

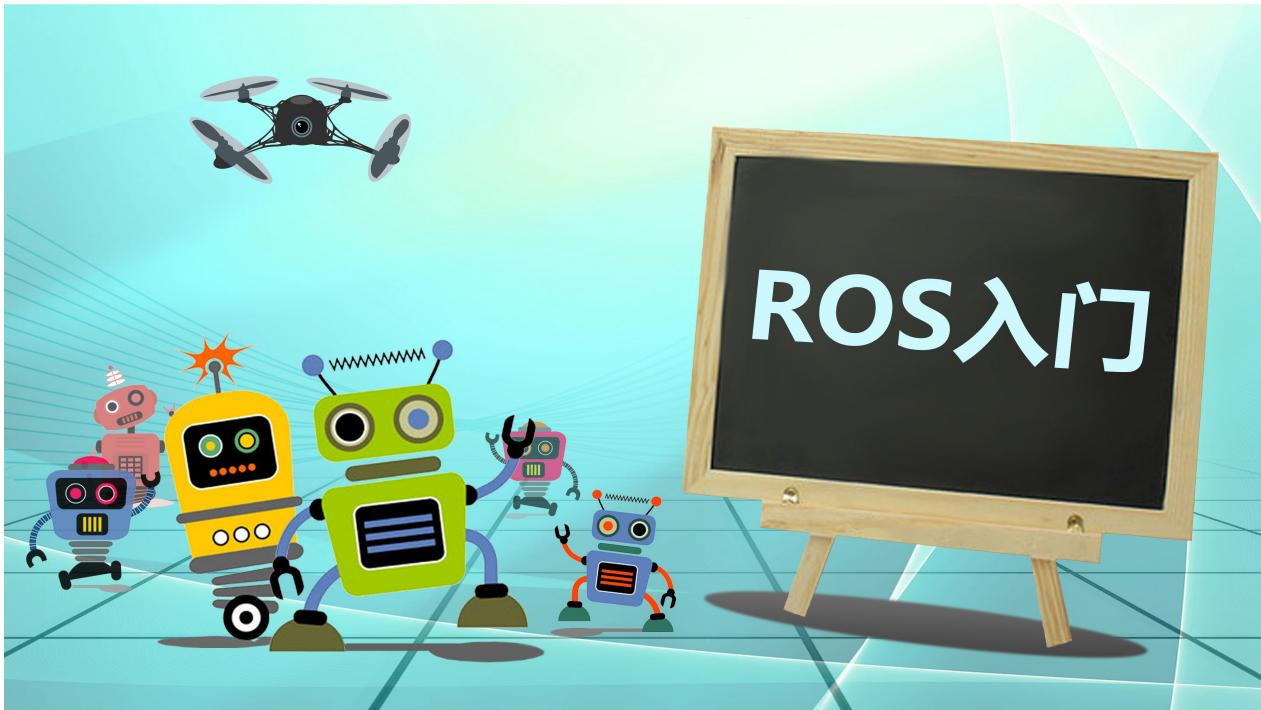
# 目錄

前言	1.1
第一章 ROS简介	1.2
机器人时代的到来	1.2.1
ROS发展历程	1.2.2
什么是ROS	1.2.3
安装ROS	1.2.4
安装ROS-Academy-for-Beginners教学包	1.2.5
二进制与源码包	1.2.6
安装RoboWare Studio	1.2.7
第二章 ROS文件系统	1.3
Catkin编译系统	1.3.1
Catkin工作空间	1.3.2
Package软件包	1.3.3
CMakeLists.txt	1.3.4
package.xml	1.3.5
Metapacakge软件元包	1.3.6
其他常见文件类型	1.3.7
第三章 ROS通信架构（一）	1.4
Node & Master	1.4.1
Launch文件	1.4.2
Topic	1.4.3
Msg	1.4.4
常见msg类型	1.4.5
第四章 ROS通信架构（二）	1.5
Service	1.5.1
Srv	1.5.2
Parameter server	1.5.3
Action	1.5.4
常见srv类型	1.5.5
常见action类型	1.5.6

第五章 常用工具	1.6
Gazebo	1.6.1
RViz	1.6.2
Rqt	1.6.3
Rosbag	1.6.4
Rosbridge	1.6.5
moveit!	1.6.6
第六章 rosCPP	1.7
Client Library与rosCPP	1.7.1
节点初始、关闭与NodeHandle	1.7.2
Topic in rosCPP	1.7.3
Service in rosCPP	1.7.4
Param in rosCPP	1.7.5
时钟	1.7.6
日志与异常	1.7.7
第七章 rospy	1.8
RPyC与主要接口	1.8.1
Topic in rospy	1.8.2
Service in rospy	1.8.3
Param与Time	1.8.4
第八章 TF与URDF	1.9
认识TF	1.9.1
TF消息	1.9.2
tf in c++	1.9.3
tf in python	1.9.4
统一机器人描述格式	1.9.5
第九章 SLAM	1.10
地图	1.10.1
Gmapping	1.10.2
Karto	1.10.3
Hector	1.10.4
第十章 Navigation	1.11
Navigation Stack	1.11.1
move_base	1.11.2

costmap	1.11.3
Map_server & Amcl	1.11.4
附录：TF数学基础	1.12
三维空间刚体运动---旋转矩阵	1.12.1
三维空间刚体运动---欧拉角	1.12.2
三维空间刚体运动---四元数	1.12.3
附录：Navigation工具包说明	1.13
amcl	1.13.1
local_base_planner	1.13.2
carrot_planner	1.13.3
clear_costmap_recovery	1.13.4
costmap_2d	1.13.5
dwa_local_planner	1.13.6
fake_localization	1.13.7
global_planner	1.13.8
map_server	1.13.9
move_base_msg	1.13.10
move_base	1.13.11
move_slow_and_clear	1.13.12
navfn	1.13.13
nav_core	1.13.14
robot_pose_ekf	1.13.15
rotate_recovery	1.13.16

# 中国大学MOOC---《机器人操作系统入门》课程讲义



## 前言

欢迎来到中国大学MOOC---《机器人操作系统入门》课程，本书是课程的配套讲义，由中科院软件所和重德智能公司共同推出，课程分为十个章节，与视频课程相对应，介绍ROS的基本原理和用法。欢迎各位朋友在中国大学MOOC平台上选修这门课程。

本讲义希望带给读者最准确、生动、易懂的ROS入门指导，为了方便读者操作练习，我们同时附有配套的教学代码示例，欢迎各位下载、学习和分享。

本讲义将长期维护，永久公开。如果你有任何问题，可在Github的issues里提出，也可以直接联系我们的邮箱。由于我们课程团队的精力实在有限，本书难免存在错误和瑕疵，欢迎各位朋友拍砖，我们将不断更新，一直努力。

## 出品



## 作者

- 柴长坤 [chaichangkun@droid.ac.cn](mailto:chaichangkun@droid.ac.cn)
- 武延军 [yanjun@iscas.ac.cn](mailto:yanjun@iscas.ac.cn)
- 常先明 [c xm@droid.ac.cn](mailto:cxm@droid.ac.cn)

## 致谢

- 安传旭 [1179735683@qq.com](mailto:1179735683@qq.com)
- 韩昊旻 [hanhaomin008@126.com](mailto:hanhaomin008@126.com)
- 韩锦飞 [hanjf\\_robin@163.com](mailto:hanjf_robin@163.com)

# 第一章 ROS简介

## 本章简介

机器人操作系统(Robot Operating System, ROS)是一个应用于机器人上的操作系统，它操作方便、功能强大，特别适用于机器人这种多节点多任务的复杂场景。因此自ROS诞生以来，受到了学术界和工业界的欢迎，如今已经广泛应用于机械臂、移动底盘、无人机、无人车等多种类的机器人上。

本章介绍ROS的产生、发展、特点和安装方法，带给你一个简单直观的ROS介绍。

## 1.1 机器人时代的到来

### 机器人时代与ROS诞生

他们速度很快，具有非凡智慧与致命力量；他们不受生命周期桎梏，是未来科技的希望。机器人和仿生机器，正要从实验室的襁褓中迸裂而出，投入市场的广阔天地，运算能力的大幅进步加上最新的机器人脑部设计，为机器人科技带来前所未有的改革，毋庸置疑，机器人即将改变人类工作场所及生活状态。神爱世人，人亦钟情所造之物，如今有幸跟随时代步伐，一起创造世界，借不灭之躯，走遍险远之地，看尽世之奇伟瑰怪。

曾经，机器人创新的门槛非常高。如果你想在任何应用领域开发出有分量的产品，你需要建立一整套能够实现你想法的系统：包括硬件设备，当然还有控制系统，界面接口，以及让机器人运行并作为测试平台的检测工具。“没有什么是现成的设备，除了一些很差，闭源的东西外。”



随着机器人领域的快速发展和复杂化，代码复用和模块化的需求日益强烈，已有的开源系统已不能很好地适应需求，2010年Willow Garage公司发布了开源机器人操作系统ROS。

# 1.2 ROS发展历程

## 1.2.1 ROS起源与发展

本世纪开始，关于人工智能的研究进入了大发展阶段，包括全方位的具体的AI，例如斯坦福大学人工智能实验室STAIR（Stanford Artificial Intelligence Robot）项目，该项目组创建了灵活的、动态的软件系统的原型，用于机器人技术。在2007年，机器人公司Willow Garage和该项目组合作，他们十分具有前瞻性的，提供了大量资源进一步扩展了这些概念，经过具体的研究测试实现之后，无数的研究人员将他们的专业性研究贡献到ROS核心概念和其基础软件包，这期间积累了众多的科学研究成果。ROS软件的开发自始至终采用开放的BSD协议，在机器人技术研究领域逐渐成为一个被广泛使用的平台。

Willow Garage公司和斯坦福大学人工智能实验室合作以后，在2009年初推出了ROS0.4，这是一个测试版的ROS，现在所用的系统框架在这个版本中已经具有了初步的雏形。之后的版本才正式开启了ROS的发展成熟之路。

## 1.2.2 历代ROS版本

ROS1.0版本发布于2010年，基于PR2机器人开发了一系列机器人相关的基础软件包。随后ROS版本迭代频繁，目前已经发布到了Lunar。目前使用人数最多的是Kinetic和Indigo这两个Long Term Support版本。

ROS版本	发布时间
Lunar Loggerhead	2017.5
Kinetic Kame	2016.5
Jade Turtle	2015.5
Indigo Igloo	2014.7
Hydro Medusa	2013.9
Groovy Galapagos	2012.12
Fuerte Turtle	2012.4
Electric Emys	2011.8
Diamondback	2011.3
C Turtle	2010.8
Box Turtle	2010.3



### 1.2.3 展望

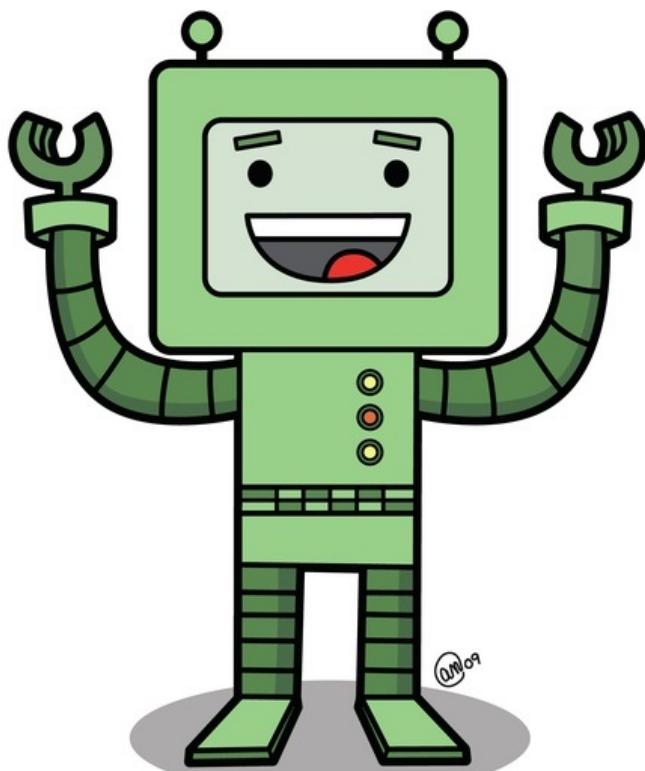
ROS的发展逐渐的趋于成熟，近年来也逐步是面对着Ubuntu的更新而更新，这说明ROS已经初步进入一种稳定的发展状态，每年进行一次更新的频率同时还保留着长期支持的版本，这使得ROS在稳步的前进发展同时，也有着开拓创新的方向。目前越来越多的机器人、无人机甚至无人车都开始采用ROS作为开发平台，尽管ROS在实用方面目前还存在一些限制，但前途非常光明。

2018年ROS2 1.0版将发布，未来ROS2将如何影响机器人领域，我们拭目以待。相信在人工智能的大发展、人机交互越来越密切、互联网+的大时代，ROS会发挥出越来越重要的作用。

## 1.3 什么是ROS

### 1.3.1 什么是ROS

机器人是一个系统工程，它涉及机械、电子、控制、通信、软件等诸多学科。以前，开发一个机器人需要花很大的功夫，你需要设计机械、画电路板、写驱动程序、设计通信架构、组装集成、调试、以及编写各种感知决策和控制算法，每一个任务都需要花费大量的时间。因此像电影《钢铁侠》中那样，仅靠一个人的力量造出一个动力超强的人形机器人机甲是不可能的。



然而随着技术进步，机器人产业分工开始走向细致化、多层次化，如今的电机、底盘、激光雷达、摄像头、机械臂等等元器件都有不同厂家专门生产。社会分工加速了机器人行业的发展。而各个部件的集成就需要一个统一的软件平台，在机器人领域，这个平台就是机器人操作系统ROS。

ROS是一个适用于机器人编程的框架，这个框架把原本松散的零部件耦合在了一起，为他们提供了通信架构。ROS虽然叫做操作系统，但并非Windows、Mac那样通常意义的操作系统，它只是连接了操作系统和你开发的ROS应用程序，所以它也算是一个中间件，基于ROS的应用程序之间建立起了沟通的桥梁，所以也是运行在Linux上的运行时环境，在这个环境上，机器人的感知、决策、控制算法可以更好的组织和运行。

以上几个关键词（框架、中间件、操作系统、运行时环境）都可以用来描述ROS的特性，作为初学者我们不必深究这些概念，随着你越来越多的使用ROS，就能够体会到它的作用。

## 1.3.2 ROS特点

ROS具有这些特点：

- 分布式点对点

ROS采用了分布式的框架，通过点对点的设计让机器人的进程可以分别运行，便于模块化的修改和定制，提高了系统的容错能力。

- 多种语言支持

ROS支持多种编程语言。C++、Python和已经在ROS中实现编译，是目前应用最广的ROS开发语言，Lisp、C#、Java等语言的测试库也已经实现。为了支持多语言编程，ROS采用了一种语言中立的接口定义语言来实现各模块之间消息传送。通俗的理解就是，ROS的通信格式和用哪种编程语言来写无关，它使用的是自身定义的一套通信接口。

- 开源社区

ROS具有一个庞大的社区ROS WIKI(<http://wiki.ros.org/>)，这个网站将会始终伴随着你ROS开发，无论是查阅功能包的参数、搜索问题还是。当前使用ROS开发的软件包已经达到数千万个，相关的机器人已经多达上千款。此外，ROS遵从BSD协议，对个人和商业应用及修改完全免费。这也促进了ROS的流行。

## 1.3.3 ROS优缺点

ROS为我们开发机器人带来了许多方便，然而它也确实存在一些问题：

优点	缺点
提供框架、工具和功能	通信实时性能有限
方便移植	系统稳定性尚不满足工业级要求
庞大的用户群体	安全性上没有防护措施
免费开源	仅支持Linux(Ubuntu)

总体来说，ROS更适合科研和开源用户使用，如果在工业场景应用（例如无人驾驶）还需要做优化和定制。为了解决实际应用的问题，ROS2.0做了很大的改进，目前正在开发之中，未来表现如何值得期待。



## 1.4 安装ROS的步骤

### 1.4.1 ROS版本选择

ROS目前只支持在Linux系统上安装部署，它的首选开发平台是Ubuntu。时至今日ROS已经相继更新推出了多种版本，供不同版本的Ubuntu开发者使用。为了提供最稳定的开发环境，ROS的每个版本都有一个推荐运行的Ubuntu版本。如下表所示：

ROS 版本	首选Ubuntu版本
Lunar	Ubuntu 17.04
<b>Kinetic(建议选用)</b>	<b>Ubuntu 16.04</b>
Jade	Ubuntu 15.04
Indigo	Ubuntu 14.04
...	...

本教程使用的平台是**Ubuntu 16.04**，ROS版本是**Kinetic**。

如果你还没有安装Ubuntu，建议选择16.04版本(<https://www.ubuntu.com/download/desktop>)。并且我们建议在本地安装，不推荐用虚拟机，这样兼容性更好。

如果你已经安装Ubuntu，请确定系统版本，在终端中输入 `cat /etc/issue` 确定Ubuntu版本号，然后选择对应的ROS版本。如果没有安装正确的ROS版本，就会出现各种各样的依赖错误，所以安装时请谨慎。

更多信息请参考[ROS官方网站](#)进行下载和安装。

### 1.4.2 安装ROS

在正式的安装前，先检查下Ubuntu初始环境是否配置正确。

打开 Ubuntu的设置 -> 软件与更新 -> Ubuntu软件 -> 勾选关键字 `universe` , `restricted` , `multiverse` 三项。如图所示:



配置完成后，就可以开始安装ROS了，打开终端。

## ① 添加sources.list

```
$ sudo sh -c ' . /etc/lsb-release && echo "deb http://mirrors.ustc.edu.cn/ros/ubuntu/ $DISTRIB_CODENAME main" > /etc/apt/sources.list.d/ros-latest.list'
```

这一步配置将镜像添加到Ubuntu系统源列表中，建议使用国内或镜像源，这样能够保证下载速度。本例使用的是中国科技大学的源。

## ② 添加keys

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

公钥是Ubuntu系统的一种安全机制，也是ROS安装中不可缺的一部分。

## ③ 系统更新

```
$ sudo apt-get update && sudo apt-get upgrade
```

更新系统，确保自己的Debian软件包和索引是最新的。

## ④ 安装ROS

ROS中有很多函数库和工具，官网提供了四种默认的安装方式，当然也可以单独安装某个特定的软件包。这四种方式包括桌面完整版安装、桌面版安装，基础版安装、单独软件包安装。推荐安装桌面完整版安装（包含ROS、rqt、rviz、通用机器人函数库、2D/3D仿真器、导航以及2D/3D感知功能），如下：

- Ubuntu 16.04安装Kinetic版本

```
$ sudo apt-get install ros-kinetic-desktop-full # Ubuntu 16.04
```

- Ubuntu 14.04安装Indigo版本

```
$ sudo apt-get install ros-indigo-desktop-full # Ubuntu 14.04
```

如果你不想安装桌面完整版，你还可以尝试以下三种方式安装：

- 桌面版安装（包含ROS、rqt、rviz以及通用机器人函数库）

```
sudo apt-get install ros-kinetic-desktop
```

- 基础版安装（包含ROS核心软件包、构建工具以及通信相关的程序库，无GUI工具）

```
sudo apt-get install ros-kinetic-ros-base
```

- 单独软件包安装（这种安装方式在运行ROS缺少某些**package**依赖时会经常用到。你可以安装某个指定的**ROS**软件包，使用软件包名称替换掉下面的**PACKAGE**）

```
sudo apt-get install ros-kinetic-PACKAGE
```

例如系统提示找不到slam-gmapping，你就可以：

```
sudo apt-get install ros-kinetic-slam-gmapping
```

要查找可用的软件包，请运行：

```
apt-cache search ros-kinetic
```

软件包的依赖问题还可能出现在重复安装ROS、错误安装软件包的过程中，出现有一些软件包无法安装，例如：

```
下列软件包有未满足的依赖关系：ros-kinetic-desktop-full：
 依賴: ros-kinetic-desktop 但是它将不会被安装；
 依賴: ros-kinetic-perception 但是它将不会被安装；
 依賴: ros-kinetic-simulators 但是它将不会被安装；
E: 无法修正错误，因为您要求某些软件包保持现状，就是它们破坏了软件包间的依赖关系。
```

出现上述问题，有可能是自己的版本不合适不兼容造成，也可能是镜像源没有更新，具体的设置参考软件和更新的截图。当然也有可能是其他原因，比如更新了忘记刷新环境source一下，重开一个终端等等。具体的问题原因可以去搜索引擎上尝试求助解决，或者登陆[ROS Wiki](#)(ROS的百科全书)去查询解决自己的具体问题。

## 1.4.3 配置ROS

配置ROS是安装完ROS之后必须的工作。

### ① 初始化rosdep

```
$ sudo rosdep init && rosdep update
```

这一步初始化rosdep，是使用ROS之前的必要一步。rosdep可以方便在你需要编译某些源码的时候为其安装一些系统依赖，同时也是某些ROS核心功能组件所必需用到的工具。

### ② ROS环境配置

```
#For Ubuntu 16.04
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

#For Ubuntu 14.04
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
```

注意：ROS的环境配置，使得你每次打开一个新的终端，ROS的环境变量都能够自动配置好，也就是添加到bash会话中，因为命令 `source /opt/ros/kinetic/setup.bash` 只在当前终端有作用，即具有单一时效性，要想每次新开一个终端都不用重新配置环境，就用echo语句将命令添加到bash会话中。

### ③ 安装rosinstall

[rosinstall](#) 是ROS中一个独立分开的常用命令行工具，它可以方便让你通过一条命令就可以给某个ROS软件包下载很多源码树。在ubuntu上安装这个工具，请运行：

```
$ sudo apt-get install python-rosinstall
```

至此，ROS的安装就结束了，下面测试ROS能否正常运行。

## 1.4.4 测试ROS

首先启动ROS，输入代码运行roscore：

```
$ roscore
```

如果出现下图所示，那么说明ROS正常启动了！

```
changkun@changkun-pc:~/catkin_ws/src/sensor_stick/scripts$ roscore
... logging to /home/changkun/.ros/log/39dce55c-7823-11e7-bc8d-1c1b0d6c4a7e/roslaunch-changkun-pc-12473.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://changkun-pc:40977/
ros_comm version 1.12.7

SUMMARY
=====

PARAMETERS
* /rosdistro: kinetic
* /rosversion: 1.12.7

NODES

auto-starting new master
process[master]: started with pid [12484]
ROS_MASTER_URI=http://changkun-pc:11311/

setting /run_id to 39dce55c-7823-11e7-bc8d-1c1b0d6c4a7e
process[rosout-1]: started with pid [12497]
started core service [/rosout]
```

接着我们测试ROS的吉祥物--小海龟，来简单的测试ROS运行是否正常，同时也来体验一下ROS的神奇与精彩！

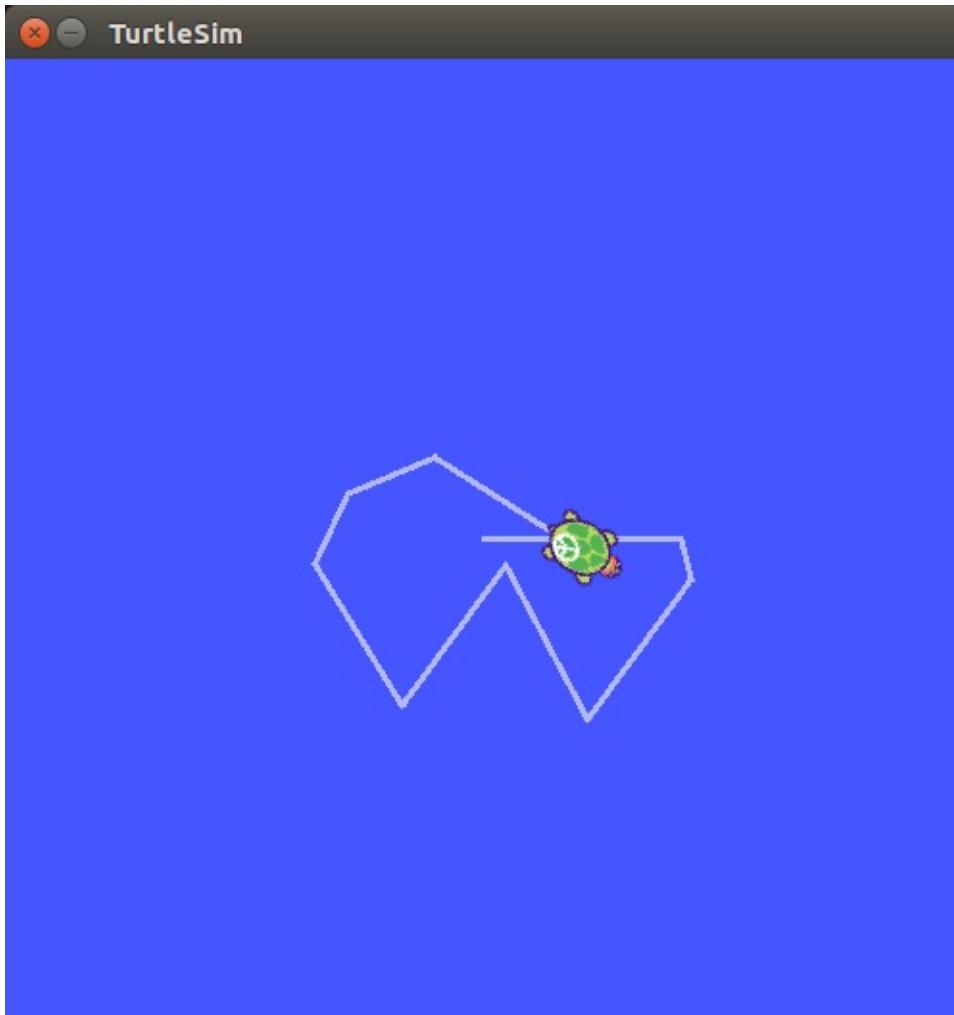
启动roscore后，重新打开一个终端窗口，输入：

```
$ rosrun turtlesim turtlesim_node
```

你还看到一只萌萌的海龟出现在屏幕上，那么该怎么样来操纵这只小海龟呢？重新打开新的一个终端，输入：

```
$ rosrun turtlesim turtle_teleop_key
```

将鼠标聚焦在第三个终端窗口上，然后通过键盘上的方向键，进行操作小海龟，如果小海龟正常移动，并且在屏幕上留下自己的移动轨迹，如下图。恭喜你，ROS已经成功的安装、配置并且运行！



至此，ROS的安装、配置与测试就全部结束了，下面就正式开启ROS精彩的旅程！

# 1.5 安装ROS-Academy-for-Beginners教学包

在1.4节我们已经用apt工具安装好了ROS，apt安装的软件包都是二进制形式，可以在系统中直接运行，它们是ROS官方提供给用户的应用程序。然而很多时候我们需要自己做一些定制改写，或者运行第三方开发的软件包，这个时候就需要下载源代码进行编译。

本节我们下载本书配套的ROS-Academy-for-Beginners软件包，给读者演示源码包下载-编译-运行的完整流程。后续章节的主要代码都基于这个软件包，请读者按照流程下载编译。

## 下载源码包

在Ubuntu系统上，确保git已经安装

```
$ sudo apt-get install git
```

然后在创建一个名为tutorial\_ws的工作空间，在它的src路径下克隆ROS-Academy-for-Beginners软件包

```
$ cd  
$ mkdir -p tutorial_ws/src      # 创建catkin工作空间  
$ cd tutorial_ws/src            # 进入src路径，克隆教学软件包  
$ git clone https://github.com/DroidAITech/ROS-Academy-for-Beginners.git
```

## 安装依赖

并且安装ROS-Academy-for-Beginners所需要的依赖

```
$ cd ~/tutorial_ws  
$ rosdep install --from-paths src --ignore-src --rosdistro=kinetic -y
```

注意：以上命令非常重要，缺少依赖将导致软件包无法正常编译和运行。

在开始编译之前，需要确保Gazebo在7.0版本以上

```
$ gazebo -v  # 确认7.0及以上
```

如果你的Gazebo版本低于7.0，则需要进行升级

```
$ 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 -
$ sudo apt-get update
$ sudo apt-get install gazebo7
```

## 编译

接着回到catkin\_ws下编译

```
$ cd ~/tutorial_ws
$ catkin_make
$ source ~/tutorial_ws/devel/setup.bash      #刷新环境 方法一
$ rospack profile #刷新环境 方法二
```

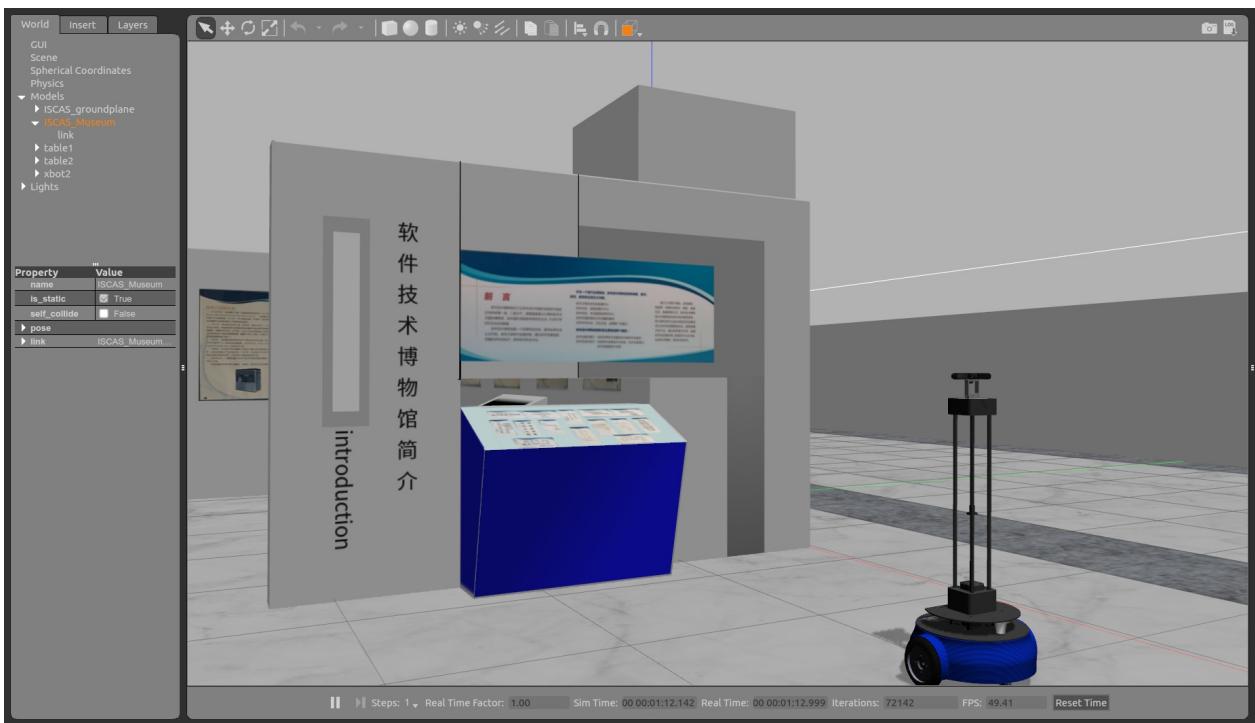
注意:source命令，编译完成后必须刷新一下工作空间的环境，否则可能找不到工作空间。许多时候我们为了打开终端就能够运行工作空间中编译好的ROS程序，我们习惯把 source ~/tutorial\_ws/devel/setup.bash 命令追加到 ~/.bashrc 文件中(rosacademy\_ws替换为你的工作空间名称)，这样每次打开终端，系统就会刷新工作空间环境。你可以通过 echo "source ~/tutorial\_ws/devel/setup.bash" >> ~/.bashrc 命令来追加。

## 运行仿真程序

编译完成后就可以运行本教学配套的仿真了，输入

```
$ rospack profile
$ roslaunch robot_sim_demo robot_spawn.launch
```

你会看到仿真画面启动，仿真界面中包括了软件博物馆和Xbot机器人模型。

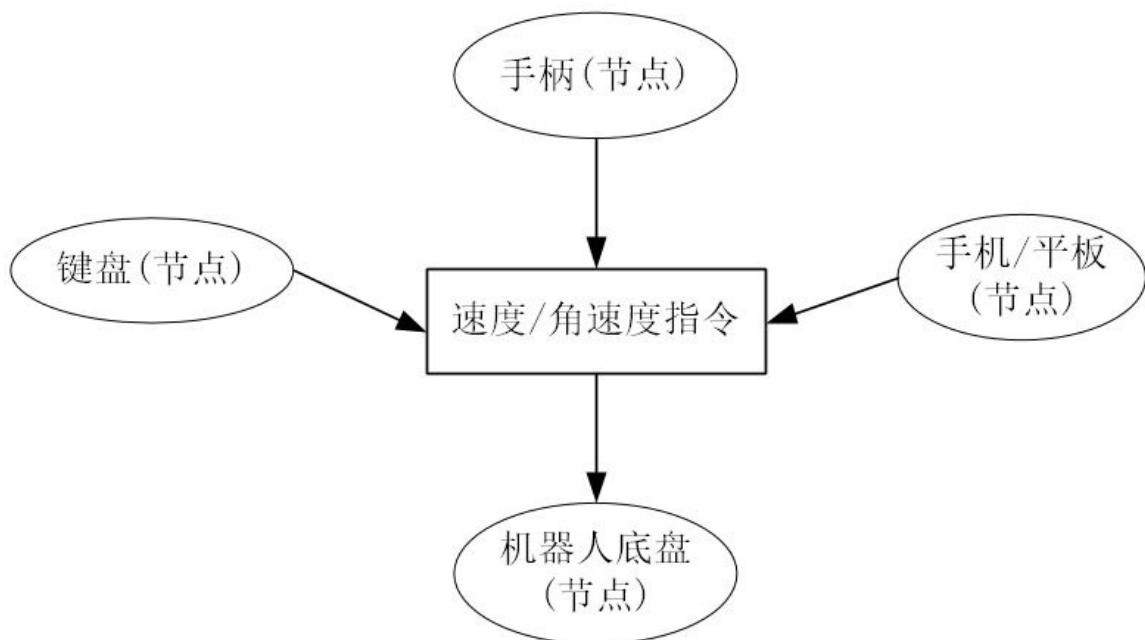


再打开一个新的终端，输入以下命令，用键盘控制机器人移动

```
$ rosrun robot_sim_demo robot_keyboard_teleop.py
```

聚焦控制程序窗口，按下i、j、l等按键，这样你就可以通过键盘来控制机器人的移动了。

当你完成了这一步，首先恭喜你，你已经完成了ROS最常见的源码下载-安装依赖-编译-运行的流程，在ROS社区有许许多多这样的软件包，基本都按照这样的流程来运行。相信你一定可以举一反三。



键盘控制仿真机器人移动这个demo展现了ROS分布式消息收发的特性。我们打开的虽然是键盘控制程序，但它可以替换为手柄控制、手机平板控制、甚至是路径规划自动控制。模拟器里的机器人并不关心是谁发给它的消息，它只关心这个消息是什么（速度、角速度等指令）。所以，每一个进程（节点）都各司其职，负责不同的任务，同时定义好消息收发的接口。如果我们现在要做路径规划任务，那么我们只用单独再开发一个节点，同样向底盘发送我们解算出的速度/角速度指令就可以了。你现在可能对ROS工作方式还一无所知，不过不要紧。后续我们会对ROS涉及的这些概念进行详细介绍，等你看完了这本书，你就能明白整个ROS框架运行的原理，甚至自己能编程实现一些功能模块了。

# 1.6 二进制包 vs. 源代码包

## 1.6.1 二进制包与源代码包

在1.4节我们通过apt方式安装了ROS系统以及相关的软件包，而在1.5节我们通过下载源码编译的方式安装了一个ROS教学软件包。这是两种常见的软件包安装方式，通常我们的软件包(Package)就可以分为二进制和源代码。

二进制包里面包括了已经编译完成，可以直接运行的程序。你通过 `sudo apt-get install` 来进行下载和解包（安装），执行完该指令后就可以马上使用了。因此这种方式简单快捷，适合比较固定、无需改动的程序。

而源代码包里是程序的原始代码，在你的计算机上必须经过编译，生成了可执行的二进制文件，方可运行。一些个人开发的程序、第三方修改或者你希望修改的程序都应当通过源代码包的来编译安装。

区别	二进制包	源代码包
下载方式	<code>apt-get install</code> /直接下载 <code>deb</code>	<code>git clone</code> /直接下载源代码
ROS包存放位置	<code>/opt/ros/kinetic/</code>	通常 <code>~/catkin_ws/src</code>
编译方式	无需编译	通过 <code>make/cmake/catkin</code>
来源	官方apt软件源	开源项目、第三方开发者
扩展性	无法修改	通过源代码修改
可读性	无法查看源代码	方便阅读源代码
优点	下载简单，安装方便	源码可修改，便于定制功能
缺点	无法修改	编译工具、软件包依赖、版本和参数
应用场景	基础软件	需要查看、开发和修改的程序

在1.4中，我们用apt-get安装了ROS及其组件，因此我们不需要编译就可以运行turtlesim程序。对于这些程序，除非我们做操作系统的设计开发才会去下载源码，否则直接用官方提供的ROS软件包；而在1.5中，ROS-Academy-for-Beginners以源码呈现，你可以看到每个demo下面的C++代码。对于这些源文件我们必须 `catkin_make` 编译，然后才能运行。

## 1.6.2 ROS二进制包的安装

在ROS中，我们可能经常会遇到缺少相关的ROS依赖的问题。有些时候你编译或者运行一些ROS程序，系统会提示找不到XXX功能包，如图所示。

遇到这样的问题，请先注意阅读错误原因，看看是否有解决方法，也可以Google一下。如果是缺少ROS的依赖，通常可以用以下命令来安装：

```
$ sudo apt-get install ros-kinetic-PACKAGE
```

将PACKAGE替换为系统提示缺啥少的软件包，例如

```
$ sudo apt-get install ros-kinetic-slam-gmapping    #GMapping-SLAM算法包  
$ sudo apt-get install ros-kinetic-turtlebot-description  #Turtlebot机器人模型包
```

所有APT官方中的ROS功能包都是按照ros--的形式来命名。

## 1.7 安装RoboWare Studio

通常ROS的程序都是用C++和Python开发的，为了提高开发的效率，我们建议用IDE来写代码。目前在Ubuntu上已经有许多IDE支持ROS开发，比如Eclipse、Qt Creator。不过这些IDE配置起来会比较麻烦，我们推荐一款适配ROS的IDE——RoboWare Studio来开发ROS。

读者可在<http://cn.roboware.me/#/Download> 下载和使用。

## 第二章 ROS文件系统

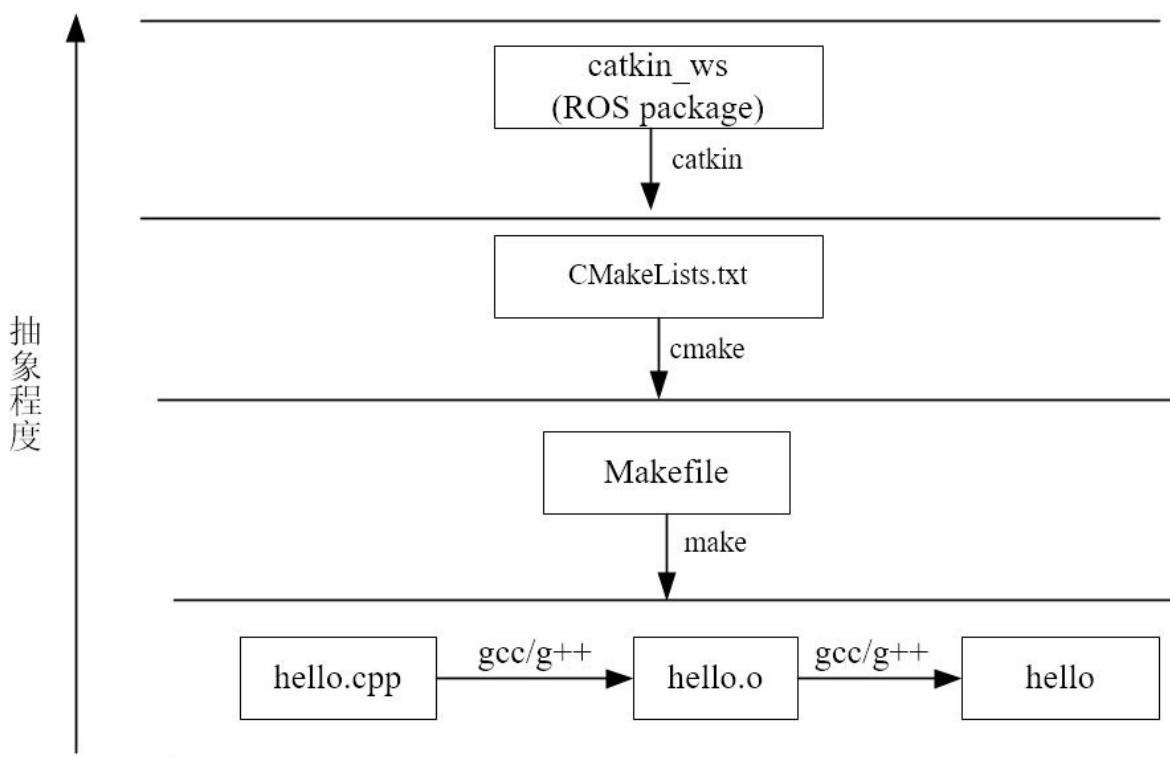
### 本章简介

本章主要介绍了ROS的工程结构，也就是ROS的文件系统结构。要学会建立一个ROS工程，首先要认识一个ROS工程，了解它们的组织架构，从根本上熟悉ROS项目的组织形式，了解各个文件的功能和作用，才能正确的进行开发和编程。

本章的主要内容有，介绍catkin的编译系统，catkin工作空间的创建和结构，package软件包的创建和结构，介绍CMakeLists.txt文件，package.xml以及其他常见文件。从而系统的梳理了ROS文件空间的结构，对于我们ROS学习和开发有着重要的作用。

## 2.1 Catkin编译系统

对于源代码包，我们只有编译才能在系统上运行。而Linux下的编译器有gcc、g++，随着源文件的增加，直接用gcc/g++命令的方式显得效率低下，人们开始用Makefile来进行编译。然而随着工程体量的增大，Makefile也不能满足需求，于是便出现了Cmake工具。CMake是对make工具的生成器，是更高层的工具，它简化了编译构建过程，能够管理大型项目，具有良好的扩展性。对于ROS这样大体量的平台来说，就采用的是CMake，并且ROS对CMake进行了扩展，于是便有了Catkin编译系统。



早期的ROS编译系统是rosbuild，但随着ROS的不断发展，rosbuild逐渐暴露出许多缺点，不能很好满足系统需要。在Groovy版本面世后，Catkin作为rosbuild的替代品被正式投入使用。Catkin操作更加简化且工作效率更高，可移植性更好，而且支持交叉编译和更加合理的功能包分配。目前的ROS同时支持着rosbuild和Catkin两种编译系统，但ROS的核心软件包也已经全部转换为Catkin。rosbuild已经被逐步淘汰，所以建议初学者直接上手Catkin。

本节我们主要来介绍catkin的编译系统。

### 2.1.1 Catkin特点

Catkin是基于CMake的编译构建系统，具有以下特点：

- Catkin沿用了包管理的传统像 `find_package()` 基础结构, `pkg-config`
- 扩展了CMake, 例如
  - 软件包编译后无需安装就可使用
  - 自动生成 `find_package()` 代码, `pkg-config` 文件
  - 解决了多个软件包构建顺序问题

一个Catkin的软件包（package）必须要包括两个文件：

- `package.xml`: 包括了package的描述信息
  - name, description, version, maintainer(s), license
  - opt. authors, url's, dependencies, plugins, etc...
- `CMakeLists.txt`: 构建package所需的CMake文件
  - 调用Catkin的函数/宏
  - 解析 `package.xml`
  - 找到其他依赖的catkin软件包
  - 将本软件包添加到环境变量

## 2.1.2 Catkin工作原理

catkin编译的工作流程如下：

1. 首先在工作空间 `catkin_ws/src/` 下递归的查找其中每一个ROS的package。
2. package中会有 `package.xml` 和 `CMakeLists.txt` 文件, Catkin(CMake)编译系统依据 `CMakeLists.txt` 文件,从而生成 `makefiles` (放在 `catkin_ws/build/` )。
3. 然后 `make` 刚刚生成的 `makefiles` 等文件, 编译链接生成可执行文件(放在 `catkin_ws/devel` )。

也就是说, Catkin就是将 `cmake` 与 `make` 指令做了一个封装从而完成整个编译过程的工具。catkin有比较突出的优点,主要是：

- 操作更加简单
- 一次配置,多次使用
- 跨依赖项目编译

## 2.1.3 使用 `catkin_make` 进行编译

要用catkin编译一个工程或软件包,只需要用 `catkin_make` 指令。一般当我们写完代码,执行一次 `catkin_make` 进行编译,调用系统自动完成编译和链接过程,构建生成目标文件。编译的一般性流程如下,在1.5节我们编译ROS-Academy-for-Beginners教学包就是这样的流程。

```
$ cd ~/catkin_ws #回到工作空间, catkin_make必须在工作空间下执行
$ catkin_make      #开始编译
$ source ~/catkin_ws/devel/setup.bash #刷新环境
```

注意: catkin编译之前需要回到工作空间目录, catkin\_make 在其他路径下编译不会成功。编译完成后,如果有新的目标文件产生(原来没有),那么一般紧跟着要source刷新环境,使得系统能够找到刚才编译生成的ROS可执行文件。这个细节比较容易遗漏,致使后面出现可执行文件无法打开等错误。

catkin\_make 命令也有一些可选参数,例如:

```
catkin_make [args]
-h, --help          帮助信息
-C DIRECTORY, --directory DIRECTORY
                   工作空间的路径(默认为'.')
--source SOURCE    src的路径(默认为'workspace_base/src')
--build BUILD      build的路径(默认为'workspace_base/build')
--use-ninja        用ninja取代make
--use-nmake        用nmake取'make'
--force-cmake      强制cmake,即使已经cmake过
--no-color         禁止彩色输出(只对catkin_make和CMake生效)
--pkg PKG [...]    只对某个PKG进行make
--only-pkg-with-deps ONLY_PKG_WITH_DEPS [ONLY_PKG_WITH_DEPS ...]
                   将指定的package列入白名单CATKIN_WHITELIST_PACKAGES,
                   之编译白名单里的package。该环境变量存在于CMakeCache.txt。
--cmake-args [CMAKE_ARGS [CMAKE_ARGS ...]]
                   传给CMake的参数
--make-args [MAKE_ARGS [MAKE_ARGS ...]]
                   传给Make的参数
--override-build-tool-check
                   用来覆盖由于不同编译工具产生的错误
```

## 2.2 Catkin工作空间

Catkin工作空间是创建、修改、编译catkin软件包的目录。catkin的工作空间，直观的形容就是一个仓库，里面装载着ROS的各种项目工程，便于系统组织管理调用。在可视化图形界面里是一个文件夹。我们自己写的ROS代码通常就放在工作空间中，本节就来介绍catkin工作空间的结构。

### 2.2.1 初始化catkin工作空间

介绍完catkin编译系统，我们来建立一个catkin的工作空间。首先我们要在计算机上创建一个初始的 `catkin_ws/` 路径，这也是catkin工作空间结构的最高层级。输入下列指令，完成初始创建。

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make #初始化工作空间
```

第一行代码直接创建了第二层级的文件夹src，这也是我们放ROS软件包的地方。第二行代码使得进程进入工作空间，然后再是`catkin_make`。

注意：1. `catkin_make`命令必须在工作空间这个路径上执行 2.原先的初始化命令`catkin_init_workspace`仍然保留

### 2.2.2 结构介绍

catkin的结构十分清晰，具体的catkin工作空间结构图如下。初看起来catkin工作空间看起来极其复杂，其实不然，catkin工作空间的结构其实非常清晰。

在工作空间下用`tree`命令，显示文件结构。

```
$ cd ~/catkin_ws
$ sudo apt install tree
$ tree
```

结果为：

```

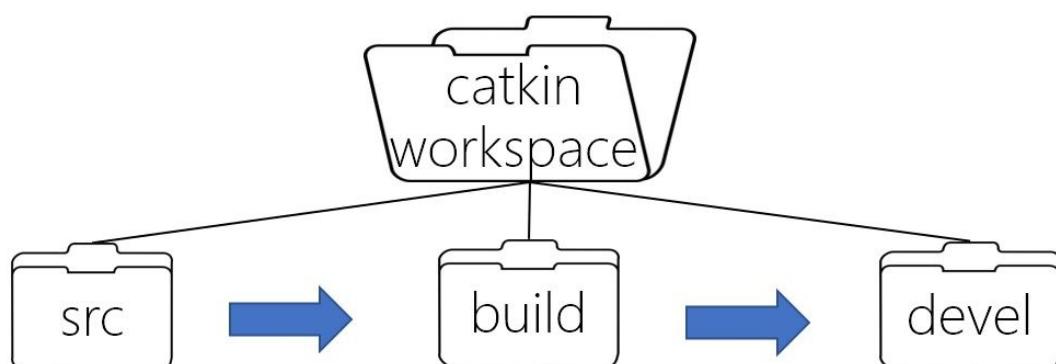
- build
|   └── catkin
|       └── catkin_generated
|           └── version
|               └── package.cmake
|
|   .....
|
|   └── catkin_make.cache
|   └── CMakeCache.txt
|   └── CMakeFiles
|       └──
|
|   .....
|
└── devel
    ├── env.sh
    ├── lib
    ├── setup.bash
    ├── setup.sh
    ├── _setup_util.py
    └── setup.zsh
└── src
    └── CMakeLists.txt -> /opt/ros/kinetic/share/catkin/cmake/toplevel.cmake

```

通过tree命令可以看到catkin工作空间的结构,它包括了 `src` 、 `build` 、 `devel` 三个路径,在有些编译选项下也可能包括其他。但这三个文件夹是catkin编译系统默认的。它们的具体作用如下:

- `src/`: ROS的catkin软件包 (源代码包)
- `build/`: catkin (CMake) 的缓存信息和中间文件
- `devel/`: 生成的目标文件 (包括头文件, 动态链接库, 静态链接库, 可执行文件等) 、环境变量

在编译过程中,它们的工作流程如图:



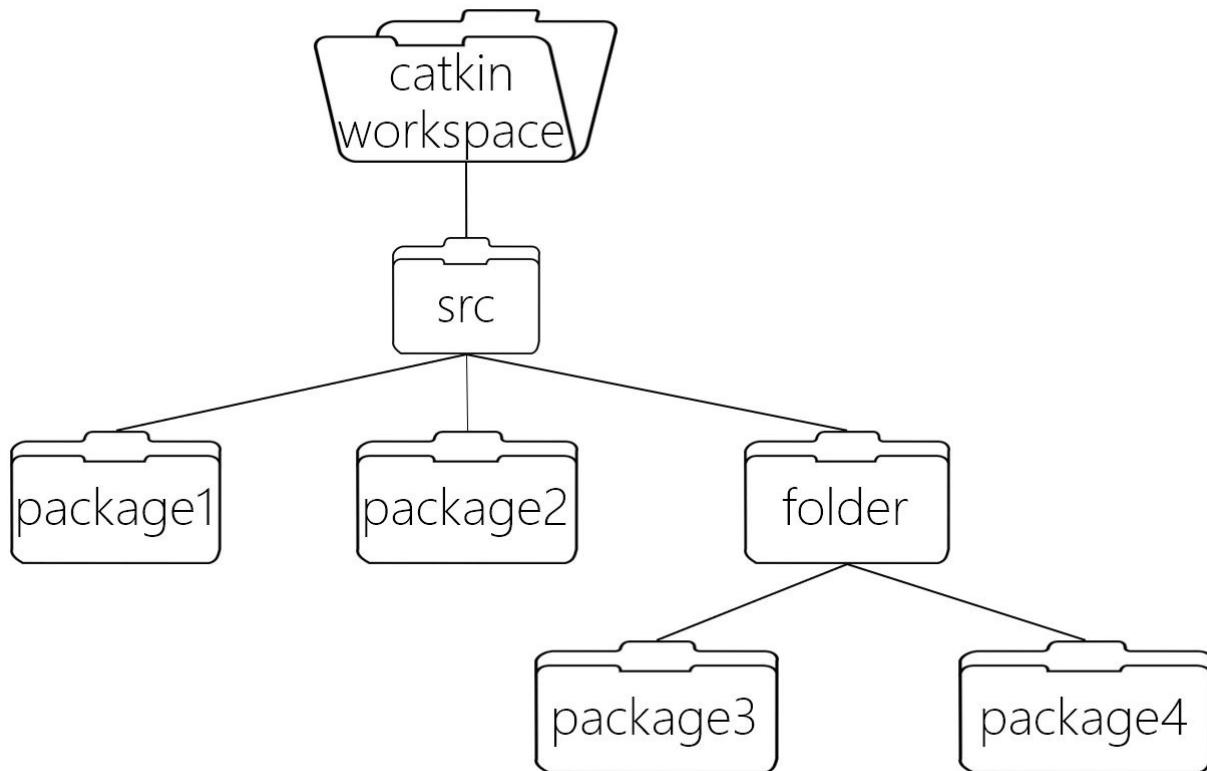
package源代码包

cmake&catkin缓存和中间文件

目标文件

后两个路径由catkin系统自动生成、管理，我们日常的开发一般不会去涉及，而主要用到的是src文件夹，我们写的ROS程序、网上下载的ROS源代码包都存放在这里。

在编译时，catkin编译系统会递归的查找和编译 src/ 下的每一个源代码包。因此你也可以把几个源代码包放到同一个文件夹下，如下图所示：



## 小结

catkin工作空间基本就是以上的结构，package是catkin工作空间的基本单元，我们在ROS开发时，写好代码，然后catkin\_make，系统就会完成所有编译构建的工作。至于更详细的package内容，我们将在下节继续介绍。

## 2.3 Package软件包

在1.6节我们曾对package软件包进行了分类，分别介绍了二进制包和源代码包。而ROS中的package的定义更加具体，它不仅是Linux上的软件包，更是catkin编译的基本单元，我们调用 `catkin_make` 编译的对象就是一个个ROS的package，也就是说任何ROS程序只有组织成 package 才能编译。所以package也是ROS源代码存放的地方，任何ROS的代码无论是C++还是Python都要放到package中，这样才能正常的编译和运行。一个package可以编译出来多个目标文件（ROS可执行程序、动态静态库、头文件等等）。

### 2.3.1 package结构

一个package下常见的文件、路径有：

```

├── CMakeLists.txt      #package的编译规则(必须)
├── package.xml         #package的描述信息(必须)
└── src/                #源代码文件
    ├── include/          #C++头文件
    ├── scripts/          #可执行脚本
    ├── msg/               #自定义消息
    ├── srv/               #自定义服务
    ├── models/             #3D模型文件
    ├── urdf/              #urdf文件
    ├── launch/             #launch文件
    └── include/            #C++头文件

```

其中定义package的是 `CMakeLists.txt` 和 `package.xml`，这两个文件是package中必不可少的。catkin编译系统在编译前，首先就要解析这两个文件。这两个文件就定义了一个package。

- `CMakeLists.txt`: 定义package的包名、依赖、源文件、目标文件等编译规则，是package不可少的成分
- `package.xml`: 描述package的包名、版本号、作者、依赖等信息，是package不可少的成分
- `src/`: 存放ROS的源代码，包括C++的源码和(.cpp)以及Python的module(.py)
- `include/`: 存放C++源码对应的头文件
- `scripts/`: 存放可执行脚本，例如shell脚本(.sh)、Python脚本(.py)
- `msg/`: 存放自定义格式的消息(.msg)
- `srv/`: 存放自定义格式的服务(.srv)
- `models/`: 存放机器人或仿真场景的3D模型(.sda, .stl, .dae等)
- `urdf/`: 存放机器人的模型描述(.urdf或.xacro)

- `launch/`: 存放launch文件(.launch或.xml)

通常ROS文件组织都是按照以上的形式，这是约定俗成的命名习惯，建议遵守。以上路径中，只有 `CMakeLists.txt` 和 `package.xml` 是必须的，其余路径根据软件包是否需要来决定。

## 2.3.2 package的创建

创建一个package需要在 `catkin_ws/src` 下,用到 `catkin_create_pkg` 命令，用法是：

```
catkin_create_pkg package depends
```

其中 `package` 是包名，`depends` 是依赖的包名，可以依赖多个软件包。

例如，新建一个package叫做 `test_pkg` ,依赖`roscpp`、`rospy`、`std_msgs`(常用依赖)。

```
$ catkin_create_pkg test_pkg roscpp rospy std_msgs
```

这样就会在当前路径下新建 `test_pkg` 软件包，包括：

```
└── CMakeLists.txt
└── include
    └── test_pkg
└── package.xml
└── src
```

`catkin_create_pkg` 帮你完成了软件包的初始化，填充好了 `CMakeLists.txt` 和 `package.xml` ，并且将依赖项填进了这两个文件中。

## 2.3.3 package相关命令

### rospack

rospack是对package管理的工具，命令的用法如下：

rostopic命令	作用
<code>rospack help</code>	显示rospack的用法
<code>rospack list</code>	列出本机所有package
<code>rospack depends [package]</code>	显示package的依赖包
<code>rospack find [package]</code>	定位某个package
<code>rospack profile</code>	刷新所有package的位置记录

以上命令如果package缺省，则默认为当前目录(如果当前目录包含`package.xml`)

## roscd

`roscd` 命令类似与Linux系统的 `cd`，改进之处在于 `roscd` 可以直接 `cd` 到ROS的软件包。

<b>rostopic</b> 命令	作用
<code>roscd [pacakge]</code>	<code>cd</code> 到ROS package所在路径

## rosls

`rosls` 也可以视为Linux指令 `ls` 的改进版，可以直接 `ls` ROS软件包的内容。

<b>rosls</b> 命令	作用
<code>rosls [pacakge]</code>	列出pacakge下的文件

## rosdep

`rosdep` 是用于管理ROS package依赖项的命令行工具，用法如下：

<b>rosdep</b> 命令	作用
<code>rosdep check [pacakge]</code>	检查package的依赖是否满足
<code>rosdep install [pacakge]</code>	安装pacakge的依赖
<code>rosdep db</code>	生成和显示依赖数据库
<code>rosdep init</code>	初始化/etc/ros/rosdep中的源
<code>rosdep keys</code>	检查package的依赖是否满足
<code>rosdep update</code>	更新本地的rosdep数据库

一个较常使用的命令是 `rosdep install --from-paths src --ignore-src --rosdistro=kinetic -y` ,用于安装工作空间中 `src` 路径下所有package的依赖项（由pacakge.xml文件指定）。

## 2.4 CMakeLists.txt

### 2.4.1 CMakeLists.txt作用

`CMakeLists.txt` 原本是Cmake编译系统的规则文件，而Catkin编译系统基本沿用了CMake的编译风格，只是针对ROS工程添加了一些宏定义。所以在写法上，catkin的 `CMakeLists.txt` 与CMake的基本一致。

这个文件直接规定了这个package要依赖哪些package，要编译生成哪些目标，如何编译等等流程。所以 `CMakeLists.txt` 非常重要，它指定了由源码到目标文件的规则，catkin编译系统在工作时首先会找到每个package下的 `CMakeLists.txt`，然后按照规则来编译构建。

### 2.4.1 CMakeLists.txt写法

`CMakeLists.txt` 的基本语法都还是按照CMake，而Catkin在其中加入了少量的宏，总体的结构如下：

```
cmake_minimum_required() #CMake的版本号
project() #项目名称
find_package() #找到编译需要的其他CMake/Catkin package
catkin_python_setup() #catkin新加宏，打开catkin的Python Module的支持
add_message_files() #catkin新加宏，添加自定义Message/Service/Action文件
add_service_files()
add_action_files()
generate_message() #catkin新加宏，生成不同语言版本的msg/srv/action接口
catkin_package() #catkin新加宏，生成当前package的cmake配置，供依赖本包的其他软件包调用
add_library() #生成库
add_executable() #生成可执行二进制文件
add_dependencies() #定义目标文件依赖于其他目标文件，确保其他目标已被构建
target_link_libraries() #链接
catkin_add_gtest() #catkin新加宏，生成测试
install() #安装至本机
```

如果你从未接触过CMake的语法，请阅读《CMake实践》：<https://github.com/Akagi201/learning-cmake/blob/master/docs/cmake-practice.pdf>。掌握CMake语法对于理解ROS工程很有帮助。

### 2.4.2 CMakeLists例子

为了详细的解释 CMakeLists.txt 的写法，我们以 turtlesim 小海龟这个 pacakge 为例，读者可 roscd 到 turtlesim 包下查看，在 turtlesim/CMakeLists.txt 的写法如下：

```

cmake_minimum_required(VERSION 2.8.3)
#CMake至少为2.8.3版

project(turtlesim)
#项目(package)名称为turtlesim，在后续文件中可使用变量${PROJECT_NAME}来引用项目名称turltesim

find_package(catkin REQUIRED COMPONENTS geometry_msgs message_generation roscpp roscpp_serialization roslib rostime std_msgs std_srvs)
#cmake宏，指定依赖的其他pacakge，实际是生成了一些环境变量，如<NAME>_FOUND, <NAME>_INCLUDE_DIRS
, <NAME>_LIBRARYS
#此处catkin是必备依赖 其余的geometry_msgs...为组件

find_package(Qt5Widgets REQUIRED)
find_package(Boost REQUIRED COMPONENTS thread)

include_directories(include ${catkin_INCLUDE_DIRS} ${Boost_INCLUDE_DIRS})
#指定C++的头文件路径
link_directories(${catkin_LIBRARY_DIRS})
#指定链接库的路径

add_message_files(DIRECTORY msg FILES
Color.msg Pose.msg)
#自定义msg文件

add_service_files(DIRECTORY srv FILES
Kill.srv
SetPen.srv
Spawn.srv
TeleportAbsolute.srv
TeleportRelative.srv)
#自定义srv文件

generate_messages(DEPENDENCIES geometry_msgs std_msgs std_srvs)
#在add_message_files、add_service_files宏之后必须加上这句话，用于生成srv msg头文件/module，生成的文件位于devel/include中

catkin_package(CATKIN_DEPENDS geometry_msgs message_runtime std_msgs std_srvs)
# catkin宏命令，用于配置ROS的package配置文件和CMake文件
# 这个命令必须在add_library()或者add_executable()之前调用，该函数有5个可选参数：
# (1) INCLUDE_DIRS - 导出包的include路径
# (2) LIBRARIES - 导出项目中的库
# (3) CATKIN_DEPENDS - 该项目依赖的其他catkin项目
# (4) DEPENDS - 该项目所依赖的非catkin CMake项目。
# (5) CFG_EXTRAS - 其他配置选项

set(turtlesim_node_SRCS
src/turtlesim.cpp
src/turtle.cpp
src/turtle_frame.cpp

```

```
)  
set(turtlesim_node_HDRS  
include/turtlesim/turtle_frame.h  
)  
#指定turtlesim_node_SRCS、turtlesim_node_HDRS变量  
  
qt5_wrap_cpp(turtlesim_node_MOCS ${turtlesim_node_HDRS})  
  
add_executable(turtlesim_node ${turtlesim_node_SRCS} ${turtlesim_node_MOCS})  
# 指定可执行文件目标turtlesim_node  
target_link_libraries(turtlesim_node Qt5::Widgets ${catkin_LIBRARIES} ${Boost_LIBRARIES})  
# 指定链接可执行文件  
add_dependencies(turtlesim_node turtlesim_gencpp)  
  
add_executable(turtle_teleop_key tutorials/teleop_turtle_key.cpp)  
target_link_libraries(turtle_teleop_key ${catkin_LIBRARIES})  
add_dependencies(turtle_teleop_key turtlesim_gencpp)  
  
add_executable(draw_square tutorials/draw_square.cpp)  
target_link_libraries(draw_square ${catkin_LIBRARIES} ${Boost_LIBRARIES})  
add_dependencies(draw_square turtlesim_gencpp)  
  
add_executable(mimic tutorials/mimic.cpp)  
target_link_libraries(mimic ${catkin_LIBRARIES})  
add_dependencies(mimic turtlesim_gencpp)  
# 同样指定可执行目标、链接、依赖  
  
install(TARGETS turtlesim_node turtle_teleop_key draw_square mimic  
RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})  
# 安装目标文件到本地系统  
  
install(DIRECTORY images  
DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}  
FILES_MATCHING PATTERN "*.png" PATTERN "*.svg")
```

## 2.5 package.xml

`package.xml` 也是一个catkin的`package`必备文件，它是这个软件包的描述文件，在较早的ROS版本(`rosbuild`编译系统)中，这个文件叫做 `manifest.xml`，用于描述`pacakge`的基本信息。如果你在网上看到一些ROS项目里包含着 `manifest.xml`，那么它多半是hydro版本之前的项目了。

### 2.5.1 package.xml作用

`pacakge.xml` 包含了`package`的名称、版本号、内容描述、维护人员、软件许可、编译构建工具、编译依赖、运行依赖等信息。

实际上 `rospack find`、`rosdep` 等命令之所以能快速定位和分析出`package`的依赖项信息，就是直接读取了每一个`pacakge`中的 `package.xml` 文件。它为用户提供了快速了解一个`pacakge`的渠道。

### 2.5.2 package.xml写法

`pacakge.xml` 遵循xml标签文本的写法，由于版本更迭原因，现在有两种格式并存（format1与format2），不过区别不大。老版本（format1）的 `pacakge.xml` 通常包含以下标签：

<code>&lt;pacakge&gt;</code>	根标记文件
<code>&lt;name&gt;</code>	包名
<code>&lt;version&gt;</code>	版本号
<code>&lt;description&gt;</code>	内容描述
<code>&lt;maintainer&gt;</code>	维护者
<code>&lt;license&gt;</code>	软件许可证
<code>&lt;buildtool_depend&gt;</code>	编译构建工具，通常为catkin
<code>&lt;build_depend&gt;</code>	编译依赖项，与Catkin中的
<code>&lt;run_depend&gt;</code>	运行依赖项

说明：其中1-6为必备标签，1是根标签，嵌套了其余的所有标签，2-6为包的各种属性，7-9为编译相关信息。

在新版本（format2）中，包含的标签为：

<pacakge>	根标记文件
<name>	包名
<version>	版本号
<description>	内容描述
<maintainer>	维护者
<license>	软件许可证
<buildtool_depend>	编译构建工具，通常为catkin
<depend>	指定依赖项为编译、导出、运行需要的依赖，最常用
<build_depend>	编译依赖项
<build_export_depend>	导出依赖项
<exec_depend>	运行依赖项
<test_depend>	测试用例依赖项
<doc_depend>	文档依赖项

由此看见新版本的 `pacakge.xml` 格式上增加了、、、，相当于将之前的build和run依赖项描述进行了细分。

目前Indigo、Kinetic、Lunar等版本的ROS都同时支持两种版本的 `package.xml`，所以无论选哪种格式都可以。

### 2.5.3 pacakge.xml例子

为了说明pacakge.xml写法，还是以turtlesim软件包为例，其 `pacakge.xml` 文件内容如下，我们添加了相关的注释：

```

<?xml version="1.0"?>           <!--本示例为老版本的pacakge.xml-->
<package>                     <!--pacakge为根标签，写在最外面-->
  <name>turtlesim</name>
  <version>0.8.1</version>
  <description>
    turtlesim is a tool made for teaching ROS and ROS packages.
  </description>
  <maintainer email="dthomas@osrfoundation.org">Dirk Thomas</maintainer>
  <license>BSD</license>

  <url type="website">http://www.ros.org/wiki/turtlesim</url>
  <url type="bugtracker">https://github.com/ros/ros_tutorials/issues</url>
  <url type="repository">https://github.com/ros/ros_tutorials</url>
  <author>Josh Faust</author>

  <!--编译工具为catkin-->
  <buildtool_depend>catkin</buildtool_depend>

  <!--编译时需要依赖以下包-->
  <build_depend>geometry_msgs</build_depend>
  <build_depend>qtbase5-dev</build_depend>
  <build_depend>message_generation</build_depend>
  <build_depend>qt5-qmake</build_depend>
  <build_depend>rosconsole</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>roscpp_serialization</build_depend>
  <build_depend>roslib</build_depend>
  <build_depend>rostime</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>std_srvs</build_depend>

  <!--运行时需要依赖以下包-->
  <run_depend>geometry_msgs</run_depend>
  <run_depend>libqt5-core</run_depend>
  <run_depend>libqt5-gui</run_depend>
  <run_depend>message_runtime</run_depend>
  <run_depend>rosconsole</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>roscpp_serialization</run_depend>
  <run_depend>roslib</run_depend>
  <run_depend>rostime</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>std_srvs</run_depend>
</package>

```

以上内容是老版本（format1）的写法，如果要写成新版本（format2）则可以改为：

```
<?xml version="1.0"?>
<package format="2">      <!--在声明pacakge时指定format2，为新版格式-->
<name>turtlesim</name>
<version>0.8.1</version>
<description>
    turtlesim is a tool made for teaching ROS and ROS packages.
</description>
<maintainer email="dthomas@osrfoundation.org">Dirk Thomas</maintainer>
<license>BSD</license>

<url type="website">http://www.ros.org/wiki/turtlesim</url>
<url type="bugtracker">https://github.com/ros/ros_tutorials/issues</url>
<url type="repository">https://github.com/ros/ros_tutorials</url>
<author>Josh Faust</author>

<!--编译工具为catkin-->
<buildtool_depend>catkin</buildtool_depend>

<!--用depend来整合build_depend和run_depend-->
<depend>geometry_msgs</depend>
<depend>roconsole</depend>
<depend>roscpp</depend>
<depend>roscpp_serialization</depend>
<depend>roslib</depend>
<depend>rostime</depend>
<depend>std_msgs</depend>
<depend>std_srvs</depend>

<!--build_depend标签未变-->
<build_depend>qtbase5-dev</build_depend>
<build_depend>message_generation</build_depend>
<build_depend>qt5-qmake</build_depend>

<!--run_depend要改为exec_depend-->
<exec_depend>libqt5-core</exec_depend>
<exec_depend>libqt5-gui</exec_depend>
<exec_depend>message_runtime</exec_depend>
</package>
```

## 2.6 Metapackage

### 2.6.1 Metapackage介绍

在一些ROS的教学资料和博客里，你可能还会看到一个Stack（功能包集）的概念，它指的是将多个功能接近、甚至相互依赖的软件包放到一个集合中去。但Stack这个概念在Hydro之后就取消了，取而代之的就是Metapackage。尽管换了个马甲，但它的作用没变，都是把一些相近的功能模块、软件包放到一起。

ROS里常见的Metapacakge有：

Metapacakge 名称	描述	链接
navigation	导航相关的功能包集	<a href="https://github.com/ros-planning/navigation">https://github.com/ros-planning/navigation</a>
moveit	运动规划相关的（主要是机械臂）功能包集	<a href="https://github.com/ros-planning/moveit">https://github.com/ros-planning/moveit</a>
image_pipeline	图像获取、处理相关的功能包集	<a href="https://github.com/ros-perception/image_common">https://github.com/ros-perception/image_common</a>
vision_opencv	ROS与OpenCV交互的功能包集	<a href="https://github.com/ros-perception/vision_opencv">https://github.com/ros-perception/vision_opencv</a>
turtlebot	Turtlebot机器人相关的功能包集	<a href="https://github.com/turtlebot/turtlebot">https://github.com/turtlebot/turtlebot</a>
pr2_robot	pr2机器人驱动功能包集	<a href="https://github.com/PR2/pr2_robot">https://github.com/PR2/pr2_robot</a>
...	...	...

以上列举了一些常见的功能包集，例如navigation、turtlebot，他们都是用于某一方面的功能，以navigation metapackage（官方介绍里仍然沿用stack的叫法）为例，它包括了以下软件包：

包名	功能
navigation	Metapacakge，依赖以下所有pacakge
amcl	定位
fake_localization	定位
map_server	提供地图
move_base	路径规划节点
nav_core	路径规划的接口类
base_local_planner	局部规划
dwa_local_planner	局部规划
...	...
	...

具体功能介绍，我们留到第九章，这里只看一个软件包navigation。这个navigation就是一个简单的pacakge，里面只有几个文件，但由于它依赖了其他所有的软件包。Catkin编译系统会明白，这些软件包都属于navigation metapacakge。

这个道理并不难理解，比如我们在安装ROS时，用到了 `sudo apt-get install ros-kinetic-desktop-full` 命令，由于它依赖了ROS所有的核心组件，我们在安装时也就能够安装整个ROS。

## 2.6.2 Metapackage写法

我们以ROS-Academy-for-beginners为例介绍meteapckage的写法，在教学包内，有一个 `ros_academy_for_beginners` 软件包，该包即为一个metapacakge，其中有且仅有两个文件：`CMakeLists.txt` 和 `pacakge.xml`。

`CMakeLists.txt` 写法如下：

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_academy_for_beginners)
find_package(catkin REQUIRED)
catkin_metapackage() #声明本软件包是一个metapacakge
```

`pacakge.xml` 写法如下：

```

<package>
  <name>ros_academy_for_beginners</name>
  <version>17.12.4</version>
  <description>
    -----
    A ROS tutorial for beginner level learners. This metapacakge includes some
    demos of topic, service, parameter server, tf, urdf, navigation, SLAM...
    It tries to explain the basic concepts and usages of ROS.
  -----
  </description>
  <maintainer email="chaichangkun@163.com">Chai Changkun</maintainer>
  <author>Chai Changkun</author>
  <license>BSD</license>
  <url>http://http://www.droid.ac.cn</url>

  <buildtool_depend>catkin</buildtool_depend>

  <run_depend>navigation_sim_demo</run_depend>      <!-- 注意这里的run_depend标签，将其他软件
包都设为依赖项-->
  <run_depend>param_demo</run_depend>
  <run_depend>robot_sim_demo</run_depend>
  <run_depend>service_demo</run_depend>
  <run_depend>slam_sim_demo</run_depend>
  <run_depend>tf_demo</run_depend>
  <run_depend>topic_demo</run_depend>

  <export>      <!-- 这里需要有export和metapackage标签，注意这种固定写法-->
    <metapackage/>
  </export>
</package>

```

metapacakge 中的以上两个文件和普通pacakge不同点是：

- CMakeLists.txt :加入了 catkin\_metapackage() 宏，指定本软件包为一个metapacakge。
- package.xml :标签将所有软件包列为依赖项，标签中添加标签声明。

metapacakge在我们实际开发一个大工程时可能有用

## 2.7 其他常见文件类型

在ROS的pacakge中，还有其他许多常见的文件类型，这里做个总结。

### 2.7.1 launch文件

launch文件一般以.launch或.xml结尾，它对ROS需要运行程序进行了打包，通过一句命令来启动。一般launch文件中会指定要启动哪些package下的哪些可执行程序，指定以什么参数启动，以及一些管理控制的命令。launch文件通常放在软件包的 launch/ 路径中中。launch文件的具体写法见3.2节。

### 2.7.2 msg/srv/action文件

ROS程序中有可能有一些自定义的消息/服务/动作文件，为程序的发者所设计的数据结构，这类的文件以 .msg , .srv , .action 结尾，通常放在package的 msg/ , srv/ , action/ 路径下。

msg文件写法见3.4节，srv文件写法见3.6节。

### 2.7.3 urdf/xacro文件

urdf/xacro文件是机器人模型的描述文件，以.urdf或.xacro结尾。它定义了机器人的连杆和关节的信息，以及它们之间的位置、角度等信息，通过urdf文件可以将机器人的物理连接信息表示出来。并在可视化调试和仿真中显示。

urdf文件的写法见第七章。

### 2.7.4 yaml文件

yaml文件一般存储了ROS需要加载的参数信息，一些属性的配置。通常在launch文件或程序中读取.yaml文件，把参数加载到参数服务器上。通常我们会把yaml文件存放在 param/ 路径下

### 2.7.5 dae/stl文件

**dae**或**stl**文件是3D模型文件，机器人的urdf或仿真环境通常会引用这类文件，它们描述了机器人的三维模型。相比urdf文件简单定义的性状，dae/stl文件可以定义复杂的模型，可以直接从solidworks或其他建模软件导出机器人装配模型，从而显示出更加精确的外形。

## 2.7.6 rviz文件

**rviz**文件本质上是固定格式的文本文件，其中存储了RViz窗口的配置（显示哪些控件、视角、参数）。通常**rviz**文件不需要我们去手动修改，而是直接在**RViz**工具里保存，下次运行时直接读取。

# 第三章 ROS通信架构（一）

## 本章简介

- ROS的通信架构是ROS的灵魂，也是整个ROS正常运行的关键所在。ROS通信架构包括各种数据的处理，进程的运行，消息的传递等等。本章主要介绍了通信架构的基础通信方式和相关概念。其中首先介绍了最小的进程单元节点Node,和节点管理器Node master。了解了ROS中的进程都是由很多的Node组成，并且由Node master来管理这些节点。
- 第二节我们介绍了ROS的“发动机”——launch文件，学习它的格式和内容，更深入的理解ROS在启动运行时它的工作都是由什么进程支配的，从而理解启动运行的原理。
- 在后面的几节我们介绍了ROS中通信方式。ROS中的通信方式有四种，主题、服务、参数服务器、动作库。每个通信方式都有自己的特点，本章首先介绍话题通信方式--topic。

## 3.1 Node & Master

### 3.1.1 Node

在ROS的世界里，最小的进程单元就是节点（node）。一个软件包里可以有多个可执行文件，可执行文件在运行之后就成了一个进程(process)，这个进程在ROS中就叫做节点。从程序角度来说，node就是一个可执行文件（通常为C++编译生成的可执行文件、Python脚本）被执行，加载到了内存之中；从功能角度来说，通常一个node负责者机器人的某一个单独的功能。由于机器人的功能模块非常复杂，我们往往不会把所有功能都集中到一个node上，而会采用分布式的方式，把鸡蛋放到不同的篮子里。例如有一个node来控制底盘轮子的运动，有一个node驱动摄像头获取图像，有一个node驱动激光雷达，有一个node根据传感器信息进行路径规划……这样做可以降低程序发生崩溃的可能性，试想一下如果把所有功能都写到一个程序中，模块间的通信、异常处理将会很麻烦。

我们在1.4节打开了小海龟的运动程序和键盘控制程序，在1.5节同样启动了键盘运动程序，这每一个程序便是一个node。ROS系统中不同功能模块之间的通信，也就是节点间的通信。我们可以把键盘控制替换为其他控制方式，而小海龟运动程序、机器人仿真程序则不用变化。这样就是一种模块化分工的思想。

### 3.1.2 Master

由于机器人的元器件很多，功能庞大，因此实际运行时往往回运行众多的node，负责感知世界、控制运动、决策和计算等功能。那么如何合理的进行调配、管理这些node？这就要利用ROS提供给我们的节点管理器master，master在整个网络通信架构里相当于管理中心，管理着各个node。node首先在master处进行注册，之后master会将该node纳入整个ROS程序中。node之间的通信也是先由master进行“牵线”，才能两两的进行点对点通信。当ROS程序启动时，第一步首先启动master，由节点管理器处理依次启动node。

### 3.1.3 启动master和node

当我们启动ROS时，首先输入命令：

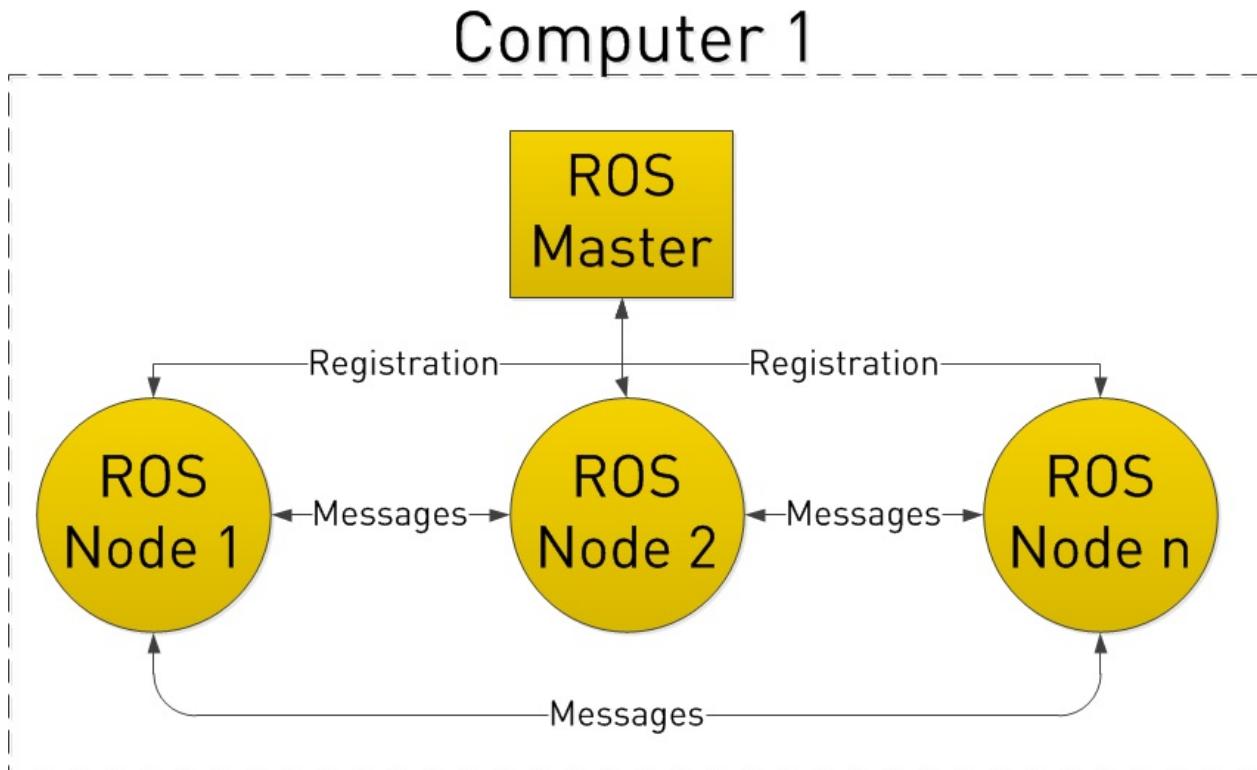
```
$ roscore
```

此时ROS master启动，同时启动的还有 rosout 和 parameter server，其中 rosout 是负责日志输出的一个节点，其作用是告知用户当前系统的状态，包括输出系统的error、warning等等，并且将log记录于日志文件中，parameter server 即是参数服务器，它并不是一个node，而是存储参数配置的一个服务器，后文我们会单独介绍。每一次我们运行ROS的节点前，都需要把master启动起来，这样才能够让节点启动和注册。

master之后，节点管理器就开始按照系统的安排协调进行启动具体的节点。节点就是一个进程，只不过在ROS中它被赋予了专用的名字——node。在第二章我们介绍了ROS的文件系统，我们知道一个package中存放着可执行文件，可执行文件是静态的，当系统执行这些可执行文件，将这些文件加载到内存中，它就成为了动态的node。具体启动node的语句是：

```
$ rosrun pkg_name node_name
```

通常我们运行ROS，就是按照这样的顺序启动，有时候节点太多，我们会选择用launch文件来启动，下一小节会有介绍。Master、Node之间以及Node之间的关系如下图所示：



### 3.1.3 rosrun和rosnode命令

**rosrun**命令的详细用法如下：

```
$ rosrun [--prefix cmd] [--debug] pkg_name node_name [ARGS]
```

rosrun将会寻找PACKAGE下的名为EXECUTABLE的可执行程序，将可选参数ARGS传入。例如在GDB下运行ros程序：

```
$ rosrun --prefix 'gdb -ex run --args' pkg_name node_name
```

**rosnode**命令的详细作用列表如下：

rosnode命令	作用
rosnode list	列出当前运行的node信息
rosnode info node_name	显示出node的详细信息
rosnode kill node_name	结束某个node
rosnode ping	测试连接节点
rosnode machine	列出在特定机器或列表机器上运行的节点
rosnode cleanup	清除不可到达节点的注册信息

以上命令中常用的为前三个，在开发调试时经常会需要查看当前node以及node信息，所以请记住这些常用命令。如果你想不起来，也可以通过 `rosnode help` 来查看 `rosnode` 命令的用法。

## 3.2 launch文件

### 3.2.1 简介

机器人是一个系统工程，通常一个机器人运行操作时要开启很多个node，对于一个复杂的机器人的启动操作应该怎么做呢？当然，我们并不需要每个节点依次进行rosrun，ROS为我们提供了一个命令能一次性启动master和多个node。该命令是：

```
$ rosrun pkg_name file_name.launch
```

rosrun命令首先会自动进行检测系统的roscore有没有运行，也即是确认节点管理器是否在运行状态中，如果master没有启动，那么rosrun就会首先启动master，然后再按照launch的规则执行。launch文件里已经配置好了启动的规则。所以 rosrun 就像是一个启动工具，能够一次性把多个节点按照我们预先的配置启动起来，减少我们在终端中一条条输入指令的麻烦。

### 3.2.2 写法与格式

launch文件同样也遵循着xml格式规范，是一种标签文本，它的格式包括以下标签：

```
<launch>      <!-- 根标签 -->
<node>       <!-- 需要启动的node及其参数 -->
<include>     <!-- 包含其他launch -->
<machine>    <!-- 指定运行的机器 -->
<env-loader>  <!-- 设置环境变量 -->
<param>       <!-- 定义参数到参数服务器 -->
<rosparam>   <!-- 启动yaml文件参数到参数服务器 -->
<arg>        <!-- 定义变量 -->
<remap>      <!-- 设定参数映射 -->
<group>      <!-- 设定命名空间 -->
</launch>     <!-- 根标签 -->
```

参考链接:<http://wiki.ros.org/roslaunch/XML>

### 3.2.3 示例

launch文件的写法和格式看起来内容比较复杂，我们先来介绍一个最简单的例子如下：

```
<launch>
<node name="talker" pkg="rospy_tutorials" type="talker" />
</launch>
```

这是官网给出的一个最小的例子，文本中的信息是，它启动了一个单独的节点 `talker`，该节点是包 `rospy_tutorials` 软件包中的节点。

然而实际中的launch文件要复杂很多，我们以 `Ros-Academy-for-Beginners` 中的 `robot_sim_demo` 为例：

```
<launch>

<!--arg是launch标签中的变量声明，arg的name为变量名，default或者value为值--&gt;
&lt;arg name="robot" default="xbot2"/&gt;
&lt;arg name="debug" default="false"/&gt;
&lt;arg name="gui" default="true"/&gt;
&lt;arg name="headless" default="false"/&gt;

<!-- Start Gazebo with a blank world --&gt;
&lt;include file="$(find gazebo_ros)/launch/empty_world.launch"&gt; &lt!--include用来嵌套仿真场景的launch文件--&gt;
&lt;arg name="world_name" value="$(find robot_sim_demo)/worlds/ROS-Academy.world"/&gt;
&lt;arg name="debug" value="$(arg debug)" /&gt;
&lt;arg name="gui" value="$(arg gui)" /&gt;
&lt;arg name="paused" value="false"/&gt;
&lt;arg name="use_sim_time" value="true"/&gt;
&lt;arg name="headless" value="$(arg headless)"/&gt;
&lt;/include&gt;

<!-- Oh, you wanted a robot? --&gt; &lt!--嵌套了机器人的launch文件--&gt;
&lt;include file="$(find robot_sim_demo)/launch/include/$(arg robot).launch.xml" /&gt;

<!--如果你想连同RViz一起启动，可以按照以下方式加入RViz这个node--&gt;
&lt;!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find robot_sim_demo)/urdf_gazebo.rviz" /--&gt;
&lt;/launch&gt;</pre>

```

这个launch文件相比上一个简单的例子来说，内容稍微有些复杂，它的作用是：启动gazebo 模拟器，导入参数内容，加入机器人模型。

## 小结

对于初学者，我们不要求掌握每一个标签是什么作用，但至少应该有一个印象。如果我们要进行自己写launch文件，可以先从改launch文件的模板入手，基本可以满足普通项目的要求。



## 3.3 Topic

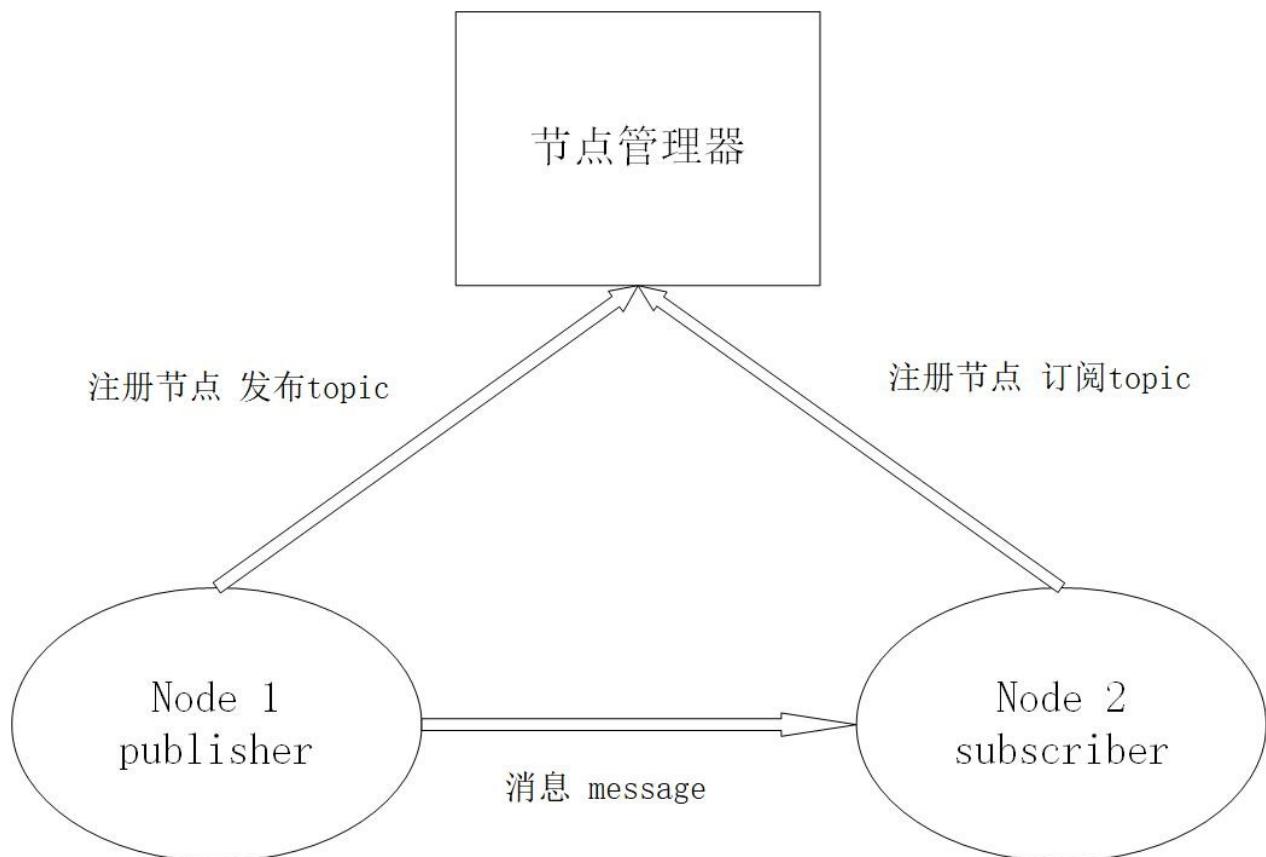
### 3.3.1 简介

ROS的通信方式是ROS最为核心的概念，ROS系统的精髓就在于它提供的通信架构。ROS的通信方式有以下四种：

- Topic 主题
- Service 服务
- Parameter Service 参数服务器
- Actionlib 动作库

### 3.3.2 Topic

ROS中的通信方式中，topic是常用的一种。对于实时性、周期性的消息，使用topic来传输是最佳的选择。topic是一种点对点的单向通信方式，这里的“点”指的是node，也就是说node之间可以通过topic方式来传递信息。topic要经历下面几步的初始化过程：首先，publisher节点和subscriber节点都要到节点管理器进行注册，然后publisher会发布topic，subscriber在master的指挥下会订阅该topic，从而建立起sub-pub之间的通信。注意整个过程是单向的。其结构示意图如下：



Subscriber接收消息会进行处理，一般这个过程叫做回调(**Callback**)。所谓回调就是提前定义好了一个处理函数（写在代码中），当有消息来就会触发这个处理函数，函数会对消息进行处理。

上图就是ROS的topic通信方式的流程示意图。topic通信属于一种异步的通信方式。下面我们通过一个示例来了解下如何使用topic通信。

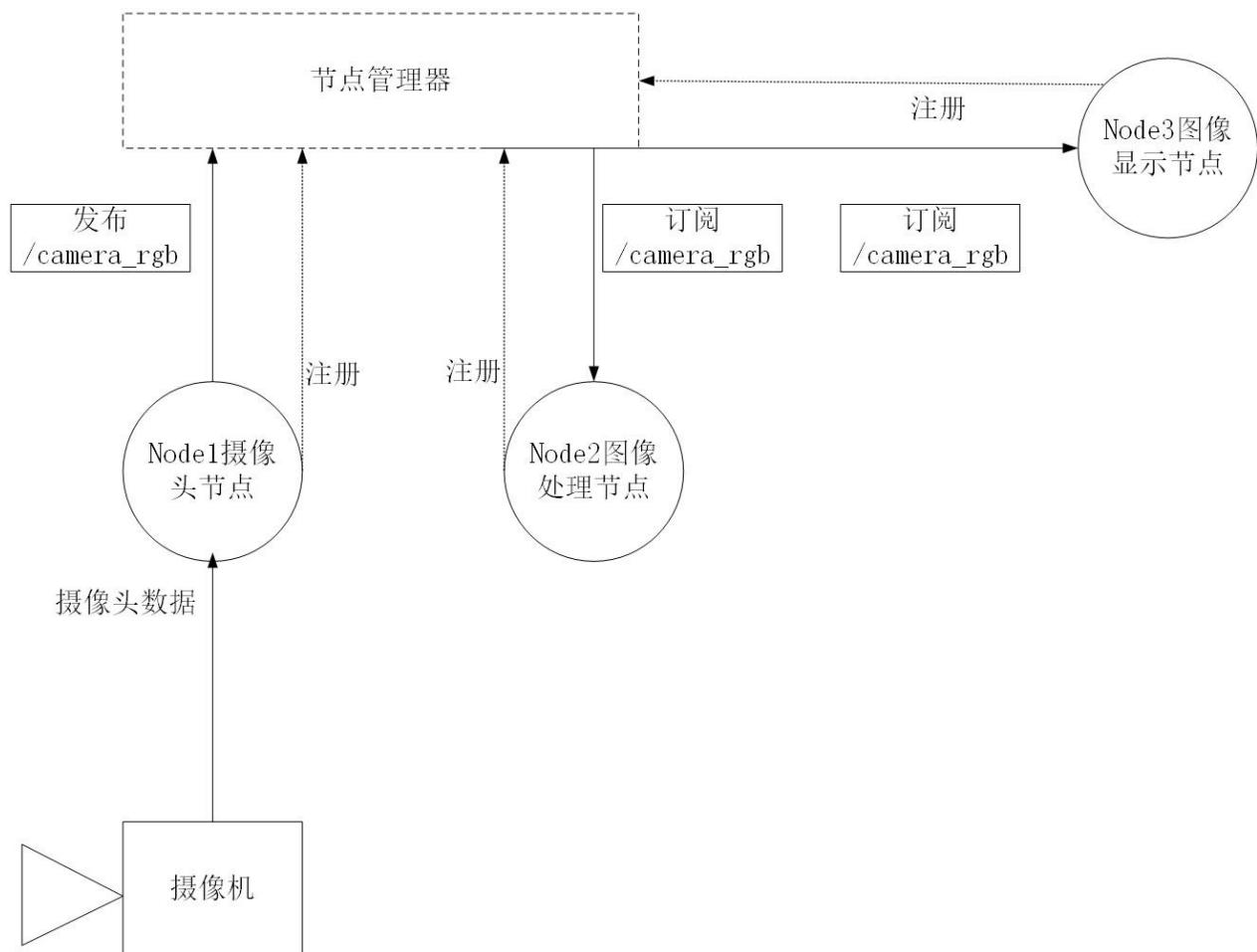
### 3.3.3 通信示例

参考下图，我们以摄像头画面的发布、处理、显示为例讲讲topic通信的流程。在机器人上的摄像头拍摄程序是一个node（圆圈表示，我们记作node1），当node1运行启动之后，它作为一个Publisher就开始发布topic。比如它发布了一个topic（方框表示），叫做`/camera_rgb`，是rgb颜色信息，即采集到的彩色图像。同时，node2假如是图像处理程序，它订阅了`/camera_rgb`这个topic，经过节点管理器的介绍，它就能建立和摄像头节点（node1）的连接。

那么怎么样来理解“异步”这个概念呢？在node1每发布一次消息之后，就会继续执行下一个动作，至于消息是什么状态、被怎样处理，它不需要了解；而对于node2图像处理程序，它只管接收和处理`/camera_rgb`上的消息，至于是谁发来的，它不会关心。所以node1、node2两者都是各司其责，不存在协同工作，我们称这样的通信方式是异步的。

机器人的计算机

笔记本



ROS是一种分布式的架构，一个topic可以被多个节点同时发布，也可以同时被多个节点接收。比如在这个场景中用户可以再加入一个图像显示的节点，我们在想看看摄像头节点的画面，则可以用自己的笔记本连接到机器人的节点管理器，然后在自己的电脑上启动图像显示节点。

这就体现了分布式系统通信的好处：扩展性好、软件复用率高。

总结三点：

1. topic通信方式是异步的，发送时调用publish()方法，发送完完成立即返回，不用等待反馈。
2. subscriber通过回调函数的方式来处理消息。
3. topic可以同时有多个subscribers，也可以同时有多个publishers。ROS中这样的例子有：/rosout、/tf等等。

### 3.3.4 操作命令

在实际应用中，我们应该熟悉topic的几种使用命令，下表详细的列出了各自的命令及其作用。

命令	作用
<code>rostopic list</code>	列出当前所有的topic
<code>rostopic info topic_name</code>	显示某个topic的属性信息
<code>rostopic echo topic_name</code>	显示某个topic的内容
<code>rostopic pub topic_name ...</code>	向某个topic发布内容
<code>rostopic bw topic_name</code>	查看某个topic的带宽
<code>rostopic hz topic_name</code>	查看某个topic的频率
<code>rostopic find topic_type</code>	查找某个类型的topic
<code>rostopic type topic_name</code>	查看某个topic的类型(msg)

如果你一时忘记了命令的写法，可以通过 `rostopic help` 或 `rostopic command -h` 查看具体用法。

### 3.3.5 测试实例

- 首先打开 ROS-Academy-for-Beginners 的模拟场景，输入 `rosrun robot_sim_demo robot_spawn.launch`，看到我们仿真的模拟环境。该 launch 文件启动了模拟场景、机器人。
- 查看当前模拟器中存在的topic，输入命令 `rostopic list`。可以看到许多topic，它们可以视为模拟器与外界交互的接口。
- 查询topic `/camera/rgb/image_raw` 的相关信息：`rostopic info /camera/rgb/image_raw`。则会显示类型信息type，发布者和订阅者的信息。
- 上步我们在演示中可以得知，并没有订阅者订阅该主题，我们指定 `image_view` 来接收这个消息，运行命令 `rosrun image_view image_view image:=<image topic> [transport]`。我们可以看到message，即是上一步中的type。
- 同理我们可以查询摄像头的深度信息depth图像。
- 在用键盘控制仿真机器人运动的时候，我们可以查看速度指令topic的内容 `rostopic echo /cmd_vel`，可以看到窗口显示的各种坐标参数在不断的变化。

通过这些实例的测试，帮助我们更快的掌握topic各种操作命令的使用，以及对topic通信的理解。

## 小结

topic的通信方式是ROS中比较常见的单向异步通信方式，它在很多时候的通信是比较易用且高效的。但是有些需要交互的通信时该方式就显露出自己的不足之处了，后续我们会介绍双向同步的通信方式**service**。

## 3.4 Message

### 3.4.1 简介

topic有很严格的格式要求，比如上节的摄像头进程中的rgb图像topic，它就必然要遵循ROS中定义好的rgb图像格式。这种数据格式就是Message。Message按照定义解释就是topic内容的数据类型，也称之为topic的格式标准。这里和我们平常用到的Massage直观概念有所不同，这里的Message不单单指一条发布或者订阅的消息，也指定为topic的格式标准。

### 3.4.2 结构与类型

基本的msg包括bool、int8、int16、int32、int64(以及uint)、float、float64、string、time、duration、header、可变长数组array[]、固定长度数组array[C]。那么具体的一个msg是怎么组成的呢？我们用一个具体的msg来了解，例如上例中的msg `sensor_msg/image`，位置存放 在 `sensor_msgs/msg/image.msg` 里，它的结构如下：

```
std_msgs/Header header
  uint32    seq
  time     stamp
  string   frame_id
  uint32    height
  uint32    width
  string   encoding
  uint8    is_bigendian
  uint32    step
  uint8[]   data
```

观察上面msg的定义，是不是很类似C语言中的结构体呢？通过具体的定义图像的宽度，高度等等来规范图像的格式。所以这就解释了Message不仅仅是我们在平时理解的一条一条的消息，而且更是ROS中topic的格式规范。或者可以理解msg是一个“类”，那么我们每次发布的内容可以理解为“对象”，这么对比来理解可能更加容易。我们实际通常不会把Message概念分的那么清，通常说Message既指的是类，也是指它的对象。而msg文件则相当于类的定义了。

### 3.4.2 操作命令

rosmsg的命令相比topic就比较少了，只有两个如下：

rosmg命令	作用
rosmg list	列出系统上所有的msg
rosmg show msg_name	显示某个msg的内容

## 3.5 常见message

本小节主要介绍常见的message类型，包括std\_msgs, sensor\_msgs, nav\_msgs, geometry\_msgs等

### Vector3.msg

```
#文件位置:geometry_msgs/Vector3.msg

float64 x
float64 y
float64 z
```

### Accel.msg

```
#定义加速度项，包括线性加速度和角加速度
#文件位置:geometry_msgs/Accel.msg
Vector3 linear
Vector3 angular
```

### Header.msg

```
#定义数据的参考时间和参考坐标
#文件位置:std_msgs/Header.msg
uint32 seq          #数据ID
time stamp         #数据时间戳
string frame_id   #数据的参考坐标系
```

### Echos.msg

```
#定义超声传感器
#文件位置:自定义msg文件
Header header
uint16 front_left
uint16 front_center
uint16 front_right
uint16 rear_left
uint16 rear_center
uint16 rear_right
```

### Quaternion.msg

```
#消息代表空间中旋转的四元数  
#文件位置:geometry_msgs/Quaternion.msg  
  
float64 x  
float64 y  
float64 z  
float64 w
```

### Imu.msg

```
#消息包含了从惯性原件中得到的数据，加速度为m/^2，角速度为rad/s  
#如果所有的测量协方差已知，则需要全部填充进来如果只知道方差，则  
#只填充协方差矩阵的对角数据即可  
#位置：sensor_msgs/Imu.msg  
  
Header header  
Quaternion orientation  
float64[9] orientation_covariance  
Vector3 angular_velocity  
float64[9] angular_velocity_covariance  
Vector3 linear_acceleration  
float64[] linear_acceleration_covariance
```

### LaserScan.msg

```
#平面内的激光测距扫描数据，注意此消息类型仅仅适配激光测距设备  
#如果有其他类型的测距设备(如声呐)，需要另外创建不同类型的消息  
#位置：sensor_msgs/LaserScan.msg  
  
Header header          #时间戳为接收到第一束激光的时间  
float32 angle_min      #扫描开始时的角度(单位为rad)  
float32 angle_max      #扫描结束时的角度(单位为rad)  
float32 angle_increment #两次测量之间的角度增量(单位为rad)  
float32 time_increment  #两次测量之间的时间增量(单位为s)  
float32 scan_time       #两次扫描之间的时间间隔(单位为s)  
float32 range_min       #距离最小值(m)  
float32 range_max       #距离最大值(m)  
float32[] ranges        #测距数据(m, 如果数据不在最小数据和最大数据之间，则抛弃)  
float32[] intensities   #强度，具体单位由测量设备确定，如果仪器没有强度测量，则数组为空即可
```

### Point.msg

```
#空间中的点的位置  
#文件位置:geometry_msgs/Point.msg  
  
float64 x  
float64 y  
float64 z
```

## Pose.msg

```
#消息定义自由空间中的位姿信息，包括位置和指向信息  
#文件位置:geometry_msgs/Pose.msg  
  
Point position  
Quaternion orientation
```

## PoseStamped.msg

```
#定义有时空基准的位姿  
#文件位置：geometry_msgs/PoseStamped.msg  
  
Header header  
Pose pose
```

## PoseWithCovariance.msg

```
#表示空间中含有不确定性的位姿信息  
#文件位置：geometry_msgs/PoseWithCovariance.msg  
  
Pose pose  
float64[36] covariance
```

## Power.msg

```
#表示电源状态，是否开启  
#文件位置：自定义msg文件  
Header header  
bool power  
#####  
bool ON = 1  
bool OFF = 0
```

## Twist.msg

```
#定义空间中物体运动的线速度和角速度  
#文件位置：geometry_msgs/Twist.msg  
  
Vector3 linear  
Vector3 angular
```

### TwistWithCovariance.msg

```
#消息定义了包含不确定性的速度量，协方差矩阵按行分别表示：  
#沿x方向速度的不确定性，沿y方向速度的不确定性，沿z方向速度的不确定性  
#绕x转动角速度的不确定性，绕y轴转动的角速度的不确定性，绕z轴转动的  
#角速度的不确定性  
#文件位置：geometry_msgs/TwistWithCovariance.msg  
  
Twist twist  
float64[36] covariance #分别表示[x; y; z; Rx; Ry; Rz]
```

### Odometry.msg

```
#消息描述了自由空间中位置和速度的估计值  
#文件位置：nav_msgs/Odometry.msg  
  
Header header  
string child_frame_id  
PoseWithCovariance pose  
TwistWithCovariance twist
```

# 第四章 **ROS**通信架构（二）

## 本章简介

继上一章节介绍了ROS通信架构中最常见的话题通信方式，本章节将继续介绍ROS通信方式中的**service**、**parameter server**、**actionlib**。通过学习这四种通信方式，了解他们的通信原理和参数命令。当然还有各自的优缺点和不同的适用方面。通过这两章的学习，大致上会对于ROS的通信架构有一个宏观的理解，为后面的学习和实际应用提供了理论基础。

## 4.1 Service

### 4.1.1 Service

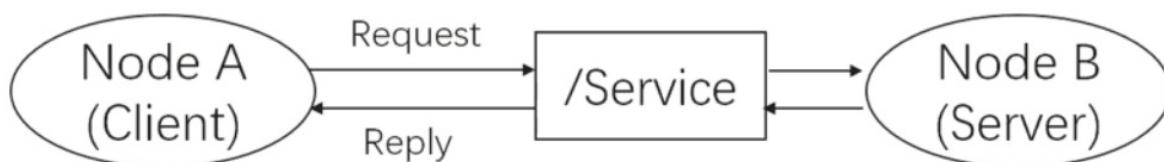
上一章我们介绍了ROS的通信方式中的topic(主题)通信，我们知道topic是ROS中的一种单向的异步通信方式。然而有些时候单向的通信满足不了通信要求，比如当一些节点只是临时而非周期性的需要某些数据，如果用topic通信方式时就会消耗大量不必要的系统资源，造成系统的低效率高功耗。这种情况下，就需要有另外一种请求-查询式的通信模型。这节我们来介绍ROS通信中的另一种通信方式——service(服务)。

### 4.1.2 工作原理

#### 简介

为了解决以上问题，service方式在通信模型上与topic做了区别。Service通信是双向的，它不仅可以发送消息，同时还会有关反馈。所以service包括两部分，一部分是请求方（Client），另一部分是应答方/服务提供方（Server）。这时请求方（Client）就会发送一个request，要等待server处理，反馈回一个reply，这样通过类似“请求-应答”的机制完成整个服务通信。

这种通信方式的示意图如下：Node B是server（应答方），提供了一个服务的接口，叫做 /Service，我们一般都会用string类型来指定service的名称，类似于topic。Node A向Node B发起了请求，经过处理后得到了反馈。



#### 过程

Service是同步通信方式，所谓同步就是说，此时Node A发布请求后会在原地等待reply，直到Node B处理完了请求并且完成了reply，Node A才会继续执行。Node A等待过程中，是处于阻塞状态的成通信。这样的通信模型没有频繁的消息传递，没有冲突与高系统资源的占用，只有接受请求才执行服务，简单而且高效。

### 4.1.3 topic VS service

我们对比一下这两种最常用的通信方式，加深我们对两者的理解和认识，具体见下表：

名称	Topic	Service
通信方式	异步通信	同步通信
实现原理	TCP/IP	TCP/IP
通信模型	Publish-Subscribe	Request-Reply
映射关系	Publish-Subscribe(多对多)	Request-Reply (多对一)
特点	接受者收到数据会回调 (Callback)	远程过程调用 (RPC) 服务器端的服务
应用场景	连续、高频的数据发布	偶尔使用的功能/具体的任务
举例	激光雷达、里程计发布数据	开关传感器、拍照、逆解计算

注意：远程过程调用(Remote Procedure Call，RPC)，可以简单通俗的理解为在一个进程里调用另一个进程的函数。

## 4.1.4 操作命令

在实际应用中，service通信方式的命令时 `rosservice`，具体的命令参数如下表：

<code>rosservice</code> 命令	作用
<code>rosservice list</code>	显示服务列表
<code>rosservice info</code>	打印服务信息
<code>rosservice type</code>	打印服务类型
<code>rosservice uri</code>	打印服务ROSRPC uri
<code>rosservice find</code>	按服务类型查找服务
<code>rosservice call</code>	使用所提供的args调用服务
<code>rosservice args</code>	打印服务参数

## 4.1.5 测试实例

- 首先依然是打开我们教材的模拟场景 `roslaunch robot_sim_demo robot_spawn.launch`。
- 输入 `rosservice list`，查看当前运行的服务。

3. 随机选择 `/gazebo/delete_light` 服务，观察名称，是删除光源的操作。
4. 输入 `rosservice info /gazebo/delete_light` 查看属性信息。可以看到信息，  
Node : `/gazebo`, Type : `gazebo_msgs/DeleteLight`, Args : `Light_name`。这里的类型  
type也就是下文介绍的srv,传递参数Light\_name
5. 输入 `rosservice call /gazebo/delete_light sun`,这里的sun 是参数名，使我们模拟场景  
中的唯一光源太阳。操作完成后可以看到场景中的光线消失。
6. 可以看到终端的回传信息：`success: True`和`sun successfully deleted`。这就是双向通信  
的信息反馈，通知操作已经成功完成。

## 小结

本节我们详细介绍了service通信方式，建议与topic通信方式进行对比记忆，这样我们能更深的理解这两种通信方式，也能在以后的学习工作中更加合理使用每个通信方式，获得更高的效率。

## 4.2 Srv

### 4.2.1 简介

类似msg文件，srv文件是用来描述服务（service数据类型的，service通信的数据格式定义在\*.srv中。它声明了一个服务，包括请求(request)和响应（reply）两部分。其格式声明如下：

举例：

msgs\_demo/srv/DetectHuman.srv

```
bool start_detect
---
my_pkg/HumanPose[] pose_data
```

msgs\_demo/msg/HumanPose.msg

```
std_msgs/Header header
string uuid
int32 number_of_joints
my_pkg/JointPose[] joint_data
```

msgs\_demo/msg/JointPose.msg

```
string joint_name
geometry_msgs/Pose pose
float32 confidence
```

以 DetectHuman.srv 文件为例，该服务例子取自OpenNI的人体检测ROS软件包。它是用来查询当前深度摄像头中的人体姿态和关节数的。srv文件格式很固定，第一行是请求的格式，中间用---隔开，第三行是应答的格式。在本例中，请求为是否开始检测，应答为一个数组，数组的每个元素为某个人的姿态（HumanPose）。而对于人的姿态，其实是一个msg，所以srv可以嵌套msg在其中，但它不能嵌套srv。

### 4.2.2 操作命令

具体的操作指令如下表：

rossrv 命令	作用
rossrv show	显示服务描述
rossrv list	列出所有服务
rossrv md5	显示服务md5sum
rossrv package	列出包中的服务
rossrv packages	列出包含服务的包

### 4.2.3 修改部分文件

定义完了msg、srv文件，还有重要的一个步骤就是修改package.xml和修改CMakeList.txt。这些文件需要添加一些必要的依赖等，例如：

```
<build_depend>** message_generation **</build_depend>
<run_depend>** message_runtime **</run_depend>
```

上述文本中“\*\*”所引就是新添加的依赖。又例如：

```
find_package(...roscpp rospy std_msgs ** message_generation **)
catkin_package(
...
CATKIN_DEPENDS ** message_runtime ** ...
...)

add_message_file(
FILES
** DetectHuman.srv **
** HumanPose.msg **
** JointPos.msg **)

** generate_messages(DEPENDENCIES std_msgs) **
```

添加的这些内容指定了srv或者msg在编译或者运行中需要的依赖。具体的作用我们初学者可不深究，我们需要了解的是，无论我们自定义了srv,还是msg，修改上述部分添加依赖都是必不可少的一步。

## 4.3 Parameter server

### 4.3.1 简介

前文介绍了ROS中常见的两种通信方式——主题和服务，这节介绍另外一种通信方式——参数服务器（parameter server）。与前两种通信方式不同，参数服务器也可以说是特殊的“通信方式”。特殊点在于参数服务器是节点存储参数的地方、用于配置参数，全局共享参数。参数服务器使用互联网传输，在节点管理器中运行，实现整个通信过程。

参数服务器，作为ROS中另外一种数据传输方式，有别于topic和service，它更加的静态。参数服务器维护着一个数据字典，字典里存储着各种参数和配置。

#### 字典简介

何为字典，其实就是一个个的键值对，我们小时候学习语文的时候，常常都会有一本字典，当遇到不认识的字了我们可以查部首查到这个字，获取这个字的读音、意义等等，而这里的字典可以对比理解记忆。键值key可以理解为语文里的“部首”这个概念，每一个key都是唯一的，参照下图：

Key	/rosdistro	/rosversion	/use_sim_time	...
Value	'kinetic'	'1.12.7'	true	...

每一个key不重复，且每一个key对应着一个value。也可以说字典就是一种映射关系，在实际的项目应用中，因为字典的这种静态的映射特点，我们往往将一些不常用到的参数和配置放入参数服务器里的字典里，这样对这些数据进行读写都将方便高效。

#### 维护方式

参数服务器的维护方式非常的简单灵活，总的来讲有三种方式：

- 命令行维护
- launch文件内读写
- node源码

下面我们来一一介绍这三种维护方式。

### 4.3.2 命令行维护

使用命令行来维护参数服务器，主要使用 `rosparam` 语句来进行操作的各种命令，如下表：

<b>rosparam</b> 命令	作用
<code>rosparam set param_key param_value</code>	设置参数
<code>rosparam get param_key</code>	显示参数
<code>rosparam load file_name</code>	从文件加载参数
<code>rosparam dump file_name</code>	保存参数到文件
<code>rosparam delete</code>	删除参数
<code>rosparam list</code>	列出参数名称

## load&&dump 文件

load和dump文件需要遵守YAML格式，YAML格式具体示例如下：

```
name: 'Zhangsan'
age: 20
gender: 'M'
score{Chinese:80,Math:90}
score_history:[85,82,88,90]
```

简明解释。就是“名称+：+值”这样一种常用的解释方式。一般格式如下：

```
key : value
```

遵循格式进行定义参数。其实就可以把YAML文件的内容理解为字典，因为它也是键值对的形式。

### 4.3.3 launch文件内读写

launch文件中有很多标签，而与参数服务器相关的标签只有两个，一个是 `<param>`，另一个是 `<rosparam>`。这两个标签功能比较相近，但 `<param>` 一般只设置一个参数，请看下例：

(1) (2) (3)

观察上例比如序号3的param就定义了一个key和一个value，交给了参数服务器维护。而序号1的param只给出了key，没有直接给出value，这里的value是由后没的脚本运行结果作为value进行定义的。序号(2)就是rosparam的典型用法，先指定一个YAML文件，然后施加command，其效果等于 `rosparam load file_name`。

## 4.3.4 node源码

除了上述最常用的两种读写参数服务器的方法，还有一种就是修改ROS的源码，也就是利用API来对参数服务器进行操作。具体内容我们学习完后面章节再进行介绍。

## 4.3.5 操作实例

1. 首先依然是打开我们教材的模拟场景 `roslaunch robot_sim_demo robot_spawn.launch`。
2. 输入 `rosparam list` 查看参数服务器上的param。
3. 查询参数信息，例如查询竖直方向重力参数。输入 `rosparam get /gazebo/gravity_z` 回车得到参数值`value=-9.8`。
4. 尝试保存一个参数到文件输入 `rosparam dump param.yaml` 之后就可以在当前路径看到该文件，也就能打开去查看到相关的参数信息。
5. 参数服务器的其他命令操作方式大致相同，我们可以多多练习，巩固对参数服务器的理解和应用。

## 参数类型

ROS参数服务器为参数值使用XMLRPC数据类型，其中包括:`strings, integers, floats, booleans, lists, dictionaries, iso8601 dates, and base64-encoded data`。

## 4.4 Action

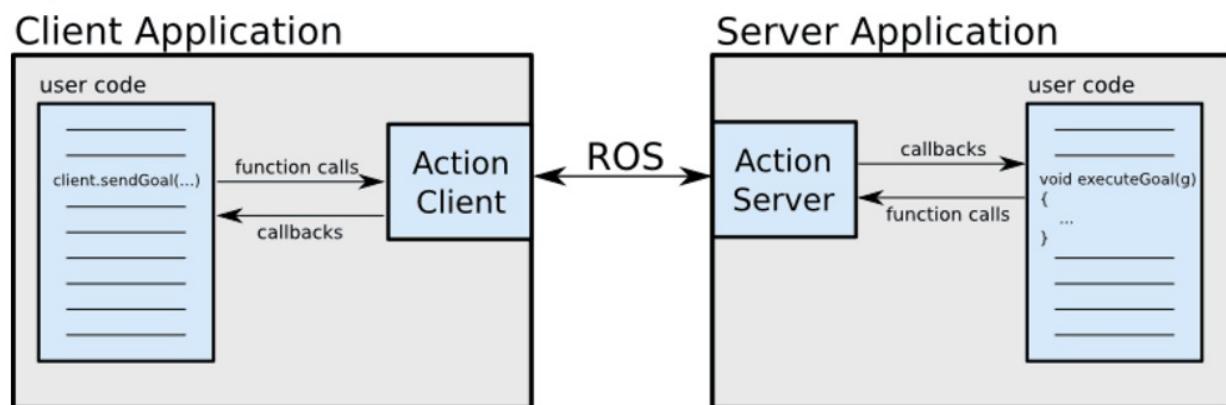
### 4.4.1 简介

Actionlib是ROS中一个很重要的库，类似service通信机制，actionlib也是一种请求响应机制的通信方式，actionlib主要弥补了service通信的一个不足，就是当机器人执行一个长时间的任务时，假如利用service通信方式，那么publisher会很长时间接受不到反馈的reply，致使通信受阻。当service通信不能很好的完成任务时候，actionlib则可以比较适合实现长时间的通信过程，actionlib通信过程可以随时被查看进度，也可以终止请求，这样的一个特性，使得它在一些特别的机制中拥有很高的效率。

### 4.4.2 通信原理

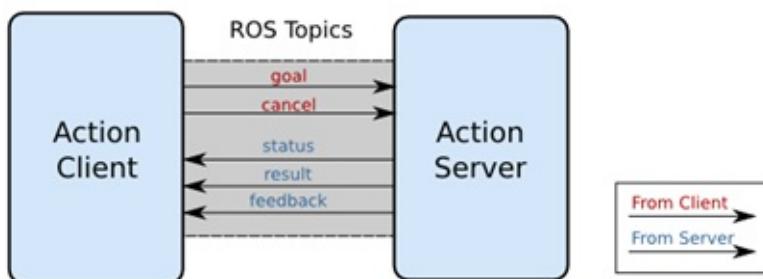
Action的工作原理是client-server模式，也是一个双向的通信模式。通信双方在ROS Action Protocol下通过消息进行数据的交流通信。client和server为用户提供一个简单的API来请求目标（在客户端）或通过函数调用和回调来执行目标（在服务器端）。

工作模式的结构示意图如下：



通信双方在ROS Action Protocol下进行交流通信是通过接口来实现,如下图:

Action Interface



我们可以看到，客户端会向服务器发送目标指令和取消动作指令，而服务器则可以给客户端发送实时的状态信息、结果信息、反馈信息等等，从而完成了service没法做到的部分。

### 4.4.3 Action 规范

利用动作库进行请求响应，动作的内容格式应包含三个部分，目标、反馈、结果。

- 目标

机器人执行一个动作，应该有明确的移动目标信息，包括一些参数的设定，方向、角度、速度等等。从而使机器人完成动作任务。

- 反馈

在动作进行的过程中，应该有实时的状态信息反馈给服务器的实施者，告诉实施者动作完成的状态，可以使实施者作出准确的判断去修正命令。

- 结果

当运动完成时，动作服务器把本次运动的结果数据发送给客户端，使客户端得到本次动作的全部信息，例如可能包含机器人的运动时长，最终姿势等等。

### 4.4.4 Action规范文件格式

Action规范文件的后缀名是.action，它的内容格式如下：

```
# Define the goal
uint32 dishwasher_id # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

### 4.4.5 Action实例详解

Actionlib是一个用来实现action的一个功能包集。我们在demo中设置一个场景，执行一个搬运的action，搬运过程中客户端会不断的发回反馈信息，最终完成整个搬运过程。

本小节的演示源码在课程的演示代码包里，[此处为链接](#)。

首先写handling.action文件，类比如上的格式。包括三个部分，目标，结果，反馈。如下：

```
# Define the goal
uint32 handling_id
---

# Define the result
uint32 Handling_completed
---

# Define a feedback message
float32 percent_complete
```

写完之后修改文件夹里CmakeLists.txt如下内容：

1. find\_package(catkin REQUIRED genmsg actionlib\_msgs actionlib)
2. add\_action\_files(DIRECTORY action FILES DoDishes.action)  
generate\_messages(DEPENDENCIES actionlib\_msgs)
3. add\_action\_files(DIRECTORY action FILES Handling.action)
4. generate\_messages( DEPENDENCIES actionlib\_msgs)

修改package.xml,添加所需要的依赖如下：

1. <build\_depend>actionlib </build\_depend>
2. <build\_depend>actionlib\_msgs</build\_depend>
3. <run\_depend>actionlib</run\_depend>
4. <run\_depend>actionlib\_msgs</run\_depend>

然后回到工作空间 catkin\_ws 进行编译.

本例中设置的的action,定义了一个搬运的例子,首先写客户端,实现功能发送action请求,包括进行目标活动,或者目标活动.之后写服务器,实验返回客户端活动当前状态信息,结果信息,和反馈信息.从而实现action.本例测试结果截图如下:

```
thinkpad-t470p:~$ rosrun action_demo handling_Client  
[1520401570.505396139]: Waiting for action server to start.  
  
thinkpad-t470p:~$ rosrun action_demo handling_Server  
[1520401916.731932031]: Handling 1 is working.  
[1520401916.731161032]: Action server started, sending goal.  
[1520401916.732157514]: Goal just went active  
[1520401916.732477783]: handling percent_complete : 10.000000  
[1520401917.732335113]: handling percent_complete : 20.000000  
[1520401918.732368567]: handling percent_complete : 30.000000  
[1520401919.732350064]: handling percent_complete : 40.000000  
[1520401920.732354637]: handling percent_complete : 50.000000  
[1520401921.732353928]: handling percent_complete : 60.000000  
[1520401922.732349312]: handling percent_complete : 70.000000  
[1520401923.732352113]: handling percent_complete : 80.000000  
[1520401924.732352190]: handling percent_complete : 90.000000  
[1520401925.732346643]: handling percent_complete : 100.000000  
[1520401926.732618550]: handling completed!  
[1520401916.731932031]: Handling 1 is working.  
[1520401926.732003408]: Handling 1 finish working.
```

## 小结

至此，ROS通信架构的四种通信方式就介绍结束，我们可以对比学习这四种通信方式，去思考每一种通信的优缺点和适用条件，在正确的地方用正确的通信方式，这样整个ROS的通信会更加高效，机器人也将更加的灵活和智能。机器人学会了通信，也就相当于有了“灵魂”。

## 4.5 常见srv类型

本小节介绍常见的 srv 类型及其定义。srv 类型相当于两个 message 通道，一个发送，一个接收。

### AddTwoInts.srv

```
#对两个整数求和，虚线前是输入量，后是返回量
#文件位置：自定义srv文件
int32 a
int32 b
---
int32 sum
```

### Empty.srv

```
#文件位置：std_srvs/Empty.srv
#代表一个空的srv类型
---
```

### GetMap.srv

```
#文件位置：nav_msgs/GetMap.srv
#获取地图，注意请求部分为空
---
nav_msgs/OccupancyGrid map
```

### GetPlan.srv

```
#文件位置：nav_msgs/GetPlan.srv
#得到一条从当前位置到目标点的路径
geometry_msgs/PoseStamped start          #起始点
geometry_msgs/PoseStamped goal           #目标点
float32 tolerance      #到达目标点的x,y方向的容错距离
---
nav_msgs/Path plan
```

### SetBool.srv

```
#文件位置：std_srvs/SetBools.srv
bool data # 启动或者关闭硬件
---
bool success # 标示硬件是否成功运行
string message # 运行信息
```

### SetCameraInfo.srv

```
#文件位置：sensor_msgs/SetCameraInfo.srv
#通过给定的CameraInfo相机信息，来对相机进行标定
sensor_msgs/CameraInfo camera_info #相机信息
---
bool success #如果调用成功，则返回true
string status_message #给出调用成功的细节
```

### SetMap.srv

```
#文件位置：nav_msgs/SetMap.srv
#以初始位置为基准，设定新的地图
nav_msgs/OccupancyGrid map
geometry_msgs/PoseWithCovarianceStamped initial_pose
---
bool success
```

### TalkerListener.srv

```
#文件位置：自定义srv文件
---
bool success # 标示srv是否成功运行
string message # 信息，如错误信息等
```

### Trigger.srv

```
#文件位置：std_srvs/Trigger.srv
---
bool success # 标示srv是否成功运行
string message # 信息，如错误信息等
```

## 4.6 常见action类型

本小节介绍常见的action类型以及其定义

### AddTwoInts.action

```
#文件位置:自定义action文件
#表示将两个整数求和
int64 a
int64 b
---
int64 sum
---
```

### AutoDocking.action

```
#文件位置:自定义action文件
#ggoal
---
#result
string text
---
#feedback
string state
string text
```

### GetMap.action

```
#文件位置:nav_msgs/GetMap.action
#获取地图信息，响应部分为空

---
nav_msgs/OccupancyGrid map
---
#无返回部分
```

### MoveBase.action

```
#文件位置:geometry_msgs/MoveBase.action
geometry_msgs/PoseStamped target_pose
---
---
geometry_msgs/PoseStamped base_position
```



# 第五章 常用工具

## 本章简介

本章主要介绍了ROS开发时常常使用的工具，分别是：

- Gazebo
- RViz
- rqt
- rosbag
- rosbridge
- moveit!

这六个工具是我们开发常常用到的工具，gazebo是一种最常用的ROS仿真工具，也是目前仿真ROS效果最好的工具；RViz是可视化工具，是将接收到的信息呈现出来；rqt则非常好用的数据流可视化工具，有了它我们可以直观的看到消息的通信架构和流通路径；rosbag则是对软件包进行操作的一个命令，此外还提供代码API，对包进行操作编写。rosbridge是一个沟通ROS和外界的功能包，moveit!是目前为止应用最广泛的开源操作软件。好好学习本章，熟练使用这几款工具对于我们的ROS学习和开发都有极大的好处，可以事半功倍。

## 5.1 Gazebo

### 5.1.1 简介

ROS中的工具就是帮助我们完成一系列的操作，使得我们的工作更加轻松高效。ROS工具的功能大概有以下几个方向：仿真、调试、可视化。本节课我们要学习的Gazebo就是实现了仿真的功能，而调试与可视化由Rviz、rqt来实现，我们下节再依次介绍。

### 5.1.2 认识 Gazebo

对于Gazebo,大家可能并不陌生，因为我们在前面的学习过程中已经数次用到这个仿真环境，无论是在topic通信还是在service通信中，我们的demo都是在Gazebo中实现。

Gazebo是一个机器人仿真工具，模拟器，也是一个独立的开源机器人仿真平台。当今市面上还有其他的仿真工具例如V—Rep、Webots等等。但是Gazebo不仅开源，也是兼容ROS最好的仿真工具。



Gazebo的功能很强大，最大的优点是对ROS的支持很好，因为Gazebo和ROS都由OSRF（Open Source Robotics Foundation）开源机器人组织来维护，Gazebo支持很多开源的物理引擎比如最典型的ODE。可以进行机器人的运动学、动力学仿真，能够模拟机器人常用的传感器（如激光雷达、摄像头、IMU等），也可以加载自定义的环境和场景。

### 5.1.3 仿真 的意义

仿真不仅仅只是做出一个很酷的3D场景，更重要的是给机器人一个逼近现实的虚拟物理环境，比如光照条件、物理距离等等。设定好具体的参数，让机器人完成我们设定的目标任务。比如一些有危险因素的测试，就可以让机器人在仿真的环境中去完成，例如无人车在交通环境复杂的交通要道的效果，我们就可以在仿真的环境下测试各种情况无人车的反应与效果，如车辆的性能、驾驶的策略、车流人流的行为模式等，又或者各种不可控因素如雨雪天气，突发事故，车辆故障等，从而收集结果参数指标信息等等，只有更大程度的逼近现实，才能得出车辆的真实效果。直到无人车在仿真条件下做到万无一失，才能放心的投放到真实环境中去使用，这即避免了危险因素对实验者的威胁，也节约了时间和资源，这就是仿真的意义。



通常一些不依赖于具体硬件的算法和场景都可以在Gazebo上仿真，例如图像识别、传感器数据融合处理、路径规划、SLAM等任务完全可以在Gazebo上仿真实现，大大减轻了对硬件的依赖。

## 5.1.4 演示

和我们前面的实例测试一样，我们打开教材的模拟场景，输入 `roslaunch robot_sim_demo robot_spawn.launch`

### 操作说明

- 平移：鼠标左键
- 旋转：鼠标滚轮中键
- 放缩：鼠标滚轮
- 界面左侧是控制面板
- 导入模型就在控制面板的insert,可以直接拖入模拟空间，也可以按需自制模型拖入。

## 5.1.5 小结

虽然Gazebo目前的功能还称不上强大，同时还存在着一些BUG，但是对于我们的入门学习也已经是足够了，随着版本的更新，Gazebo也在越来越强大。

## 5.2 RViz

### 5.2.1 简介

本节课介绍的是我们在ROS开发中非常常用的一个工具，基本上的调试和开发都离不开这个工具——RViz(the Robit Visualization tool)机器人可视化工具，可视化的作用是直观的，它极大的方便了监控和调试等操作。



### 5.2.2 演示

依然打开教材的模拟场景，输入 `roslaunch robot_sim_demo robot_spawn_launch`，之后在命令行打开新的终端直接输入 `$ rviz` 打开工具。

和Gazebo一样，也会显示出一个3D环境，不过操作上有所不同，具体操作如下：

- 平移：鼠标滚轮中键
- 旋转：鼠标左键
- 放缩：鼠标滚轮
- 左侧控制面板，可以添加插件

RViz的插件种类繁多功能强大，非常适合我们开发调试ROS程序。

### 5.2.3 差异

虽然从界面上来看，RViz和Gazebo非常相似，但实际上两者有着很大的不同，Gazebo实现的是仿真，提供一个虚拟的世界，RViz实现的是可视化，呈现接收到的信息。左侧的插件相当于是一个个的subscriber, RViz接收信息，并且显示。所以RViz和Gazebo有本质的差异。

### 5.2.4 小结

RViz和Gazebo是我们常用的ROS工具，更好的利用这些工具是我们ROS进阶的基础。具体的操作和使用可以参考我们的官方演示视频，跟着视频去实战演练，熟悉这两个工具。

## 5.3 rqt

### 5.3.1 简介

rqt是一个基于qt开发的可视化工具，拥有扩展性好、灵活易用、跨平台等特点，主要作用和RViz一致都是可视化，但是和RViz相比，rqt要高级一个层次，。

### 5.3.2 命令

- `rqt_graph` :显示通信架构
- `rqt_plot` : 绘制曲线
- `rqt_console` : 查看日志

#### **rqt\_graph**

`rqt_graph`是来显示通信架构，也就是我们上一章所讲的内容节点、主题等等，当前有哪些Node和topic在运行，消息的流向是怎样，都能通过这个语句显示出来。此命令由于能显示系统的全貌，所以非常的常用。

#### **rqt\_plot**

`rqt_plot`将一些参数，尤其是动态参数以曲线的形式绘制出来。当我们在开发时查看机器人的原始数据，我们就能利用`rqt_plot`将这些原始数据用曲线绘制出来，非常的直观，利于我们分析数据。

#### **rqt\_console**

`rqt_console`里存在一些过滤器，我们可以利用它方便的查到我们需要的日志。

### 5.3.3 实例测试

1. 首先打开我们教材的模拟场景，输入 `roslaunch robot_sim_demo robot_spawn.launch`
2. 输入命令语句 `rqt_graph` ,显示出了当前环境下运行的Node和topic，十分直观的看到通信结构以及消息流向。注意在椭圆形的代表节点，矩形代表topic。
3. 输入命令语句 `rqt_plot` ,显示出曲线坐标窗口，在上方输入框里添加或者删除topic，比如我们查看速度，可以在框里设置好topic后，移动机器人，就可以看到自动绘制的线速度

或者角速度曲线。

4. 输入命令语句 `rqt_console`，显示日志的输出，配合 `rqt_logger_level` 查看日志的级别。

### 5.3.4 小结

`rqt_graph`这个功能是强大的，它使得我们初学者可以直观的看到ROS的通信架构和信息流，方便我们理解的同时，也使得我们能够最快的纠错等等。`rqt_plot`绘制数据曲线图，也是极大的帮助我们了解数据的变化态势，理解数据流的作用，用曲线来显示我们的操作，精确直观。`rqt_console`配合`rqt_logger_level`，查看日志，对于查找错误和DeBug都有很大帮助。

## 5.4 Rosbag

### 5.4.1 简介

`rosbag`是一个用于记录和回放ROS主题的工具。它旨在提高性能，并避免消息的反序列化和重新排序。`rosbag package`提供了命令行工具和代码API，可以用C++或者python来编写包。而且`rosbag`命令行工具和代码API是稳定的，始终保持向后的兼容性。

### 5.4.2 命令

`rosbag`对软件包来操作，一个包是ROS用于存储ROS消息数据的文件格式，`rosbag`命令可以记录、回放和操作包。指令列表如下：

命令	作用
<code>check</code>	确定一个包是否可以在当前系统中进行，或者是否可以迁移。
<code>decompress</code>	压缩一个或多个包文件。
<code>filter</code>	解压一个或多个包文件。
<code>fix</code>	在包文件中修复消息，以便在当前系统中播放。
<code>help</code>	获取相关命令指示帮助信息
<code>info</code>	总结一个或多个包文件的内容。
<code>play</code>	以一种时间同步的方式回放一个或多个包文件的内容。
<code>record</code>	用指定主题的内容记录一个包文件。
<code>reindex</code>	重新索引一个或多个包文件。

- [参考链接](#)

### 5.4.3 小结

`rosbag`通过命令行能够对软件包进行很多的操作，更重要的拥有代码API，可以对包进行重新编写。增加一个ROS API，用于通过服务调用与播放和录制节点进行交互。

## 5.5 Rosbridge

### 5.5.1 简介

Rosbridge是一个用在ROS系统和其他系统之间的一个功能包,就像是它的名字一样,起到一个"桥梁"的作用,使得ros系统和其他系统能够进行交互.Rosbridge为非ROS程序提供了一个JSON API,有许多与Rosbridge进行交互的前端,包括一个用于Web浏览器交互的WebSocket服务器。Rosbridge\_suite是一个包含Rosbridge的元程序包,用于Rosbridge的各种前端程序包(如WebSocket程序包)和帮助程序包。

### 5.5.2 协议和实现

Rosbridge主要包含两部分内容:协议(Potocol)和实现(Implementation)

#### 协议

Rosbridge Protocol提供了非ROS程序与ROS通信的具体的格式规范,规范基于JSON格式,包括订阅topic,发布message,调用server,设置参数,压缩消息等等。例如订阅topic的格式规范如下:

```
{ "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

此规范与所用的编程语言和传输方式无关,任何可以发送JSON格式的语音和传输方式都可以Rosbridge protocol进行交流,并且与ROS进行交互。

#### 实现

Rosbridge\_suite元程序包是实现Rosbridge Protocol并提供WebSocket传输层的包的集合。

这些软件包包括:

- Rosbridge\_library:核心rosbridge软件包。Rosbridge\_library负责获取JSON字符串并将命令发送到ROS,反过来接收处理ROS发过来的信息,将之转换为JSON字符串,并将结果转交给非ROS程序。
- rosapi:通过服务调用来访问某些ROS操作,这些服务通常为ROS客户端库保留的服务。这些操作包括获取和设置参数,获取主题列表等等。

- `rosbridge_server`：虽然Rosbridge\_library提供JSON到ROS转换，但它将传输层留给其他人。Rosbridge\_server提供了一个WebSocket连接，所以浏览器可以与ROS“交谈”。Roslibjs是一个浏览器的JavaScript库，可以通过`rosbridge_server`与ROS进行交流。

[源码](#)

### 5.5.3 安装与使用

安装

Rosbridge是基于ROS的，首先要确保自己正确的安装完成了ROS之后可以启动终端执行命令：

```
sudo apt-get install ros- <rosdistro> -rosbridge-server
```

中间的为自己的ROS版本，依照自己的版本进行安装。

使用

关于更深入的使用，可以参考本课程的视频课程，简单的入门使用可以参考链接如下：

[参考链接](#)

## 5.6 moveit!

### 5.6.1 简介

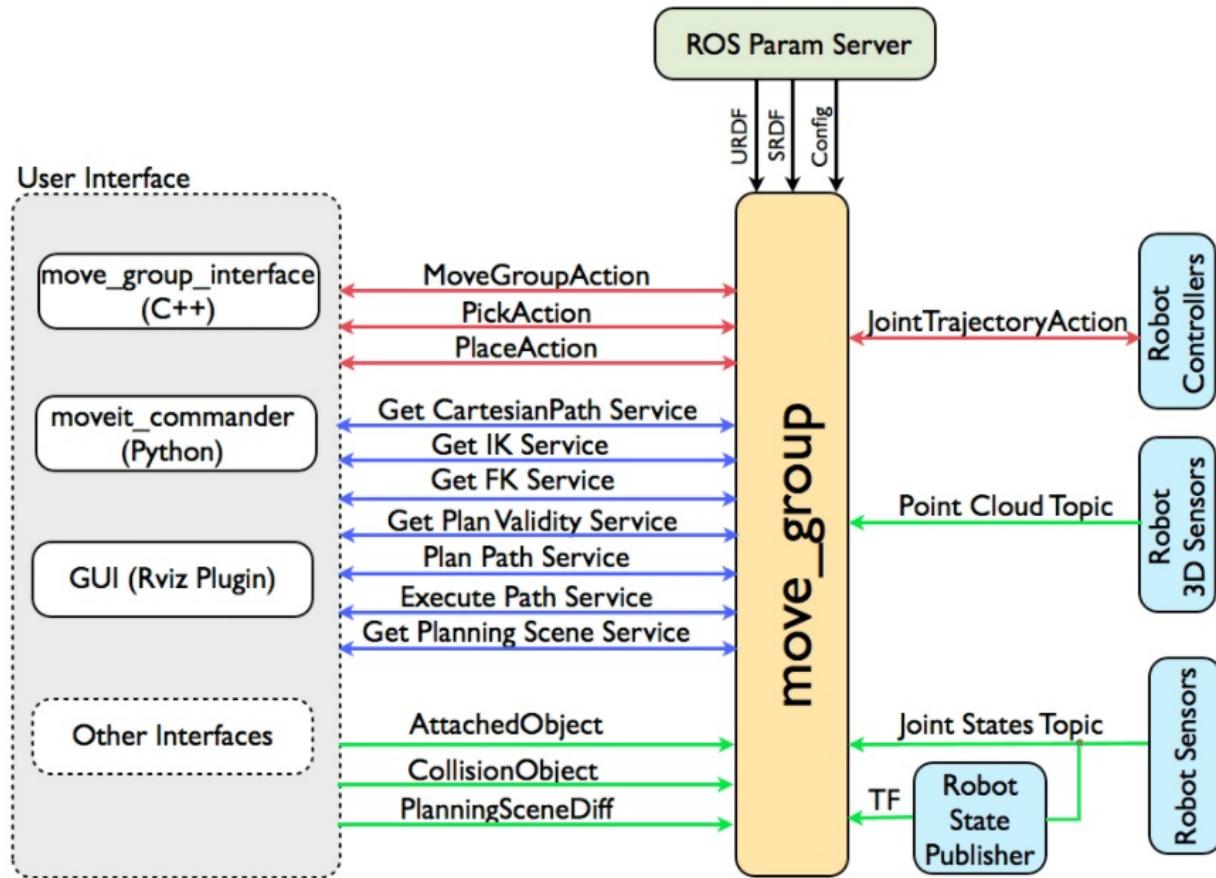
2012年，一款叫做moveit!的移动操作软件诞生了，moveit！最初在Willow Garage由Sachin Chitta，Ioan Sucan，Gil E. Jones，Acorn Pooley，Suat Gedikli，Dave Hershberger开发，它融合了研究者在运动规划、操纵、3D感知、运动学、控制和导航方面的最新进展，为操作者提供了一个易于使用的平台，使用它可以开发先进的机器人应用，也被广泛应用于工业，商业，研发和其他领域。由于以上特性，moveit！一跃成为在机器人上最广泛使用的开源操作软件，截止2017年，已经被用于超过65台机器人。

- [moveit!视频链接](#)

### 5.6.2 使用

moveit!的使用通过为用户提供接口来调用它，包括C++、Python、GUI三种接口。ROS中的`move_group`节点充当整合器，整合多个独立组件，提供ROS风格的Action和service。`move_group`通过ROS topic和action与机器人通讯，获取机器人的位置、节点等状态，获取数据再传递给机器人的控制器。

`move_group`节点获取到节点状态信息或者机器人变换信息时候，会通过控制器的接口去处理这些信息，比如进行坐标转换、规划场景、3D感知。另外，`move_group`的结构比较容易扩展，不仅具有独立的能力如抓放，运动规划，也可扩展自公共类，但实际作为独立的插件运行。moveit!系统结构图如下：



[官网链接](#)

# 第六章 rosccpp

## 本章简介

从本章开始，我们就要正式的接触ROS编程了。在之前的章节，你了解到用命令行启动ROS程序、发送指令消息，或使用可视化界面来调试机器人。你可能很想知道，这些工具到底是如何实现这些功能的。起始这些工具本质上都是基于ROS的客户端库（Client Libarary）实现的，所谓客户端库，简单的理解就是一套接口，ROS为我们机器人开发者提供了不同语言的接口，比如rosccpp是C++语言ROS接口，rospy是python语言的ROS接口，我们直接调用它所提供的函数就可以实现topic、service等通信功能。

本章我们介绍rosccpp，给你介绍rosccpp的基本函数，告诉你用C++开发ROS的基本方法。本章的内容需要有C++的基础，如果你对C++比较陌生，建议先学习C++编程。

# 6.1 Client Library与roscpp

## 6.1.1 Client Library简介

ROS为机器人开发者们提供了不同语言的编程接口，比如C++接口叫做roscpp，Python接口叫做rospy，Java接口叫做rosjava。尽管语言不通，但这些接口都可以用来创建topic、service、param，实现ROS的通信功能。Clinet Library有点类似开发中的Helper Class，把一些常用的基本功能做了封装。

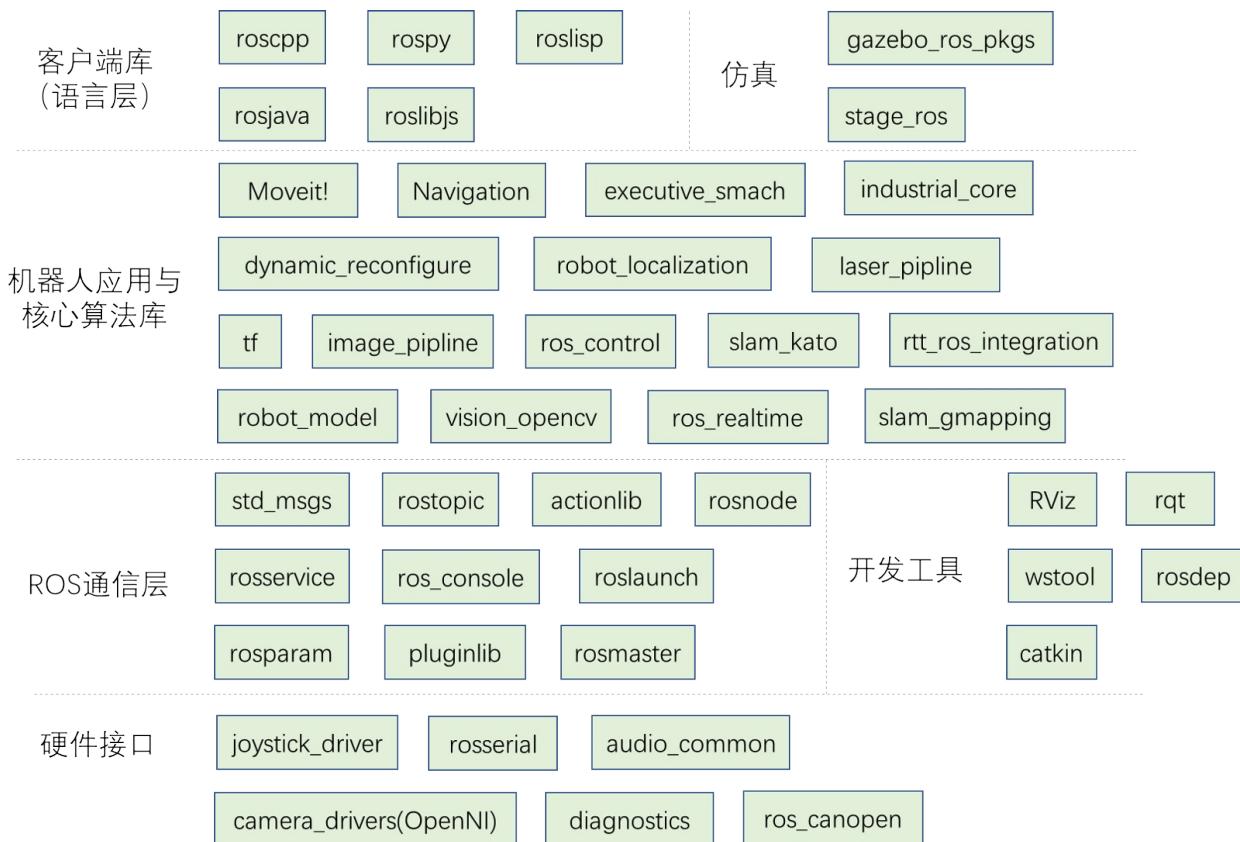
目前ROS支持的Clinet Library包括：

Client Library	介绍
roscpp	ROS的C++库，是目前最广泛应用的ROS客户端库，执行效率高
rospy	ROS的Python库，开发效率高，通常用在对运行时间没有太大要求的场合，例如配置、初始化等操作
roslisp	ROS的LISP库
rosacs	Mono/.NET库，可用任何Mono/.NET语言，包括C#，Iron Python，Iron Ruby等
rosgo	ROS Go语言库
rosjava	ROS Java语言库
rosnodejs	Javascript客户端库
...	...

目前最常用的只有roscpp和rospy，而其余的语言版本基本都还是测试版。

从开发客户端库的角度看，一个客户端库，至少需要能够包括master注册、名称管理、消息收发等功能。这样才能给开发者提供对ROS通信架构进行配置的方法。

整个ROS包括的packages如下，你可以看到roscpp、rospy处于什么位置。



## 6.1.2 roscpp

roscpp位于 /opt/ros/kinetic 之下，用C++实现了ROS通信。在ROS中，C++的代码是通过catkin这个编译系统（扩展的CMake）来进行编译构建的。所以简单地理解，你也可以把roscpp就当作为一个C++的库，我们创建一个CMake工程，在其中include了roscpp等ROS的libraries，这样就可以在工程中使用ROS提供的函数了。

通常我们要调用ROS的C++接口，首先就需要 `#include <ros/ros.h>`。

roscpp的主要部分包括：

- `ros::init()`：解析传入的ROS参数，创建node第一步需要用到的函数
- `ros::NodeHandle`：和topic、service、param等交互的公共接口
- `ros::master`：包含从master查询信息的函数
- `ros::this_node`：包含查询这个进程(node)的函数
- `ros::service`：包含查询服务的函数
- `ros::param`：包含查询参数服务器的函数，而不需要用到NodeHandle
- `ros::names`：包含处理ROS图资源名称的函数

具体可见：<http://docs.ros.org/api/roscpp/html/index.html>

以上功能可以分为以下几类：

- Initialization and Shutdown 初始与关闭

- Topics 话题
- Services 服务
- Parameter Server 参数服务器
- Timers 定时器
- NodeHandles 节点句柄
- Callbacks and Spinning 回调和自旋（或者翻译叫轮询？）
- Logging 日志
- Names and Node Information 名称管理
- Time 时钟
- Exception 异常

看到这么多接口，千万别觉得复杂，我们日常开发并不会用到所有的功能，你只需对要有一些印象，掌握几个比较常见和重要的用法就足够了。下面我们就来介绍关键的用法。

## 6.2 节点初始、关闭以及**NodeHandle**

当执行一个ROS程序，就被加载到了内存中，就成为了一个进程，在ROS里叫做节点。每一个ROS的节点尽管功能不同，但都有必不可少的一些步骤，比如初始化、销毁，需要通行的场景通常都还需要节点的句柄。这一节我们来学习**Node**最基本的一些操作。

### 6.2.1 初始化节点

对于一个C++写的ROS程序，之所以它区别于普通C++程序，是因为代码中做了两层工作：

1. 调用了 `ros::init()` 函数，从而初始化节点的名称和其他信息，一般我们ROS程序一开始都会以这种方式开始。
2. 创建 `ros::NodeHandle` 对象，也就是节点的句柄，它可以用来创建Publisher、Subscriber以及做其他事情。

句柄(Handle)这个概念可以理解为一个“把手”，你握住了门把手，就可以很容易把整扇门拉开，而不必关心门是什么样子。**NodeHandle**就是对节点资源的描述，有了它你就可以操作这个节点了，比如为程序提供服务、监听某个topic上的消息、访问和修改param等等。

### 6.2.2 关闭节点

通常我们要关闭一个节点可以直接在终端上按 `ctrl + c`，系统会自动触发SIGINT句柄来关闭这个进程。你也可以通过调用 `ros::shutdown()` 来手动关闭节点，但通常我们很少这样做。

以下是一个节点初始化、关闭的例子。

```
#include<ros/ros.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "your_node_name");
    ros::NodeHandle nh;
    //....节点功能
    //....
    ros::spin(); //用于触发topic、service的响应队列
    return 0;
}
```

这段代码是最常见的一个ROS程序的执行步骤，通常要启动节点，获取句柄，而关闭的工作系统自动帮我们完成，如果有特殊需要你也可以自定义。你可能很关心句柄可以用来做些什么，接下来我们来看看**NodeHandle**常用的成员函数。

## 6.2.3 NodeHandle常用成员函数

NodeHandle是Node的句柄，用来对当前节点进行各种操作。在ROS中，NodeHandle是一个定义好的类，通过 `include<ros/ros.h>`，我们可以创建这个类，以及使用它的成员函数。

NodeHandle常用成员函数包括：

```
//创建话题的publisher
ros::Publisher advertise(const string &topic, uint32_t queue_size, bool latch=false);
//第一个参数为发布话题的名称
//第二个是消息队列的最大长度，如果发布的消息超过这个长度而没有被接收，那么就的消息就会出队。通常设为一个较小的数即可。
//第三个参数是否锁存。某些话题并不是会以某个频率发布，比如/map这个topic，只有在初次订阅或者地图更新这两种情况下，/map才会发布消息。这里就用到了锁存。

//创建话题的subscriber
ros::Subscriber subscribe(const string &topic, uint32_t queue_size, void(*)(M));
//第一个参数是订阅话题的名称
//第二个参数是订阅队列的长度，如果受到的消息都没来得及处理，那么新消息入队，就消息就会出队
//第三个参数是回调函数指针，指向回调函数来处理接收到的消息

//创建服务的server，提供服务
ros::ServiceServer advertiseService(const string &service, bool(*srv_func)(Mreq &, Mres &));
//第一个参数是service名称
//第二个参数是服务函数的指针，指向服务函数。指向的函数应该有两个参数，分别接受请求和响应。

//创建服务的client
ros::ServiceClient serviceClient(const string &service_name, bool persistent=false);
//第一个参数是service名称
//第二个参数用于设置服务的连接是否持续，如果为true，client将会保持与远程主机的连接，这样后续的请求会快一些。通常我们设为false

//查询某个参数的值
bool getParam(const string &key, std::string &s);
bool getParam (const std::string &key, double &d) const;
bool getParam (const std::string &key, int &i) const;
//从参数服务器上获取key对应的值，已重载了多个类型

//给参数赋值
void setParam (const std::string &key, const std::string &s) const;
void setParam (const std::string &key, const char *s) const;
void setParam (const std::string &key, int i) const;
//给key对应的val赋值，重载了多个类型的val
```

可以看出，NodeHandle对象在ROS C++程序里非常重要，各种类型的通信都需要用NodeHandle来创建完成。下面我们具体来看topic、service和param这三种基本通信方式的写法。



## 6.3 topic in roscpp

### 6.3.1 Topic通信

Topic是ROS里一种异步通信的模型，一般是节点间分工明确，有的只负责发送，有的只负责接收处理。对于绝大多数的机器人应用场景，比如传感器数据收发，速度控制指令的收发，Topic模型是最适合的通信方式。

为了讲明白topic通信的编程思路，我们首先来看 `topic_demo` 中的代码，这个程序是一个消息收发的例子：自定义一个类型为gps的消息（包括位置x，y和工作状态state信息），一个node以一定频率发布模拟的gps消息，另一个node接收并处理，算出到原点的距离。源代码见 `ROS-Academy-for-Beginners/topic_demo`

### 6.3.2 创建gps消息

在代码中，我们会用到自定义类型的gps消息，因此就需要来自定义gps消息，在msg路径下创建 `gps.msg`：见 `topic_demo/msg/gps.msg`

```
string state    #工作状态
float32 x        #x坐标
float32 y        #y坐标
```

以上就定义了一个gps类型的消息，你可以把它理解成一个C语言中的结构体，类似于

```
struct gps
{
    string state;
    float32 x;
    float32 y;
}
```

在程序中对一个gps消息进行创建修改的方法和对结构体的操作一样。

当你创建完了msg文件，记得修改 `CMakeLists.txt` 和 `package.xml`，从而让系统能够编译自定义消息。在 `CMakeLists.txt` 中需要改动

```

find_package(catkin REQUIRED COMPONENTS
roscpp
std_msgs
message_generation    #需要添加的地方
)

add_message_files(FILES gps.msg)
#catkin在cmake之上新增的命令，指定从哪个消息文件生成

generate_messages(DEPENDENCIES std_msgs)
#catkin新增的命令，用于生成消息
#DEPENDENCIES后面指定生成msg需要依赖其他什么消息，由于gps.msg用到了float32这种ROS标准消息，因此需要再把std_msgs作为依赖

```

package.xml 中需要的改动

```

<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>

```

当你完成了以上所有工作，就可以回到工作空间，然后编译了。编译完成之后会在 `devel` 路径下生成 `gps.msg` 对应的头文件，头文件按照C++的语法规则定义了 `topic_demo::gps` 类型的数据。

要在代码中使用自定义消息类型，只要 `#include <topic_demo/gps.h>`，然后声明，按照对结构体操作的方式修改内容即可。

```

topic_demo::gps mygpsmsg;
mygpsmsg.x = 1.6;
mygpsmsg.y = 5.5;
mygpsmsg.state = "working";

```

### 6.3.3 消息发布节点

定义完了消息，就可以开始写ROS代码了。通常我们会把消息收发的两端分成两个节点来写，一个节点就是一个完整的C++程序。

见 `topic_demo/src/talker.cpp`

```
#include <ros/ros.h>
#include <topic_demo/gps.h> //自定义msg产生的头文件

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); //用于解析ROS参数，第三个参数为本节点名
    ros::NodeHandle nh; //实例化句柄，初始化node

    topic_demo::gps msg; //自定义gps消息并初始化
    ...

    ros::Publisher pub = nh.advertise<topic_demo::gps>("gps_info", 1); //创建publisher，往
    "gps_info"话题上发布消息
    ros::Rate loop_rate(1.0); //定义发布的频率，1HZ
    while (ros::ok()) //循环发布msg
    {
        ... //处理msg
        pub.publish(msg); //以1HZ的频率发布msg
        loop_rate.sleep(); //根据前面的定义的loop_rate, 设置1s的暂停
    }
    return 0;
}
```

机器人上几乎所有的传感器，几乎都是按照固定频率发布消息这种通信方式来传输数据，只是发布频率和数据类型的区别。

### 6.3.4 消息接收节点

见 `topic_demo/src/listener.cpp`

```

#include <ros/ros.h>
#include <topic_demo/gps.h>
#include <std_msgs/Float32.h>

void gpsCallback(const topic_demo::gps::ConstPtr &msg)
{
    std_msgs::Float32 distance; //计算离原点(0,0)的距离
    distance.data = sqrt(pow(msg->x, 2)+pow(msg->y, 2));
    ROS_INFO("Listener: Distance to origin = %f, state: %s",distance.data,msg->state.c
_str()); //输出
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("gps_info", 1, gpsCallback); //设置回调函数gpsCallb
ack
    ros::spin(); //ros::spin()用于调用所有可触发的回调函数，将进入循环，不会返回，类似于在循环里反复
    调用spinOnce()
    //而ros::spinOnce()只会去触发一次
    return 0;
}

```

在topic接收方，有一个比较重要的概念，就是回调(**CallBack**)，在本例中，回调就是预先给`gps_info`话题传来的消息准备一个回调函数，你事先定义好回调函数的操作，本例中是计算到原点的距离。只有当有消息来时，回调函数才会被触发执行。具体去触发的命令就是`ros::spin()`，它会反复的查看有没有消息来，如果有就会让回调函数去处理。

因此千万不要认为，只要指定了回调函数，系统就回去自动触发，你必须`ros::spin()`或者`ros::spinOnce()`才能真正使回调函数生效。

## 6.3.5 CMakeLists.txt文件修改

在`CMakeLists.txt`添加以下内容，生成可执行文件

```

add_executable(talker src/talker.cpp) #生成可执行文件talker
add_dependencies(talker topic_demo_generate_messages_cpp)
#表明在编译talker前，必须先生编译完成自定义消息
#必须添加add_dependencies，否则找不到自定义的msg产生的头文件
#表明在编译talker前，必须先生编译完成自定义消息
target_link_libraries(talker ${catkin_LIBRARIES}) #链接

add_executable(listener src/listener.cpp ) #声称可执行文件listener
add_dependencies(listener topic_demo_generate_messages_cpp)
target_link_libraries(listener ${catkin_LIBRARIES})#链接

```

以上cmake语句告诉catkin编译系统如何去编译生成我们的程序。这些命令都是标准的cmake命令，如果不理解，请查阅cmake教程。

之后经过 `catkin_make`，一个自定义消息+发布接收的基本模型就完成了。

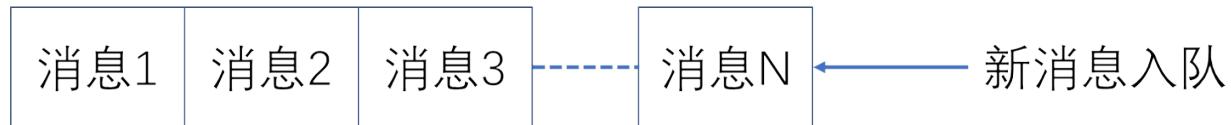
## 扩展：回调函数与**spin()**方法

回调函数在编程中是一种重要的方法，在维基百科上的解释是：

In computer programming, a callback is any executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at a given time.

回调函数作为参数被传入到了另一个函数中（在本例中传递的是函数指针），在未来某个时刻（当有新的message到达），就会立即执行。Subscriber接收到消息，实际上是先把消息放到一个队列中去，如图所示。队列的长度在Subscriber构建的时候设置好了。当有spin函数执行，就会去处理消息队列中队首的消息。

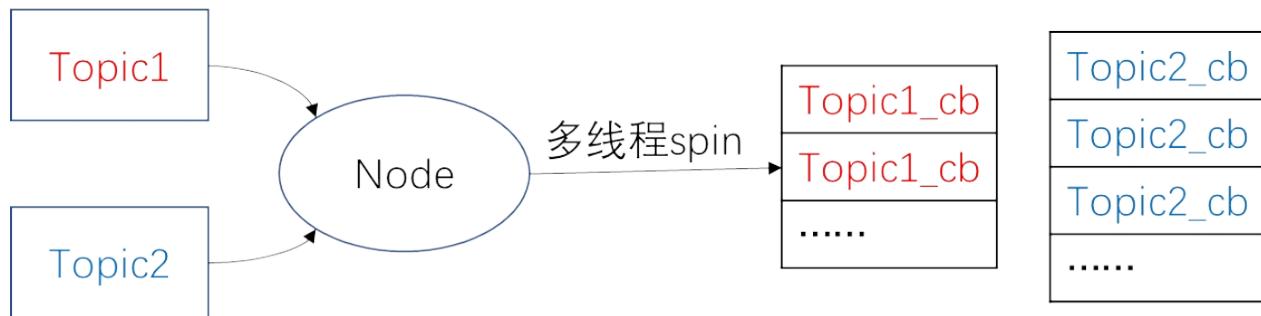
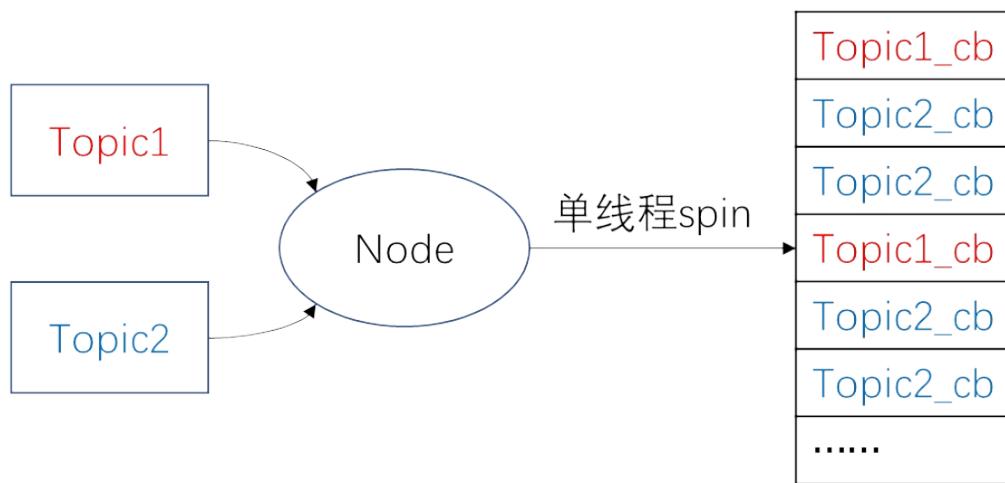
### 消息订阅队列(回调队列)



spin具体处理的方法又可分为阻塞/非阻塞,单线程/多线程，在ROS函数接口层面我们有4种spin的方式：

spin方法	阻塞	线程
<code>ros::spin()</code>	阻塞	单线程
<code>ros::spinOnce()</code>	非阻塞	单线程
<code>ros::MultiThreadedSpin()</code>	阻塞	多线程
<code>ros::AsyncMultiThreadedSpin()</code>	非阻塞	多线程

阻塞与非阻塞的区别我们已经讲了，下面来看看单线程与多线程的区别：



我们常用的 `spin()`、`spinOnce()` 是单个线程逐个处理回调队列里的数据。有些场合需要用到多线程分别处理，则可以用到 `MultiThreadedSpin()`、`AsyncMultiThreadedSpin()`。

## 6.4 service in roscpp

### 6.4.1 Service通信

Service是一种请求-反馈的通信机制。请求的一方通常被称为客户端，提供服务的一方叫做服务器端。Service机制相比于Topic的不同之处在于：

1. 消息的传输是双向的，有反馈的，而不是单一的流向。
2. 消息往往不会以固定频率传输，不连续，而是在需要时才会向服务器发起请求。

在ROS中如何请求或者提供一个服务，我们来看 `service_demo` 的代码：一个节点发出服务请求（姓名，年龄），另一个节点进行服务响应，答复请求。

### 6.4.2 创建Greeting服务

创建 `service_demo/Greeting.srv` 文件，内容包括：

```
string name      #短横线上边部分是服务请求的数据
int32 age
---
string feedback #短横线下面是服务回传的内容。
```

`.srv`格式的文件创建后，也需要修改 `CMakeLists.txt`，在其中加入

```
add_service_files(FILES Greeting.srv)
```

其余与添加`msg`的改动一样。然后进行 `catkin_make`，系统就会生成在代码中可用的`Greeting`类型。在代码中使用，只需要 `#include <service_demo/Greeting.h>`，然后即可创建该类型的`srv`。

```
service_demo::Greeting grt; //grt分为grt.request和grt.response两部分
grt.request.name = "HAN"; //不能用grt.name或者grt.age来访问
grt.request.age = "20";
...
```

新生成的`Greeting`类型的服务，其结构体的风格更为明显，可以这么理解，一个`Greeting`服务结构体中嵌套了两个结构体，分别是请求和响应：

```

struct Greeting
{
    struct Request
    {
        string name;
        int age;
    }request;
    struct Response
    {
        string feedback;
    }response;
}

```

## 6.4.3 创建提供服务节点(server)

service\_demo/srv/server.cpp 内容如下：

```

#include <ros/ros.h>
#include <service_demo/Greeting.h>

bool handle_function(service_demo::Greeting::Request &req, service_demo::Greeting::Response &res){
    //显示请求信息
    ROS_INFO("Request from %s with age %d", req.name.c_str(), req.age);
    //处理请求，结果写入response
    res.feedback = "Hi " + req.name + ". I'm server!";
    //返回true，正确处理了请求
    return true;
}

int main(int argc, char** argv){
    ros::init(argc, argv, "greetings_server");           //解析参数，命名节点
    ros::NodeHandle nh;                                //创建句柄，实例化node
    ros::ServiceServer service = nh.advertiseService("greetings", handle_function); / 
    //写明服务的处理函数
    ros::spin();
    return 0;
}

```

在以上代码中，服务的处理操作都写在 `handle_function()` 中，它的输入参数就是 `Greeting` 的 `Request` 和 `Response` 两部分，而非整个 `Greeting` 对象。通常在处理函数中，我们对 `Request` 数据进行需要的操作，将结果写入到 `Response` 中。在 `roscpp` 中，处理函数返回值是 `bool` 型，也就是服务是否成功执行。不要理解成输入 `Request`，返回 `Response`，在 `rospy` 中是这样的。

## 6.4.4 创建服务请求节点(client)

service\_demo/srv/client.cpp 内容如下：

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "greetings_client");// 初始化，节点命名为"greetings_client"
    ros::NodeHandle nh;
    ros::ServiceClient client = nh.serviceClient<service_demo::Greeting>("greetings");
    // 定义service客户端，service名字为“greetings”，service类型为Service_demo

    // 实例化srv，设置其request消息的内容，这里request包含两个变量，name和age，见Greeting.srv
    service_demo::Greeting srv;
    srv.request.name = "HAN";
    srv.request.age = 20;

    if (client.call(srv))
    {
        // 注意我们的response部分中的内容只包含一个变量response，另，注意将其转变成字符串
        ROS_INFO("Response from server: %s", srv.response.feedback.c_str());
    }
    else
    {
        ROS_ERROR("Failed to call service Service_demo");
        return 1;
    }
    return 0;
}
```

以上代码比较关键的地方有两处，一个是建立一个ServiceClient，另一个是开始调用服务。建立client需要用 `nh.serviceClient<service_demo::Greeting>("greetings")`，指明服务的类型和服务的名称。而调用时可以直接用 `client.call(srv)`，返回结果不是response，而是是否成功调用远程服务。

`CMakeLists.txt` 和 `pacakge.xml` 修改方法和 `topic_demo` 修改方法类似，不再赘述。

## 6.5 param in rosCPP

### 6.5.1 Parameter Server

严格来说，param并不能称作一种通信方式，因为它往往只是用来存储一些静态的设置，而不是动态变化的。所以关于param的操作非常轻巧，非常简单。关于param的API，rosCPP为我们提供了两套，一套是放在 `ros::param namespace` 下，另一套是在 `ros::NodeHandle` 下，这两套API的操作完全一样，用哪一个取决于你的习惯。

### 6.5.2 param\_demo

我们来看看在C++中如何进行param\_demo的操作，`param_demo/param.cpp` 文件，内容包括：

```
#include<ros/ros.h>

int main(int argc, char **argv){
    ros::init(argc, argv, "param_demo");
    ros::NodeHandle nh;
    int parameter1, parameter2, parameter3, parameter4, parameter5;

    //Get Param的三种方法
    //① ros::param::get()获取参数“param1”的value，写入到parameter1上
    bool ifget1 = ros::param::get("param1", parameter1);
    //② ros::NodeHandle::getParam()获取参数，与①作用相同
    bool ifget2 = nh.getParam("param2", parameter2);
    //③ ros::NodeHandle::param()类似于①和②
    //但如果get不到指定的param，它可以给param指定一个默认值(如33333)
    nh.param("param3", parameter3, 33333);

    if(ifget1) //param是否取得
    ...

    //Set Param
    //① ros::param::set()设置参数
    parameter4 = 4;
    ros::param::set("param4", parameter4);
    //② ros::NodeHandle::setParam()设置参数
    parameter5 = 5;
    nh.setParam("param5", parameter5);

    //Check Param
    //① ros::NodeHandle::hasParam()
    bool ifparam5 = nh.hasParam("param5");
    //② ros::param::has()
    bool ifparam6 = ros::param::has("param6");

    //Delete Param
    //① ros::NodeHandle::deleteParam()
    bool ifdeleted5 = nh.deleteParam("param5");
    //② ros::param::del()
    bool ifdeleted6 = ros::param::del("param6");
    ...
}
```

以上是rosccpp中对param进行增删改查所有操作的方法，非常直观。

### 6.5.3 param\_demo 中的 launch 文件

实际项目中我们对参数进行设置，尤其是添加参数，一般都不是在程序中，而是在launch文件中。因为launch文件可以方便的修改参数，而写成代码之后，修改参数必须重新编译。因此我们会在launch文件中将param都定义好，比如这个demo正确的打开方式应该是 roslaunch

```
param_demo param_demo_cpp.launch
```

param\_demo/launch/param\_demo\_cpp.launch 内容为：

```
<launch>
    <!--param参数配置-->
    <param name="param1" value="1" />
    <param name="param2" value="2" />

    <!--rosparam参数配置-->
    <rosparam>
        param3: 3
        param4: 4
        param5: 5
    </rosparam>
    <!--以上写法将参数转成YAML文件加载，注意param前面必须为空格，不能用Tab，否则YAML解析错误-->
    <!--rosparam file="$(find robot_sim_demo)/config/xbot2_control.yaml" command="load
" /-->
    <node pkg="param_demo" type="param_demo" name="param_demo" output="screen" />
</launch>
```

通过和两个标签我们设置好了5个param，从而在之前的代码中进行增删改查的操作。

## 6.6 时钟

### 6.6.1 Time 与 Duration

ROS里经常用到的一个功能就是时钟，比如计算机器人移动距离、设定一些程序的等待时间、设定计时器等等。rosCPP同样给我们提供了时钟方面的操作。

具体来说，rosCPP里有两种时间的表示方法，一种是时刻（`ros::Time`），一种是时长（`ros::Duration`）。无论是Time还是Duration都具有相同的表示方法：

...

```
int32 sec
```

```
int32 nsec
```

...

Time/Duration都由秒和纳秒组成。要使用Time和Duration，需要 `#include <ros/time.h>` 和 `#include <ros/duration.h>`

```
ros::Time begin = ros::Time::now(); //获取当前时间
ros::Time at_some_time1(5,20000000); //5.2s
ros::Time at_some_time2(5.2) //同上，重载了float类型和两个uint类型的构造函数
ros::Duration one_hour(60*60,0); //1h

double secs1 = at_some_time1.toSec(); //将Time转为double型时间
double secs2 = one_hour.toSec(); //将Duration转为double型时间
```

Time和Duration表示的概念并不相同，Time指的是某个时刻，而Duration指的是某个时段，尽管他们的数据结构都相同，但是用在不同的场景下。ROS为我们重载了Time、Duration类型之间的加减运算，比如：

```
ros::Time t1 = ros::Time::now() - ros::Duration(5.5); //t1是5.5s前的时刻，Time加减Duration
n返回都是Time
ros::Time t2 = ros::Time::now() + ros::Duration(3.3); //t2是当前时刻往后推3.3s的时刻
ros::Duration d1 = t2 - t1; //从t1到t2的时长，两个Time相减返回Duration类型
ros::Duration d2 = d1 - ros::Duration(0,300); //两个Duration相减，还是Duration
```

以上是Time、Duration之间的加减运算，要注意没有Time+Time的做法。

### 6.6.2 sleep

通常在机器人任务执行中可能有需要等待的场景，这时就要用到sleep功能，rosCPP中提供了两种sleep的方法：

```

ros::Duration(0.5).sleep(); //用Duration对象的sleep方法休眠

ros::Rate r(10); //10HZ
while(ros::ok())
{
    r.sleep();
    //定义好sleep的频率，Rate对象会自动让整个循环以10hz休眠，即使有任务执行占用了时间
}

```

## 6.6.3 Timer

Rate的功能是指定一个频率，让某些动作按照这个频率来循环执行。与之类似的是ROS中的定时器Timer，它是通过设定回调函数和触发时间来实现某些动作的反复执行，创建方法和topic中的subscriber很像。

```

void callback1(const ros::TimerEvent&)
{
    ROS_INFO("Callback 1 triggered");
}

void callback2(const ros::TimerEvent&)
{
    ROS_INFO("Callback 2 triggered");
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;

    ros::Timer timer1 = n.createTimer(ros::Duration(0.1), callback1); //timer1每0.1s触发
    一次callback1函数
    ros::Timer timer2 = n.createTimer(ros::Duration(1.0), callback2); //timer2每1.0s触发
    一次callback2函数

    ros::spin(); //千万别忘了spin，只有spin了才能真正去触发回调函数

    return 0;
}

```

## 6.7 日志和异常

### 6.7.1 Log

ROS为开发者和用户提供了一套日志记录和输出系统，这套系统的实现方式是基于topic，也就是每个节点都会把一些日志信息发到一个统一的topic上去，这个topic就是 /rosout。

rosout 本身也是一个node，它专门负责进行日志的记录。我们在启动master的时候，系统就会附带启动rosout。

在roscpp中进行日志的输出，需要先 `include <ros/console.h>` ,这个头文件包括了五个级别的日志输出接口，分别是：

- DEBUG
- INFO
- WARN
- ERROR
- FATAL 用法非常简单：

```
ROS_DEBUG("The velocity is %f", vel);
ROS_WARN("Warn: the use is deprecated.");
ROS_FATAL("Cannot start this node.");
...
```

当然也可以在一些特定场景，特定条件下输出，不过对于普通开发者来说可能用不到这么复杂的功能。具体可参考：<http://wiki.ros.org/roscpp/Overview/Logging>

### 6.7.2 Exception

roscpp中有两种异常类型，当有以下两种错误时，就会抛出异常：

```
ros::InvalidNodeNameException
当无效的基础名称传给ros::init(), 通常是名称中有/, 就会触发

ros::InvalidNameException
当无效名称传给了roscpp
```

# 第七章 **rospy**

## 本章简介

上一章我们学习了C++语音的ROS接口roscpp，本章我们来学习ROS的另一个接口rospy，也即是Python语音的接口。客户端库（Client Libarary）不仅仅指的是C++、Python语言的接口，其实是各种语言的接口统称。rospy中函数的定义，函数的用法都和roscpp不相同。

本章依旧需要一定的Python编程基础，不熟悉该语言需要先行学习Python编程基础。本章的主要内容有rospy的一些常用的函数，以及一些重要的类。结合这些内容，本章还给出了topic demo和服务 demo的具体格式和写法，方便读者更直观的学习rospy的编写。

## 7.1 rosipy与主要接口

### 7.1.1 rosipy vs roscpp

rosipy是Python版本的ROS客户端库，提供了Python编程需要的接口，你可以认为rosipy就是一个Python的模块(Module)。这个模块位于 /opt/ros/kineetic/lib/python2.7/dist-packages/rospy 之中。

rosipy包含的功能与roscpp相似，都有关于node、topic、service、param、time相关的操作。但同时rosipy和roscpp也有一些区别：

1. rosipy没有一个NodeHandle，像创建publisher、subscriber等操作都被直接封装成了rosipy中的函数或类，调用起来简单直观。
2. rosipy一些接口的命名和roscpp不一致，有些地方需要开发者注意，避免调用错误。

相比于C++的开发，用Python来写ROS程序开发效率大大提高，诸如显示、类型转换等细节不再需要我们注意，节省时间。但Python的执行效率较低，同样一个功能用Python运行的耗时会高于C++。因此我们开发SLAM、路径规划、机器视觉等方面的算法时，往往优先选择C++。

ROS中绝大多数基本指令，例如 rostopic，roslaunch 都是用python开发的，简单轻巧。

### 7.1.2 ROS中Python代码的组织方式

要介绍rosipy，就不得不提Python代码在ROS中的组织方式。通常来说，Python代码有两种组织方式，一种是单独的一个Python脚本，适用于简单的程序，另一种是Python模块，适合体量较大的程序。

#### 单独的Python脚本

对于一些小体量的ROS程序，一般就是一个Python文件，放在script/路径下，非常简单。

```
your_package
|- script/
|- your_script.py
|-
|...
```

#### Python模块

当程序的功能比较复杂，放在一个脚本里搞不定时，就需要把一些功能放到Python Module里，以便其他的脚本来调用。ROS建议我们按照以下规范来建立一个Python的模块：

```
your_package
|- src/
| |- your_package/
| | |- __init__.py
| | |- modulefiles.py
| | |- scripts/
| | |- your_script.py
| |- setup.py
```

在src下建立一个与你的package同名的路径，其中存放 `__init__.py` 以及你的模块文件。这样就建立好了ROS规范的Python模块，你可以在你的脚本中调用。如果你不了解`init.py`的作用，可以参考这篇博客<http://www.cnblogs.com/Lands-ljk/p/5880483.html> ROS中的这种Python模块组织规范与标准的Python模块规范并不完全一致，你当然可以按照Python的标准去建立一个模块，然后在你的脚本中调用，但是我们还是建议按照ROS推荐的标准来写，这样方便别人去阅读。

通常我们常用的ROS命令，大多数其实都是一个个Python模块，源代码存放在ros\_comm仓库的tools路径下：[https://github.com/ros/ros\\_comm/tree/lunar-devel/tools](https://github.com/ros/ros_comm/tree/lunar-devel/tools) 你可以看到每一个命令行工具（如rosbag、rosmg）都是用模块的形式组织核心代码，然后在 `script/` 下建立一个脚本来调用模块。

### 7.1.3 常用 rospy 的 API

这里分类整理了rospy常见的一些用法，请你浏览一遍，建立一个初步的影响。具体API请查看<http://docs.ros.org/api/rospy/html/rospy-module.html>

#### Node相关

返回值	方法	作用
	<code>rospy.init_node(name, argv=None, anonymous=False)</code>	注册和初始化node
MasterProxy	<code>rospy.get_master()</code>	获取master的句柄
bool	<code>rospy.is_shutdown()</code>	节点是否关闭
	<code>rospy.on_shutdown(fn)</code>	在节点关闭时调用fn函数
str	<code>get_node_uri()</code>	返回节点的URI
str	<code>get_name()</code>	返回本节点的全名
str	<code>get_namespace()</code>	返回本节点的名字空间
...	...	...

## Topic相关

函数：

返回值	方法	作用
<code>[[str, str]]</code>	<code>get_published_topics()</code>	返回正在被发布的所有topic名称和类型
Message	<code>wait_for_message(topic, topic_type, time_out=None)</code>	等待某个topic的message
	<code>spin()</code>	触发topic或service的回调/处理函数，会阻塞直到关闭节点
...	...	...

Publisher类：

返回值	方法	作用
	<code>init(self, name, data_class, queue_size=None)</code>	构造函数
	<code>publish(self, msg)</code>	发布消息
str	<code>unregister(self)</code>	停止发布
...	...	...

Subscriber类：

返回值	方法	作用
	<code>init_(self, name, data_class, call_back=None, queue_size=None)</code>	构造函数
	<code>unregister(self, msg)</code>	停止订阅
...	...	...

## Service相关

函数：

返回值	方法	作用
	<code>wait_for_service(service, timeout=None)</code>	阻塞直到服务可用
...	...	...

Service类(server)：

返回值	方法	作用
	<code>init(self, name, service_class, handler)</code>	构造函数，handler为处理函数，service_class为srv类型
	<code>shutdown(self)</code>	关闭服务的server
...	...	...

ServiceProxy类(client)：

返回值	方法	作用
	<code>init(self, name, service_class)</code>	构造函数，创建client
	<code>call(self, args, *kwds)</code>	发起请求
	<code>call(self, args, *kwds)</code>	同上
	<code>close(self)</code>	关闭服务的client
...	...	...

## Param相关

函数：

返回值	方法	作用
XmlRpcLegalValue	get_param(param_name, default=_unspecified)	获取参数的值
[str]	get_param_names()	获取参数的名称
	set_param(param_name, param_value)	设置参数的值
	delete_param(param_name)	删除参数
bool	has_param(param_name)	参数是否存在于参数服务器上
str	search_param()	搜索参数
...	...	...

## 时钟相关

函数：

返回值	方法	作用
Time	get_rostime()	获取当前时刻的Time对象
float	get_time()	返回当前时间，单位秒
	sleep(duration)	执行挂起
...	...	...

Time类：

返回值	方法	作用
	init(self, secs=0, nsecs=0)	构造函数
Time	now()	静态方法 返回当前时刻的Time对象
...	...	...

Duration类：

返回值	方法	作用
	init(self, secs=0, nsecs=0)	构造函数
...	...	...

## 7.2 topic in rospy

与5.3节类似，我们用python来写一个节点间消息收发的demo，同样还是创建一个自定义的gps类型的消息，一个节点发布模拟的gps信息，另一个接收和计算距离原点的距离。

### 7.2.1 自定义消息的生成

gps.msg 定义如下：

```
string state    #工作状态
float32 x       #x坐标
float32 y       #y坐标
```

我们需要修改 CMakeLists.txt 文件，方法见5.3节，这里需要强调一点的就是，对创建的msg进行 catkin\_make 会在 ~/catkin\_ws/devel/lib/python2.7/dist-packages/topic\_demo 下生成msg模块（module）。有了这个模块，我们就可以在python程序中 from topic\_demo.msg import gps ,从而进行gps类型消息的读写。

### 7.2.2 消息发布节点

与C++的写法类似，我们来看topic用Python如何编写程序，

见 topic\_demo/scripts/pytalker.py :

```

#!/usr/bin/env python
#coding=utf-8
import rospy
#导入自定义的数据类型
from topic_demo.msg import gps

def talker():
    #Publisher 函数第一个参数是话题名称，第二个参数 数据类型，现在就是我们定义的msg 最后一个是缓冲区的大小
    #queue_size: None (不建议) #这将设置为阻塞式同步收发模式！
    #queue_size: 0 (不建议) #这将设置为无限缓冲区模式，很危险！
    #queue_size: 10 or more #一般情况下，设为10。queue_size太大了会导致数据延迟不同步。
    pub = rospy.Publisher('gps_info', gps, queue_size=10)
    rospy.init_node('pytalker', anonymous=True)
    #更新频率是1hz
    rate = rospy.Rate(1)
    x=1.0
    y=2.0
    state='working'
    while not rospy.is_shutdown():
        #计算距离
        rospy.loginfo('Talker: GPS: x=%f ,y= %f',x,y)
        pub.publish(gps(state,x,y))
        x=1.03*x
        y=1.01*y
        rate.sleep()

if __name__ == '__main__':
    talker()

```

以上代码与C++的区别体现在这几个方面：

1. `rospy`创建和初始化一个node，不再需要用`NodeHandle`。`rospy`中没有设计`NodeHandle`这个句柄，我们创建topic、service等等操作都直接用`rospy`里对应的方法就行。
2. `rospy`中节点的初始化并一定得放在程序的开头，在`Publisher`建立后再初始化也没问题。
3. 消息的创建更加简单，比如`gps`类型的可以直接用类似于构造函数的方式`gps(state,x,y)`来创建。
4. 日志的输出方式不同，C++中是`ROS_INFO()`，而Python中是`rospy.loginfo()`
5. 判断节点是否关闭的函数不同，C++用的是`ros::ok()`而Python中的接口是`rospy.is_shutdown()`

通过以上的区别可以看出，`roscpp`和`rospy`的接口并不一致，在名称上要尽量避免混用。在实现原理上，两套客户端库也有各自的实现，并没有基于一个统一的核心库来开发。这也是ROS在设计上不足的地方。

ROS2就解决了这个问题，ROS2中的客户端库包括了`rclcpp (ROS Clinet Library C++)`、`rclpy (ROS Client Library Python)`，以及其他语言的版本，他们都是基于一个共同的核心ROS客户端库`rcl`来开发的，这个核心库由C语言实现。

## 7.2.3 消息订阅节点

见 `topic_demo/scripts/pylistener.py` :

```
#!/usr/bin/env python
#coding=utf-8
import rospy
import math
#导入msg
from topic_demo.msg import gps

#回调函数输入的应该是msg
def callback(gps):
    distance = math.sqrt(math.pow(gps.x, 2)+math.pow(gps.y, 2))
    rospy.loginfo('Listener: GPS: distance=%f, state=%s', distance, gps.state)

def listener():
    rospy.init_node('pylistener', anonymous=True)
    #Subscriber函数第一个参数是topic的名称，第二个参数是接受的数据类型 第三个参数是回调函数的名称
    rospy.Subscriber('gps_info', gps, callback)
    rospy.spin()

if __name__ == '__main__':
    listener()
```

在订阅节点的代码里，rospy与roscpp有一个不同的地方：rospy里没有 `spinOnce()`，只有 `spin()`。

建立完talker和listener之后，经过 `catkin_make`，就完成了python版的topic通信模型。

## 7.3 Service in rospy

本节用python来写一个节点间，利用Service通信的demo，与5.4类似，创建一个节点，发布模拟的gps信息，另一个接收和计算距离原点的距离。

### 7.3.1 srv文件

在5.4节，我们已经说过要建立一个名为 `Greeting.srv` 的服务文件，内容如下：

```
string name #短横线上边部分是服务请求的数据
int32 age
--- #短横线下面是服务回传的内容
string feedback
```

然后修改 `CMakeLists.txt` 文件。ROS的catkin编译系统会将你自定义的msg、srv（甚至还有action）文件自动编译构建，生成对应的C++、Python、LISP等语言下可用的库或模块。许多初学者错误地以为，只要建立了一个msg或srv文件，就可以直接在程序中使用，这是不对的，必须在 `CMakeLists.txt` 中添加关于消息创建、指定消息/服务文件那几个宏命令。

### 7.3.2 创建提供服务节点(server)

见 `service_demo/scripts/server_demo.py` :

```
#!/usr/bin/env python
#coding=utf-8
import rospy
from service_demo.srv import *

def server_srv():
    # 初始化节点，命名为 "greetings_server"
    rospy.init_node("greetings_server")
    # 定义service的server端，service名称为"greetings"， service类型为Greeting
    # 收到的request请求信息将作为参数传递给handle_function进行处理
    s = rospy.Service("greetings", Greeting, handle_function)
    rospy.loginfo("Ready to handle the request:")
    # 阻塞程序结束
    rospy.spin()

def handle_function(req):
    # 注意我们是如何调用request请求内容的，是将其认为是一个对象的属性，在我们定义
    # 的Service_demo类型的service中，request部分的内容包含两个变量，一个是字符串类型的name，另外
    # 一个是整数类型的age
    rospy.loginfo('Request from %s with age %d', req.name, req.age)
    # 返回一个Service_demo.Response实例化对象，其实就是返回一个response的对象，其包含的内容为我们
    # 在Service_demo.srv中定义的
    # response部分的内容，我们定义了一个string类型的变量feedback，因此，此处实例化时传入字符串即可
    return GreetingResponse("Hi %s. I' server!"%req.name)

# 如果单独运行此文件，则将上面定义的server_srv作为主函数运行
if __name__=="__main__":
    server_srv()
```

以上代码中可以看出Python和C++在ROS服务通信时，server端的处理函数有区别：C++的handle\_function()传入的参数是整个srv对象的request和response两部分，返回值是bool型，显示这次服务是否成功的处理，也就是：

```
bool handle_function(service_demo::Greeting::Request &req, service_demo::Greeting::Res
ponse &res){
...
    return true;
}
```

而Python的handle\_function()传入的只有request，返回值是response，即：

```
def handle_function(req):
...
    return GreetingResponse("Hi %s. I' server!"%req.name)
```

这也是ROS在两种语言编程时的差异之一。相比来说Python的这种思维方式更加简单，符合我们的思维习惯。

### 7.3.3 创建服务请求节点(client)

service\_demo/srv/client.cpp 内容如下：

```
#!/usr/bin/env python
# coding:utf-8
import rospy
from service_demo.srv import *

def client_srv():
    rospy.init_node('greetings_client')
    # 等待有可用的服务 "greetings"
    rospy.wait_for_service("greetings")
    try:
        # 定义service客户端，service名称为"greetings"，service类型为Greeting
        greetings_client = rospy.ServiceProxy("greetings", Greeting)

        # 向server端发送请求，发送的request内容为name和age，其值分别为"HAN"， 20
        # 此处发送的request内容与srv文件中定义的request部分的属性是一致的
        #resp = greetings_client("HAN",20)
        resp = greetings_client.call("HAN",20)
        rospy.loginfo("Message From server:%s"%resp.feedback)
    except rospy.ServiceException, e:
        rospy.logwarn("Service call failed: %s"%e)

# 如果单独运行此文件，则将上面函数client_srv()作为主函数运行
if __name__=="__main__":
    client_srv()
```

以上代码中 `greetings_client.call("HAN",20)` 等同于 `greetings_client("HAN",20)`。

## 7.4 param与time

### 7.4.1 param\_demo

相比roscpp中有两套对param操作的API， rospy关于param的函数就显得简单多了，包括了增删查改等用法：

```
rospy.get_param() , rospy.set_param() , rospy.has_param() , rospy.delete_param() , ro  
spy.search_param() , rospy.get_param_names() 。
```

下面我们来看看param\_demo里的代码：

```
#!/usr/bin/env python
# coding:utf-8
import rospy

def param_demo():
    rospy.init_node("param_demo")
    rate = rospy.Rate(1)
    while(not rospy.is_shutdown()):
        #get param
        parameter1 = rospy.get_param("/param1")
        parameter2 = rospy.get_param("/param2", default=222)
        rospy.loginfo('Get param1 = %d', parameter1)
        rospy.loginfo('Get param2 = %d', parameter2)

        #delete param
        rospy.delete_param('/param2')

        #set param
        rospy.set_param('/param2', 2)

        #check param
        ifparam3 = rospy.has_param('/param3')
        if(ifparam3):
            rospy.loginfo('/param3 exists')
        else:
            rospy.loginfo('/param3 does not exist')

        #get all param names
        params = rospy.get_param_names()
        rospy.loginfo('param list: %s', params)

        rate.sleep()

if __name__=="__main__":
    param_demo()
```

## 7.4.2 time\_demo

### 时钟

rospy中的关于时钟的操作和rosCPP是一致的，都有Time、Duration和Rate三个类。首先，Time和Duration前者标识的是某个时刻（例如今天22:00），而Duration表示的是时长（例如一周）。但他们具有相同的结构（秒和纳秒）：

```
int32 secs
int32 nsecs
```

### 创建Time和Duration：

rospy中的Time和Duration的构造函数类似，都是 `_init_(self, secs=0, nsecs=0)`，指定秒和纳秒( $1\text{ns} = 10^{-9}\text{s}$ )

```
time_now1 = rospy.get_rostime() #当前时刻的Time对象 返回Time对象
time_now2 = rospy.Time.now() #同上
time_now3 = rospy.get_time() #得到当前时间，返回float 4单位秒
time_4 = rospy.Time(5) #创建5s的时刻
duration = rospy.Duration(3*60) #创建3min时长
```

关于Time、Duration之间的加减法和类型转换，和rosCPP中的完全一致，请参考5.6节，此处不再重复。

### sleep

```
duration.sleep() #挂起
rospy.sleep(duration) #同上，这两种方式效果完全一致

loop_rate = Rate(5) #利用Rate来控制循环频率
while(rospy.is_shutdown()):
    loop_rate.sleep() #挂起，会考虑上次loop_rate.sleep的时间
```

关于sleep的方法，Rate类中的sleep主要用来保持一个循环按照固定的频率，循环中一般都是发布消息、执行周期性任务的操作。这里的sleep会考虑上次sleep的时间，从而使整个循环严格按照指定的频率。

### 定时器Timer

rospy里的定时器和rosCPP中的也类似，只不过不是用句柄来创建，而是直接 `rospy.Timer(Duration, callback)`，第一个参数是时长，第二个参数是回调函数。

```
def my_callback(event):
    print 'Timer called at ' + str(event.current_real)

rospy.Timer(rospy.Duration(2), my_callback) #每2s触发一次callback函数
rospy.spin()
```

同样不要忘了 `rospy.spin()`，只有 `spin` 才能触发回调函数。回调函数的传入值是 `TimerEvent` 类型，该类型包括以下几个属性：

```
rospy.TimerEvent
last_expected
理想情况下为上一次回调应该发生的时间
last_real
上次回调实际发生的时间
current_expected
本次回调应该发生的时间
current_real
本次回调实际发生的时间
last_duration
上次回调所用的时间（结束-开始）
```

# 第八章 TF与URDF

## 本章简介

机器人的坐标变换一直以来是机器人学的一个难点，我们人类在进行一个简单的动作时，从思考到实施行动再到完成动作可能仅仅需要几秒钟，但是机器人来讲就需要大量的计算和坐标转换。其中的坐标转换TF和URDF是本章要详细介绍的内容。

首先我们从认识TF开始，然后学习TF消息和TF树，在后面我们还介绍了TF的数据类型和在C++以及Python中的一些函数和类。也简单介绍了统一机器人描述格式URDF。学习了TF和URDF，我们才开始真正的深入认识ROS。

## 8.1 认识TF

### 8.1.1 简介

TF是一个ROS世界里的一个基本的也是很重要的概念，所谓TF(TransForm)，就是坐标转换。在现实生活中，我们做出各种行为模式都可以在很短的时间里完成，比如拿起身边的物品，但是在机器人的世界里，则远远没有那么简单。观察下图，我们来分析机器人拿起身边的物品需要做到什么，而TF又起到什么样的作用。



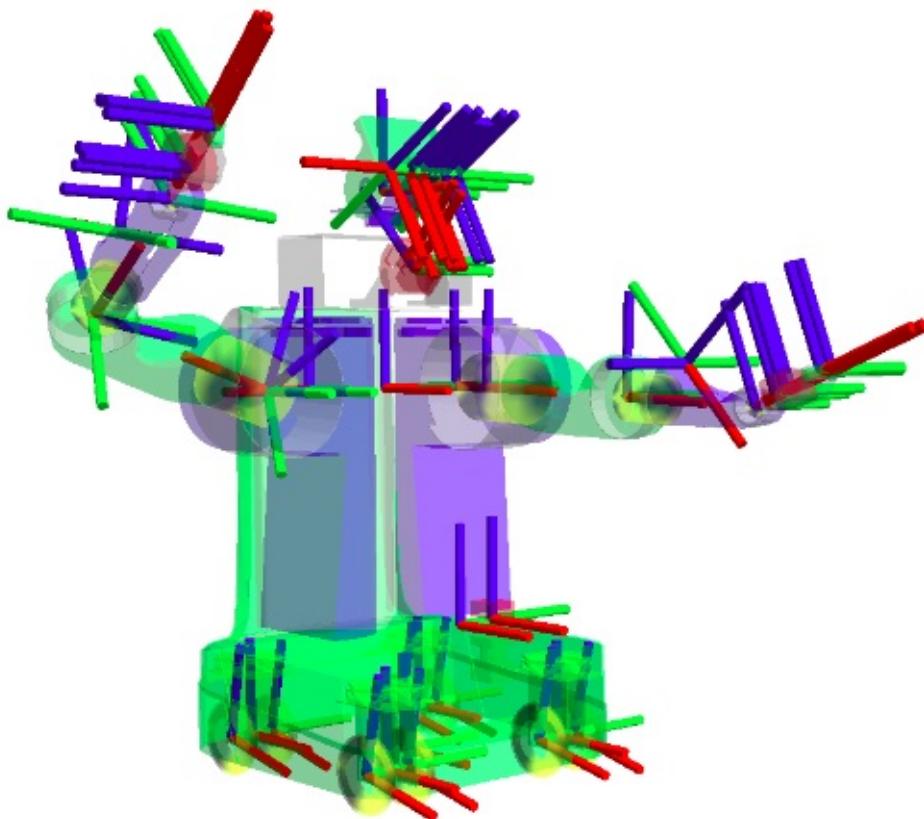
观察这个机器人，我们直观上不认为拿起物品会有什么难度，站在人类的立场上，我们也许会想到手向前伸，抓住，手收回。就完成了这整个一系列的动作。但是如今的机器人远远没有这么智能，它能得到的只是各种传感器发送回来的数据，然后它再处理各种数据进行操作，比如手臂弯曲45度，再向前移动20cm等这样的各种十分精确的数据，尽管如此，机器

人依然没法做到像人类一样自如的进行各种行为操作，那么在这个过程中，TF又扮演着什么样的角色呢？还拿该图来说，当机器人的“眼睛”获取一组数据，关于物体的坐标方位，但是相对于机器人手臂来说，这个坐标只是相对于机器人头部的传感器，并不直接适用于机器人手臂执行，那么物体相对于头部和手臂之间的坐标转换，就是TF。

坐标变换包括了位置和姿态两个方面的变换，ROS中的tf是一个可以让用户随时记录多个坐标系的软件包。tf保持缓存的树形结构中的坐标系之间的关系，并且允许用户在任何期望的时间点在任何两个坐标系之间转换点，矢量等。

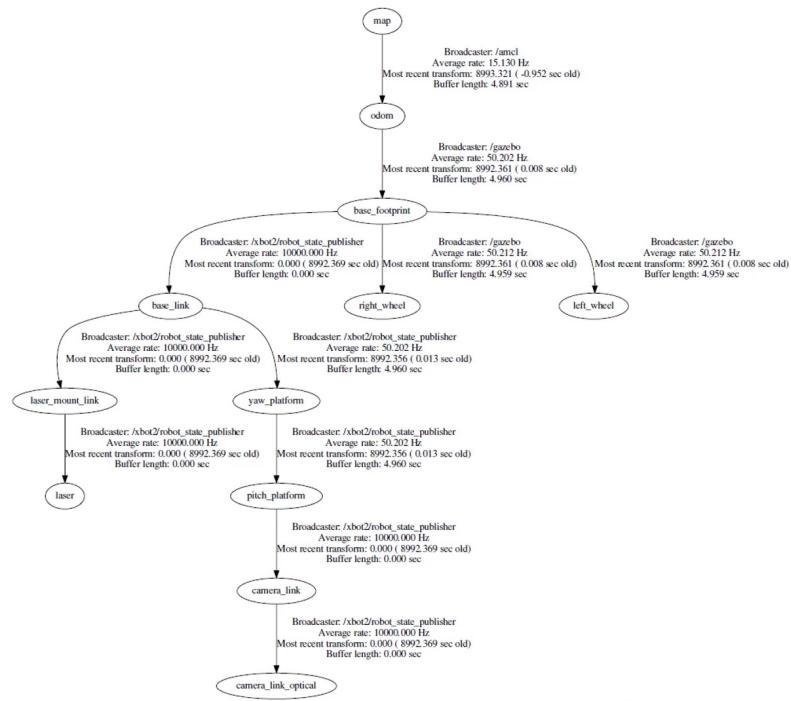
### 8.1.2 ROS中的TF

tf的定义不是那么的死板，它可以被当做是一种标准规范，这套标准定义了坐标转换的数据格式和数据结构。tf本质是树状的数据结构，所以我们通常称之为“**tf tree**”，tf也可以看成是一个topic: /tf，话题中的message保存的就是tf tree的数据结构格式。维护了整个机器人的甚至是地图的坐标转换关系。tf还可以看成是一个package，它当中包含了很多的工具。比如可视化，查看关节间的tf, debug tf等等。tf含有一部分的接口，就是我们前面章节介绍的rosccpp和rospy里关于tf的API。所以可以看成是话题转换的标准，话题，工具，接口。



观察上图，我们可以看到ROS数据结构的一个抽象图，ROS中机器人模型包含大量的部件，这些部件统称之为**link**，每一个link上面对应着一个**frame**，即一个坐标系。link和frame概念是绑定在一起的。像上图pr2模型中我们可以看到又很多的frame，错综复杂的铺置在机器人的各个link上，维护各个坐标系之间的关系，就要靠着tf tree来处理，维护着各个坐标系之间的联通。如下图：

## TF树

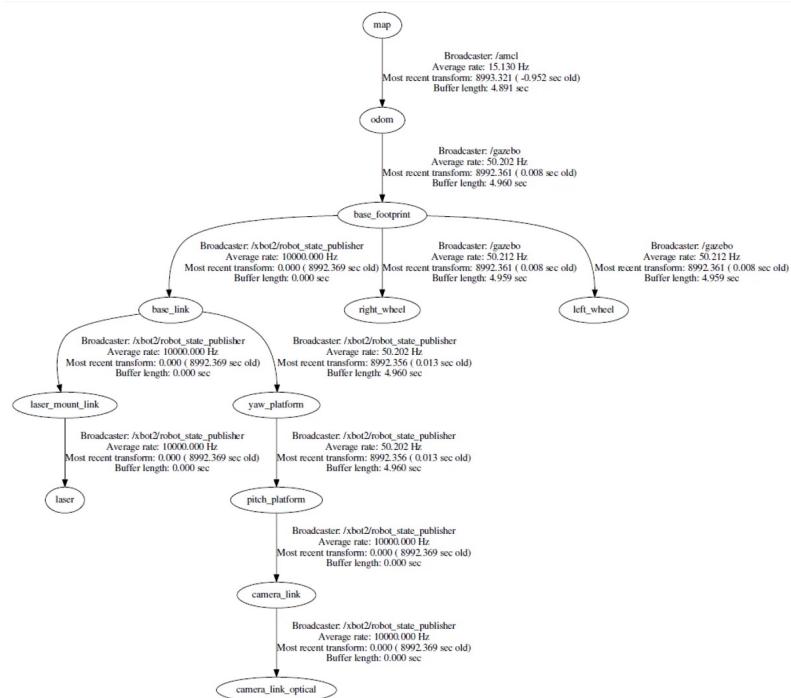


上图是我们常用的robot\_sim\_demo运行起来的tf tree结构，每一个圆圈代表一个frame,对应着机器人上的一个link，任意的两个frame之间都必须是联通的，如果出现某一环节的断裂，就会引发error系统报错。所以完整的tf tree不能有任何断层的地方，这样我们才能查清楚任意两个frame之间的关系。仔细观察上图，我们发现每两个frame之间都有一个broadcaster,这就是为了使得两个frame之间能够正确连通，中间都会有一个Node来发布消息来broadcaster.如果缺少Node来发布消息维护连通，那么这两个frame之间的连接就会断掉。broadcaster就是一个publisher,如果两个frame之间发生了相对运动，broadcaster就会发布相关消息。

## 8.2 TF消息

### 8.2.1 TransformStamped.msg

TF树



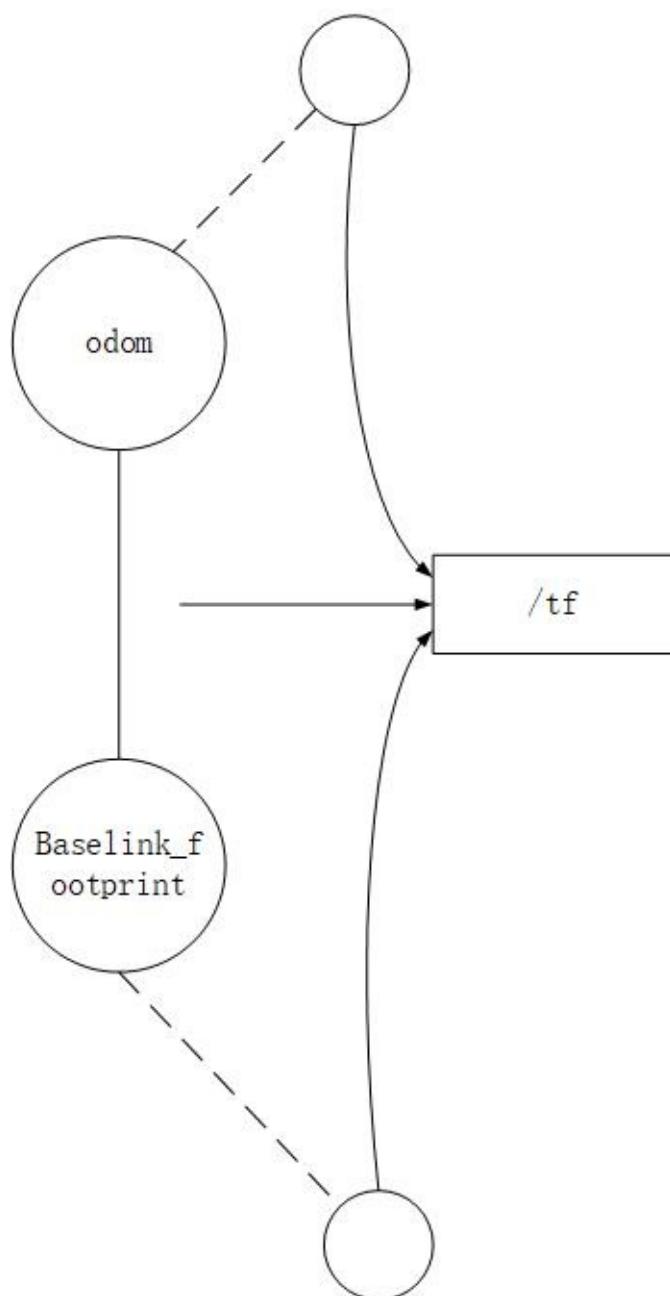
上一节在介绍ROS中的TF时候我们已经初步的的认识了TF和TF树，了解了在每个frame之间都会有**broadcaster**来发布消息维系坐标转换·那么这个消息到底是什么样子的呢？这个消息**TransformStampde.msg**,它就是处理两个frame之间一小段tf的数据格式·

### 8.2.2 格式规范

**TransformStamped.msg**的格式规范如下：

```
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
```

观察标准的格式规范，首先header定义了序号，时间以及frame的名称·接着还写了child\_frame，这两个frame之间要做那种变换就是由geometry\_msgs/Transform来定义·Vector3三维向量表示平移，Quaternion四元数表示旋转·像下图TF树中的两个frame之间的消息，就是由这种格式来定义的·odom就是frame\_id,baselink\_footprint就是child\_frame\_id.我们知道，一个topic上面，可能会有很多个node向上面发送消息。如图所示，不仅有我们看到的frame发送坐标变换个tf，还有别的frame也在同样的向它发送消息。最终，许多的TransformStamped.msg发向tf，形成了TF树。



### 8.2.3 TF树的数据类型

上面我们讲了，TF tree是由很多的frame之间TF拼接而成。那么TF tree是什么类型呢？如下：

- tf/tfMessage.msg
- tf2\_msgs/TFMessage.msg

这里TF的数据类型有两个，主要的原因是版本的迭代。自ROS Hydro以来，tf第一代已被“弃用”，转而支持tf2。tf2相比tf更加简单高效。此外也添加了一些新的功能。

由于tf2是一个重大的变化，tf API一直保持现有的形式。由于tf2具有tf特性的超集和一部分依赖关系，所以tf实现已经被移除，并被引用到tf2下。这意味着所有用户都将与tf2兼容。官网建议新工作直接使用tf2，因为它有一个更清洁的界面，和更好的使用体验。

如何查看自己使用的TF是哪一个版本，使用命令 `rostopic info /tf` 即可。

## 8.2.4 格式定义

tf/tfMessage.msg或tf2\_msgs/TFMessage标准格式规范如下：

```
geometry_msgs/TransformStamped[] transforms
    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
```

如上，一个TransformStamped数组就是一个TF tree。

## 8.3 tf in c++

### 8.3.1 简介

前面内容我们介绍了TF的基本的概念和TF树消息的格式类型，我们知道，TF不仅仅是一个标准、话题，它还是一个接口。本节课我们就介绍c++中TF的一些函数和写法。

### 8.3.2 数据类型

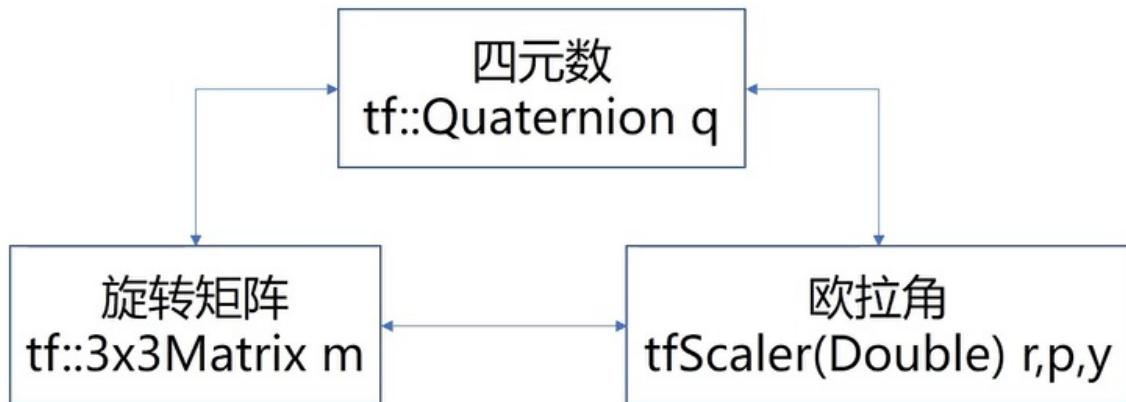
C++中给我们提供了很多TF的数据类型，如下表：

名称	数据类型
向量	tf::Vector3
点	tf::Point
四元数	tf::Quaternion
3*3矩阵（旋转矩阵）	tf::Matrix3x3
位姿	tf::pose
变换	tf::Transform
带时间戳的以上类型	tf::Stamped
带时间戳的变换	tf::StampedTransform

易混注意：虽然此表的最后带时间戳的变换数据类型为tf::StampedTransform,和上节我们所讲的geometry\_msgs/TransformStamped.msg看起来很相似，但是其实数据类型完全不一样，tf::StampedTransform只能用在C++里，只是C++的一个类，一种数据格式，并不是一个消息。而geometry\_msgs/TransformStamped.msg是一个message,它依赖于ROS，与语言无关，也即是无论何种语言，C++、Python、Java等等，都可以发送该消息。

### 8.3.3 数据转换

在TF里有可能会遇到各种各样数据的转换，例如常见的四元数、旋转矩阵、欧拉角这三种数据之间的转换。tf in roscpp给了我们解决该问题的函数。详细源码在我们教学课程的代码包中，在此不再详述。



### 8.3.4 TF类

#### **tf::TransformBroadcaster**类

```

transformBroadcaster()
void sendTransform(const StampedTransform &transform)
void sendTransform(const std::vector<StampedTransform> &transforms)
void sendTransform(const geometry_msgs::TransformStamped &transform)
void sendTransform(const std::vector<geometry_msgs::TransformStamped> &transforms)
  
```

这个类在前面讲TF树的时候提到过，这个**broadcaster**就是一个**publisher**,而**sendTransform**的作用是来封装**publish**的函数。在实际的使用中，我们需要在某个**Node**中构建**tf::TransformBroadcaster**类，然后调用**sendTransform()**,将**transform**发布到 `/tf` 的一段**transform**上。`/tf` 里的**transform**为我们重载了多种不同的函数类型，具体如上。

#### **tf::TransformListener**类

```

void lookupTranform(const std::string &target_frame,const std::string &source_frame,co
nstant ros::Time &time,StampedTransform &transform)const
bool canTransform()
bool waitForTransform()const
  
```

上一个类是向 `/tf` 上发的类，那么这一个就是从 `/tf` 上接收的类。首先看**lookuptransform()**函数，第一个参数是目标坐标系，第二个参数为源坐标系，也即是得到从源坐标系到目标坐标系之间的转换关系，第三个参数为查询时刻，第四个参数为存储转换关系的位置。值得注意，第三个参数通常用 `ros::Time(0)` ,这个表示为最新的坐标转换关系，

而 `ros::time::now` 则会因为收发延迟的原因，而不能正确获取当前最新的坐标转换关系。`canTransform()`是用来判断两个`transform`之间是否连通，`waitForTransform(const`是用来等待某两个`transform`之间的连通。

## 8.4 tf in python

### 8.4.1 简介

我们知道tf中不仅有C++的接口，也有Python的接口。相比C++，tf在Python中的具体实现相对简单好用。

### 8.4.2 数据类型

TF的相关数据类型，向量、点、四元数、矩阵都可以表示成类似数组形式，就是它们都可以用Tuple，List，Numpy Array来表示。例如：

```
t = (1.0, 1.5, 0) #平移
q = [1, 0, 0, 0] #四元数
m = numpy.identity(3) #旋转矩阵
```

第一个平移数据使用Tuple表示的，同时也可以用List表示成`t=[1.0, 1.5, 0]`,也能用`numpy.array([1.0, 1.5, 0])`来表示都是可以的。这些数据类型没有特殊对应，全部是通用的，所以这里也就没有了各种数据类型的转换的麻烦。

### 8.4.3 TF库

#### `tf.transformations`

基本数学运算函数

函数	注释
<code>euler_matrix(ai,aj,ak,axes='sxyz')</code>	欧拉角到矩阵
<code>eulaer_form_matrix(matrix,axes='sxyz')</code>	矩阵到欧拉角
<code>eular_from_quaternion(quaternion,axes='sxyz')</code>	四元数到欧拉角
<code>quaternion_form_euler(ai,aj,ak,axes='sxyz')</code>	欧拉角到四元数
<code>quaternion_matrix(quaternion)</code>	四元数到矩阵
<code>quaternion_form_matrix(matrix)</code>	矩阵到四元数
.....	.....

使用该函数库时候，首先`import tf`，`tf.transformations`给我们提供了一些基本的数学运算函数如上，使用起来非常方便。

## 8.4.4 TF类

### tf.TransformListener类

方法	作用
canTransform(self,target_frame,source_frame,time)	frame是否相通
waitForTransform(self,target_frame,source_frame,time,timeout)	阻塞直到frame相通
lookup Transform(self,target_frame,source_frame,time)	查看相对的tf，返回(trans,quat)

tf.TransformListener类中主要包含以上三种方法，它的构造函数不需要填值。注意这里的time参数，依然是使用 `rospy.Time(0)` 而不是 `rospy.Time.now()`。具体原因上节已经介绍，这里不再赘述。除了上述三种重要的方法，这个类中还有一些辅助用的方法如下：

方法	作用
chain(target_frame,target_time,source_frame,source_time,fixed_frame)	frame的连接
frameExists(self,frame_id)	frame是否存在
getFrameStrings(self)	返回所有tf的
fromTranslationRotation(translation,rotation)	根据平移和旋转 4X4矩阵
transformPoint(target_frame,point_msg)	将PointStamp转换到新frame
transformPose(target_frame,pose_msg)	将PoseStamp转换到新frame
transformQuaternion(target_frame,quat_msg)	将QuaternionStamp 返回相同类
...	...

### tf.TransformBroadcaster类

类似的，我们介绍的是发布方，tf.TransformBroadcaster类。该类的构造函数也是不需要填值，成员函数有两个如下：

- `sendTransform(translation,rotation,time,child,parent)`#向/tf发布消息
- `sendTransformMessage(transform)`#向/tf发布消息

第一个`sendTransform()`把transform的平移和旋转填好，打上时间戳，然后表示出从父到子的frame流，然后发向 /tf 的topic。第二种是发送transform已经封装好的Message给 /tf，这两种不同的发送方式，功能是一致的。

## 8.4.5 TF相关工具命令

1. 根据当前的tf树创建一个pdf图：

```
$ rosrun tf view_frames
```

这个工具首先订阅 /tf ，订阅5秒钟，根据这段时间接受到的tf信息，绘制成一张tf tree，然后创建成一个pdf图。

2. 查看当前的tf树：

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

该命令同样是查询tf tree的，但是与第一个命令的区别是该命令是动态的查询当前的tf tree,当前的任何变化都能当即看到，例如何时断开何时连接，捕捉到这些然后通过rqt插件显示出来。

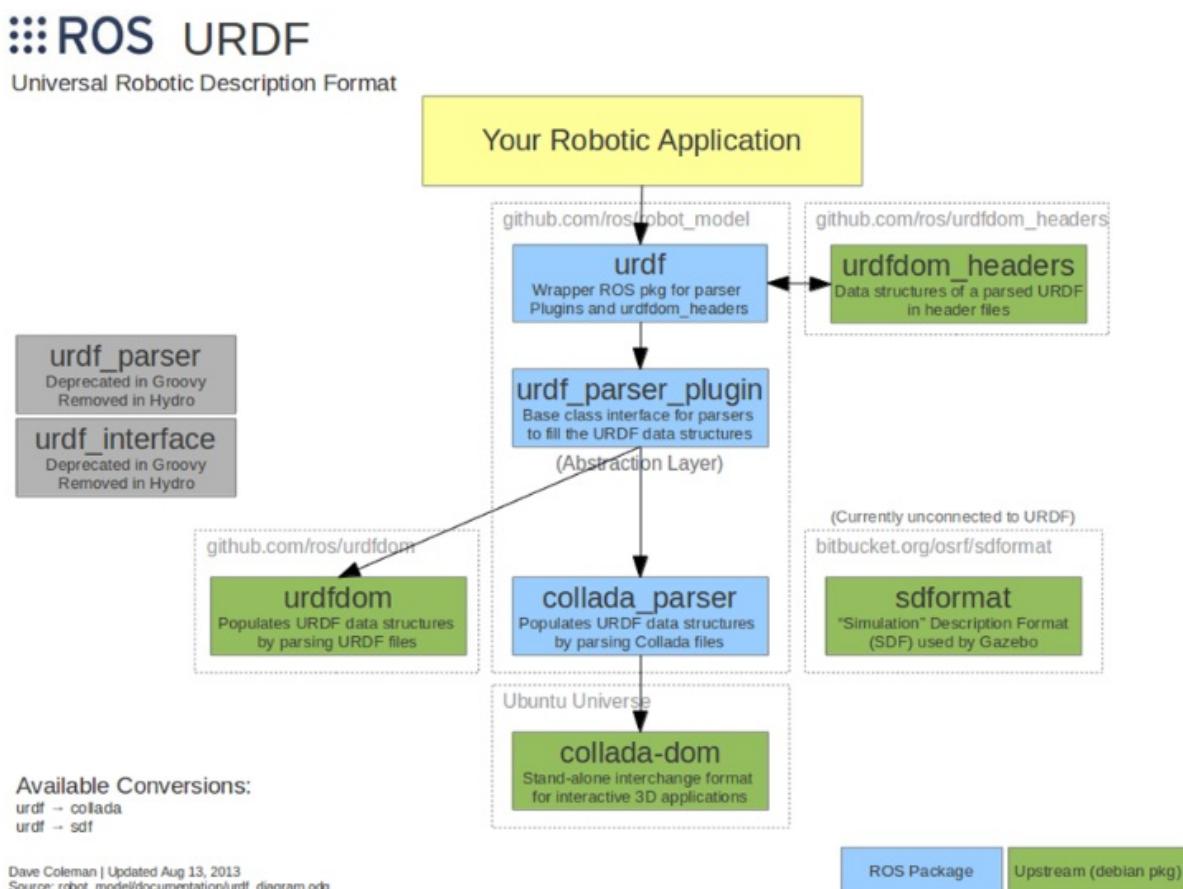
3. 查看两个frame之间的变换关系：

```
$ rosrun tf tf_echo[reference_frame][target_frame]
```

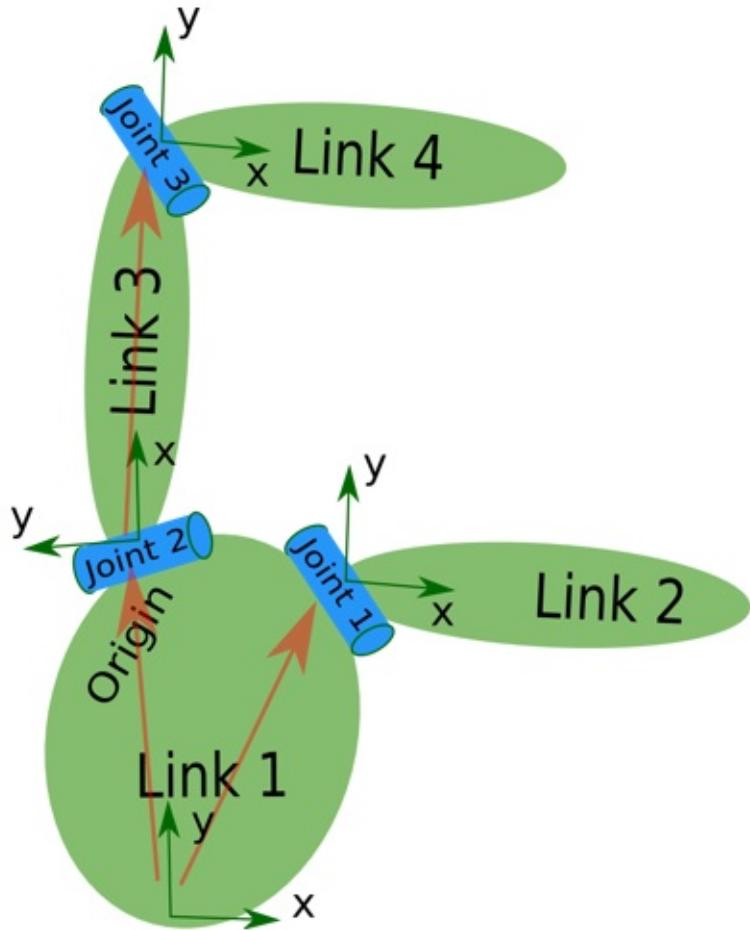
# 8.5 统一机器人描述格式URDF

## 8.5.1 URDF基础

URDF（Unified Robot Description Format）统一机器人描述格式，URDF使用XML格式描述机器人文件。URDF语法规规范，参考链接：<http://wiki.ros.org/urdf/XML>，URDF组件，是由不同的功能包和组件组成：



其中`urdf_parser`和`urder_interface`已经在hydro之后的版本中去除了。  
`urdf_paser_plugin`是URDF基础的插件，衍生出了`urdfdom`（面向URDF文件）和`collar_parser`（面向相互文件）。在URDF当中，当你想要描述一个机器人的时候，例如小车的`base_link`和右轮`right`，两个link之间需要`joint`来连接。参考下图：



## 8.5.2 制作URDF模型

\$\$\quad (1) \text{ 添加基本模型}

我们以构建一个小车为例子，为大家讲解这部分的内容：（相关的示例代码可以从我们的tf\_demo中找到），我们的想法是，首先构建base\_link作为小车的父坐标系，然后在base\_link基础上，再构建左轮，右轮 和雷达 link. 最后不同的link之间通过joint来连接。参考代码如下：

小技巧：`sudo apt-get install liburdfdom-tools`，安装完毕后，执行检查`check_urdf my_car.urdf`如果一切正常，就会有如下显示：

```
robot name is: mycar
----- Successfully Parsed XML -----
root Link: base link has 3 child(ren)
    child(1): left
    child(2): right
    child(3): rplidar
```

\$\$\quad (2) \text{ 添加机器人link之间的相对位置关系}

在基础模型之上，我们需要为机器人之间link来设相对位置和朝向的关系，URDF中通过`<origin>`来描述这种关系。

```
<robot name="mycar">
  <link name="base_link" />
  <link name="right" />
  <link name="left" />
  <link name="rplidar" />

  <joint name="right_joint" type="continuous">
    <parent link="base_link"/>
    <child link="right"/>
    <origin rpy="1.57075 0 0" xyz="0 -0.2 0.07"/>
  </joint>

  <joint name="leftf_joint" type="continuous">
    <parent link="base_link"/>
    <child link="left"/>
    <origin rpy="-1.57075 0 0" xyz="0 0.2 0.07"/>
  </joint>

  <joint name="rplidar_joint" type="fixed">
    <parent link="base_link"/>
    <child link="rplidar"/>
    <origin xyz="0.2 0 0.12"/>
  </joint>
</robot>
```

\$\$\backslash quad \text{ (3) } \text{添加模型的尺寸, 形状和颜色等}

在已经设置好模型的link基础上，添加模型的形状（例如圆柱或长方体），相对于link的位置，颜色等。其中形状用`<geometry>`来描述，颜色用`<color>`来描述。

```
<robot name="mycar">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length=".06" radius="0.27"></cylinder>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0.1"/>
      <material name="white">
        <color rgba="1 1 1 1"/>
      </material>
    </visual>
  </link>

  <link name="right">
    <visual>
      <geometry>
        <cylinder length="0.04" radius="0.07"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>

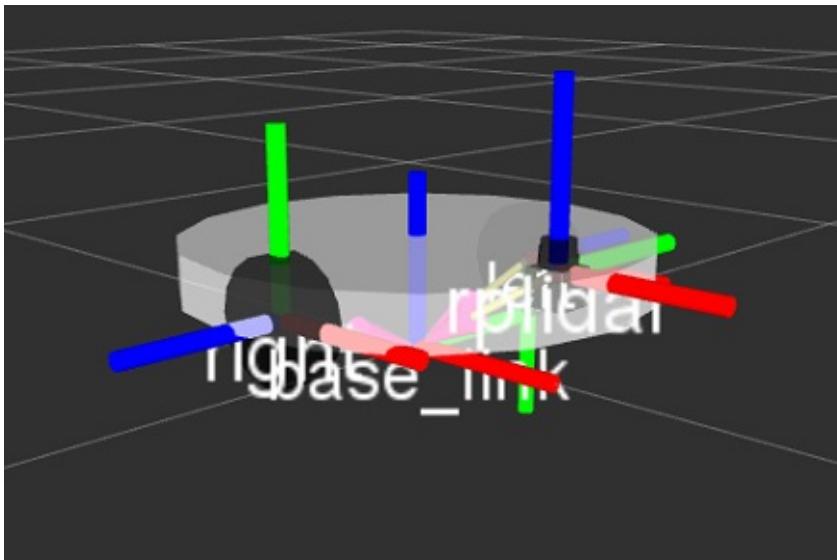
  <link name="left">
    <visual>
      <geometry>
        <cylinder length="0.04" radius="0.07"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="black"/>
    </visual>
  </link>
```

\$\$\$quad\$\$\$ (4) 显示URDF模型

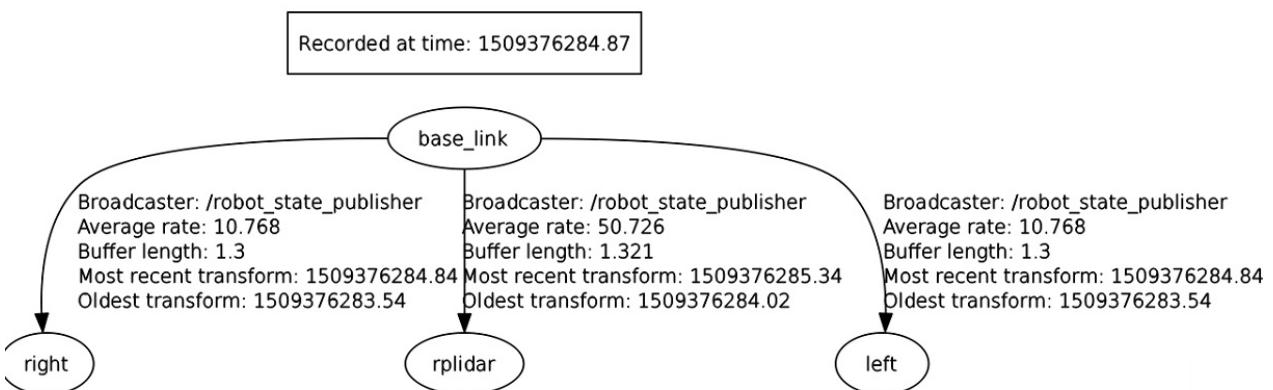
想要在rviz中显示出我们制作好的小车的URDF模型，可以写一个launch文件，参考如下：

```
<launch>
  <arg name="model" default="$(find tf_demo)/urdf/mycar.urdf"/>
  <param name="robot_description" command="$(find xacro)/xacro.py $(arg model)" />
  <param name="use_gui" value="false"/>
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find tf_demo/urdf/mycar.rviz" output="screen"/>
</launch>
```

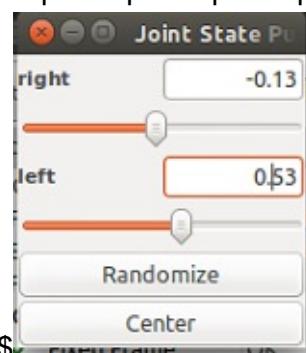
除了launch文件中的前3句，导入我们制作小车的URDF模型外，还需要添加joint state publisher和robot state publisher这两个节点。效果如下：



另外，我们可以输入`rosrun rqt_tf_tree rqt_tf_tree`，可以看到以下`tf`树：



小技巧：你可以将launch文件中的param name="use\_gui"的值由false改成true会弹出一个窗口，同一移动进度条，可以临时改变joint的朝向。

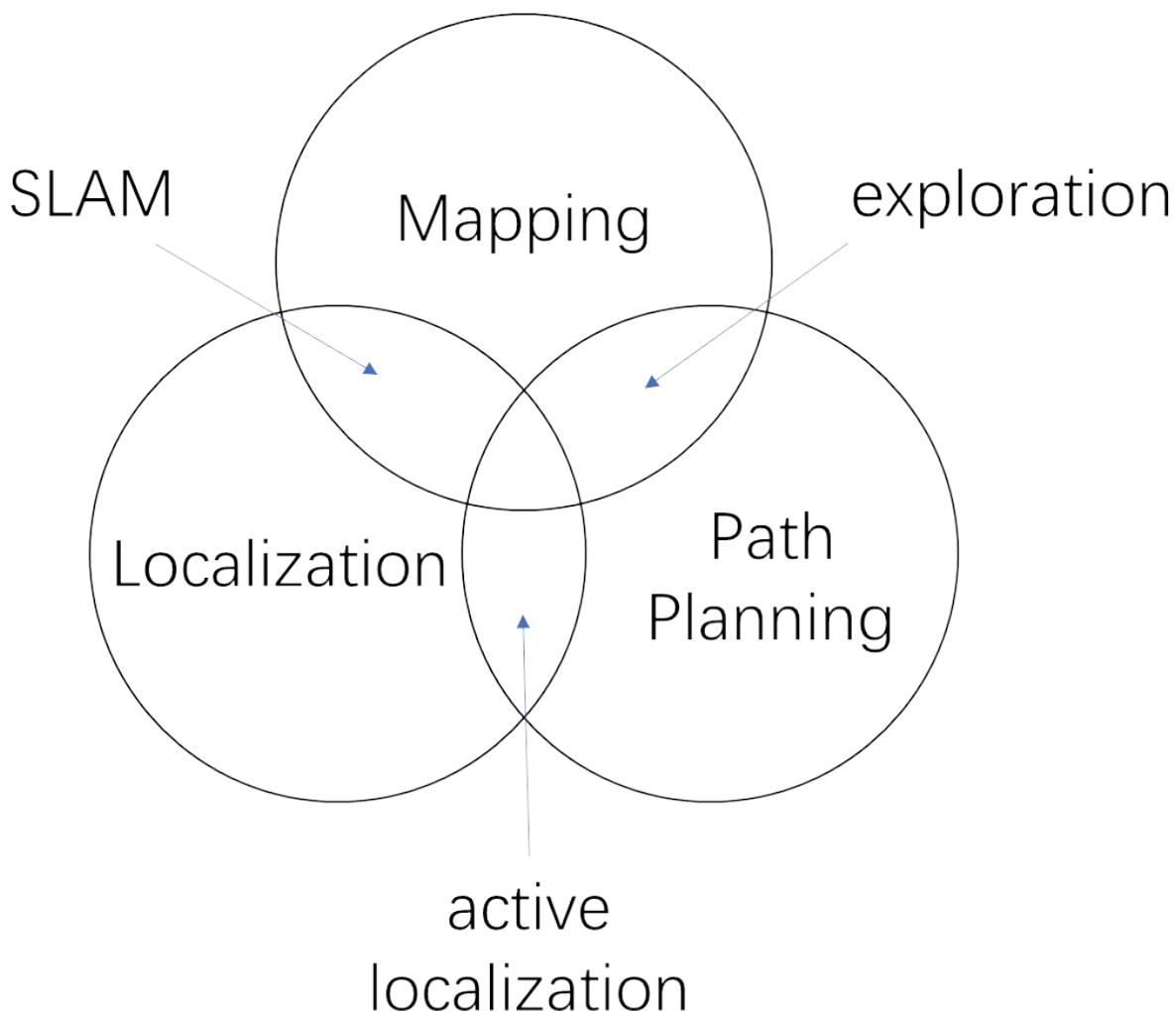


# 第九章 SLAM

## 本章简介

机器人研究的问题包含许许多多的领域，我们常见的几个研究的问题包括：建图(Mapping)、定位(Localization)和路径规划（Path Planning），如果机器人带有机械臂，那么运动规划（Motion Planning）也是重要的一个环节。而同步定位与建图（SLAM）问题位于定位和建图的交集部分。

SLAM需要机器人在未知的环境中逐步建立起地图，然后根据地区确定自身位置，从而进一步定位。



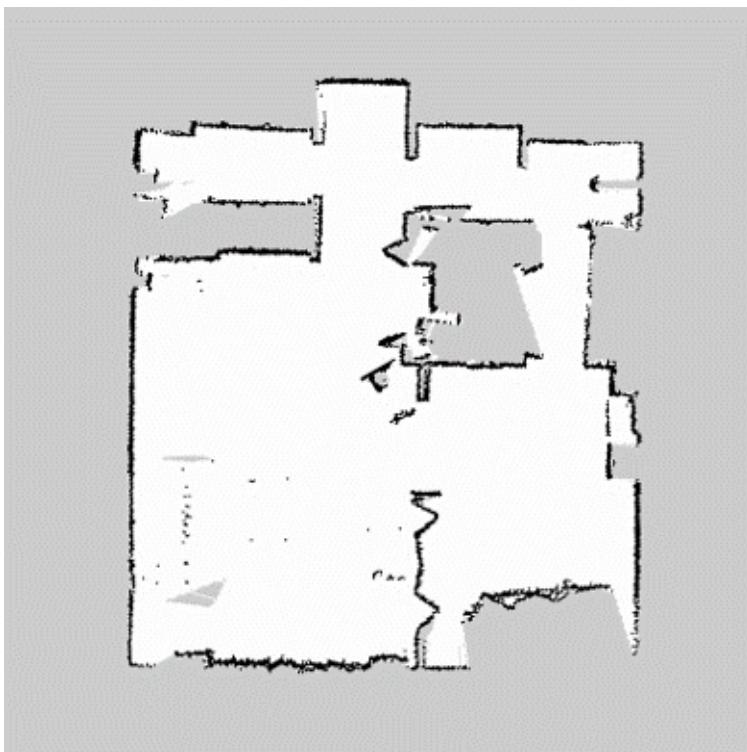
这一章我们来看ROS中SLAM的一些功能包，也就是一些常用的SLAM算法，例如Gmapping、Karto、Hector、Cartographer等算法。这一章我们不会去关注算法背后的数学原理，而是更注重工程实现上的方法，告诉你SLAM算法包是如何工作的，怎样快速的搭建起SLAM算法。



## 9.1 地图

### 9.1.1 直观印象

ROS中的地图很好理解，就是一张普通的灰度图像，通常为pgm格式。这张图像上的黑色像素表示障碍物，白色像素表示可行区域，灰色是未探索的区域。如下图所示，



在SLAM建图的过程中，你可以在RViz里看到一张地图被逐渐建立起来的过程，类似于一块块拼图被拼接成一张完整的地图。这张地图对于我们定位、路径规划都是不可缺少的信息。事实上，地图在ROS中是以Topic的形式维护和呈现的，这个Topic名称就叫做 /map，它的消息类型是 nav\_msgs/OccupancyGrid。

### 9.1.2 锁存

由于 /map 中实际上存储的是一张图片，为了减少不必要的开销，这个Topic往往采用锁存 (latched) 的方式来发布。什么是锁存？其实就是：地图如果没有更新，就维持着上次发布的内容不变，此时如果有新的订阅者订阅消息，这时只会收到一个 /map 的消息，也就是上次发布的消息；只有地图更新了（比如SLAM又建出来新的地图），这时 /map 才会发布新的内容。锁存器的作用就是，将发布者最后一次发布的消息保存下来，然后把它自动发送给后来的订阅者。这种方式非常适合变动较慢、相对固定的数据（例如地图），然后只发布一次，相比于同样的消息不定的发布，锁存的方式既可以减少通信中对带宽的占用，也可以减少消息资源维护的开销。

### 9.1.3 nav\_msgs/OccupancyGrid

然后我们来看一下地图的OccupancyGrid类型是如何定义的，你可以通过 `rosmsg show nav_msgs/OccupancyGrid` 来查看消息，或者直接 `rosed nav_msgs OccupancyGrid.msg` 来查看srv文件。

```
std_msgs/Header header #消息的报头
uint32 seq
time stamp
string frame_id #地图消息绑定在TF的那个frame上，一般为map
nav_msgs/MapMetaData info #地图相关信息
    time map_load_time #加载时间
    float32 resolution #分辨率 单位：m/pixel
    uint32 width #宽 单位：pixel
    uint32 height #高 单位：pixel
    geometry_msgs/Pose origin #原点
        geometry_msgs/Point position
            float64 x
            float64 y
            float64 z
        geometry_msgs/Quaternion orientation
            float64 x
            float64 y
            float64 z
            float64 w
int8[] data #地图具体信息
```

这个srv文件定义了/map话题的数据结构，包含了三个主要的部分:header, info和data。

header是消息的报头，保存了序号、时间戳、frame等通用信息，info是地图的配置信息，它反映了地图的属性，data是真正存储这张地图数据的部分，它是一个可变长数组，`int8`后面加了`[]`，你可以理解为一个类似于vector的容器，它存储的内容有`width*height`个`int8`型的数据，也就是这张地图上每个像素。

## 9.2 Gmapping

### 9.2.1 Gmapping SLAM软件包

Gmapping算法是目前基于激光雷达和里程计方案里面比较可靠和成熟的一个算法，它基于粒子滤波，采用RBPF的方法效果稳定，许多基于ROS的机器人都跑的是gmapping\_slam。这个软件包位于ros-perception组织中的[slam\\_gmapping](#)仓库中。其中的 `slam_gmapping` 是一个 metapackage，它依赖了 `gmapping`，而算法具体实现都在 `gmapping` 软件包中，该软件包中的 `slam_gmapping` 程序就是我们在ROS中运行的SLAM节点。如果你感兴趣，可以阅读一下 `gmapping` 的源代码。

如果你的ROS安装的是desktop-full版本，应该默认会带gmapping。你可以用以下命令来检测gmapping是否安装

```
apt-cache search ros-$ROS_DISTRO-gmapping
```

如果提示没有，可以直接用apt安装

```
sudo apt-get install ros-$ROS_DISTRO-gmapping
```

gmapping在ROS上运行的方法很简单

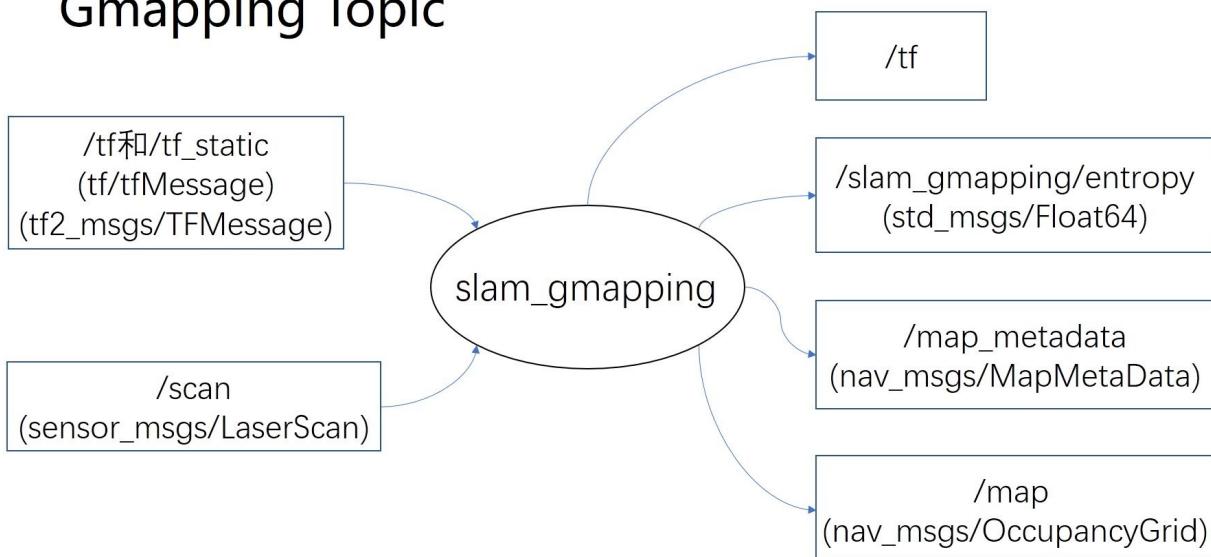
```
roslaunch gmapping slam_gmapping
```

但由于gmapping算法中需要设置的参数很多，这种启动单个节点的效率很低。所以往往我们会把gmapping的启动写到launch文件中，同时把gmapping需要的一些参数也提前设置好，写进launch文件或yaml文件。具体可参考教学软包中的 `slam_sim_demo` 中的 `gmapping_demo.launch` 和 `robot_gmapping.launch.xml` 文件。

### 9.2.2 Gmapping SLAM计算图

gmapping的作用是根据激光雷达和里程计（Odometry）的信息，对环境地图进行构建，并且对自身状态进行估计。因此它得输入应当包括激光雷达和里程计的数据，而输出应当有自身位置和地图。下面我们从计算图（消息的流向）的角度来看看gmapping算法的实际运行中的结构：

# Gmapping Topic



位于中心的是我们运行的 `slam_gmapping` 节点，这个节点负责整个gmapping SLAM的工作。它的输入需要有两个：

## 输入

- `/tf` 以及 `/tf_static`：坐标变换，类型为第一代的 `tf/tfMessage` 或第二代的 `tf2_msgs/TFMessage` 其中一定得提供的有两个`tf`，一个 是 `base_frame` 与 `laser_frame` 之间的`tf`，即机器人底盘和激光雷达之间的变换；一个 是 `base_frame` 与 `odom_frame` 之间的`tf`，即底盘和里程计原点之间的坐标变换。`odom_frame` 可以理解为里程计原点所在的坐标系。
- `/scan`：激光雷达数据，类型为 `sensor_msgs/LaserScan`

`/scan` 很好理解，Gmapping SLAM所必须的激光雷达数据，而 `/tf` 是一个比较容易忽视的细节。尽管 `/tf` 这个Topic听起来很简单，但它维护了整个ROS三维世界里的转换关系，而 `slam_gmapping` 要从中读取的数据是 `base_frame` 与 `laser_frame` 之间的`tf`,只有这样才能够把周围障碍物变换到机器人坐标系下，更重要的是 `base_frame` 与 `odom_frame` 之间的`tf`，这个`tf`反映了里程计（电机的光电码盘、视觉里程计、IMU）的监测数据，也就是机器人里程计测得走了多少距离，它会把这段变换发布到 `odom_frame` 和 `laser_frame` 之间。

因此 `slam_gmapping` 会从 `/tf` 中获得机器人里程计的数据。

## 输出

- `/tf`：主要是输出 `map_frame` 和 `odom_frame` 之间的变换
- `/slam_gmapping/entropy`：`std_msgs/Float64` 类型，反映了机器人位姿估计的分散程度
- `/map`：`slam_gmapping` 建立的地图
- `/map_metadata`：地图的相关信息

输出的 `/tf` 里又一个很重要的信息，就是 `map_frame` 和 `odom_frame` 之间的变换，这其实就是对机器人的定位。通过连通 `map_frame` 和 `odom_frame`，这样 `map_frame` 与 `base_frame` 甚至与 `laser_frame` 都连通了。这样便实现了机器人在地图上的定位。

同时，输出的Topic里还有 `/map`，在上一节我们介绍了地图的类型，在SLAM场景中，地图是作为SLAM的结果被不断地更新和发布。

## 9.2.3 里程计误差及修正

目前ROS中常用的里程计广义上包括车轮上的光电码盘、惯性导航元件（IMU）、视觉里程计，你可以只用其中的一个作为 `odom`，也可以选择多个进行数据融合，融合结果作为 `odom`。通常来说，实际ROS项目中的里程计会发布两个Topic：

- `/odom`：类型为 `nav_msgs/Odometry`，反映里程计估测的机器人位置、方向、线速度、角速度信息。
- `/tf`：主要是输出 `odom_frame` 和 `base_frame` 之间的 `tf`。这段 `tf` 反映了机器人的位置和方向变换，数值与 `/odom` 中的相同。

由于以上三种里程计都是对机器人的位姿进行估计，存在着累计误差，因此当运动时间较长时，`odom_frame` 和 `base_frame` 之间变换的真实值与估计值的误差会越来越大。你可能会想，能否用激光雷达数据来修正 `odom_frame` 和 `base_frame` 的 `tf`。事实上 `gmapping` 不是这么做的，里程计估计的是多少，`odom_frame` 和 `base_frame` 的 `tf` 就显示多少，永远不会去修正这段 `tf`。`gmapping` 的做法是把里程计误差的修正发布到 `map_frame` 和 `odom_frame` 之间的 `tf` 上，也就是把误差补偿在了地图坐标系和里程计原点坐标系之间。通过这种方式来修正定位。

这样 `map_frame` 和 `base_frame`，甚至和 `laser_frame` 之间就连通了，实现了机器人在地图上的定位。

- `/odom`

## 9.2.4 服务

`slam_gmapping` 也提供了一个服务：

- `/dynamic_map`：其 `srv` 类型为 `nav_msgs/GetMap`，用于获取当前的地图。

该 `srv` 定义如下：`nav_msgs/GetMap.srv`

```
# Get the map as a nav_msgs/OccupancyGrid
---
nav_msgs/OccupancyGrid map
```

可见该服务的请求为空，即不需要传入参数，它会直接反馈当前地图。

## 9.2.5 参数

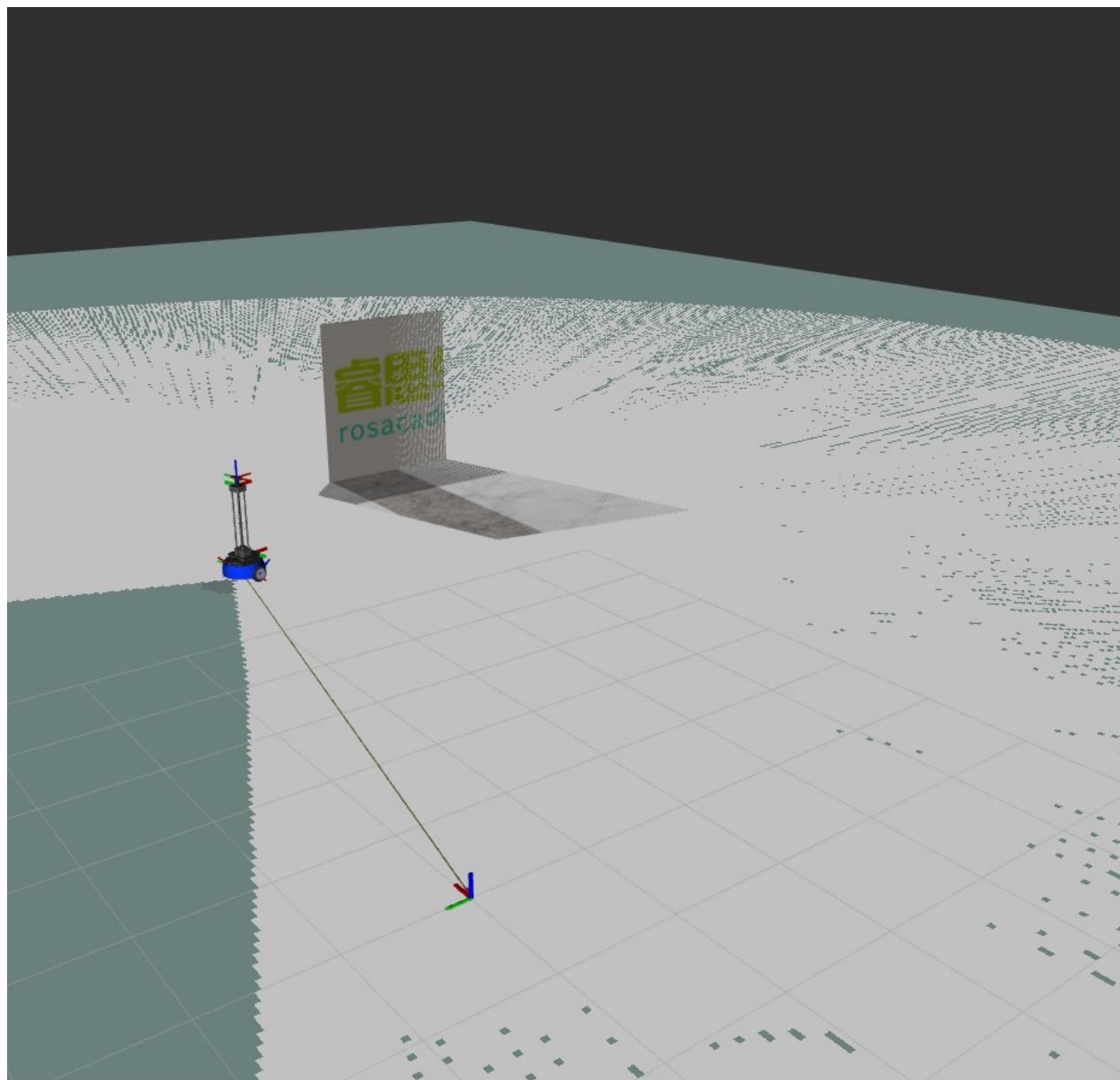
slam\_gmapping 需要的参数很多，这里以 slam\_sim\_demo 教学包中的 gmapping\_demo 的参数为例，注释了一些比较重要的参数，具体请查看 ROS-Academy-for-Beginners/slam\_sim\_demo/launch/include/robot\_gmapping.launch.xml

```

<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">
  <param name="base_frame" value="$(arg base_frame)"/> <!-- 底盘坐标系 -->
  <param name="odom_frame" value="$(arg odom_frame)"/> <!-- 里程计坐标系 -->
  <param name="map_update_interval" value="1.0"/> <!-- 更新时间(s)，每多久更新一次地图，不是频率 -->
  <param name="maxUrange" value="20.0"/> <!-- 激光雷达最大可用距离，在此之外的数据截断不用 -->
  <param name="maxRange" value="25.0"/> <!-- 激光雷达最大距离 -->
  <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="minimumScore" value="200"/>
  <param name="srr" value="0.01"/>
  <param name="srt" value="0.02"/>
  <param name="str" value="0.01"/>
  <param name="stt" value="0.02"/>
  <param name="linearUpdate" value="0.5"/>
  <param name="angularUpdate" value="0.436"/>
  <param name="temporalUpdate" value="-1.0"/>
  <param name="resampleThreshold" value="0.5"/>
  <param name="particles" value="80"/>
  <param name="xmin" value="-25.0"/>
  <param name="ymin" value="-25.0"/>
  <param name="xmax" value="25.0"/>
  <param name="ymax" value="25.0"/>
  <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"/>
  <remap from="scan" to="$(arg scan_topic)"/>
</node>
```

## 演示截图

gmapping 算法演示效果图如下：

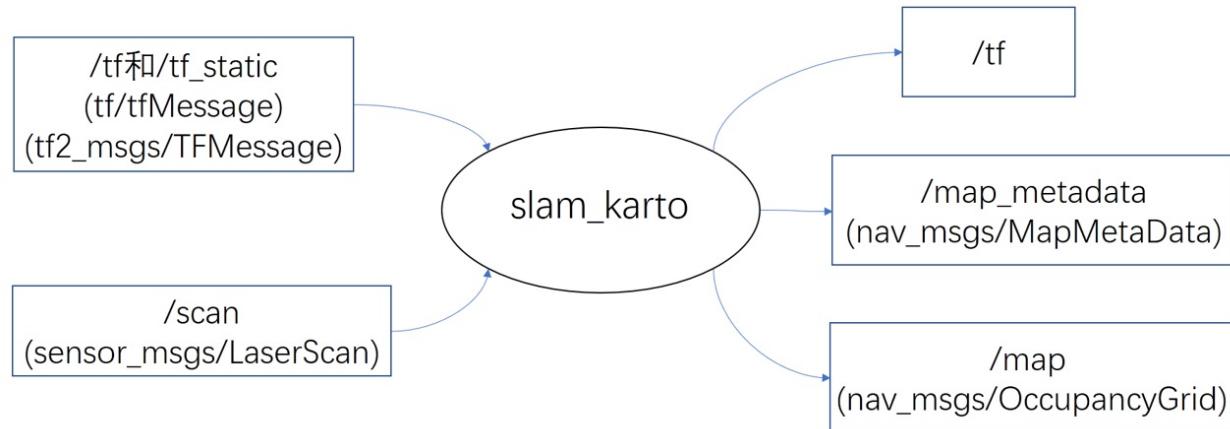


## 9.3 Karto

### 9.3.1 Karto SLAM计算图

Karto SLAM和Gmapping SLAM在工作方式上非常类似，如下图所示

#### Karto Topic



输入的Topic同样是 /tf 和 /scan ，其中 /tf 里要连通 odom\_frame 与 base\_frame ，还有 laser\_frame 。这里和Gmapping完全一样。

唯一不同的地方是输出，slam\_karto的输出少相比slam\_gmapping少了一个位姿估计的分散程度.

### 9.3.2 服务

与Gmapping相同，提供 /dynamic\_map 服务

### 9.3.3 参数

这里以 ROS-Academy-for-Beginners 中的 karto\_slam 为例，选取了它的参数文件 slam\_sim\_demo/param/karto\_params.yaml ，关键位置做了注释：

```

# General Parameters
use_scan_matching: true
use_scan_barycenter: true
minimum_travel_distance: 0.2
minimum_travel_heading: 0.174          #in radians
scan_buffer_size: 70
scan_buffer_maximum_scan_distance: 20.0
link_match_minimum_response_fine: 0.8
link_scan_maximum_distance: 10.0
loop_search_maximum_distance: 4.0
do_loop_closing: true
loop_match_minimum_chain_size: 10
loop_match_maximum_variance_coarse: 0.4      # gets squared later
loop_match_minimum_response_coarse: 0.8
loop_match_minimum_response_fine: 0.8

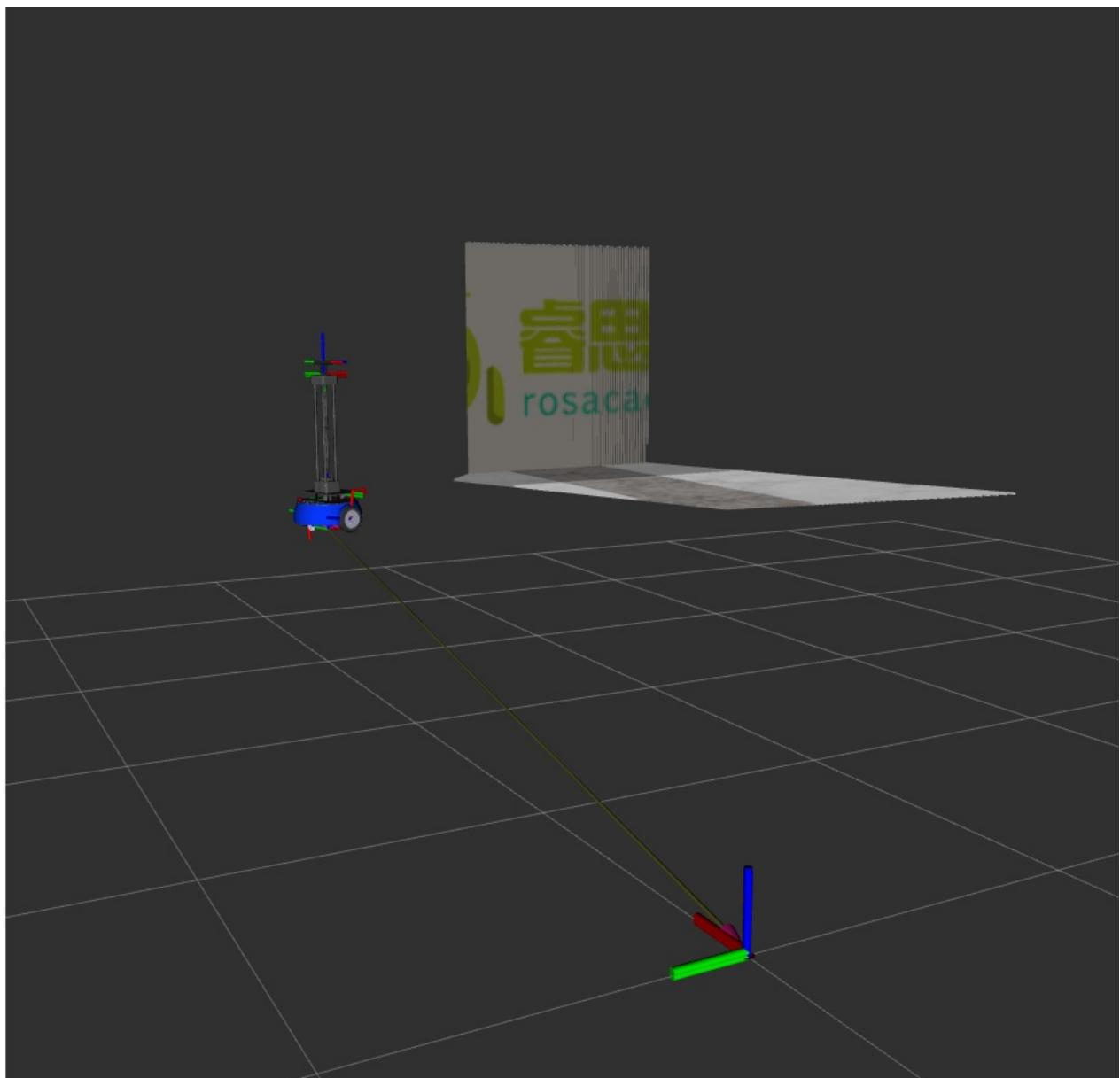
# Correlation Parameters - Correlation Parameters
correlation_search_space_dimension: 0.3
correlation_search_space_resolution: 0.01
correlation_search_space_smear_deviation: 0.03

# Correlation Parameters - Loop Closure Parameters
loop_search_space_dimension: 8.0
loop_search_space_resolution: 0.05
loop_search_space_smear_deviation: 0.03

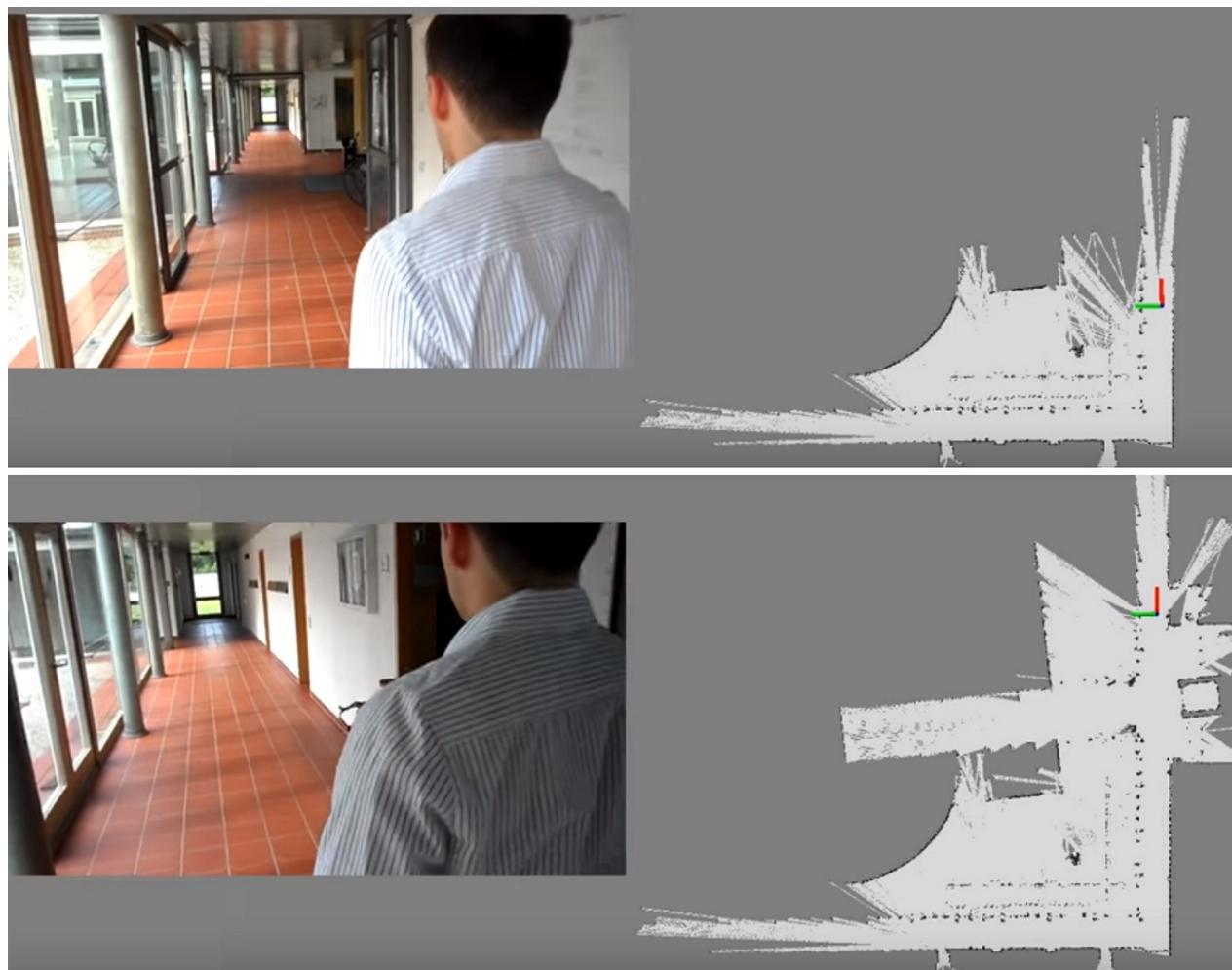
# Scan Matcher Parameters
distance_variance_penalty: 0.3           # gets squared later
angle_variance_penalty: 0.349            # in degrees (gets converted to radians t
hen squared)
fine_search_angle_offset: 0.00349        # in degrees (gets converted to radian
s)
coarse_search_angle_offset: 0.349         # in degrees (gets converted to radians)
coarse_angle_resolution: 0.0349           # in degrees (gets converted to radians
)
minimum_angle_penalty: 0.9
minimum_distance_penalty: 0.5
use_response_expansion: false

```

## 演示截图



## 9.4 Hector



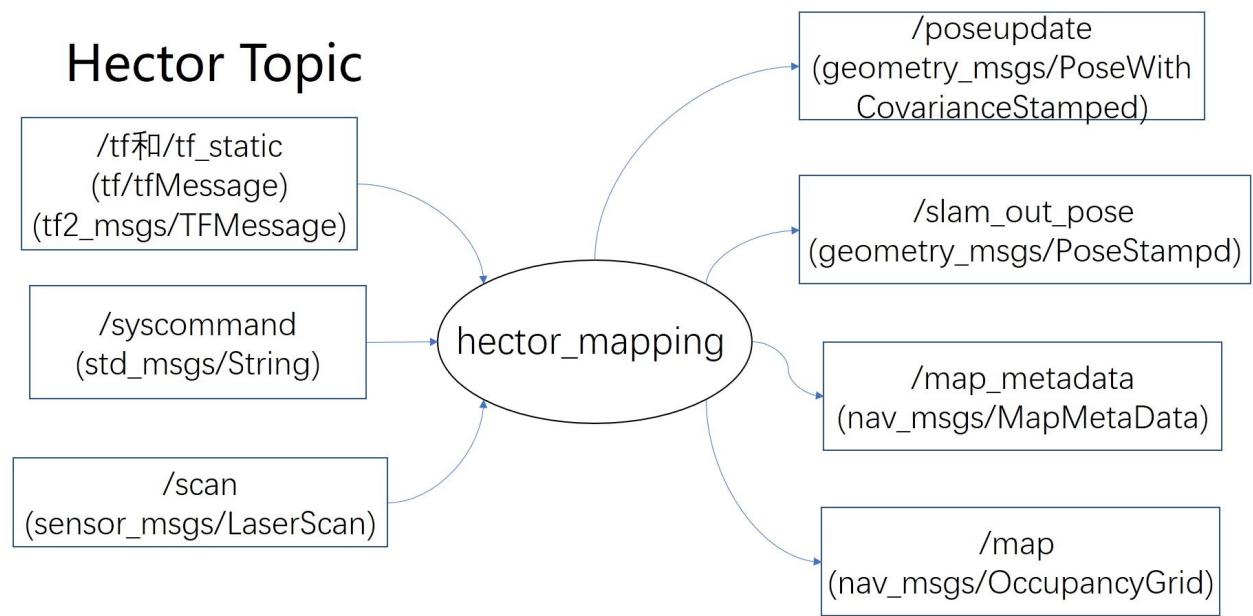
Hector SLAM效果

### 9.4.1 Hector SLAM计算图

Hector SLAM算法不同于前面两种算法，Hector只需要激光雷达数据，而不需要里程计数据。这种算法比较适合手持式的激光雷达，并且对激光雷达的扫描频率有一定要求。

Hector算法的效果不如Gmapping、Karto，因为它仅用到激光雷达信息。这样建图与定位的依据就不如多传感器结合的效果好。但Hector适合手持移动或者本身就没有里程计的机器人使用。

Hector的计算图，如下所示



位于中心的节点叫作 `hector_mapping`，它的输入和其他SLAM框架类似，都包括了 `/tf` 和 `/scan`，另外Hector还订阅一个 `/syscommand` Topic，这是一个字符串型的Topic，当接收到 `reset` 消息时，地图和机器人的位置都会初始化到最初最初的位置。

在输出的Topic方面，hector多了一个 `/poseupdate` 和 `/slam_out_pose`，前者是具有协方差的机器人位姿估计，后者是没有协方差的位姿估计。

### 9.3.2 服务

与Gmapping相同，提供 `/dynamic_map` 查询地图服务

### 9.3.3 参数

以 `ROS-Academy-for-Beginners` 中的 `hector_slam` 为例，选取了它的launch文件 `slam_sim_demo/launch/hector_demo.launch` 为例，关键位置做了注释：

```
<node pkg="hector_mapping" type="hector_mapping" name="hector_height_mapping" output="screen">
    <param name="scan_topic" value="scan" />
    <param name="base_frame" value="base_link" />
    <param name="odom_frame" value="odom" />

    <param name="output_timing" value="false"/>
    <param name="advertise_map_service" value="true"/>
    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="false"/>
    <param name="pub_map_odom_transform" value="true"/>
    <param name="map_with_known_poses" value="false"/>

    <param name="map_pub_period" value="1"/>
    <param name="update_factor_free" value="0.45"/>

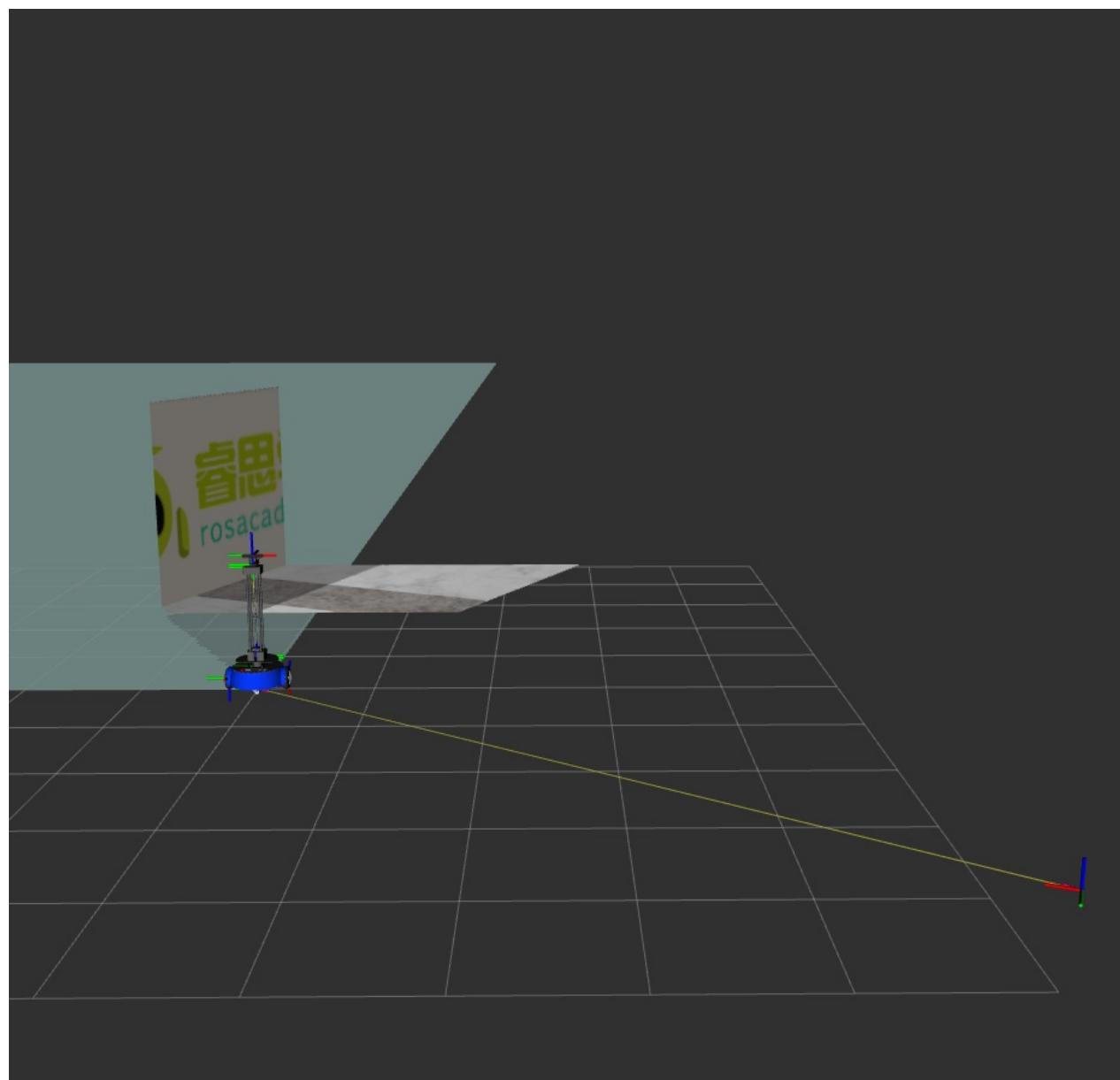
    <param name="map_update_distance_thresh" value="0.1"/>
    <param name="map_update_angle_thresh" value="0.05"/>

    <param name="map_resolution" value="0.05"/>
    <param name="map_size" value="1024"/>
    <param name="map_start_x" value="0"/>
    <param name="map_start_y" value="0"/>

</node>
```

## 演示截图

hector算法演示效果图如下：



# 第十章 Navigation

Navigation是机器人最基本的功能之一，ROS为我们提供了一整套Navigation的解决方案，包括全局与局部的路径规划、代价地图、异常行为恢复、地图服务器等等，这些开源工具包极大地减少了我们开发的工作量，任何一套移动机器人硬件平台经过这套方案就可以快速部署实现。

# 10.1 Navigation Stack

## 10.1.1 Navigation Stack

Navigation Stack是一个ROS的metapackage，里面包含了ROS在路径规划、定位、地图、异常行为恢复等方面package，其中运行的算法都堪称经典。Navigation Stack的主要作用就是路径规划，通常是输入各传感器的数据，输出速度。一般我们的ROS都预装了Navigation。

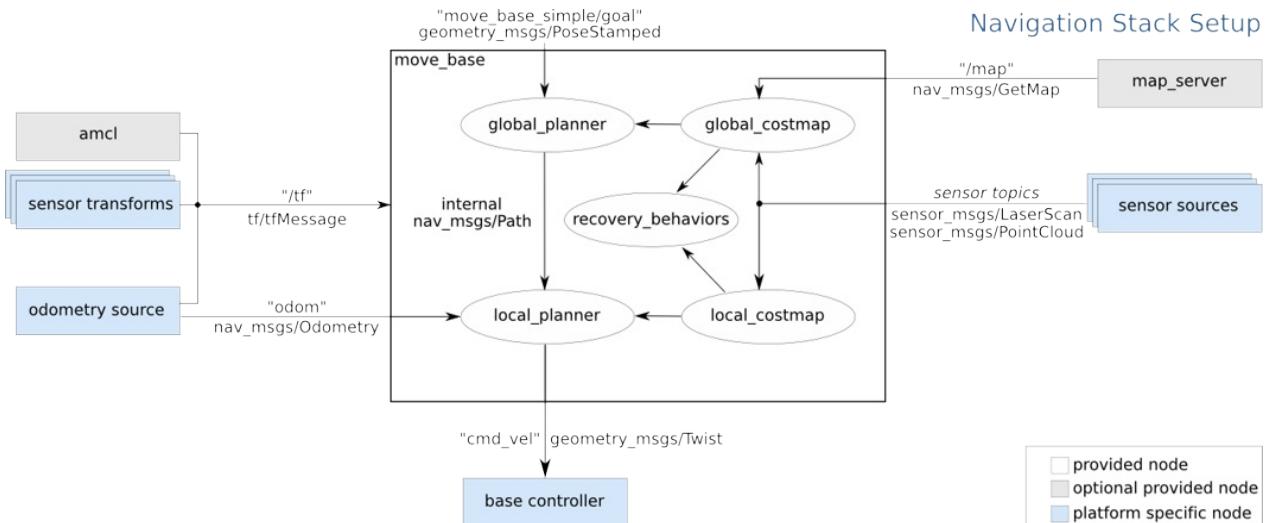
Navigation Stack的源代码位于<https://github.com/ros-planning/navigation>，包括了以下几个package：

包名	功能
amcl	定位
fake_localization	定位
map_server	提供地图
move_base	路径规划节点
nav_core	路径规划的接口类，包括base_local_planner、base_global_planner和recovery_behavior三个接口
base_local_planner	实现了Trajectory Rollout和DWA两种局部规划算法
dwa_local_planner	重新实现了DWA局部规划算法
parrot_planner	实现了较简单的全局规划算法
navfn	实现了Dijkstra和A*全局规划算法
global_planner	重新实现了Dijkstra和A*全局规划算法
clear_costmap_recovery	实现了清除代价地图的恢复行为
rotate_recovery	实现了旋转的恢复行为
move_slow_and_clear	实现了缓慢移动的恢复行为
costmap_2d	二维代价地图
voxel_grid	三维小方块（体素？）
robot_pose_ekf	机器人位姿的卡尔曼滤波

这么多package，你可能会觉得很乱，不过担心，在使用中其实还是比较简单的，我们接下来会对常用的主要功能进行介绍。

## 10.1.2 Navigation工作框架

机器人的自主导航功能基本全靠Navigation中的pacakge，来看这张图：



上图中位于导航功能正中心的是 `move_base` 节点，可以理解为一个强大的路径规划器，在实际的导航任务中，你只需要启动这一个node，并且给他提供数据，就可以规划出路径和速度。`move_base` 之所以能做到路径规划，是因为它包含了很多的插件，像图中的白色圆圈 `global_planner`、`local_planner`、`global_costmap`、`local_costmap`、`recovery_behaviors`。这些插件用于负责一些更细微的任务：全局规划、局部规划、全局地图、局部地图、恢复行为。而每一个插件其实也都是一个package，放在Navigation Stack里。关于`move_base`我们后面会进一步介绍，先来看看 `move_base` 外围有哪些输入输出。

### 输入

- `/tf` : 提要提供的tf包括 `map_frame`、`odom_frame`、`base_frame` 以及机器人各关节之间的一棵tf树。
- `/odom` : 里程计信息
- `/scan` 或 `/pointcloud` : 传感器的输入信息，最常用的是激光雷达 (sensor\_msgs/LaserScan类型)，也有用点云数据(sensor\_msgs/PointCloud)的。
- `/map` : 地图，可以由SLAM程序来提供，也可以由 `map_server` 来指定已知地图。

以上四个Topic是必须持续提供给导航系统的，下面一个是可随时发布的topic：

- `move_base_simple/goal` : 目标点位置。

有几点需要注意：

1.`move_base`并不会去发布tf，因为对于路径规划问题来说，假设地图和位置都是已知的，定位和建图是其他节点的事情。2.`sensor_topics`一般输入是激光雷达数据，但也有输入点云的情况。3.图中`map_server`是灰色，代表可选，并不表示 `/map` 这个topic是可选，必须提供地图给`move_base`。

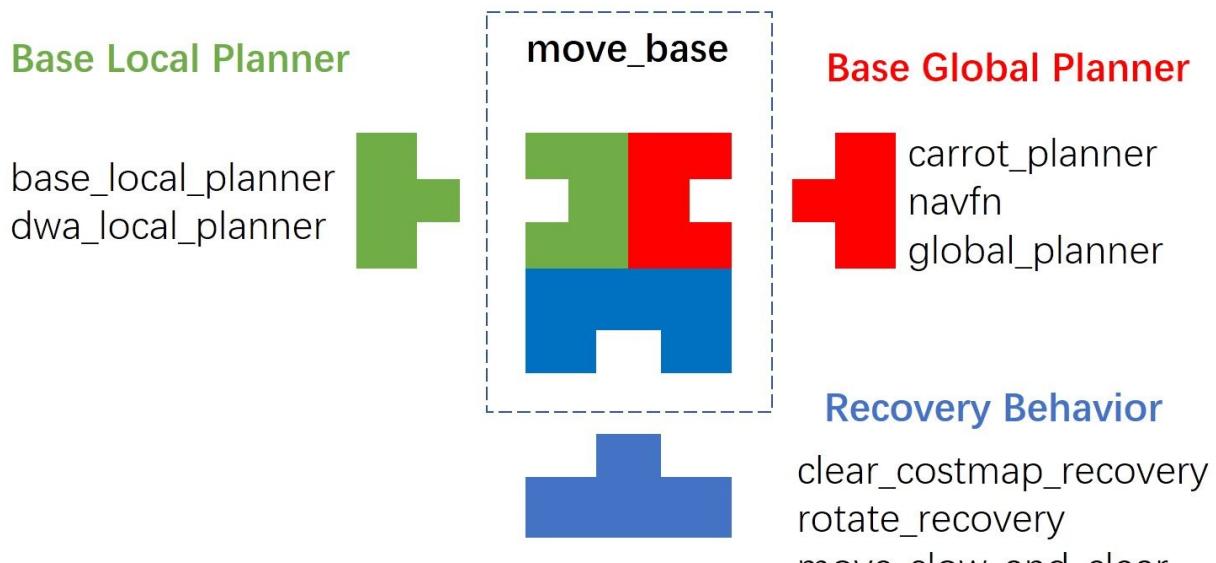
## 输出

- `/cmd_vel` : `geometry_msgs/Twist` 类型，为每一时刻规划的速度信息。

## 10.2 move\_base

### 10.2.1 move\_base与插件

move\_base算得上是Navigation中的核心节点，之所以称之为核心，是因为它在导航的任务中处于支配地位，其他的一些package都是它的插件。来看这张图



move\_base要运行起来，需要选择好插件，包括三种插件：base\_local\_planner、base\_global\_planner 和 recovery\_behavior，这三种插件都得指定，否则系统会指定默认值。

Navigation为我们提供了不少候选的插件，可以在配置move\_base时选择。

#### base\_local\_planner插件：

- base\_local\_planner: 实现了Trajectory Rollout和DWA两种局部规划算法
- dwa\_local\_planner: 实现了DWA局部规划算法，可以看作是base\_local\_planner的改进版本

#### base\_global\_planner插件：

- parrot\_planner: 实现了较简单的全局规划算法
- navfn: 实现了Dijkstra和A\*全局规划算法
- global\_planner: 重新实现了Dijkstra和A\*全局规划算法，可以看作navfn的改进版

## recovery\_behavior插件：

- clear\_costmap\_recovery: 实现了清除代价地图的恢复行为
- rotate\_recovery: 实现了旋转的恢复行为
- move\_slow\_and\_clear: 实现了缓慢移动的恢复行为

除了以上三个需要指定的插件外，还有一个costmap插件，该插件默认已经选择好，无法更改。

以上所有的插件都是继承于 nav\_core 里的接口，nav\_core 属于一个接口 package，它只定义了三种插件的规范，也可以说定义了三种接口类，然后分别由以上的插件来继承和实现这些接口。因此如果你要研究路径规划算法，不妨研究一下 nav\_core 定义的路径规划工作流程，然后仿照 dwa\_local\_planner 或其他插件来实现。

除了以上三个需要指定的插件外，还有一个costmap插件，该插件默认已经选择好， 默认即为**costmap\_2d**，不可更改，但costmap\_2d提供了不同的Layer可以供我们设置，在9.3节我们会进行介绍。

在这里插件的概念并不是我们抽象的描述，而是在ROS里catkin编译系统能够认出的，并且与其他节点能够耦合的**C++**库，插件是可以动态加载的类，也就是说插件不需要提前链接到ROS的程序上，只需在运行时加载插件就可以调用其中的功能。

具体关于插件的介绍，有兴趣请看<http://wiki.ros.org/pluginlib>，本书不做过多介绍。

## 10.2.2 插件选择(参数)

既然我们知道了move\_base具体的一些插件，那如何来选择呢？其实非常简单。在move\_base的参数设置里可以选择插件。move\_base的参数包括以下内容：

参数	默认值	功能
~base_global_planner	navfn/NavfnROS	设置全局规划器
~base_local_planner	base_local_planner/TrajectoryPlannerROS	设置局部规划器
~recovery_behaviors	[{name: conservative_reset, type: clear_costmap_recovery/ClearCostmapRecovery}, {name: rotate_recovery, type: rotate_recovery/RotateRecovery}, {name: aggressive_reset, type: clear_costmap_recovery/ClearCostmapRecovery}]	设置恢复行为
...	...	...

除了这三个选择插件的参数，还有控制频率、误差等等参数。具体请看[http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base)介绍。在ROS-Academy-for-Beginners的代码中的`navigation_sim_demo`例子中，由于要配置的参数太多，通常会将配置写在一个yaml文件中，我们用`param/move_base_params.yaml`来保存以上参数。而关于一些具体插件，比如`dwa_local_planner`则也会创建一个文件`param/dwa_local_planner.yaml`来保存它的设置。

## 10.2.3 Topic与Service

`move_base`输入输出的Topic在9.1节已经做了介绍，这里不再赘述。

`move_base`包含的Service包括：

- `make_plan`: `nav_msgs/GetPlan`类型，请求为一个目标点，响应为规划的轨迹，但不执行该轨迹。
- `clear_unknown_space`: `std_srvs/Empty`类型，允许用户清除未知区域地图。
- `clear_costmaps`: `std_srvs/Empty`类型，允许用户清楚代价地图上的障碍物。

## 10.3 costmap

costmap是Navigation Stack里的代价地图，它其实也是move\_base插件，本质上是C++的动态链接库，用过catkin\_make之后生成.so文件，然后move\_base在启动时会通过动态加载的方式调用其中的函数。

### 10.3.1 代价地图

之前我们在介绍SLAM时讲过ROS里的地图的概念，地图就是 /map 这个topic，它也是一张图片，一个像素代表了实际的一块面积，用灰度值来表示障碍物存在的可能性。然而在实际的导航任务中，光有一张地图是不够的，机器人需要能动态的把障碍物加入，或者清楚已经不存在的障碍物，有些时候还要在地图上标出危险区域，为路径规划提供更有用的信息。

因为导航的需要，所以出现了代价地图。你可以将代价地图理解为，在 /map 之上新加的另外几层地图，不仅包含了原始地图信息，还加入了其他辅助信息。

代价地图有以下特点： 1 . 首先，代价地图有两张，一张是 local\_costmap ，一张是 global\_costmap ，分别用于局部路径规划器和全局路径规划器，而这两个costmap都默认并且只能选择 costmap\_2d 作为插件。 2 . 无论是 local\_costmap 还是 global\_costmap ，都可以配置他们的Layer，可以选择多个层次。costmap的Layer包括以下几种：

- Static Map Layer : 静态地图层，通常都是SLAM建立完成的静态地图。
- Obstacle Map Layer : 障碍地图层，用于动态的记录传感器感知到的障碍物信息。
- 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 : 你还可以通过插件的形式自己实现costmap，目前已有 Social Costmap Layer 、 Range Sensor Layer 等开源插件。

可以同时选择多个Layer并存。

### 10.3.2 地图插件的选择

与9.2节中move\_base插件的配置类似，costmap配置也同样用yaml 来保存，其本质是维护在参数服务器上。由于costmap通常分为local和global的costmap，我们习惯把两个代价地图分开。以 ROS-Academy-for-Beginners 为例，配置写在了param文件夹下的 global\_costmap\_params.yaml 和 local\_costmap\_params.yaml 里。

global\_costmap\_params.yaml:

```
global_costmap:  
  global_frame: /map  
  robot_base_frame: /base_footprint  
  update_frequency: 2.0  
  publish_frequency: 0.5  
  static_map: true  
  rolling_window: false  
  transform_tolerance: 0.5  
  plugins:  
    - {name: static_layer,           type: "costmap_2d::StaticLayer"}  
    - {name: voxel_layer,           type: "costmap_2d::VoxelLayer"}  
    - {name: inflation_layer,      type: "costmap_2d::InflationLayer"}
```

### local\_costmap\_params.yaml:

```
local_costmap:  
  global_frame: /map  
  robot_base_frame: /base_footprint  
  update_frequency: 5.0  
  publish_frequency: 2.0  
  static_map: false  
  rolling_window: true  
  width: 4.0  
  height: 4.0  
  resolution: 0.05  
  origin_x: 5.0  
  origin_y: 0  
  transform_tolerance: 0.5  
  plugins:  
    - {name: voxel_layer,           type: "costmap_2d::VoxelLayer"}  
    - {name: inflation_layer,      type: "costmap_2d::InflationLayer"}
```

在 `plugins` 一项中可以设置Layer的种类，可以多层叠加。在本例中，考虑到局部地图并不需要静态地图，而只考虑传感器感知到的障碍物，因此可以删去`StaticLayer`。

## 10.4 map\_server & amcl

在某些固定场景下，我们已经知道了地图（无论通过SLAM还是测量），这样机器人每次启动最好就能直接加载已知地图，而是每次开机都重建。在这种情况下，就需要有一个节点来发布`/map`，提供场景信息了。

### 10.3.1 map\_server

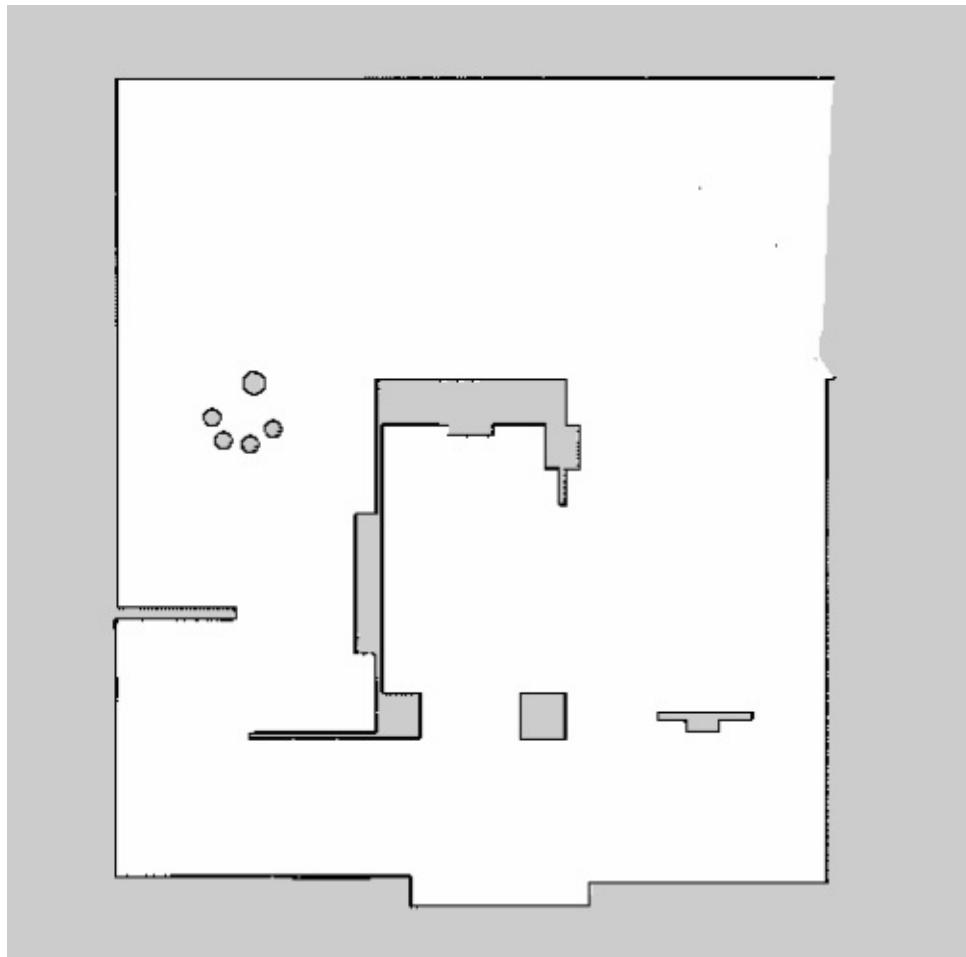
`map_server`是一个和地图相关的功能包，它可以将已知地图发布出来，供导航和其他功能使用，也可以保存SLAM建立的地图。

要让`map_server`发布`/map`，需要输入给它两个文件：

- 地图文件，通常为`pgm`格式；
- 地图的描述文件，通常为`yaml`格式

例如在`ROS-Academy-for-Beginners`里，我们提供了软件博物馆的地图文件，见`slam_sim_demo/maps`下：

## Software\_Museum.pgm



## Software\_Museum.yaml

```

image: Software_Museum.pgm #指定地图文件
resolution: 0.050000 #地图的分辨率 单位为m/pixel
origin: [-25.000000, -25.000000, 0.000000] #地图的原点
negate: 0 #0代表 白色为空闲 黑色为占据
occupied_thresh: 0.65 #当占据的概率大于0.65认为被占据
free_thresh: 0.196 #当占据的概率小于0.196认为无障碍

```

其中占据的概率  $\text{occ} = (255 - \text{color\_avg}) / 255.0$ ， $\text{color\_avg}$  为 RGB 三个通道的平均值。

有了以上两个文件，你可以通过指令来加载这张地图，map\_server 相关命令如下：

map_server 命令	作用
<code>rosrun map_server map_server Software_Museum.yaml</code>	加载自定义的地图
<code>rosrun map_server map_saver -f mymap</code>	保存当前地图为 mymap.pgn 和 mymap.yaml

当我运行 `rosrun map_server map_server ***.yaml` 时，会有以下的通信接口：

## Topic

通常我们是在launch文件中加载map\_server，发布地图。而map\_server发布的消息包括：

- /map\_metadata: 发布地图的描述信息
- /map: 发布锁存的地图消息

## Service

amcl的服务只有一个：

- static\_map: 用于请求和响应当前的静态地图。

## Param

amcl有一个参数需要设置，就是发布地图的frame。

- ~frame\_id: string类型，默认为map。绑定发布的地图与tf中的哪个frame，通常就是map。

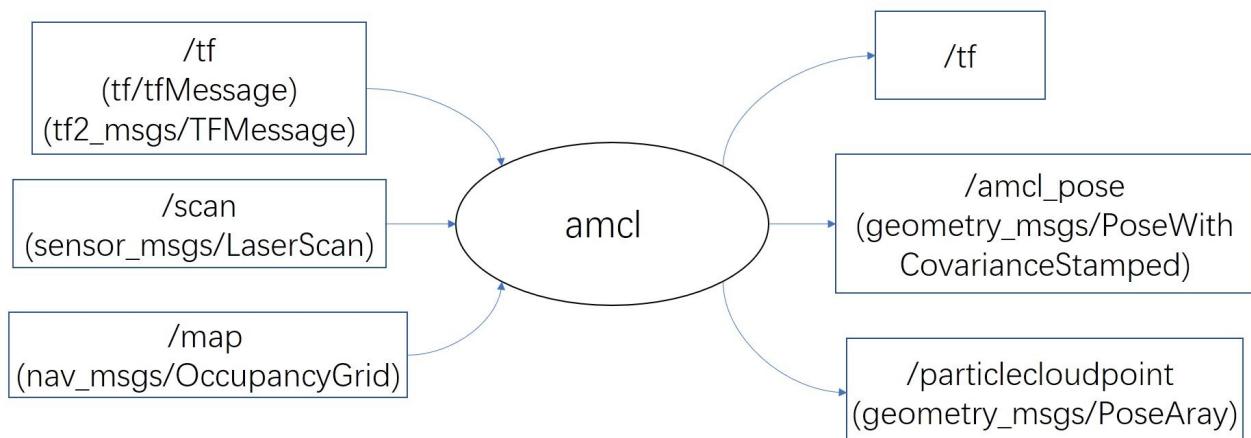
有两个概念不要搞混淆，map既是一个topic，也是一个frame，前者是topic通信方式中的一个话题，信息交互的频道，后者是tf中的一个坐标系，map\_frame需要和其他的frame相连通。

## 10.3.2 amcl

Adaptive Monte Carlo Localization(AMCL)，蒙特卡洛自适应定位是一种很常用的定位算法，它通过比较检测到的障碍物和已知地图来进行定位。

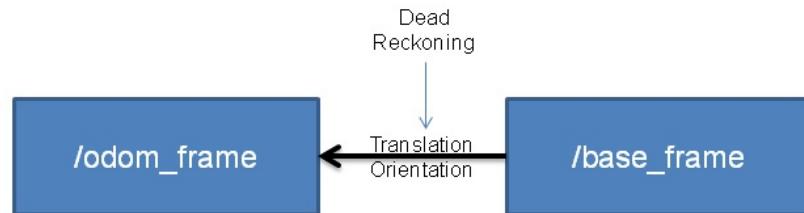
AMCL上的通信架构如上图所示，与之前SLAM的框架很像，最主要的区别是 /map 作为输入，而不是输出，因为AMCL算法只负责定位，而不管建图。

### AMCL

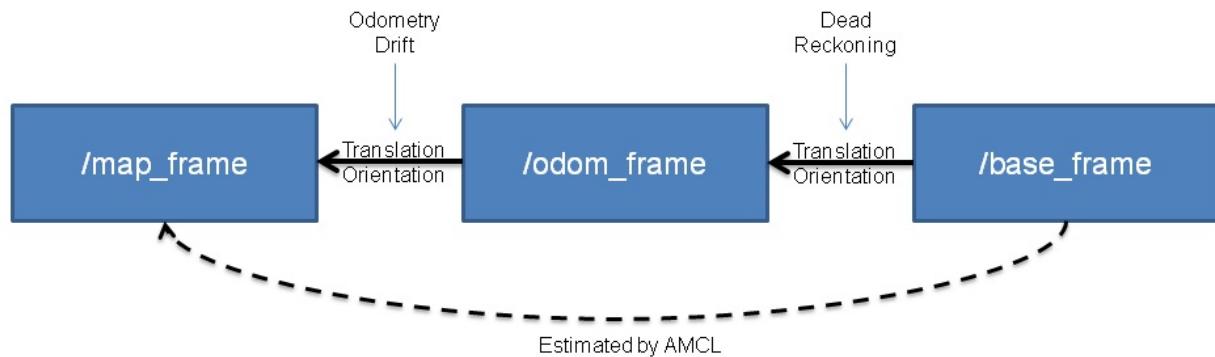


同时还有一点需要注意，AMCL定位会对里程计误差进行修正，修正的方法是把里程计误差加到 `map_frame` 和 `odom_frame` 之间，而 `odom_frame` 和 `base_frame` 之间是里程计的测量值，这个测量值并不会被修正。这一工程实现与之前gmapping、karto的做法是相同的。

## Odometry Localization

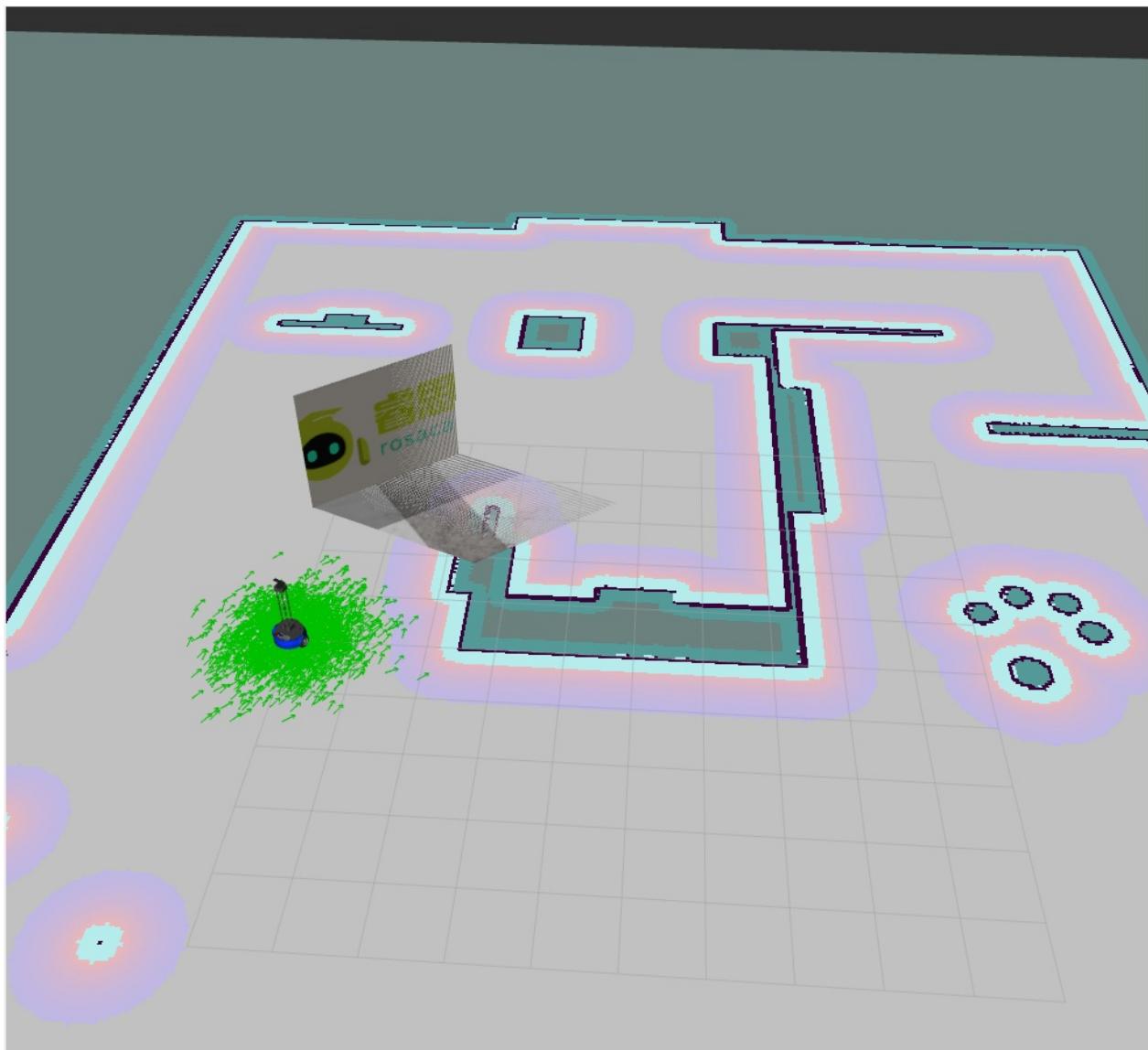


## AMCL Map Localization



## 演示截图

amcl算法演示效果图如下：



# TF附录

## 本章简介

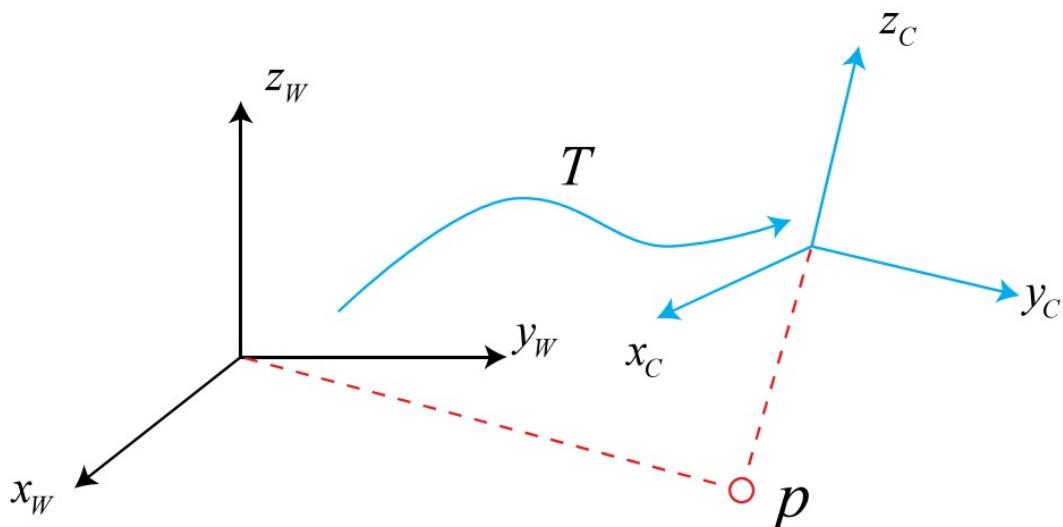
机器人的坐标变换一直以来是机器人学的一个难点，我们人类在进行一个简单的动作时，从思考到实施行动再到完成动作可能仅仅需要几秒钟，但是机器人来讲就需要大量的计算和坐标转换。其中的坐标转换TF和统一机器人描述格式URDF就是本章要详细介绍的内容。

本章的主要内容，首先是坐标转换的前提数学知识：三维空间刚体运动，包括旋转矩阵、欧拉角、四元数等。之后详细介绍了TF和URDF，最后介绍了制作URDF模型的过程。

# 7.1 三维空间刚体运动

## 7.1.1 旋转矩阵

在机器人运动的过程当中，我们通常会设定一个惯性坐标系（或者叫世界坐标系），姑且认为这个坐标系是固定不动的。例如： $X_W, Y_W, Z_W$  是固定不动的世界坐标系， $X_C, Y_C, Z_C$  是机器人坐标系。存在一个向量  $P$ ，在世界坐标系下的坐标是  $P_W$ ，在移动机器人坐标系下的坐标是  $P_C$ ，通常情况下，我们通过传感器已知移动机器人坐标系统下的坐标  $P_C$ ，来求  $P$  在世界坐标系下的坐标  $P_W$



为了求  $P_W$ ，我们必须知道机器人坐标系  $X_C, Y_C, Z_C$  相对于世界坐标系  $X_W, Y_W, Z_W$  做了哪些变换。我们定义世界坐标系经过变换矩阵  $T$  之后得到机器人坐标系（这可以通过计算里程和IMU的数据进行测量出来）（这也说明了为什么在机器人刚刚启动的时候 `odom` 和 `base_link` 坐标系必须是重合的，不然没有办法计算旋转矩阵），另外一般情况下，移动机器人运动是一个刚体运动，也就是说机器人的形状和大小不会因为坐标系不同而改变，这种变换叫做欧式变换。一个欧式变换可以由旋转和平移两个部分组成。首先我们考虑旋转问题，假设在世界坐标系下的单位正交基  $(e_x, e_y, e_z)$ ，在移动机器人坐标系下的单位正交基  $(e^x, e^y, e^z)$ ，那么，根据向量  $P$  的模可知：

$$\|P\| = \sqrt{e_x^2 + e_y^2 + e_z^2} = \sqrt{e^x_1^2 + e^y_1^2 + e^z_1^2}$$

因此， $P^T C = [e_x, e_y, e_z]^T \cdot [e^x, e^y, e^z]$ ，我们将  $[e_x, e_y, e_z]^T \cdot [e^x, e^y, e^z]$  记做旋转矩阵  $R$ ，因此上面的表达式可以简化为  $P^T C = R \cdot P^T W$ 。接下来是平移部分，假设平移部分是  $P^T C$  经过平移向量  $t$  后得到  $P^T C$ ，那么可以得到  $P^T C = P^T C + t = R \cdot P^T W + t$ 。所以通过旋转矩阵  $R$  和平移向量  $t$ ，我们可以描述从世界坐标系到移动机器人坐标系的坐标变换。但是

这种表达方式存在一个问题，对于连续的位置变换，例如机器人坐标系是随着时间在不断变换的，上面这种表达方式并不是一个线性的表达方式，假设我们经历了两次变换

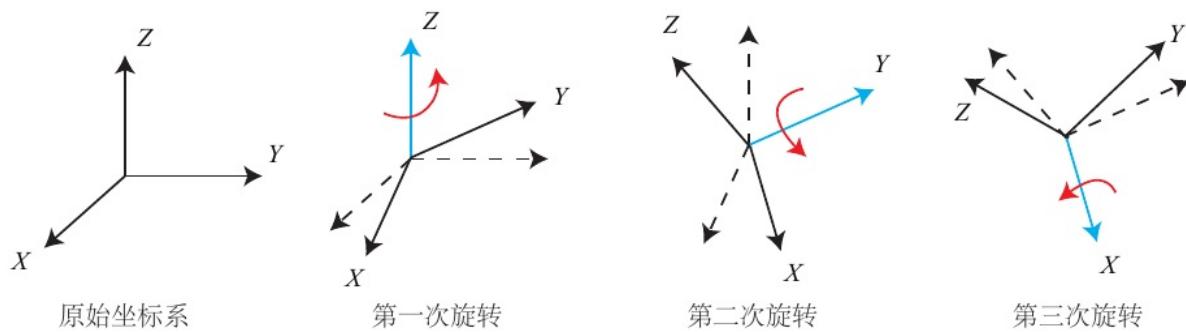
$\$\$R_{1,t_1}\$$ 和 $\$\$R_{2,t_2}\$$ 且满足：从a到b的变换 $\$\$b=R_1a+t_1\$\$$ ，从b到c的变换

$\$\$c=R_2b+t_2\$\$$ 。那么从a到c的变换是 $\$\$c=R_2(R_1a+t_1)+t_2\$\$$ 。并不是我们希望的形式 $\$\$c=Ra+t\$\$$ 。（然后我们采用齐次坐标的方式进行表达，详细的部分参考李群李代数）

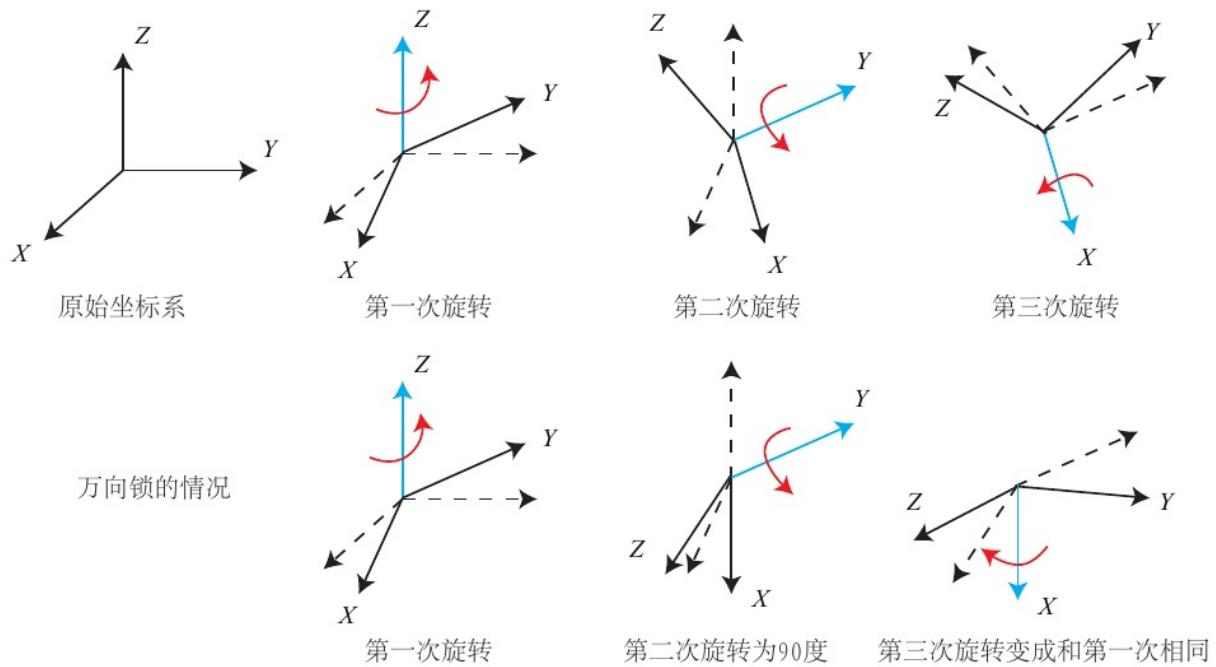
# 7.1 三维空间刚体运动

## 7.1.2 欧拉角

旋转本身就是一个很直观的现象。欧拉角可以提供一种非常直观的方式。他利用3个分离的转角，把一次旋转分解成3次绕不同的轴进行旋转。例如先绕x轴旋转，再绕y轴旋转，最后绕z轴旋转，这样就得到一个xyz轴的旋转。在欧拉角中一个常用的是“航偏-俯仰-翻滚”(yaw-pitch-roll)。可以简单记忆rpy-xyz。其中roll-对应着绕x轴旋转后的翻滚角。Pitch对应着绕y轴旋转后的俯仰值，yawd对应着绕z轴旋转后的航偏值。那么旋转部分就可以通过roll-pitch-yaw这三个量来描述。



在使用欧拉角这种表达方式的时候，会存在万向锁的问题。也就是一旦旋转pitch为90度，就会导致第一次旋转和第三次转换等价，丢失了一个表示维度。万向锁现象如下图所示





## 7.1 三维空间刚体运动

### 7.1.3 四元数

旋转矩阵用9个量来描述3自由度的旋转，具有冗余性；欧拉角虽然用3个量来描述3自由度的旋转，但是具有万向锁的问题，因此我们选择用四元数，（ROS当中描述转向的都是采用的四元数）。一个四元数拥有一个实部和三个虚部组成。

$\begin{aligned} q = & w + xi + yj + zk \end{aligned}$

三个虚部满足以下关系

$\begin{aligned} i^2 = j^2 = k^2 = -1 \\ ij = k, ji = -k, \quad kj = i, \quad kj = -i, \quad ki = j, \quad jk = -i \end{aligned}$

写成矩阵的样子就是 $\begin{bmatrix} w & x & y & z \end{bmatrix}^T$ ，其中 $\begin{vmatrix} q \end{vmatrix}^2 = w^2 + x^2 + y^2 + z^2 = 1$ ，从欧拉角到四元数的公式：

$\begin{aligned} q = & \begin{bmatrix} w & x & y & z \end{bmatrix} = \begin{bmatrix} \cos(\text{roll}/2) \cos(\text{pitch}/2) \cos(\text{yaw}/2) + \sin(\text{roll}/2) \sin(\text{pitch}/2) \sin(\text{yaw}/2) & \sin(\text{roll}/2) \cos(\text{pitch}/2) \cos(\text{yaw}/2) - \sin(\text{roll}/2) \sin(\text{pitch}/2) \sin(\text{yaw}/2) \\ \cos(\text{roll}/2) \sin(\text{pitch}/2) \cos(\text{yaw}/2) & \cos(\text{roll}/2) \sin(\text{pitch}/2) \sin(\text{yaw}/2) + \sin(\text{roll}/2) \cos(\text{pitch}/2) \cos(\text{yaw}/2) \\ -\sin(\text{roll}/2) \cos(\text{pitch}/2) & \cos(\text{roll}/2) \cos(\text{pitch}/2) \sin(\text{yaw}/2) - \sin(\text{roll}/2) \sin(\text{pitch}/2) \cos(\text{yaw}/2) \end{bmatrix} \end{aligned}$

从四元数转化到欧拉角公式

$\begin{aligned} \begin{bmatrix} \text{roll} & \text{pitch} & \text{yaw} \end{bmatrix} = & \begin{bmatrix} \arctan(2(wx + yz)) & \arcsin(2(wy - zx)) & \arctan(2(wz + xy)) \end{bmatrix} \end{aligned}$



# Navigation 工具包说明

source: <https://github.com/ros-planning/navigation>

## amcl工具包

输入参数：地图(包括了初始位置)，扫描数据(scan)，tf

输出参数：位置估计

注册topic：

```
scan(sensor_msgs/LaserScan) #激光数据  
tf(tf/tfMessage) #坐标转换  
initialpose(geometry_msgs/PoseWithCovarianceStamped) #初始位置和均值和方差  
map(nav_msgs/OccupancyGrid) #当use_map_topic设置为True时，amcl使用map这个topic接收地图
```

发布topic：

```
amcl_pose(geometry_msgs/PoseWithCovarianceStamped) #机器人在地图中的位姿估计，包括估计方差  
particlecloud(geometry_msgs/PoseArray) #滤波器估计的位置  
tf(tf/tfMessage) #发布从odom到map的转换关系
```

Services：

```
global_localization(std_srvs/Empty) #初始化全局定位，所有粒子完全随机分布在地图上  
request_nomotion_update(std_srvs/Empty) #手动更新粒子并发布更新后的粒子
```

调用的Services：

```
static_map(nav_msgs/GetMap) #amcl调用此服务接收地图，用以基于激光扫描的定位
```

相关参数：

amcl中可设置的相关参数，分为滤波器filter相关，扫描模型laser相关，里程计odometry模型相关

```

~min_particles(int, default:100) #最小粒子数
~max_particles(int, default:5000) #最大粒子数
~kld_err(double, default:0.01) #估计出的分布模型与真实分布模型的最大允许误差
~kld_z(double, default:0.99) #
~update_min_d(double, default:0.2 m) #两次滤波器更新之间平移的距离
~update_min_a(double, default:pi/6.0 rad) #两次滤波器更新之间旋转的弧度
~resample_interval(int, default:2) #两次采样之间滤波器更新的次数
~transform_tolerance(double, default:0.1 s) #
~recovery_alpha_slow(double, default:0.0) #slow average weight滤波器指数衰减速率，用于决定何时通过添加随机位姿进行恢复，可设置为0.001
~recovery_alpha_fast(double, default:0.0) #slow average weight滤波器衰减速率，可设置为0.1
~initial_pose_x(double, default:0.0 m) #初始位姿mean(x)，用以初始化高斯滤波器
~initial_pose_y(double, default:0.0 m) #初始位姿mean(y)，初始化高斯滤波器
~initial_pose_a(double, default:0.0 rad) #初始位姿mean(yaw)，初始化高斯滤波器
~initial_cov_xx(double, default:0.5*0.5) #初始位姿协方差covariance(x*x)，初始化高斯滤波器
~initial_cov_yy(double, default:0.5*0.5) #初始位姿协方差covariance(y*y)，初始化高斯滤波器
~initial_cov_aa(double, default:(pi/12)*(pi/12)) #初始位姿协方差covariance(yaw*yaw)，初始化高斯滤波器
~gui_publish_rate(double, default:-1.0 Hz) #扫描数据和路径可视化的最大速率，-1.0表示禁止
~save_pose_rate(double, default:0.5 Hz) #保存位姿(位姿+方差)的速率，保存的位姿用于不断初始化滤波器
~use_map_topic(bool, default:false) #设置为True时，amcl注册到map这个topic上，接收地图
~first_map_only(bool, default:false) #设置为True时，amcl使用固定的初始地图，即地图不会更新

```

激光模型相关参数：

```

~laser_min_range(double, default:-1.0) #最小扫描距离
~laser_max_range(double, default:-1.0) #最大扫描距离
~laser_max_beams(int, default:30) #均分扫描激光束数目
~laser_z_hit(double, default:0.95) #z_hit部分的混合权重
~laser_z_short(double, default:0.1) #z_short部分的混合权重
~laser_z_max(double, default:0.05) #z_max部分的混合权重
~laser_z_rand(double, default:0.05) #z_rand部分的混合权重
~laser_sigma_hit(double, default:0.2m) #z_hit部分的标准差
~laser_lambda_short(double, default:0.1) #z_short部分的指数衰减系数
~laser_likelihood_max_dist(double, default:2.0m) #对障碍物进行膨胀操作的最大距离，当模型为likelihood_field时使用
~laser_model_type(string, default:"likelihood_field") #选择模型，"beam","likelihood_field","likelihood_field_prob"

```

## Odometry模型相关

当odom\_model\_type为“diff”时，采用sample\_model\_odometry算法，见Probabilistic Robotics一书第136页，

此模型使用odom\_alpha1~odom\_alpha4参数，各参数定义见书中。当odom\_model\_type设置为“omni”时，使用odom\_alpha1~odom\_alpha5

五个参数。

```
~odom_model_type(string, default:"diff") #设置odom模型，可选"diff","omni","diff-corrected","omni-corrected"
~odom_alpha1(double, default:0.2) #从运动旋转分量估计出的旋转量的估计噪声
~odom_alpha2(double, default:0.2) #从运动平移分量估计出的旋转量的估计噪声
~odom_alpha3(double, default:0.2) #从运动平移分量估计出的平移量的估计噪声
~odom_alpha4(double, default:0.2) #从运动旋转分量估计出的平移量的估计噪声
~odom_alpha5(double, default:0.2) #平移相关噪声参数
~odom_frame_id(string, default:"odom") #里程计使用的坐标系
~base_frame_id(string, default:"base_link") #机器人底座坐标系
~global_frame_id(string, default:"map") #定位系统所发布的坐标系
~tf_broadcast(bool, default:true) #设置为false时，禁止amcl发布从global frame到odometry坐标系的转换
```

## base\_local\_planner工具包

### 发布的topic

```
~<name>/global_plan(nav_msgs/Path) #使用全局规划出的路径的一部分，以便局部规划器沿迹前进，同时也可显示在屏幕上
~<name>/global_plan(nav_msgs/Path) #得分最高的局部规划或轨迹，用以显示
~<name>/cost_cloud(sensor_msgs/PointCloud2) #损失网格，主要是为直观显示，可修改publish_cost_grid_pc参数是否启用此功能
```

### 注册的topic

```
odom(nav_msgs/Odometry) #接收里程计消息，包含机器人当前的速度信息。
```

### 参数

此工具包中包含的参数主要有：机器人设置相关，目标容差，前向模拟，路径得分，震荡预防，全局规划

#### 机器人设置参数

```
~<name>/acc_lim_x(double, default:2.5) #x方向最大加速度
~<name>/acc_lim_y(double, default:2.5) #y方向最大加速度
~<name>/acc_lim_theta(double, default:3.2) #旋转角加速度最大值
~<name>/max_vel_x(double, default:0.5) #最大前进速度
~<name>/min_vel_x(double, default:0.1) #最小前进速度，设置到适当的值以保证机器人底盘能够克服摩擦力前进
~<name>/max_vel_theta(double, default:1.0) #最大旋转角速度
~<name>/min_vel_theta(double, default:-1.0) #最小旋转角速度
~<name>/min_in_place_vel_theta(double, default:0.4) #原地旋转最小旋转角速度
~<name>/backup_vel(double, default:-0.1) #不建议使用，已改换成escape_vel
~<name>/escape_vel(double, default:-0.1) #退出速度，单位为m/s，数值为负
~<name>/holonomic_robot(bool, default:true) #确定速度命令是从holonomic还是非holonomic的机器上f发出的
~<name>/y_vels(list, default:[-0.3, -0.1, 0.1, 0.3]) #holonomic机器人的偏向移动速度
```

#### 目标容错相关参数

```
~<name>/yaw_goal_tolerance(double, default:0.05) #到达目的地的偏航角偏差阈值
~<name>/xy_goal_tolerance(double, default:0.10) #到达目的地的x&y平面距离偏差阈值
~<name>/latch_xy_goal_tolerance(bool, default:false) #如果目标点锁定，()
```

## 前进模拟参数

```

~<name>/sim_time(double, default:1.0s) #前进模拟的时间
~<name>/sim_granularity(double, default:0.025m) #步长大小，轨迹上采样点间的距离长度
~<name>/angular_sim_granularity(double, default:~<name>/sim_granularity) #步长大小，旋转分量
~<name>/vx_samples(integer, default:3) #每次使用x方向速度的样点数目
~<name>/vtheta_samples(integer, default:20) #每次使用的旋转方向的样点数目
~<name>/controller_frequency(double, default:20.0) #控制器被调用的频率

```

## 轨迹评分模型参数

**cost = pdist\_scale\*(**轨迹终点到路径的距离，以地图网格数或者m做单位，与**meter\_scoring**参数有关)

```

+ gdist_scale*(轨迹终点到局部目标点的距离，以地图网格数目或者m做单位，与meter_scoring有关)
+ occdist_scale*(轨迹上的最大障碍物损失)

~<name>/meter_scoring(bool, default:false) #决定是否使用gdist_scale和pdist_scale参数分别确定goal_distance和path_distance以地图网格数或m来表示
~<name>/pdist_scale(double, default:0.6) #决定控制器与规划路径的接近程度，最大为5.0
~<name>/gdist_scale(double, default:0.8) #决定控制器与局部目标点的接近程度，同时控制速度，最大为5.0
~<name>/occdist_scale(double, default:0.01) #决定控制器试图避障的程度
~<name>/heading_lookahead(double, default:0.325) #当原地转动得分不同时，向前看到的距离
~<name>/heading_scoring(bool, default:faulse) #是根据机器人与路径的方向还是根据其与路径的距离打分
~<name>/heading_scoring_timestep(double, default) #以时间s衡量的沿轨迹的转动幅度(启用heading_scoring)
~<name>/dwa(bool, default:true) #是否使用dwa算法，另外可用Trajectory Rollout，实验证明dwa与TR性能相似，但计算量更小
~<name>/publish_cost_grid_pc(bool, default:false) #规划路径时是否发布损失函数网格图
~<name>/global_frame_id(string, default:odom) #cost_cloud的参考系，应当与局部损失图全局坐标一致

```

## 防振荡模型参数

```

~<name>/oscillation_reset_dist(double, default:0.05) #振荡标志距离，即来回运动的距离在0.05时可认为时振动

```

## 全局规划参数

```

~<name>/prune_plan(bool, default:true) #确定是否抹去机器人走过的轨迹，设置为true时，机器人会抹去身后1m以外的轨迹

```



## ROS相关参数

```
~<name>/step_size(double, default:损失地图的分辨率) #机器人后退步长，单位为米  
~<name>/min_dist_from_robot(double, default:0.10) #机器人与发送目标点位置的最小距离，即当机器  
人到达此位置时，即向局部规划器发送目标点位置
```

实施一种修复机制，将距离机器人一定距离范围外的代价地图恢复为静态地图

#### 相关参数

```
~<name>/reset_distance(double, default:3.0) # 阈值半径，默认为3m，距离机器人3m外的代价地图被  
恢复成静态地图
```

## 注册的Topic

```
~<name>/footprint(geometry_msgs/Polygon) #机器人底盘几何特性，描述形状
```

## 发布的Topic

```
~<name>/grid(nav_msgs/OccupancyGrid) #损失地图的网格值  
~<name>/grid_updates(map_msgs/OccupancyGridUpdate) #损失地图更新后的值  
~<name>/voxel_grid(costmap_2d/VoxelGrid) #选择性发布voxel格点
```

## Hydro前版本相关参数

Hydro之后的ROS版本都提供costmap\_2d层的插件。如果你没有提供插件参数，程序默认设置为

Hydro前版本，且会加载一系列默认命名空间内的默认插件。默认的命名空间有static\_layer, obstacle\_layer和inflation\_layer。

## 插件

```
~<name>/plugins(sequence, default:Hydro版本前) #插件序列。每种插件都是一个字典，名字和类型。名字用来定义命名空间。
```

## 坐标系和tf参数

```
~<name>/global_frame(string, default:"/map") #代价地图的全局坐标系  
~<name>/robot_base_frame(string, default:"base_link") #机器人底座所对应的坐标系  
~<name>/transform_tolerance(double, default:0.2) #坐标转换延迟，单位s
```

## 速率参数

```
~<name>/update_frequency(double, default:0.5) #地图更新频率，Hz  
~<name>/publish_frequency(double, default:0.0) #地图发布频率
```

## 地图管理参数

```
~<name>/rolling_window(bool, default:false) #是否使用滚动窗口查看代价地图，如果static_map设置为true，此参数必须设置为false  
~<name>/always_send_full_costmap(bool, default:false) #如果为true，则如果代价地图有更新，则将整个代价地图发送到~<name>/grid，如果为false，则只发送改变的部分到~<name>/grid_updates  
  
~<name>/width(int, default:10) #地图宽度  
~<name>/height(int, default:10) #地图高度  
~<name>/resolution(int, default:0.05) #地图分辨率  
~<name>/origin_x(double, default:0.0) #全局坐标系的x坐标  
~<name>/origin_y(double, default:0.0) #全局坐标系的y坐标
```

使用DWA算法，实现平面局部导航，给定全局规划和代价地图，局部规划起产生速度指令，发送给移动底座。

本软件支持所有底座可用闭包多边形或者圆来描述的机器人。其配置参数可在launch文件中设置。这些参数

也可动态再配置。

## 简介

dwa\_local\_planner提供一个能够驱动底座的控制器，该控制器连接了路径规划器和机器人。使用地图，

规划器产生从起点到目标点的运动轨迹，在移动时，规划器在机器人周围产生一个函数，用网格地图表示。

控制器的工作就是利用这个函数来确定发送给机器人的速度 $dx$ ,  $dy$ ,  $dtheta$

### DWA算法的基本思想

1. 在机器人控制空间离散采样( $dx$ ,  $dy$ ,  $dtheta$ )
2. 对每一个采样的速度进行前向模拟，看看在当前状态下，使用该采样速度移动一小段时间后会发生什么。
3. 评价前向模拟得到的每个轨迹，是否接近障碍物，是否接近目标，是否接近全局路径以及速度等等。舍弃非法路径
4. 选择得分最高的路径，发送对应的速度给底座

### 发布的topics

```
~<name>/global_plan(nav_msgs/Path) #局部规划器所跟踪的全局路径部分。  
~<name>/local_plan(nav_msgs/Path) #在评价函数中得分最高的局部路径
```

### 注册的topics

```
odom(nav_msgs/Odometry) #里程计信息给局部规划器提供当前速度信息。
```

### 可调参数

#### 机器人本身参数

```

~<name>/acc_lim_x(double, default:2.5) #x方向加速度的绝对值
~<name>/acc_lim_y(double, default:2.5) #y方向加速度的绝对值
~<name>/acc_lim_th(double, default:3.2) #旋转加速度的绝对值
~<name>/max_trans_vel(double, default:0.55) #平移速度最大值绝对值
~<name>/min_trans_vel(double, default:0.1) #平移速度最小值的绝对值
~<name>/max_vel_x(double, default:0.55) #x方向最大速度的绝对值
~<name>/min_vel_x(double, default:0.0) #x方向最小值绝对值，如果是负值表示后退运动
~<name>/max_vel_y(double, default:0.1) #y方向最大速度的绝对值
~<name>/min_vel_y(double, default:-0.1) #y方向最小速度的绝对值
~<name>/max_rot_vel(double, default:1.0) #最大旋转速度的绝对值
~<name>/min_rot_vel(double, default:0.4) #最小旋转速度的绝对值

```

## 目标点容错参数

```

~<name>/yaw_goal_tolerance(double, default:0.05) #到达目标点时偏航角误差
~<name>/xy_goal_tolerance(double, default:0.10) #到达目标点时，在xy平面内与目标点的距离误差
~<name>/latch_xy_goal_tolerance(double, default:false) #设置为true时，如果到达容错距离内，则机器人就会原地旋转，即便转动时会跑出容错距离外

```

## 前进仿真参数

```

~<name>/sim_time(double, default:1.7) #向前仿真轨迹的时间
~<name>/sim_granularity(double, default:0.025) #步长，轨迹上采样点之间的距离
~<name>/vx_samples(integer, default:3) #x方向速度空间的采样点数
~<name>/vy_samples(integer, default:10) #y方向速度空间采样点数
~<name>/vth_samples(integer, default:20) #旋转方向的速度空间采样点数
~<name>/controller_frequency(double, default:20.0) #控制器被调用的频率

```

## 轨迹评分参数

```

~<name>/path_distance_bias(double, default:32.0) #定义控制器与给定路径接近程度
~<name>/goal_distance_bias(double, default:24.0) #定义控制器与局部目标点的接近程度，并控制速度
~<name>/occ_dist_scale(double, default:0.01) #定义控制器躲避障碍物的程度
~<name>/stop_time_buffer(double, default:0.2) #为防止碰撞，机器人必须提前停止的时间长度。
~<name>.scaling_speed(double, default:0.25) #启动机器人底座的速度
~<name>/max_scaling_factor(double, default:0.2) #最大缩放参数

```

## 防振荡参数

```

~<name>/oscillation_reset_dist(double, default:0.05) #机器人运动多远距离才会重置振荡标记

```

## 全局参数

```
~<name>/prune_plan(bool, default:true) #机器人前进时是否清除身后1m外的轨迹
```

## 简介

本包提供一个node，`fake_localization`，用来替代定位系统，提供可被`amcl`使用的接口，常在仿真中

使用，计算量更少。它将`odometry`数据转换成位姿信息、粒子云、`tf`数据，数据格式与`amcl`发布的数据相同

## 节点**fake\_localization**

### 注册的topic

```
base_pose_ground_truth(nav_msgs/Odometry) #仿真器发布的机器人位置  
initialpose(geometry_msgs/PoseWithCovarianceStamped) #允许使用rviz或nav_view等工具设置位姿  
。
```

### 发布的topics

```
amcl_pose(geometry_msgs/PoseWithCovarianceStamped) #转发仿真器发布的位姿信息  
particlecloud(geometry_msgs/PoseArray) #用以在rviz和nav_view中可视化的粒子云
```

### 相关参数

```
~odom_frame_id(string, default:"odom") #里程计坐标系  
~delta_x(double, default:0.0) #仿真坐标系和节点发布的地图坐标系之间的x方向距离  
~delta_y(double, default:0.0) #仿真坐标系和节点发布的地图坐标系之间的y方向距离  
~delta_yaw(double, default:0.0) #仿真坐标系和节点发布的地图坐标系之间的偏航角弧度  
~global_frame_id(string, default:/map) #全局坐标系  
~base_frame_id(string, default:base_link) #机器人自身的底座坐标系
```

# 全局路径规划器

## 标准方式

所有参数使用默认值

## 网格路径方式

```
use_grid_path=True
```

此中方式下，路径是一系列网格中心的连线

## 简单势函数计算

```
use_quadratic=False
```

## 发布的topic

```
~<name>/plan(nav_msgs/Path) #最新计算得到的规划路径，每次解算出新的路径，路径规划器都会将其发布
```

## 相关参数

```
~<name>/allow_unknown(bool, default:true) #是否允许规划器规划穿过未知区域的路径  
~<name>/default_tolerance(double, default:0.0) #路径规划器规划出的终点容错距离  
~<name>/visualize_potential(bool, default:false) #是否显示从PointCloud2计算得到的势区域  
~<name>/use_dijkstra(bool, default:true) #如果设置为true，则使用dijkstra算法，否则使用A*算法  
~<name>/use_quadratic(bool, default:true) #如果设置为true，使用二次函数近似势函数。否则使用更加简单的计算方式  
~<name>/use_grid_path(bool, default:false) #如果设置为true，则创建一条沿着网格边界的路径，否则使用梯度下降法。  
~<name>/old_navfn_behavior(bool, default:false) #如果出于某些原因，你想让global_planner完全复制navfn的行为，则设置为true
```

map\_server提供了map\_server节点，提供地图数据。也提供map\_saver命令行工具，可以保存动态产生的地图

### Map格式

保存的地图是图片文件和yaml文件。yaml文件描述地图信息，图片文件将地图进行编码

#### 图片格式

图片文件描述地图是否有障碍无等信息，白色像素代表无障碍，黑色像素代表障碍物，两者之间的代表未知区域。

实际上地图可以是彩色或者灰度图，但大多数地图都是灰度图。YAML文件中定义三种区域之间的颜色分类阈值。

某种灰度的填充率可以用下面公式计算 $\text{occ} = (255 - \text{color\_avg}) / 255.0$ 。

### YAML格式

```
image: testmap.png
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

### map\_server介绍

是一个ROS节点，用于从硬盘中读取地图文件，

#### 用法

```
map_server <map.yaml>
```

#### 示例

```
rosrun map_server map_server mymap.yaml
```

需要指出的是地图数据必须从latched topic或者service中获取。

#### 发布的topic

```
map_metadata(nav_msgs/MapMetaData) #从这个topic中接收地图元数据
map(nav_msgs/OccupancyGrid) #从这个topic中接收地图
```

### Services

```
static_map(nav_msgs/GetMap) #从这个service中接收地图
```

## 相关参数

```
~frame_id(string, default:"map") #发布的地图所使用的坐标系
```

## map\_saver

map\_saver保存地图到硬盘，例如，从SLAM的建图service保存到硬盘

```
rosrun map_server map_saver [-f mapname]
```

map\_saver接收地图数据并将其写入到map.png和map.yaml文件中。-f指定保存文件名

例子：

```
rosrun map_server map_saver -f mymap
```

## 注册的topic

```
map(nav_msgs/OccupancyGrid) #接收地图
```

保存了用于和move\_base节点进行通讯的消息。这些消息是MoveBase.action自动产生的。

### MoveBase.action

```
geometry_msgs/PoseStamped target_pose  
---  
---  
geometry_msgs/PoseStamped base_position
```

target\_pose是目标点，base\_position是底座现在的位置。

## 简介

move\_base包采用action机制，使移动底座到达给定的目标点。move\_base这个节点连接全局规划器和局部

规划器，以便能够完成全局导航任务。它能够支持任何继承自nav\_core::BaseGlobalPlanner接口的

全局规划器和任何继承自nav\_core::BaseLocalPlanner接口的局部规划器。move\_base维护两个代价地图

一个属于全局规划器，另一个属于局部规划器

## Action注册的topic

```
move_base/goal(move_base_msgs/MoveBaseActionGoal) #move_base所搜寻的目标  
move_base/cancel(actionlib_msgs/GoalID) #请求取消目标点
```

## Action发布的topic

```
move_base/feedback(move_base_msgs/MoveBaseActionFeedback) #反馈包括底座当前相对于世界坐标系  
的位置  
move_base/status(actionlib_msgs/GoalStatusArray) #提供发送给move_base action的目标的状态信  
息  
move_base/result(move_base_msgs/MoveBaseActionResult) #对move_base action来说，result为  
空
```

## 节点注册的topic

```
move_base_simple/goal(geometry_msgs/PoseStamped) #为不需要跟踪目标执行状态的情况提供非action  
接口
```

## 节点发布的topic

```
cmd_vel(geometry_msgs/Twist) #速度指令流，用以移动底座
```

## Services

```
~make_plan(nav_msgs/GetPlan) #允许用户在指定位姿后获取规划出的路径，而不必真正沿着路径运动  
~clear_unknown_space(std_srvs/Empty) #允许用户直接清除机器人周围的未知区域，适合costmap停止了  
很长时间，在一个新的地方重新启动的时候使用  
~clear_costmaps(std_srvs/Empty) #允许用户告诉move_base清除costmaps中的障碍物，可能导致撞上物  
体
```

## 相关参数

```
~base_global_planner(string, default:"navfn/NavfnROS") #指定用于move_base的全局规划器插件  
名称  
~base_local_planner(string, default:"base_local_planner/TrajectoryPlannerROS") #指定用于  
move_base的局部规划器名称  
~recovery_behaviors(list, default:[{name:conservative_reset, type:clear_costmap_recover  
y/ClearCostmapRecovery},{name:rotate_recovery,type:rotate_recovery/RotateRecovery},{n  
ame:aggressive_reset,type:clear_costmap_recovery/ClearCostmapRecovery}]) #用于move_base  
修复的插件列表，当move_base找不到可行的路径规划方案时，move_base将按照这些顺序执行操作。每个操作执行  
完后，move_base会在此尝试生成路径规划方案，如果成功，则继续正常操作；否则，启动下一个修复操作。  
~controller_frequency(double, default:20.0) #控制循环的执行频率，也可认为是速度发布指令的频率  
~planner_patience(double, default:5.0) #规划器在空间清除操作前，能够进行路径规划的时间  
~conservative_reset_dist(double, default:3.0) #当在地图中清理出空间时，距离机器人几米远的障碍  
会从代价地图中清除  
~recovery_behavior_enabled(bool, default:true) #是否启用move_base修复机制  
~clearing_rotation_allowed(bool, default:true) #在清除空间操作时，是否允许底座原地旋转  
~shutdown_costmaps(bool, default:false) #当move_base不活动时，是否关闭costmaps  
~oscillation_timeout(double, default:0.0) #执行修复机制前，允许振荡的时长  
~oscillation_distance(double, default:0.5) #来回运动在多大距离以上不会被认为是振荡  
~planner_frequency(double, default:0.0) #全局规划操作的执行频率。如果设置为0.0，则全局规划器仅  
在接收到新的目标点或者局部规划器报告路径堵塞时才会重新执行规划操作  
~max_planning_retries(int32, default:-1) #在执行修复操作前，允许规划器重新规划路径的次数，-1.0  
表示无限多次重新尝试
```

move\_slow\_and\_clear::MoveSlowAndClear是一种简单的修复机制，它将costmap中的信息清除，并

限制机器人的速度。注意，这种修复机制并不是绝对安全，机器人还是有可能撞上物体，当使用者自定义速度时

就会发生这种情况。另外，这种修复机制只兼容允许通过dynamic\_reconfigure设置最大速度的局部规划器

，例如dwa\_local\_planner

### 相关参数

```
~<name>/clearing_distance(double, default:0.5) #半径范围内的障碍物将被清除  
~<name>/limited_trans_speed(double, default:0.25) #执行修复机制时的平移速度  
~<name>/limited_rot_speed(double, default:0.25) #执行修复机制时的旋转速度  
~<name>/limited_distance(double, default:0.3) #机器人运动的距离  
~<name>/planner_namespace(string, default:"DWAPlannerROS") #用于重新配置参数的  
规划器命名空间，其中的max_trans_vel和max_rot_vel将会被重新配置。
```

navfn提供了一个快速插入的导航函数，用于对机器人移动底座进行全局路径规划。规划器假设机器人底座为圆形，用代价地图的形式找到代价最小的路径。使用的算法是Dijkstra，同时也支持A\*算法。

### 发布的topic

```
~<name>/plan(nav_msgs/Path) #由navfn计算得到的最新路径，每次planner计算得到新的路径，都会发布
```

### 参数

```
~<name>/allow_unknown(dool, default:true) #是否允许navfn创建穿过未知区域的路径  
~<name>/planner_window_x(double, default:0.0) #限制规划器作用的范围  
~<name>/planner_window_y(double, default:0.0) #限制规划器作用的范围  
~<name>/default_tolerance(double, default:0.0) #目标点的容错距离，规划器规划到的点距离真正的目标点有偏差  
~<name>/visualize_potential(bool, default:false) #是否显示navfn用PointCloud2计算得到的势函数
```

nav\_core包提供了机器人导航的通用接口，现在主要提供了  
BaseGlobalPlanner, BaseLocalPlanner

和RecoveryBehavior接口。所有希望用做move\_base插件的规划器和修复器都必须继承自这些接口。

### BaseGlobalPlanner

nav\_core::BaseGlobalPlanner提供了全局规划器的接口，所有的全局规划器要想作为插件用在move\_base

节点中，都必须继承自这个接口。目前使用此接口的全局规划器有：

```
global_planner  
navfn  
carrot_planner
```

### BaseLocalPlanner

nav\_core::BaseLocalPlanner提供了局部规划器的接口，所有的局部规划器若想作为插件应用在move\_base

中，都必须继承此接口。目前使用此接口的局部规划器有：

```
base_local_planner  
eband_local_planner  
teb_local_planner
```

### RecoveryBehavior

nav\_core::RecoveryBehavior提供修复机制的接口，所有的修复机制若要作为插件应用在move\_base

中，都必须继承此接口，目前使用此接口的修复机制有：

```
clear_costmap_recovery  
rotate_recovery
```

robot\_pose\_ekf包是用来估计机器人的3D位姿。它使用EKF和机器人的6维度模型，结合轮式里程计，IMU，

视觉里程计等。其基本思想是提供不同传感器之间的松耦合。

### 如何使用**robot\_pose\_ekf**

配置：

在robot\_pose\_ekf下有一个launch文件，此文件中包含了一些可配置的参数：

```
freq :滤波器的更新发布频率  
sensor_timeout :当传感器停止向滤波器发送消息，滤波器会等多长时间  
odom_used, imu_used, vo_used :是否启用这些输入
```

运行

Build：

```
$ rosdep install robot_pose_ekf  
$ roscd robot_pose_ekf  
$ rosmake
```

Run:

```
$ rosrun robot_pose_ekf robot_pose_ekf
```

节点

```
robot_pose_ekf
```

### 注册的topic

```
odom(nav_msgs/Odometry) #2D位姿，实际上是3D位姿，只是忽略了z位置和roll, pitch  
imu_data(sensor_msgs/Imu) #3D姿态  
vo(nav_msgs/Odometry) #3D位姿
```

注意，robot\_pose\_ekf并不需要3种传感器同时具备，每种传感器信息都能够估计出机器人的位姿和对应的协方差。同时，你也可以加入自己的传感器，例如GPS等

### 发布的topic

```
robot_pose_ekf/odom_combined(geometry_msgs/PoseWithCovarianceStamped) #滤波器输出
```

提供的**tf**变换

```
odom_combined -> base_footprint
```

本软件包提供了一种通过旋转360度来清除空间的修复机制

#### 旋转修复相关参数

```
~<name>/sim_granularity(double, default: 0.017) #两次检测间转动的弧度，  
~<name>/frequency(double, default: 20.0) #速度发送频率
```

#### TrajectoryPlannerROS相关参数

##### 局部规划器相关参数

```
~TrajectoryPlannerROS/yaw_goal_tolerance(double, default:0.05) #目标点角度容错  
~TrajectoryPlannerROS/acc_lim_th(double, default:3.2) #旋转加速度rad/sec^2  
~TrajectoryPlannerROS/max_rotational_vel(double, default:1.0) #最大旋转速度rad/sec^2  
~TrajectoryPlannerROS/min_in_place_rotational_vel(double, default:0.4) #最小原地旋转速度
```