

###写在前面###

ROS 机器人操作系统在机器人应用领域很流行，依托代码开源和模块间协作等特性，给机器人开发者带来了很大的方便。我们的机器人“miiboo”中的大部分程序也采用 ROS 进行开发，所以本文就重点对 ROS 基础知识进行详细的讲解，给不熟悉 ROS 的朋友起到一个抛砖引玉的作用。本文主要内容：

- 1.ROS 是什么
- 2.ROS 系统整体架构
- 3.在 ubuntu16.04 中安装 ROS kinetic
- 4.如何编写 ROS 的第一个程序 hello_world
- 5.编写简单的消息发布器和订阅器
- 6.编写简单的 service 和 client
- 7.理解 tf 的原理
- 8.理解 roslaunch 在大型项目中的作用
- 9.熟练使用 rviz
- 10.在实际机器人上运行 ROS 高级功能预览

###正文###

- 1.ROS 是什么



（图 1）ROS 的图标

ROS 是一个适用于机器人的开源的元操作系统。其实它并不是一个真正的操作系统，其底层的任务调度、编译、寻址等任务还是由 Linux 操作系统完成，也就是说 ROS 实际上是运行在 Linux 上的次级操作系统。但是 ROS 提供了操作系统应用的各种服务（如：硬件抽象、底层设备控制、常用函数实现、进程间消息传递、软件包管理等），也提供了用于获取、编译、跨平台运行代码的工具和函数。ROS 主要采用松耦合点对点进程网络通信，目前主要还是支持 Ubuntu 系统，windows 和 Mac OS 目前支持的还不好，所以推荐在 Ubuntu 系统上安装使用 ROS。

1.1 ROS 的特性

	特性
1	松耦合的机制方便机器人软件框架的组织
2	最丰富的机器人功能库，方便快速搭建原型
3	非常便利的数据记录、分析、仿真工具，方便调试
4	学界和产业界的标准，方便学习和交流

（图 2）ROS 的特性

总结起来就是，使用 ROS 能够方便迅速的搭建机器人原型。ROS 使用了 BSD 许可证，这是一个很宽松的开放许可证，允许在商业和闭源产品中使用，这一点对开发产品的创业公司很重要。ROS 当前的代码统计量，总行数超过 1400 万，作者超过 2477 名。代码语言以 C++ 为主，63.98%的代码是用 C++编写的，排名第二的是 python，占 13.57%，可以说 ROS 基

本上都是使用这两种语言，来实现大部分的功能。

1.2 ROS 的结构

这里主要从四个方面来解读 ROS 的结构，设计思想、核心概念、核心模块、核心工具。

ROS 的设计思路主要是分布式架构，将机器人的功能和软件做成一个个节点，然后每个节点通过 **topic** 进行沟通，这些节点可以部署在同一台机器上，也可以部署在不同机器上，还可以部署在互联网上。

ROS 的核心概念主要是节点和用于节点间通信的话题与服务。管理器 **Master** 管理节点与话题之间通信的过程，并且还提供一个参数服务用于全局参数的配置。ROS 通过功能包集 **stack** 和功能包 **package** 来组织代码。

ROS 的核心模块包括：通信结构基础、机器人特性功能、工具集。通信结构基础主要是消息传递、记录回放消息、远程过程调用、分布式参数系统；机器人特性功能主要是标准机器人消息、机器人几何库、机器人描述语言、抢占式远程过程调用、诊断、位置估计、定位导航；工具集主要是命令式工具、可视化工具、图形化接口。

ROS 核心工具很丰富，ROS 常用命令工具是 **rostopic**、**rosservice**、**roscall**、**rospack**、**roslaunch**、**roscpp**、**roscppcli**、**rostopic**、**rosservice**、**roscall**、**rospack**、**roslaunch**、**roscpp**、**roscppcli**；ROS 常用可视化工具是 **rqt**、**rviz**；ROS 用于存储与回放数据的工具 **roscpp**；ROS 的 **log** 系统记录软件运行的相关信息；ROS 还拥有强大的第三方工具支持：三维仿真环境 **Gazebo**、计算机视觉库 **OpenCV**、点云库 **PCL**、机械臂控制库 **Movel**、工业应用库 **Industrial**、机器人编程工具箱 **MRPT**、实时控制库 **Orocos**。

1.3 ROS 的学习资源

官网：www.ros.org

源码：github.com

Wiki：wiki.ros.org

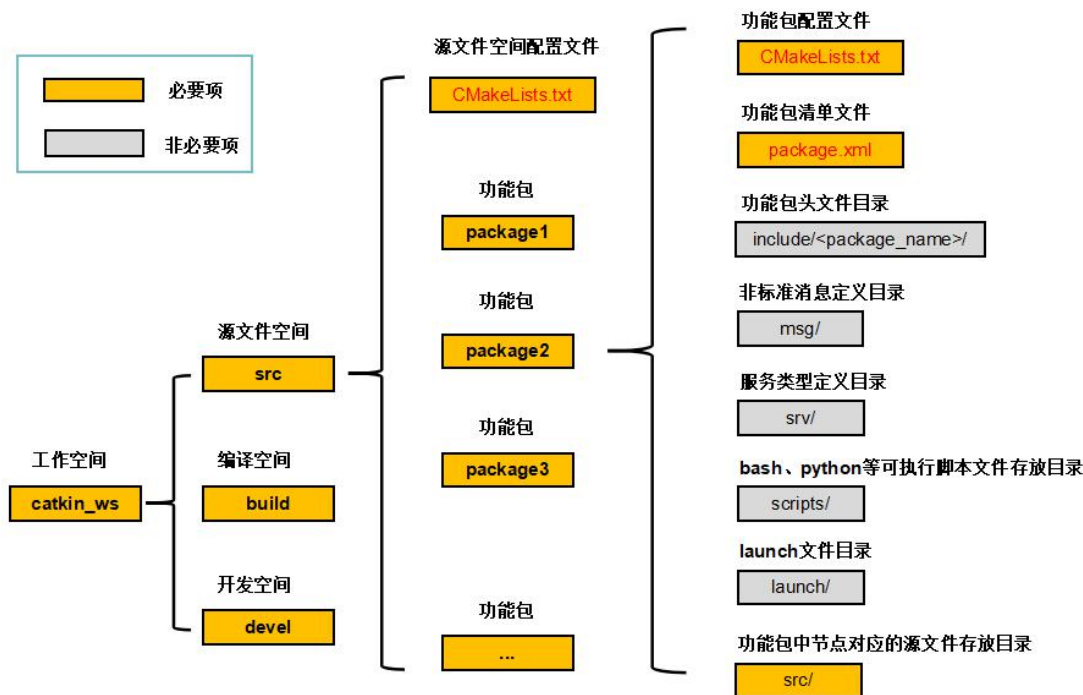
问答：answers.ros.org

2. ROS 系统整体架构

由于 ROS 系统的组织架构比较复杂，简单从一个方面来说明很难说清楚。按照 ROS 官方的说法，我们可以从 3 个方面来理解 ROS 系统整体架构，这 3 个方面分别是文件系统级、计算图级、开源社区级。

2.1. 从文件系统级理解 ROS 架构

如果你是刚刚接手 ROS 方面的开发或项目，你肯定会觉得 ROS 中的各种概念非常奇怪，但是当你使用 ROS 熟练之后，你就觉得这些概念很好理解了。与其他操作系统相似，一个 ROS 程序的不同组件要被放在不同的文件夹下，这些文件夹是根据不同的功能来对文件进行组织的，如图 3。



(图 3) 文件系统级理解 ROS 架构

(1) 工作空间

工作空间是一个包含功能包、可编辑源文件和编译包的文件夹，当你想同时编译不同的功能包时非常有用，并且可以保存本地开发包。当然，用户可以根据自己的需要创建多个工作空间，在每个工作空间中开发不同用途的功能包。不过作为学习，我们先以一个工作空间为例。如图 3，我们创建了一个名为 `catkin_ws` 的工作空间，该工作空间下会有 3 个文件夹：`src`、`build`、`devel`。

src 源文件空间：这个文件夹放置各个功能包和一个用于这些功能包的 CMake 配置文件 `CMakeLists.txt`。这里做一下说明，由于 ROS 中的源码采用 `catkin` 工具进行编译，而 `catkin` 工具又是基于 `cmake` 技术的，所以我们会在 `src` 源文件空间和各个功能包中都会见到一个文件 `CMakeLists.txt`，这个文件就是起编译配置的作用。

build 编译空间：这个文件夹放置 CMake 和 `catkin` 编译功能包时产生的缓存、配置、中间文件等。

devel 开发空间：这个文件夹放置编译好的可执行程序，这些可执行程序是不需要安装就能直接运行的。一旦功能包源码编译和测试通过后，可以将这些编译好的可执行文件直接导出与其他开发人员分享。

(2) 功能包

功能包是 ROS 中软件组织的基本形式，一个功能包具有用于创建 ROS 程序的最小结构和最少内容，它可以包含 ROS 运行的进程（节点）、配置文件等。如图 3，一个功能包中主要包含这几个文件：

CMakeLists.txt 功能包配置文件：用于这个功能包 `cmake` 编译时的配置文件。

package.xml 功能包清单文件：用 xml 的标签格式标记这个功能包的各类相关信息，比如包的名称、依赖关系等。主要作用是为了更容易的安装和分发功能包。

include/<package_name> 功能包头文件目录：你可以把你的功能包程序包含的 *.h 头文件放在这里，`include` 下之所以还要加一级路径 `<package_name>` 是为了更好的区分自己定义的头文件和系统标准头文件，`<package_name>` 用实际功能包的名称替代。不过这个文件夹不是必要项，比如有些程序没有头文件的情况。

msg 非标准消息定义目录：消息是 ROS 中一个进程（节点）发送到其他进程（节点）的信息，消息类型是消息的数据结构，ROS 系统提供了很多标准类型的消息可以直接使用，如果你要使用一些非标准类型的消息，就需要自己来定义该类型的消息，并把定义的文件放在这里。不过这个文件夹不是必要项，比如程序中只使用标准类型的消息的情况。

srv 服务类型定义目录：服务是 ROS 中进程（节点）间的请求/响应通信过程，服务类型是服务请求/响应的数据结构，服务类型的定义放在这里。如果要调用此服务，你需要使用该功能包名称和服务名称。不过这个文件夹不是必要项，比如程序中不使用服务的情况。

scripts 可执行脚本文件存放目录：这里用于存放 bash、python 或其他脚本的可执行文件。不过这个文件夹不是必要项，比如程序中不使用可执行脚本的情况。

launch 文件目录：这里用于存放 *.launch 文件，*.launch 文件用于启动 ROS 功能包中的一个或多个节点，在含有多个节点启动的大型项目中很有用。不过这个文件夹不是必要项，节点也可以不通过 launch 文件启动。

src 功能包中节点源文件存放目录：一个功能包中可以有多个进程（节点）程序来完成不同的功能，每个进程（节点）程序都是可以单独运行的，这里用于存放这些进程（节点）程序的源文件，你可以在这里再创建文件夹和文件来按你的需求组织源文件，源文件可以用 c++、python 等来书写。

为了创建、修改、使用功能包，ROS 给我们提供了一些实用的工具，常用的有下面这些工具。

rospack: 用于获取信息或在系统中查找工作空间。

catkin_create_pkg: 用于在工作空间的 src 源空间下创建一个新的功能包。

catkin_make: 用于编译工作空间中的功能包。

rosdep: 用于安装功能包的系统依赖项。

rqt_dep: 用于查看功能包的依赖关系图。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

（3）消息

消息是 ROS 中一个进程（节点）发送到其他进程（节点）的信息，消息类型是消息的数据结构，ROS 系统提供了很多标准类型的消息可以直接使用，如果你要使用一些非标准类型的消息，就需要自己来定义该类型的消息。

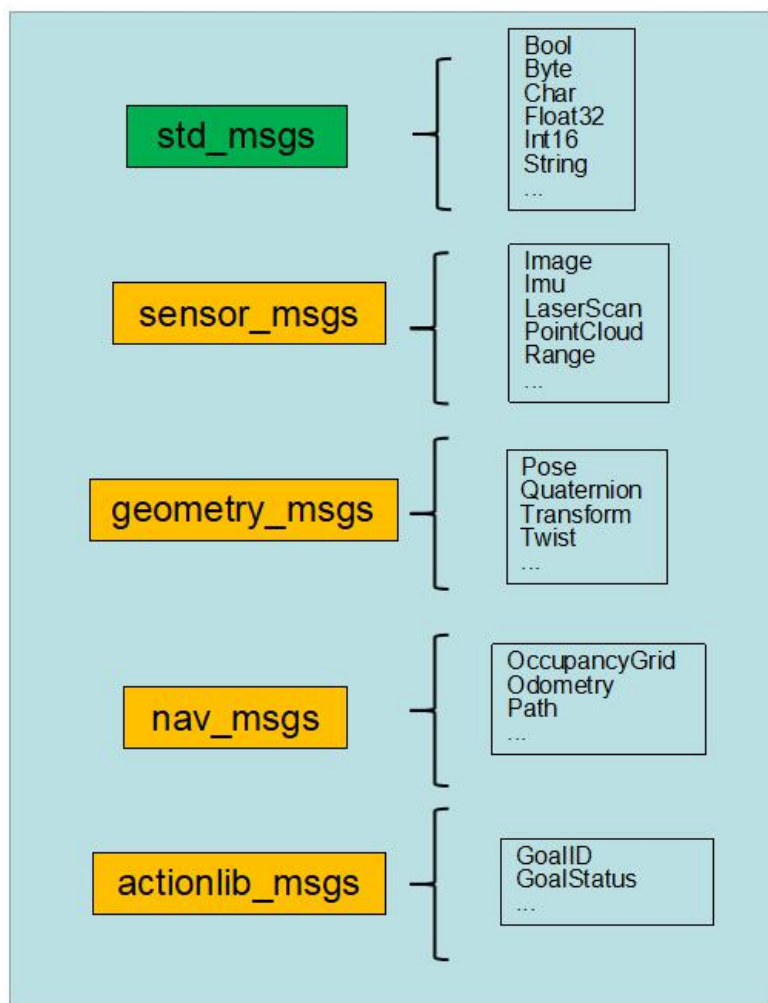
ROS 使用了一种精简的消息类型描述语言来描述 ROS 进程（节点）发布的数据值。通过这种描述语言对消息类型的定义，ROS 可以在不同的编程语言（如 c++、python 等）书写的程序中使用此消息。不管是 ROS 系统提供的标准类型消息，还是用户自定义的非标准类型消息，定义文件都是以 *.msg 作为扩展名。消息类型的定义分为两个主要部分：字段的数据类型和字段的名称，简单点说就是结构体中的变量类型和变量名称。比如下面的一个示例消息定义文件 example.msg 的内容，如图 4，int32、float32、string 就是字段的数据类型，id、vel、name 就是字段的名称。

example.msg 文件的内容	
int32	id
float32	vel
string	name

（图 4）一个示例消息定义文件

在大多数情况下，我们都可以使用 ROS 系统提供的标准类型的消息来完成的任务，这得益于

ROS 系统提供了丰富的标准类型的消息。经常用到的类型包括：基本类型（std_msgs）、通用类型（sensor_msgs、geometry_msgs、nav_msgs、actionlib_msgs），如图 5。



（图 5）ROS 系统提供的常用标准类型的消息

不难发现 std_msgs 下面定义的是经 ROS 封装后的最基本的数据类型，比如 Bool、Char、Int16 等；sensor_msgs 下面定义的是跟传感器数据相关的数据类型，比如 Image 对应的就是摄像头的数据类型，Imu 对应的就是 IMU 传感器的数据类型，LaserScan 对应的就是激光雷达的数据类型，PointCloud 对应的就是点云扫描传感器（如深度相机）的数据类型，Range 对应的就是距离测量传感器（如超声波、红外测距）的数据类型；geometry_msgs 下定义的是跟几何有关的数据类型，比如 Pose 用来描述机器人在空间的位姿，Quaternion 用四元数描述空间方向，Transform 用来描述不同坐标系之间的转移关系，Twist 用来描述机器人运动时的位姿、速度等状态信息；nav_msgs 下定义的是跟机器人导航相关的数据类型，比如 OccupancyGrid 是栅格地图的数据类型，Odometry 是机器人通过轮式码盘或其他方式融合得到的里程计的数据类型，Path 是路径规划算法计算得到的导航路劲的数据类型；actionlib_msgs 下定义的是 actionlib 控制过程相关的数据类型，比如 GoalID 描述发送出去的导航目标的 ID 号，GoalStatus 描述执行导航目标过程的过程状态信息。如果了解更多 ROS 系统的消息类型的细节，最好的方式是去 ROS wiki 看官方的文档，链接如下：

http://wiki.ros.org/std_msgs/

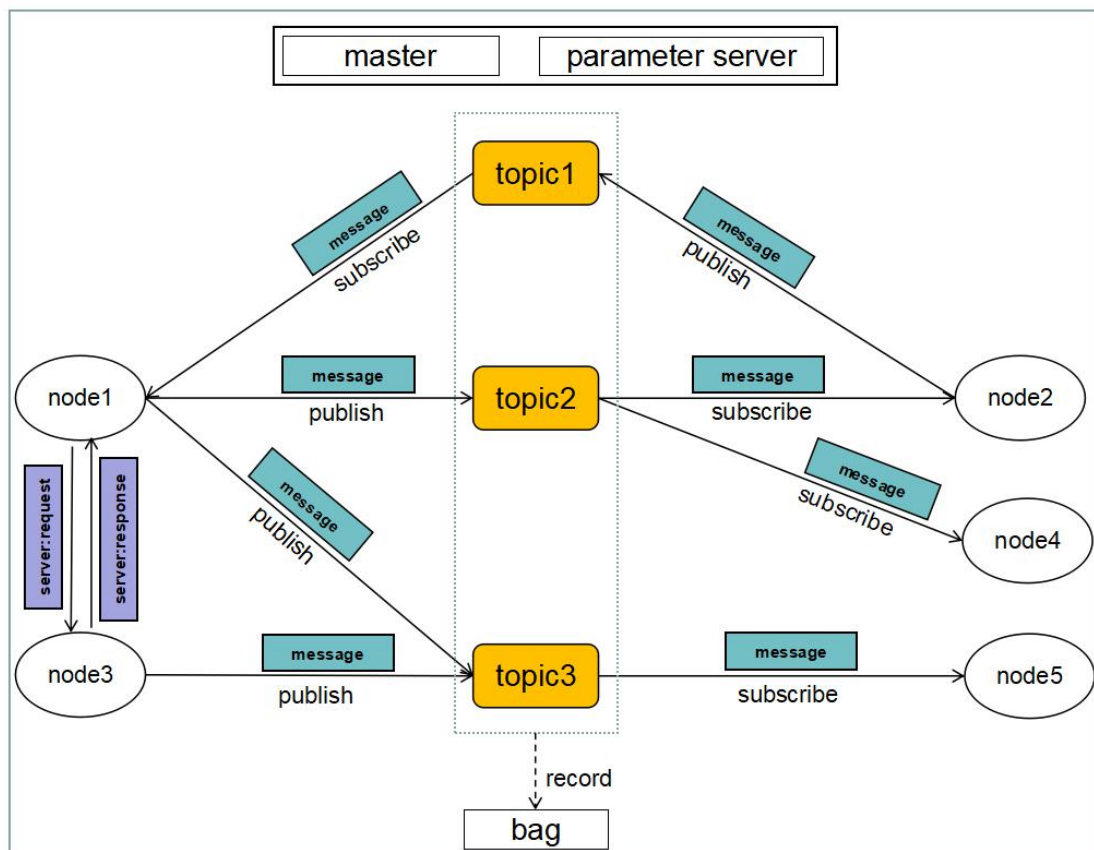
http://wiki.ros.org/common_msgs/

（4）服务

服务是 ROS 中进程（节点）间的请求/响应通信过程，服务类型是服务请求/响应的数据结构。服务类型的定义借鉴了消息类型的定义方式，所以这里就不在赘述了。区别在于，消息数据是 ROS 进程（节点）间多对多广播式通信过程中传递的信息；服务数据是 ROS 进程（节点）间点对点的请求/响应通信过程传递的信息。

2.2.从计算图级理解 ROS 架构

ROS 会创建一个连接所有进程（节点）的网络，其中的任何进程（节点）都可以访问此网络，并通过该网络与其他进程（节点）交互，获取其他进程（节点）发布的信息，并将自身数据发布到网络上，这个计算图网络中的节点（node）、主题（topic）、服务（server）等都要有唯一的名称做标识，如图 6。



（图 6）计算图级理解 ROS 架构

（1）节点

节点是主要的计算执行进程，功能包中创建的每个可执行程序在被启动加载到系统进程后，该进程就是一个 ROS 节点，如图 6 中的 node1、node2、node3 等都是节点（node）。节点都是各自独立的可执行文件，能够通过主题（topic）、服务（server）或参数服务器（parameter server）与其他节点通信。ROS 通过使用节点将代码和功能解耦，提高了系统的容错力和可维护性。所以你最好让每一个节点都具有特定的单一的功能，而不是创建一个包罗万象的大节点。节点如果用 c++ 进行编写，需要用到 ROS 提供的库 roscpp；节点如果用 python 进行编写，需要用到 ROS 提供的库 rospy。

ROS 提供了处理节点的工具，用于节点信息、状态、可用性等查询操作，例如可以用下面的命令对正在运行的节点进行操作。

rostopic info <node_name>: 用于输出当前节点信息。

rostopic kill <node_name>: 用于杀死正在运行节点进程来结束节点的运行。

rostopic list: 用于列出当前活动的节点。

roscat machine <hostname>: 用于列出指定计算机上运行的节点。

roscat ping <node_name>: 用于测试节点间的网络连通性。

roscat cleanup: 用于将无法访问节点的注册信息清除。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(2) 消息

节点通过消息（message）完成彼此的沟通。消息包含一个节点发送给其他节点的信息数据。关于消息类型的知识在前面已经讲述了，这里就不再展开。

ROS 提供了获取消息相关信息的命令工具，这里列举出一些常用的命令，来具体看看吧。

rostopic show <message_type>: 用于显示一条消息的字段。

rostopic list: 用于列出所有消息。

rostopic package <package_name>: 用于列出功能包的所有消息。

rostopic packages: 用于列出所有具有该消息的功能包。

rostopic users <message_type>: 用于搜索使用该消息类型的代码文件。

rostopic md5 <message_type>: 用于显示一条消息的 MD5 求和结果。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(3) 主题

每个消息都必须发布到相应的主题（topic），通过主题来实现在 ROS 计算图网络中的路由转发。当一个节点发送数据时，我们就说该节点正在向主题发布消息；节点可以通过订阅某个主题，接收来自其他节点的消息。通过主题进行消息路由不需要节点之间直接连接，这就意味着发布者节点和订阅者节点之间不需要知道彼此是否存在，这样就保证了发布者节点与订阅者节点之间的解耦合。同一个主题可以有多个订阅者也可以有多个发布者，不过要注意必须使用不同的节点发布同一个主题。每个主题都是强类型的，不管是发布消息到主题还是从主题中订阅消息，发布者和订阅者定义的消息类型必须与主题的消息类型相匹配。

ROS 提供了操作主题的命令工具，这里列举出一些常用的命令，来具体看看吧。

rostopic bw </topic_name>: 用于显示主题所使用的带宽。

rostopic echo </topic_name>: 用于将主题中的消息数据输出到屏幕。

rostopic find <message_type>: 用于按照消息类型查找主题。

rostopic hz </topic_name>: 用于显示主题的发布频率。

rostopic info </topic_name>: 用于输出活动主题、发布的主题、主题订阅者和服务的信息。

rostopic list: 用于列出当前活动主题的列表。

rostopic pub </topic_name> <message_type> <args>: 用于通过命令行将数据发布到主题。

rostopic type </topic_name>: 用于输出主题中发布的消息类型。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(4) 服务

在一些特殊的场合，节点间需要点对点的高效率通信并及时获取应答，这个时候就需要用服务的方式进行交互。提供服务的节点叫服务端，向服务端发起请求并等待响应的节点叫客户端，客户端发起一次请求并得到服务端的一次响应，这样就完成了一次服务通信过程，例如图 6 中，node1 向 node3 发起一次请求，并得到 node3 返回给 node1 的响应。服务通

信过程中服务的数据类型需要用户自己定义，与消息不同，节点并不提供标准服务类型。服务类型的定义文件都是以*.srv 为扩展名，并且被放在功能包的 srv/文件夹下。

ROS 提供了操作服务的命令工具，这里列举出一些常用的命令，来具体看看吧。

rosservice call </service_name> <args>: 用于通过命令行参数调用服务。

rosservice find <service_type>: 用于根据服务类型查询服务。

rosservice info </service_name>: 用于输出服务信息。

rosservice list: 用于列出活动服务清单。

rosservice type </service_name>: 用于输出服务类型。

rosservice uri </service_name>: 用于输出服务的 ROSRPC URI。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(5) 节点管理器

节点管理器 (master) 用于节点的名称注册和查找等，也负责设置节点间的通信。如果在你的整个 ROS 系统中没有节点管理器，就不会有节点、消息、服务之间的通信。由于 ROS 本身就是一个分布式的网络系统，所以你可以在某台计算机上运行节点管理器，在这台计算机和其他台计算机上运行节点。

ROS 中提供了跟节点管理器相关的命令行工具，就是 **roscore**。

roscore: 用于启动节点管理器，这个命令会加载 ROS 节点管理器和其他 ROS 核心组件。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(6) 参数服务器

参数服务器 (parameter server) 能够使数据通过关键词存储在一个系统的核心位置。通过使用参数，就能够在节点运行时动态配置节点或改变节点的工作任务。参数服务器是可通过网络访问的共享的多变量字典，节点使用此服务器来存储和检索运行时的参数。

ROS 中关于参数服务器的命令行工具，请看下面的常用命令。

rosparam list: 用于列出参数服务器中的所有参数。

rosparam get <parameter_name>: 用于获取参数服务器中的参数值。

rosparam set <parameter_name> <value>: 用于设置参数服务器中参数的值。

rosparam delete <parameter_name>: 用于将参数从参数服务器中删除。

rosparam dump <file>: 用于将参数服务器的参数保存到一个文件。

rosparam load <file>: 用于从文件将参数加载到参数服务器。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

(7) 消息记录包

消息记录包 (bag) 是一种用于保存和回放 ROS 消息数据的文件格式。消息记录包是一种用于存储数据的重要机制，它可以帮助记录一些难以收集的传感器数据，然后通过反复回放数据进行算法的性能开发和测试。ROS 创建的消息记录包文件以*.bag 为扩展名，通过播放、停止、后退操作该文件，可以像实时会话一样在 ROS 中再现情景，便于算法的反复调试。

ROS 提供消息记录包相关的命令行工具，请看下面的常用命令。

rosbag <args>: 用来录制、播放和执行操作。

关于这些工具命令的具体使用方法，会在后面的章节中结合实例进行具体的讲解。这里只是先介绍给大家，让大家有个概念上的了解，感兴趣的朋友也可以自己上网了解这些命令的具体用法。

2.3.从开源社区级理解 ROS 架构

ROS 开源社区级的概念主要是 ROS 资源，即通过各个独立的网络社区分享 ROS 方面的软件 and 知识。

(1) ROS 发行版

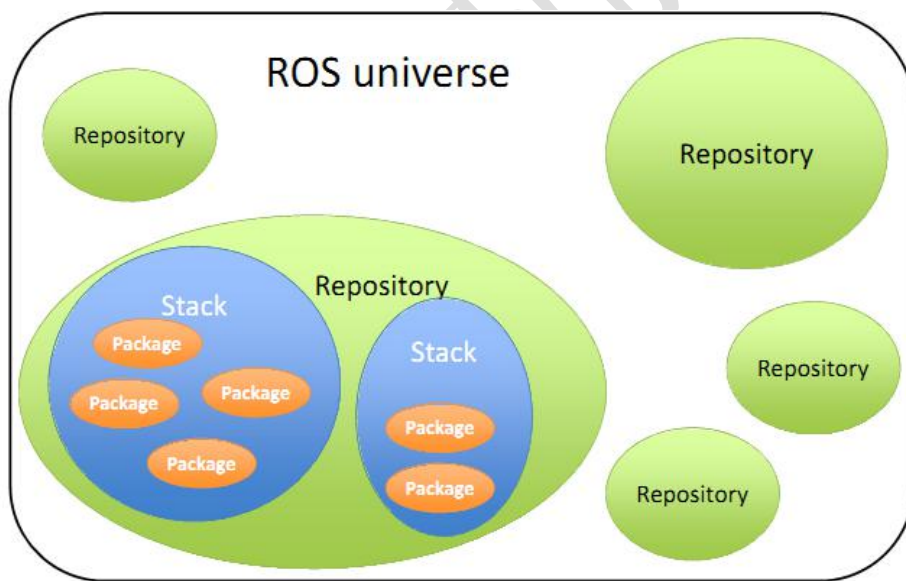
ROS 发行版跟 Linux 发行版起类似的作用，ROS 发行版是内置了一系列常用功能包的 ROS 系统安装包，可以被直接安装到我们的操作系统中。如图 7，是 ROS 的各个发行版。

发行时间	版本名称	对应 ubuntu 版本	稳定版本
2018	ROS-melodic	ubuntu18.04	是
2017	ROS-lunar	ubuntu17.04	否
2016	ROS-kinetic	ubuntu16.04	是
2015	ROS-jade	ubuntu15.04	否
2014	ROS-indigo	ubuntu14.04	是

(图 7) ROS 的各个发行版

(2) ROS 软件代码库

ROS 依赖于开源或共享软件的源代码，这些代码由不同的机构共享与发布，比如 github 源码共享，ubuntu 软件仓库等等。如图 8，是 ROS 软件代码库的社区组织形式。



(图 8) ROS 软件代码库的社区组织形式

(3) ROS 文档社区

ROS wiki 是记录有关 ROS 系统各种文档的主要论坛社区，任何人都可以注册账户、贡献自己的文件、提供更正或更新、编写教程及其他行为。感兴趣可以进入 ROS wiki 的主页面瞧瞧 <http://wiki.ros.org/>。

(4) ROS 问答社区

ROS 开发者可以通过这个资源去提问和寻找 ROS 相关的答案，ROS Answer 主页面 <https://answers.ros.org/>。

3.在 ubuntu16.04 中安装 ROS kinetic



（图 9）安装 ROS 软硬件配置总结

使用 ROS 进行机器人的学习和开发，一般需要一个机器人平台和一个调试工作平台。机器人平台上出于性价比和功耗考虑一般采用 ARM 嵌入式板作为硬件设备，比如树莓派 3、RK3399 开发板、nvidia-jetson-tx2 等；调试工作平台一般采用 x86 个人电脑，比如笔记本电脑、台式电脑等。ARM 嵌入式板的厂家一般都会提供相应定制化的 ubuntu 系统，定制化主要体现在硬件外设驱动和一些加快系统速度的优化，作为软件开发人员可以不必考虑这些问题直接当做普通 ubuntu 使用就行了，比如针对树莓派 3 定制化的 ubuntu-mate 系统；x86 个人电脑上的话就可以直接安装 ubuntu 官方发布的系统就行了，你可以把 ubuntu 直接安装在物理机上，这样 ubuntu 运行会更加流畅，你也可以把 ubuntu 安装在虚拟机上，这样可以更方便的切换使用机器上原来的系统（如 windows 系统）和虚拟机上的 ubuntu 系统。不管是采用何种硬件，在硬件上以何种方式安装上 ubuntu 及其定制化 ubuntu，一旦我们拥有了一个可用的 ubuntu 系统，就可以在这个 ubuntu 系统上安装当下流行的 ROS 发行版了（本文写作时最流行的 ROS 发行版是 ROS-kinetic）。安装 ROS 软硬件配置的总结，如图 9。

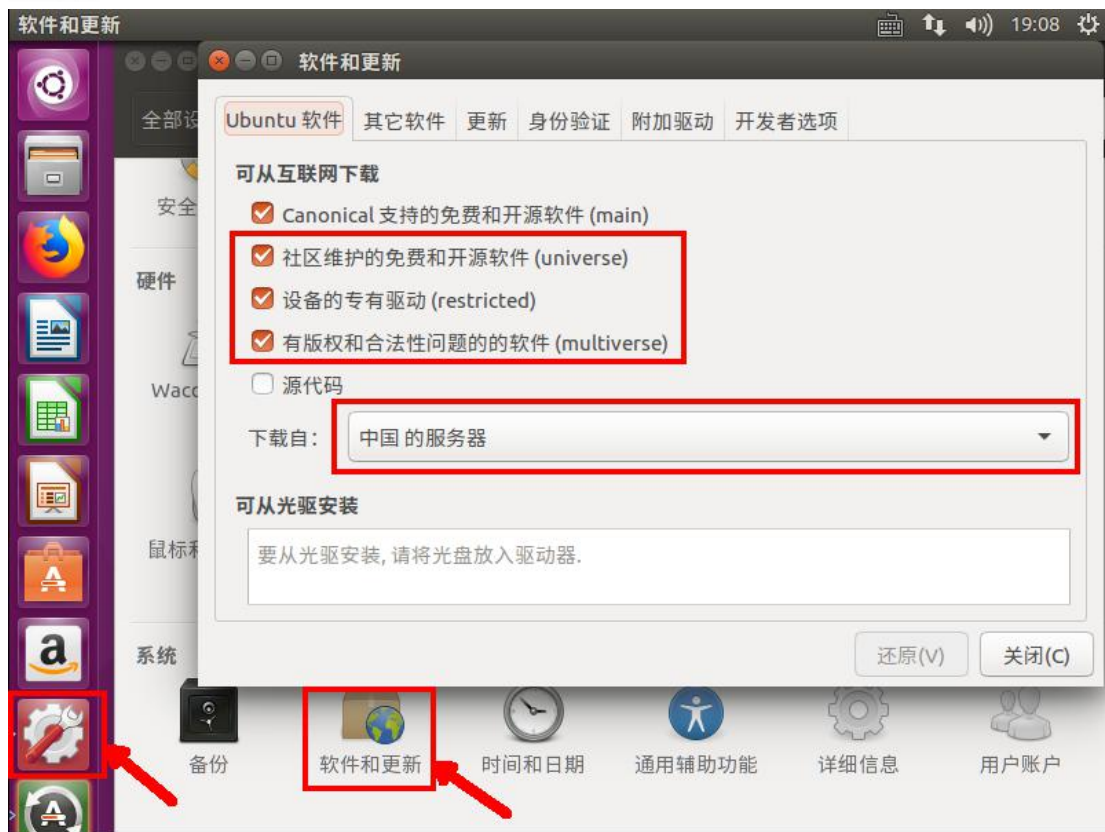
机器人平台上安装 ROS 软硬件配置，推荐大家使用树莓派 3 作为硬件，ubuntu-mate-16.04 作为操作系统，安装 ROS-kinetic 这个 ROS 发行版。这一部分的具体讲解将在后续的文章中展开。

调试工作平台上安装 ROS 软硬件配置，推荐大家使用 x86 个人电脑（笔记本电脑、台式电脑都可以）作为硬件，在虚拟机上运行 ubuntu16.04 系统，安装 ROS-kinetic 这个 ROS 发行版。由于在虚拟机上安装运行 ubuntu16.04 系统已经在前面的文章《Linux 基础知识》中详细讲解了，这里就默认我们已经拥有了一个运行在虚拟机上的 ubuntu16.04 系统了。接下来就着重讲解如何安装 ROS 发行版 ROS-kinetic。

其实在 ubuntu 上安装 ROS，有很详细的 ROS 官方教程，感兴趣的朋友可以直接参考官方教程 <http://wiki.ros.org/kinetic/Installation/Ubuntu>。由于官方教程用英文书写，为了方便大家阅读，我将官方教程翻译过来，方便大家学习，下面正式进入安装。**温馨提醒，由于不同的编辑器对过长的句子换行的规则不同，下面的命令被自动换行后可能影响正常的阅读，请直接参阅官方教程中的命令** <http://wiki.ros.org/kinetic/Installation/Ubuntu>。

（1）配置 ubuntu 的资源库

系统设置->软件和更新->Ubuntu 软件，可以打开如图 10 中的资源库配置界面，确保“universe”，“restricted”、“multiverse”被勾选了，“下载自”选项中选择“中国的服务器”，这样下载更新软件速度会更快点。不过一般情况下，以上选项都是默认设置好了的。



(图 10) 资源库配置界面

(2) 设置 ubuntu 的 sources.list

打开命令行终端，输入如下命令：

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
```

(3) 设置 keys

打开命令行终端，输入如下命令：

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key 421C365BD9FF1F717815A3895523BAEED01FA116
```

(4) 安装 ros-kinetic-desktop-full 完整版

打开命令行终端，分别输入如下两条命令：

```
sudo apt-get update

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

小技巧，如果安装过程提示“下载错误”，请耐心重试上面的两条命令，这个错误多半是由于网络故障造成的。

(5) 初始化 rosdep

打开命令行终端，分别输入如下两条命令：

```
sudo rosdep init  
  
rosdep update
```

(6) 配置环境变量

打开命令行终端，分别输入如下两条命令：

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc  
  
source ~/.bashrc
```

(7) 安装 rosinstall

打开命令行终端，输入如下命令：

```
sudo apt install python-rosinstall python-rosinstall-generator  
python-wstool build-essential
```

(8) 测试 ros 安装成功与否

打开命令行终端，输入如下命令：

```
roscore
```

如果此时出现以下内容

```
setting /run_id to 4cb2a932-06c0-11e9-9ff2-000c2985f3ab  
process[rosout-1]: started with pid [38686]  
started core service [/rosout]
```

那么恭喜你，ROS 已经成功的安装上了！！！！

4. 如何编写 ROS 的第一个程序 `hello_world`

既然 ROS 已经成功安装好了，大家一定很想亲自动手编一个 ROS 的小程序试试手，没错，这一节就隆重请出程序界的起手式例程 `hello_world`。

通过起手式例程 `hello_world`，可以学到工作空间的创建、功能包的创建、功能包的源代码编写、功能包的编译配置、功能包的编译、功能包的启动运行等知识。

(1) 工作空间的创建

打开命令行终端，分别输入如下命令：


```
#先切回主目录
cd ~/

#新建工作空间文件夹
mkdir catkin_ws

#在 catkin_ws 目录下新建 src 文件夹
cd catkin_ws
mkdir src

#初始化 src 目录，生成的 CMakeLists.txt 为功能包编译配置
cd src
catkin_init_workspace

#切回 catkin_ws 目录，对该工作空间执行一次编译
cd ~/catkin_ws
catkin_make

#环境变量配置，使新建的 catkin_ws 工作空间可用
source devel/setup.bash
```

这时，便已经创建好了一个 ROS 的工作空间了，接下来就是在 `catkin_ws` 工作空间下的 `src` 目录下新建功能包并进行功能包程序编写了，如果了解工作空间的详细目录结构，请参考本章节“2.1.从文件系统级理解 ROS 架构”这有对 ROS 的文件组织结构进行详解。

(2) 功能包的创建

继续在命令行终端中，输入如下命令：

```
#在 catkin_ws/src/下创建取名为 hello_world 的功能包，
#ROS 功能包命名规范：只允许使用小写字母、数字和下划线，
#且首字符必须为一个小写字母。
cd ~/catkin_ws/src/
catkin_create_pkg hello_world
```

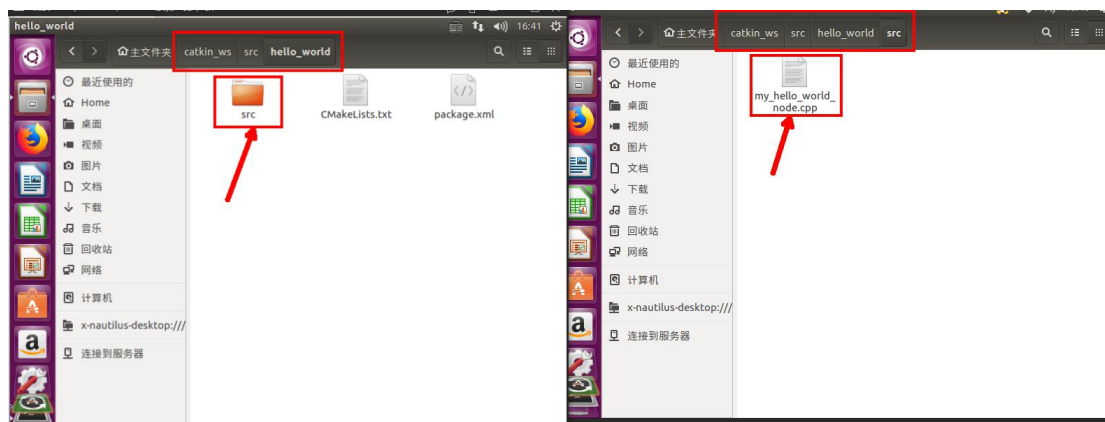
这时候在 `~/catkin_ws/src/` 目录下能看到一个叫 `hello_world` 的文件夹，文件夹名称就是功能包的名称以及功能包的唯一标识符，这个说明功能包创建成功了。

(3) 功能包的源代码编写

由于我们只是要编写一个能打印“hello world”的程序，所以就很简单了，这里先以 `c++` 代码作为示范，后面会讲解 `python` 的。一些在线教程建议在你的功能包目录中创建 `src` 目录用来存放 `c++` 源文件，这个附加的组织结构是很有益处的，特别是对含有很多种类型文件的大型功能包，不过不是严格必要的。出于编程规范，我这里也会采用这样的建议把 `c++` 源文

件放在功能包中的 `src` 目录下。

所以，首先在 `hello_world` 目录下新建 `src` 目录，再在新建的 `src` 目录下新建一个 `my_hello_world_node.cpp` 文件。这里的新建目录和文件的方法可以在图形界面下直接操作会更方便一点，如图 11。



（图 11）在功能包中新建文件夹及文件
用文本编辑器 `gedit` 打开 `my_hello_world_node.cpp` 文件，并输入如下内容。

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello_node");
    ros::NodeHandle nh;
    ROS_INFO_STREAM("hello world!!!");
}
```

这里对代码做一个解析：

`#include "ros/ros.h"`

这一句是包含头文件 `ros/ros.h`，这是 ROS 提供的 C++ 客户端库，是必须包含的头文件，在后面的编译配置中要添加相应的依赖库 `roscpp`。

`ros::init(argc, argv, "hello_node");`

这一句是初始化 `ros` 节点并指明节点的名称，这里给节点取名为 `hello_node`，一旦程序运行后就可以在 `ros` 的计算图中被注册为 `hello_node` 名称标识的节点。

`ros::NodeHandle nh;`

这一句是声明一个 `ros` 节点的句柄，初始化 `ros` 节点必须的。

`ROS_INFO_STREAM("hello world!!!");`

这一句是调用了 `roscpp` 库提供的方法 `ROS_INFO_STREAM` 来打印信息，这里打印字符串 `"hello world!!!"`。

（4）功能包的编译配置

声明依赖库：

对于我们的 my_hello_world_node.cpp 程序来说，我们包含了<ros/ros.h>这个库，因此我们需要添加名为 roscpp 的依赖库。

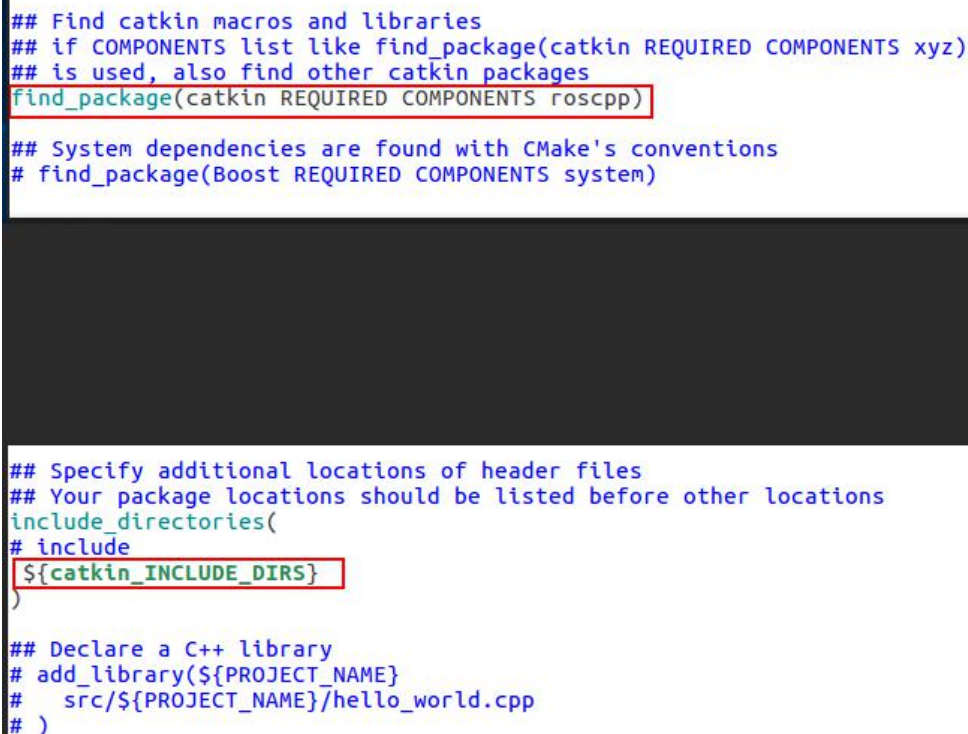
首先，用文本编辑器 gedit 打开功能包目录下的 CMakeLists.txt 文件，在 find_package(catkin REQUIRED ...) 字段中添加 roscpp，添加后的字段如下：

```
find_package(catkin REQUIRED COMPONENTS roscpp)
```

同时，找到 include_directories(...) 字段，去掉 \${catkin_INCLUDE_DIRS} 前面的注释，如下：

```
include_directories(  
  # include  
  ${catkin_INCLUDE_DIRS}  
)
```

添加好后的效果如图 12 所示。



```
## Find catkin macros and libraries  
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)  
## is used, also find other catkin packages  
find_package(catkin REQUIRED COMPONENTS roscpp)  
  
## System dependencies are found with CMake's conventions  
# find_package(Boost REQUIRED COMPONENTS system)  
  
  
  
  
  
  
## Specify additional locations of header files  
## Your package locations should be listed before other locations  
include_directories(  
  # include  
  $${catkin_INCLUDE_DIRS}  
)  
  
## Declare a C++ library  
# add_library(${PROJECT_NAME}  
#   src/${PROJECT_NAME}/hello_world.cpp  
# )
```

(图 12) 在 CMakeLists.txt 中添加 roscpp 依赖库

然后，用文本编辑器 gedit 打开功能包目录下的 package.xml 文件，找到这样一句话 <buildtool_depend>catkin</buildtool_depend>，在这句话的下面添加如下内容：

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

添加好后的效果如图 13 所示。

```
<!-- <doc_depend>doxygen</doc_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_export_depend>roscpp</build_export_depend>
<exec_depend>roscpp</exec_depend>

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

(图 13) 在 package.xml 中添加 roscpp 依赖库

声明可执行文件:

就接下来,我们需要在 CMakeLists.txt 中添加两句,我的习惯在文件最后一行添加就好了,来声明我们需要创建的可执行文件。

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

第一行声明了我们想要的可执行文件的文件名,以及生成此可执行文件所需的源文件列表。

如果你有多个源文件,把它们列在此处,并用空格将其区分开。

第二行告诉 Cmake 当链接此可执行文件时需要链接哪些库(在上面的 find_package 中定义)。如果你的包中包括多个可执行文件,为每一个可执行文件复制和修改上述两行代码。

添加好后的效果如图 14 所示。

```
## Install C++ headers for the package
# install(DIRECTORY include/${PROJECT_NAME}/
#   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
#   FILES_MATCHING PATTERN "*.h"
#   PATTERN ".svn" EXCLUDE
# )

## Mark other files for installation (e.g. launch and bag files, etc.)
# install(FILES
#   # myfile1
#   # myfile2
#   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# )

#####
## Testing ##
#####

## Add gtest based cpp test target and link libraries
# catkin_add_gtest(${PROJECT_NAME}-test test/test_hello_world.cpp)
# if(TARGET ${PROJECT_NAME}-test)
#   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# endif()

## Add folders to be run by python nosetests
# catkin_add_nosetests(test)
```

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

(图 14) 在 CMakeLists.txt 声明可执行文件

(5) 功能包的编译

功能包的编译配置好后,就可以开始编译了,这里有两种编译方式,一种是编译工作空间内的所有功能包,另一种是编译工作空间内的指定功能包,两种编译方式各有用处,下面分别讲解。

第一种，编译工作空间内的所有功能包：

```
cd ~/catkin_ws/  
catkin_make
```

第二种，编译工作空间内的指定功能包：

其实就是加入参数 `-DCATKIN_WHITELIST_PACKAGES=""`，在双引号中填入需要编译的功能包名字，用空格分割。

```
cd ~/catkin_ws/  
catkin_make -DCATKIN_WHITELIST_PACKAGES="hello_world"
```

（6）功能包的启动运行

首先，需要用 `roscore` 命令来启动 ROS 节点管理器，ROS 节点管理器是所有节点运行的基础。打开命令行终端，输入命令：

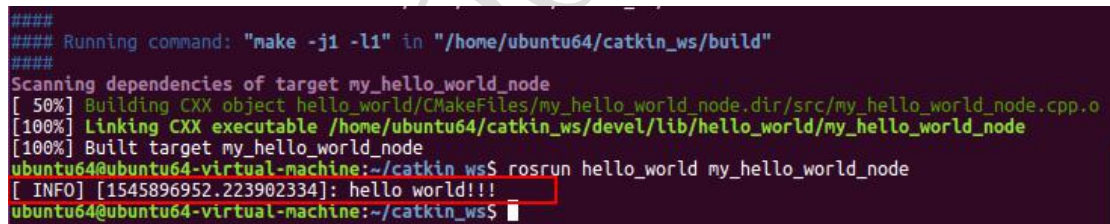
```
roscore
```

然后，用 `source devel/setup.bash` 激活 `catkin_ws` 工作空间，用 `roslaunch <package_name> <node_name>` 启动功能包中的节点。

再打开一个命令行终端，分别输入命令：

```
cd ~/catkin_ws/  
source devel/setup.bash  
roslaunch hello_world my_hello_world_node
```

看到有输出 `hello world!!!` 就说明程序已经正常执行了，按照我们的设计程序正常打印后会自动结束，如图 15。



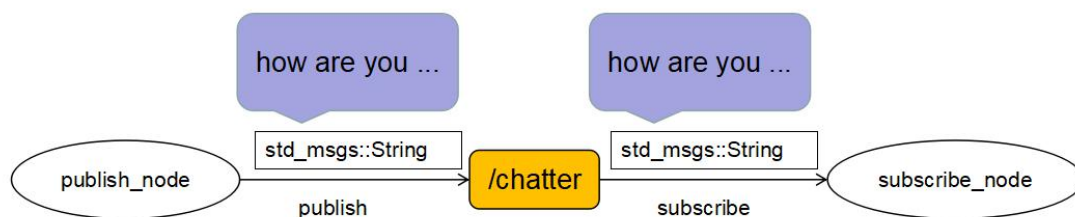
```
####  
#### Running command: "make -j1 -l1" in "/home/ubuntu64/catkin_ws/build"  
####  
Scanning dependencies of target my_hello_world_node  
[ 50%] Building CXX object hello_world/CMakeFiles/my_hello_world_node.dir/src/my_hello_world_node.cpp.o  
[100%] Linking CXX executable /home/ubuntu64/catkin_ws/devel/lib/hello_world/my_hello_world_node  
[100%] Built target my_hello_world_node  
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ roslaunch hello_world my_hello_world_node  
[ INFO] [1545896952.223902334]: hello world!!!  
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$
```

（图 15）hello_world 功能包中 my_hello_world_node 节点执行结果

到这里，恭喜你学会了 ROS 的第一个程序 `hello world!!!`

5. 编写简单的消息发布器和订阅器

通过上一节编写 ROS 的第一个程序 `hello_world`，我们对 ROS 的整个编程开发过程有了基本的了解，现在我们就来编写真正意义上的使用 ROS 进行节点间通信的程序。由于之前已经建好了 `catkin_ws` 这样一个工作空间，以后开发的功能包都将放在这里面，这里给新建的功能包取名为 `topic_example`，在这个功能包中分别编写两个节点程序 `publish_node.cpp` 和 `subscribe_node.cpp`，发布节点（`publish_node`）向话题（`chatter`）发布 `std_msgs::String` 类型的消息，订阅节点（`subscribe_node`）从话题（`chatter`）订阅 `std_msgs::String` 类型的消息，这里消息传递的具体内容是一句问候语 “`how are you ...`”，通信网络结构如图 16。



(图 16) 消息发布与订阅 ROS 通信网络结构图

(1) 功能包的创建

在 `catkin_ws/src/` 目录下新建功能包 `topic_example`，并在创建时显式的指明依赖 `roscpp` 和 `std_msgs`。打开命令行终端，输入命令：

```
cd ~/catkin_ws/src/

#创建功能包 topic_example 时，显式的指明依赖 roscpp 和 std_msgs，
#依赖会被默认写到功能包的 CMakeLists.txt 和 package.xml 中
catkin_create_pkg topic_example roscpp std_msgs
```

(2) 功能包的源代码编写

功能包中需要编写两个独立可执行的节点，一个节点用来发布消息，另一个节点用来订阅消息，所以需要在新建的功能包 `topic_example/src/` 目录下新建两个文件 `publish_node.cpp` 和 `subscribe_node.cpp`，并将下面的代码分别填入。

首先，介绍发布节点 `publish_node.cpp`，代码如下：

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

#include <sstream>

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

    ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);
    int count = 0;

    while (ros::ok())
    {
        std_msgs::String msg;

        std::stringstream ss;
        ss << "how are you " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());

        chatter_pub.publish(msg);

        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }

    return 0;
}
```

对发布节点代码进行解析。

`#include "ros/ros.h"`

这一句是包含头文件 `ros/ros.h`，这是 ROS 提供的 C++客户端库，是必须包含的头文件。

`#include "std_msgs/String.h"`

由于代码中需要使用 ROS 提供的标准消息类型 `std_msgs::String`，所以需要包含此头文件。

`ros::init(argc, argv, "publish_node");`

这一句是初始化 `ros` 节点并指明节点的名称，这里给节点取名为 `publish_node`，一旦程序运行后就可以在 `ros` 的计算图中被注册为 `publish_node` 名称标识的节点。



```
ros::NodeHandle nh;
```

这一句是声明一个 `ros` 节点的句柄，初始化 `ros` 节点必须的。

```
ros::Publisher chatter_pub = nh.advertise<std_msgs::String>("chatter", 1000);
```

这句话告诉 `ros` 节点管理器我们将会 `chatter` 这个话题上发布 `std_msgs::String` 类型的消息。这里的参数 `1000` 是表示发布序列的大小，如果消息发布的太快，缓冲区中的消息大于 `1000` 个的话就会开始丢弃先前发布的消息。

```
ros::Rate loop_rate(10);
```

这句话是用来指定自循环的频率，这里的参数 `10` 表示 `10Hz` 频率，需要配合该对象的 `sleep()` 方法来使用。

```
while (ros::ok()) {...}
```

`roscpp` 会默认安装以 `SIGINT` 句柄，这句话就是用来处理由 `ctrl+c` 键盘操作、该节点被另一同名节点踢出 `ROS` 网络、`ros::shutdown()` 被程序在某个地方调用、所有 `ros::NodeHandle` 句柄都被销毁等触发而使 `ros::ok()` 返回 `false` 值的情况。

```
std_msgs::String msg;
```

定义了一个 `std_msgs::String` 消息类型的对象，该对象有一个数据成员 `data` 用于存放我们即将发布的数据。要发布出去的数据将被填充到这个对象的 `data` 成员中。

```
chatter_pub.publish(msg);
```

利用定义好的发布者对象将消息数据发布出去，这一句执行后，`ROS` 网络中的其他节点便可以收到此消息中的数据。

```
ros::spinOnce();
```

这一句是让回调函数有机会被执行的声明，这个程序里面并没有回调函数，所以这一句可以不要，这里只是为了程序的完整规范性才放上来的。

```
loop_rate.sleep();
```

前面讲过，这一句是通过休眠来控制自循环的频率的。

接着，介绍订阅节点 `subscribe_node.cpp`，代码如下：

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

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]",msg->data.c_str());
}

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

    ros::Subscriber chatter_sub = nh.subscribe("chatter", 1000,chatterCallback);

    ros::spin();

    return 0;
}
```

对订阅节点代码进行解析。

之前解释过的类似的代码就不做过多的解释了，这里重点解释一下前面没遇到过的代码。

`void chatterCallback(const std_msgs::String::ConstPtr& msg)`

```
{
    ROS_INFO("I heard: [%s]",msg->data.c_str());
}
```

这是一个回调函数，当有消息到达 `chatter` 话题时会自动被调用一次，这个回调函数里面就是一句话，用来打印从话题中订阅的消息数据。

`ros::Subscriber chatter_sub = nh.subscribe("chatter", 1000,chatterCallback);`

这句话告诉 `ros` 节点管理器我们将会从 `chatter` 这个话题中订阅消息，当有消息到达时会自动调用这里指定的 `chatterCallback` 回调函数。这里的参数 `1000` 是表示订阅序列的大小，如果消息处理的速度不够快，缓冲区中的消息大于 `1000` 个的话就会开始丢弃先前接收的消息。

`ros::spin();`

这一句话让程序进入自循环的挂起状态，从而让程序以最好的效率接收消息并调用回调函数。如果没有消息到达，这句话不会占用很多 `CPU` 资源，所以这句话可以放心使用。一旦 `ros::ok()` 被触发而返回 `false`，`ros::spin()` 的挂起状态将停止并自动跳出。简单点说，程序执行到这一句，就在这里不断自循环，与此同时检查是否有消息到达并决定是否调用回调函数。

（3）功能包的编译配置及编译

创建功能包 `topic_example` 时，显式的指明依赖 `roscpp` 和 `std_msgs`，依赖会被默认写到功能包的 `CMakeLists.txt` 和 `package.xml` 中，所以只需要在 `CMakeLists.txt` 文件的末尾行加入以下几句用于声明可执行文件就可以了：

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

接下来，就可以用下面的命令对功能包进行编译了：

```
cd ~/catkin_ws/
catkin_make -DCATKIN_WHITELIST_PACKAGES="topic_example"
```

（4）功能包的启动运行

首先，需要用 `roscore` 命令来启动 ROS 节点管理器，ROS 节点管理器是所有节点运行的基础。打开命令行终端，输入命令：

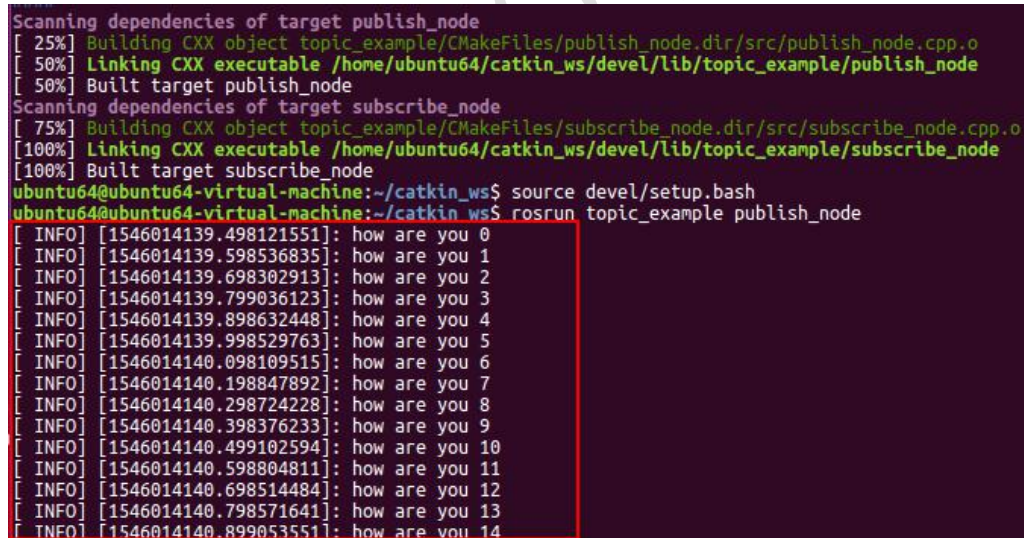
```
roscore
```

然后，用 `source devel/setup.bash` 激活 `catkin_ws` 工作空间，用 `roslaunch <package_name> <node_name>` 启动功能包中的发布节点。

再打开一个命令行终端，分别输入命令：

```
cd ~/catkin_ws/
source devel/setup.bash
roslaunch topic_example publish_node
```

看到有输出 “how are you ...”，就说明发布节点已经正常启动并开始不断向 `chatter` 话题发布消息数据，如图 17。



```
Scanning dependencies of target publish_node
[ 25%] Building CXX object topic_example/CMakeFiles/publish_node.dir/src/publish_node.cpp.o
[ 50%] Linking CXX executable /home/ubuntu64/catkin_ws/devel/lib/topic_example/publish_node
[ 50%] Built target publish_node
Scanning dependencies of target subscribe_node
[ 75%] Building CXX object topic_example/CMakeFiles/subscribe_node.dir/src/subscribe_node.cpp.o
[100%] Linking CXX executable /home/ubuntu64/catkin_ws/devel/lib/topic_example/subscribe_node
[100%] Built target subscribe_node
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ source devel/setup.bash
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ roslaunch topic_example publish_node
[ INFO] [1546014139.498121551]: how are you 0
[ INFO] [1546014139.598536835]: how are you 1
[ INFO] [1546014139.698302913]: how are you 2
[ INFO] [1546014139.799036123]: how are you 3
[ INFO] [1546014139.898632448]: how are you 4
[ INFO] [1546014139.998529763]: how are you 5
[ INFO] [1546014140.098109515]: how are you 6
[ INFO] [1546014140.198847892]: how are you 7
[ INFO] [1546014140.298724228]: how are you 8
[ INFO] [1546014140.398376233]: how are you 9
[ INFO] [1546014140.499102594]: how are you 10
[ INFO] [1546014140.598804811]: how are you 11
[ INFO] [1546014140.698514484]: how are you 12
[ INFO] [1546014140.798571641]: how are you 13
[ INFO] [1546014140.899053551]: how are you 14
```

（图 17）发布节点已经正常启动

最后，用 `source devel/setup.bash` 激活 `catkin_ws` 工作空间，用 `roslaunch <package_name> <node_name>` 启动功能包中的订阅节点。

再打开一个命令行终端，分别输入命令：

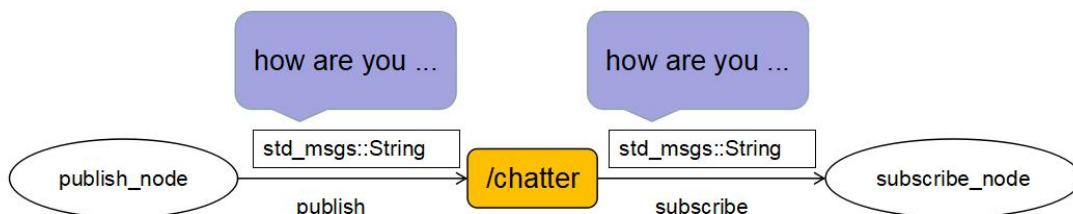
```
cd ~/catkin_ws/
source devel/setup.bash
roslaunch topic_example subscribe_node
```

看到有输出 “I heard:[how are you ...]”，就说明订阅节点已经正常启动并开始不断从 chatter 话题接收消息数据，如图 18。

```
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ rosrn topic_example subscribe_node
[INFO] [1546014659.912035522]: I heard: [how are you 5187]
[INFO] [1546014660.010475044]: I heard: [how are you 5188]
[INFO] [1546014660.112755375]: I heard: [how are you 5189]
[INFO] [1546014660.210772452]: I heard: [how are you 5190]
[INFO] [1546014660.310034127]: I heard: [how are you 5191]
[INFO] [1546014660.410705533]: I heard: [how are you 5192]
[INFO] [1546014660.510671755]: I heard: [how are you 5193]
[INFO] [1546014660.610042155]: I heard: [how are you 5194]
[INFO] [1546014660.710673034]: I heard: [how are you 5195]
[INFO] [1546014660.810156943]: I heard: [how are you 5196]
[INFO] [1546014660.911099887]: I heard: [how are you 5197]
[INFO] [1546014661.010595801]: I heard: [how are you 5198]
[INFO] [1546014661.110310353]: I heard: [how are you 5199]
[INFO] [1546014661.210797459]: I heard: [how are you 5200]
[INFO] [1546014661.310072920]: I heard: [how are you 5201]
[INFO] [1546014661.411726254]: I heard: [how are you 5202]
[INFO] [1546014661.510606708]: I heard: [how are you 5203]
[INFO] [1546014661.611106597]: I heard: [how are you 5204]
[INFO] [1546014661.711170961]: I heard: [how are you 5205]
```

(图 18) 订阅节点已经正常启动

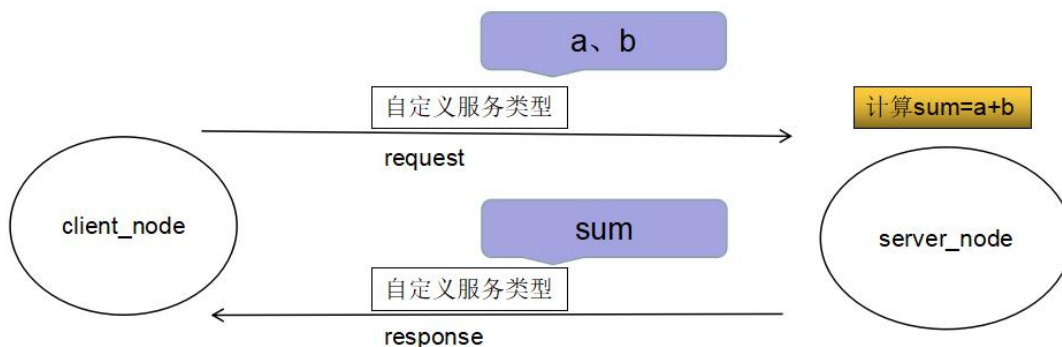
到这里，我们编写的消息发布器和订阅器就大功告成了，为了加深对整个程序工作流程的理解，我再把消息发布与订阅的 ROS 通信网络结构图拿出来，加深一下理解。



(图 19) 消息发布与订阅 ROS 通信网络结构图

6. 编写简单的 service 和 client

上一节介绍了两个 ros 节点通过发布与订阅消息的通信方式，现在就介绍 ros 节点间通信的另外一种方式服务。我们将学到：如何自定义服务类型、server 端节点编写、client 端节点编写等。我就以实现两个整数求和为例，client 端节点向 server 端节点发送 a、b 的请求，server 端节点返回响应 $sum=a+b$ 给 client 端节点，通信网络结构如图 20。



(图 20) 服务请求与响应 ROS 通信网络结构图

(1) 功能包的创建

在 catkin_ws/src/目录下新建功能包 service_example，并在创建时显式的指明依赖 roscpp 和 std_msgs，依赖 std_msgs 将作为基本数据类型用于定义我们的服务类型。打开命令行终端，输入命令：

```
cd ~/catkin_ws/src/
```

```
#创建功能包 service_example 时，显式的指明依赖 roscpp 和 std_msgs，  
#依赖会被默认写到功能包的 CMakeLists.txt 和 package.xml 中  
catkin_create_pkg service_example roscpp std_msgs
```

(2) 在功能包中创建自定义服务类型

通过前面的学习，我们知道服务通信过程中服务的数据类型需要用户自己定义，与消息不同，节点并不提供标准服务类型。服务类型的定义文件都是以*.srv 为扩展名，并且被放在功能包的 srv/文件夹下。

服务类型定义：

首先，在功能包 service_example 目录下新建 srv 目录，然后在 service_example/srv/目录中创建 AddTwoInts.srv 文件，文件内容如下：

```
int64 a  
int64 b  
---  
int64 sum
```

服务类型编译配置：

定义好我们的服务类型后，要想让该服务类型能在 c++、python 等代码中被使用，必须要做相应的编译与运行配置。编译依赖 message_generation，运行依赖 message_runtime。

打开功能包中的 CMakeLists.txt 文件，找到下面这段代码：

```
find_package(catkin REQUIRED COMPONENTS  
  roscpp  
  std_msgs  
)
```

将 message_generation 添加进去，添加后的代码如下：

```
find_package(catkin REQUIRED COMPONENTS  
  roscpp  
  std_msgs  
  message_generation  
)
```

继续，找到这段代码：

```
# add_service_files(  
#   FILES  
#   Service1.srv  
#   Service2.srv  
# )
```

去掉这段代码的#注释，将自己的服务类型定义文件 AddTwoInts.srv 填入，修改好后的代码如下：


```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

继续，找到这段代码：

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

去掉这段代码的#注释，`generate_messages` 的作用是自动创建我们自定义的消息类型*.msg 与服务类型*.srv 相对应的*.h，由于我们定义的服务类型使用了 `std_msgs` 中的 `int64` 基本类型，所以必须向 `generate_messages` 指明该依赖，修改好后的代码如下：

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

然后打开功能包中的 `package.xml` 文件，填入下面三句依赖：

```
<build_depend>message_generation</build_depend>
<build_export_depend>message_generation</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

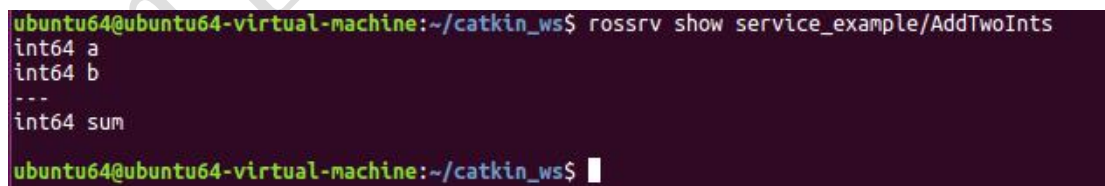
检查 ROS 是否识别新建的服务类型：

我们通过<功能包名/服务类型名>找到该服务，打开命令行终端，输入命令：

```
source ~/catkin_ws/devel/setup.bash

rossrv show service_example/AddTwoInts
```

看到下面的输出，如图 21，就说明新建服务类型能被 ROS 识别，新建服务类型成功了。



```
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ rossrv show service_example/AddTwoInts
int64 a
int64 b
---
int64 sum
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$
```

（图 21）新建服务类型能被 ROS 识别

（3）功能包的源代码编写

功能包中需要编写两个独立可执行的节点，一个节点用来作为 `client` 端发起请求，另一个节点用来作为 `server` 端响应请求，所以需要在新建的功能包 `service_example/src/` 目录下新建两个文件 `server_node.cpp` 和 `client_node.cpp`，并将下面的代码分别填入。

首先，介绍 `server` 节点 `server_node.cpp`，代码如下：

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

bool add_execute(service_example::AddTwoInts::Request &req,
                 service_example::AddTwoInts::Response &res)
{
    res.sum = req.a + req.b;
    ROS_INFO("recieve request: a=%ld,b=%ld",(long int)req.a,(long int)req.b);
    ROS_INFO("send response: sum=%ld",(long int)res.sum);
    return true;
}

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

    ros::ServiceServer service = nh.advertiseService("add_two_ints",add_execute);
    ROS_INFO("service is ready!!!");
    ros::spin();

    return 0;
}
```

对 server 节点代码进行解析。

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

包含头文件 `ros/ros.h`，这是 ROS 提供的 C++ 客户端库，是必须包含的头文件，就不多说了。
`service_example/AddTwoInts.h` 是由编译系统自动根据我们的功能包和在功能包下创建的 `*.srv` 文件生成的对应的头文件，包含这个头文件，程序中就可以使用我们自定义服务的数据类型了。

```
bool add_execute(...)
```

这个函数实现两个 `int64` 整数求和的服务，两个 `int64` 值从 `request` 获取，返回求和结果装入 `response` 里，`request` 与 `response` 的具体数据类型都在前面创建的 `*.srv` 文件中被定义，这个函数返回值为 `bool` 型。

```
ros::init(argc,argv,"server_node");
ros::NodeHandle nh;
```

初始化 `ros` 节点并指明节点的名称，声明一个 `ros` 节点的句柄，就不多说了。

```
ros::ServiceServer service = nh.advertiseService("add_two_ints",add_execute);
```

这一句是创建服务，并将服务加入到 ROS 网络中，并且这个服务在 ROS 网络中以名称 `add_two_ints` 唯一标识，以便于其他节点通过服务名称进行请求。

ros::spin();

这一句话让程序进入自循环的挂起状态，从而让程序以最好的效率接收客户端的请求并调用回调函数，就不多说了。

接着，介绍 **client** 节点 **client_node.cpp**，代码如下：

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

#include <iostream>

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

    ros::ServiceClient client =
        nh.serviceClient<service_example::AddTwoInts>("add_two_ints");
    service_example::AddTwoInts srv;

    while(ros::ok())
    {
        long int a_in, b_in;
        std::cout<<"please input a and b:";
        std::cin>>a_in>>b_in;

        srv.request.a = a_in;
        srv.request.b = b_in;
        if(client.call(srv))
        {
            ROS_INFO("sum=%ld", (long int)srv.response.sum);
        }
        else
        {
            ROS_INFO("failed to call service add_two_ints");
        }
    }
    return 0;
}
```

对 **client** 节点代码进行解析。

之前解释过的类似的代码就不做过多的解释了，这里重点解释一下前面没遇到过的代码。

ros::ServiceClient client =

nh.serviceClient<service_example::AddTwoInts>("add_two_ints");

这一句创建 **client** 对象，用来向 ROS 网络中名称叫 **add_two_ints** 的 **service** 发起请求。

```
service_example::AddTwoInts srv;
```

定义了一个 `service_example::AddTwoInts` 服务类型的对象，该对象中的成员正是我们在 `*.srv` 文件中定义的 `a`、`b`、`sum`，我们将待请求的数据填充到数据成员 `a`、`b`，请求成功后返回结果会被自动填充到数据成员 `sum` 中。

```
if(client.call(srv)){...}
```

这一句便是通过 `client` 的方法 `call` 来向 `service` 发起请求，请求传入的参数 `srv` 在上面已经介绍过了。

（4）功能包的编译配置及编译

创建功能包 `service_example` 时，显式的指明依赖 `roscpp` 和 `std_msgs`，依赖会被默认写到功能包的 `CMakeLists.txt` 和 `package.xml` 中，并且在功能包中创建 `*.srv` 服务类型时已经对服务的编译与运行做了相关配置，所以只需要在 `CMakeLists.txt` 文件的末尾行加入以下几句用于声明可执行文件就可以了：

```
add_executable(server_node src/server_node.cpp)
target_link_libraries(server_node ${catkin_LIBRARIES})
add_dependencies(server_node service_example_gencpp)

add_executable(client_node src/client_node.cpp)
target_link_libraries(client_node ${catkin_LIBRARIES})
add_dependencies(client_node service_example_gencpp)
```

这里面的 `add_executable` 用于声明可执行文件。`target_link_libraries` 用于声明可执行文件创建时需要链接的库。`add_dependencies` 用于声明可执行文件的依赖项，由于我们自定义了 `*.srv`，`service_example_gencpp` 的作用是让编译系统自动根据我们的功能包和在功能包下创建的 `*.srv` 文件生成的对应的头文件和库文件，`service_example_gencpp` 这个名称是由功能包名称 `service_example` 加上 `_gencpp` 后缀而来的，后缀很好理解：生成 `c++` 文件就是 `_gencpp`，生成 `python` 文件就是 `_genpy`。

接下来，就可以用下面的命令对功能包进行编译了：

```
cd ~/catkin_ws/
catkin_make -DCATKIN_WHITELIST_PACKAGES="service_example"
```

（5）功能包的启动运行

首先，需要用 `roscore` 命令来启动 ROS 节点管理器，ROS 节点管理器是所有节点运行的基础。打开命令行终端，输入命令：

```
roscore
```

然后，用 `source devel/setup.bash` 激活 `catkin_ws` 工作空间，用 `roslun <package_name> <node_name>` 启动功能包中的 `server` 节点。

再打开一个命令行终端，分别输入命令：

```
cd ~/catkin_ws/
source devel/setup.bash
roslun service_example server_node
```

看到有输出“service is ready!!!”，就说明 server 节点已经正常启动并开始等待 client 节点向自己发起请求了，如图 22。

```
ubuntu64@ubuntu64-virtual-machine:~/catkin_ws$ rosrn service_example server_node
[ INFO] [1546276940.824395885]: service is ready!!!
```

(图 22) server 节点已经正常启动

最后，用 `source devel/setup.bash` 激活 catkin_ws 工作空间，用 `rosrn <package_name> <node_name>` 启动功能包中的 client 节点。

再打开一个命令行终端，分别输入命令：

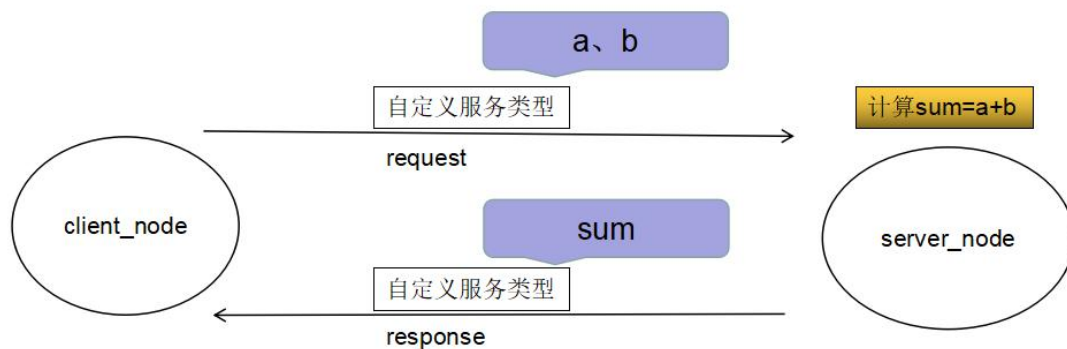
```
cd ~/catkin_ws/
source devel/setup.bash
rosrn service_example client_node
```

看到有输出提示信息“please input a and b:”后，键盘键入两个整数，以空格分割，输入完毕后回车。如果看到输出信息“sum=xxx”，就说明 client 节点向 server 端发起的请求得到了响应，打印出来的 sum 就是响应结果，这样就完成了一次服务请求的通信过程，如图 23。

```
ubuntu64@ubuntu64-virtual-machine:~$ rosrn service_example client_node
please input a and b:100 200
[ INFO] [1546277464.809197537]: sum=300
please input a and b:█
```

(图 23) client 节点已经正常启动

到这里，我们编写的 server 和 client 就大功告成了，为了加深对整个程序工作流程的理解，我再把 server 与 client 的 ROS 通信网络结构图拿出来，加深一下理解。



(图 24) 服务请求与响应 ROS 通信网络结构图

7.理解 tf 的原理

(1) 机器人中的坐标系

一个机器人系统中通常会有多个三维参考坐标系，而且这些坐标系之间的相对关系随时间推移会变化。这里举一个实际的机器人应用场景例子，来说明这种关系和变化：

全局世界坐标系：通常为激光 slam 构建出来的栅格地图的坐标系 map。

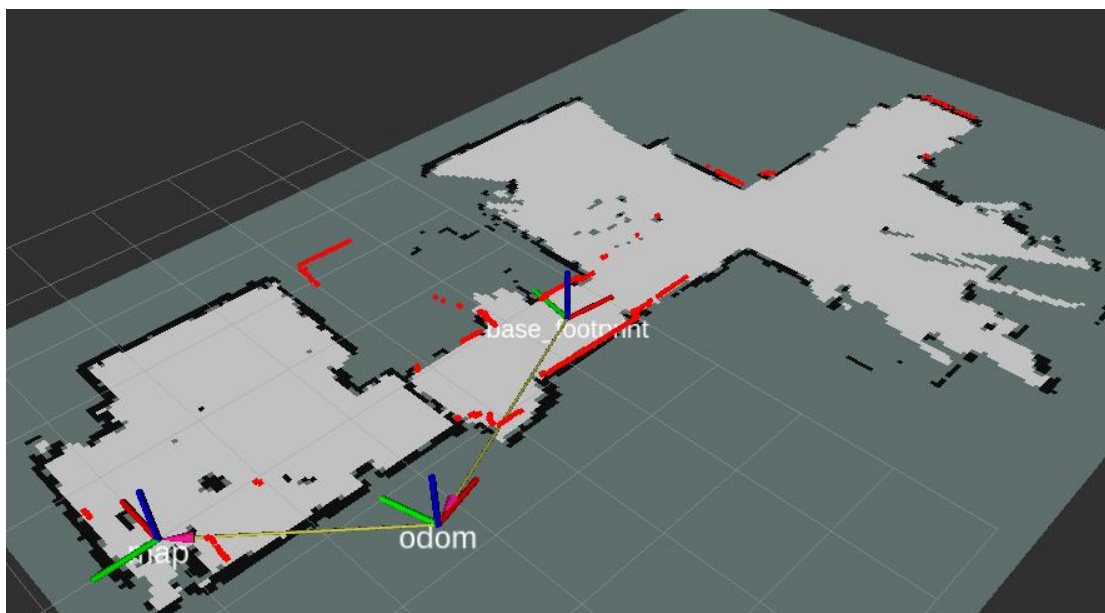
机器人底盘坐标系：通常为机器人底盘的坐标系 base_footprint。

机器人上各部件自己的坐标系：比如激光雷达、imu 等传感器自己的坐标系 base_laser_link、imu_link。

这些坐标系之间的关系有些是静态的、有些是动态的。比如当机器人底盘移动的过程中，机器人底盘与世界的相对关系 map->base_footprint 就会随之变化；而安装在机器人底盘上的

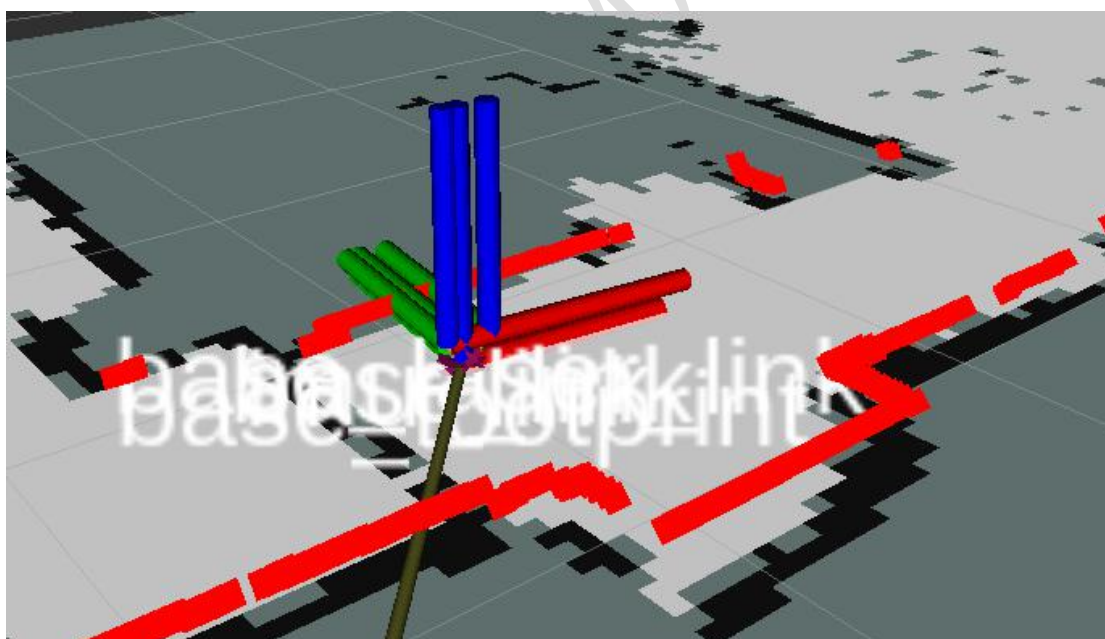
激光雷达、imu 这些传感器与机器人底盘的相对关系 $\text{base_footprint} \rightarrow \text{base_laser_link}$ 、 $\text{base_footprint} \rightarrow \text{imu_link}$ 就不会随之变化。其实，这个很好理解。

如图 25 中， $\text{map} \rightarrow \text{base_footprint}$ 会随着底盘的移动而变化，即动态坐标系关系。



(图 25) 动态坐标系关系

如图 26 中， $\text{base_footprint} \rightarrow \text{base_laser_link}$ 、 $\text{base_footprint} \rightarrow \text{imu_link}$ 不会随着底盘的移动而变化，即静态坐标系关系。



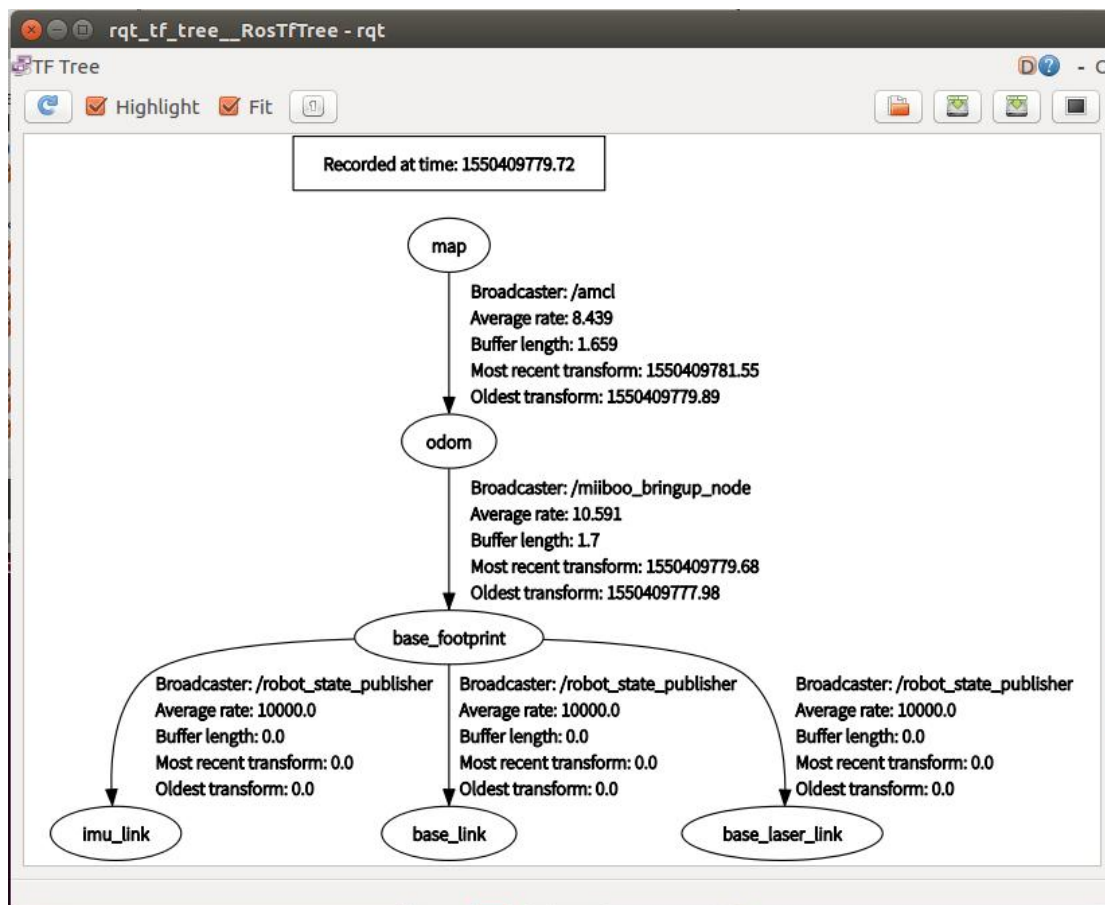
(图 26) 静态坐标系关系

(2) 机器人坐标关系工具 tf

由于坐标及坐标转换在机器人系统中非常重要，特别是机器人在环境地图中自主定位和导航、机械手臂对物体进行复杂的抓取任务，都需要精确的知道机器人各部件之间的相对位置及机器人在工作环境中的相对位置。因此 ROS 专门提供了 **tf** 这个工具用于简化这些工作。**tf** 可以让用户随时跟踪多个坐标系的关系，机器人各个坐标系之间的关系是通过一种树型数据结构来存储和维护的，即 **tf tree**。借助这个 **tf tree**，用户可以在任意时间将点、向量等数

据的坐标在两个坐标系中完成坐标值变换。

如图 27，为一个自主导航机器人的 tf tree 结构图。圆圈中是坐标系的名称，箭头表示两个坐标系之间的关系，箭头上会显示该坐标关系的发布者、发布速率、时间戳等信息。



(图 27) 一个自主导航机器人的 tf tree 结构图

(3) 使用 tf

使用 tf 分为两个部分，广播 tf 变换、监听 tf 变换。

广播 tf 变换：

ROS 网络中的节点可以向系统广播坐标系之间的变换关系。比如负责机器人全局定位的 amcl 节点会广播 map->odom 的变换关系，负责机器人局部定位的轮式里程计算节点会广播 odom->base_footprint 的变换关系，机器人底盘上安装的传感器与底盘的变换关系可以通过 urdf 机器人模型进行广播（urdf 将在后面实际机器人中进行讲解）。每个节点的广播都可以直接将变换关系插入 tf tree，不需要进行同步。通过多个节点广播坐标变换的关系，便可以实现 tf tree 的动态维护。

关于广播 tf 变换的具体程序实现，请直接参考 ROS 官方教程 <http://wiki.ros.org/tf/Tutorials>

监听 tf 变换：

ROS 网络中的节点可以从系统监听坐标系之间的变换关系，并从中查询所需要的坐标变换。比如要知道机器人底盘当前在栅格地图坐标系下的什么地方，就可以通过监听 map->base_footprint 来实现，比如要知道机器人底盘坐标系上的某个坐标点在世界坐标系下的坐标是多少，就可以通过监听 map->base_footprint，并通过 map->base_footprint 这个变换查询出变换后的坐标点取值。

关于监听 tf 变换的具体程序实现，请直接参考 ROS 官方教程 <http://wiki.ros.org/tf/Tutorials>

8.理解 roslaunch 在大型项目中的作用

(1) roslaunch 的作用

在一个大型的机器人项目中，经常涉及到多个 **node** 协同工作，并且每个 **node** 都有很多可设置的 **parameter**。比如我们的机器人 **miiboo_nav** 导航项目，涉及到地图服务节点、定位算法节点、运动控制节点、底盘控制节点、激光雷达数据获取节点等众多节点，和几百个影响着这些 **node** 行为模式的 **parameter**。如果全部手动 **roslaunch** 逐个启动 **node** 并传入 **parameter**，工程的复杂程度将难以想象。所以这个时候就需要用 **roslaunch** 来解决问题，将需要启动的节点和需要设置的 **parameter** 全部写入一个 ***.launch** 文件，然后用 **roslaunch** 一次性的启动 ***.launch** 文件，这样所有的节点就轻而易举的启动了。**miiboo_nav** 导航项目的 **miiboo_nav.launch** 文件内容如图 28。

```
miiboo_nav.launch x
<launch>
  <!-- Map server -->
  <arg name="map_path" default="/home/ubuntu/map/carto_map.yaml" />
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_path)"/>

  <!-- Run AMCL -->
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find miiboo_nav)/config/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Run move base -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen" clear_params="true">
    <rosparam file="$(find miiboo_nav)/config/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find miiboo_nav)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find miiboo_nav)/config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/move_base_params.yaml" command="load" />
    <rosparam file="$(find miiboo_nav)/config/global_planner_params.yaml" command="load" />
  </node>
</launch>
```

(图 28) **miiboo_nav** 导航项目的 **miiboo_nav.launch** 文件内容

(2) launch 标签介绍

launch 文件采用 **xml** 文本标记语言进行编写，对比较常用的标签进行介绍。

<launch>标签:

这个是顶层标签，所有的描述标签都要写在 **<launch></launch>** 之间。

```
<launch>
...
</launch>
```

<node>标签:

这个是最常见的标签，每个 **node** 标签里包含了 **ROS** 图中节点的名称属性 **name**、该节点所在的包名 **pkg**、节点的类型 **type** (**type** 为可执行文件名称，如果节点用 **c++** 编写；**type** 为 ***.py**，如果节点用 **python** 编写)、调试属性 **output** (如果 **output=“screen”**，终端输出信息将被打印到当前控制台，而不是存入 **ROS** 日志文件)。

```
<node name="xx" pkg="xx" type="xx" output="xx">
...
</node>
```

<include>标签:

这个标签是用于导入另一个 ***.launch** 文件到当前文件。也就是说高层级的 **launch** 文件可以通过 **include** 的方法调用其它 **launch** 文件，这样可以使 **launch** 文件的组织方式更加模块化，便于移植与复用。

```
<include file="$(find pkg_name)/launch/xx.launch"/>
```

<remap>标签:

这个标签是用于将 **topic** 的名称进行重映射，**from** 中填入原来的 **topic** 名称，**to** 中填入新的 **topic** 名称。**<remap>** 标签根据放置在 **launch** 文件的层级不同，在相应的层级起作用。

```
<remap from="orig_topic_name" to="new_topic_name"/>
```

<param>标签:

这个标签用于在参数服务器中创建或设置一个指定名称的参数值。

```
<param name="param_name" type="xx" value="xx"/>
```

<rosparam>标签:

这个标签用于从 **yaml** 文件中一次性导入大量参数到参数服务器中。

```
<rosparam command="load" file="$(find pkg_name)/path_to_file.yaml"/>
```

<arg>标签:

这个标签用于在 **launch** 文件中定义用于存储的临时变量，该标签定义的变量只在当前 **launch** 文件中使用。推荐使用第一种方式赋值，这样可以方便从命令行中传入参数。

```
<arg name="xx" default="xx"/>  
或者  
<arg name="xx" value="xx"/>
```

<group>标签:

这个标签用于将 **node** 批量划分到某个命名空间。便于大项目中节点的批量管理。

```
<group ns="group_one">  
  < node ... />  
  < node ... />  
</group>  
  
<group ns="group_two">  
  < node ... />  
  < node ... />  
</group>
```

(3) launch 的使用方法

首先在相应功能包目录下新建一个 **launch** 文件夹。

然后在 **launch** 文件夹中新建 ***.launch** 文件，并按照上面的 **launch** 标签规则编写好 **launch** 文件的内容。

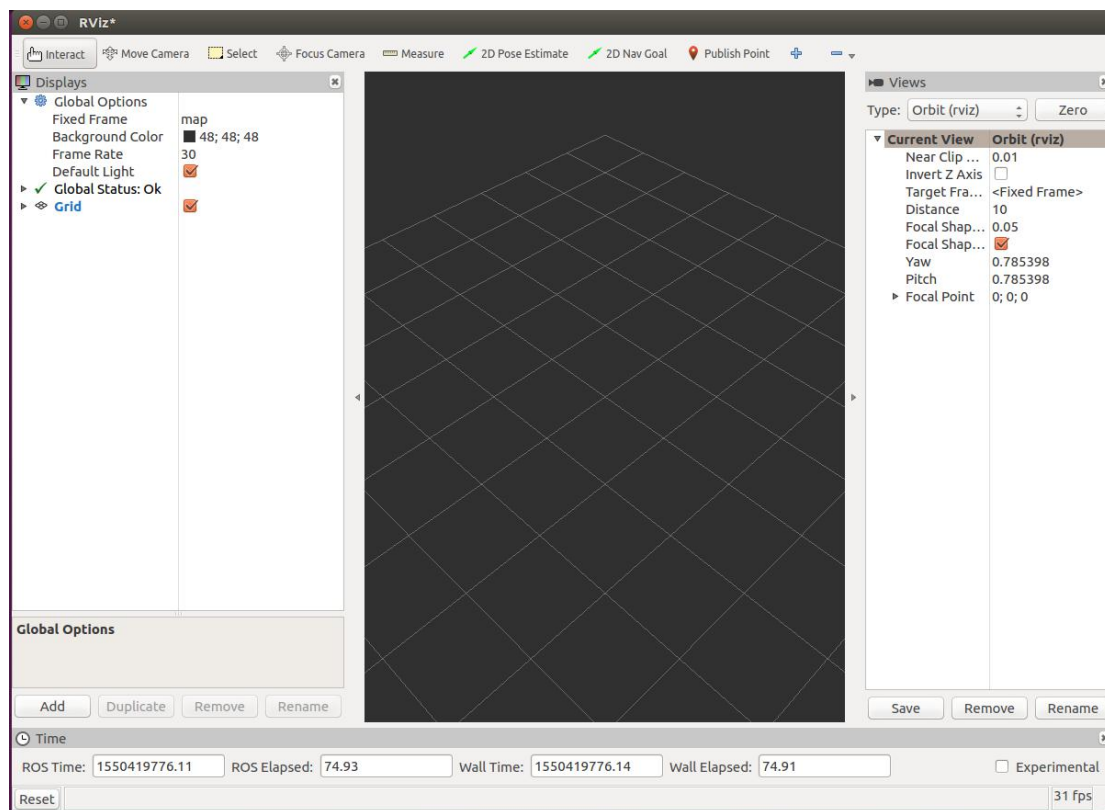
最后在终端中用 **roslaunch** 命令启动 **launch** 文件，命令如下：

```
cd ~/catkin_ws/  
source devel/setup.bash  
roslaunch <pkg_name> <file_name.launch>
```

特别说明，由于 **roslaunch** 命令会自动去启动 **roscore**，所以不需要像之前使用 **roslaunch** 那样特意

9. 熟练使用 rviz

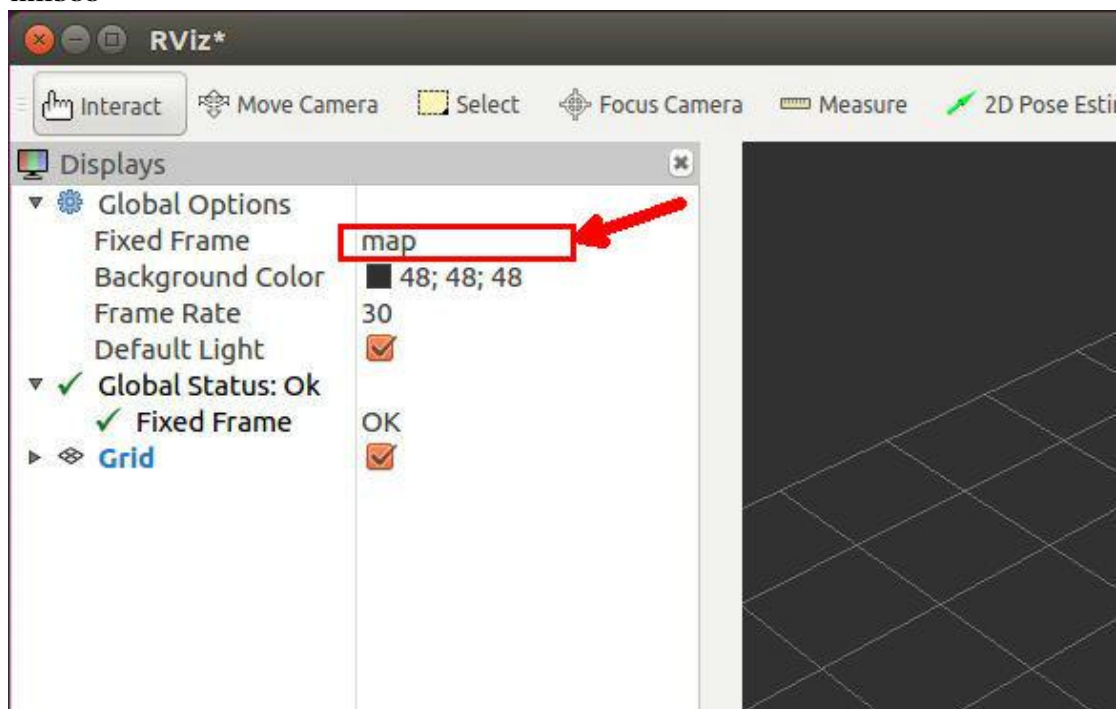
(1) rviz 整体界面



(图 29) rviz 整体界面

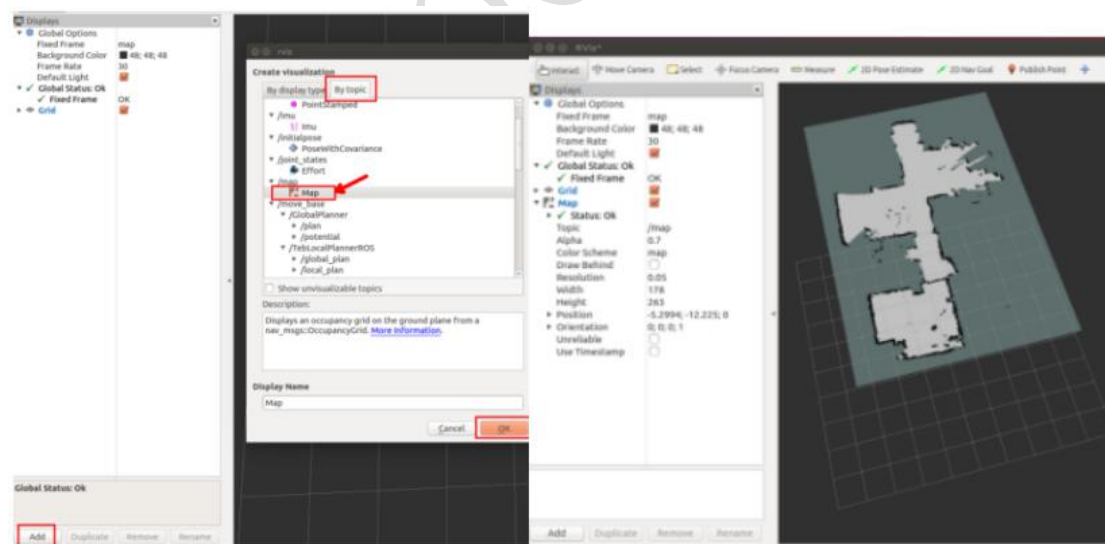
rviz 是 ROS 自带的图形化工具，可以很方便的让用户通过图形界面开发调试 ROS。操作界面也十分简洁，如图 29，界面主要分为上侧菜单区、左侧显示内容设置区、中间显示区、右侧显示视角设置区、下侧 ROS 状态区。

(2) 添加显示内容



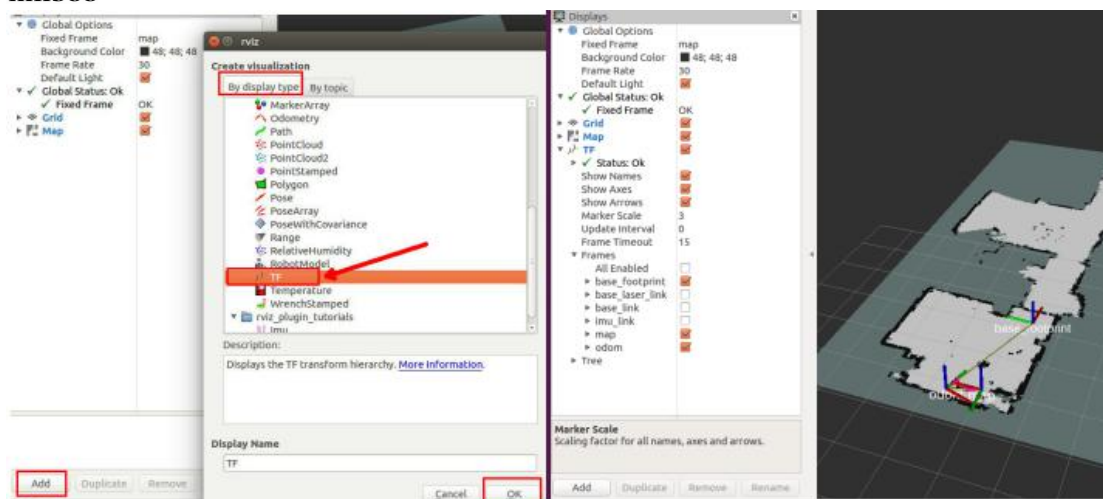
(图 30) 设置 Global Options

如图 30，启动 rviz 界面后，首先要对 Global Options 进行设置，Global Options 里面的参数是一些全局显示相关的参数。其中的 Fixed Frame 参数是全局显示区域依托的坐标系，我们知道机器人中有很多坐标系，坐标系之间有各自的转换关系，有些是静态关系，有些是动态关系，不同的 Fixed Frame 参数有不同的显示效果，在导航机器人应用中，一般将 Fixed Frame 参数设置为 map，也就是以 map 坐标系作为全局坐标系。值得注意的是，在机器人的 tf tree 里面必须要有 map 坐标系，否则该选项栏会包 error。至于 Global Options 里面的其他参数可以不用管，默认就行了。



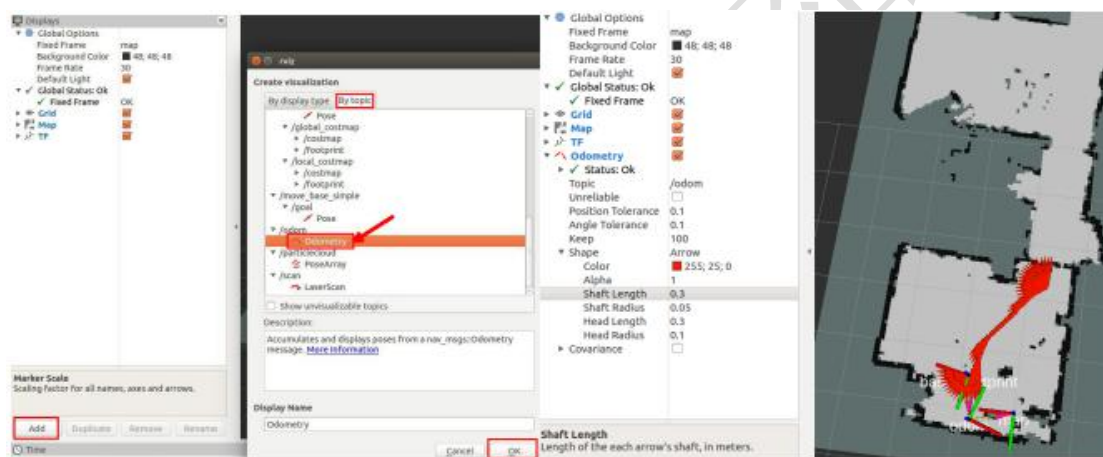
(图 31) 添加地图显示

如图 31，在机器人导航应用中，我们常常需要用 rviz 观察机器人建立的地图，在机器人发布了地图到主题的情况下，我们就可以通过 rviz 订阅地图相应主题（一般是/map 主题）来显示地图。订阅地图的/map 主题方法很简单，首先点击 rviz 界面左下角[add]按钮，然后在弹出的对话框中选择[By topic]，最后在列出的 topic 名字中找到我们要订阅的主题名字/map，最后点击[OK]就完成了对/map 主题的订阅。订阅成功后，会在 rviz 左侧栏中看到 Map 项，并且中间显示区正常显示出地图。



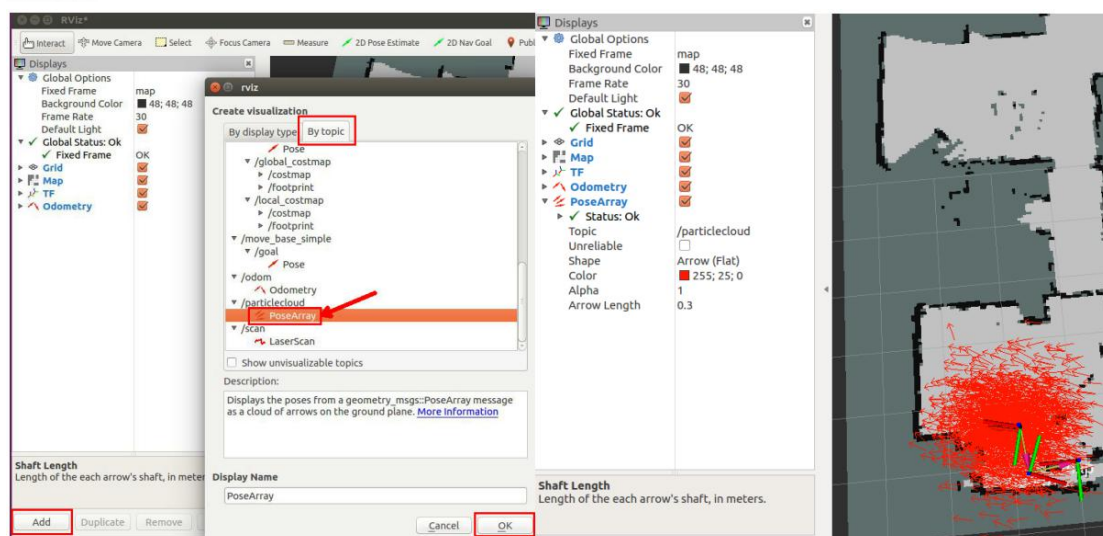
(图 32) 添加 tf 显示

如图 32，在机器人导航应用中，除了观察地图外，我们常常还需要观察机器人在地图中的位置以及各个坐标系的关系是否工作正常，这个时候就需要通过 rviz 来显示 tf。和上面添加显示主题的方法类似，这里添加 TF 这个类型主题就可以了。说明一下，添加主题可以按主题类型查找，也可以按主题名称查找。上面添加地图主题就是按主题名称查找的，这里添加 tf 主题是按主题类型查找的。



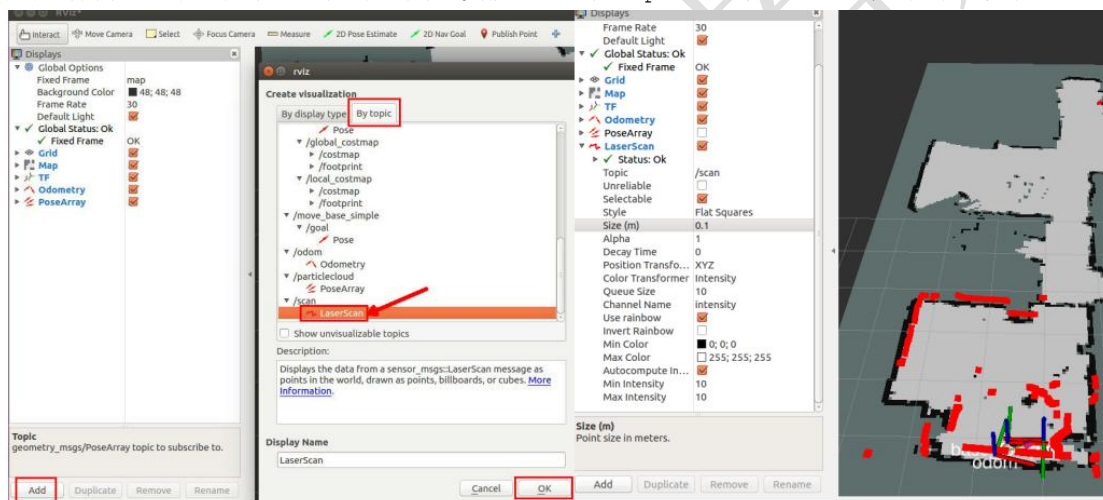
(图 33) 添加里程计显示

如图 33，我们可以通过 rviz 订阅里程计来观察机器人的运动轨迹（图中红色箭头连接起来的轨迹）。和上面添加显示主题的方法类似，这里添加 /odom 这个主题就可以了。这里特别说明一点，**我们需要去掉左侧栏中 Odometry 里面 Covariance 项后面后面的勾**，也就是在 Odometry 显示中不启用 Covariance 信息。Covariance 是描述里程计误差的协方差矩阵，如果启用 Covariance 来描述 Odometry 将导致显示效果很难看，所以建议去掉。



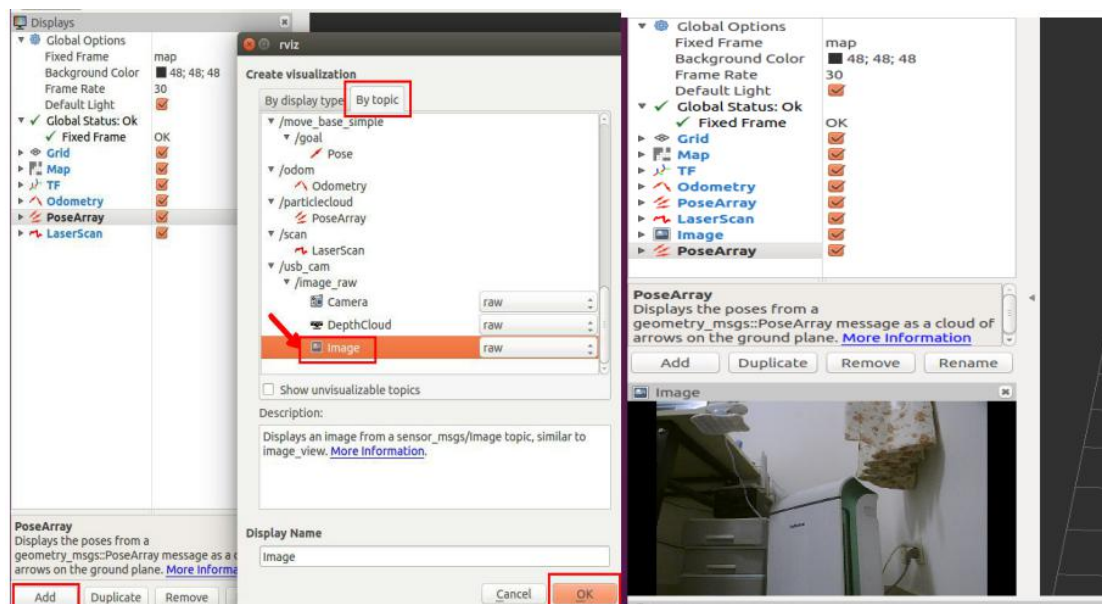
(图 34) 添加机器人位置粒子滤波点显示

如图 34，在机器人导航中，通常采用 AMCL 粒子滤波来实现机器人的全局定位。通过 rviz 可以显示全局定位的例子点。和上面添加显示主题的方法类似，这里添加 /particlecloud 这个主题就可以了。



(图 35) 添加激光雷达显示

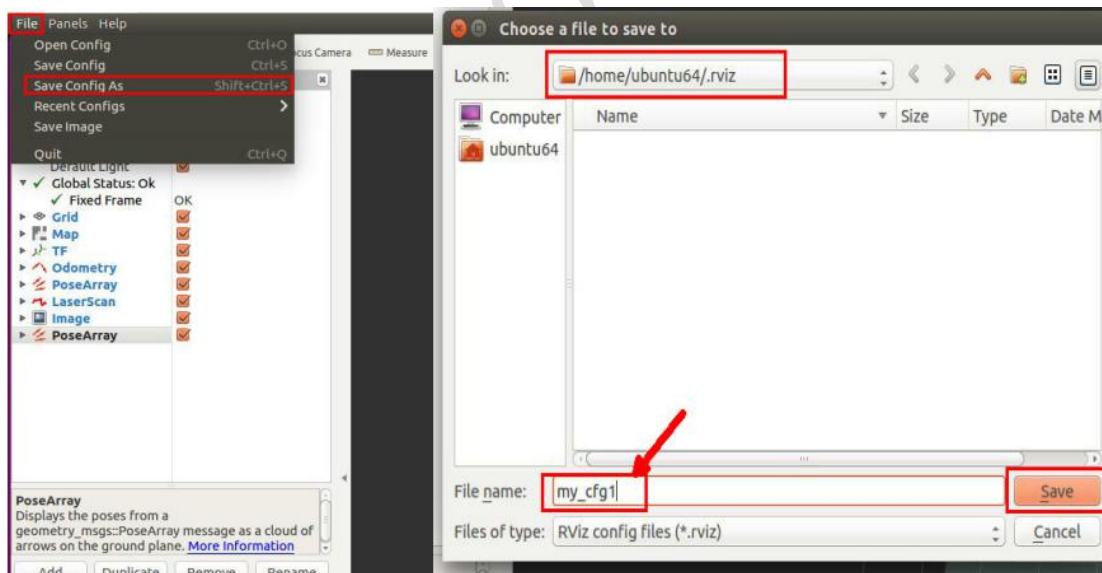
如图 35，机器人 SLAM 和导航中用到的核心传感器激光雷达数据，我们可以通过 rviz 显示激光雷达数据（图中红色点组成的轮廓）。和上面添加显示主题的方法类似，这里添加 /scan 这个主题就可以了。



(图 36) 添加摄像头显示

如图 36, rviz 还可以订阅摄像头发布的主题, 这样在 rviz 上就可以实现远程视频监控了。和上面添加显示主题的方法类似, 这里添加/usb_cam/image_raw 这个主题就可以了。通过上面的实例, 我们已经知道在 rviz 中订阅需要显示的主题了, 被订阅的主题会在 rviz 左侧栏中列出, 并且主题的显示与否是相互独立的, 可以通过勾选的方式决定是否显现该主题, 主题项下拉条目中有很多参数可以设置, 这些参数决定显示的风格等等, 可以根据需要进行设置。其他一些不常用的主题订阅实例没有给出, 有需要可以依葫芦画瓢在 rviz 中进行订阅显示就行了。

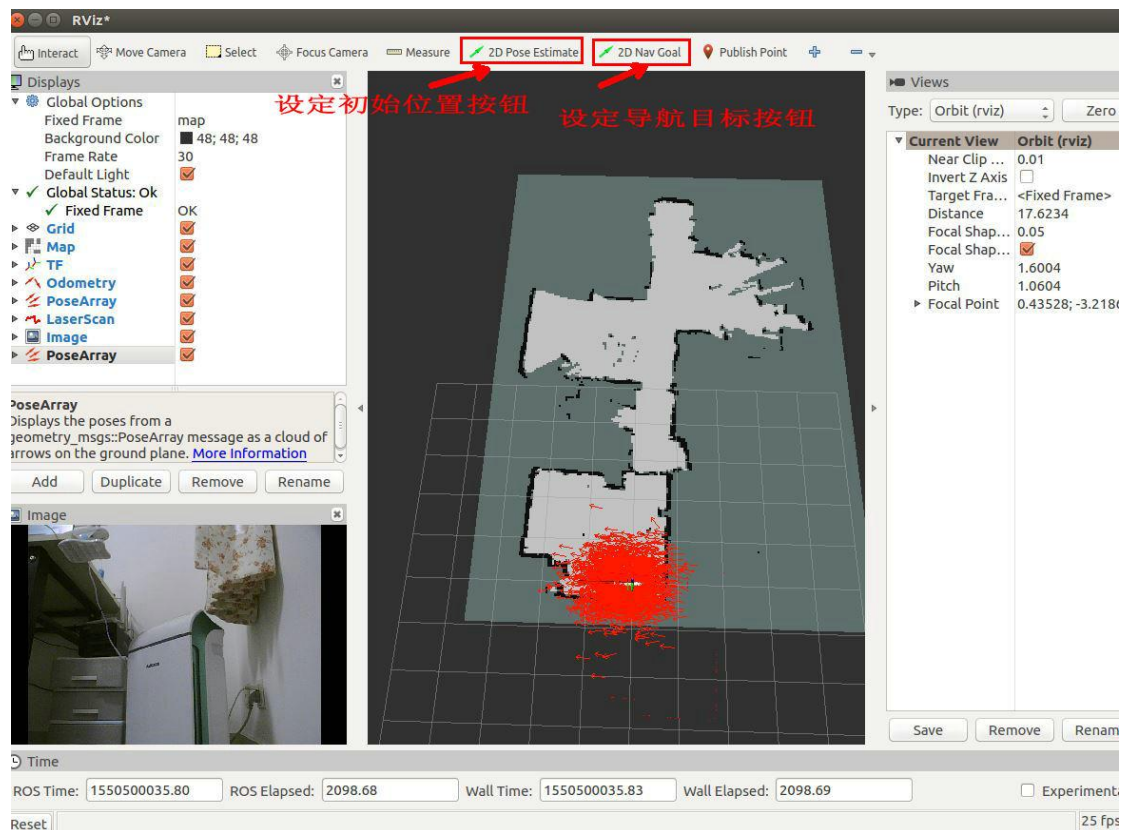
(3) 主界面中常用按钮



(图 37) rviz 显示配置保存

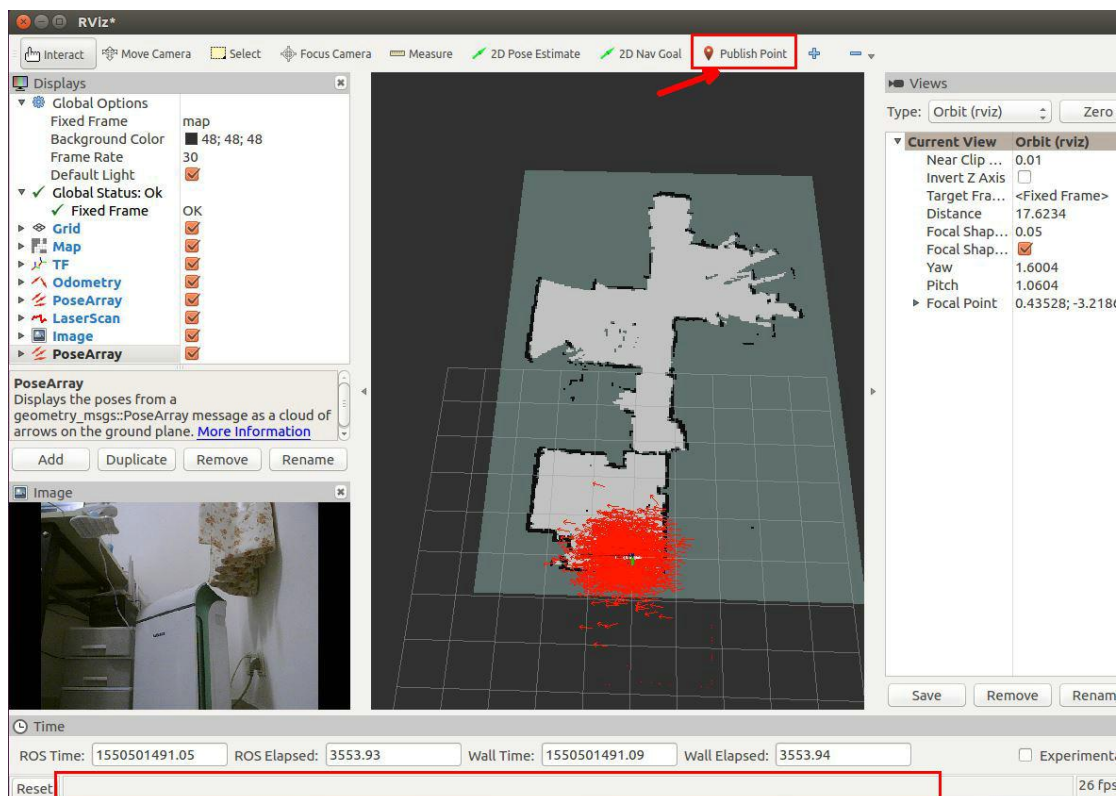
在上面的添加显示内容的实例中, 我们在 rviz 中添加了很多主题显示项, 并对各显示项的参数做了相应的设置。为了下次启动 rviz 时, 能直接显示这些内容和风格, 我们需要将当前的 rviz 显示风格以配置文件的方式保存, 下次启动 rviz 后只需要载入这个配置文件就能进入相应的显示风格。很简单, 点击 rviz 左上角[file]菜单, 在下拉中选择[Save Config As], 在弹出来的对话框中给配置文件取一个名字(我取名为my_cfg1), 然后直接 Save, my_cfg1.rviz 配置文件会被保存到系统中 rviz 的默认目录。下次启动 rviz 后, 通过点击 rviz 左上角[file]菜单, 在下拉中选择[Open Config], 打开相应的配置

文件就行了。如图 37。



(图 38) 机器人初始位置设定与导航目标设定

在机器人导航中，当机器人刚启动的时候，机器人位置往往需要人为给定一个大概的估计位置，这样有利于 AMCL 粒子滤波中粒子点的快速收敛。如图 38, 点击 [2D Pose Estimate] 按钮，然后在地图中找到机器人大致的位置后再次点击鼠标左键并保持按下状态，拖动鼠标来指定机器人的朝向，最后松手就完成对机器人初始位置的设定了。其实就是两步，先指定机器人的位置，再指定机器人的朝向。我们可以在地图中指定导航目标点，让机器人自动导航到我们的指定的位置。通过 [2D Nav Goal] 按钮就可以完成。操作步骤和机器人初始位置的设定是类似的，就不赘述了。



(图 39) 获取地图中指定点的坐标值

有时候我们需要知道地图中某个位置的坐标值，比如我们获取地图中各个位置的坐标值并填入巡逻轨迹中，让机器人按照指定巡逻路线巡逻。通过[Publish Point]按钮就可以知道地图中的任意位置的坐标值，点击[Publish Point]按钮，然后将鼠标放置到想要获取坐标值的位置，rviz 底部显示栏中就会出现相应的坐标值。

(4) rviz 启动方法

首先需要启动 roscore，然后启动 rviz，命令如下：

```
#打开终端，输入下面命令
roscore
#再打开一个终端，输入下面命令
rviz
```

10. 在实际机器人上运行 ROS 高级功能预览

到这里，《ROS 入门》这一章节教程就讲完了，由于是为了帮助大家入门 ROS，所以教程中的所有内容都只是在我们的调试工作平台（就是 PC 电脑）上进行的演示。实际的 ROS 机器人开发中，需要有一个调试工作平台（比如 PC 电脑）和一个机器人平台（比如安装有 ROS 的树莓派 3、搭载激光雷达、IMU、摄像头，且能移动的机器人底盘）。这样，我们就可以将 SLAM 建图算法、自主导航算法、语音交互算法、图像算法等放到机器人平台运行；再利用调试工作平台远程连接到机器人平台，就可以非常方便的远程调试 SLAM 建图算法、自主导航算法、语音交互算法、图像算法等算法了。所以在接下来的教程中将在实际的机器人平台（也就前面提到的我们的 miiboo 机器人）上展开，这里先提前让大家预览一下 miiboo 机器人上的 ROS 编程与算法开发的高级内容：

- (1) miiboo 机器人上的传感器 ROS 驱动开发
- (2) 基于 google-cartographer 的 SLAM 建图算法开发
- (3) 基于 ros-navigation 的机器人自主导航算法开发



- (4) 集成语音识别合成和自然语言处理的聊天机器人开发
- (5) 机器人与人工智能未来展望