阅对象

buffer = tf2\_ros.Buffer()

listener = tf2\_ros.TransformListener(buffer)

rate = rospy.Rate(1)

while not rospy.is\_shutdown():

# 4.创建一个 radar 坐标系中的坐标点

point\_source = PointStamped()

point\_source.header.frame\_id = "turtle1"

point\_source.header.stamp = rospy.Time.now()#py里Time.now()可以，cpp里不可以

point\_source.point.x = 10

point\_source.point.y = 2

point\_source.point.z = 3

try:

# 5.调研订阅对象的 API 将 4 中的点坐标转换成相对于 world 的坐标

point\_target = buffer.transform(point\_source,"world",rospy.Duration(1))

rospy.loginfo("转换结果:x = %.2f, y = %.2f, z = %.2f",

point\_target.point.x,

point\_target.point.y,

point\_target.point.z)

except Exception as e:

rospy.logerr("异常:%s",e)

# 6.spin

rate.sleep()

权限设置以及配置文件此处略。

### 4.多坐标变换

需求：描述:现有坐标系统，父级坐标系统 world,下有两子级系统 son1，son2，son1 相对于 world，以及 son2 相对于 world 的关系是已知的，求 son1原点在 son2中的坐标，又已知在 son1中一点的坐标，要求求出该点在 son2 中的坐标

实现分析:

1首先，需要发布 son1 相对于 world，以及 son2 相对于 world 的坐标消息

2然后，需要订阅坐标发布消息，并取出订阅的消息，借助于 tf2 实现 son1 和 son2 的转换

3最后，还要实现坐标点的转换

实现流程: **方案A:C++实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs、turtlesim

**2.发布方**

为了方便，使用静态坐标变换发布:

<launch>

<node pkg="tf2\_ros" type="static\_transform\_publisher" name="son1" args="0.2 0.8 0.3 0 0 0 /world /son1" output="screen" />

<node pkg="tf2\_ros" type="static\_transform\_publisher" name="son2" args="0.5 0 0 0 0 0 /world /son2" output="screen" />

</launch>

**3.订阅方**

/\*

实现流程:

1.包含头文件

2.初始化 ros 节点

3.创建 ros 句柄

4.创建 TF 订阅对象

5.解析订阅信息中获取 son1 坐标系原点在 son2 中的坐标

解析 son1 中的点相对于 son2 的坐标

6.spin

\*/

//1.包含头文件

#include "ros/ros.h"

#include "tf2\_ros/transform\_listener.h"

#include "tf2\_geometry\_msgs/tf2\_geometry\_msgs.h"

#include "geometry\_msgs/TransformStamped.h"

#include "geometry\_msgs/PointStamped.h"

int main(int argc, char \*argv[])

{ setlocale(LC\_ALL,"");

// 2.初始化 ros 节点

ros::init(argc,argv,"many");

// 3.创建 ros 句柄

ros::NodeHandle nh;

// 4.创建 TF 订阅对象

tf2\_ros::Buffer buffer; **//可以不用buffer头文件**

tf2\_ros::TransformListener listener(buffer);

// 5.解析订阅信息中获取 son1 坐标系原点在 son2 中的坐标

ros::Rate r(1);

while (ros::ok())

{

try

{

// 解析 son1 中的点相对于 son2 的坐标

geometry\_msgs::TransformStamped tfs = buffer.lookupTransform("son2","son1",ros::Time(0));//可以获取两个坐标系之间的转换关系

ROS\_INFO("Son1 相对于 Son2 的坐标关系:父坐标系ID=%s",tfs.header.frame\_id.c\_str());

ROS\_INFO("Son1 相对于 Son2 的坐标关系:子坐标系ID=%s",tfs.child\_frame\_id.c\_str());

ROS\_INFO("Son1 相对于 Son2 的坐标关系:x=%.2f,y=%.2f,z=%.2f",

tfs.transform.translation.x,

tfs.transform.translation.y,

tfs.transform.translation.z

);

// 坐标点解析

geometry\_msgs::PointStamped ps;

ps.header.frame\_id = "son1";

ps.header.stamp = ros::Time::now();

ps.point.x = 1.0;

ps.point.y = 2.0;

ps.point.z = 3.0;

geometry\_msgs::PointStamped psAtSon2;

psAtSon2 = buffer.transform(ps,"son2");

ROS\_INFO("在 Son2 中的坐标:x=%.2f,y=%.2f,z=%.2f",

psAtSon2.point.x,

psAtSon2.point.y,

psAtSon2.point.z

);

}

catch(const std::exception& e)

{

// std::cerr << e.what() << '\n';

ROS\_INFO("异常信息:%s",e.what());

}

r.sleep();

// 6.spin

ros::spinOnce();

}

return 0;

}

配置文件此处略。

**方案B:Python实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs、turtlesim

**2.发布方**

同上

**3.订阅方**

#!/usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

from geometry\_msgs.msg import TransformStamped

from tf2\_geometry\_msgs import PointStamped

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("frames\_sub\_p")

# 3.创建 TF 订阅对象

buffer = tf2\_ros.Buffer()

listener = tf2\_ros.TransformListener(buffer)

rate = rospy.Rate(1)

while not rospy.is\_shutdown():

try:

# 4.调用 API 求出 son1 相对于 son2 的坐标关系

#lookup\_transform(self, target\_frame, source\_frame, time, timeout=rospy.Duration(0.0)):

tfs = buffer.lookup\_transform("son2","son1",rospy.Time(0))

rospy.loginfo("son1 与 son2 相对关系:")

rospy.loginfo("父级坐标系:%s",tfs.header.frame\_id)

rospy.loginfo("子级坐标系:%s",tfs.child\_frame\_id)

rospy.loginfo("相对坐标:x=%.2f, y=%.2f, z=%.2f",

tfs.transform.translation.x,

tfs.transform.translation.y,

tfs.transform.translation.z,

)

# 5.创建一依赖于 son1 的坐标点，调用 API 求出该点在 son2 中的坐标

point\_source = PointStamped()

point\_source.header.frame\_id = "son1"

point\_source.header.stamp = rospy.Time.now()

point\_source.point.x = 1

point\_source.point.y = 1

point\_source.point.z = 1

point\_target = buffer.transform(point\_source,"son2",rospy.Duration(0.5))

rospy.loginfo("point\_target 所属的坐标系:%s",point\_target.header.frame\_id)

rospy.loginfo("坐标点相对于 son2 的坐标:(%.2f,%.2f,%.2f)",

point\_target.point.x,

point\_target.point.y,

point\_target.point.z

)

except Exception as e:

rospy.logerr("错误提示:%s",e)

rate.sleep()

# 6.spin

# rospy.spin()

权限设置以及配置文件此处略。

### 5.坐标系关系查看

在机器人系统中，涉及的坐标系有多个，为了方便查看，ros 提供了专门的工具，可以用于生成显示坐标系关系的 pdf 文件，该文件包含树形结构的坐标系图谱。

**1准备**

首先调用rospack find tf2\_tools查看是否包含该功能包，如果没有，请使用如下命令安装:

sudo apt install ros-melodic-tf2-tools

**2使用**

**2.1生成 pdf 文件**

启动坐标系广播程序之后，运行如下命令:

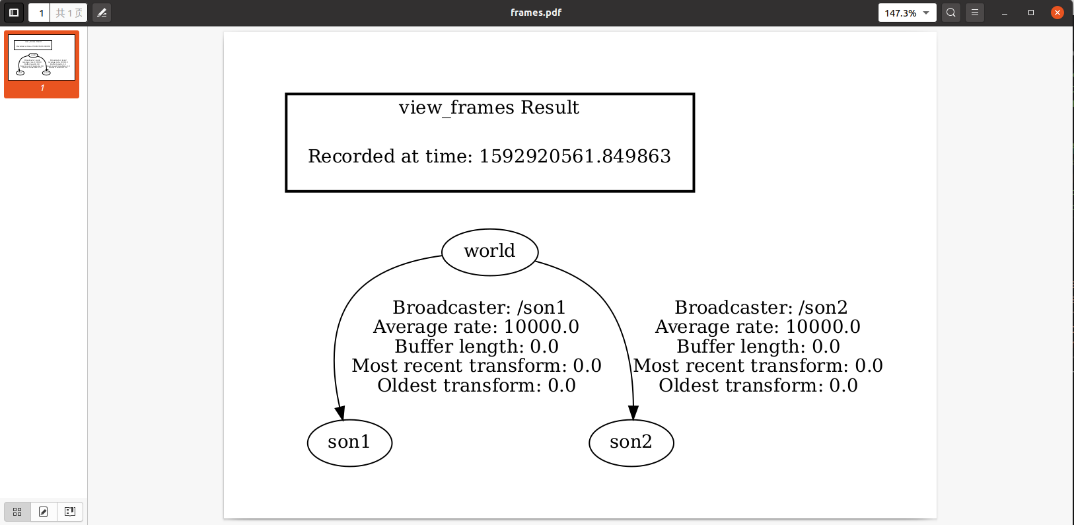
rosrun tf2\_tools view\_frames.py

查看当前目录会生成一个 frames.pdf 文件

或者rosrun echo 坐标系1（根） 坐标系2（子）

**2.2查看文件**

可以直接进入目录打开文件，或者调用命令查看文件:evince frames.pdf



### 6.TF坐标变换实操

需求描述:程序启动之初: 产生两只乌龟，中间的乌龟(A) 和 左下乌龟(B), B 会自动运行至A的位置，并且键盘控制时，只是控制 A 的运动，但是 B 可以跟随 A 运行

实现分析:

乌龟跟随实现的核心，是乌龟A和B都要发布相对世界坐标系的坐标信息，然后，订阅到该信息需要转换获取A相对于B坐标系的信息，最后，再生成速度信息，并控制B运动。

1启动乌龟显示节点

2在乌龟显示窗体中生成一只新的乌龟(需要使用服务)

3编写两只乌龟发布坐标信息的节点

4编写订阅节点订阅坐标信息并生成新的相对关系生成速度信息

实现流程: **准备工作**

1.了解如何创建第二只乌龟，且不受键盘控制

创建第二只乌龟需要使用rosservice,话题使用的是 spawn

rosservice call /spawn "x: 1.0 y: 1.0 theta: 1.0 name: 'turtle2'"

键盘是无法控制第二只乌龟运动的，因为使用的话题: /第二只乌龟名称/cmd\_vel,对应的要控制乌龟运动必须发布对应的话题消息

2.了解如何获取两只乌龟的坐标

是通过话题 /乌龟名称/pose 来获取的

x: 1.0 //x坐标

y: 1.0 //y坐标

theta: -1.21437060833 //角度

linear\_velocity: 0.0 //线速度

angular\_velocity: 1.0 //角速度

**方案A:C++实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs、turtlesim

**2.服务客户端(生成乌龟)**

/\* 创建第二只小乌龟\*/

#include "ros/ros.h"

#include "turtlesim/Spawn.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

//执行初始化

ros::init(argc,argv,"set\_turtle”);

//创建节点

ros::NodeHandle nh;

//创建服务客户端

ros::ServiceClient client = nh.serviceClient<turtlesim::Spawn>(“/spawn”);

//client.waitForExistence();

ros::service::waitForService("/spawn");

turtlesim::Spawn spawn;

spawn.request.name = "turtle2";

spawn.request.x = 1.0;

spawn.request.y = 2.0;

spawn.request.theta = 3.12415926;

bool flag = client.call(spawn);

if (flag)

{

ROS\_INFO("乌龟%s创建成功!",spawn.response.name.c\_str());

}

else

{

ROS\_INFO("乌龟2创建失败!");

}

ros::spin();

return 0;

}

**3.发布方(发布两只乌龟的坐标信息)**

可以订阅乌龟的位姿信息，然后再转换成坐标信息，两只乌龟的实现逻辑相同，只是订阅的话题名称，生成的坐标信息等稍有差异，可以将差异部分通过参数传入:

该节点需要启动两次

每次启动时都需要传入乌龟节点名称(第一次是 turtle1 第二次是 turtle2)

/\* 该文件实现:需要订阅 turtle1 和 turtle2 的 pose，然后广播相对 world 的坐标系信息

注意: 订阅的两只 turtle,除了命名空间(turtle1 和 turtle2)不同外,

其他的话题名称和实现逻辑都是一样的，

所以我们可以将所需的命名空间通过 args 动态传入

实现流程:

1.包含头文件

2.初始化 ros 节点

3.解析传入的命名空间

4.创建 ros 句柄

5.创建订阅对象

6.回调函数处理订阅的 pose 信息

6-1.创建 TF 广播器

6-2.将 pose 信息转换成 TransFormStamped

6-3.发布

7.spin

\*/

//1.包含头文件

#include "ros/ros.h"

#include "turtlesim/Pose.h"

#include "tf2\_ros/transform\_broadcaster.h"

#include "tf2/LinearMath/Quaternion.h"

#include "geometry\_msgs/TransformStamped.h"

//保存乌龟名称

std::string turtle\_name;

void doPose(const turtlesim::Pose::ConstPtr& pose){

// 6-1.创建 TF 广播器 ---------------------------------------- 注意 static

static tf2\_ros::TransformBroadcaster broadcaster;//static使广播器只初始化一次

// 6-2.将 pose 信息转换成 TransFormStamped

geometry\_msgs::TransformStamped tfs;

tfs.header.frame\_id = "world";

tfs.header.stamp = ros::Time::now();

tfs.child\_frame\_id = turtle\_name;

tfs.transform.translation.x = pose->x;

tfs.transform.translation.y = pose->y;

tfs.transform.translation.z = 0.0;

tf2::Quaternion qtn;

qtn.setRPY(0,0,pose->theta);

tfs.transform.rotation.x = qtn.getX();

tfs.transform.rotation.y = qtn.getY();

tfs.transform.rotation.z = qtn.getZ();

tfs.transform.rotation.w = qtn.getW();

// 6-3.发布

broadcaster.sendTransform(tfs);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ros 节点

ros::init(argc,argv,"turtle\_frames\_pub");

// 3.解析传入的命名空间

if (argc != 2)

{

ROS\_ERROR("请传入正确的参数");

} else {

turtle\_name = argv[1];

ROS\_INFO("乌龟 %s 坐标发送启动",turtle\_name.c\_str());

}

// 4.创建 ros 句柄

ros::NodeHandle nh;

// 5.创建订阅对象

ros::Subscriber sub = nh.subscribe<turtlesim::Pose>(turtle\_name + "/pose",1000,doPose);

// 6.回调函数处理订阅的 pose 信息

// 7.spin

ros::spin();

return 0;

}

**4.订阅方(解析坐标信息并生成速度信息)**

/\*

订阅 turtle1 和 turtle2 的 TF 广播信息，查找并转换时间最近的 TF 信息

将 turtle1 转换成相对 turtle2 的坐标，在计算线速度和角速度并发布\*/

//1.包含头文件

#include "ros/ros.h"

#include "tf2\_ros/transform\_listener.h"

#include "geometry\_msgs/TransformStamped.h"

#include "geometry\_msgs/Twist.h"

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ros 节点

ros::init(argc,argv,"turtlee\_frames\_sub");

// 3.创建 ros 句柄

ros::NodeHandle nh;

// 4.创建 TF 订阅对象

tf2\_ros::Buffer buffer;

tf2\_ros::TransformListener listener(buffer);

// 5.处理订阅到的 TF

// 需要创建发布 /turtle2/cmd\_vel 的 publisher 对象

ros::Publisher pub = nh.advertise<geometry\_msgs::Twist>("/turtle2/cmd\_vel",1000);

ros::Rate rate(10);

while (ros::ok())

{

try

{

//5-1.先获取 turtle1 相对 turtle2 的坐标信息

geometry\_msgs::TransformStamped tfs = buffer.lookupTransform("turtle2","turtle1",ros::Time(0));

//5-2.根据坐标信息生成速度信息 -- geometry\_msgs/Twist.h

geometry\_msgs::Twist twist;

twist.linear.x = 0.5 \* sqrt(pow(tfs.transform.translation.x,2) + pow(tfs.transform.translation.y,2));//0.5和4是系数随便设的

twist.angular.z = 4 \* atan2(tfs.transform.translation.y,tfs.transform.translation.x);

//5-3.发布速度信息 -- 需要提前创建 publish 对象

pub.publish(twist);

}

catch(const std::exception& e)

{

// std::cerr << e.what() << '\n';

ROS\_INFO("错误提示:%s",e.what());

}

rate.sleep();

// 6.spin

ros::spinOnce();

}

return 0;

}

配置文件此处略。

**5.运行**

使用 launch 文件组织需要运行的节点，内容如下:

<launch>

<!-- 启动乌龟节点与键盘控制节点 -->

<node pkg="turtlesim" type="turtlesim\_node" name="turtle1" output="screen" />

<node pkg="turtlesim" type="turtle\_teleop\_key" name="key\_control" output="screen"/>

<!-- 启动创建第二只乌龟的节点 -->

<node pkg="demo\_tf" type="set\_turtle" name="turtle2" output="screen" />

<!-- 启动两个坐标发布节点 -->

<node pkg="demo\_tf" type="turtle \_pub" name="t1" output="screen" args="turtle1" />

<node pkg="demo\_tf" type=" turtle \_pub" name="t2" output="screen" args="turtle2" />

<!-- 启动坐标转换节点 -->

<node pkg="demo\_tf " type="turtle\_sub" name="listener" output="screen" />

</launch>

**方案B:Python实现**

**1.创建功能包**

创建项目功能包依赖于 tf2、tf2\_ros、tf2\_geometry\_msgs、roscpp rospy std\_msgs geometry\_msgs、turtlesim

**2.服务客户端(生成乌龟)**

#! /usr/bin/env python

#1.导包

import rospy

from turtlesim.srv import Spawn, SpawnRequest, SpawnResponse

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ros 节点

rospy.init\_node("turtle\_spawn\_p")

# 3.创建服务客户端

client = rospy.ServiceProxy("/spawn",Spawn)

# 4.等待服务启动

client.wait\_for\_service()

# 5.创建请求数据

req = SpawnRequest()

req.x = 1.0

req.y = 1.0

req.theta = 3.14

req.name = "turtle2"

# 6.发送请求并处理响应

try:

response = client.call(req)

rospy.loginfo("乌龟创建成功，名字是:%s",response.name)

except Exception as e:

rospy.loginfo("服务调用失败....")

**3.发布方(发布两只乌龟的坐标信息)**

#! /usr/bin/env python

# 1.导包

import rospy

import sys

from turtlesim.msg import Pose

from geometry\_msgs.msg import TransformStamped

import tf2\_ros

import tf\_conversions

turtle\_name = ""

def doPose(pose):

# rospy.loginfo("x = %.2f",pose.x)

#1.创建坐标系广播器

broadcaster = tf2\_ros.TransformBroadcaster()

#2.将 pose 信息转换成 TransFormStamped

tfs = TransformStamped()

tfs.header.frame\_id = "world"

tfs.header.stamp = rospy.Time.now()

tfs.child\_frame\_id = turtle\_name

tfs.transform.translation.x = pose.x

tfs.transform.translation.y = pose.y

tfs.transform.translation.z = 0.0

qtn = tf\_conversions.transformations.quaternion\_from\_euler(0, 0, pose.theta)

tfs.transform.rotation.x = qtn[0]

tfs.transform.rotation.y = qtn[1]

tfs.transform.rotation.z = qtn[2]

tfs.transform.rotation.w = qtn[3]

#3.广播器发布 tfs

broadcaster.sendTransform(tfs)

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ros 节点

rospy.init\_node("sub\_tfs\_p")

# 3.解析传入的命名空间

rospy.loginfo("-------------------------------%d",len(sys.argv))

if len(sys.argv) < 2:

rospy.loginfo("请传入参数:乌龟的命名空间")

else:

turtle\_name = sys.argv[1]

rospy.loginfo("///////////////////乌龟:%s",turtle\_name)

rospy.Subscriber(turtle\_name + "/pose",Pose,doPose)

# 4.创建订阅对象

# 5.回调函数处理订阅的 pose 信息

# 5-1.创建 TF 广播器

# 5-2.将 pose 信息转换成 TransFormStamped

# 5-3.发布

# 6.spin

rospy.spin()

**4.订阅方(解析坐标信息并生成速度信息)**

#! /usr/bin/env python

# 1.导包

import rospy

import tf2\_ros

from geometry\_msgs.msg import TransformStamped, Twist

import math

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ros 节点

rospy.init\_node("sub\_tfs\_p")

# 3.创建 TF 订阅对象

buffer = tf2\_ros.Buffer()

listener = tf2\_ros.TransformListener(buffer)

# 4.处理订阅到的 TF

rate = rospy.Rate(10)

# 创建速度发布对象

pub = rospy.Publisher("/turtle2/cmd\_vel",Twist,queue\_size=1000)

while not rospy.is\_shutdown():

rate.sleep()

try:

#def lookup\_transform(self, target\_frame, source\_frame, time, timeout=rospy.Duration(0.0)):

trans = buffer.lookup\_transform("turtle2","turtle1",rospy.Time(0))

# rospy.loginfo("相对坐标:(%.2f,%.2f,%.2f)",

# trans.transform.translation.x,

# trans.transform.translation.y,

# trans.transform.translation.z

# )

# 根据转变后的坐标计算出速度和角速度信息

twist = Twist()

# 间距 = x^2 + y^2 然后开方

twist.linear.x = 0.5 \* math.sqrt(math.pow(trans.transform.translation.x,2) + math.pow(trans.transform.translation.y,2))

twist.angular.z = 4 \* math.atan2(trans.transform.translation.y, trans.transform.translation.x)

pub.publish(twist)

except Exception as e:

rospy.logwarn("警告:%s",e)

权限设置以及配置文件此处略。

**5.运行**

使用 launch 文件组织需要运行的节点，内容如下:

<launch>

<node pkg="turtlesim" type="turtlesim\_node" name="turtle1" output="screen" />

<node pkg="turtlesim" type="turtle\_teleop\_key" name="key\_control" output="screen"/>

<node pkg="demo\_tf" type="set\_turtle.py" name="turtle\_spawn" output="screen"/>

<node pkg="demo\_tf" type="turtle\_pub.py" name="t1" args="turtle1" output="screen"/>

<node pkg="demo\_tf" type="turtle\_pub.py" name="t2" args="turtle2" output="screen"/>

<node pkg="demo\_tf" type="turtle\_sub.py" name="tf\_sub" output="screen"/>

</launch>

### 7.TF2与TF

**1.TF2与TF比较\_简介**

TF2已经替换了TF，TF2是TF的超集，建议学习 TF2 而非 TF

TF2 功能包的增强了内聚性，TF 与 TF2 所依赖的功能包是不同的，TF 对应的是tf包，TF2 对应的是tf2和tf2\_ros包，在 TF2 中不同类型的 API 实现做了分包处理。

TF2 实现效率更高，比如在:TF2 的静态坐标实现、TF2 坐标变换监听器中的 Buffer 实现等

**2.TF2与TF比较\_静态坐标变换演示**

接下来，我们通过静态坐标变换来演示TF2的实现效率。

**2.1启动 TF2 与 TF 两个版本的静态坐标变换**

TF2 版静态坐标变换:rosrun tf2\_ros static\_transform\_publisher 0 0 0 0 0 0 /base\_link /laser

TF 版静态坐标变换:rosrun tf static\_transform\_publisher 0 0 0 0 0 0 /base\_link /laser 100

会发现，TF 版本的启动中最后多一个参数，该参数是指定发布频率

**2.2运行结果比对**

使用rostopic查看话题，包含/tf与/tf\_static, 前者是 TF 发布的话题，后者是 TF2 发布的话题，分别调用命令打印二者的话题消息

rostopic echo /tf: 当前会循环输出坐标系信息

rostopic echo /tf\_static: 坐标系信息只有一次

**2.3结论**

如果是静态坐标转换，那么不同坐标系之间的相对状态是固定的，既然是固定的，那么没有必要重复发布坐标系的转换消息，很显然的，tf2 实现较之于 tf 更为高效

## Rosbag

机器人传感器获取到的信息，有时我们可能需要时时处理，有时可能只是采集数据，事后分析，比如:

机器人导航实现中，可能需要绘制导航所需的全局地图，地图绘制实现，有两种方式，方式1：可以控制机器人运动，将机器人传感器感知到的数据时时处理，生成地图信息。方式2：同样是控制机器人运动，将机器人传感器感知到的数据留存，事后，再重新读取数据，生成地图信息。两种方式比较，显然方式2使用上更为灵活方便。

在ROS中关于数据的留存以及读取实现，提供了专门的工具: rosbag。

概念:用于录制和回放 ROS 主题的一个工具集。

作用:实现了数据的复用，方便调试、测试。

本质

rosbag本质也是ros的节点，当录制时，rosbag是一个订阅节点，可以订阅话题消息并将订阅到的数据写入磁盘文件；当重放时，rosbag是一个发布节点，可以读取磁盘文件，发布文件中的话题消息。

### 1.rosbag使用\_命令行

需求:ROS 内置的乌龟案例并操作，操作过程中使用 rosbag 录制，录制结束后，实现重放

实现:

**1.准备**

创建目录保存录制的文件

mkdir ./xxx

cd xxx

**2.开始录制**

rosbag record -a -O 目标文件

-a:所有数据，-O：保存成目标文件

操作小乌龟一段时间，结束录制使用 ctrl + c，在创建的目录中会生成bag文件。

rosbag record buffer exceeded. Dropping oldest queued message.解决办法：

提高rosbag的缓存空间：rosbag record -o /home/inin/data/ -b 4096 /occam/stitched\_image0 /occam/image\_tiles0

指令如上所示，rosbag 中加入-b num ，即为将缓存空间设置成num大小

**3.查看文件**

rosbag info 文件名

**4.回放文件**

rosbag play 文件名

重启乌龟节点，会发现，乌龟按照录制时的轨迹运动。

**5.\*.bag.active恢复**

①切换到xxx.bag.active文件所在的目录下；

②命令行输入rosbag reindex xxx.bag.active；

③输入rosbag fix xxx.bag.active result.bag；

在第二步结束后，除了原来的以.bag.active为后缀的文件之外，还会生成一个以.bag.org.active为后缀的文件，注意该文件只是中间文件，第三部输入的时候不要对该文件进行修复。

在第三步结束之后，会生成正常的.bag文件。

### 2.rosbag使用\_编码

命令实现不够灵活，可以使用编码的方式，增强录制与回放的灵活性(功能包依赖rosbag)

**方案A:C++实现**

**1.写 bag**

#include "ros/ros.h"

#include "rosbag/bag.h"

#include "std\_msgs/String.h"

int main(int argc, char \*argv[])

{

ros::init(argc,argv,"bag\_write");

ros::NodeHandle nh;

//创建bag对象

rosbag::Bag bag;

//打开

//直接test.bag为相对路径，在test\_ws空间下

bag.open(“/home/xuwenyu/test\_ws/src/demo\_bag/bag/test.bag”,rosbag::BagMode::Write);

//写

std\_msgs::String msg;//std::string无法传入

msg.data = “hello world”;

bag.write(“/chatter”,ros::Time::now(),msg);

bag.write(“/chatter”,ros::Time::now(),msg);

bag.write(“/chatter”,ros::Time::now(),msg);

bag.write(“/chatter”,ros::Time::now(),msg);

//关闭

bag.close();

return 0;

}

**2.读bag**

#include “ros/ros.h”

#include “rosbag/bag.h”

#include “rosbag/view.h”

#include “std\_msgs/String.h”

#include “std\_msgs/Int32.h”

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,””);

ros::init(argc,argv,”bag\_read”);

ros::NodeHandle nh;

//创建 bag 对象

rosbag::Bag bag;

//打开 bag 文件

bag.open(“/home/xuwenyu/test\_ws/src/demo\_bag/bag/test.bag “,rosbag::BagMode::Read);

//读数据

//高级for（forrange）：for (auto && i : rosbag::View(bag)){}

for (rosbag::MessageInstance const m : rosbag::View(bag))

{

std::string topic = m.getTopic;

ros::Time time = m.gettime();

//std\_msgs::stringsptr也可以

std\_msgs::String::ConstPtr p = m.instantiate<std\_msgs::String>();//instantiate取出消息的值，返回指针

if(p != nullptr){

ROS\_INFO("读取的数据:%s",p->data.c\_str());

}

}

//关闭文件流

bag.close();

return 0;

}

**方案B:Python实现**

**1.写 bag**

#! /usr/bin/env python

import rospy

import rosbag

from std\_msgs.msg import String

if \_\_name\_\_ == "\_\_main\_\_":

#初始化节点

rospy.init\_node("w\_bag\_p")

# 创建 rosbag 对象

bag = rosbag.Bag("/home/rosdemo/demo/test.bag",'w')

# 写数据

s = String()

s.data= "hahahaha"

bag.write("chatter",s)

bag.write("chatter",s)

bag.write("chatter",s)

# 关闭流

bag.close()

**2.读bag**

#! /usr/bin/env python

import rospy

import rosbag

from std\_msgs.msg import String

if \_\_name\_\_ == "\_\_main\_\_":

#初始化节点

rospy.init\_node("w\_bag\_p")

# 创建 rosbag 对象

bag = rosbag.Bag("/home/rosdemo/demo/test.bag",'r')

# 读数据

bagMessage = bag.read\_messages("chatter")

for topic,msg,t in bagMessage:

rospy.loginfo("%s,%s,%s",topic,msg,t)

# 关闭流

bag.close()

## rqt工具箱

在ROS中，提供了rqt工具箱，在调用工具时以图形化操作代替了命令操作，应用更便利，提高了操作效率，优化了用户体验。

概念：ROS基于 QT 框架，针对机器人开发提供了一系列可视化的工具，这些工具的集合就是rqt

作用：可以方便的实现 ROS 可视化调试，并且在同一窗口中打开多个部件，提高开发效率，优化用户体验。

组成

rqt 工具箱组成有三大部分：

rqt——核心实现，开发人员无需关注

rqt\_common\_plugins——rqt 中常用的工具套件

rqt\_robot\_plugins——运行中和机器人交互的插件(比如: rviz)

### 1.rqt安装启动与基本使用

1.安装

一般只要你安装的是desktop-full版本就会自带工具箱

如果需要安装可以以如下方式安装

sudo apt-get install ros-noetic-rqt

sudo apt-get install ros-noetic-rqt-common-plugins

2.启动

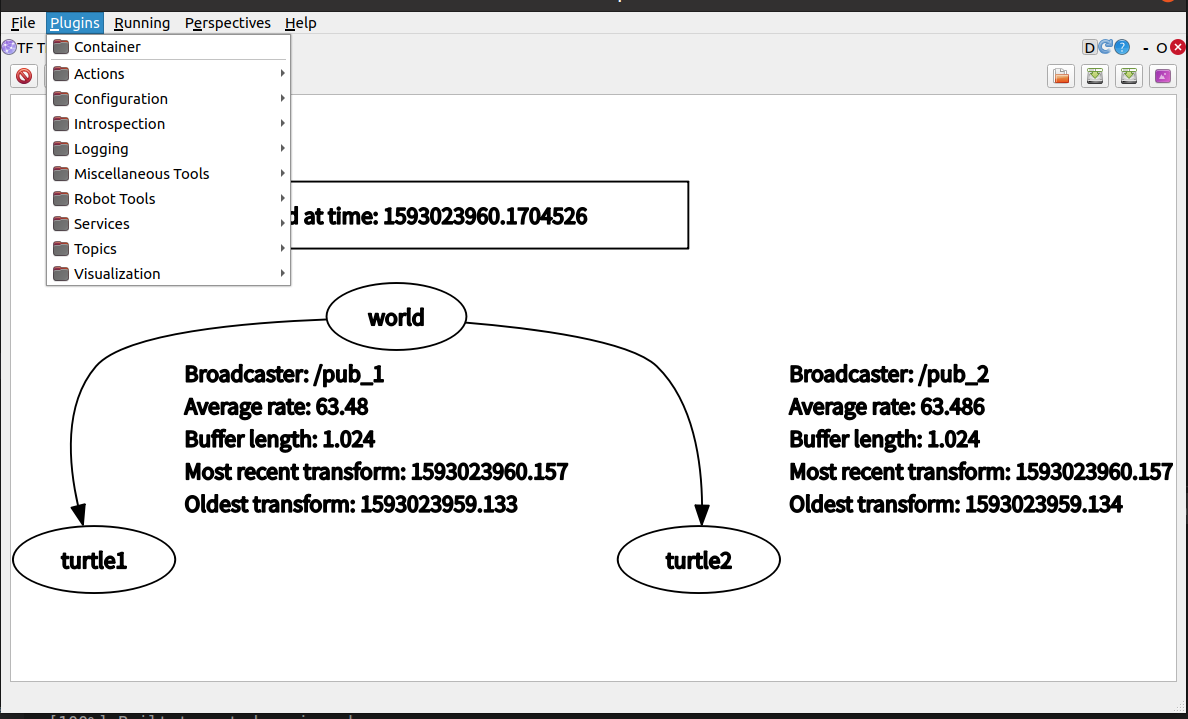
rqt的启动方式有两种:

方式1:rqt

方式2:rosrun rqt\_gui rqt\_gui

3.基本使用

启动 rqt 之后，可以通过 plugins 添加所需的插件



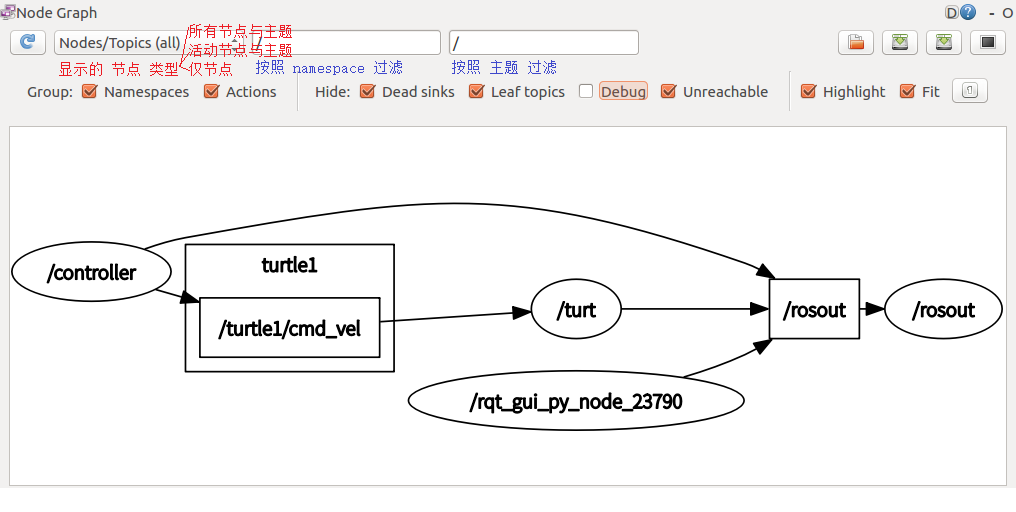
### 2.rqt常用插件:rqt\_graph

简介:可视化显示计算图

启动:可以在 rqt 的 plugins 中添加，或者使用rqt\_graph启动

椭圆：节点

方框：话题



### 3.rqt常用插件:rqt\_console

简介:rqt\_console 是 ROS 中用于显示和过滤日志的图形化插件

准备:编写 Node 节点输出各个级别的日志信息

#include "ros/ros.h"

int main(int argc, char \*argv[])

{

ros::init(argc,argv,"log\_demo");

ros::NodeHandle nh;

ros::Rate r(0.3);

while (ros::ok())

{

ROS\_DEBUG("Debug message d");

ROS\_INFO("Info message oooooooooooooo");

ROS\_WARN("Warn message wwwww");

ROS\_ERROR("Erroe message EEEEEEEEEEEEEEEEEEEE");

ROS\_FATAL("Fatal message FFFFFFFFFFFFFFFFFFFFFFFFFFFFF");

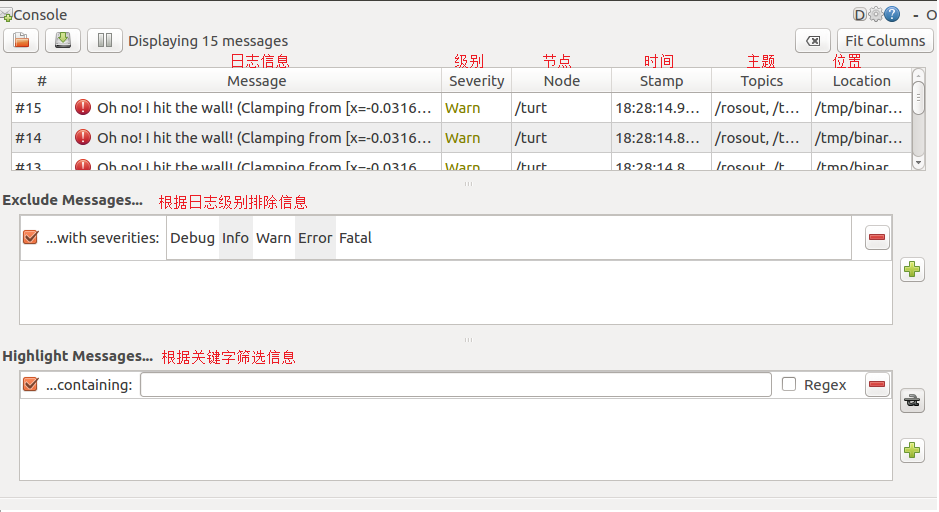
r.sleep();

}

return 0;

}

启动:可以在 rqt 的 plugins 中添加，或者使用rqt\_console启动

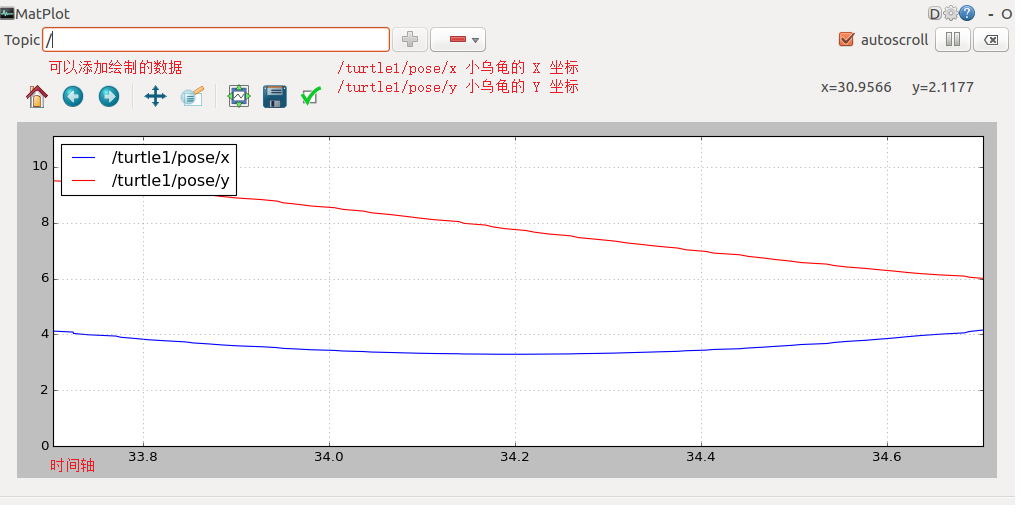


### 4.rqt常用插件:rqt\_plot

简介:图形绘制插件，可以以 2D 绘图的方式绘制发布在 topic 上的数据

准备:启动 turtlesim 乌龟节点与键盘控制节点，通过 rqt\_plot 获取乌龟位姿

启动:可以在 rqt 的 plugins 中添加，或者使用rqt\_plot启动



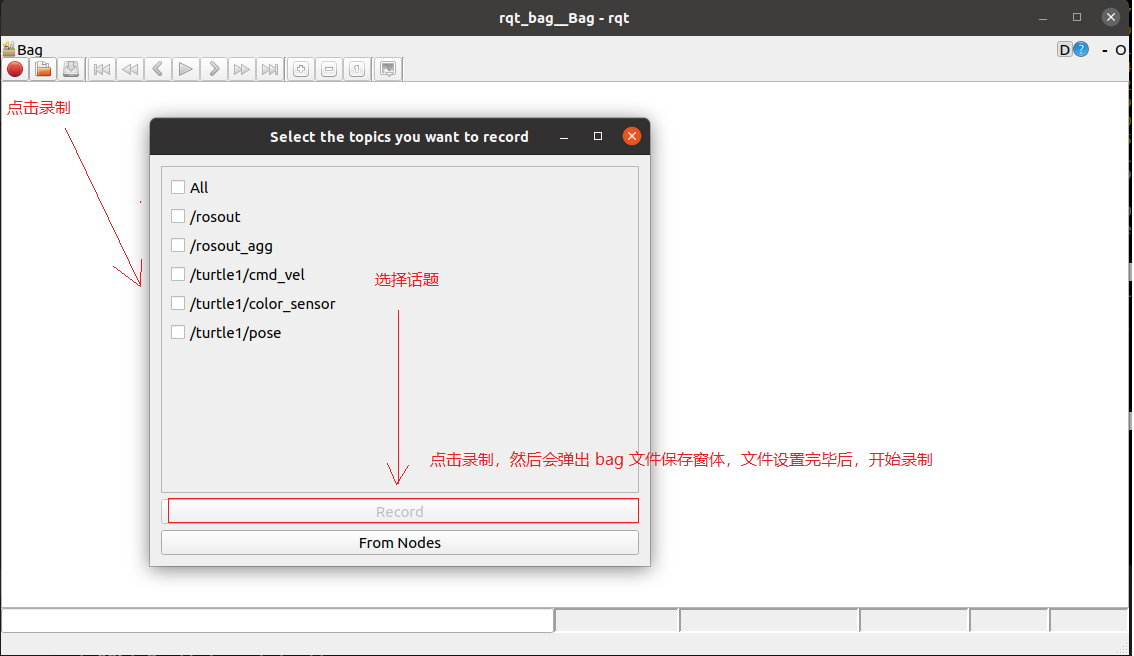
### 5.rqt常用插件:rqt\_bag

简介:录制和重放 bag 文件的图形化插件

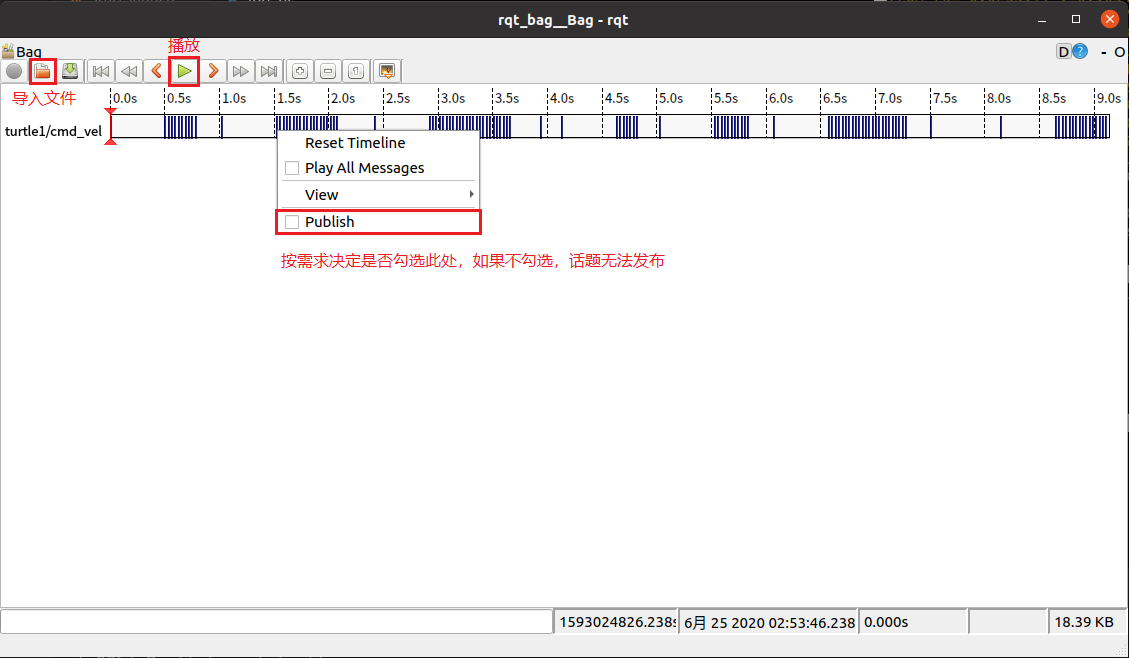
准备:启动 turtlesim 乌龟节点与键盘控制节点

启动:可以在 rqt 的 plugins 中添加，或者使用rqt\_bag启动

录制:



重放:



### rqt常用插件:rqt\_reconfigure

参数动态配置工具

安装ros-melodic-rqt-reconfigure

安装后rosrun rqt\_reconfigure rqt\_reconfigure

# 机器人系统仿真

## 概述

**1.概念**

机器人系统仿真：是通过计算机对实体机器人系统进行模拟的技术，在 ROS 中，仿真实现涉及的内容主要有三:对机器人建模(URDF)、创建仿真环境(Gazebo)以及感知环境(Rviz)等系统性实现。

**2.相关组件**

**2.1URDF**

URDF是 Unified Robot Description Format 的首字母缩写，直译为统一(标准化)机器人描述格式，可以以一种 XML 的方式描述机器人的部分结构，比如底盘、摄像头、激光雷达、机械臂以及不同关节的自由度.....,该文件可以被 C++ 内置的解释器转换成可视化的机器人模型，是 ROS 中实现机器人仿真的重要组件

**2.2rviz**

RViz 是 ROS Visualization Tool 的首字母缩写，直译为ROS的三维可视化工具。它的主要目的是以三维方式显示ROS消息，可以将 数据进行可视化表达。例如:可以显示机器人模型，可以无需编程就能表达激光测距仪（LRF）传感器中的传感 器到障碍物的距离，RealSense、Kinect或Xtion等三维距离传感器的点云数据（PCD， Point Cloud Data），从相机获取的图像值等

以“ros- [ROS\_DISTRO] -desktop-full”命令安装ROS时，RViz会默认被安装。

运行使用命令rviz或rosrun rviz rviz

如果rviz没有安装，请调用如下命令自行安装:

sudo apt install ros-[ROS\_DISTRO]-rviz

**2.3gazebo**

Gazebo是一款3D动态模拟器，用于显示机器人模型并创建仿真环境,能够在复杂的室内和室外环境中准确有效地模拟机器人。与游戏引擎提供高保真度的视觉模拟类似，Gazebo提供高保真度的物理模拟，其提供一整套传感器模型，以及对用户和程序非常友好的交互方式。

以“ros- [ROS\_DISTRO] -desktop-full”命令安装ROS时，gzebo会默认被安装。

运行使用命令gazebo或rosrun gazebo\_ros gazebo

注意2:如果 gazebo没有安装，请自行安装:

1.添加源:

sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb\_release -cs` main"

>

/etc/apt/sources.list.d/gazebo-stable.list'

wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -

2.安装：

sudo apt update

sudo apt install gazebo11

sudo apt install libgazebo11-dev

## URDF集成Rviz基本流程

需求描述:在 Rviz 中显示一个盒状机器人

实现流程： **1.创建功能包，导入依赖**

创建一个新的功能包，名称自定义，导入依赖包:urdf与xacro

在当前功能包下，再新建几个目录:

urdf: 存储 urdf 文件的目录

meshes:机器人模型渲染文件(暂不使用)

config: 配置文件

launch: 存储 launch 启动文件

**2.编写 URDF 文件**

新建一个子级文件夹:urdf(可选)，文件夹中添加一个.urdf文件,复制如下内容(单位:m,rad)

<robot name="mycar">

<link name="base\_link">

<visual>

<geometry>

<box size="0.5 0.2 0.1" />

</geometry>

</visual>

</link>

</robot>

**3.在 launch 文件中集成 URDF 与 Rviz**

在launch目录下，新建一个 launch 文件，该 launch 文件需要启动 Rviz，并导入 urdf 文件，Rviz 启动后可以自动载入解析urdf文件，并显示机器人模型，核心问题:导入 urdf 文件:在 ROS 中，可以将 urdf 文件的路径设置到参数服务器，使用的参数名是:robot\_description,示例代码如下:

<launch>

<!-- 设置参数 -->

<param name="robot\_description" textfile="$(find 包名)/urdf/urdf/urdf01\_HelloWorld.urdf" />

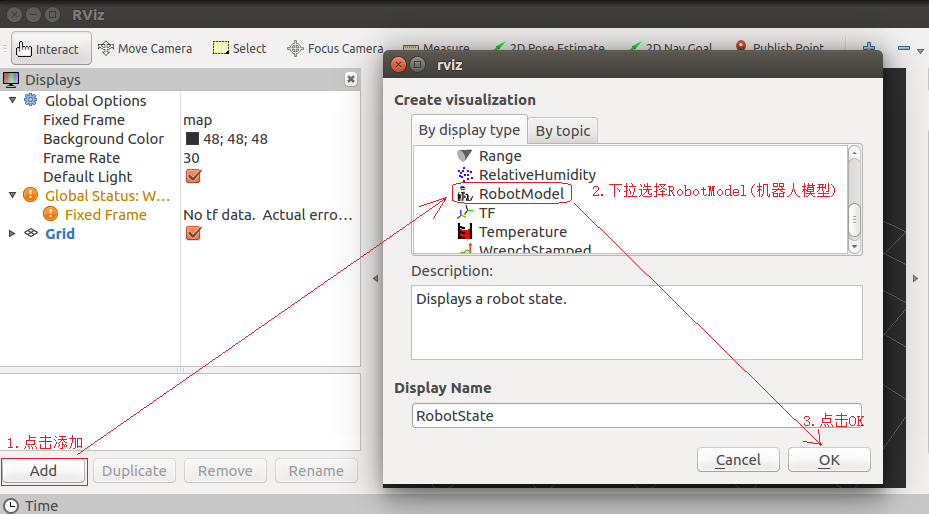
<!-- 启动 rviz -->

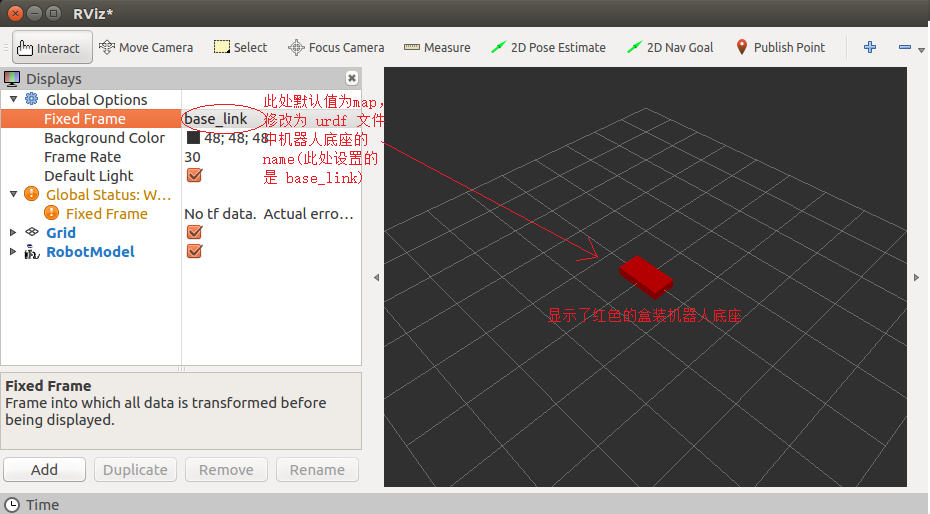
<node pkg="rviz" type="rviz" name="rviz" />

</launch>

**4.在 Rviz 中显示机器人模型**

rviz 启动后，会发现并没有盒装的机器人模型，这是因为默认情况下没有添加机器人显示组件，需要手动添加，添加方式如下:

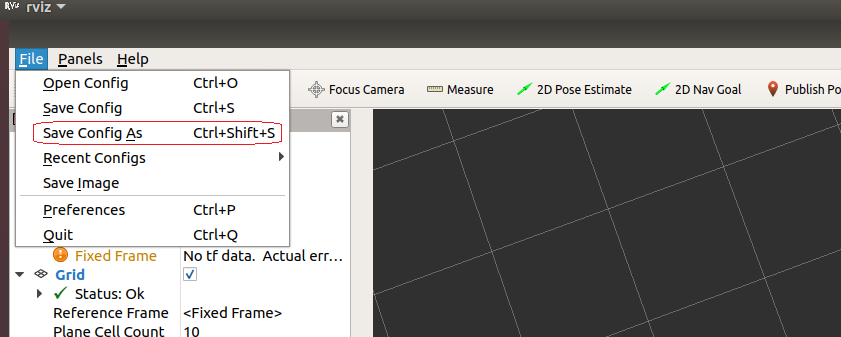




**5.优化 rviz 启动**

重复启动launch文件时，Rviz 之前的组件配置信息不会自动保存，需要重复执行步骤4的操作，为了方便使用，可以使用如下方式优化:

首先，将当前配置保存进config目录



然后，launch文件中 Rviz 的启动配置添加参数:args,值设置为-d 配置文件路径

<launch>

<param name="robot\_description" textfile="$(find 包名)/urdf/urdf/urdf01\_HelloWorld.urdf" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find 报名)/config/rviz/show\_mycar.rviz" />

</launch>

再启动时，就可以包含之前的组件配置了，使用更方便快捷。

## URDF语法详解

URDF 文件是一个标准的 XML 文件，在 ROS 中预定义了一系列的标签用于描述机器人模型

robot 根标签，类似于 launch文件中的launch标签

link 连杆标签

joint 关节标签

gazebo 集成gazebo需要使用的标签

关于gazebo标签，后期在使用 gazebo 仿真时，才需要使用到，用于配置仿真环境所需参数，比如: 机器人材料属性、gazebo插件等，但是该标签不是机器人模型必须的，只有在仿真时才需设置

### 1.URDF语法详解01\_robot

urdf 中为了保证 xml 语法的完整性，使用了robot标签作为根标签，所有的 link 和 joint 以及其他标签都必须包含在 robot 标签内,在该标签内可以通过 name 属性设置机器人模型的名称

**1.属性**

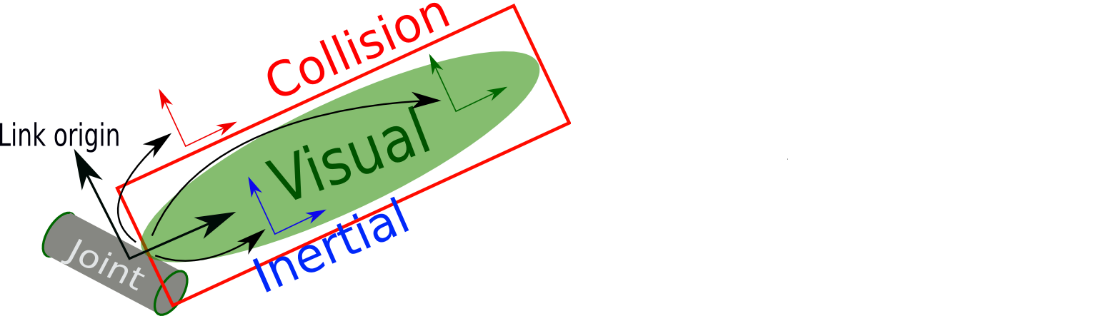
name: 指定机器人模型的名称

**2.子标签**

其他标签都是子级标签

### 2.URDF语法详解02\_link

urdf 中的 link 标签用于描述机器人某个部件(也即刚体部分)的外观和物理属性，比如: 机器人底座、轮子、激光雷达、摄像头...每一个部件都对应一个 link, 在 link 标签内，可以设计该部件的形状、尺寸、颜色、惯性矩阵、碰撞参数等一系列属性。



**1.属性**

name ---> 为连杆命名

**2.子标签**

visual ---> 描述外观(对应的数据是可视的)

geometry 设置连杆的形状

标签1: box(盒状)

属性:size=长(x) 宽(y) 高(z)

标签2: cylinder(圆柱)

属性:radius=半径 length=高度

标签3: sphere(球体)

属性:radius=半径

标签4: mesh(为连杆添加皮肤)

属性: filename=资源路径(格式:< mesh filename="package://<packagename>/<path>/xxx.dae" />)

origin 设置偏移量与倾斜弧度

属性1: xyz=x偏移 y便宜 z偏移

属性2: rpy=x翻滚 y俯仰 z偏航 (单位是弧度)

metrial 设置材料属性(颜色)

属性: name

标签: color

属性: rgba=红绿蓝权重值与透明度 (每个权重值以及透明度取值[0,1])

collision ---> 连杆的碰撞属性

Inertial ---> 连杆的惯性矩阵(惯性参数)

**urdf中含有中文字符导致出现:UnicodeEncodeError: ‘ascii‘ codec can‘t encode characters in position：**

**在melodic的urdf中不能加中文注释，需要在目录/opt/ros/melodic/lib/python2.7/dist-package下新建sitecustomize.py**：

#coding=utf8

import sys

reload(sys)

sys.setdefaultencoding('utf8')

**3.案例**

需求:分别生成长方体、圆柱与球体的机器人部件

<link name="base\_link">

<visual>

<!-- 形状 -->

<geometry>

<!-- 长方体的长宽高 -->

<!-- <box size="0.5 0.3 0.1" /> -->

<!-- 圆柱，半径和长度 -->

<!-- <cylinder radius="0.5" length="0.1" /> -->

<!-- 球体，半径-->

<!-- <sphere radius="0.3" /> -->

</geometry>

<!-- xyz坐标 rpy翻滚俯仰与偏航角度(3.14=180度 1.57=90度) -->

<origin xyz="0 0 0" rpy="0 0 0" />

<!-- 颜色: r=red g=green b=blue a=alpha -->

<material name="black">

<color rgba="0.7 0.5 0 0.5" />

</material>

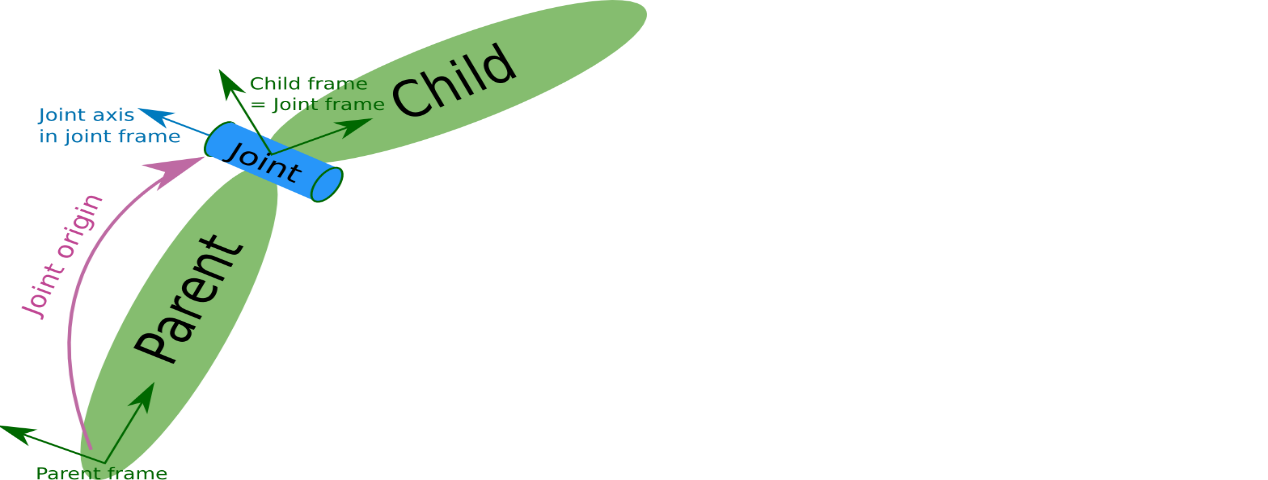
</visual>

</link>

### 3.URDF语法详解03\_joint

urdf 中的 joint 标签用于描述机器人关节的运动学和动力学属性，还可以指定关节运动的安全极限，机器人的两个部件(分别称之为 parent link 与 child link)以"关节"的形式相连接，不同的关节有不同的运动形式: 旋转、滑动、固定、旋转速度、旋转角度限制....,比如:安装在底座上的轮子可以360度旋转，而摄像头则可能是完全固定在底座上。

joint标签对应的数据在模型中是不可见的



**1.属性**

name ---> 为关节命名

type ---> 关节运动形式

continuous: 旋转关节，可以绕单轴无限旋转

revolute: 旋转关节，类似于 continues,但是有旋转角度限制

prismatic: 滑动关节，沿某一轴线移动的关节，有位置极限

planer: 平面关节，允许在平面正交方向上平移或旋转

floating: 浮动关节，允许进行平移、旋转运动

fixed: 固定关节，不允许运动的特殊关节

**2.子标签**

parent(必需的)

parent link的名字是一个强制的属性：

link:父级连杆的名字，是这个link在机器人结构树中的名字。

child(必需的)

child link的名字是一个强制的属性：

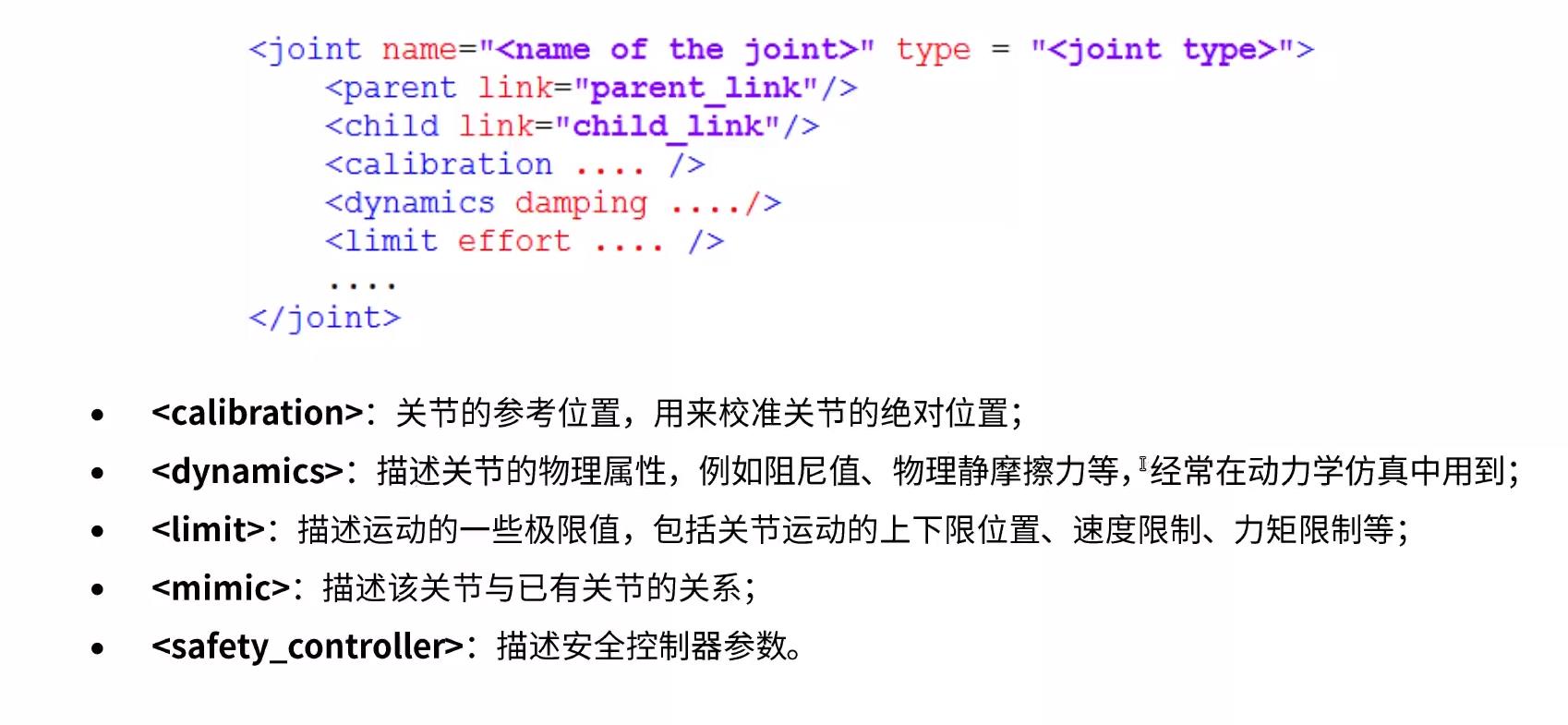
link:子级连杆的名字，是这个link在机器人结构树中的名字。

origin

属性: xyz=各轴线上的偏移量 rpy=各轴线上的偏移弧度。

axis

属性: xyz用于设置围绕哪个关节轴运动。



**3.案例**

需求:创建机器人模型，底盘为长方体，在长方体的前面添加一摄像头，摄像头可以沿着 Z轴 360 度旋转。

<!--

需求: 创建机器人模型，底盘为长方体，

在长方体的前面添加一摄像头，

摄像头可以沿着 Z 轴 360 度旋转

-->

<robot name="mycar">

<!-- 底盘 -->

<link name="base\_link">

<visual>

<geometry>

<box size="0.5 0.2 0.1" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="blue">

<color rgba="0 0 1.0 0.5" />

</material>

</visual>

</link>

<!-- 摄像头 -->

<link name="camera">

<visual>

<geometry>

<box size="0.02 0.05 0.05" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="red">

<color rgba="1 0 0 0.5" />

</material>

</visual>

</link>

<!-- 关节 -->

<joint name="camera2baselink" type="continuous">

<parent link="base\_link"/>

<child link="camera" />

<!-- 需要计算两个 link 的物理中心之间的偏移量 -->

<origin xyz="0.2 0 0.075" rpy="0 0 0" />

<axis xyz="0 0 1" />

</joint>

</robot>

launch文件示例如下:

<launch>

<param name="robot\_description" textfile="$(find demo\_urdf)/urdf/joint.urdf" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find demo\_urdf)/config/joint.rviz" />

<!-- 添加关节状态发布节点（发布每个joint（除fixed类型）的状态，而且可以通过UI界面对joint进行控制） -->

<node pkg="joint\_state\_publisher" type="joint\_state\_publisher" name="joint\_state\_publisher" />

<!-- 添加机器人状态发布节点（将机器人各个links、joints之间的关系通过TF的形式，整理成三维位姿信息发布） -->

<node pkg="robot\_state\_publisher" type="robot\_state\_publisher" name="robot\_state\_publisher" />

<!-- 可选:用于控制关节运动的节点 -->

<node pkg="joint\_state\_publisher\_gui" type="joint\_state\_publisher\_gui" name="joint\_state\_publisher\_gui" />

</launch>

PS:

1.状态发布节点在此是必须的; 2.关节运动控制节点(可选)，会生成关节控制的UI，用于测试关节运动是否正常。

**4.base\_footprint优化urdf**

前面实现的机器人模型是半沉到地下的，因为默认情况下: 底盘的中心点位于地图原点上，所以会导致这种情况产生，可以使用的优化策略，将初始 link 设置为一个尺寸极小的 link(比如半径为 0.001m 的球体，或边长为 0.001m 的立方体)，然后再在初始 link 上添加底盘等刚体，这样实现，虽然仍然存在初始link半沉的现象，但是基本可以忽略了。这个初始 link 一般称之为 base\_footprint

<!--使用 base\_footprint 优化-->

<robot name="mycar">

<!-- 设置一个原点(机器人中心点的投影) -->

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="0.001" />

</geometry>

</visual>

</link>

<!-- 添加底盘 -->

<link name="base\_link">

<visual>

<geometry>

<box size="0.5 0.2 0.1" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="blue">

<color rgba="0 0 1.0 0.5" />

</material>

</visual>

</link>

<!-- 底盘与原点连接的关节 -->

<joint name="base\_link2base\_footprint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link" />

<origin xyz="0 0 0.05" />

</joint>

<!-- 添加摄像头 -->

<link name="camera">

<visual>

<geometry>

<box size="0.02 0.05 0.05" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="red">

<color rgba="1 0 0 0.5" />

</material>

</visual>

</link>

<!-- 关节 -->

<joint name="camera2baselink" type="continuous">

<parent link="base\_link"/>

<child link="camera" />

<origin xyz="0.2 0 0.075" rpy="0 0 0" />

<axis xyz="0 0 1" />

</joint>

</robot>

**5.遇到问题以及解决**

问题1:命令行输出如下错误提示

UnicodeEncodeError: 'ascii' codec can't encode characters in position 463-464: ordinal not in range(128)

[joint\_state\_publisher-3] process has died [pid 4443, exit code 1, cmd /opt/ros/melodic/lib/joint\_state\_publisher/joint\_state\_publisher \_\_name:=joint\_state\_publisher \_\_log:=/home/rosmelodic/.ros/log/b38967c0-0acb-11eb-aee3-0800278ee10c/joint\_state\_publisher-3.log].

log file: /home/rosmelodic/.ros/log/b38967c0-0acb-11eb-aee3-0800278ee10c/joint\_state\_publisher-3\*.log

rviz中提示坐标变换异常，导致机器人部件显示结构异常

原因:编码问题导致的

解决:去除URDF中的中文注释

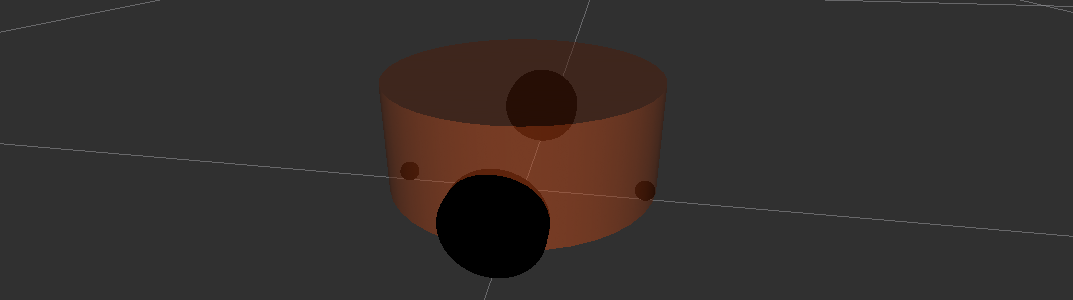
问题2:[ERROR] [1584370263.037038]: Could not find the GUI, install the 'joint\_state\_publisher\_gui' package

解决:sudo apt install ros-melodic-joint-state-publisher-gui

### 4.URDF练习

需求描述:创建一个四轮圆柱状机器人模型，机器人参数如下,底盘为圆柱状，半径 10cm，高 8cm，四轮由两个驱动轮和两个万向支撑轮组成，两个驱动轮半径为 3.25cm,轮胎宽度1.5cm，两个万向轮为球状，半径 0.75cm，底盘离地间距为 1.5cm(与万向轮直径一致)

结果演示:



实现流程: **1.新建urdf以及launch文件**

**urdf 文件**

<robot name="mycar">

<!-- 设置 base\_footprint -->

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="0.001" />

</geometry>

</visual>

</link>

<!-- 添加底盘 -->

<!-- 添加驱动轮 -->

<!-- 添加万向轮(支撑轮) -->

</robot>

**launch 文件**

<launch>

<!-- 将 urdf 文件内容设置进参数服务器 -->

<param name="robot\_description" textfile="$(find demo\_urdf)/urdf/test.urdf" />

<!-- 启动 rivz -->

<node pkg="rviz" type="rviz" name="rviz\_test" args="-d $(find demourdf)/config/helloworld.rviz" />

<!-- 启动机器人状态和关节状态发布节点 -->

<node pkg="robot\_state\_publisher" type="robot\_state\_publisher" name="robot\_state\_publisher" />

<node pkg="joint\_state\_publisher" type="joint\_state\_publisher" name="joint\_state\_publisher" />

<!-- 启动图形化的控制关节运动节点 -->

<node pkg="joint\_state\_publisher\_gui" type="joint\_state\_publisher\_gui" name="joint\_state\_publisher\_gui" />

</launch>

**2.底盘搭建**

<!-- 参数

形状:圆柱

半径:10 cm

高度:8 cm

离地:1.5 cm -->

<link name="base\_link">

<visual>

<geometry>

<cylinder radius="0.1" length="0.08" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="yellow">

<color rgba="0.8 0.3 0.1 0.5" />

</material>

</visual>

</link>

<joint name="base\_link2base\_footprint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link"/>

<origin xyz="0 0 0.055" />

</joint>

**3.添加驱动轮**

<!-- 添加驱动轮 -->

<!--

驱动轮是侧翻的圆柱

参数

半径: 3.25 cm

宽度: 1.5 cm

颜色: 黑色

关节设置:

x = 0

y = 底盘的半径 + 轮胎宽度 / 2

z = 离地间距 + 底盘长度 / 2 - 轮胎半径 = 1.5 + 4 - 3.25 = 2.25(cm)

axis = 0 1 0

-->

<link name="left\_wheel">

<visual>

<geometry>

<cylinder radius="0.0325" length="0.015" />

</geometry>

<origin xyz="0 0 0" rpy="1.5705 0 0" />

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

</visual>

</link>

<joint name="left\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="left\_wheel" />

<origin xyz="0 0.1 -0.0225" />

<axis xyz="0 1 0" />

</joint>

<link name="right\_wheel">

<visual>

<geometry>

<cylinder radius="0.0325" length="0.015" />

</geometry>

<origin xyz="0 0 0" rpy="1.5705 0 0" />

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

</visual>

</link>

<joint name="right\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="right\_wheel" />

<origin xyz="0 -0.1 -0.0225" />

<axis xyz="0 1 0" />

</joint>

**4.添加万向轮**

<!-- 添加万向轮(支撑轮) -->

<!--

参数

形状: 球体

半径: 0.75 cm

颜色: 黑色

关节设置:

x = 自定义(底盘半径 - 万向轮半径) = 0.1 - 0.0075 = 0.0925(cm)

y = 0

z = 底盘长度 / 2 + 离地间距 / 2 = 0.08 / 2 + 0.015 / 2 = 0.0475

axis= 1 1 1

-->

<link name="front\_wheel">

<visual>

<geometry>

<sphere radius="0.0075" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

</visual>

</link>

<joint name="front\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="front\_wheel" />

<origin xyz="0.0925 0 -0.0475" />

<axis xyz="1 1 1" />

</joint>

<link name="back\_wheel">

<visual>

<geometry>

<sphere radius="0.0075" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

</visual>

</link>

<joint name="back\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="back\_wheel" />

<origin xyz="-0.0925 0 -0.0475" />

<axis xyz="1 1 1" />

</joint>

### 5.URDF工具

在 ROS 中，提供了一些工具来方便 URDF 文件的编写，比如:

check\_urdf命令可以检查复杂的 urdf 文件是否存在语法问题

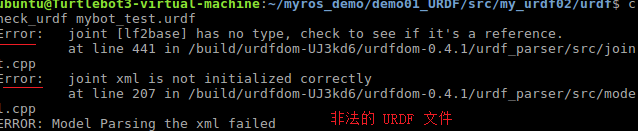
urdf\_to\_graphiz命令可以查看 urdf 模型结构，显示不同 link 的层级关系

当然，要使用工具之前，首先需要安装，安装命令:sudo apt install liburdfdom-tools

1.check\_urdf 语法检查

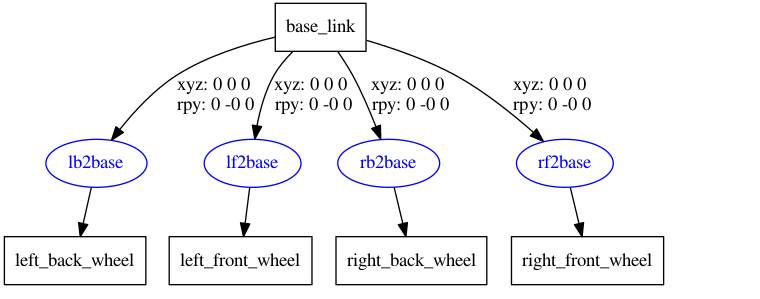
进入urdf文件所属目录，调用:check\_urdf urdf文件，如果不抛出异常，说明文件合法,否则非法





2.urdf\_to\_graphiz 结构查看

进入urdf文件所属目录，调用:urdf\_to\_graphiz urdf文件，当前目录下会生成 pdf 文件



## URDF优化\_xacro

概念:Xacro 是 XML Macros 的缩写，Xacro 是一种 XML 宏语言，是可编程的 XML。

原理:

Xacro 可以声明变量，可以通过数学运算求解，使用流程控制控制执行顺序，还可以通过类似函数的实现，封装固定的逻辑，将逻辑中需要的可变的数据以参数的方式暴露出去，从而提高代码复用率以及程序的安全性。

### 1.Xacro\_快速体验

需求描述:

使用xacro优化上一节案例中驱动轮实现，需要使用变量封装底盘的半径、高度，使用数学公式动态计算底盘的关节点坐标，使用 Xacro 宏封装轮子重复的代码并调用宏创建两个轮子(注意: 在此，演示 Xacro 的基本使用，不必要生成合法的 URDF )。

准备:创建功能包，导入 urdf 与 xacro。

**1.Xacro文件编写**

编写 Xacro 文件，以变量的方式封装属性(常量半径、高度、车轮半径...)，以函数的方式封装重复实现(车轮的添加)。

<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 属性封装 -->

<xacro:property name="wheel\_radius" value="0.0325" />

<xacro:property name="wheel\_length" value="0.0015" />

<xacro:property name="PI" value="3.1415927" />

<xacro:property name="base\_link\_length" value="0.08" />

<xacro:property name="lidi\_space" value="0.015" />

<!-- 宏 -->

<xacro:macro name="wheel\_func" params="wheel\_name flag" >

<link name="${wheel\_name}\_wheel">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="${PI / 2} 0 0" />

<material name="wheel\_color">

<color rgba="0 0 0 0.3" />

</material>

</visual>

</link>

<!-- joint -->

<joint name="${wheel\_name}2link" type="continuous">

<parent link="base\_link" />

<child link="${wheel\_name}\_wheel" />

<!--

x 无偏移

y 车体半径

z z= 车体高度 / 2 + 离地间距 - 车轮半径

-->

<origin xyz="0 ${0.1 \* flag} ${(base\_link\_length / 2 + lidi\_space - wheel\_radius) \* -1}" rpy="0 0 0" />

<axis xyz="0 1 0" />

</joint>

</xacro:macro>

<xacro:wheel\_func wheel\_name="left" flag="1" />

<xacro:wheel\_func wheel\_name="right" flag="-1" />

</robot>

**2.Xacro文件转换成 urdf 文件**

命令行进入 xacro文件 所属目录，执行:rosrun xacro xacro xxx.xacro > xxx.urdf, 会将 xacro 文件解析为 urdf 文件。

### 2.Xacro\_语法详解

xacro 提供了可编程接口，类似于计算机语言，包括变量声明调用、函数声明与调用等语法实现。在使用 xacro 生成 urdf 时，根标签robot中必须包含命名空间声明:xmlns:xacro=<http://wiki.ros.org/xacro>

1.属性与算数运算

用于封装 URDF 中的一些字段，比如: PAI 值，小车的尺寸，轮子半径 ....

属性定义：<xacro:property name="xxxx" value="yyyy" />

属性调用：${属性名称}

算数运算：${数学表达式}

2.宏

类似于函数实现，提高代码复用率，优化代码结构，提高安全性

宏定义：

<xacro:macro name="宏名称" params="参数列表(多参数之间使用空格分隔)">

.....

参数调用格式: ${参数名}

</xacro:macro>

宏调用：<xacro:宏名称 参数1=“xxx” 参数2=“xxx” />

3.文件包含

机器人由多部件组成，不同部件可能封装为单独的 xacro 文件，最后再将不同的文件集成，组合为完整机器人，可以使用文件包含实现

<robot name="xxx" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="my\_base.xacro" />

<xacro:include filename="my\_camera.xacro" />

<xacro:include filename="my\_laser.xacro" />

....

</robot>

### 3．Xacro\_完整使用流程示例

需求描述:使用 Xacro 优化 URDF 版的小车底盘模型实现

**1.编写 Xacro 文件**

<!--

使用 xacro 优化 URDF 版的小车底盘实现：

实现思路:

1.将一些常量、变量封装为 xacro:property

比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度 ....

2.使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动轮与支撑轮

-->

<!-- 根标签，必须声明 xmlns:xacro -->

<robot name="my\_base" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- 封装变量、常量 -->

<xacro:property name="PI" value="3.141"/>

<!-- 宏:黑色设置 -->

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

<!-- 底盘属性 -->

<xacro:property name="base\_footprint\_radius" value="0.001" /> <!-- base\_footprint 半径 -->

<xacro:property name="base\_link\_radius" value="0.1" /> <!-- base\_link 半径 -->

<xacro:property name="base\_link\_length" value="0.08" /> <!-- base\_link 长 -->

<xacro:property name="earth\_space" value="0.015" /> <!-- 离地间距 -->

<!-- 底盘 -->

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="${base\_footprint\_radius}" />

</geometry>

</visual>

</link>

<link name="base\_link">

<visual>

<geometry>

<cylinder radius="${base\_link\_radius}" length="${base\_link\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="yellow">

<color rgba="0.5 0.3 0.0 0.5" />

</material>

</visual>

</link>

<joint name="base\_link2base\_footprint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link" />

<origin xyz="0 0 ${earth\_space + base\_link\_length / 2 }" />

</joint>

<!-- 驱动轮 -->

<!-- 驱动轮属性 -->

<xacro:property name="wheel\_radius" value="0.0325" /><!-- 半径 -->

<xacro:property name="wheel\_length" value="0.015" /><!-- 宽度 -->

<!-- 驱动轮宏实现 -->

<xacro:macro name="add\_wheels" params="name flag">

<link name="${name}\_wheel">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />

<material name="black" />

</visual>

</link>

<joint name="${name}\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="${name}\_wheel" />

<origin xyz="0 ${flag \* base\_link\_radius} ${-(earth\_space + base\_link\_length / 2 - wheel\_radius) }" />

<axis xyz="0 1 0" />

</joint>

</xacro:macro>

<xacro:add\_wheels name="left" flag="1" />

<xacro:add\_wheels name="right" flag="-1" />

<!-- 支撑轮 -->

<!-- 支撑轮属性 -->

<xacro:property name="support\_wheel\_radius" value="0.0075" /> <!-- 支撑轮半径 -->

<!-- 支撑轮宏 -->

<xacro:macro name="add\_support\_wheel" params="name flag" >

<link name="${name}\_wheel">

<visual>

<geometry>

<sphere radius="${support\_wheel\_radius}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="black" />

</visual>

</link>

<joint name="${name}\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="${name}\_wheel" />

<origin xyz="${flag \* (base\_link\_radius - support\_wheel\_radius)} 0 ${-(base\_link\_length / 2 + earth\_space / 2)}" />

<axis xyz="1 1 1" />

</joint>

</xacro:macro>

<xacro:add\_support\_wheel name="front" flag="1" />

<xacro:add\_support\_wheel name="back" flag="-1" />

</robot>

**2.集成launch文件**

方式1:先将 xacro 文件转换出 urdf 文件，然后集成

先将 xacro 文件解析成 urdf 文件:rosrun xacro xacro xxx.xacro > xxx.urdf然后再按照之前的集成方式直接整合 launch 文件,内容示例:

<launch>

<param name="robot\_description" textfile="$(find demo\_urdf)/urdf/my\_base.urdf" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find demo\_urdf)/config/helloworld.rviz" />

<node pkg="joint\_state\_publisher" type="joint\_state\_publisher" name="joint\_state\_publisher" output="screen" />

<node pkg="robot\_state\_publisher" type="robot\_state\_publisher" name="robot\_state\_publisher" output="screen" />

<node pkg="joint\_state\_publisher\_gui" type="joint\_state\_publisher\_gui" name="joint\_state\_publisher\_gui" output="screen" />

</launch>

方式2:在 launch 文件中直接加载 xacro(建议使用)

<launch>

<!—加载robot\_description时使用command属性，属性值就是调用 xacro 功能包的 xacro 程序直接解析 xacro 文件。 -->

<param name="robot\_description" command="$(find xacro)/xacro $(find demo\_urdf)/urdf/my\_base.xacro" />

<!—也可以是两个后缀的my\_base.urdf.xacro -->

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find demo\_urdf)/config/helloworld.rviz" />

<node pkg="joint\_state\_publisher" type="joint\_state\_publisher" name="joint\_state\_publisher" output="screen" />

<node pkg="robot\_state\_publisher" type="robot\_state\_publisher" name="robot\_state\_publisher" output="screen" />

<node pkg="joint\_state\_publisher\_gui" type="joint\_state\_publisher\_gui" name="joint\_state\_publisher\_gui" output="screen" />

</launch>

### 4.Xacro\_实操

需求描述:在前面小车底盘基础之上，添加摄像头和雷达传感器。

实现流程:

**1.摄像头和雷达 Xacro 文件实现**

摄像头 xacro 文件:

<!-- 摄像头相关的 xacro 文件 -->

<robot name="my\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 摄像头属性 -->

<xacro:property name="camera\_length" value="0.01" /> <!-- 摄像头长度(x) -->

<xacro:property name="camera\_width" value="0.025" /> <!-- 摄像头宽度(y) -->

<xacro:property name="camera\_height" value="0.025" /> <!-- 摄像头高度(z) -->

<xacro:property name="camera\_x" value="0.08" /> <!-- 摄像头安装的x坐标 -->

<xacro:property name="camera\_y" value="0.0" /> <!-- 摄像头安装的y坐标 -->

<xacro:property name="camera\_z" value="${base\_link\_length / 2 + camera\_height / 2}" /> <!-- 摄像头安装的z坐标:底盘高度 / 2 + 摄像头高度 / 2 -->

<!-- 摄像头关节以及link -->

<link name="camera">

<visual>

<geometry>

<box size="${camera\_length} ${camera\_width} ${camera\_height}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="black" />

</visual>

</link>

<joint name="camera2base\_link" type="fixed">

<parent link="base\_link" />

<child link="camera" />

<origin xyz="${camera\_x} ${camera\_y} ${camera\_z}" />

</joint>

</robot>

雷达 xacro 文件:

<!--小车底盘添加雷达-->

<robot name="my\_laser" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 雷达支架 -->

<xacro:property name="support\_length" value="0.15" /> <!-- 支架长度 -->

<xacro:property name="support\_radius" value="0.01" /> <!-- 支架半径 -->

<xacro:property name="support\_x" value="0.0" /> <!-- 支架安装的x坐标 -->

<xacro:property name="support\_y" value="0.0" /> <!-- 支架安装的y坐标 -->

<xacro:property name="support\_z" value="${base\_link\_length / 2 + support\_length / 2}" /> <!-- 支架安装的z坐标:底盘高度 / 2 + 支架高度 / 2 -->

<link name="support">

<visual>

<geometry>

<cylinder radius="${support\_radius}" length="${support\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="red">

<color rgba="0.8 0.2 0.0 0.8" />

</material>

</visual>

</link>

<joint name="support2base\_link" type="fixed">

<parent link="base\_link" />

<child link="support" />

<origin xyz="${support\_x} ${support\_y} ${support\_z}" />

</joint>

<!-- 雷达属性 -->

<xacro:property name="laser\_length" value="0.05" /> <!-- 雷达长度 -->

<xacro:property name="laser\_radius" value="0.03" /> <!-- 雷达半径 -->

<xacro:property name="laser\_x" value="0.0" /> <!-- 雷达安装的x坐标 -->

<xacro:property name="laser\_y" value="0.0" /> <!-- 雷达安装的y坐标 -->

<xacro:property name="laser\_z" value="${support\_length / 2 + laser\_length / 2}" /> <!-- 雷达安装的z坐标:支架高度 / 2 + 雷达高度 / 2 -->

<!-- 雷达关节以及link -->

<link name="laser">

<visual>

<geometry>

<cylinder radius="${laser\_radius}" length="${laser\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="black" />

</visual>

</link>

<joint name="laser2support" type="fixed">

<parent link="support" />

<child link="laser" />

<origin xyz="${laser\_x} ${laser\_y} ${laser\_z}" />

</joint>

</robot>

**2.组合底盘摄像头与雷达的 xacro 文件**

<!-- 组合小车底盘与摄像头与雷达 -->

<robot name="my\_car\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="my\_base.urdf.xacro" />

<xacro:include filename="my\_camera.urdf.xacro" />

<xacro:include filename="my\_laser.urdf.xacro" />

</robot>

**3.launch 文件**

<launch>

<param name="robot\_description" command="$(find xacro)/xacro $(find demo\_urdf)/urdf/xacro/my\_base\_camera\_laser.xacro" />

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find demo\_urdf)/config/helloworld.rviz" />

<node pkg="joint\_state\_publisher" type="joint\_state\_publisher" name="joint\_state\_publisher" output="screen" />

<node pkg="robot\_state\_publisher" type="robot\_state\_publisher" name="robot\_state\_publisher" output="screen" />

<node pkg="joint\_state\_publisher\_gui" type="joint\_state\_publisher\_gui" name="joint\_state\_publisher\_gui" output="screen" />

</launch>

## Rviz中控制机器人模型运动

Arbotix:Arbotix 是一款控制电机、舵机的控制板，并提供相应的 ros 功能包，这个功能包的功能不仅可以驱动真实的 Arbotix 控制板，它还提供一个差速控制器，通过接受速度控制指令更新机器人的 joint 状态，从而帮助我们实现机器人在 rviz 中的运动。

这个差速控制器在 arbotix\_python 程序包中，完整的 arbotix 程序包还包括多种控制器，分别对应 dynamixel 电机、多关节机械臂以及不同形状的夹持器。

**Arbotix使用流程**

需求描述:控制机器人模型在 rviz 中做圆周运动

实现流程: **1.安装 Arbotix**

方式1:命令行调用

sudo apt-get install ros-melodic-arbotix

方式2:源码安装

先从 github 下载源码，然后调用 catkin\_make 编译

git clone <https://github.com/vanadiumlabs/arbotix_ros.git>

**2.创建新功能包，准备机器人 urdf、xacro**

urdf 和 xacro 调用上一讲实现即可

**3.添加 arbotix 所需的配置文件（yaml文件）**

# 该文件是控制器配置,一个机器人模型可能有多个控制器，比如: 底盘、机械臂、夹持器(机械手)....

# 因此，根 name 是 controller

controllers: {

# 单控制器设置

base\_controller: {

#类型: 差速控制器

type: diff\_controller,

#参考坐标

base\_frame\_id: base\_footprint,

#两个轮子之间的间距

base\_width: 0.2,

#控制频率

ticks\_meter: 2000,

#PID控制参数，使机器人车轮快速达到预期速度

Kp: 12,

Kd: 12,

Ki: 0,

Ko: 50,

#加速限制

accel\_limit: 1.0

}

}

**4.launch 文件中配置 arbotix 节点**

<node name="arbotix" pkg="arbotix\_python" type="arbotix\_driver" output="screen">

<rosparam file="$(find demo\_urdf)/config/arbotix.yaml" command="load" />

<param name="sim" value="true" />

</node>

代码解释:

<node> 调用了 arbotix\_python 功能包下的 arbotix\_driver 节点

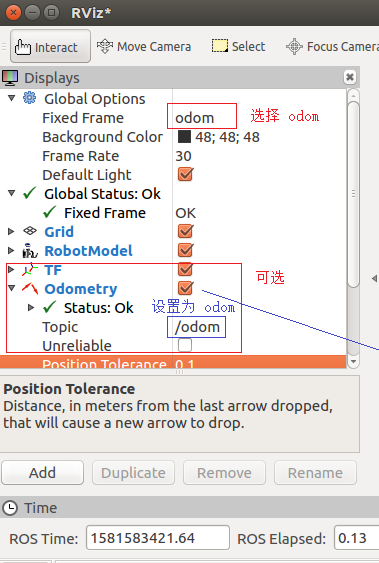
<rosparam> arbotix 驱动机器人运行时，需要获取机器人信息，可以通过 file 加载配置文件

<param> 在仿真环境下，需要配置 sim 为 true

**5.启动 launch 文件并控制机器人模型运动**

启动launch:roslaunch xxxx ....launch

配置 rviz:



控制小车运动: 可以发布 cmd\_vel 话题消息控制小车运动

## URDF集成Gazebo

### URDF与Gazebo基本集成流程

**1.创建功能包**

创建新功能包，导入依赖包: **urdf、xacro、gazebo\_ros、gazebo\_ros\_control、gazebo\_plugins**

**2.编写URDF文件**

<!-- 创建一个机器人模型(盒状即可)，显示在 Gazebo 中 -->

<robot name="mycar">

<link name="base\_link">

<visual>

<geometry>

<box size="0.5 0.2 0.1" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="yellow">

<color rgba="0.5 0.3 0.0 1" />

</material>

</visual>

<collision>

<geometry>

<box size="0.5 0.2 0.1" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

</collision>

<inertial>

<origin xyz="0 0 0" />

<mass value="6" />

<inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1" />

</inertial>

</link>

<gazebo reference="base\_link">

<material>Gazebo/Black</material>

</gazebo>

</robot>

注意， 当 URDF 需要与 Gazebo 集成时，和 Rviz 有明显区别:

1.必须使用 collision 标签，因为既然是仿真环境，那么必然涉及到碰撞检测，collision 提供碰撞检测的依据。

2.必须使用 inertial 标签，此标签标注了当前机器人某个刚体部分的惯性矩阵，用于一些力学相关的仿真计算。

3.颜色设置，也需要重新使用 gazebo 标签标注，因为之前的颜色设置为了方便调试包含透明度，仿真环境下没有此选项。

4.Gazebo报错 [Err] [REST.cc:205] Error in REST request：

sudo gedit ~/.ignition/fuel/config.yaml

将url: https://api.ignitionfuel.org

改为url: https://api.ignitionrobotics.org

**3.启动Gazebo并显示模型**

<launch>

<!-- 将 Urdf 文件的内容加载到参数服务器 -->

<param name="robot\_description" textfile="$(find demo\_gazebo)/urdf/helloworld.urdf" />

<!-- 启动 gazebo的仿真环境，当前环境为空环境 -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch" />

<!-- 在 gazebo 中显示机器人模型 -->

<node pkg="gazebo\_ros" type="spawn\_model" name="model" args="-urdf -model mycar（自定义的机器人名称） -param robot\_description" />

<!--

在 Gazebo 中加载一个机器人模型，该功能由 gazebo\_ros 下的 spawn\_model 提供:

-urdf 加载的是 urdf 文件

-model mycar 模型名称是 mycar

-param robot\_description 从参数 robot\_description 中载入模型

-x 模型载入的 x 坐标

-y 模型载入的 y 坐标

-z 模型载入的 z 坐标

-->

</launch>

### 2.URDF集成Gazebo相关设置

**1.collision**

如果机器人link是标准的几何体形状，和link的 visual 属性设置一致即可，除material。

**2.inertial**

惯性矩阵的设置需要结合link的质量与外形参数动态生成，标准的球体、圆柱与立方体的惯性矩阵公式如下(已经封装为 xacro 实现):

球体惯性矩阵

<!-- Macro for inertia matrix -->

<xacro:macro name="sphere\_inertial\_matrix" params="m r">

<inertial>

<mass value="${m}" />

<inertia ixx="${2\*m\*r\*r/5}" ixy="0" ixz="0"

iyy="${2\*m\*r\*r/5}" iyz="0"

izz="${2\*m\*r\*r/5}" />

</inertial>

</xacro:macro>

圆柱惯性矩阵

<xacro:macro name="cylinder\_inertial\_matrix" params="m r h">

<inertial>

<mass value="${m}" />

<inertia ixx="${m\*(3\*r\*r+h\*h)/12}" ixy = "0" ixz = "0"

iyy="${m\*(3\*r\*r+h\*h)/12}" iyz = "0"

izz="${m\*r\*r/2}" />

</inertial>

</xacro:macro>

立方体惯性矩阵

<xacro:macro name="Box\_inertial\_matrix" params="m l w h">

<inertial>

<mass value="${m}" />

<inertia ixx="${m\*(h\*h + l\*l)/12}" ixy = "0" ixz = "0"

iyy="${m\*(w\*w + l\*l)/12}" iyz= "0"

izz="${m\*(w\*w + h\*h)/12}" />

</inertial>

</xacro:macro>

需要注意的是，原则上，除了 base\_footprint 外，机器人的每个刚体部分都需要设置惯性矩阵，且惯性矩阵必须经计算得出，如果随意定义刚体部分的惯性矩阵，那么可能会导致机器人在 Gazebo 中出现抖动，移动等现象。

**3.颜色设置**

在 gazebo 中显示 link 的颜色，必须要使用指定的标签:

<gazebo reference="link节点名称">

<material>Gazebo/Blue</material>

</gazebo>

PS：material 标签中，设置的值区分大小写，颜色可以设置为 Red Blue Green Black .....

### 3.URDF集成Gazebo实操

需求描述:将之前的机器人模型(xacro版)显示在 gazebo 中

实现流程: **1.编写封装惯性矩阵算法的 xacro 文件**

<robot name="base" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- Macro for inertia matrix -->

<xacro:macro name="sphere\_inertial\_matrix" params="m r">

<inertial>

<mass value="${m}" />

<inertia ixx="${2\*m\*r\*r/5}" ixy="0" ixz="0"

iyy="${2\*m\*r\*r/5}" iyz="0"

izz="${2\*m\*r\*r/5}" />

</inertial>

</xacro:macro>

<xacro:macro name="cylinder\_inertial\_matrix" params="m r h">

<inertial>

<mass value="${m}" />

<inertia ixx="${m\*(3\*r\*r+h\*h)/12}" ixy = "0" ixz = "0"

iyy="${m\*(3\*r\*r+h\*h)/12}" iyz = "0"

izz="${m\*r\*r/2}" />

</inertial>

</xacro:macro>

<xacro:macro name="Box\_inertial\_matrix" params="m l w h">

<inertial>

<mass value="${m}" />

<inertia ixx="${m\*(h\*h + l\*l)/12}" ixy = "0" ixz = "0"

iyy="${m\*(w\*w + l\*l)/12}" iyz= "0"

izz="${m\*(w\*w + h\*h)/12}" />

</inertial>

</xacro:macro>

</robot>

**2.复制相关 xacro 文件，并设置 collision inertial 以及 color 等参数**

**A.底盘 Xacro 文件**

<!--

使用 xacro 优化 URDF 版的小车底盘实现：

实现思路:

1.将一些常量、变量封装为 xacro:property

比如:PI 值、小车底盘半径、离地间距、车轮半径、宽度 ....

2.使用 宏 封装驱动轮以及支撑轮实现，调用相关宏生成驱动轮与支撑轮

-->

<!-- 根标签，必须声明 xmlns:xacro -->

<robot name="my\_base" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- 封装变量、常量 -->

<!-- PI 值设置精度需要高一些，否则后续车轮翻转量计算时，可能会出现肉眼不能察觉的车轮倾斜，从而导致模型抖动 -->

<xacro:property name="PI" value="3.1415926"/>

<!-- 宏:黑色设置 -->

<material name="black">

<color rgba="0.0 0.0 0.0 1.0" />

</material>

<!-- 底盘属性 -->

<xacro:property name="base\_footprint\_radius" value="0.001" /> <!-- base\_footprint 半径 -->

<xacro:property name="base\_link\_radius" value="0.1" /> <!-- base\_link 半径 -->

<xacro:property name="base\_link\_length" value="0.08" /> <!-- base\_link 长 -->

<xacro:property name="earth\_space" value="0.015" /> <!-- 离地间距 -->

<xacro:property name="base\_link\_m" value="0.5" /> <!-- 质量 -->

<!-- 底盘 -->

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="${base\_footprint\_radius}" />

</geometry>

</visual>

</link>

<link name="base\_link">

<visual>

<geometry>

<cylinder radius="${base\_link\_radius}" length="${base\_link\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="yellow">

<color rgba="0.5 0.3 0.0 0.5" />

</material>

</visual>

<collision>

<geometry>

<cylinder radius="${base\_link\_radius}" length="${base\_link\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

</collision>

<xacro:cylinder\_inertial\_matrix m="${base\_link\_m}" r="${base\_link\_radius}" h="${base\_link\_length}" />

</link>

<joint name="base\_link2base\_footprint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link" />

<origin xyz="0 0 ${earth\_space + base\_link\_length / 2 }" />

</joint>

<gazebo reference="base\_link">

<material>Gazebo/Yellow</material>

</gazebo>

<!-- 驱动轮 -->

<!-- 驱动轮属性 -->

<xacro:property name="wheel\_radius" value="0.0325" /><!-- 半径 -->

<xacro:property name="wheel\_length" value="0.015" /><!-- 宽度 -->

<xacro:property name="wheel\_m" value="0.05" /> <!-- 质量 -->

<!-- 驱动轮宏实现 -->

<xacro:macro name="add\_wheels" params="name flag">

<link name="${name}\_wheel">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />

<material name="black" />

</visual>

<collision>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="${PI / 2} 0.0 0.0" />

</collision>

<xacro:cylinder\_inertial\_matrix m="${wheel\_m}" r="${wheel\_radius}" h="${wheel\_length}" />

</link>

<joint name="${name}\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="${name}\_wheel" />

<origin xyz="0 ${flag \* base\_link\_radius} ${-(earth\_space + base\_link\_length / 2 - wheel\_radius) }" />

<axis xyz="0 1 0" />

</joint>

<gazebo reference="${name}\_wheel">

<material>Gazebo/Red</material>

</gazebo>

</xacro:macro>

<xacro:add\_wheels name="left" flag="1" />

<xacro:add\_wheels name="right" flag="-1" />

<!-- 支撑轮 -->

<!-- 支撑轮属性 -->

<xacro:property name="support\_wheel\_radius" value="0.0075" /> <!-- 支撑轮半径 -->

<xacro:property name="support\_wheel\_m" value="0.03" /> <!-- 质量 -->

<!-- 支撑轮宏 -->

<xacro:macro name="add\_support\_wheel" params="name flag" >

<link name="${name}\_wheel">

<visual>

<geometry>

<sphere radius="${support\_wheel\_radius}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="black" />

</visual>

<collision>

<geometry>

<sphere radius="${support\_wheel\_radius}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

</collision>

<xacro:sphere\_inertial\_matrix m="${support\_wheel\_m}" r="${support\_wheel\_radius}" />

</link>

<joint name="${name}\_wheel2base\_link" type="continuous">

<parent link="base\_link" />

<child link="${name}\_wheel" />

<origin xyz="${flag \* (base\_link\_radius - support\_wheel\_radius)} 0 ${-(base\_link\_length / 2 + earth\_space / 2)}" />

<axis xyz="1 1 1" />

</joint>

<gazebo reference="${name}\_wheel">

<material>Gazebo/Red</material>

</gazebo>

</xacro:macro>

<xacro:add\_support\_wheel name="front" flag="1" />

<xacro:add\_support\_wheel name="back" flag="-1" />

</robot>

注意: 如果机器人模型在 Gazebo 中产生了抖动，滑动，缓慢位移 .... 诸如此类情况，请查看

惯性矩阵是否设置了，且设置是否正确合理

车轮翻转需要依赖于 PI 值，如果 PI 值精度偏低，也可能导致上述情况产生

**B.摄像头 Xacro 文件**

<!-- 摄像头相关的 xacro 文件 -->

<robot name="my\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 摄像头属性 -->

<xacro:property name="camera\_length" value="0.01" /> <!-- 摄像头长度(x) -->

<xacro:property name="camera\_width" value="0.025" /> <!-- 摄像头宽度(y) -->

<xacro:property name="camera\_height" value="0.025" /> <!-- 摄像头高度(z) -->

<xacro:property name="camera\_x" value="0.08" /> <!-- 摄像头安装的x坐标 -->

<xacro:property name="camera\_y" value="0.0" /> <!-- 摄像头安装的y坐标 -->

<xacro:property name="camera\_z" value="${base\_link\_length / 2 + camera\_height / 2}" /> <!-- 摄像头安装的z坐标:底盘高度 / 2 + 摄像头高度 / 2 -->

<xacro:property name="camera\_m" value="0.01" /> <!-- 摄像头质量 -->

<!-- 摄像头关节以及link -->

<link name="camera">

<visual>

<geometry>

<box size="${camera\_length} ${camera\_width} ${camera\_height}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="black" />

</visual>

<collision>

<geometry>

<box size="${camera\_length} ${camera\_width} ${camera\_height}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

</collision>

<xacro:Box\_inertial\_matrix m="${camera\_m}" l="${camera\_length}" w="${camera\_width}" h="${camera\_height}" />

</link>

<joint name="camera2base\_link" type="fixed">

<parent link="base\_link" />

<child link="camera" />

<origin xyz="${camera\_x} ${camera\_y} ${camera\_z}" />

</joint>

<gazebo reference="camera">

<material>Gazebo/Blue</material>

</gazebo>

</robot>

**C.雷达 Xacro 文件**

<!--小车底盘添加雷达-->

<robot name="my\_laser" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 雷达支架 -->

<xacro:property name="support\_length" value="0.15" /> <!-- 支架长度 -->

<xacro:property name="support\_radius" value="0.01" /> <!-- 支架半径 -->

<xacro:property name="support\_x" value="0.0" /> <!-- 支架安装的x坐标 -->

<xacro:property name="support\_y" value="0.0" /> <!-- 支架安装的y坐标 -->

<xacro:property name="support\_z" value="${base\_link\_length / 2 + support\_length / 2}" /> <!-- 支架安装的z坐标:底盘高度 / 2 + 支架高度 / 2 -->

<xacro:property name="support\_m" value="0.02" /> <!-- 支架质量 -->

<link name="support">

<visual>

<geometry>

<cylinder radius="${support\_radius}" length="${support\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="red">

<color rgba="0.8 0.2 0.0 0.8" />

</material>

</visual>

<collision>

<geometry>

<cylinder radius="${support\_radius}" length="${support\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

</collision>

<xacro:cylinder\_inertial\_matrix m="${support\_m}" r="${support\_radius}" h="${support\_length}" />

</link>

<joint name="support2base\_link" type="fixed">

<parent link="base\_link" />

<child link="support" />

<origin xyz="${support\_x} ${support\_y} ${support\_z}" />

</joint>

<gazebo reference="support">

<material>Gazebo/White</material>

</gazebo>

<!-- 雷达属性 -->

<xacro:property name="laser\_length" value="0.05" /> <!-- 雷达长度 -->

<xacro:property name="laser\_radius" value="0.03" /> <!-- 雷达半径 -->

<xacro:property name="laser\_x" value="0.0" /> <!-- 雷达安装的x坐标 -->

<xacro:property name="laser\_y" value="0.0" /> <!-- 雷达安装的y坐标 -->

<xacro:property name="laser\_z" value="${support\_length / 2 + laser\_length / 2}" /> <!-- 雷达安装的z坐标:支架高度 / 2 + 雷达高度 / 2 -->

<xacro:property name="laser\_m" value="0.1" /> <!-- 雷达质量 -->

<!-- 雷达关节以及link -->

<link name="laser">

<visual>

<geometry>

<cylinder radius="${laser\_radius}" length="${laser\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

<material name="black" />

</visual>

<collision>

<geometry>

<cylinder radius="${laser\_radius}" length="${laser\_length}" />

</geometry>

<origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />

</collision>

<xacro:cylinder\_inertial\_matrix m="${laser\_m}" r="${laser\_radius}" h="${laser\_length}" />

</link>

<joint name="laser2support" type="fixed">

<parent link="support" />

<child link="laser" />

<origin xyz="${laser\_x} ${laser\_y} ${laser\_z}" />

</joint>

<gazebo reference="laser">

<material>Gazebo/Black</material>

</gazebo>

</robot>

**D.组合底盘、摄像头与雷达的 Xacro 文件**

<!-- 组合小车底盘与摄像头 -->

<robot name="my\_car\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="my\_head.urdf.xacro" />

<xacro:include filename="my\_base.urdf.xacro" />

<xacro:include filename="my\_camera.urdf.xacro" />

<xacro:include filename="my\_laser.urdf.xacro" />

</robot>

**3.在 gazebo 中执行**

<launch>

<!-- 将 Urdf 文件的内容加载到参数服务器 -->

<param name="robot\_description" command="$(find xacro)/xacro $(find demo\_gazebo)/urdf/xacro/my\_base\_camera\_laser.urdf.xacro" />

<!-- 启动 gazebo -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch" />

<!-- 在 gazebo 中显示机器人模型 -->

<node pkg="gazebo\_ros" type="spawn\_model" name="model" args="-urdf -model mycar -param robot\_description" />

</launch>

### 4.Gazebo仿真环境搭建

Gazebo 中创建仿真实现方式有两种:

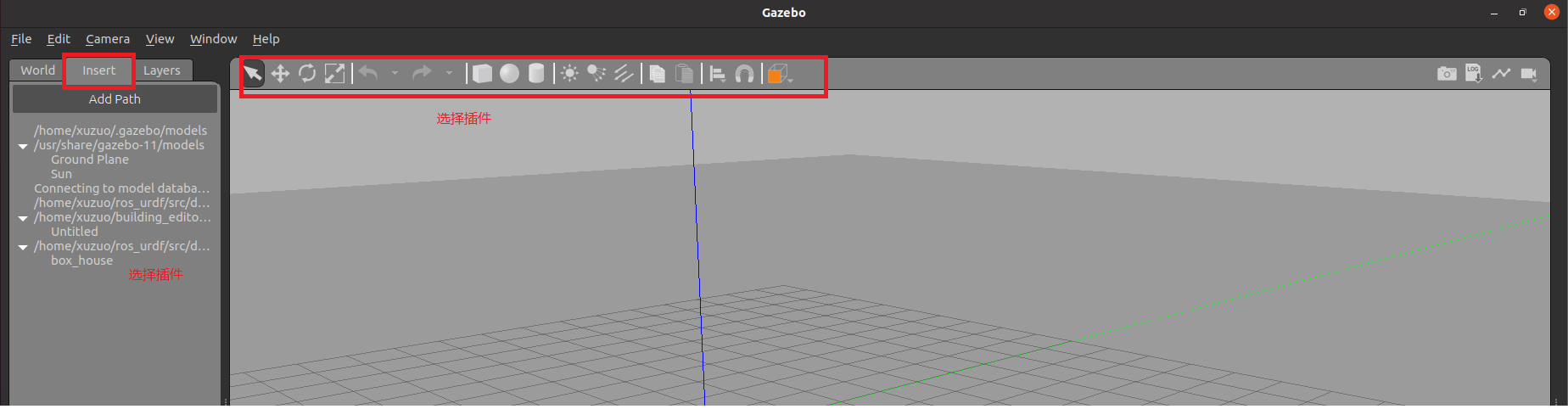
方式1: 直接添加内置组件创建仿真环境

方式2: 手动绘制仿真环境(更为灵活)

也还可以直接下载使用官方或第三方提高的仿真环境插件。

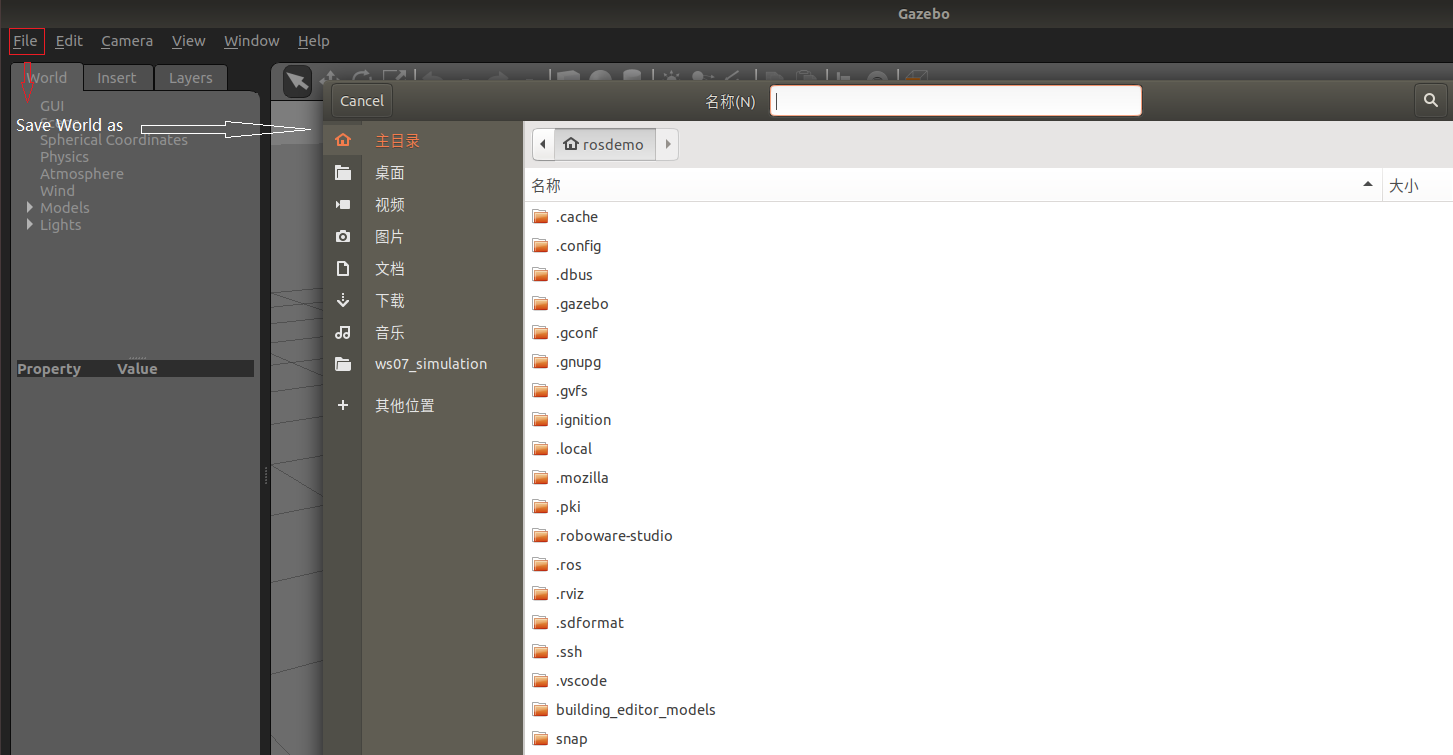
**1.添加内置组件创建仿真环境**

1.1启动 Gazebo 并添加组件



1.2保存仿真环境

添加完毕后，选择 file ---> Save World as 选择保存路径(功能包下: worlds 目录)，文件名自定义，后缀名设置为 .world



* 1. 启动

<launch>

<!-- 将 Urdf 文件的内容加载到参数服务器 -->

<param name="robot\_description" command="$(find xacro)/xacro $(find demo\_gazebo)/xacro/my\_base\_camera\_laser.urdf.xacro" />

<!-- 启动 gazebo -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="world\_name" value="$(find demo\_gazebo)/worlds/hello.world" />

</include>

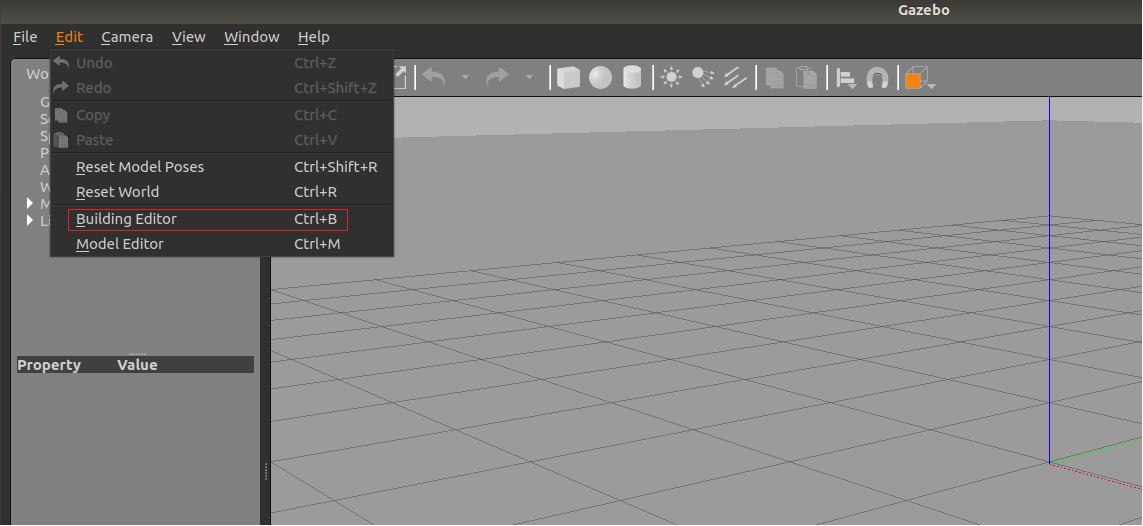
<!-- 在 gazebo 中显示机器人模型 -->

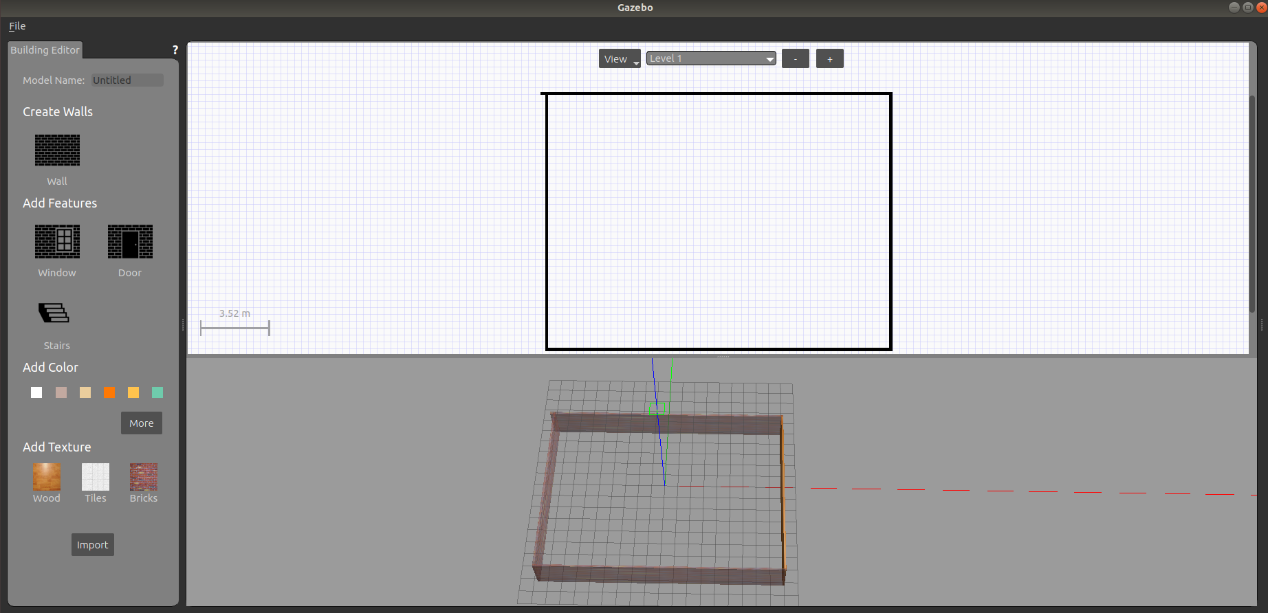
<node pkg="gazebo\_ros" type="spawn\_model" name="model" args="-urdf -model mycar -param robot\_description" />

</launch>

**2.自定义仿真环境**

2.1 启动 gazebo 打开构建面板，绘制仿真环境





2.2 保存构建的环境

点击: 左上角 file ---> Save (保存路径功能包下的: models)

然后 file ---> Exit Building Editor

2.3 保存为 world 文件

可以像方式1一样再添加一些插件，然后保存为 world 文件(保存路径功能包下的: worlds)

2.4 启动

同方式1

**3.使用官方提供的插件**

当前 Gazebo 提供的仿真道具有限，还可以下载官方支持，可以提供更为丰富的仿真实现，具体实现如下:

3.1 下载官方模型库

git clone https://github.com/osrf/gazebo\_models

3.2 将模型库复制进 gazebo

将得到的gazebo\_models文件夹内容复制到 /usr/share/gazebo-\*/models

3.3 应用

重启 Gazebo，选择左侧菜单栏的 insert 可以选择并插入相关道具了

## URDF、Gazebo与Rviz综合应用

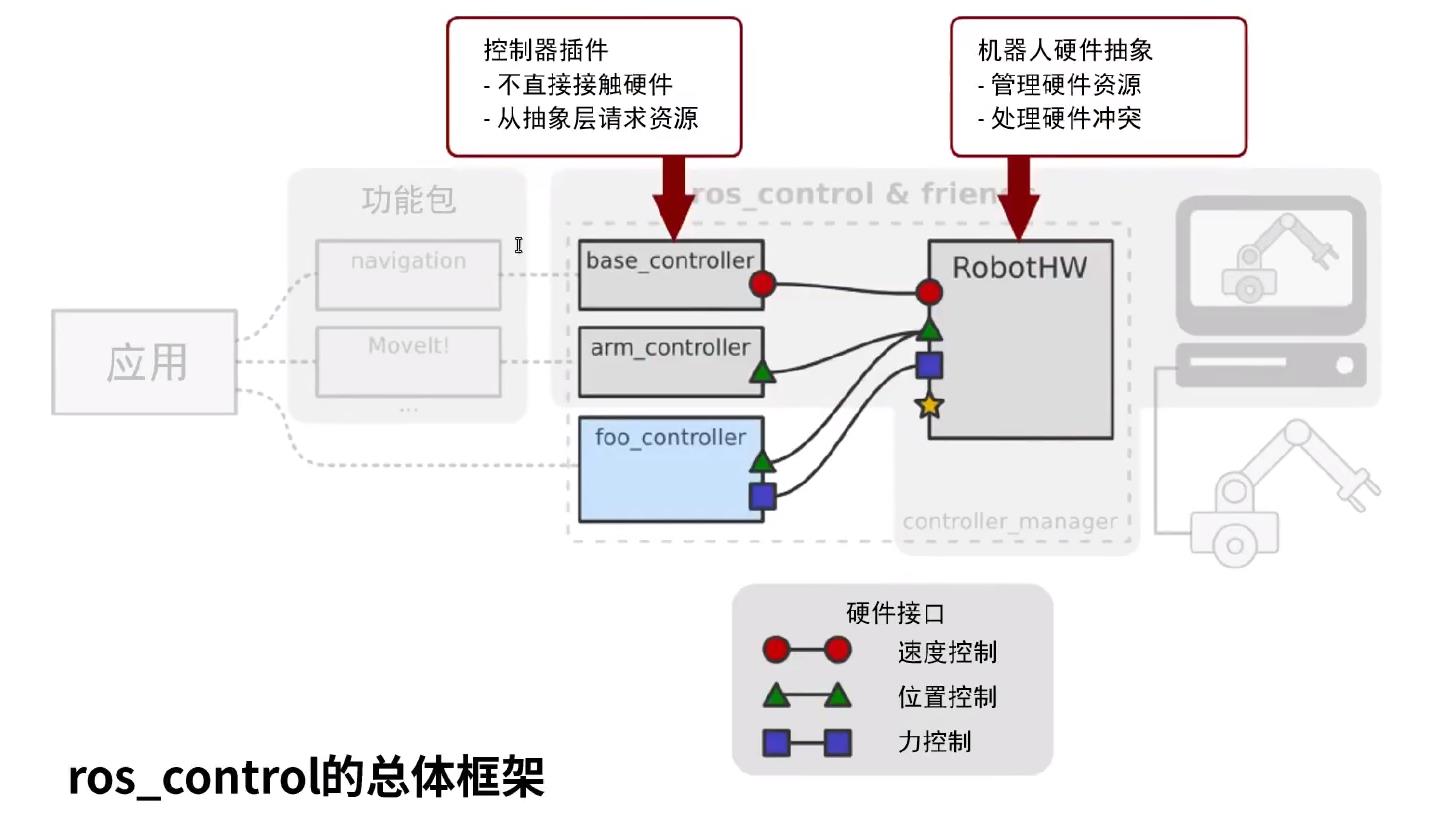
### 1.机器人运动控制以及里程计信息显示

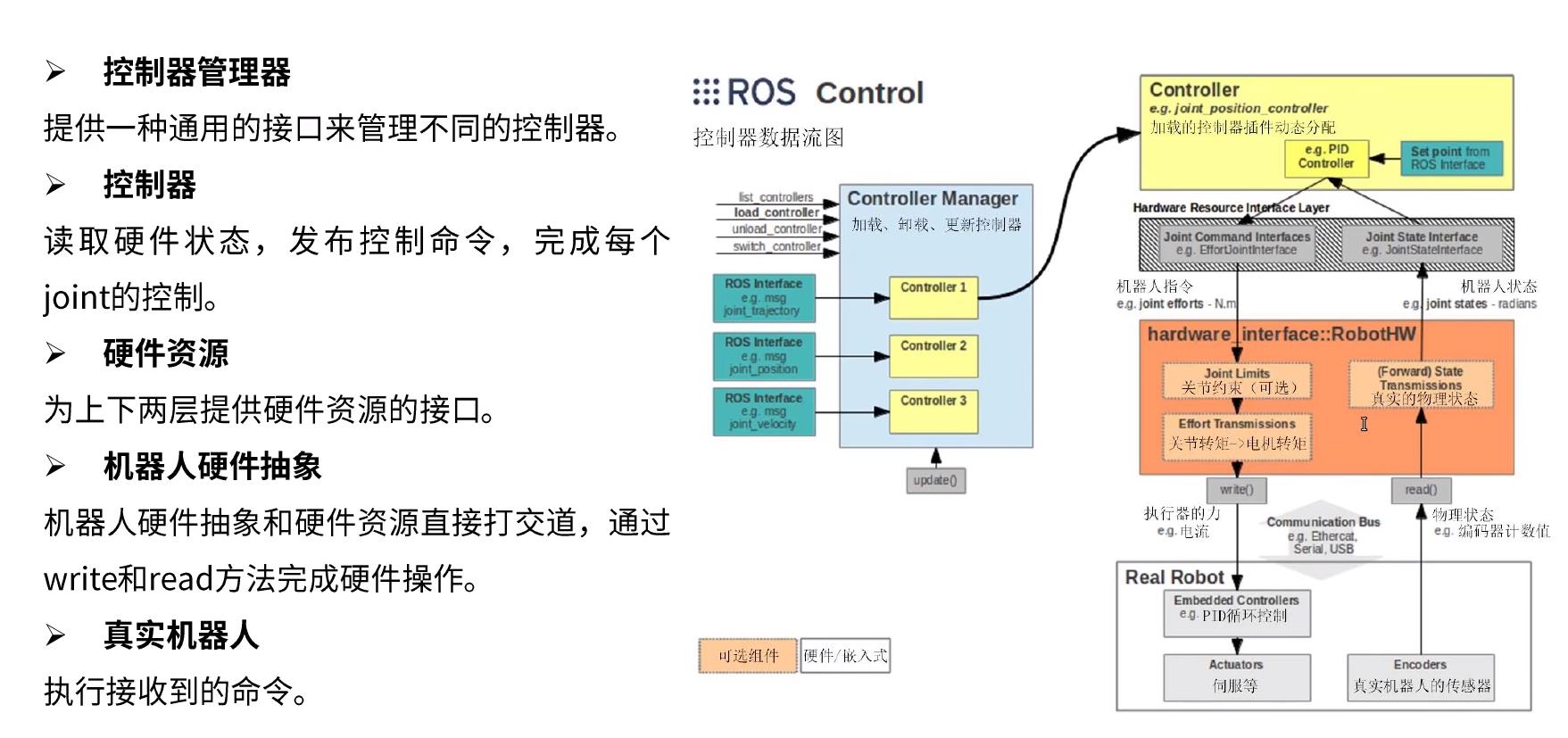
**1.ros\_control 简介**

场景:同一套 ROS 程序，如何部署在不同的机器人系统上，比如：开发阶段为了提高效率是在仿真平台上测试的，部署时又有不同的实体机器人平台，不同平台的实现是有差异的，如何保证 ROS 程序的可移植性？ROS 内置的解决方式是 ros\_control。

ros\_control:是一组软件包，它包含了控制器接口，控制器管理器，传输和硬件接口。ros\_control 是一套机器人控制的中间件，是一套规范，不同的机器人平台只要按照这套规范实现，那么就可以保证 与ROS 程序兼容，通过这套规范，实现了一种可插拔的架构设计，大大提高了程序设计的效率与灵活性。

gazebo 已经实现了 ros\_control 的相关接口，如果需要在 gazebo 中控制机器人运动，直接调用相关接口即可





**2.运动控制实现流程(Gazebo)**

已经创建完毕的机器人模型，编写一个单独的 xacro 文件，为机器人模型添加传动装置以及控制器

**2.1 为 joint 添加传动装置以及控制器**

两轮差速配置

<robot name="my\_car\_move" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 传动实现:用于连接控制器与关节 -->

<xacro:macro name="joint\_trans" params="joint\_name">

<!-- Transmission is important to link the joints and the controller -->

<transmission name="${joint\_name}\_trans">

<type>transmission\_interface/SimpleTransmission</type>

<joint name="${joint\_name}">

<hardwareInterface>hardware\_interface/VelocityJointInterface</hardwareInterface>

</joint>

<actuator name="${joint\_name}\_motor">

<hardwareInterface>hardware\_interface/VelocityJointInterface</hardwareInterface>

<mechanicalReduction>1</mechanicalReduction>

</actuator>

</transmission>

</xacro:macro>

<!-- 每一个驱动轮都需要配置传动装置 -->

<xacro:joint\_trans joint\_name="left\_wheel2base\_link" />

<xacro:joint\_trans joint\_name="right\_wheel2base\_link" />

<!-- 控制器 -->

<gazebo>

<plugin name="differential\_drive\_controller" filename="libgazebo\_ros\_diff\_drive.so">

<rosDebugLevel>Debug</rosDebugLevel>

<publishWheelTF>true</publishWheelTF>

<robotNamespace>/</robotNamespace><!—机器人命名空间 -->

<publishTf>1</publishTf>

<publishWheelJointState>true</publishWheelJointState>

<alwaysOn>true</alwaysOn>

<updateRate>100.0</updateRate>

<legacyMode>true</legacyMode>

<leftJoint>left\_wheel2base\_link</leftJoint> <!-- 左轮 -->

<rightJoint>right\_wheel2base\_link</rightJoint> <!-- 右轮 -->

<wheelSeparation>${base\_link\_radius \* 2}</wheelSeparation> <!-- 车轮间距 -->

<wheelDiameter>${wheel\_radius \* 2}</wheelDiameter> <!-- 车轮直径 -->

<broadcastTF>1</broadcastTF>

<wheelTorque>30</wheelTorque>

<wheelAcceleration>1.8</wheelAcceleration>

<commandTopic>cmd\_vel</commandTopic> <!-- 运动控制话题 -->

<odometryFrame>odom</odometryFrame>

<odometryTopic>odom</odometryTopic> <!-- 里程计话题 -->

<robotBaseFrame>base\_footprint</robotBaseFrame> <!-- 根坐标系 -->

</plugin>

</gazebo>

</robot>

**2.2 xacro文件集成**

<!-- 组合小车底盘与摄像头 -->

<robot name="my\_car\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="my\_head.urdf.xacro" />

<xacro:include filename="my\_base.urdf.xacro" />

<xacro:include filename="my\_camera.urdf.xacro" />

<xacro:include filename="my\_laser.urdf.xacro" />

<xacro:include filename="move.urdf.xacro" />

</robot>

**2.3 启动 gazebo并控制机器人运动**

<launch>

<!-- 将 Urdf 文件的内容加载到参数服务器 -->

<param name="robot\_description" command="$(find xacro)/xacro $(find demo\_gazebo)/urdf/xacro/my\_base\_camera\_laser.urdf.xacro" />

<!-- 启动 gazebo -->

<include file="$(find demo\_gazebo)/launch/empty\_world.launch">

<arg name="world\_name" value="$(find demo\_gazebo)/worlds/hello.world" />

</include>

<!-- 在 gazebo 中显示机器人模型 -->

<node pkg="gazebo\_ros" type="spawn\_model" name="model" args="-urdf -model mycar -param robot\_description" />

</launch>

启动 launch 文件，使用 topic list 查看话题列表，会发现多了 /cmd\_vel 然后发布 vmd\_vel 消息控制即可,使用命令控制(或者可以编写单独的节点控制)

rostopic pub -r 10 /cmd\_vel geometry\_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'

**3.Rviz查看里程计信息**

在 Gazebo 的仿真环境中，机器人的里程计信息以及运动朝向等信息是无法获取的，可以通过 Rviz 显示机器人的里程计信息以及运动朝向

里程计: 机器人相对出发点坐标系的位姿状态(X 坐标 Y 坐标 Z坐标以及朝向)。

**3.1启动 Rviz**

<launch>

<!-- 启动 rviz -->

<node pkg="rviz" type="rviz" name="rviz" />

<!-- 关节以及机器人状态发布节点 -->

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher" />

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="robot\_state\_publisher" />

</launch>

**3.2 添加组件**

执行 launch 文件后，在 Rviz 中添加图示组件:



### 2.雷达信息仿真以及显示

**1.Gazebo 仿真雷达**

**1.1 新建 Xacro 文件，配置雷达传感器信息**

<robot name="my\_sensors" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 雷达 -->

<gazebo reference="laser">

<sensor type="ray" name="rplidar">

<pose>0 0 0 0 0 0</pose>

<visualize>true</visualize>

<update\_rate>5.5</update\_rate>

<ray>

<scan>

<horizontal>

<samples>360</samples> <!--采样范围 -->

<resolution>1</resolution><!--分辨率：多少度一个采样点 -->

<min\_angle>-3</min\_angle><!--最小角度（在左边） -->

<max\_angle>3</max\_angle><!--最大角度（在右边） -->

</horizontal>

</scan>

<range>

<min>0.10</min>

<max>30.0</max>

<resolution>0.01</resolution>

</range>

<noise><!--噪声 -->

<type>gaussian</type>

<mean>0.0</mean>

<stddev>0.01</stddev>

</noise>

</ray>

<plugin name="gazebo\_rplidar" filename="libgazebo\_ros\_laser.so">

<topicName>/scan</topicName>

<frameName>laser</frameName>

</plugin>

</sensor>

</gazebo>

</robot>

**1.2 xacro 文件集成**

<!-- 组合小车底盘与传感器 -->

<robot name="my\_car\_camera" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="my\_head.urdf.xacro" />

<xacro:include filename="my\_base.urdf.xacro" />

<xacro:include filename="my\_camera.urdf.xacro" />

<xacro:include filename="my\_laser.urdf.xacro" />

<xacro:include filename="move.urdf.xacro" />

<!-- 雷达仿真的 xacro 文件 -->

<xacro:include filename="my\_sensors\_laser.urdf.xacro" />

</robot>

**1.3启动仿真环境**

编写launch文件，启动gazebo，此处略...

**2.Rviz 显示雷达数据**

先启动 rviz,添加雷达信息显示插件(LaserScan:laser)

### 3.摄像头信息仿真以及显示

**1.Gazebo 仿真摄像头**

**1.1 新建 Xacro 文件，配置摄像头传感器信息**

<robot name="my\_sensors" xmlns:xacro="http://wiki.ros.org/xacro">

<!-- 被引用的link -->

<gazebo reference="camera">

<!-- 类型设置为 camara -->

<sensor type="camera" name="camera\_node"><!—senser标签描述传感器 -->

<update\_rate>30.0</update\_rate> <!-- 更新频率 -->

<!-- 摄像头基本信息设置 -->

<camera name="head">

<horizontal\_fov>1.3962634</horizontal\_fov>

<image>

<width>1280</width>

<height>720</height>

<format>R8G8B8</format>

</image>

<clip>

<near>0.02</near>

<far>300</far>

</clip>

<noise>

<type>gaussian</type>

<mean>0.0</mean>

<stddev>0.007</stddev>

</noise>

</camera>

<!-- 核心插件（加载摄像头仿真插件） -->

<plugin name="gazebo\_camera" filename="libgazebo\_ros\_camera.so">

<alwaysOn>true</alwaysOn>

<updateRate>0.0</updateRate>

<cameraName>/camera</cameraName>

<imageTopicName>image\_raw</imageTopicName>

<cameraInfoTopicName>camera\_info</cameraInfoTopicName>

<frameName>camera</frameName>

<hackBaseline>0.07</hackBaseline>

<distortionK1>0.0</distortionK1>

<distortionK2>0.0</distortionK2>

<distortionK3>0.0</distortionK3>

<distortionT1>0.0</distortionT1>

<distortionT2>0.0</distortionT2>

</plugin>

</sensor>

</gazebo>

</robot>

**1.2 xacro 文件集成**

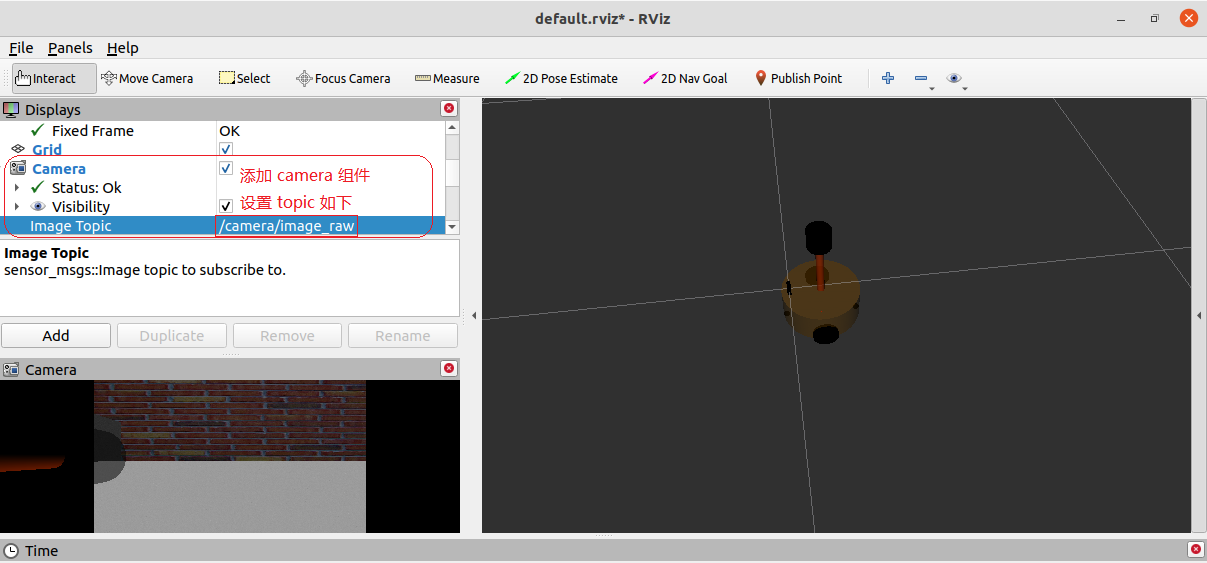
将步骤1的 Xacro 文件集成进总的机器人模型文件

**1.3启动仿真环境**

编写launch文件，启动gazebo，此处略...

**2.Rviz 显示摄像头数据**

执行 gazebo 并启动 Rviz,在 Rviz 中添加摄像头组件。



### 4.kinect信息仿真以及显示

**1.Gazebo仿真Kinect**

**1.1 新建 Xacro 文件，配置 kinetic传感器信息**

<robot name="my\_sensors" xmlns:xacro="http://wiki.ros.org/xacro">

<gazebo reference="kinect link名称">

<sensor type="depth" name="camera">

<always\_on>true</always\_on>

<update\_rate>20.0</update\_rate>

<camera>

<horizontal\_fov>${60.0\*PI/180.0}</horizontal\_fov>

<image>

<format>R8G8B8</format>

<width>640</width>

<height>480</height>

</image>

<clip>

<near>0.05</near>

<far>8.0</far>

</clip>

</camera>

<plugin name="kinect\_camera\_controller" filename="libgazebo\_ros\_openni\_kinect.so">

<cameraName>camera</cameraName>

<alwaysOn>true</alwaysOn>

<updateRate>10</updateRate>

<imageTopicName>rgb/image\_raw</imageTopicName>

<depthImageTopicName>depth/image\_raw</depthImageTopicName>

<pointCloudTopicName>depth/points</pointCloudTopicName>

<cameraInfoTopicName>rgb/camera\_info</cameraInfoTopicName>

<depthImageCameraInfoTopicName>depth/camera\_info</depthImageCameraInfoTopicName>

<frameName>kinect link名称</frameName><!—kinect的点云数据link名称，正常为kinect的link名称，例如上面的support，用kinect显示点云数据时改为support\_depth -->

<baseline>0.1</baseline>

<distortion\_k1>0.0</distortion\_k1>

<distortion\_k2>0.0</distortion\_k2>

<distortion\_k3>0.0</distortion\_k3>

<distortion\_t1>0.0</distortion\_t1>

<distortion\_t2>0.0</distortion\_t2>

<pointCloudCutoff>0.4</pointCloudCutoff>

</plugin>

</sensor>

</gazebo>

</robot>

**1.2 xacro 文件集成**

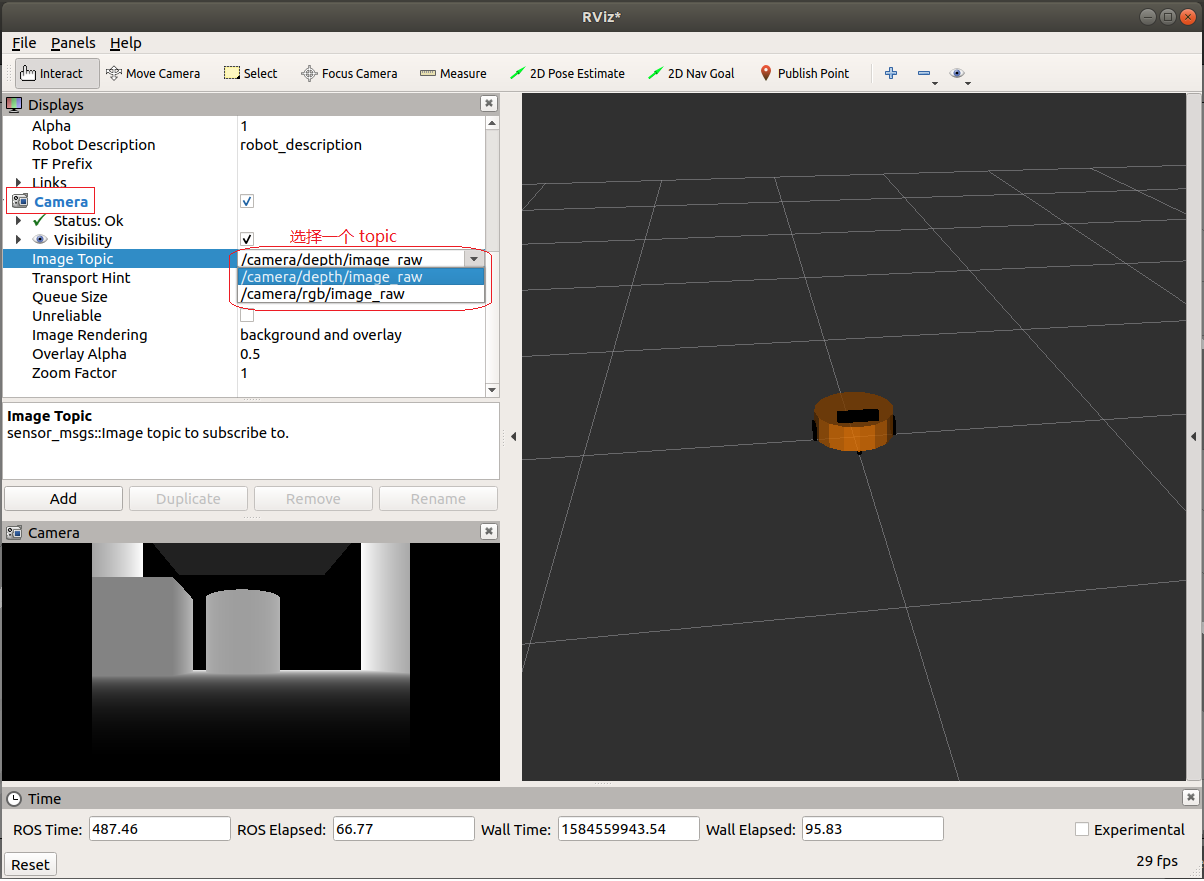
将步骤1的 Xacro 文件集成进总的机器人模型文件

**1.3启动仿真环境**

编写launch文件，启动gazebo，此处略...

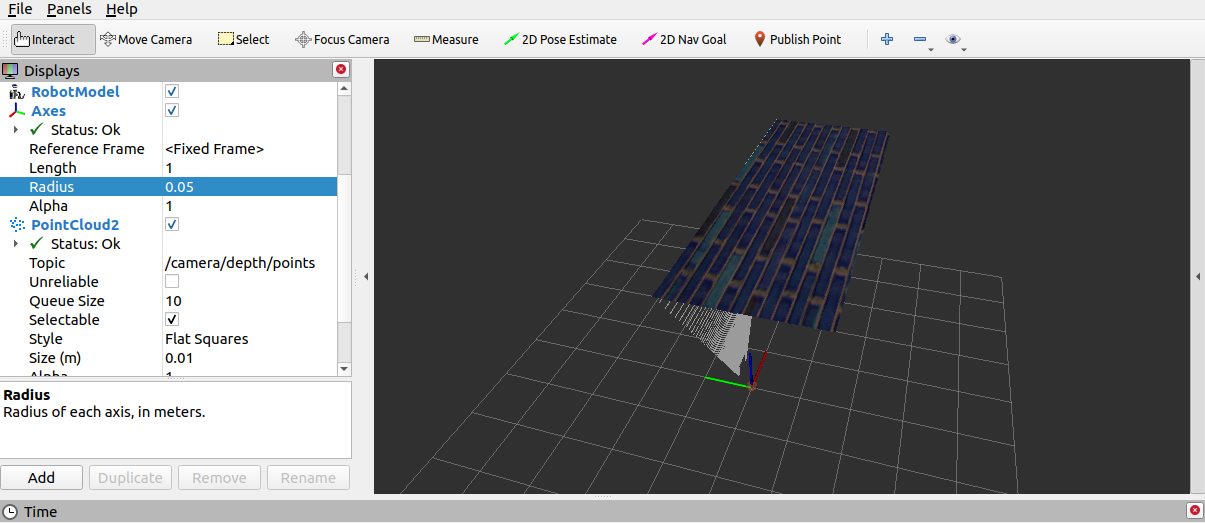
**2 Rviz 显示 Kinect 数据**

启动 rviz,添加摄像头组件查看数据



**补充:kinect 点云数据显示**

在kinect中也可以以点云的方式显示感知周围环境，在 rviz 中操作如下:



问题:在rviz中显示时错位。

原因:在kinect中图像数据与点云数据使用了两套坐标系统，且两套坐标系统位姿并不一致。

解决:1.在插件中为kinect设置坐标系，修改配置文件的<frameName>标签内容：

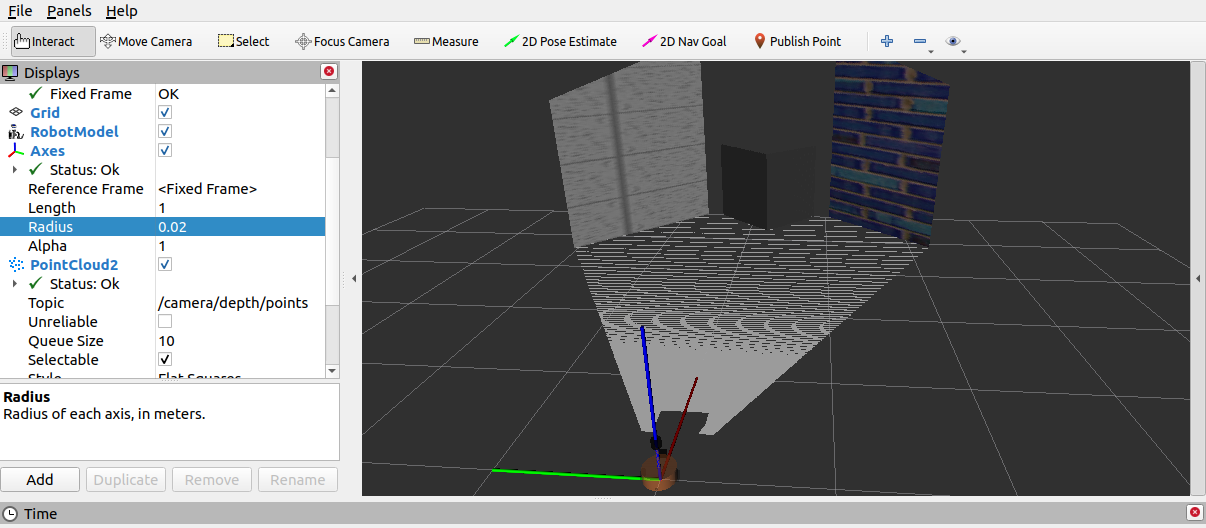
<frameName>support\_depth</frameName>

1. 发布新设置的坐标系到kinect连杆的坐标变换关系，在启动rviz的launch中，添加:

(kinect坐标系欧拉角旋转顺序：z，y，x)

<node pkg="tf2\_ros" type="static\_transform\_publisher" name="static\_transform\_publisher" args="0 0 0 -1.57 0 -1.57 /support /support\_depth" />

1. 启动rviz，重新显示。

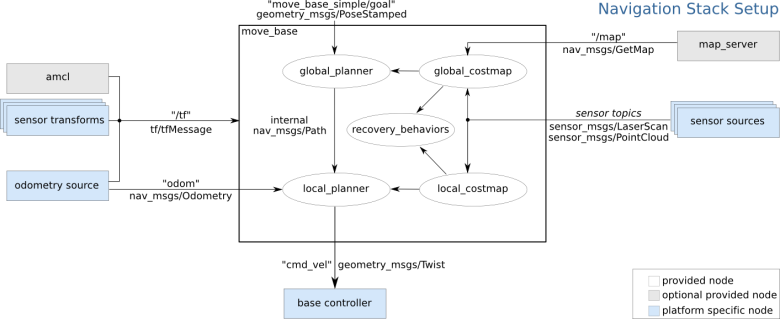


# 机器人导航(仿真)

## 概述

### 1.导航模块简介

机器人是如何实现导航的呢？或换言之，机器人是如何从 A 点移动到 B 点呢？ROS 官方为了提供了一张导航功能包集的图示,该图中囊括了 ROS 导航的一些关键技术:



假定我们已经以特定方式配置机器人，导航功能包集将使其可以运动。上图概述了这种配置方式。白色的部分是必须且已实现的组件，灰色的部分是可选且已实现的组件，蓝色的部分是必须为每一个机器人平台创建的组件。

总结下来，涉及的关键技术有如下五点:

**1.全局地图**

在现实生活中，当我们需要实现导航时，可能会首先参考一张全局性质的地图，然后根据地图来确定自身的位置、目的地位置，并且也会根据地图显示来规划一条大致的路线.... 对于机器人导航而言，也是如此，在机器人导航中地图是一个重要的组成元素，当然如果要使用地图，首先需要绘制地图。关于地图建模技术不断涌现，这其中有一门称之为 SLAM 的理论脱颖而出:

SLAM(simultaneous localization and mapping),也称为CML (Concurrent Mapping and Localization), 即时定位与地图构建，或并发建图与定位。SLAM问题可以描述为: 机器人在未知环境中从一个未知位置开始移动,在移动过程中根据位置估计和地图进行自身定位，同时在自身定位的基础上建造增量式地图，以绘制出外部环境的完全地图。

在 ROS 中，较为常用的 SLAM 实现也比较多，比如: gmapping、hector\_slam、cartographer、rgbdslam、ORB\_SLAM ....

当然如果要完成 SLAM ，机器人必须要具备感知外界环境的能力，尤其是要具备获取周围环境深度信息的能力。感知的实现需要依赖于传感器，比如: 激光雷达、摄像头、RGB-D摄像头...

SLAM 可以用于地图生成，而生成的地图还需要被保存以待后续使用，在 ROS 中保存地图的功能包是 map\_server

另外注意: SLAM 虽然是机器人导航的重要技术之一，但是 二者并不等价，确切的讲，SLAM 只是实现地图构建和即时定位。

**2.自身定位**

导航伊始和导航过程中，机器人都需要确定当前自身的位置，如果在室外，那么 GPS 是一个不错的选择，而如果室内、隧道、地下或一些特殊的屏蔽 GPS 信号的区域，由于 GPS 信号弱化甚至完全不可用，那么就必须另辟蹊径了，比如前面的 SLAM 就可以实现自身定位，除此之外，ROS 中还提供了一个用于定位的功能包: amcl

amcl(adaptiveMonteCarloLocalization)自适应的蒙特卡洛定位,是用于2D移动机器人的概率定位系统。它实现了自适应（或KLD采样）蒙特卡洛定位方法，该方法使用粒子过滤器根据已知地图跟踪机器人的姿态。

**3.路径规划**

导航就是机器人从A点运动至B点的过程，在这一过程中，机器人需要根据目标位置计算全局运动路线，并且在运动过程中，还需要时时根据出现的一些动态障碍物调整运动路线，直至到达目标点，该过程就称之为路径规划。在 ROS 中提供了 move\_base 包来实现路径规则,该功能包主要由两大规划器组成:

**全局路径规划(gloable\_planner)**

根据给定的目标点和全局地图实现总体的路径规划，使用 Dijkstra 或 A\* 算法进行全局路径规划，计算最优路线，作为全局路线

**本地时时规划(local\_planner)**

在实际导航过程中，机器人可能无法按照给定的全局最优路线运行，比如:机器人在运行中，可能会随时出现一定的障碍物... 本地规划的作用就是使用一定算法(Dynamic Window Approaches) 来实现障碍物的规避，并选取当前最优路径以尽量符合全局最优路径

全局路径规划与本地路径规划是相对的，全局路径规划侧重于全局、宏观实现，而本地路径规划侧重与当前、微观实现。

**4.运动控制**

导航功能包集假定它可以通过话题"cmd\_vel"发布geometry\_msgs/Twist类型的消息，这个消息基于机器人的基座坐标系，它传递的是运动命令。这意味着必须有一个节点订阅"cmd\_vel"话题， 将该话题上的速度命令转换为电机命令并发送。

**5.环境感知**

感知周围环境信息，比如: 摄像头、激光雷达、编码器...，摄像头、激光雷达可以用于感知外界环境的深度信息，编码器可以感知电机的转速信息，进而可以获取速度信息并生成里程计信息。

在导航功能包集中，环境感知也是一重要模块实现，它为其他模块提供了支持。其他模块诸如: SLAM、amcl、move\_base 都需要依赖于环境感知。

### 2.导航之坐标系

**1.简介**

定位是导航中的重要实现之一，所谓定位，就是参考某个坐标系(比如:以机器人的出发点为原点创建坐标系)在该坐标系中标注机器人。定位原理看似简单，但是这个这个坐标系不是客观存在的，我们也无法以上帝视角确定机器人的位姿，定位实现需要依赖于机器人自身，机器人需要逆向推导参考系原点并计算坐标系相对关系，该过程实现常用方式有两种:

（1）通过里程计定位:时时收集机器人的速度信息计算并发布机器人坐标系与父级参考系的相对关系。

（2）通过传感器定位:通过传感器收集外界环境信息通过匹配计算并发布机器人坐标系与父级参考系的相对关系。

**2.特点**

两种定位方式都有各自的优缺点。

里程计定位:

优点:里程计定位信息是连续的，没有离散的跳跃。

缺点:里程计存在累计误差，不利于长距离或长期定位。

传感器定位:

优点:比里程计定位更精准；

缺点:传感器定位会出现跳变的情况，且传感器定位在标志物较少的环境下，其定位精度会大打折扣。

两种定位方式优缺点互补，应用时一般二者结合使用。

**3.坐标系变换**

上述两种定位实现中，机器人坐标系一般使用机器人模型中的根坐标系(base\_link 或 base\_footprint)，里程计定位时，父级坐标系一般称之为 odom，如果通过传感器定位，父级参考系一般称之为 map。当二者结合使用时，map 和 odom 都是机器人模型根坐标系的父级，这是不符合坐标变换中"单继承"的原则的，所以，一般会将转换关系设置为: map -> doom -> base\_link 或 base\_footprint。

## 导航实现

准备工作

请先安装相关的ROS功能包:

安装 gmapping 包(用于构建地图):sudo apt install ros-<ROS版本>-gmapping

安装地图服务包(用于保存与读取地图):sudo apt install ros-<ROS版本>-map-server

安装 navigation 包(用于定位以及路径规划):sudo apt install ros-<ROS版本>-navigation

新建功能包，并导入依赖: gmapping map\_server amcl move\_base

### 1.SLAM建图

**1.gmapping简介**

gmapping 是ROS开源社区中较为常用且比较成熟的SLAM算法之一，gmapping可以根据移动机器人里程计数据和激光雷达数据来绘制二维的栅格地图，对应的，gmapping对硬件也有一定的要求:

该移动机器人可以发布里程计消息

机器人需要发布雷达消息(该消息可以通过水平固定安装的雷达发布，或者也可以将深度相机消息转换成雷达消息)

关于里程计与雷达数据，仿真环境中可以正常获取的

**2.gmapping节点说明**

gmapping 功能包中的核心节点是:slam\_gmapping。为了方便调用，需要先了解该节点订阅的话题、发布的话题、服务以及相关参数。

**2.1订阅的Topic**

tf (tf/tfMessage)

用于雷达、底盘与里程计之间的坐标变换消息。

scan(sensor\_msgs/LaserScan)

SLAM所需的雷达信息。

**2.2发布的Topic**

map\_metadata(nav\_msgs/MapMetaData)

地图元数据，包括地图的宽度、高度、分辨率等，该消息会固定更新。

map(nav\_msgs/OccupancyGrid)

地图栅格数据，一般会在rviz中以图形化的方式显示。

~entropy(std\_msgs/Float64)

机器人姿态分布熵估计(值越大，不确定性越大)。

**2.3服务**

dynamic\_map(nav\_msgs/GetMap)

用于获取地图数据。

**2.4参数**

~base\_frame(string, default:"base\_link")

机器人基坐标系。

~map\_frame(string, default:"map")

地图坐标系。

~odom\_frame(string, default:"odom")

里程计坐标系。

~map\_update\_interval(float, default: 5.0)

地图更新频率，根据指定的值设计更新间隔。

~maxUrange(float, default: 80.0)

激光探测的最大可用范围(超出此阈值，被截断)。

~maxRange(float)

激光探测的最大范围。

.... 参数较多，上述是几个较为常用的参数，其他参数介绍可参考官网。

**2.5所需的坐标变换**

雷达坐标系→基坐标系

一般由 robot\_state\_publisher 或 static\_transform\_publisher 发布。

基坐标系→里程计坐标系

一般由里程计节点发布。

**2.6发布的坐标变换**

地图坐标系→里程计坐标系

地图到里程计坐标系之间的变换。

**3.gmapping使用**

3.1编写gmapping节点相关launch文件

launch文件编写可以参考 github 的演示 launch文件：<https://github.com/ros-perception/slam_gmapping/blob/melodic-devel/gmapping/launch/slam_gmapping_pr2.launch>

<launch>

<param name="use\_sim\_time" value="true"/>仿真

<node pkg="gmapping" type="slam\_gmapping" name="slam\_gmapping" output="screen">

<remap from="scan" to="scan"/><!-- 雷达话题 -->

<param name="base\_frame" value="base\_footprint"/><!--底盘坐标系-->

<param name="odom\_frame" value="odom"/> <!--里程计坐标系-->

<param name="map\_update\_interval" value="5.0"/>

<param name="maxUrange" value="16.0"/>

<param name="sigma" value="0.05"/>

<param name="kernelSize" value="1"/>

<param name="lstep" value="0.05"/>

<param name="astep" value="0.05"/>

<param name="iterations" value="5"/>

<param name="lsigma" value="0.075"/>

<param name="ogain" value="3.0"/>

<param name="lskip" value="0"/>

<param name="srr" value="0.1"/>

<param name="srt" value="0.2"/>

<param name="str" value="0.1"/>

<param name="stt" value="0.2"/>

<param name="linearUpdate" value="1.0"/>

<param name="angularUpdate" value="0.5"/>

<param name="temporalUpdate" value="3.0"/>

<param name="resampleThreshold" value="0.5"/>

<param name="particles" value="30"/>

<param name="xmin" value="-50.0"/>

<param name="ymin" value="-50.0"/>

<param name="xmax" value="50.0"/>

<param name="ymax" value="50.0"/>

<param name="delta" value="0.05"/>

<param name="llsamplerange" value="0.01"/>

<param name="llsamplestep" value="0.01"/>

<param name="lasamplerange" value="0.005"/>

<param name="lasamplestep" value="0.005"/>

</node>

<node pkg="joint\_state\_publisher" name="joint\_state\_publisher" type="joint\_state\_publisher" />

<node pkg="robot\_state\_publisher" name="robot\_state\_publisher" type="robot\_state\_publisher" />

<node pkg="rviz" type="rviz" name="rviz" />

<!-- 可以保存 rviz 配置并后期直接使用-->

<!- -<node pkg="rviz" type="rviz" name="rviz" args="-d $(find demo\_nav)/rviz/slam.rviz"/> -->

</launch>

3.2执行

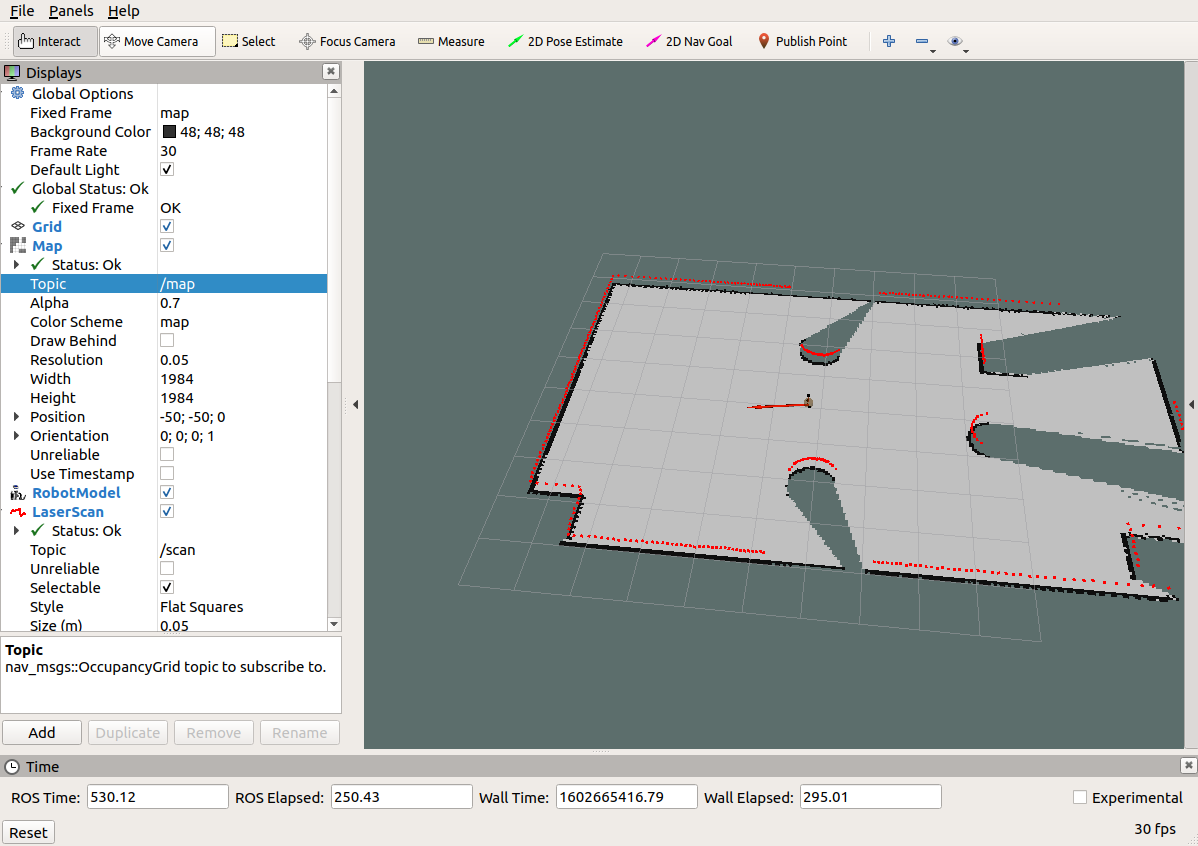
1.先启动 Gazebo 仿真环境

2.然后再启动地图绘制的 launch 文件

3.启动键盘键盘控制节点，用于控制机器人运动建图

rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

4.在 rviz 中添加组件，显示栅格地图



### 2.地图服务

**1.map\_server简介**

map\_server功能包中提供了两个节点: map\_saver 和 map\_server，前者用于将栅格地图保存到磁盘，后者读取磁盘的栅格地图并以服务的方式提供出去。

map\_server安装前面也有介绍，命令如下:

sudo apt install ros-<ROS版本>-map-server

**2.map\_server使用之地图保存节点(map\_saver)**

2.1map\_saver节点说明

订阅的topic:

map(nav\_msgs/OccupancyGrid)

订阅此话题用于生成地图文件。

2.2地图保存launch文件

<launch>

<arg name="filename" value="$(find demo\_nav)/map/map" />arg必须在上面

<node name="map\_saver" pkg="map\_server" type="map\_saver" args="-f $(arg filename)" />

</launch>

其中$(find demo\_nav)/map/map"是指地图的保存路径以及保存的文件名称。

SLAM建图完毕后，执行该launch文件即可。

xxx.pgm 本质是一张图片，直接使用图片查看程序即可打开。

xxx.yaml 保存的是地图的元数据信息，用于描述图片，内容格式如下:

image: /home/rosmelodic/ws02\_nav/src/mycar\_nav/map/nav.pgm

resolution: 0.050000

origin: [-50.000000, -50.000000, 0.000000]

negate: 0

occupied\_thresh: 0.65

free\_thresh: 0.196

解释:

image:被描述的图片资源路径，可以是绝对路径也可以是相对路径。

resolution: 图片分片率(单位: m/像素)。

origin: 地图中右下像素的二维姿势，为（x，y，偏航），偏航为逆时针旋转（偏航= 0表示无旋转）。（相对于rviz中原点的位姿）

occupied\_thresh: 占用概率大于此阈值的像素被视为完全占用。

free\_thresh: 占用率小于此阈值的像素被视为完全空闲。

negate: 是否应该颠倒白色/黑色自由/占用的语义。

map\_server 中障碍物计算规则:

地图中的每一个像素取值在 [0,255] 之间，白色为 255，黑色为 0，该值设为 x；

map\_server 会将像素值作为判断是否是障碍物的依据，首先计算比例: p = (255 - x) / 255.0，白色为0，黑色为1(negate为true，则p = x / 255.0)；

根据步骤2计算的比例判断是否是障碍物，如果 p > occupied\_thresh 那么视为障碍物，如果 p < free\_thresh 那么视为无物。

备注:图片也可以根据需求编辑。

白色：可通行区域；黑色：障碍物；蓝灰：未知区域

**3.map\_server使用之地图服务(map\_server)**

**3.1map\_server节点说明**

发布的话题：

map\_metadata（nav\_msgs / MapMetaData）

发布地图元数据。

map（nav\_msgs / OccupancyGrid）

地图数据。

服务：

static\_map（nav\_msgs / GetMap）

通过此服务获取地图。

参数：

〜frame\_id（字符串，默认值：“map”）

地图坐标系。

**3.2地图读取**

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="map.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find demo\_nav)/map/$(arg map)"/>

</launch>

**3.3地图显示**

在 rviz 中使用 map 组件可以显示栅格地图：

### 3.定位

所谓定位就是推算机器人自身在全局地图中的位置，当然，SLAM中也包含定位算法实现，不过SLAM的定位是用于构建全局地图的，是属于导航开始之前的阶段，而当前定位是用于导航中，导航中，机器人需要按照设定的路线运动，通过定位可以判断机器人的实际轨迹是否符合预期。在ROS的导航功能包集navigation中提供了 amcl 功能包，用于实现导航中的机器人定位。

**1.amcl简介**

AMCL(adaptive Monte Carlo Localization) 是用于2D移动机器人的概率定位系统，它实现了自适应（或KLD采样）蒙特卡洛定位方法，可以根据已有地图使用粒子滤波器推算机器人位置。

amcl已经被集成到了navigation包，navigation安装前面也有介绍，命令如下:

sudo apt install ros-<ROS版本>-navigation

**2.amcl节点说明**

amcl 功能包中的核心节点是:amcl。为了方便调用，需要先了解该节点订阅的话题、发布的话题、服务以及相关参数。

**2.1订阅的Topic**

scan(sensor\_msgs/LaserScan)

激光雷达数据。

tf(tf/tfMessage)

坐标变换消息。

initialpose(geometry\_msgs/PoseWithCovarianceStamped)

用来初始化粒子滤波器的均值和协方差。

map(nav\_msgs/OccupancyGrid)

获取地图数据。

**2.2发布的Topic**

amcl\_pose(geometry\_msgs/PoseWithCovarianceStamped)

机器人在地图中的位姿估计。

particlecloud(geometry\_msgs/PoseArray)

位姿估计集合，rviz中可以被 PoseArray 订阅然后图形化显示机器人的位姿估计集合。

tf(tf/tfMessage)

发布从 odom 到 map 的转换。

**2.3服务**

global\_localization(std\_srvs/Empty)

初始化全局定位的服务。

request\_nomotion\_update(std\_srvs/Empty)

手动执行更新和发布更新的粒子的服务。

set\_map(nav\_msgs/SetMap)

手动设置新地图和姿态的服务。

**2.4调用的服务**

static\_map(nav\_msgs/GetMap)

调用此服务获取地图数据。

**2.5参数**

~odom\_model\_type(string, default:"diff")

里程计模型选择: "diff","omni","diff-corrected","omni-corrected" (diff 差速、omni 全向轮)

~odom\_frame\_id(string, default:"odom")

里程计坐标系。

~base\_frame\_id(string, default:"base\_link")

机器人极坐标系。

~global\_frame\_id(string, default:"map")

地图坐标系。

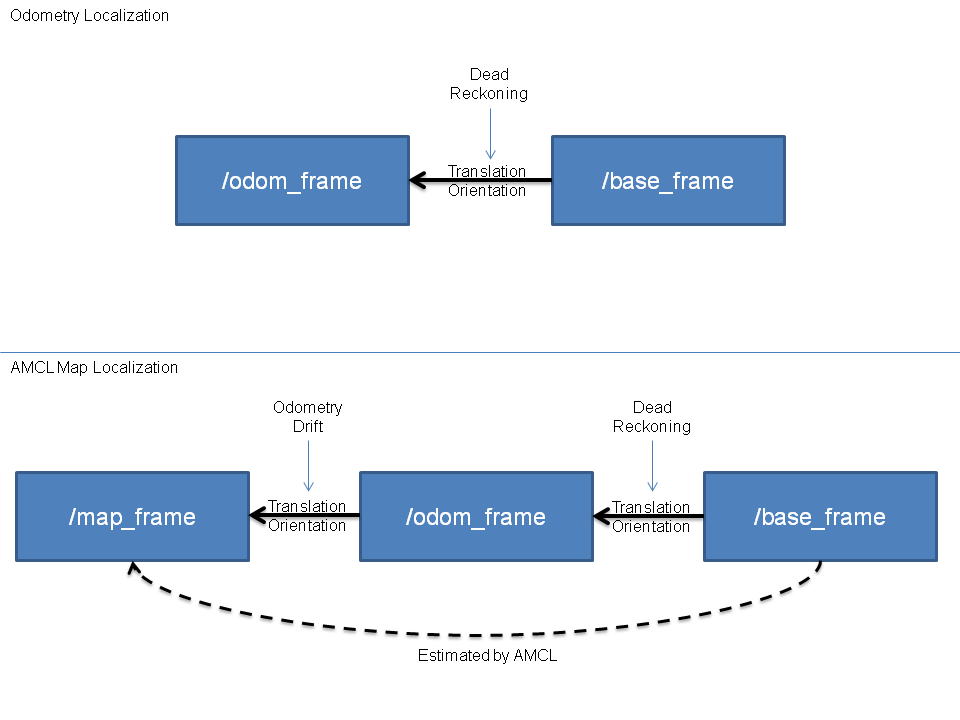
.... 参数较多，上述是几个较为常用的参数，其他参数介绍可参考官网。

**2.6坐标变换**

里程计本身也是可以协助机器人定位的，不过里程计存在累计误差且一些特殊情况时(车轮打滑)会出现定位错误的情况，amcl 则可以通过估算机器人在地图坐标系下的姿态，再结合里程计提高定位准确度。

里程计定位:只是通过里程计数据实现 /odom\_frame 与 /base\_frame 之间的坐标变换。

amcl定位: 可以提供 /map\_frame 、/odom\_frame 与 /base\_frame 之间的坐标变换。



**3.amcl使用**

**3.1编写amcl节点相关的launch文件**

关于launch文件的实现，在amcl功能包下的example目录已经给出了示例，可以作为参考，具体实现:

roscd amcl

ls examples

该目录下会列出两个文件: amcl\_diff.launch 和 amcl\_omni.launch 文件，前者适用于差分移动机器人，后者适用于全向移动机器人，可以按需选择，此处参考前者，新建 launch 文件，复制 amcl\_diff.launch 文件内容并修改如下:

<launch>

<node pkg="amcl" type="amcl" name="amcl" output="screen">

<!-- Publish scans from best pose at a max of 10 Hz -->

<param name="odom\_model\_type" value="diff"/><!-- 里程计模式为差分 -->

<param name="odom\_alpha5" value="0.1"/>

<param name="transform\_tolerance" value="0.2" />

<param name="gui\_publish\_rate" value="10.0"/>

<param name="laser\_max\_beams" value="30"/>

<param name="min\_particles" value="500"/>

<param name="max\_particles" value="5000"/>

<param name="kld\_err" value="0.05"/>

<param name="kld\_z" value="0.99"/>

<param name="odom\_alpha1" value="0.2"/>

<param name="odom\_alpha2" value="0.2"/>

<!-- translation std dev, m -->

<param name="odom\_alpha3" value="0.8"/>

<param name="odom\_alpha4" value="0.2"/>

<param name="laser\_z\_hit" value="0.5"/>

<param name="laser\_z\_short" value="0.05"/>

<param name="laser\_z\_max" value="0.05"/>

<param name="laser\_z\_rand" value="0.5"/>

<param name="laser\_sigma\_hit" value="0.2"/>

<param name="laser\_lambda\_short" value="0.1"/>

<param name="laser\_lambda\_short" value="0.1"/>

<param name="laser\_model\_type" value="likelihood\_field"/>

<!-- <param name="laser\_model\_type" value="beam"/> -->

<param name="laser\_likelihood\_max\_dist" value="2.0"/>

<param name="update\_min\_d" value="0.2"/>

<param name="update\_min\_a" value="0.5"/>

<param name="odom\_frame\_id" value="odom"/><!-- 里程计坐标系 -->

<param name="base\_frame\_id" value="base\_footprint"/><!-- 添加机器人基坐标系 -->

<param name="global\_frame\_id" value="map"/><!-- 添加地图坐标系 -->

<param name="resample\_interval" value="1"/>

<param name="transform\_tolerance" value="0.1"/>

<param name="recovery\_alpha\_slow" value="0.0"/>

<param name="recovery\_alpha\_fast" value="0.0"/>

</node>

</launch>

**3.2编写测试launch文件**

amcl节点是不可以单独运行的，运行 amcl 节点之前，需要先加载全局地图，然后启动 rviz 显示定位结果，上述节点可以集成进launch文件，内容示例如下:

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="map.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find demo\_nav)/map/$(arg map)"/>

<!-- 启动AMCL节点 -->

<include file="$(find demo\_nav)/launch/amcl.launch" />

<!-- 运行rviz -->

<node pkg="rviz" type="rviz" name="rviz"/>

<node pkg="joint\_state\_publisher" name="joint\_state\_publisher" type="joint\_state\_publisher" />

<node pkg="robot\_state\_publisher" name="robot\_state\_publisher" type="robot\_state\_publisher" />

</launch>

**3.3执行**

1.先启动 Gazebo 仿真环境(此过程略)；

2.启动键盘控制节点：

rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

3.启动上一步中集成地图服务、amcl 与 rviz 的 launch 文件；

4.在启动的 rviz 中，添加RobotModel、Map组件，分别显示机器人模型与地图，添加 posearray 插件，设置topic为particlecloud来显示 amcl 预估的当前机器人的位姿，箭头越是密集，说明当前机器人处于此位置的概率越高；

5.通过键盘控制机器人运动，会发现 posearray 也随之而改变。

### 4.路径规划

**1.move\_base简介**

move\_base 功能包提供了基于动作(action)的路径规划实现，move\_base 可以根据给定的目标点，控制机器人底盘运动至目标位置，并且在运动过程中会连续反馈机器人自身的姿态与目标点的状态信息。如前所述(7.1)move\_base主要由全局路径规划与本地路径规划组成。

move\_base已经被集成到了navigation包，navigation安装前面也有介绍，命令如下:

sudo apt install ros-<ROS版本>-navigation

**2.move\_base节点说明**

move\_base功能包中的核心节点是:move\_base。为了方便调用，需要先了解该节点action、订阅的话题、发布的话题、服务以及相关参数。

**2.1动作**

动作订阅

move\_base/goal(move\_base\_msgs/MoveBaseActionGoal)

move\_base 的运动规划目标。

move\_base/cancel(actionlib\_msgs/GoalID)

取消目标。

动作发布

move\_base/feedback(move\_base\_msgs/MoveBaseActionFeedback)

连续反馈的信息，包含机器人底盘坐标。

move\_base/status(actionlib\_msgs/GoalStatusArray)

发送到move\_base的目标状态信息。

move\_base/result(move\_base\_msgs/MoveBaseActionResult)

操作结果(此处为空)。

**2.2订阅的Topic**

move\_base\_simple/goal(geometry\_msgs/PoseStamped)

运动规划目标(与action相比，没有连续反馈，无法追踪机器人执行状态)。

**2.3发布的Topic**

cmd\_vel(geometry\_msgs/Twist)

输出到机器人底盘的运动控制消息。

**2.4服务**

~make\_plan(nav\_msgs/GetPlan)

请求该服务，可以获取给定目标的规划路径，但是并不执行该路径规划。

~clear\_unknown\_space(std\_srvs/Empty)

允许用户直接清除机器人周围的未知空间。

~clear\_costmaps(std\_srvs/Empty)

允许清除代价地图中的障碍物，可能会导致机器人与障碍物碰撞，请慎用。

**2.5参数**

请参考官网。

**3.move\_base与代价地图**

**3.1概念**

机器人导航(尤其是路径规划模块)是依赖于地图的，地图在SLAM时已经有所介绍了，ROS中的地图其实就是一张图片，这张图片有宽度、高度、分辨率等元数据，在图片中使用灰度值来表示障碍物存在的概率。不过SLAM构建的地图在导航中是不可以直接使用的，因为：

SLAM构建的地图是静态地图，而导航过程中，障碍物信息是可变的，可能障碍物被移走了，也可能添加了新的障碍物，导航中需要时时的获取障碍物信息；

在靠近障碍物边缘时，虽然此处是空闲区域，但是机器人在进入该区域后可能由于其他一些因素，比如：惯性、或者不规则形体的机器人转弯时可能会与障碍物产生碰撞，安全起见，最好在地图的障碍物边缘设置警戒区，尽量禁止机器人进入...

所以，静态地图无法直接应用于导航，其基础之上需要添加一些辅助信息的地图，比如时时获取的障碍物数据，基于静态地图添加的膨胀区等数据。

**3.2组成**

代价地图有两张:global\_costmap(全局代价地图) 和 local\_costmap(本地代价地图)，前者用于全局路径规划，后者用于本地路径规划。

两张代价地图都可以多层叠加,一般有以下层级:

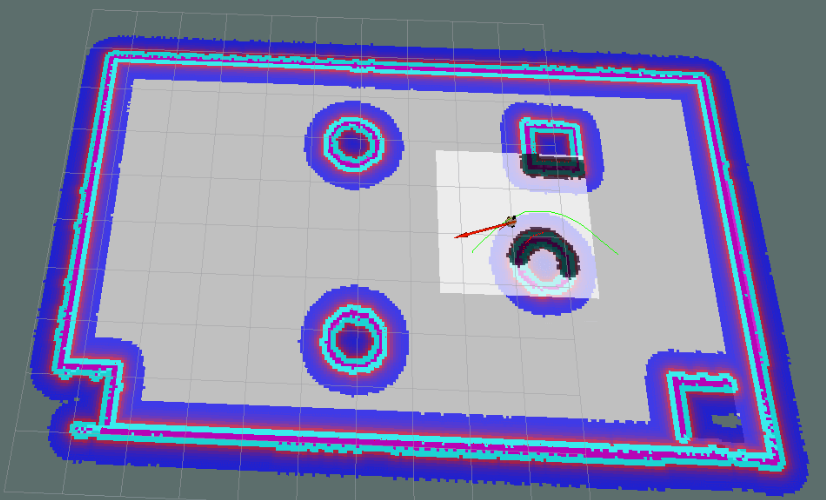
Static Map Layer：静态地图层，SLAM构建的静态地图。

Obstacle Map Layer：障碍地图层，传感器感知的障碍物信息。

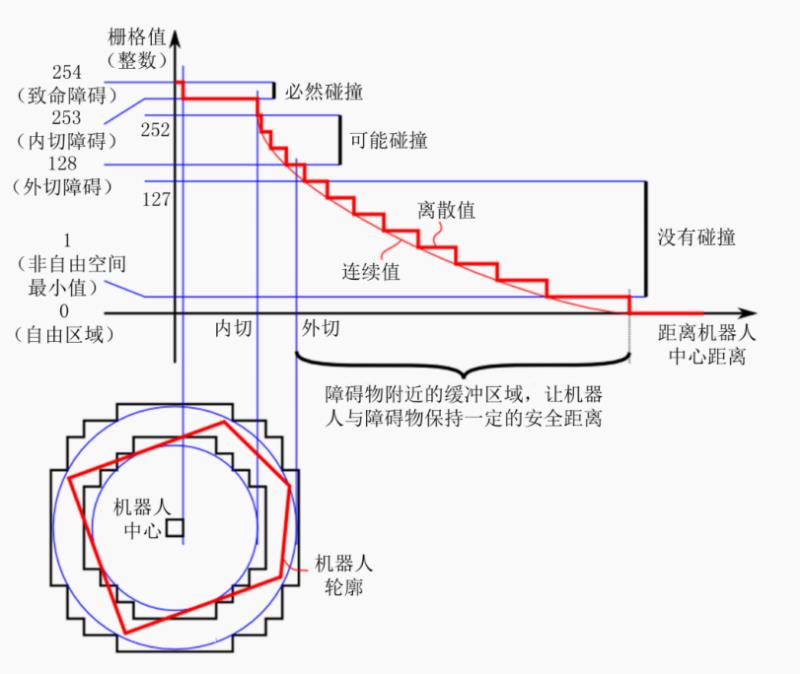
Inflation Layer：膨胀层，在以上两层地图上进行膨胀（向外扩张），以避免机器人的外壳会撞上障碍物。

Other Layers：自定义costmap。

多个layer可以按需自由搭配。



**3.3碰撞算法**



上图中，横轴是距离机器人中心的距离，纵轴是代价地图中栅格的灰度值。

致命障碍:栅格值为254，此时障碍物与机器人中心重叠，必然发生碰撞；

内切障碍:栅格值为253，此时障碍物处于机器人的内切圆内，必然发生碰撞；

外切障碍:栅格值为[128,252]，此时障碍物处于其机器人的外切圆内，处于碰撞临界，不一定发生碰撞；

非自由空间:栅格值为(0,127]，此时机器人处于障碍物附近，属于危险警戒区，进入此区域，将来可能会发生碰撞；

自由区域:栅格值为0，此处机器人可以自由通过；

未知区域:栅格值为255，还没探明是否有障碍物。

膨胀空间的设置可以参考非自由空间。

**4.move\_base使用**

路径规划算法在move\_base功能包的move\_base节点中已经封装完毕了，但是还不可以直接调用，因为算法虽然已经封装了，但是该功能包面向的是各种类型支持ROS的机器人，不同类型机器人可能大小尺寸不同，传感器不同，速度不同，应用场景不同....最后可能会导致不同的路径规划结果，那么在调用路径规划节点之前，我们还需要配置机器人参数。具体实现如下:

**4.1launch文件**

关于move\_base节点的调用，模板如下:

<launch>

<node pkg="move\_base" type="move\_base" respawn="false" name="move\_base" output="screen" clear\_params="true">

<rosparam file="$(find 功能包)/param/costmap\_common\_params.yaml" command="load" ns="global\_costmap" />

<rosparam file="$(find 功能包)/param/costmap\_common\_params.yaml" command="load" ns="local\_costmap" />

<rosparam file="$(find 功能包)/param/local\_costmap\_params.yaml" command="load" />

<rosparam file="$(find 功能包)/param/global\_costmap\_params.yaml" command="load" />

<rosparam file="$(find 功能包)/param/base\_local\_planner\_params.yaml" command="load" />

</node>

</launch>

启动了 move\_base 功能包下的 move\_base 节点，respawn 为 false，意味着该节点关闭后，不会被重启；clear\_params 为 true，意味着每次启动该节点都要清空私有参数然后重新载入；通过 rosparam 会载入若干 yaml 文件用于配置参数，这些yaml文件的配置以及作用详见下一小节内容。

**4.2配置文件**

关于配置文件的编写，可以参考一些成熟的机器人的路径规划实现，比如: turtlebot3，github链接：https://github.com/ROBOTIS-GIT/turtlebot3/tree/master/turtlebot3\_navigation/param，先下载这些配置文件备用。

在功能包下新建 param 目录，复制下载的文件到此目录: costmap\_common\_params\_burger.yaml、local\_costmap\_params.yaml、global\_costmap\_params.yaml、base\_local\_planner\_params.yaml，并将costmap\_common\_params\_burger.yaml 重命名为:costmap\_common\_params.yaml。

配置文件修改以及解释:

**4.2.1costmap\_common\_params.yaml**

该文件是move\_base 在全局路径规划与本地路径规划时调用的通用参数，包括:机器人的尺寸、距离障碍物的安全距离、传感器信息等。配置参考如下:

#机器人几何参，如果机器人是圆形，设置 robot\_radius,如果是其他形状设置 footprint

robot\_radius: 0.12 #圆形，当前机器人半径为0.1，需要考虑突出物（驱动轮突出了一些）

# footprint: [[-0.12, -0.12], [-0.12, 0.12], [0.12, 0.12], [0.12, -0.12]] #其他形状，四个点坐标

obstacle\_range: 3.0 # 用于障碍物探测，比如: 值为 3.0，意味着检测到距离小于 3 米的障碍物时，就会引入代价地图

raytrace\_range: 3.5 # 用于清除障碍物，比如：值为 3.5，意味着清除代价地图中 3.5 米以外的障碍物

#膨胀半径，扩展在碰撞区域以外的代价区域，使得机器人规划路径避开障碍物

inflation\_radius: 0.2

#代价比例系数，越大则代价值越小

cost\_scaling\_factor: 3.0

#地图类型

map\_type: costmap

#导航包所需要的传感器

observation\_sources: scan

#对传感器的坐标系和数据进行配置。这个也会用于代价地图添加和清除障碍物。例如，你可以用激光雷达传感器用于在代价地图添加障碍物，再添加kinect用于导航和清除障碍物。

scan: {sensor\_frame: laser, data\_type: LaserScan, topic: scan, marking: true, clearing: true}

**4.2.2global\_costmap\_params.yaml**

该文件用于全局代价地图参数设置:

global\_costmap: #参数服务器命名空间

global\_frame: map #地图坐标系

robot\_base\_frame: base\_footprint #机器人坐标系

# 以此实现坐标变换

update\_frequency: 1.0 #代价地图更新频率，相对于局部代价地图设置小一点

publish\_frequency: 1.0 #代价地图的发布频率，相对于局部代价地图设置小一点

transform\_tolerance: 0.5 #等待坐标变换发布信息的超时时间

static\_map: true # 是否使用一个地图或者地图服务器来初始化全局代价地图，如果不使用静态地图，这个参数为false.

**4.2.3local\_costmap\_params.yaml**

该文件用于局部代价地图参数设置:

local\_costmap:

global\_frame: odom #里程计坐标系

robot\_base\_frame: base\_footprint #机器人坐标系

update\_frequency: 10.0 #代价地图更新频率，相对于局部代价地图设置高一点

publish\_frequency: 10.0 #代价地图的发布频率，相对于局部代价地图设置高一点

transform\_tolerance: 0.5 #等待坐标变换发布信息的超时时间

static\_map: false #不需要静态地图，可以提升导航效果

rolling\_window: true #是否使用动态窗口，默认为false，在静态的全局地图中，地图不会变化

width: 3 # 局部地图宽度 单位是 m

height: 3 # 局部地图高度 单位是 m

resolution: 0.05 # 局部地图分辨率 单位是 m，一般与静态地图分辨率保持一致

**4.2.4base\_local\_planner\_params**

基本的局部规划器参数配置，这个配置文件设定了机器人的最大和最小速度限制值，也设定了加速度的阈值。

TrajectoryPlannerROS:

# Robot Configuration Parameters

max\_vel\_x: 0.5 # X 方向最大速度

min\_vel\_x: 0.1 # X 方向最小速速

max\_vel\_theta: 1.0 #

min\_vel\_theta: -1.0

min\_in\_place\_vel\_theta: 1.0 #原地旋转最小角速度

acc\_lim\_x: 1.0 # X 加速限制

acc\_lim\_y: 0.0 # Y 加速限制

acc\_lim\_theta: 0.6 # 角速度加速限制

# Goal Tolerance Parameters，目标公差

xy\_goal\_tolerance: 0.10

yaw\_goal\_tolerance: 0.05

# Differential-drive robot configuration

# 是否是全向移动机器人

holonomic\_robot: false

# Forward Simulation Parameters，前进模拟参数，若要使全局路径规划和局部更贴合，要调下面参数，比如让sim\_time更大

sim\_time: 0.8

vx\_samples: 18

vtheta\_samples: 20

sim\_granularity: 0.05

**4.2.5参数配置技巧**

以上配置在实操中，可能会出现机器人在本地路径规划时与全局路径规划不符而进入膨胀区域出现假死的情况，如何尽量避免这种情形呢？

全局路径规划与本地路径规划虽然设置的参数是一样的，但是二者路径规划和避障的职能不同，可以采用不同的参数设置策略:

全局代价地图可以将膨胀半径和障碍物系数设置的偏大一些；

本地代价地图可以将膨胀半径和障碍物系数设置的偏小一些。

这样，在全局路径规划时，规划的路径会尽量远离障碍物，而本地路径规划时，机器人即便偏离全局路径也会和障碍物之间保留更大的自由空间，从而避免了陷入“假死”的情形。

1. **注释掉costmap\_common\_params.yaml中的：**

#膨胀半径，扩展在碰撞区域以外的代价区域，使得机器人规划路径避开障碍物

inflation\_radius: 0.2

#代价比例系数，越大则代价值越小

cost\_scaling\_factor: 3.0

1. **在global\_costmap\_params.yaml中加上：**

inflation\_radius: 0.5

cost\_scaling\_factor: 10.0

1. **在local\_costmap\_params.yaml中加上：**

inflation\_radius: 0.3

cost\_scaling\_factor: 5.0

**4.3launch文件集成**

如果要实现导航，需要集成地图服务、amcl 、move\_base 与 Rviz 等，集成示例如下:

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="nav.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find demo\_nav)/map/$(arg map)"/>

<!-- 启动AMCL节点 -->

<include file="$(find demo\_nav)/launch/amcl.launch" />

<!-- 运行move\_base节点 -->

<include file="$(find demo\_nav)/launch/nav.launch" />

<!-- 运行rviz -->

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find mycar\_nav)/rviz/nav.rviz" />

<node pkg="joint\_state\_publisher" name="joint\_state\_publisher" type="joint\_state\_publisher" />

<node pkg="robot\_state\_publisher" name="robot\_state\_publisher" type="robot\_state\_publisher" />

</launch>

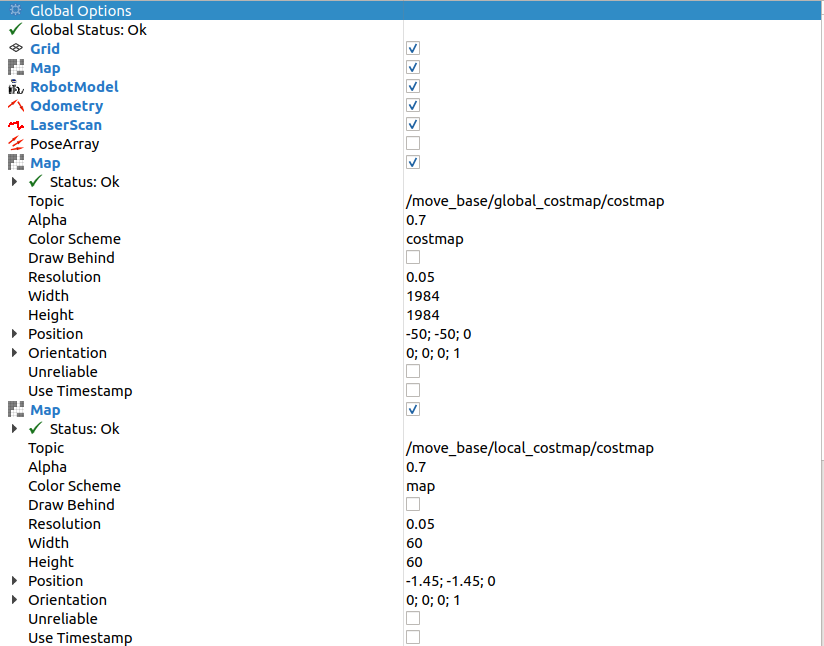
4.4测试

1.先启动 Gazebo 仿真环境；

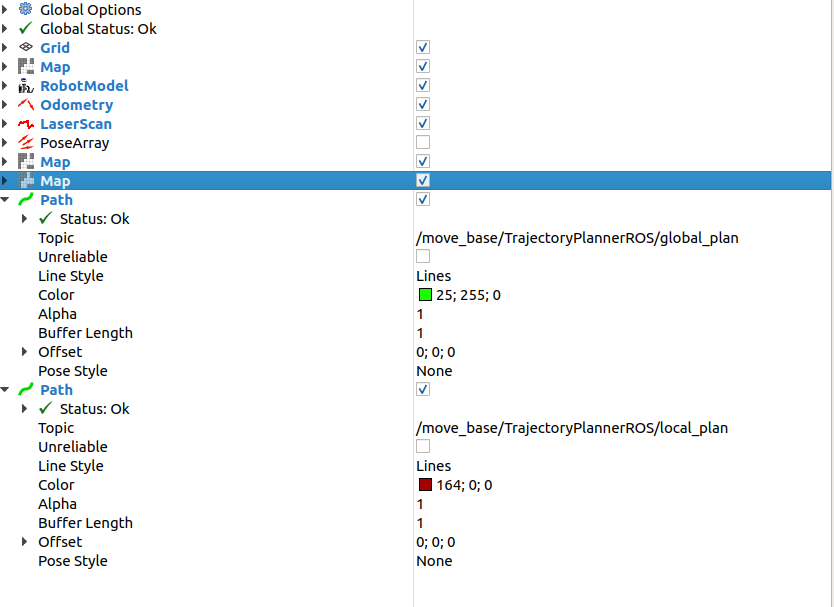
2.启动导航相关的 launch 文件；

3.添加Rviz组件(参考演示结果),可以将配置数据保存，后期直接调用；

全局代价地图与本地代价地图组件配置如下:



全局路径规划与本地路径规划组件配置如下:



4.通过Rviz工具栏的 2D Nav Goal设置目的地实现导航。

5.也可以在导航过程中，添加新的障碍物，机器人也可以自动躲避障碍物。

### 5.导航与SLAM建图

实现机器人自主移动的SLAM建图

**1.编写launc文件**

<launch>

<!-- 启动SLAM节点 -->

<include file="$(find demo\_nav)/launch/slam.launch" />

<!-- 运行move\_base节点 -->

<include file="$(find demo\_nav)/launch/nav.launch" />

<!-- 运行rviz -->

<node pkg="rviz" type="rviz" name="rviz" args="-d $(find mycar\_nav)/rviz/nav.rviz" />

<node pkg="joint\_state\_publisher" name="joint\_state\_publisher" type="joint\_state\_publisher" />

<node pkg="robot\_state\_publisher" name="robot\_state\_publisher" type="robot\_state\_publisher" />

</launch>

**2.测试**

1.首先运行gazebo仿真环境；

2.然后执行launch文件；

3.在rviz中通过2D Nav Goal设置目标点，机器人开始自主移动并建图了；

4.最后可以使用 map\_server 保存地图。

### 6.深度图像转激光数据

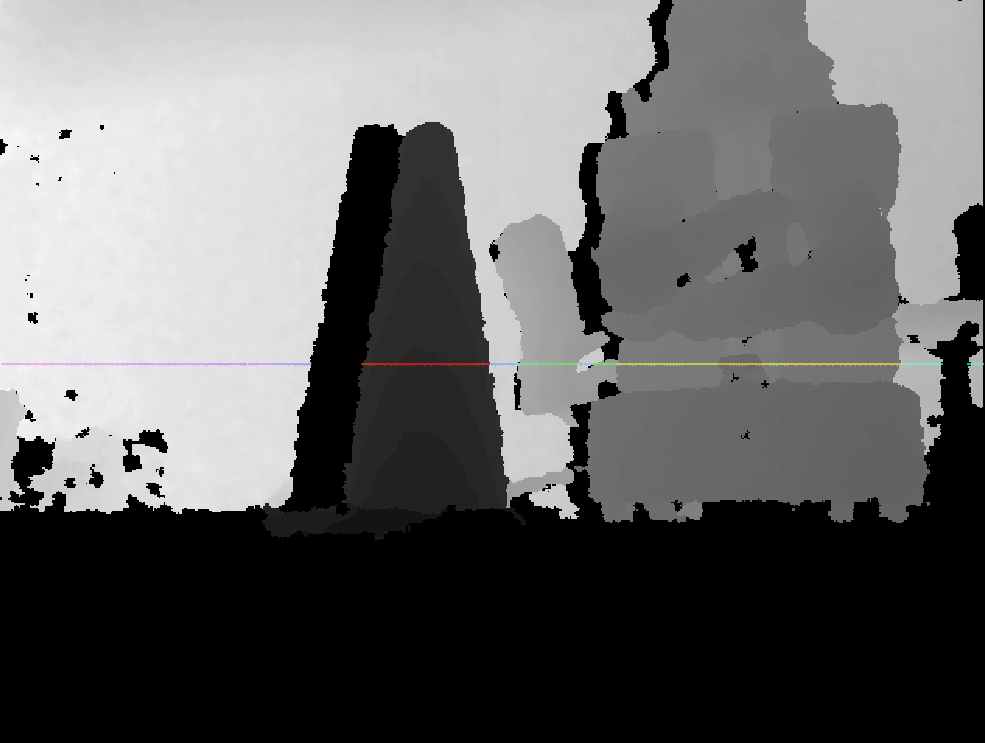
介绍ROS中的一个功能包:depthimage\_to\_laserscan，顾名思义，该功能包可以将深度图像信息转换成激光雷达信息。

**1.depthimage\_to\_laserscan简介**

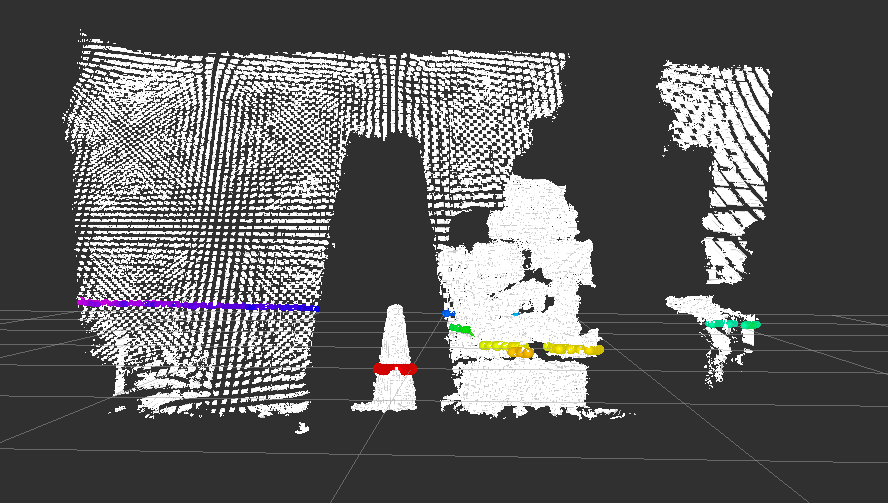
**1.1原理**

depthimage\_to\_laserscan将实现深度图像与雷达数据转换的原理比较简单，雷达数据是二维的、平面的，深度图像是三维的，是若干二维(水平)数据的纵向叠加，如果将三维的数据转换成二维数据，只需要取深度图的某一层即可，为了方面理解，请看官方示例:

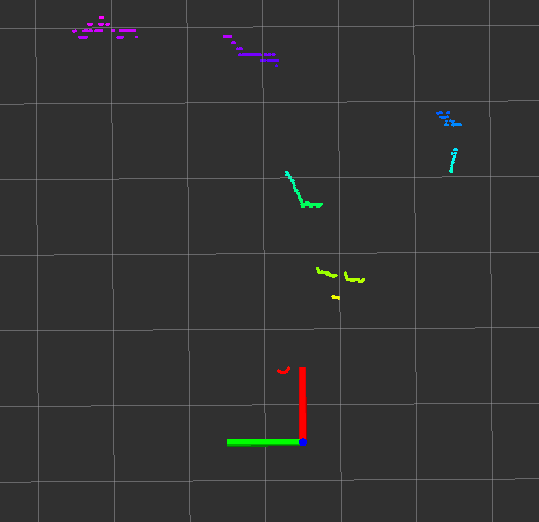
深度相机发布的图片信息，图中彩线对应的是要转换成雷达信息的数据



上图以点云的方式显示更为直观，图中彩线对应的仍然是要转换成雷达信息的数据



转换之后的结果图(俯视)



**1.2优缺点**

优点:深度相机的成本一般低于激光雷达，可以降低硬件成本；

缺点: 深度相机较之于激光雷达无论是检测范围还是精度都有不小的差距，SLAM效果可能不如激光雷达理想。

**1.3安装**

使用之前请先安装,命令如下:

sudo apt-get install ros-melodic-depthimage-to-laserscan

**2.depthimage\_to\_laserscan节点说明**

depthimage\_to\_laserscan 功能包的核心节点是:depthimage\_to\_laserscan ，为了方便调用，需要先了解该节点订阅的话题、发布的话题以及相关参数。

**2.1订阅的Topic**

image(sensor\_msgs/Image)

输入图像信息。

camera\_info(sensor\_msgs/CameraInfo)

关联图像的相机信息。通常不需要重新映射，因为camera\_info将从与image相同的命名空间中进行订阅。

**2.2发布的Topic**

scan(sensor\_msgs/LaserScan)

发布转换成的激光雷达类型数据。

**2.3参数**

该节点参数较少，只有如下几个，一般需要设置的是: output\_frame\_id。

~scan\_height(int, default: 1 pixel)

设置用于生成激光雷达信息的象素行数。

~scan\_time(double, default: 1/30.0Hz (0.033s))

两次扫描的时间间隔。

~range\_min(double, default: 0.45m)

返回的最小范围。结合range\_max使用，只会获取 range\_min 与 range\_max 之间的数据。

~range\_max(double, default: 10.0m)

返回的最大范围。结合range\_min使用，只会获取 range\_min 与 range\_max 之间的数据。

~output\_frame\_id(str, default: camera\_depth\_frame)

激光信息的ID。

**3.depthimage\_to\_laserscan使用**

**3.1编写launch文件**

编写launch文件执行，将深度信息转换成雷达信息

<launch>

<node pkg="depthimage\_to\_laserscan" type="depthimage\_to\_laserscan" name="depthimage\_to\_laserscan">

<remap from="image" to="/camera/depth/image\_raw" />

<!—value值好像无所谓是camera还是support，注释掉都能用 -->

<param name="output\_frame\_id" value="camera" />

</node>

</launch>

订阅的话题需要根据深度相机发布的话题设置，output\_frame\_id需要与深度相机的坐标系一致。

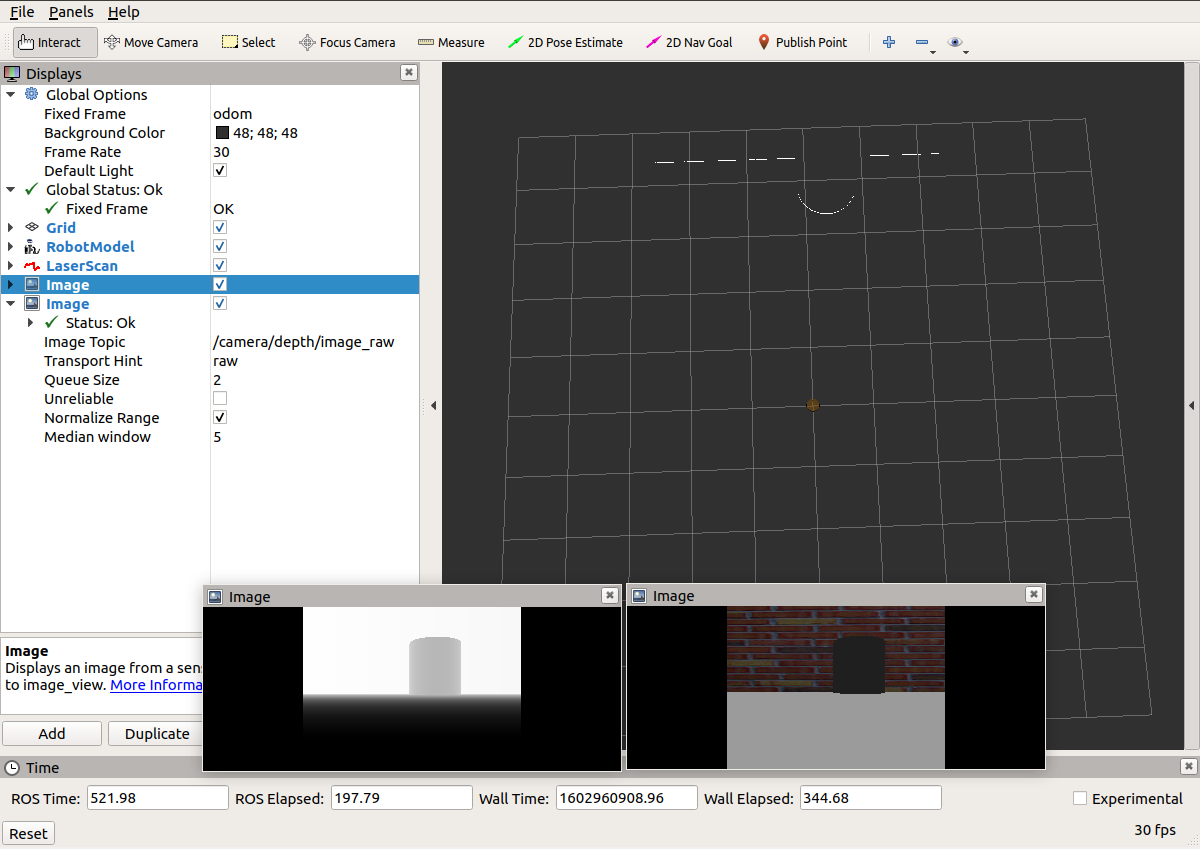
**3.2修改URDF文件**

经过信息转换之后，深度相机也将发布雷达数据，为了不产生混淆，可以注释掉 xacro 文件中的关于激光雷达的部分内容。

**3.3执行**

1.启动gazebo仿真环境

2.启动rviz并添加相关组件(image、LaserScan)，结果如下:



## 导航相关消息

### 1.地图

地图相关的消息主要有两个:

nav\_msgs/MapMetaData

地图元数据，包括地图的宽度、高度、分辨率等。

nav\_msgs/OccupancyGrid

地图栅格数据，一般会在rviz中以图形化的方式显示。

**1.nav\_msgs/MapMetaData**

调用rosmsg info nav\_msgs/MapMetaData显示消息内容如下:

time map\_load\_time

float32 resolution #地图分辨率

uint32 width #地图宽度,像素

uint32 height #地图高度，像素

geometry\_msgs/Pose origin #地图位姿数据

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

**2.nav\_msgs/OccupancyGrid**

调用 rosmsg info nav\_msgs/OccupancyGrid显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

#--- 地图元数据

nav\_msgs/MapMetaData info

time map\_load\_time

float32 resolution

uint32 width

uint32 height

geometry\_msgs/Pose origin

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

#--- 地图内容数据，数组长度 = width \* height

int8[] data

slam中data数据为-1（未知，无符号整型对应255），0（空闲），100（占据）

全局规划中：0（导航时当作未知），1-99,100（占据）

### 2.里程计

里程计相关消息是:nav\_msgs/Odometry，调用rosmsg info nav\_msgs/Odometry 显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

string child\_frame\_id

geometry\_msgs/PoseWithCovariance pose

geometry\_msgs/Pose pose #里程计位姿

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

float64[36] covariance

geometry\_msgs/TwistWithCovariance twist

geometry\_msgs/Twist twist #速度

geometry\_msgs/Vector3 linear

float64 x

float64 y

float64 z

geometry\_msgs/Vector3 angular

float64 x

float64 y

float64 z

# 协方差矩阵

float64[36] covariance

### 3.坐标变换

坐标变换相关消息是: tf/tfMessage，调用rosmsg info tf/tfMessage 显示消息内容如下:

geometry\_msgs/TransformStamped[] transforms #包含了多个坐标系相对关系数据的数组

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

string child\_frame\_id

geometry\_msgs/Transform transform

geometry\_msgs/Vector3 translation

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion rotation

float64 x

float64 y

float64 z

float64 w

### 4.定位

定位相关消息是:geometry\_msgs/PoseArray，调用rosmsg info geometry\_msgs/PoseArray显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

geometry\_msgs/Pose[] poses #预估的点位姿组成的数组

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

### 5.目标点与路径规划

目标点相关消息是:move\_base\_msgs/MoveBaseActionGoal，调用rosmsg info move\_base\_msgs/MoveBaseActionGoal显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

actionlib\_msgs/GoalID goal\_id

time stamp

string id

move\_base\_msgs/MoveBaseGoal goal

geometry\_msgs/PoseStamped target\_pose

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

geometry\_msgs/Pose pose #目标点位姿

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

路径规划相关消息是:nav\_msgs/Path，调用rosmsg info nav\_msgs/Path显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

geometry\_msgs/PoseStamped[] poses #由一系列点组成的数组

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

geometry\_msgs/Pose pose

geometry\_msgs/Point position

float64 x

float64 y

float64 z

geometry\_msgs/Quaternion orientation

float64 x

float64 y

float64 z

float64 w

### 6.激光雷达

激光雷达相关消息是:sensor\_msgs/LaserScan，调用rosmsg info sensor\_msgs/LaserScan显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

float32 angle\_min #起始扫描角度(rad)

float32 angle\_max #终止扫描角度(rad)

float32 angle\_increment #测量值之间的角距离(rad)

float32 time\_increment #测量间隔时间(s)

float32 scan\_time #扫描间隔时间(s)

float32 range\_min #最小有效距离值(m)

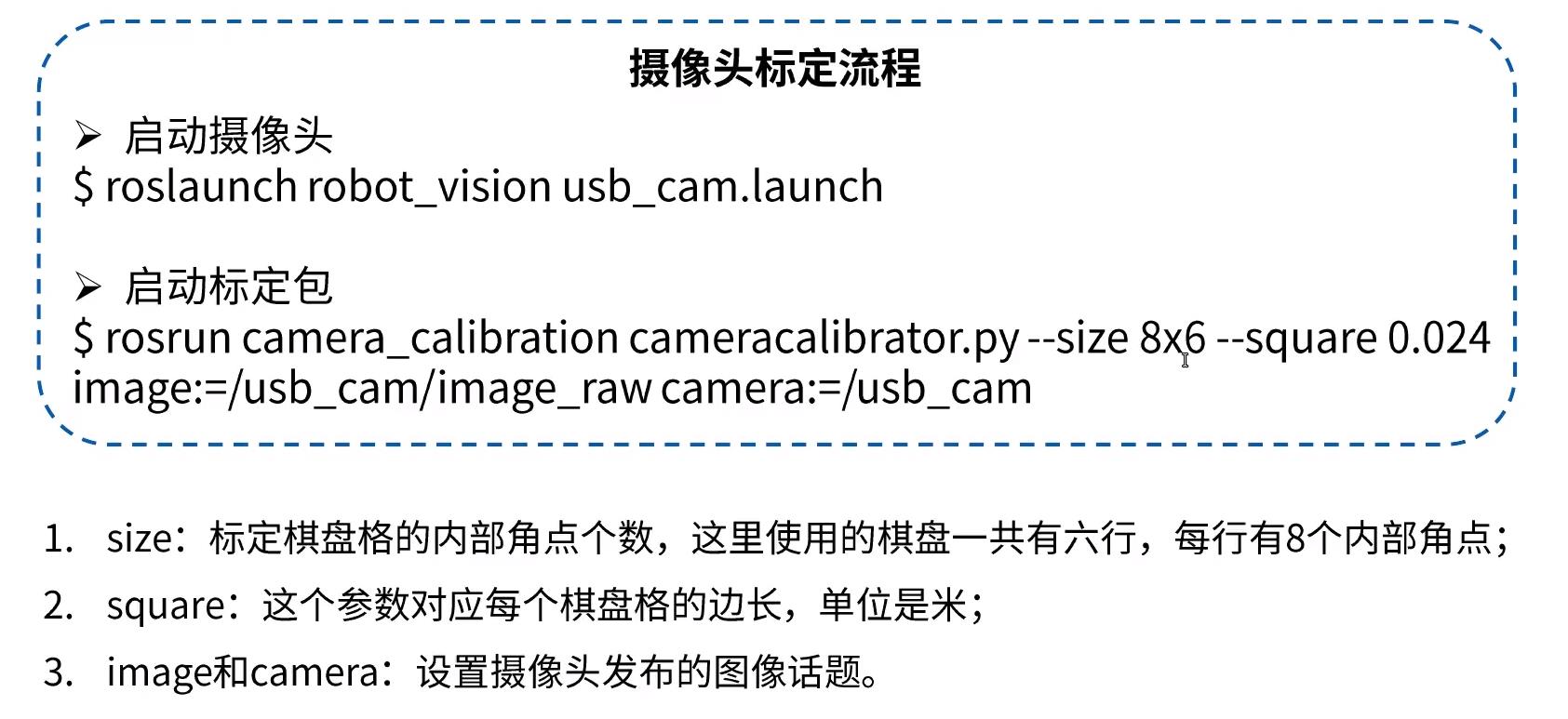
float32 range\_max #最大有效距离值(m)

float32[] ranges #一个周期的扫描数据

float32[] intensities #扫描强度数据，如果设备不支持强度数据，该数组为空

### 相机

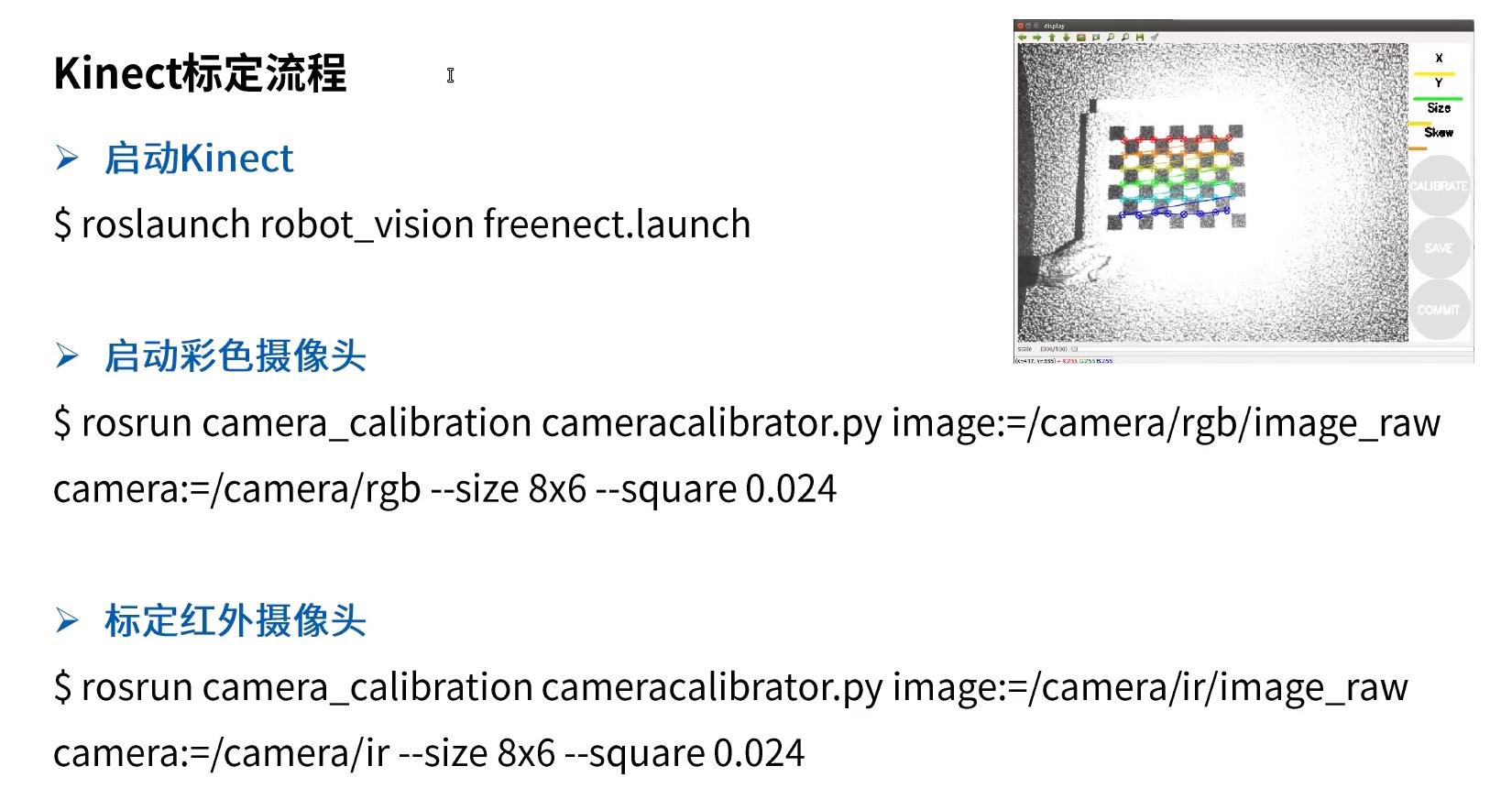
摄像头标定：sudo apt-get install ros-melodic-camera-calibration











将标定结果保存路径中的ost.yaml（可改名）放到功能包下，在launch文件中加载标定文件





深度相机相关消息有:sensor\_msgs/Image、sensor\_msgs/CompressedImage、sensor\_msgs/PointCloud2

sensor\_msgs/Image 对应的一般的图像数据，sensor\_msgs/CompressedImage 对应压缩后的图像数据，sensor\_msgs/PointCloud2 对应的是点云数据(带有深度信息的图像数据)。

调用rosmsg info sensor\_msgs/Image（二维头像）显示消息内容如下:

std\_msgs/Header header#消息头，包含消息序号，时间戳，绑定坐标系

uint32 seq

time stamp

string frame\_id

uint32 height #高度

uint32 width #宽度

string encoding #编码格式:RGB、YUV等，不涉及图像压缩编码

uint8 is\_bigendian #图像数据的大小端存储模式

uint32 step #一行图像数据的字节数，作为步进参数

uint8[] data #存储图像数据的数组，长度等于 step \* height个字节

调用rosmsg info sensor\_msgs/CompressedImage显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

string format #压缩编码格式(jpeg、png、bmp)

uint8[] data #压缩后的数据

调用rosmsg info sensor\_msgs/PointCloud2（三维头像）显示消息内容如下:

std\_msgs/Header header

uint32 seq

time stamp

string frame\_id

uint32 height #高度

uint32 width #宽度

sensor\_msgs/PointField[] fields #每个点的数据类型

uint8 INT8=1 #相当于枚举

uint8 UINT8=2

uint8 INT16=3

uint8 UINT16=4

uint8 INT32=5

uint8 UINT32=6

uint8 FLOAT32=7

uint8 FLOAT64=8

string name #x，y，z，rgb（颜色信息）

uint32 offset #上面name的偏移量

uint8 datatype #上面的数据类型（常量值，枚举）

uint32 count #占几个

bool is\_bigendian #图像大小端存储模式

uint32 point\_step #单点的数据字节步长

uint32 row\_step #一行数据的字节步长

uint8[] data #存储点云的数组，总长度为 row\_step \* height

bool is\_dense #是否有无效点

# 机器人平台设计

大部分无关被省略了

## 控制系统

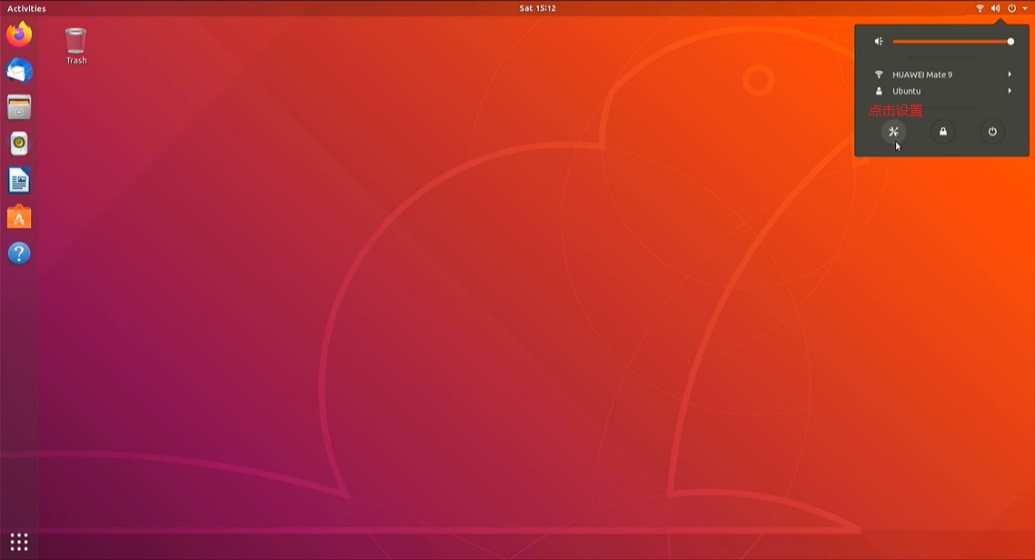
### 1.分布式框架

当前分布式框架搭建时，树莓派是作为主机，而PC则作为从机，关于分布式框架的搭建流程，在第4章4.7节中已有详细介绍，按照流程实现即可，不过在实现此流程前，还需要做准备工作：为树莓派连接无线网络，并设置固定IP，实现如下：

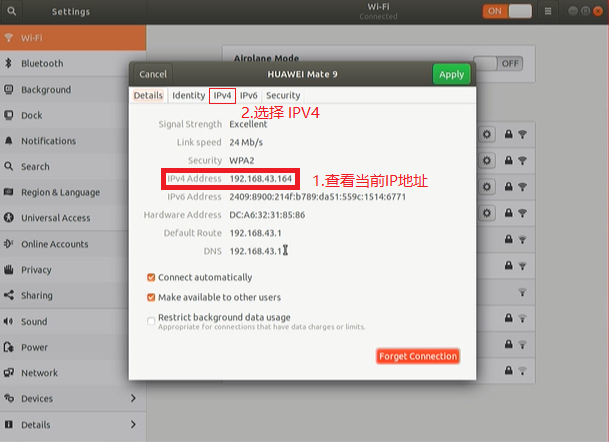
1.硬件准备：启动树莓派；

2.为树莓派连接无线网络；

3.为树莓派配置静态IP；









固定IP配置完毕后，按照4.7节演示，配置分布式框架即可。

### 2.ssh远程连接

概念：SSH（Secure Shell）是一种通用的、功能强大的、基于软件的网络安全解决方案。计算机每次向网络发送数据时，SSH都会自动对其进行加密。数据到达目的地时，SSH自动对加密数据进行解密。整个过程都是透明的，使用OpenSSH工具将会增进你的系统安全性。SSH安装容易、使用简单。

实现：SSH实现架构上分为客户端和服务器端两大部分，客户端是数据的发送方，服务端是数据的接收方，当前场景下，我们需要从PC端发送数据到树莓派，那么PC端属于客户端，而树莓派属于服务端，整个实现具体流程是:

**1.安装SSH客户端与服务端**

默认情况下，Ubuntu系统已经安装了SSH客户端，因此只需要在树莓派安装服务端即可(如果树莓派安装的是服务版的Ubuntu，默认会安装SSH服务并已设置成了开机自启动):

sudo apt-get install openssh-server

如果客户端需要自行安装，那么调用如下命令:

sudo apt-get install openssh-client

**2.服务端启动SSH服务**

树莓派启动 ssh 服务:

sudo /etc/init.d/ssh start

启动后查看服务是否正常运行:

ps -e | grep ssh

如果启动成功，会包含 sshd 与 ssh 两个程序。

以后需要频繁的使用ssh登录树莓派，为了简化实现，可以将树莓派的ssh服务设置为开机自启动，命令如下:

sudo systemctl enable ssh

**3.客户端远程登陆服务端**

登陆树莓派可以调用如下命令:

ssh 账号@ip地址

然后根据提示，录入登陆密码，即可成功登陆。

如果退出登陆，可以调用exit命令: exit

**4.实现数据传输**

上传文件:

scp 本地文件路径 账号@ip:树莓派路径

上传文件夹:

scp -r 本地文件夹路径 账号@ip:树莓派路径

下载文件:

scp 账号@ip:树莓派路径 本地文件夹路径

下载文件夹:

scp -r 账号@ip:树莓派路径 本地文件夹路径

**使用优化**

每次登陆树莓派时，都需要输入密码，使用不方便，可以借助密钥简化登陆过程，实现免密登陆，提高操作效率，实现思想是:生成一对公钥私钥，私钥存储在本地，公钥上传至服务器，每次登陆时，本地直接上传私钥到服务器，服务器有匹配的公钥就认为是合法用户，直接创建SSH连接即可。具体实现步骤只有两步:

**1.生成密钥对**

本地客户端生成公私钥：（一路回车默认即可）

ssh-keygen

上面这个命令会在用户目录.ssh文件夹下创建公私钥:

id\_rsa （私钥）

id\_rsa.pub (公钥)

**2.将公钥上传至树莓派**

上传命令:

ssh-copy-id -i ~/.ssh/id\_rsa.pub 账号@ip

上面这条命令是写到服务器上的ssh目录下，该目录下有文件authorized\_keys保存了公钥内容。

以后再登陆树莓派就无需录入密码了。

### 3.树莓派安装ROS

<http://www.autolabor.com.cn/book/ROSTutorials/di-8-zhang-gou-jian-lun-shi-cha-fen-ji-qi-ren/87-ji-qi-ren-ping-tai-she-ji-zhi-kong-zhi-xi-tong/872-kong-zhi-xi-tong-shi-xian-shu-mei-pai-an-zhuang-ros.html>

# 机器人导航(实体)

## 概述

实体机器人导航与仿真环境下的导航核心实现基本一致，主要区别在于导航实现之前，基本环境的搭建有所不同，比如:导航场景、传感器、机器人模型等，大致区别如下:



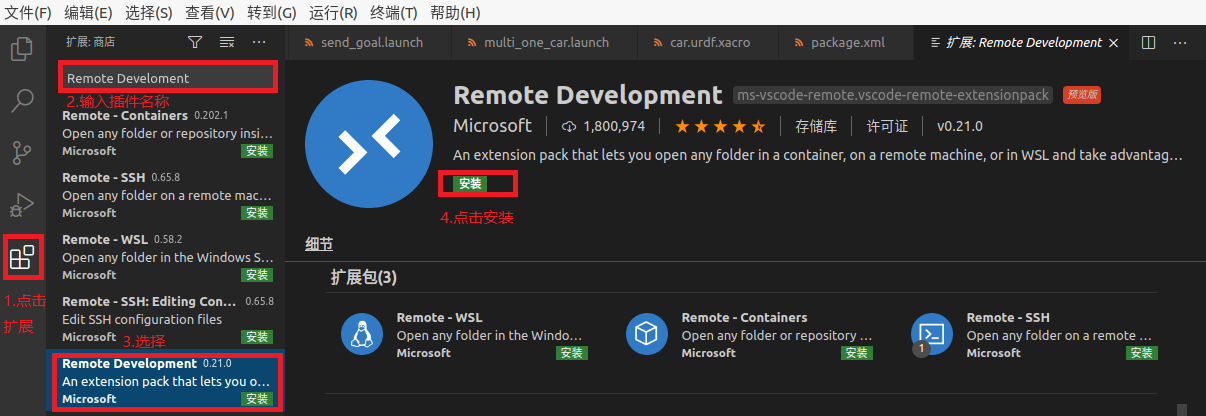
## VScode远程开发

**1.准备工作**

VScode远程开发依赖于ssh，请首先按照8.1.2内容配置ssh远程连接。

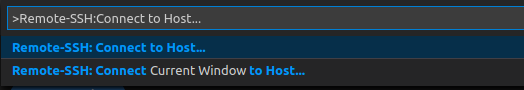
**2.为VScode安装远程开发插件**

启动VScode，首先点击侧边栏的扩展按钮，然后在扩展：商店的搜索栏输入Remote Development并点击同名插件，最后在右侧显示区中点击安装。

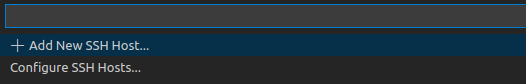


**3.配置远程连接**

步骤1：使用快捷键ctrl + shift + p打开命令输入窗口，并输入Remote-SSH:Connect to Host...，弹出列表中选择与之同名的选项。

****

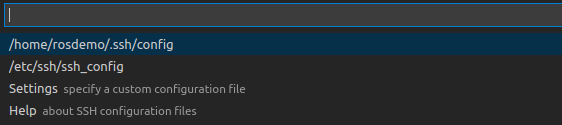
步骤2：步骤1完成将弹出一个新的命令窗口如下，选择下拉列表中的 Add New SSH Host。

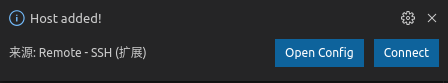


步骤3：步骤2完成又将弹出一个新的命令窗口，在其中输入：ssh ubuntu@192.168.43.164，其中，ubuntu需要替换为你的登录账号，192.18.43.164 则替换为你的树莓派的ip地址。



步骤4：选择步骤3完成后的弹窗列表中的第一个选项(或直接回车)，即可完成配置，配置成果后会有提示信息。



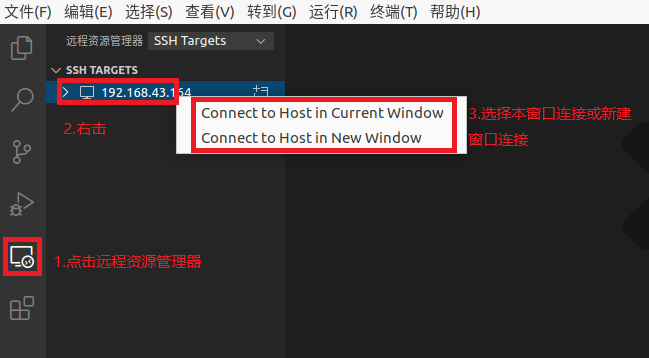


**4.使用**

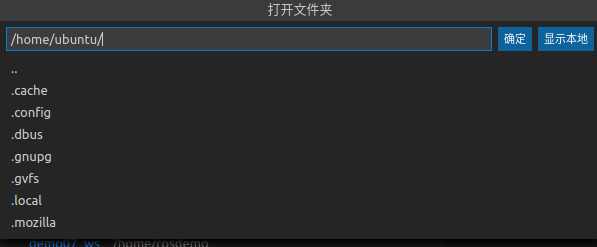
步骤1：继续使用快捷键ctrl + shift + p打开命令输入窗口，并输入Remote-SSH:Connect to Host...，此时列表中将显示步骤3中配置的ip地址，直接选择，选择后，VScode将打卡一个新的窗口。



或者，也可以点击侧边栏的远程资源管理器，在弹出的服务器列表中选择要连接的服务器，并右击，选择在本窗口或新窗口中实现远程连接。



步骤2：选择菜单栏的文件下的打开文件夹，在弹窗列表中选择需要打开的文件夹并点击确定即可。



## 导航实现

### 1.准备工作

**1.1分布式架构**

分布式架构搭建完毕且能正常运行，在PC端可以远程登陆机器人端。

**1.2功能包安装**

在机器人端安装导航所需功能包：

安装 gmapping 包(用于构建地图):sudo apt install ros-<ROS版本>-gmapping

安装地图服务包(用于保存与读取地图):sudo apt install ros-<ROS版本>-map-server

安装 navigation 包(用于定位以及路径规划):sudo apt install ros-<ROS版本>-navigation

新建功能（包名自定义，比如：nav），并导入依赖: gmapping map\_server amcl move\_base

**1.3机器人模型以及坐标变换**

机器人的不同部件有不同的坐标系，我们需要将这些坐标系集成进同一坐标树，实现方案有两种：

1.不同的部件相对于机器人底盘其位置都是固定的，可以通过发布静态坐标变换（static\_transform\_publisher）以实现集成；

2．可以通过加载机器人URDF文件结合 robot\_state\_publisher、joint\_state\_publisher实现不同坐标系的集成。

方案1在上一章中已做演示：

<!-- 机器人启动文件：

当不包含机器人模型时，需要发布坐标变换 -->

<launch>

<include file="$(find mycar\_start)/launch/start.launch" />

<node name="camera2basefootprint" pkg="tf2\_ros" type="static\_transform\_publisher" args="0.08 0 0.1 0 0 0 /base\_footprint /camera\_link"/>

<node name="rplidar2basefootprint" pkg="tf2\_ros" type="static\_transform\_publisher" args="0 0 0.1 0 0 0 /base\_footprint /laser"/>

</launch>

接下来介绍方案2的实现。

**1.3.1 创建机器人模型相关的功能包**

创建功能包:catkin\_create\_pkg mycar\_description urdf xacro。

**1.3.2 准备机器人模型文件**

在功能包下新建 urdf 目录，编写具体的 urdf 文件（机器人模型相关URDF文件的编写可以参考第6章内容），示例如下：

文件car.urdf.xacro用于集成不同的机器人部件，内容如下：

<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="car\_base.urdf.xacro" />

<xacro:include filename="car\_camera.urdf.xacro" />

<xacro:include filename="car\_laser.urdf.xacro" />

</robot>

文件car\_base.urdf.xacro机器人底盘实现，内容如下：

<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:property name="footprint\_radius" value="0.001" />

<link name="base\_footprint">

<visual>

<geometry>

<sphere radius="${footprint\_radius}" />

</geometry>

</visual>

</link>

<xacro:property name="base\_radius" value="0.1" />

<xacro:property name="base\_length" value="0.08" />

<xacro:property name="lidi" value="0.015" />

<xacro:property name="base\_joint\_z" value="${base\_length / 2 + lidi}" />

<link name="base\_link">

<visual>

<geometry>

<cylinder radius="0.1" length="0.08" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="baselink\_color">

<color rgba="1.0 0.5 0.2 0.5" />

</material>

</visual>

</link>

<joint name="link2footprint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link" />

<origin xyz="0 0 0.055" rpy="0 0 0" />

</joint>

<xacro:property name="wheel\_radius" value="0.0325" />

<xacro:property name="wheel\_length" value="0.015" />

<xacro:property name="PI" value="3.1415927" />

<xacro:property name="wheel\_joint\_z" value="${(base\_length / 2 + lidi - wheel\_radius) \* -1}" />

<xacro:macro name="wheel\_func" params="wheel\_name flag">

<link name="${wheel\_name}\_wheel">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="${PI / 2} 0 0" />

<material name="wheel\_color">

<color rgba="0 0 0 0.3" />

</material>

</visual>

</link>

<joint name="${wheel\_name}2link" type="continuous">

<parent link="base\_link" />

<child link="${wheel\_name}\_wheel" />

<origin xyz="0 ${0.1 \* flag} ${wheel\_joint\_z}" rpy="0 0 0" />

<axis xyz="0 1 0" />

</joint>

</xacro:macro>

<xacro:wheel\_func wheel\_name="left" flag="1" />

<xacro:wheel\_func wheel\_name="right" flag="-1" />

<xacro:property name="small\_wheel\_radius" value="0.0075" />

<xacro:property name="small\_joint\_z" value="${(base\_length / 2 + lidi - small\_wheel\_radius) \* -1}" />

<xacro:macro name="small\_wheel\_func" params="small\_wheel\_name flag">

<link name="${small\_wheel\_name}\_wheel">

<visual>

<geometry>

<sphere radius="${small\_wheel\_radius}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="wheel\_color">

<color rgba="0 0 0 0.3" />

</material>

</visual>

</link>

<joint name="${small\_wheel\_name}2link" type="continuous">

<parent link="base\_link" />

<child link="${small\_wheel\_name}\_wheel" />

<origin xyz="${0.08 \* flag} 0 ${small\_joint\_z}" rpy="0 0 0" />

<axis xyz="0 1 0" />

</joint>

</xacro:macro >

<xacro:small\_wheel\_func small\_wheel\_name="front" flag="1"/>

<xacro:small\_wheel\_func small\_wheel\_name="back" flag="-1"/>

</robot>

文件car\_camera.urdf.xacro机器人摄像头实现，内容如下：

<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:property name="camera\_length" value="0.02" />

<xacro:property name="camera\_width" value="0.05" />

<xacro:property name="camera\_height" value="0.05" />

<xacro:property name="joint\_camera\_x" value="0.08" />

<xacro:property name="joint\_camera\_y" value="0" />

<xacro:property name="joint\_camera\_z" value="${base\_length / 2 + camera\_height / 2}" />

<link name="camera">

<visual>

<geometry>

<box size="${camera\_length} ${camera\_width} ${camera\_height}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="black">

<color rgba="0 0 0 0.8" />

</material>

</visual>

</link>

<joint name="camera2base" type="fixed">

<parent link="base\_link" />

<child link="camera" />

<origin xyz="${joint\_camera\_x} ${joint\_camera\_y} ${joint\_camera\_z}" rpy="0 0 0" />

</joint>

</robot>

文件car\_laser.urdf.xacro机器人雷达实现，内容如下：

<robot name="mycar" xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:property name="support\_radius" value="0.01" />

<xacro:property name="support\_length" value="0.15" />

<xacro:property name="laser\_radius" value="0.03" />

<xacro:property name="laser\_length" value="0.05" />

<xacro:property name="joint\_support\_x" value="0" />

<xacro:property name="joint\_support\_y" value="0" />

<xacro:property name="joint\_support\_z" value="${base\_length / 2 + support\_length / 2}" />

<xacro:property name="joint\_laser\_x" value="0" />

<xacro:property name="joint\_laser\_y" value="0" />

<xacro:property name="joint\_laser\_z" value="${support\_length / 2 + laser\_length / 2}" />

<link name="support">

<visual>

<geometry>

<cylinder radius="${support\_radius}" length="${support\_length}" />

</geometry>

<material name="yellow">

<color rgba="0.8 0.5 0.0 0.5" />

</material>

</visual>

</link>

<joint name="support2base" type="fixed">

<parent link="base\_link" />

<child link="support"/>

<origin xyz="${joint\_support\_x} ${joint\_support\_y} ${joint\_support\_z}" rpy="0 0 0" />

</joint>

<link name="laser">

<visual>

<geometry>

<cylinder radius="${laser\_radius}" length="${laser\_length}" />

</geometry>

<material name="black">

<color rgba="0 0 0 0.5" />

</material>

</visual>

</link>

<joint name="laser2support" type="fixed">

<parent link="support" />

<child link="laser"/>

<origin xyz="${joint\_laser\_x} ${joint\_laser\_y} ${joint\_laser\_z}" rpy="0 0 0" />

</joint>

</robot>

**1.3.3 在launch文件加载机器人模型**

launch 文件（文件名称自定义，比如：car.launch）内容示例如下：

<launch>

<!-- 将 Urdf 文件的内容加载到参数服务器 -->

<param name="robot\_description" command="$(find xacro)/xacro $(find mycar\_description)/urdf/car.urdf.xacro" />

<node pkg="joint\_state\_publisher" name="joint\_state\_publisher" type="joint\_state\_publisher" />

<node pkg="robot\_state\_publisher" name="robot\_state\_publisher" type="robot\_state\_publisher" />

</launch>

为了使用方便，还可以将该文件包含进启动机器人的launch文件中，示例如下：

<launch>

<include file="$(find ros\_arduino\_python)/launch/arduino.launch" />

<include file="$(find usb\_cam)/launch/usb\_cam-test.launch" />

<include file="$(find rplidar\_ros)/launch/rplidar.launch" />

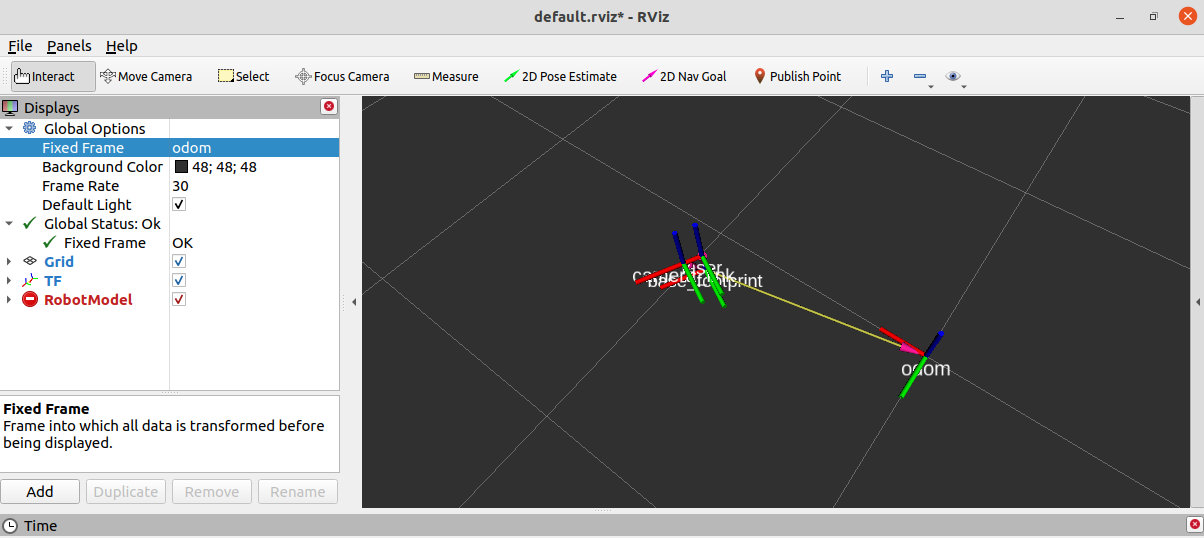
<!-- 机器人模型加载文件 -->

<include file="$(find mycar\_description)/launch/car.launch" />

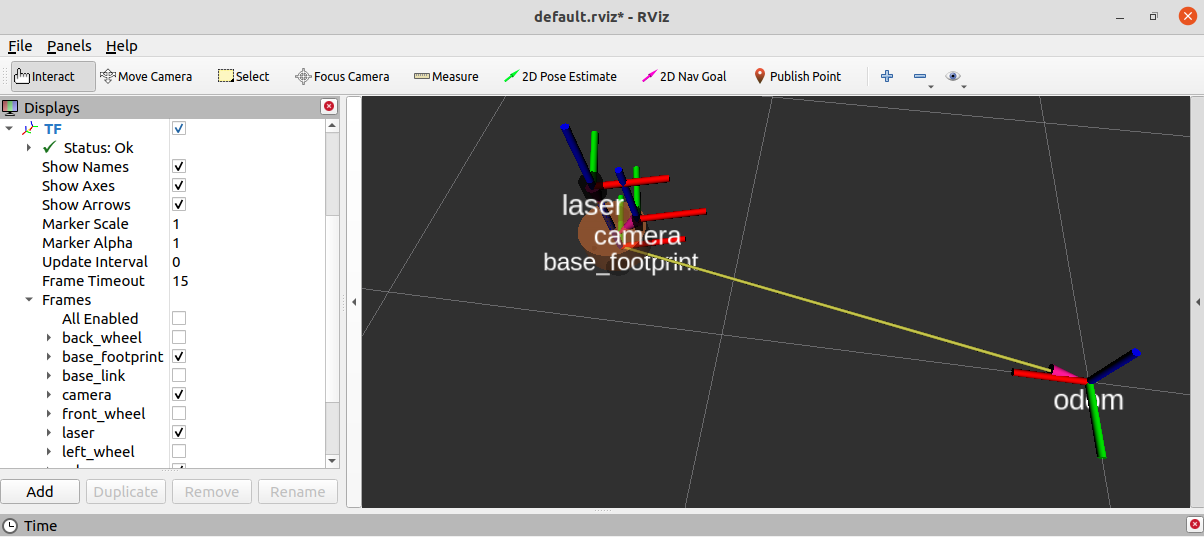
</launch>

**1.4结果演示**

不使用机器人模型时，机器人端启动机器人(使用包含TF坐标换的launch文件)，从机端启动rviz，在rviz中添加RobotModel与TF组件，rviz中结果(此时显示机器人模型异常，且TF中只有代码中发布的坐标变换):



使用机器人模型时，机器人端加载机器人模型（执行上一步的launch文件）且启动机器人，从机端启动rviz，，在rviz中添加RobotModel与TF组件rviz中结果(此时显示机器人模型，且TF坐标变换正常):



### 2.SLAM建图

**2.1编写gmapping节点相关launch文件**

在上一节创建的导航功能包中新建 launch 目录，并新建launch文件（文件名自定义，比如： gmapping.launch），代码示例如下:

<launch>

<node pkg="gmapping" type="slam\_gmapping" name="slam\_gmapping" output="screen">

<remap from="scan" to="scan"/><!-- 雷达话题 -->

<param name="base\_frame" value="base\_footprint"/><!--底盘坐标系-->

<param name="odom\_frame" value="odom"/> <!--里程计坐标系-->

<param name="map\_update\_interval" value="5.0"/>

<param name="maxUrange" value="16.0"/>

<param name="sigma" value="0.05"/>

<param name="kernelSize" value="1"/>

<param name="lstep" value="0.05"/>

<param name="astep" value="0.05"/>

<param name="iterations" value="5"/>

<param name="lsigma" value="0.075"/>

<param name="ogain" value="3.0"/>

<param name="lskip" value="0"/>

<param name="srr" value="0.1"/>

<param name="srt" value="0.2"/>

<param name="str" value="0.1"/>

<param name="stt" value="0.2"/>

<param name="linearUpdate" value="1.0"/>

<param name="angularUpdate" value="0.5"/>

<param name="temporalUpdate" value="3.0"/>

<param name="resampleThreshold" value="0.5"/>

<param name="particles" value="30"/>

<param name="xmin" value="-50.0"/>

<param name="ymin" value="-50.0"/>

<param name="xmax" value="50.0"/>

<param name="ymax" value="50.0"/>

<param name="delta" value="0.05"/>

<param name="llsamplerange" value="0.01"/>

<param name="llsamplestep" value="0.01"/>

<param name="lasamplerange" value="0.005"/>

<param name="lasamplestep" value="0.005"/>

</node>

</launch>

**2.2执行**

1.执行相关launch文件，启动机器人并加载机器人模型：roslaunch mycar\_start start.launch；

2.启动地图绘制的 launch 文件：roslaunch nav gmapping.launch；

3.启动键盘键盘控制节点，用于控制机器人运动建图：rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py

4.在 rviz 中添加地图显示组件，通过键盘控制机器人运动，同时，在rviz中可以显示gmapping发布的栅格地图数据了，该显示结果与仿真环境下类似。下一步，还需要将地图单独保存。

### 3.地图服务

**3.1地图保存launch文件**

首先在自定义的导航功能包下新建 map 目录，用于保存生成的地图数据。地图保存的语法比较简单，编写一个launch文件，内容如下:

<launch>

<arg name="filename" value="$(find nav)/map/nav" />

<node name="map\_save" pkg="map\_server" type="map\_saver" args="-f $(arg filename)" />

</launch>

测试:

首先，参考上一节，依次启动仿真环境，键盘控制节点与SLAM节点；

然后，通过键盘控制机器人运动并绘图；

最后，通过上述地图保存方式保存地图。

结果：在指定路径下会生成两个文件，xxx.pgm 与 xxx.yaml

**3.2地图读取**

通过 map\_server 的 map\_server 节点可以读取栅格地图数据，编写 launch 文件如下:

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="nav.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find mycar\_nav)/map/$(arg map)"/>

</launch>

其中参数是地图描述文件的资源路径，执行该launch文件，该节点会发布话题:map(nav\_msgs/OccupancyGrid)，最后，在 rviz 中使用 map 组件可以显示栅格地图。

### 4.定位

**4.1编写amcl节点相关的launch文件**

<launch>

<node pkg="amcl" type="amcl" name="amcl" output="screen">

<!-- Publish scans from best pose at a max of 10 Hz -->

<param name="odom\_model\_type" value="diff"/><!-- 里程计模式为差分 -->

<param name="odom\_alpha5" value="0.1"/>

<param name="transform\_tolerance" value="0.2" />

<param name="gui\_publish\_rate" value="10.0"/>

<param name="laser\_max\_beams" value="30"/>

<param name="min\_particles" value="500"/>

<param name="max\_particles" value="5000"/>

<param name="kld\_err" value="0.05"/>

<param name="kld\_z" value="0.99"/>

<param name="odom\_alpha1" value="0.2"/>

<param name="odom\_alpha2" value="0.2"/>

<!-- translation std dev, m -->

<param name="odom\_alpha3" value="0.8"/>

<param name="odom\_alpha4" value="0.2"/>

<param name="laser\_z\_hit" value="0.5"/>

<param name="laser\_z\_short" value="0.05"/>

<param name="laser\_z\_max" value="0.05"/>

<param name="laser\_z\_rand" value="0.5"/>

<param name="laser\_sigma\_hit" value="0.2"/>

<param name="laser\_lambda\_short" value="0.1"/>

<param name="laser\_lambda\_short" value="0.1"/>

<param name="laser\_model\_type" value="likelihood\_field"/>

<!-- <param name="laser\_model\_type" value="beam"/> -->

<param name="laser\_likelihood\_max\_dist" value="2.0"/>

<param name="update\_min\_d" value="0.2"/>

<param name="update\_min\_a" value="0.5"/>

<param name="odom\_frame\_id" value="odom"/><!-- 里程计坐标系 -->

<param name="base\_frame\_id" value="base\_footprint"/><!-- 添加机器人基坐标系 -->

<param name="global\_frame\_id" value="map"/><!-- 添加地图坐标系 -->

</node>

</launch>

**4.2编写测试launch文件**

amcl节点是不可以单独运行的，运行 amcl 节点之前，需要先加载全局地图，然后启动 rviz 显示定位结果，上述节点可以集成进launch文件，内容示例如下:

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="nav.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find nav)/map/$(arg map)"/>

<!-- 启动AMCL节点 -->

<include file="$(find nav)/launch/amcl.launch" />

</launch>

**4.3执行**

1.执行相关launch文件，启动机器人并加载机器人模型：roslaunch mycar\_start start.launch；

2.启动键盘控制节点：rosrun teleop\_twist\_keyboard teleop\_twist\_keyboard.py；

3.启动上一步中集成地图服务、amcl 的 launch 文件：roslaunch nav test\_amcl.launch；

4.启动rviz并添加RobotModel、Map组件，分别显示机器人模型与地图，添加 posearray 插件，设置topic为particlecloud来显示 amcl 预估的当前机器人的位姿，箭头越是密集，说明当前机器人处于此位置的概率越高；

5.通过键盘控制机器人运动，会发现 posearray 也随之而改变。运行结果与仿真环境下类似。

### 5.路径规划

**5.1编写launch文件**

关于move\_base节点的调用，模板如下:

<launch>

<node pkg="move\_base" type="move\_base" respawn="false" name="move\_base" output="screen" clear\_params="true">

<rosparam file="$(find nav)/param/costmap\_common\_params.yaml" command="load" ns="global\_costmap" />

<rosparam file="$(find nav)/param/costmap\_common\_params.yaml" command="load" ns="local\_costmap" />

<rosparam file="$(find nav)/param/local\_costmap\_params.yaml" command="load" />

<rosparam file="$(find nav)/param/global\_costmap\_params.yaml" command="load" />

<rosparam file="$(find nav)/param/base\_local\_planner\_params.yaml" command="load" />

</node>

</launch>

**5.2编写配置文件**

可参考仿真实现。

**1.costmap\_common\_params.yaml**

该文件是move\_base 在全局路径规划与本地路径规划时调用的通用参数，包括:机器人的尺寸、距离障碍物的安全距离、传感器信息等。配置参考如下:

#机器人几何参，如果机器人是圆形，设置 robot\_radius,如果是其他形状设置 footprint

robot\_radius: 0.12 #圆形

# footprint: [[-0.12, -0.12], [-0.12, 0.12], [0.12, 0.12], [0.12, -0.12]] #其他形状

obstacle\_range: 3.0 # 用于障碍物探测，比如: 值为 3.0，意味着检测到距离小于 3 米的障碍物时，就会引入代价地图

raytrace\_range: 3.5 # 用于清除障碍物，比如：值为 3.5，意味着清除代价地图中 3.5 米以外的障碍物

#膨胀半径，扩展在碰撞区域以外的代价区域，使得机器人规划路径避开障碍物

inflation\_radius: 0.2

#代价比例系数，越大则代价值越小

cost\_scaling\_factor: 3.0

#地图类型

map\_type: costmap

#导航包所需要的传感器

observation\_sources: scan

#对传感器的坐标系和数据进行配置。这个也会用于代价地图添加和清除障碍物。例如，你可以用激光雷达传感器用于在代价地图添加障碍物，再添加kinect用于导航和清除障碍物。

scan: {sensor\_frame: laser, data\_type: LaserScan, topic: scan, marking: true, clearing: true}

**2.global\_costmap\_params.yaml**

该文件用于全局代价地图参数设置:

global\_costmap:

global\_frame: map #地图坐标系

robot\_base\_frame: base\_footprint #机器人坐标系

# 以此实现坐标变换

update\_frequency: 1.0 #代价地图更新频率

publish\_frequency: 1.0 #代价地图的发布频率

transform\_tolerance: 0.5 #等待坐标变换发布信息的超时时间

static\_map: true # 是否使用一个地图或者地图服务器来初始化全局代价地图，如果不使用静态地图，这个参数为false.

**3.local\_costmap\_params.yaml**

该文件用于局部代价地图参数设置:

local\_costmap:

global\_frame: odom #里程计坐标系

robot\_base\_frame: base\_footprint #机器人坐标系

update\_frequency: 10.0 #代价地图更新频率

publish\_frequency: 10.0 #代价地图的发布频率

transform\_tolerance: 0.5 #等待坐标变换发布信息的超时时间

static\_map: false #不需要静态地图，可以提升导航效果

rolling\_window: true #是否使用动态窗口，默认为false，在静态的全局地图中，地图不会变化

width: 3 # 局部地图宽度 单位是 m

height: 3 # 局部地图高度 单位是 m

resolution: 0.05 # 局部地图分辨率 单位是 m，一般与静态地图分辨率保持一致

**4.base\_local\_planner\_params**

基本的局部规划器参数配置，这个配置文件设定了机器人的最大和最小速度限制值，也设定了加速度的阈值。

TrajectoryPlannerROS:

# Robot Configuration Parameters

max\_vel\_x: 0.5 # X 方向最大速度

min\_vel\_x: 0.1 # X 方向最小速速

max\_vel\_theta: 1.0 #

min\_vel\_theta: -1.0

min\_in\_place\_vel\_theta: 1.0

acc\_lim\_x: 1.0 # X 加速限制

acc\_lim\_y: 0.0 # Y 加速限制

acc\_lim\_theta: 0.6 # 角速度加速限制

# Goal Tolerance Parameters，目标公差

xy\_goal\_tolerance: 0.10

yaw\_goal\_tolerance: 0.05

# Differential-drive robot configuration

# 是否是全向移动机器人

holonomic\_robot: false

# Forward Simulation Parameters，前进模拟参数

sim\_time: 0.8

vx\_samples: 18

vtheta\_samples: 20

sim\_granularity: 0.05

**5.3launch文件集成**

如果要实现导航，需要集成地图服务、amcl 、move\_base 等，集成示例如下:

<launch>

<!-- 设置地图的配置文件 -->

<arg name="map" default="nav.yaml" />

<!-- 运行地图服务器，并且加载设置的地图-->

<node name="map\_server" pkg="map\_server" type="map\_server" args="$(find nav)/map/$(arg map)"/>

<!-- 启动AMCL节点 -->

<include file="$(find nav)/launch/amcl.launch" />

<!-- 运行move\_base节点 -->

<include file="$(find nav)/launch/move\_base.launch" />

</launch>

**5.4测试**

1.执行相关launch文件，启动机器人并加载机器人模型：roslaunch mycar\_start start.launch；

2.启动导航相关的 launch 文件：roslaunch nav nav.launch；

3.添加Rviz组件实现导航（参考仿真实现）。

### 6.导航与SLAM建图

**6.1编写launc文件**

当前launch文件（名称自定义，比如：auto\_slam.launch）实现，无需调用map\_server的相关节点，只需要启动SLAM节点与move\_base节点，示例内容如下:

<launch>

<!-- 启动SLAM节点 -->

<include file="$(find nav)/launch/gmapping.launch" />

<!-- 运行move\_base节点 -->

<include file="$(find nav)/launch/move\_base.launch" />

</launch>

**6.2测试**

1.执行相关launch文件，启动机器人并加载机器人模型：roslaunch mycar\_start start.launch；

2.然后执行当前launch文件：roslaunch nav auto\_slam.launch；

3.在rviz中通过2D Nav Goal设置目标点，机器人开始自主移动并建图了；

4.最后可以使用 map\_server 保存地图。

# ROS进阶

## action通信

关于action通信，我们先从之前导航中的应用场景开始介绍，描述如下:

机器人导航到某个目标点,此过程需要一个节点A发布目标信息，然后一个节点B接收到请求并控制移动，最终响应目标达成状态信息。

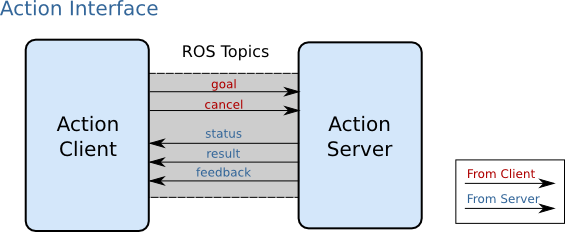
乍一看，这好像是服务通信实现，因为需求中要A发送目标，B执行并返回结果，这是一个典型的基于请求响应的应答模式，不过，如果只是使用基本的服务通信实现，存在一个问题：导航是一个过程，是耗时操作，如果使用服务通信，那么只有在导航结束时，才会产生响应结果，而在导航过程中，节点A是不会获取到任何反馈的，从而可能出现程序"假死"的现象，过程的不可控意味着不良的用户体验，以及逻辑处理的缺陷(比如:导航中止的需求无法实现)。更合理的方案应该是:导航过程中，可以连续反馈当前机器人状态信息，当导航终止时，再返回最终的执行结果。在ROS中，该实现策略称之为:action 通信。

概念：在ROS中提供了actionlib功能包集，用于实现 action 通信。action 是一种类似于服务通信的实现，其实现模型也包含请求和响应，但是不同的是，在请求和响应的过程中，服务端还可以连续的反馈当前任务进度，客户端可以接收连续反馈并且还可以取消任务。

action结构图解:



action通信接口图解:



goal:目标任务;

cacel:取消任务;

status:服务端状态;

result:最终执行结果(只会发布一次);

feedback:连续反馈(可以发布多次)

案例：创建两个ROS 节点，服务器和客户端，客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，为了良好的用户体验，需要服务器在计算过程中，每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

### 1.action通信自定义action文件

action、srv、msg 文件内的可用数据类型一致，且三者实现流程类似:

**1.定义action文件**

首先新建功能包，并导入依赖: roscpp rospy std\_msgs actionlib actionlib\_msgs；

然后功能包下新建 action 目录，新增 Xxx.action(比如:AddInts.action)。

action 文件内容组成分为三部分:请求目标值、最终响应结果、连续反馈，三者之间使用---分割示例内容如下:

#目标值

int32 num

---

#最终结果

int32 result

---

#连续反馈

float64 progress\_bar

1. **编辑配置文件**

**package.xml(不改也没事)**

<build\_depend>actionlib</build\_depend>

<exec\_depend>actionlib</exec\_depend>

<build\_depend>actionlib\_msgs</build\_depend>

<exec\_depend>actionlib\_msgs</exec\_depend>

**CMakeLists.txt**

find\_package

(catkin REQUIRED COMPONENTS

roscpp

rospy

std\_msgs

actionlib

actionlib\_msgs

)

add\_action\_files(

FILES

AddInts.action

)

generate\_messages(

DEPENDENCIES

std\_msgs

actionlib\_msgs

)

catkin\_package(

# INCLUDE\_DIRS include

# LIBRARIES demo\_action

CATKIN\_DEPENDS roscpp rospy std\_msgs actionlib actionlib\_msgs

# DEPENDS system\_lib

)

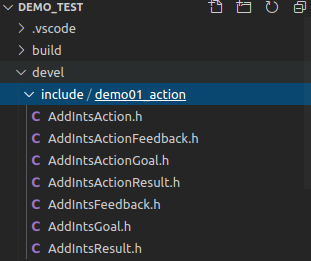
3.编译

编译后会生成一些中间文件。

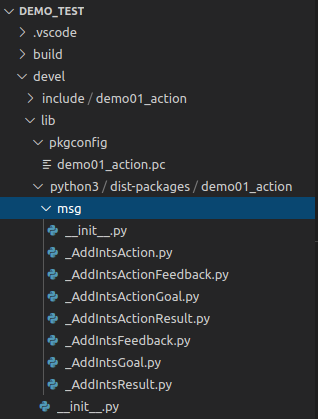
msg文件(.../工作空间/devel/share/包名/msg/xxx.msg):



C++ 调用的文件(.../工作空间/devel/include/包名/xxx.h):



Python 调用的文件(.../工作空间/devel/lib/python3/dist-packages/包名/msg/xxx.py):



### 2.action通信自定义action文件调用A(C++)

需求:创建两个ROS 节点，服务器和客户端，客户端可以向服务器发送目标数据N(一个整型数据)服务器会计算 1 到 N 之间所有整数的和,这是一个循环累加的过程，返回给客户端，这是基于请求响应模式的，又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，为了良好的用户体验，需要服务器在计算过程中，每累加一次，就给客户端响应一次百分比格式的执行进度，使用 action实现。

流程: **0.vscode配置**

需要像之前自定义 msg 实现一样配置c\_cpp\_properies.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/xxx/yyy工作空间/devel/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

**1.服务端**

#include "ros/ros.h"

#include "actionlib/server/simple\_action\_server.h"

#include "demo\_action/addAction.h"

/\*

需求:

创建两个ROS节点，服务器和客户端，

客户端可以向服务器发送目标数据N（一个整型数据）

服务器会计算1到N之间所有整数的和，这是一个循环累加的过程，返回给客户端，

这是基于请求响应模式的，

又已知服务器从接收到请求到产生响应是一个耗时操作，每累加一次耗时0.1s，

为了良好的用户体验，需要服务器在计算过程中，

每累加一次，就给客户端响应一次百分比格式的执行进度，使用action实现。

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action服务对象;

5.处理请求,产生反馈与响应;

6.spin().

\*/

typedef actionlib::SimpleActionServer<demo\_action::addAction> Server;//重定义

void cb(const demo\_action::addGoalConstPtr &goal,Server\* server){//必须是server指针，不能是引用

//获取目标值

int num = goal->num;

ROS\_INFO("目标值:%d",num);

//累加并响应连续反馈

int result = 0;

demo\_action::addFeedback feedback;//连续反馈

ros::Rate rate(10);//通过频率设置休眠时间

for (int i = 1; i <= num; i++)

{

result += i;

//组织连续数据并发布

feedback.progress\_bar = i / (double)num;

server->publishFeedback(feedback);

rate.sleep();

}

//设置最终结果

demo\_action::addResult r;

r.result = result;

server->setSucceeded(r);

ROS\_INFO("最终结果:%d",r.result);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

ROS\_INFO("action服务端实现");

// 2.初始化ROS节点;

ros::init(argc,argv,"add\_server");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action服务对象;

/\*SimpleActionServer(ros::NodeHandle n, 节点名

std::string name, 话题名

boost::function<void (const demo01\_action::AddIntsGoalConstPtr &)> execute\_callback, 回调函数

bool auto\_start)是否自动启动，false时需要调用server.start()启动

\*/

// actionlib::SimpleActionServer<demo\_action::addAction> server(上面的);

Server server(nh,"add",boost::bind(&cb,\_1,&server),false);

server.start();

// 5.处理请求,产生反馈与响应;

// 6.spin().

ros::spin();

return 0;

}

PS:可以先配置CMakeLists.tx文件并启动上述action服务端，然后通过 rostopic 查看话题，向action相关话题发送消息，或订阅action相关话题的消息。

例：rostopic list

rostopic pub /add/goal + 两次tab补齐，修改num

rostopic echo /add/feedback

**2.客户端**

#include "ros/ros.h"

#include "actionlib/client/simple\_action\_client.h"

#include "demo\_action/addAction.h"

/\*

流程:

1.包含头文件;

2.初始化ROS节点;

3.创建NodeHandle;

4.创建action客户端对象;

5.发送目标，处理反馈以及最终结果;

6.spin().

\*/

typedef actionlib::SimpleActionClient<demo\_action::addAction> Client;

//处理最终结果

void done\_cb(const actionlib::SimpleClientGoalState &state, const demo\_action::addResultConstPtr &result){

if (state.state\_ == state.SUCCEEDED)

{

ROS\_INFO("最终结果:%d",result->result);

} else {

ROS\_INFO("任务失败！");

}

}

//服务已经激活

void active\_cb(){

ROS\_INFO("服务已经被激活....");

}

//处理连续反馈

void feedback\_cb(const demo\_action::addFeedbackConstPtr &feedback){

ROS\_INFO("当前进度:%.2f",feedback->progress\_bar);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化ROS节点;

ros::init(argc,argv,"add\_client");

// 3.创建NodeHandle;

ros::NodeHandle nh;

// 4.创建action客户端对象;

// SimpleActionClient(ros::NodeHandle & n, const std::string & name, bool spin\_thread = true)

// actionlib::SimpleActionClient<demo\_action::addAction> client(nh,"add");

Client client(nh,"addInts",true);

//等待服务启动, 只可以使用client.waitForServer();,之前服务中等待启动的另一种方式ros::service::waitForService("add");不适用

client.waitForServer();

// 5.发送目标，处理反馈以及最终结果;

/\*

void sendGoal(const demo01\_action::AddIntsGoal &goal,

boost::function<void (const actionlib::SimpleClientGoalState &state, const demo01\_action::AddIntsResultConstPtr &result)> done\_cb,

boost::function<void ()> active\_cb,

boost::function<void (const demo01\_action::AddIntsFeedbackConstPtr &feedback)> feedback\_cb)

\*/

demo\_action::addGoal goal;

goal.num = 10;

client.sendGoal(goal,&done\_cb,&active\_cb,&feedback\_cb);

// 6.spin().

ros::spin();

return 0;

}

**3.编译配置文件**

add\_executable(action\_server src/action\_server.cpp)

add\_executable(action\_client src/action\_client.cpp)

...

add\_dependencies(action\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

add\_dependencies(action\_client ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

...

target\_link\_libraries(action\_server

${catkin\_LIBRARIES}

)

target\_link\_libraries(action\_client

${catkin\_LIBRARIES}

)

**4.执行**

首先启动 roscore，然后分别启动action服务端与action客户端

### 3.action通信自定义action文件调用(Python)

**0.vscode配置**

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"python.autoComplete.extraPaths": [

"/opt/ros/noetic/lib/python3/dist-packages",

"/xxx/yyy工作空间/devel/lib/python2.7/dist-packages"

]

}

**1.服务端**

#! /usr/bin/env python

# coding=utf-8

import rospy

import actionlib

from demo\_action.msg import \*

class MyActionServer:

def \_\_init\_\_(self):

#SimpleActionServer(name, ActionSpec, execute\_cb=None, auto\_start=True)

self.server = actionlib.SimpleActionServer("add",addAction,self.cb,False)

self.server.start()

rospy.loginfo("服务端启动")

def cb(self,goal):

rospy.loginfo("服务端处理请求:")

#1.解析目标值

num = goal.num

#2.循环累加，连续反馈

rate = rospy.Rate(10)

sum = 0

for i in range(1,num + 1):

# 累加

sum = sum + i

# 计算进度并连续反馈

feedBack = i / num

rospy.loginfo("当前进度:%.2f",feedBack)

feedBack\_obj = addFeedback()

feedBack\_obj.progress\_bar = feedBack

self.server.publish\_feedback(feedBack\_obj)

rate.sleep()

#3.响应最终结果

result = addResult()

result.result = sum

self.server.set\_succeeded(result)

rospy.loginfo("响应结果:%d",sum)

if \_\_name\_\_ == "\_\_main\_\_":

rospy.init\_node("action\_server\_p")

server = MyActionServer()

rospy.spin()

**2.客户端**

#! /usr/bin/env python

# coding=utf-8

import rospy

import actionlib

from demo\_action.msg import \*

def done\_cb(state,result):

if state == actionlib.GoalStatus.SUCCEEDED:

rospy.loginfo("响应结果:%d",result.result)

def active\_cb():

rospy.loginfo("服务被激活....")

def fb\_cb(fb):

rospy.loginfo("当前进度:%.2f",fb.progress\_bar)

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ROS 节点

rospy.init\_node("action\_client\_p")

# 3.创建 action Client 对象

client = actionlib.SimpleActionClient("add",addAction)

# 4.等待服务

client.wait\_for\_server()

# 5.组织目标对象并发送

goal\_obj = addGoal()

goal\_obj.num = 10

client.send\_goal(goal\_obj,done\_cb,active\_cb,fb\_cb)

# 6.编写回调, 激活、连续反馈、最终响应

# 7.spin

rospy.spin()

**3.编辑配置文件**

先为 Python 文件添加可执行权限:chmod +x \*.py

catkin\_install\_python(PROGRAMS

scripts/action\_server\_p.py

scripts/action\_client\_p.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

## 动态参数

参数服务器的数据被修改时，如果节点不重新访问，那么就不能获取修改后的数据，例如在乌龟背景色修改的案例中，先启动乌龟显示节点，然后再修改参数服务器中关于背景色设置的参数，那么窗体的背景色是不会修改的，必须要重启乌龟显示节点才能生效。而一些特殊场景下，是要求要能做到动态获取的，也即，参数一旦修改，能够通知节点参数已经修改并读取修改后的数据，比如：

机器人调试时，需要修改机器人轮廓信息(长宽高)、传感器位姿信息....，如果这些信息存储在参数服务器中，那么意味着需要重启节点，才能使更新设置生效，但是希望修改完毕之后，某些节点能够即时更新这些参数信息。

在ROS中针对这种场景已经给出的解决方案: dynamic reconfigure 动态配置参数。

动态配置参数，之所以能够实现即时更新，因为被设计成 CS 架构，客户端修改参数就是向服务器发送请求，服务器接收到请求之后，读取修改后的是参数。

案例:编写两个节点，一个节点可以动态修改参数，另一个节点时时解析修改后的数据。

### 1.动态参数客户端

**1.新建功能包**

新建功能包，添加依赖:roscpp rospy std\_msgs dynamic\_reconfigure。

**2.添加.cfg文件**

新建 cfg 文件夹，添加 xxx.cfg 文件(并添加可执行权限)，cfg 文件其实就是一个 python 文件,用于生成参数修改的客户端(GUI)。

#! /usr/bin/env python

# coding=utf-8

"""

4生成动态参数 int,double,bool,string,列表

5实现流程:

6 1.导包

7 2.创建生成器

8 3.向生成器添加若干参数

9 4.生成中间文件并退出

10

"""

# 1.导包

from dynamic\_reconfigure.parameter\_generator\_catkin import \*

PACKAGE = "demo\_reconfigure"

# 2.创建生成器

gen = ParameterGenerator()

# 3.向生成器添加若干参数

#add(name, paramtype, level, description, default=None, min=None, max=None, edit\_method="")

gen.add("int\_param",int\_t,0,"整型参数",50,0,100)

gen.add("double\_param",double\_t,0,"浮点参数",1.57,0,3.14)

gen.add("string\_param",str\_t,0,"字符串参数","hello world ")

gen.add("bool\_param",bool\_t,0,"bool参数",True)

many\_enum = gen.enum([gen.const("small",int\_t,0,"a small size"),

gen.const("mediun",int\_t,1,"a medium size"),

gen.const("big",int\_t,2,"a big size")

],"a car size set")

gen.add("list\_param",int\_t,0,"列表参数",0,0,2, edit\_method=many\_enum)

# 4.生成中间文件并退出

exit(gen.generate(PACKAGE,"dr\_node","dr"))

chmod +x xxx.cfg添加权限

**3.配置 CMakeLists.txt**

generate\_dynamic\_reconfigure\_options(

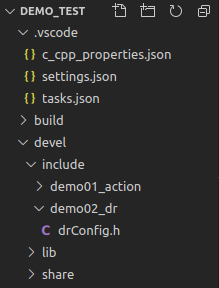
cfg/config.cfg

)

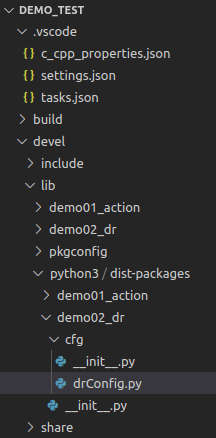
4.编译

编译后会生成中间文件

C++ 需要调用的头文件:



Python需要调用的文件:



### 2.动态参数服务端(C++)

**0.vscode配置**

需要像之前自定义 msg 实现一样配置c\_cpp\_properies.json文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/xxx/yyy工作空间/devel/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

**1.服务器代码实现**

#include "ros/ros.h"

#include "dynamic\_reconfigure/server.h"

#include "demo\_reconfigure/drConfig.h"

/\*

动态参数服务端: 参数被修改时直接打印

实现流程:

1.包含头文件

2.初始化 ros 节点

3.创建服务器对象

4.创建回调对象(使用回调函数，打印修改后的参数)

5.服务器对象调用回调对象

6.spin()

\*/

void cb(demo\_reconfigure::drConfig& config, uint32\_t level){

ROS\_INFO("动态参数解析数据:%d,%.2f,%d,%s,%d",

config.int\_param,

config.double\_param,

config.bool\_param,

config.string\_param.c\_str(),

config.list\_param

);

}

int main(int argc, char \*argv[])

{

setlocale(LC\_ALL,"");

// 2.初始化 ros 节点

ros::init(argc,argv,"dr");

// 3.创建服务器对象

dynamic\_reconfigure::Server<demo\_reconfigure::drConfig> server;

// 4.创建回调对象(使用回调函数，打印修改后的参数)

dynamic\_reconfigure::Server<demo\_reconfigure::drConfig>::CallbackType cbType;

cbType = boost::bind(&cb,\_1,\_2);

// 5.服务器对象调用回调对象

server.setCallback(cbType);

// 6.spin()

ros::spin();

return 0;

}

**2.编译配置文件**

add\_executable(dr\_server src/dr\_server.cpp)

...

add\_dependencies(dr\_server ${${PROJECT\_NAME}\_EXPORTED\_TARGETS} ${catkin\_EXPORTED\_TARGETS})

...

target\_link\_libraries(dr\_server

${catkin\_LIBRARIES}

)

**3.执行**

先启动roscore

启动服务端:rosrun 功能包 xxxx

启动客户端:rosrun rqt\_gui rqt\_gui -s rqt\_reconfigure或rosrun rqt\_reconfigure rqt\_reconfigure

最终可以通过客户端提供的界面修改数据，并且修改完毕后，服务端会即时输出修改后的结果。

PS:ROS版本较新时，可能没有提供客户端相关的功能包导致rosrun rqt\_reconfigure rqt\_reconfigure调用会抛出异常。

### 3.动态参数服务端 (Python)

**0.vscode配置**

需要像之前自定义 msg 实现一样配置settings.json 文件，如果以前已经配置且没有变更工作空间，可以忽略，如果需要配置，配置方式与之前相同:

{

"python.autoComplete.extraPaths": [

"/opt/ros/noetic/lib/python3/dist-packages",

"/xxx/yyy工作空间/devel/lib/python3/dist-packages"

]

}

**1.服务器代码实现**

#! /usr/bin/env python

# coding=utf-8

import rospy

from dynamic\_reconfigure.server import Server

from demo\_reconfigure.cfg import drConfig

"""

动态参数服务端: 参数被修改时直接打印

实现流程:

1.导包

2.初始化 ros 节点

3.创建服务对象

4.回调函数处理

5.spin

"""

# 回调函数

def cb(config,level):

rospy.loginfo("python 动态参数服务解析:%d,%.2f,%d,%s,%d",

config.int\_param,

config.double\_param,

config.bool\_param,

config.string\_param,

config.list\_param

)

return config

if \_\_name\_\_ == "\_\_main\_\_":

# 2.初始化 ros 节点

rospy.init\_node("dr\_p")

# 3.创建服务对象

server = Server(drConfig,cb)

# 4.回调函数处理

# 5.spin

rospy.spin()

**2.编辑配置文件**

先为 Python 文件添加可执行权限:chmod +x \*.py

catkin\_install\_python(PROGRAMS

scripts/dr\_server.py

DESTINATION ${CATKIN\_PACKAGE\_BIN\_DESTINATION}

)

**3.执行**

先启动roscore

启动服务端:rosrun 功能包 xxxx.py

启动客户端:rosrun rqt\_gui rqt\_gui -s rqt\_reconfigure或rosrun rqt\_reconfigure rqt\_reconfigure

## pluginlib

pluginlib直译是插件库，所谓插件字面意思就是可插拔的组件，比如:以计算机为例，可以通过USB接口自由插拔的键盘、鼠标、U盘...都可以看作是插件实现，其基本原理就是通过规范化的USB接口协议实现计算机与USB设备的自由组合。同理，在软件编程中，插件是一种遵循一定规范的应用程序接口编写出来的程序，插件程序依赖于某个应用程序，且应用程序可以与不同的插件程序自由组合。在ROS中，也会经常使用到插件，场景如下:

1.导航插件:在导航中，涉及到路径规划模块，路径规划算法有多种，也可以自实现，导航应用时，可能需要测试不同算法的优劣以选择更合适的实现，这种场景下，ROS中就是通过插件的方式来实现不同算法的灵活切换的。

2.rviz插件:在rviz中已经提供了丰富的功能实现，但是即便如此，特定场景下，开发者可能需要实现某些定制化功能并集成到rviz中，这一集成过程也是基于插件的。

概念

pluginlib是一个c++库， 用来从一个ROS功能包中加载和卸载插件(plugin)。插件是指从运行时库中动态加载的类。通过使用Pluginlib，不必将某个应用程序显式地链接到包含某个类的库，Pluginlib可以随时打开包含类的库，而不需要应用程序事先知道包含类定义的库或者头文件。

**需求:以插件的方式实现正多边形的相关计算。**

**1.准备**

创建功能包xxx导入依赖: roscpp pluginlib。

在 VSCode中需要配置 .vscode/c\_cpp\_properties.json文件中关于 includepath 选项的设置。

{

"configurations": [

{

"browse": {

"databaseFilename": "",

"limitSymbolsToIncludedHeaders": true

},

"includePath": [

"/opt/ros/noetic/include/\*\*",

"/usr/include/\*\*",

"/.../yyy工作空间/功能包/include/\*\*" //配置 head 文件的路径

],

"name": "ROS",

"intelliSenseMode": "gcc-x64",

"compilerPath": "/usr/bin/gcc",

"cStandard": "c11",

"cppStandard": "c++17"

}

],

"version": 4

}

**2.创建基类**

在 xxx/include/xxx下新建C++头文件: polygon\_base.h，所有的插件类都需要继承此基类，内容如下:

#ifndef \_POLYGON\_BASE\_H\_//标识符以头文件名大写命名，前后加\_

#define \_POLYGON\_BASE\_H\_

namespace polygon\_base

{

class RegularPolygon

{

public:

virtual void initialize(double side\_length) = 0;

virtual double area() = 0;

virtual ~RegularPolygon(){}

protected:

RegularPolygon(){}

};

};

#endif

PS:基类必须提供无参构造函数，所以关于多边形的边长没有通过构造函数而是通过单独编写的initialize函数传参。

**3.创建插件**

在 xxx/include/xxx下新建C++头文件:polygon\_plugins.h，内容如下:

#ifndef \_POLYGON\_PLUGINS\_H\_

#define \_POLYGON\_PLUGINS\_H\_

#include <xxx/polygon\_base.h>

#include <cmath>

namespace polygon\_plugins

{

class Triangle : public polygon\_base::RegularPolygon

{

public:

Triangle(){}

void initialize(double side\_length)

{

side\_length\_ = side\_length;

}

double area()

{

return 0.5 \* side\_length\_ \* getHeight();

}

double getHeight()

{

return sqrt((side\_length\_ \* side\_length\_) - ((side\_length\_ / 2) \* (side\_length\_ / 2)));

}

private:

double side\_length\_;

};

class Square : public polygon\_base::RegularPolygon

{

public:

Square(){}

void initialize(double side\_length)

{

side\_length\_ = side\_length;

}

double area()

{

return side\_length\_ \* side\_length\_;

}

private:

double side\_length\_;

};

};

#endif

该文件中创建了正方形与三角形两个衍生类继承基类。

**4.注册插件**

在 src 目录下新建 polygon\_plugins.cpp 文件，内容如下:

//pluginlib 宏，可以注册插件类

#include <pluginlib/class\_list\_macros.h>

#include <xxx/polygon\_base.h>

#include <xxx/polygon\_plugins.h>

//参数1:衍生类 参数2:基类

PLUGINLIB\_EXPORT\_CLASS(polygon\_plugins::Triangle, polygon\_base::RegularPolygon)

PLUGINLIB\_EXPORT\_CLASS(polygon\_plugins::Square, polygon\_base::RegularPolygon)

该文件会将两个衍生类注册为插件。

**5.构建插件库**

在 CMakeLists.txt 文件中设置内容如下:

include\_directories(include ${catkin\_INCLUDE\_DIRS})#自定义的头文件需要,srv/msg/action这类原有的自定义不用

add\_library(polygon\_plugins src/polygon\_plugins.cpp)# 有自定义库（源文件）时需：#声明c++库

至此，可以调用 catkin\_make 编译，编译完成后，在工作空间/devel/lib目录下，会生成相关的 .so 文件。

**6.使插件可用于ROS工具链**

**6.1配置xml**

功能包下新建文件:polygon\_plugins.xml,内容如下:

<!-- 插件库的相对路径 -->

<library path="lib/libpolygon\_plugins">

<!-- type="插件类" base\_class\_type="基类" -->

<class type="polygon\_plugins::Triangle" base\_class\_type="polygon\_base::RegularPolygon">

<!-- 描述信息 -->

<description>This is a triangle plugin.</description>

</class>

<class type="polygon\_plugins::Square" base\_class\_type="polygon\_base::RegularPolygon">

<description>This is a square plugin.</description>

</class>

</library>

**6.2导出插件**

package.xml文件中设置内容如下:

<export>

<xxx plugin="${prefix}/polygon\_plugins.xml" />

</export>

标签<xxx />的名称应与基类所属的功能包名称一致，plugin属性值为上一步中创建的xml文件。

编译后，可以调用rospack plugins --attrib=plugin xxx命令查看配置是否正常，如无异常，会返回 .xml 文件的完整路径，这意味着插件已经正确的集成到了ROS工具链。

**7.使用插件**

src 下新建c++文件:polygon\_loader.cpp，内容如下:

//类加载器相关的头文件

#include <pluginlib/class\_loader.h>

#include <xxx/polygon\_base.h>

int main(int argc, char\*\* argv)

{

//类加载器 -- 参数1:基类功能包名称 参数2:基类全限定名称

pluginlib::ClassLoader<polygon\_base::RegularPolygon> poly\_loader("xxx", "polygon\_base::RegularPolygon");

try

{

//创建插件类实例 -- 参数:插件类全限定名称

boost::shared\_ptr<polygon\_base::RegularPolygon> triangle = poly\_loader.createInstance("polygon\_plugins::Triangle");

triangle->initialize(10.0);

boost::shared\_ptr<polygon\_base::RegularPolygon> square = poly\_loader.createInstance("polygon\_plugins::Square");

square->initialize(10.0);

ROS\_INFO("Triangle area: %.2f", triangle->area());

ROS\_INFO("Square area: %.2f", square->area());

}

catch(pluginlib::PluginlibException& ex)

{

ROS\_ERROR("The plugin failed to load for some reason. Error: %s", ex.what());

}

return 0;

}

**8.执行**

修改CMakeLists.txt文件，内容如下:

add\_executable(polygon\_loader src/polygon\_loader.cpp)

target\_link\_libraries(polygon\_loader ${catkin\_LIBRARIES})

编译然后执行:polygon\_loader

加上add\_dependencies和target\_link\_libraries里面加上库名好像无影响

## nodelet

ROS通信是基于Node(节点)的，Node使用方便、易于扩展，可以满足ROS中大多数应用场景，但是也存在一些局限性，由于一个Node启动之后独占一根进程，不同Node之间数据交互其实是不同进程之间的数据交互，当传输类似于图片、点云的大容量数据时，会出现延时与阻塞的情况，比如：

现在需要编写一个相机驱动，在该驱动中有两个节点实现:其中节点A负责发布原始图像数据，节点B订阅原始图像数据并在图像上标注人脸。如果节点A与节点B仍按照之前实现，两个节点分别对应不同的进程，在两个进程之间传递容量可观图像数据，可能就会出现延时的情况，那么该如何优化呢？

ROS中给出的解决方案是:Nodelet，通过Nodelet可以将多个节点集成进一个进程。

概念

nodelet软件包旨在提供在同一进程中运行多个算法(节点)的方式，不同算法之间通过传递指向数据的指针来代替了数据本身的传输(类似于编程传值与传址的区别)，从而实现零成本的数据拷贝。

nodelet功能包的核心实现也是插件，是对插件的进一步封装:

不同算法被封装进插件类，可以像单独的节点一样运行；

在该功能包中提供插件类实现的基类:Nodelet；

并且提供了加载插件类的类加载器:NodeletLoader。

作用

应用于大容量数据传输的场景，提高节点间的数据交互效率，避免延时与阻塞。

### 1.使用演示

在ROS中内置了nodelet案例

**1.案例简介**

以“ros- [ROS\_DISTRO] -desktop-full”命令安装ROS时，nodelet默认被安装，如未安装，请调用如下命令自行安装:

sudo apt install ros-<<ROS\_DISTRO>>-nodelet-tutorial-math

在该案例中，定义了一个Nodelet插件类:Plus，这个节点可以订阅一个数字，并将订阅到的数字与参数服务器中的 value 参数相加后再发布。

需求:在同一线程中启动两个Plus节点A与B，向A发布一个数字，然后经A处理后，再发布并作为B的输入，最后打印B的输出。

**2.内置案例调用**

**1.启动roscore**

**2.启动manager**

rosrun nodelet nodelet manager \_\_name:=mymanager

\_\_name:= 用于设置管理器名称。

**3.添加nodelet节点**

添加第一个节点:

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n1 \_value:=100

添加第二个节点:

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n2 \_value:=-50 /n2/in:=/n1/out

PS: 解释

rosrun nodelet nodelet load nodelet\_tutorial\_math/Plus mymanager \_\_name:=n1 \_value:=100

rosnode list 查看，nodelet 的节点名称是: /n1；

rostopic list 查看，订阅的话题是: /n1/in，发布的话题是: /n1/out；

rosparam list查看，参数名称是: /n1/value。

rosrun nodelet nodelet standalone nodelet\_tutorial\_math/Plus mymanager \_\_name:=n2 \_value:=-50 /n2/in:=/n1/out

第二个nodelet 与第一个同理；

第二个nodelet 订阅的话题由 /n2/in 重映射为 /n1/out。

优化:也可以将上述实现集成进launch文件:

<launch>

<!-- 设置nodelet管理器 -->

<node pkg="nodelet" type="nodelet" name="mymanager" args="manager" output="screen" />

<!-- 启动节点1，名称为 n1, 参数 /n1/value 为100 -->

<node pkg="nodelet" type="nodelet" name="n1" args="load nodelet\_tutorial\_math/Plus mymanager" output="screen" >

<param name="value" value="100" />

</node>

<!-- 启动节点2，名称为 n2, 参数 /n2/value 为-50 -->

<node pkg="nodelet" type="nodelet" name="n2" args="load nodelet\_tutorial\_math/Plus mymanager" output="screen" >

<param name="value" value="-50" />

<remap from="/n2/in" to="/n1/out" />

</node>

</launch>

**4.执行**

向节点n1发布消息:

rostopic pub -r 10 /n1/in std\_msgs/Float64 "data: 10.0"

打印节点n2发布的消息:

rostopic echo /n2/out

最终输出结果应该是:60。

### 2.nodelet自定义实现

nodelet本质也是插件，实现流程与插件实现流程类似，并且更为简单，不需要自定义接口，也不需要使用类加载器加载插件类。

需求:参考 nodelet 案例，编写 nodelet 插件类，可以订阅输入数据，设置参数，发布订阅数据与参数相加的结果。

**1.准备**

新建功能包，导入依赖: roscpp、nodelet；

**2.创建插件类并注册插件**

#include "nodelet/nodelet.h"

#include "pluginlib/class\_list\_macros.h"

#include "ros/ros.h"

#include "std\_msgs/Float64.h"

namespace nodelet\_demo\_ns {

class MyPlus: public nodelet::Nodelet

{

public:

MyPlus(){

value = 0.0;

}

void onInit(){

//获取 NodeHandle

ros::NodeHandle& nh = getPrivateNodeHandle();

//从参数服务器获取参数

nh.getParam("value",value);

//创建发布与订阅对象

pub = nh.advertise<std\_msgs::Float64>("out",100);

sub = nh.subscribe<std\_msgs::Float64>("in",100,&MyPlus::doCb,this);

}

//回调函数

void doCb(const std\_msgs::Float64::ConstPtr& p){

double num = p->data;

//数据处理

double result = num + value;

std\_msgs::Float64 r;

r.data = result;

//发布

pub.publish(r);

}

private:

ros::Publisher pub;

ros::Subscriber sub;

double value;

};

}

//pluginlib 宏，可以注册插件类

PLUGINLIB\_EXPORT\_CLASS(nodelet\_demo\_ns::MyPlus,nodelet::Nodelet)

**3.构建插件库**

CMakeLists.txt配置如下：

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

add\_library(mynodeletlib

src/myplus.cpp

)

...

target\_link\_libraries(mynodeletlib

${catkin\_LIBRARIES}

)

编译后，会在 工作空间/devel/lib/先生成文件: libmynodeletlib.so。

**4.使插件可用于ROS工具链**

**4.1配置xml**

新建 xml 文件，名称自定义(比如:my\_plus.xml)，内容如下：

<library path="lib/libmynodeletlib">

<class name="demo\_nodelet/MyPlus" type="nodelet\_demo\_ns::MyPlus" base\_class\_type="nodelet::Nodelet" >

<description>hello</description>

</class>

</library>

**4.2导出插件**

<export>

<!-- Other tools can request additional information be placed here -->

<nodelet plugin="${prefix}/my\_plus.xml" />

</export>

**5.执行**

可以通过launch文件执行nodelet，示例内容如下:

<launch>

<node pkg="nodelet" type="nodelet" name="my" args="manager" output="screen" />

<node pkg="nodelet" type="nodelet" name="p1" args="load demo\_nodelet/MyPlus my" output="screen">

<param name="value" value="100" />

<remap from="/p1/out" to="con" />

</node>

<node pkg="nodelet" type="nodelet" name="p2" args="load demo\_nodelet/MyPlus my" output="screen">

<param name="value" value="-50" />

<remap from="/p2/in" to="con" />

</node>

</launch>

运行launch文件，可以参考上一节方式向 p1发布数据，并订阅p2输出的数据，最终运行结果也与上一节类似。