新手教程：

从 Ubuntu 到 ROS-Gazebo-Moveit

机械臂运动规划仿真平台

Qiu Jialing
邱嘉聆

目录

从 Ubuntu 到 ROS-Gazebo-Moveit 机械臂运动规划仿真平台

一、目的

学习了解多轴机器人控制系统的整体框架，并学习如何搭建针对六自由度机械臂轨迹规划的仿真实验平台。

二、内容

2.1 多轴机器人控制系统

2.1.1 多轴机器人控制系统框架图

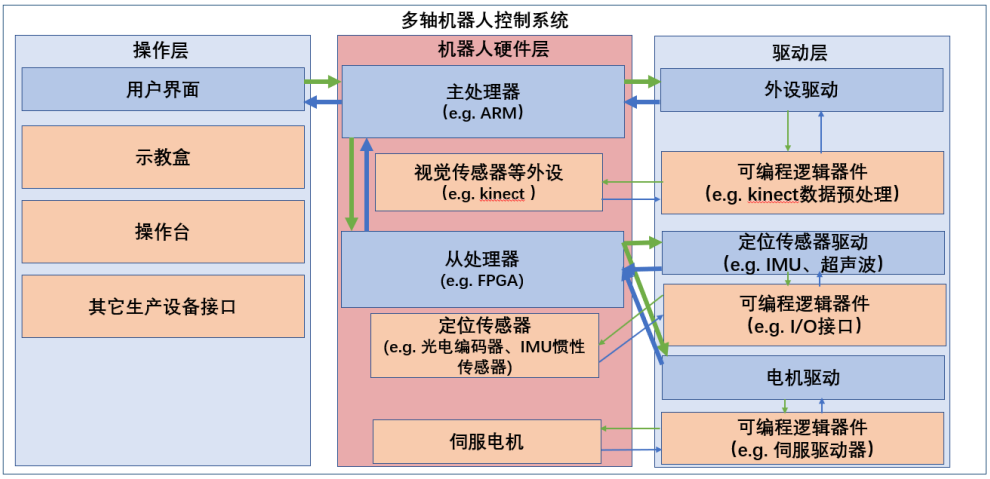


图 1. 多轴机器人控制系统框架图（图中绿色箭头代表控制信号，蓝色箭头代表反馈信号）

多轴机器人是一类拥有多个关节与连杆，各关节可以被各自独立或者联动控制，从而使机器人达到特定空间位置并执行特定动作的机器人。多轴机器人控制系统的功能即是让人可以方便的通过用户界面为机器人设定目标，同时驱动机器人平稳、精准地完成这一目标。这样就把多轴机器人控制系统分为三层：操作层，驱动层和机器人硬件层（图 1）。操作层由 Windows、Linux 或者定制的工业操作系统搭载用户界面软件形成，其形态可以是示教盒，操作台，或是附属于其它生产设备，用户界面需要包含的功能包括指定目标、完成参数配置、提供机器人实时状态监测、安全警报等。驱动层和机器人硬件层是驱动机器人完成设定目标的主力。驱动层包括软件和硬件两部分，驱动软件搭载在对应的处理器系统中，硬件即可编程逻辑器件装置在对应的传感器或者电机附近。硬件层，以主从控制系统为例，包含主处理器和从处理器，主处理器接收操作层发来的消息，并处理视觉传感器这一类感知周围环境的外设所采集的信息，提取相关目标信息并反馈给操作层，同时向从处理器发送该目标信息。从处理器接收主处理器发来消息，并向与机器人自身状态相关的传感器和电机驱动发送控制命令并接收反馈消息。

机器人处理器主要包括 ARM、DSP 和 FPGA。ARM 是一类嵌入式微处理器，DSP 具有快速计算的特点，可以用来实现复杂的信号处理和控制算法，而 FPGA 是现场可编程门阵列，是在单片 IC 上集成了逻辑阵列的可编程连接模块，适用于高速数字逻辑设计。

视觉传感器 Kinect 是微软公司发布的一款 3d 视觉传感器。IMU 惯性传感器和光电编码器为基于惯性传感器的航位推算方法和基于里程计的航迹推算方法提供数据，从而在机器人初始位置确定的情况下，计算每一时刻位姿相对上一时刻的位置和方向角变化，估算机器人实时位姿。

2.1.2 ROS 机器人操作系统功能

ROS（robot operation system）是一款开源的机器人操作系统，它有别于传统意义上的“操作系统”，提供了一系列工具为多个处理器之间的通讯提供了便利，能实现分布式的机器人控制系统。同时作为一个开

源平台，提供了大量可供复用的代码包，执行常用的传感器数据处理、底层设备控制和求解机器人运动轨迹等功能，并且能够方便的以不同的编程语言向 ROS 中添加扩展代码包。结合 ROS 兼容的硬件仿真软件 Gazebo 等，可以实现在虚拟环境中对各种控制算法进行试验。

2.1.2.1 ROS 机器人操作系统框架

整个 ROS 机器人操作系统可以分为三个层次，最上是代码库社区，社区中储存有各种门类的代码库，库中包含了代码堆，而代码堆里又分为了一些实现各功能模块的代码包。其次是文件系统，ROS 中的文件有很多种类型，包括各种定义文件，脚本文件，头文件，启动文件，清单文件。实现某一具体功能的文件被存储在一个包（package）中，几个相互依赖作用的包储存在一个堆（stack）中。除安装 ROS 后已经有的基本功能包之外，如果用户想要添加自己的扩展功能包，则需要借助 Catkin 工作空间。Catkin 工作空间相当于一个总的堆，在其中可以使用 catkin-make 工具编译其中所有的功能包。图中是 Catkin 各种空间下的文件体系。

图 2. Catkin 工作空间文件框架

最下一个层次是 ROS 的计算图层（图 3），ROS 的计算图由节点（Node）、话题（Topic）、服务（Service）和消息（Message），四种主要的元素组成，同时为弥补服务（Service）的不足，补充了另一个行动（action）元素。图中是一个主从式控制系统中的 ROS 计算图结构简图。图中 Node1 和 Node3 间通过 Topic1 进行联系，Node2 和 Node5 间以服务方式进行联系，Node4 间和 Node5 间以行动方式进行联系。

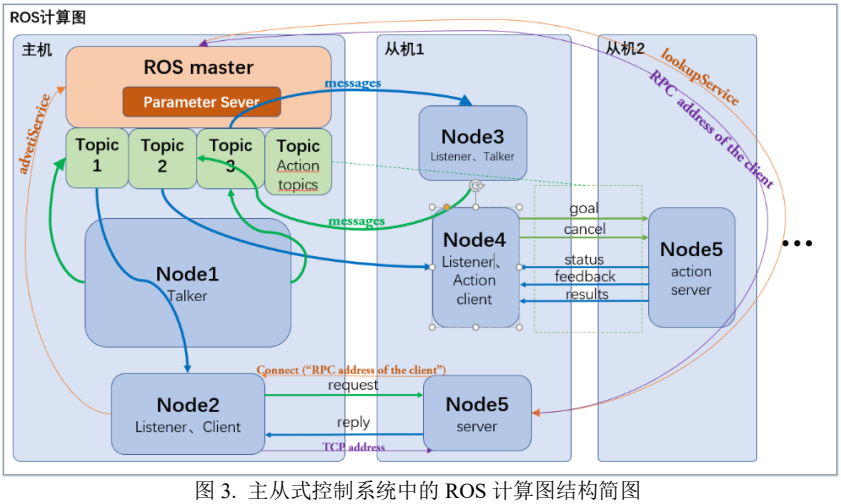


图 3. 主从式控制系统中的 ROS 计算图结构简图

节点是处理某一计算进程的模块，对应到 1.1 中的控制系统框架图上，是各驱动模块。而话题、服务和行动都是为节点间的信息交流提供工具。消息则是用于交流的信息，可以根据用户需求定义成不同的数据类型。ROS master 是 ROS 中的大脑，是一个节点、话题、服务和行动的管理器，它通过 RPC（Remote Procedure Call Protocol，远程过程调用）提供了将各个节点、话题、服务和行动登记造册，并在其中进行查找，使各个节点能够相互发现和联系的功能。ROS master 中的参数服务器（Parameter Sever）可以存储各类配置参数，并在不同节点上取用。

几种不同节点间通讯方式具有不同的特性。话题方式让消息以发布加订阅的方式传递。一个节点在给定话题中发布消息，另一个节点针对这个话题订阅消息。话题方式支持多个节点同时发布、订阅同一个主题的消息，是一种异步的通讯方式。服务方式中节点与节点之间的连接是直接，一个服务节点仅仅需要通过 ROS master 查询信息获得用户节点的地址，便可与之建立连接，客户端调用服务端上某一程序后，必须等待程序响应才能运行其它程序，是一种同步的通讯方式。行动方式是基于话题方式的一种便于接受反馈信息的异步通讯方式，可以在远程调用服务器上某程序后，周期性的获得服务器上的信息，或者取消发送到服务器上的某远程调用，期间可以转而执行其他的程序，机制灵活更加灵活。

2.2 ROS-Gazebo 六自由度机械臂仿真实验平台搭建过程

本教程中所使用的机器人控制系统是 ROS 控制系统，仿真软件是 Gazebo 物理仿真引擎。

整个仿真实验平台所用到的系统环境是：Ubuntu 18.04+ROS melodic+Gazebo9.0+Moveit。

2.2.1 Ubuntu 系统

如 1.2 节中介绍的 ROS 并不是传统的操作系统，而是便于传统操作系统能与底层硬件间沟通的中间件。要理解这个中间件是如何发挥作用的，首先需要理解操作系统是如何运作的。下面将以具有 Linux 内核的 Ubuntu 操作系统为例进行介绍。

2.2.1.1 Linux 内核是什么

以一块载有 Ubuntu 系统的瑞星 G3288 ARM 开发板为例（图 4），说明 ROS 是如何利用 Ubuntu 系统进行工作的。

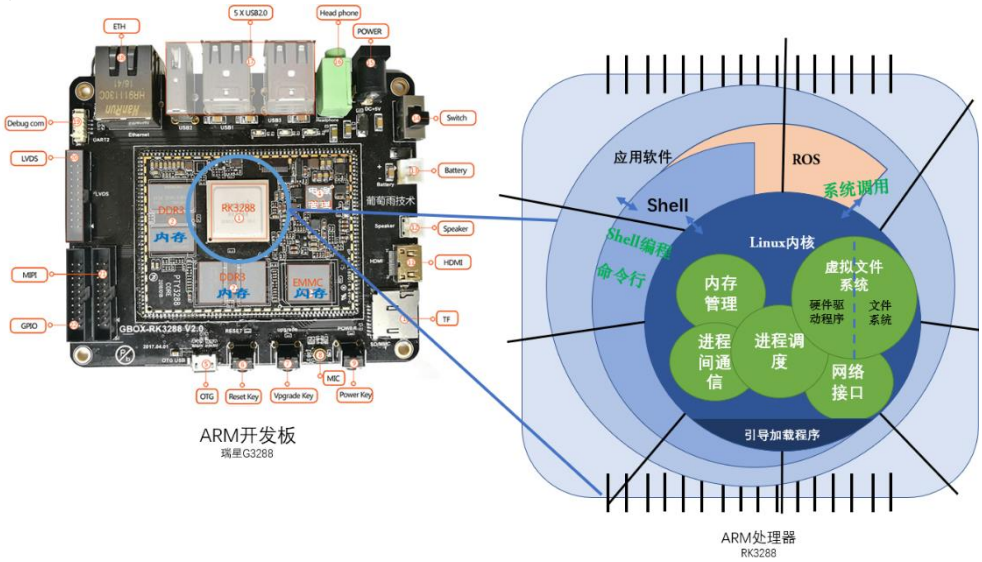


图 4. ARM 开发板上安装有 ROS 的 Ubuntu 系统

Ubuntu 系统是在 Linux 内核的基础上安装有各种应用软件的一个操作系统，而操作系统的内核的功能实际上就是为其上的各种应用程序管理硬件，管理通过开发板上接口接入的各种设备，管理正在运行的各种进程。为实现这些功能，Linux 内核设计为由五个子系统组成，即进程调度，内存管理，虚拟文件系统，网络接口和进程间通信。内核中包含有各设备的驱动程序和引导加载程序，将系统加载到开发板上，保证内核能够与开发版上的设备沟通。应用程序通过 Shell 或者采用系统调用的方式与内核交流，调用内核中的函数，其中通过 Shell 与内核交流又有采用 Shell 编程或者命令行两种方式。ROS 与应用程序与内核沟通的方式类似。

2.2.1.2 Ubuntu 系统安装

本教程中，由于只是搭建一个仿真平台，所以所有程序均在个人 PC 端上执行。那么第一步就是在 PC 端上安装 Ubuntu 系统，这里介绍 Windows10 与 Ubuntu18.04 双系统的安装方法。

1） 启动盘制作

从 Ubuntu 官网上下载 Ubuntu 18.04.1 桌面版系统的镜像文件：

> https://ubuntu.com/download/desktop

从以下地址下载启动盘制作工具：

> https://rufus.ie/

准备一个 4GB 以上的 U 盘，将 U 盘连接到电脑上，并格式化这一 U 盘。打开 Rufus 启动盘制作工具（图 5），在设备一栏选取刚刚连接的 U 盘。接着在启动项一栏 选择 FreeDOS，分区方式和目标系统类型保持默认不变。点击启动项选择右侧的选择按钮，在弹出的文件浏览器窗口中找到并选中刚刚下载好的 Ubuntu-18.04.1-desktop-amd64.iso 镜像文件。卷标号会自动更新。最后点击开始，等待写入过程完成，点击关闭。

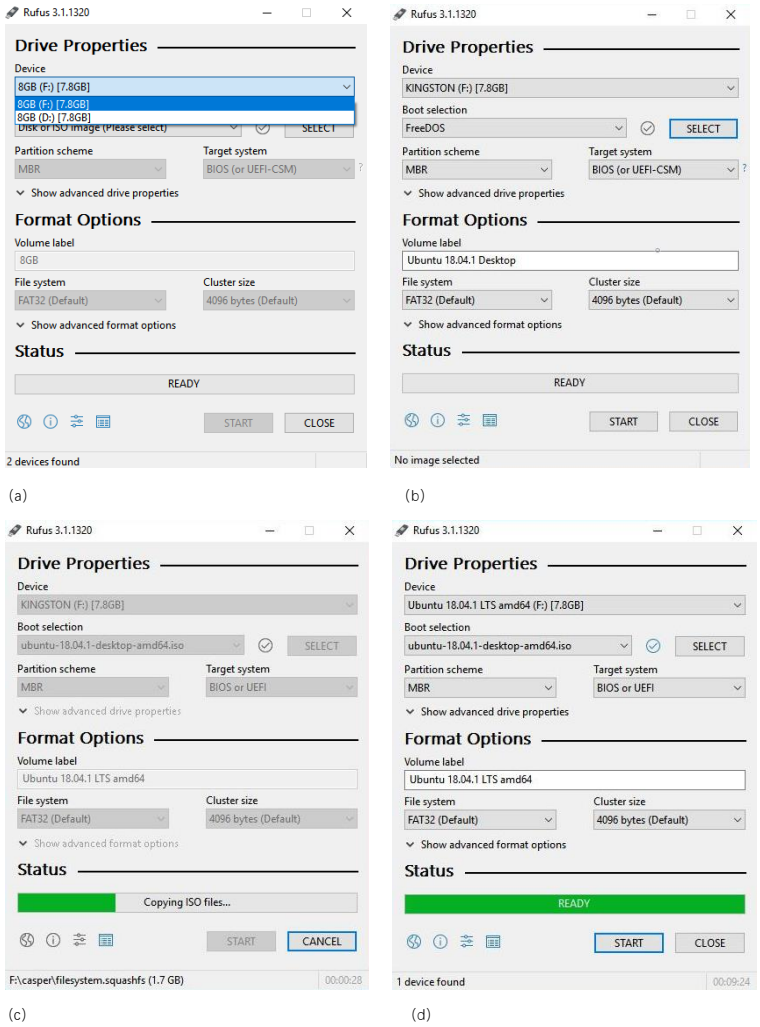


图 5. Ubuntu 系统启动盘制作过程

2） 安装双系统的磁盘分区

为防止同一台 PC 上的两个操作系统之间相互干扰，Ubuntu 必须被安装在单独的磁盘分区上。而且由于 Ubuntu 有自己的文件系统，在安装时会先将这一分区格式化。

使用 windows10 系统自带的磁盘管理工具即可以更改磁盘分区，这一工具的使用方法参照网络。在我的电脑上最终的磁盘分区如图 6 所示。

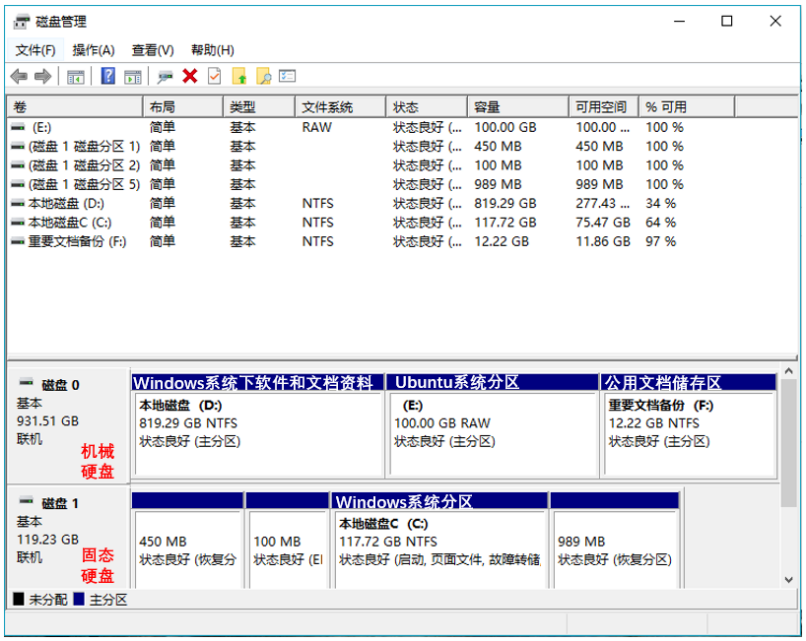


图 6. 安装 Ubuntu+Windows10 双系统时电脑的磁盘分区

原有的 Windows 系统装在固态硬盘上，而在机械硬盘上划分出 E 盘作为 Ubuntu 系统分区，F 盘作为两个系统间公用文档的储存区域。

分区中的注意事项有以下几点：

a) 一般个人 PC 电脑中都有固态硬盘和机械硬盘，固态硬盘性能更好，所以一般 Windows 系统需要装在固态硬盘上才能保证运行流畅。但 Linux 系统即使在机械硬盘上也可以流畅运行。

b) 由于在 2.1.1 节中提到过的 Linux 内核中的“虚拟文件系统”，就是将一切硬件虚拟为文件进行处理，硬件文件都存在/dev 目录下。在 Ubuntu 系统中文件目录与磁盘分区是相对独立的，是先有文件的目录树，再将某一分区“挂载”到某一目录下，表明将这一目录中的文件储存在这块分区中，与 Windows 系统中包含盘符的文件目录相区别。

Ubuntu 的主要文件目录有：

/ 根目录，其它所有目录都在根下
/boot 存放 Ubuntu 内核和系统启动文件
/bin 存储二进制可执行命令文件。/usr/bin 也存储了一些基于用户的命令文件
/root 根用户目录
/home 普通用户的默认目录
/sbin 系统命令存储位置。/usr/sbin 也存储了一些系统命令
/mnt 包括系统引导后被挂载文件的文件系统的挂载点
/dev 存储设备文件，包括所有外部设备，如硬盘、鼠标、键盘等
/etc 系统配置及服务配置文件、启动命令的目录
/lib 存储各种程序所需的共享库文件，
/lost+found 该文件一般为空，当系统非法关机后，会存放一些零散文件
/var 存储很多不断变化的文件，如日志文件
/usr 包括与用户直接有关的文件和目录，如应用程序，以及支持它们的库文件
/media 存放 Ubuntu 系统自动挂载的设备文件
/proc 这是一个虚拟的目录，它是内存的映射，显示内核及进程信息的虚拟文件系统。
/tmp 存储系统和用户的临时文件，该文件为任何用户都提供读写权限。
/initrd 用来在计算机启动时挂载 initrd.img 映像文件的目录，以及载入所需设备模块的目录。
/opt 作为可选文件和程序的存放目录，主要被第三方开发者用来简易安装和卸载软件
/srv 存储系统提供的服务数据
/sys 系统设备和文件层次结构，并向用户提供详细的内核数据信息

c） 在本次 Ubuntu 系统安装过程中，由于没有特殊需求，便直接将 Ubuntu 系统分区挂载到了根目录下，所有目录都存储在一个分区当中。

3） 系统安装过程

关闭电脑，将制作好的 U 盘启动盘插入电脑中，启动电脑，进入 BIOS 启动界面（依据电脑品牌的不同按下不同的按键），滚动滑条选择：

> Install Ubuntu

安装教程参考官方说明：

> https://tutorials.ubuntu.com/tutorial/tutorial-install-ubuntu-desktop#10

2.1.2.3 Ubuntu 系统相关概念

Ubuntu 系统的一大优势就是开放的环境，使用 Ubuntu 系统从网络上获取开发所需的各种库文件十分容易。仰赖于上一节介绍的目录树，Ubuntu 将各种软件所共享的库文件存储在/lib 目录下，节约了存储空间，再加上可以使用各种命令行工具从网络上下载和安装库，使库的使用和扩展变得简单。要理解这种在整个系统乃至网络内共享资源，而不是各个软件各自为政的机制，需要了解两个概念：“源”和“环境变量”。

1） 什么是 Ubuntu 的“源”

Ubuntu 的“源”，实际上就是一个软件仓库，用户可以在这个仓库中搜索软件，下载安装转件，并获得软件更新。通过官方认证的软件可以入住官方源，未获得官方认证的软件也可以入住第三方的源，也就是 PPA 软件源。系统只要知道了这个“源”的网址，就可以从源中获得软件，这些网址被统一存储到一个文件中，也就是：

```
/etc/apt/sources.list   #官方源
/etc/apt/sources.list.d   #非官方源
```

默认的官方源储存在美国的服务器上，一般需要替换为国内的镜像源，比如清华的镜像源：

```
https://mirrors.tuna.tsinghua.edu.cn/help/ubuntu/     #清华镜像源使用帮助网站
# 替换 Ubuntu18.04 系统中原有的 sources.list 文件
# 默认注释了源码镜像以提高 apt update 速度，如有需要可自行取消注释
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic main restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-updates main restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-updates main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-backports main restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-backports main restricted universe multiverse
deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-security main restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-security main restricted universe multiverse

# 预发布软件源，不建议启用
# deb https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-proposed main restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-proposed main restricted universe multiverse
```

Ubuntu 系统的命令行工具使得用户可以通过“终端”输入命令，给内核下达指令。比如与“源”相关的命令：

0	#sudo 是指以管理员权限进行操作
1	sudo apt-cache search package #在源中搜索含有“package”关键字的软件包
2	sudo apt-get install package #从源中下载安装名为 “package” 的软件包，必须使用准确的包名称
3	sudo apt-get update #从源中获得最新的软件包列表，列表会储存到本地，这样在使用 1、2 命令时，内核会从本地的列表中搜索定位这一软件包，之后联网进行搜索或下载
4	sudo apt-get upgrade #更新已下载的软件包（包括系统更新）
5	sudo apt-get upgrade package #更新包名为“package”的软件包

2） 什么是环境变量

环境变量是一个多义的概念，而 Ubuntu 系统中的环境变量是一系列值为某一文件路径的变量，系统通

过读取这些变量，来了解对应的资源储存在哪个文件路径中。

Ubuntu 系统的操作方法有两种，一种是通过“终端”也就是“命令行”，另一种是通过图形化界面，一般会同时使用两种操作方法。 Ubuntu 系统支持多个终端同时运行，而且可以使某些环境只在终端运行期间存在，也就是说可以提前为所有的终端统一配置一些环境变量，也可以在某一终端运行过程中随时添加特别的环境变量。统一配置要用到的文件是.bashrc 文件。文件路径为/home/xxx/.bashrc 的文件是用户 xxx 在启动新终端时，终端会读取的文件。如下是一个/home/xxx/.bashrc 的例子：

```
source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash
export PATH=$PATH:/home /tools/arduino-1.8.8  #将路径添加到环境变量 PATH 的末尾
export PATH=/path/to/your/dir:$PATH  #将路径添加到环境变量 PATH 的开头
export ROS_MASTER_URI=http://localhost:11311  #命名一个新的环境变量
export ROS_HOSTNAME=localhost
export TURTLEBOT3_MODEL=burger
export TURTLEBOT3_MODEL=waffle
export TURTLEBOT_3D_SENSOR=kinect
# 在 ros 的任何一个工作空间中都存在一个 setup.bash 文件, 包含了这个工作空间中所需的环境变量配置, 而 source 命令用来读取并执行其后文件路径中的命令，export 命令用来添加路径到环境变量中。而$PATH 表示该环境变量的值。
# 如果需要在用户启动的所有新终端中都包含某个新的库，只需要在这个.bashrc 文件中加入新的环境变量。
```

在 Ubuntu 系统的终端中可以用到的命令还有很多，将随着教程进行逐一学习。

2.2.2 在 Ubuntu 系统上搭建 ROS 环境

2.2.2.1 ROS 桌面完整版安装

本次是在 Ubuntu18.04 上安装 ros-melodic 版本。打开一个新终端并在其中输入如下命令：

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'  #添加 packages.ros.org 到第三方源列表中
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654  #添加安装密匙
sudo apt update  #从源中更新可用软件包列表
sudo apt install ros-melodic-desktop-full  #安装 ros-melodic 桌面完整版
sudo rosdep init  #初始化 rosdep 工具，便于安装系统依赖项
rosdep update
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc  #将“”中的语句添加到.bashrc 文件末尾（~/是用户主目录的缩写）
source ~/.bashrc  #读取并执行.bashrc 文件
sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential  #安装一些附加工具
```

2.2.2.2 ROS 完整版中包含的几个重要功能包

ROS 完整版安装时已经包含 RVIZ，GAZEBO 两个重要的三维模拟仿真功能包。RVIZ 做的事情是订阅以话题、参数的形式发布的图像、模型、路径等信息，对其进行可视化的渲染，让开发者更容易理解数据的意义，对于多自由度机械臂，RVIZ 需要知道模型的变换矩阵来可视化机械臂模型。而 GAZEBO 是对真实物理引擎的仿真工具，可以仿真真实电机和传感器，GAZEBO 需要知道机械臂的各种物理参数来完成仿真，可以像控制真实机械臂一样控制在 GAZEBO 中控制机械臂。

2.2.2.3 ROS 扩展包的安装

1） 安装命令：

```
sudo apt-get install ros-melodic-PACKAGE   #PACKAGE 需要被替换为所需的扩展包名称
```

还需要安装的主要扩展包是 Moveit！运动规划包，扩展包名称名为“moveit”。Moveit 功能包提供了一系列移动操作的基础功能：包含运动规划，操作控制，3D 感知，运动学，碰撞检测等等。

2） 什么是依赖项

一个实现复杂功能的功能包会需要用到多种基本功能，而实现这些基本功能的程序通常不用开发者再次编写，就需要依赖别的已经编写好的功能包，这就是依赖项。在 ROS 中依赖项会在 cmakelist.txt 和 package.xml 中被声明，并且对于不同功能的依赖项有不同的标签。一个 package.xml 中依赖项声明的例子：

```
  <depend>roslaunch</depend>
  <depend>robot_state_publisher</depend>
  <depend>rviz</depend>
  <depend>joint_state_publisher</depend>
  <depend>gazebo</depend>
```

安装一个功能包的所有依赖项所需命令：

```
cd ~/catkin_ws/src    #cd 命令代表进入某一文件路径
sudo rosdep install --from-paths PACKAGE PATH --ignore-src --rosdistro melodic -r -y
```

安装完依赖项，电脑的软件配置就基本完成了。

2.2.3 机械臂模型设计与虚拟硬件配置

2.2.3.1 SolidWorks 机械臂模型设计

本教程中需要用到一个打磨机械臂（图 7）的三维模型，这个三维模型（图 8）由手臂和打磨头两部分组成。下面将介绍如何为自己的机械臂三维模型添加连杆坐标系。

图 7. 打磨机械臂实物图

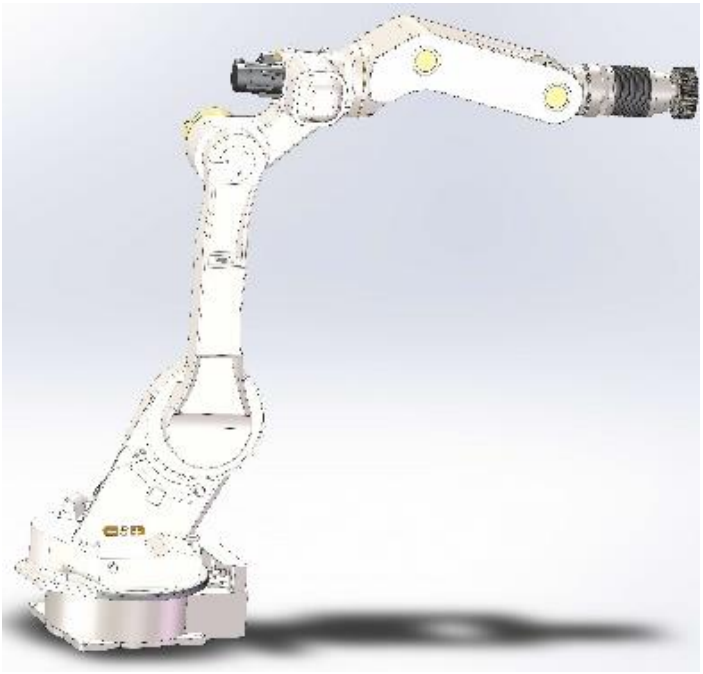


图 8. 打磨机械臂三维模型图

1） SolidWorks 中的坐标系体系

为将 solidworks 中建立的三维模型转化为 ros、rviz 和 gazebo 都可以识别的通用机器人描述文件形式，首先需要为机器人的各个关节建立坐标系。这一步看起来很简单，实际上因为 SW 的一个装配体中存在多个文件层级：装配体、子装配体、零件、特征、草图，而且在各个层级中都有各自的基准坐标系。在建立新的坐标系时要特别注意正在操作的是哪一个文件层级，选取参考的点、线、面时也要注意它们所处的层级（图 9）。

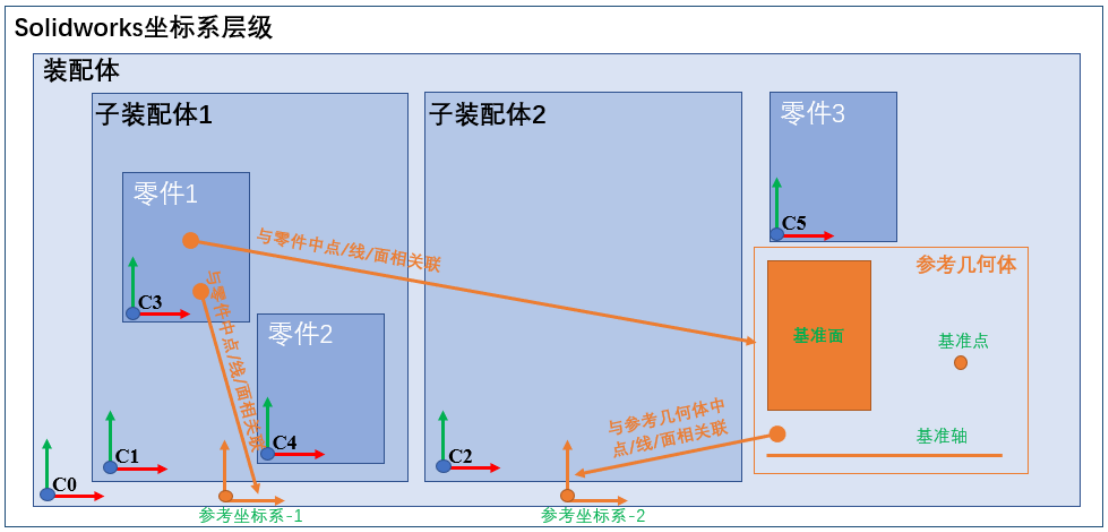


图 9. SolidWorks 中的坐标系层级框图

要得到机械臂的变换矩阵，首先需要建立机械臂各个连杆的坐标系，但是由于 SW 中的多个文件层级存在，在建立坐标系的过程中多了一些注意事项。

a) 将整个机器人完全装配好后，为每个连杆分别生成子装配体。子装配体中应该包含该连杆的所有零件。

b) 在 Solidworks 中每个装配体、子装配体和零件都会含有一个默认的世界坐标系即图中 C0-C5。在这三个层级中都可以自定义参考坐标系，这些参考坐标系处于各自的世界坐标系中，也就是说 Solidworks 没法计算不同层级中坐标系之间的变换矩阵。所以在本教程中，所有连杆的参考坐标

系应该被建立在总装配体层级中，如图中“参考坐标系-1”和“参考坐标系-2”。

c) 建立坐标系时，最不容易出错的方法就是：先调整机械臂到一个合适的位姿，然后固定所有零部件，最后找到合适的点线面来建立参考坐标系，如果找不到合适的点线面则先建立辅助的参考几何体。

d) 如果需要能够随着机械臂位姿变换而自动更新的坐标系，则需要特别注意 Solidworks 中的“关联”。即参考坐标系-1 关联的点/线/面只能是子装配体 1 里的，参考坐标系-2 同理。不论此关联是直接，或者通过参考几何体间接发生。

依照以上原则建好坐标系后，得到的模型如下图：

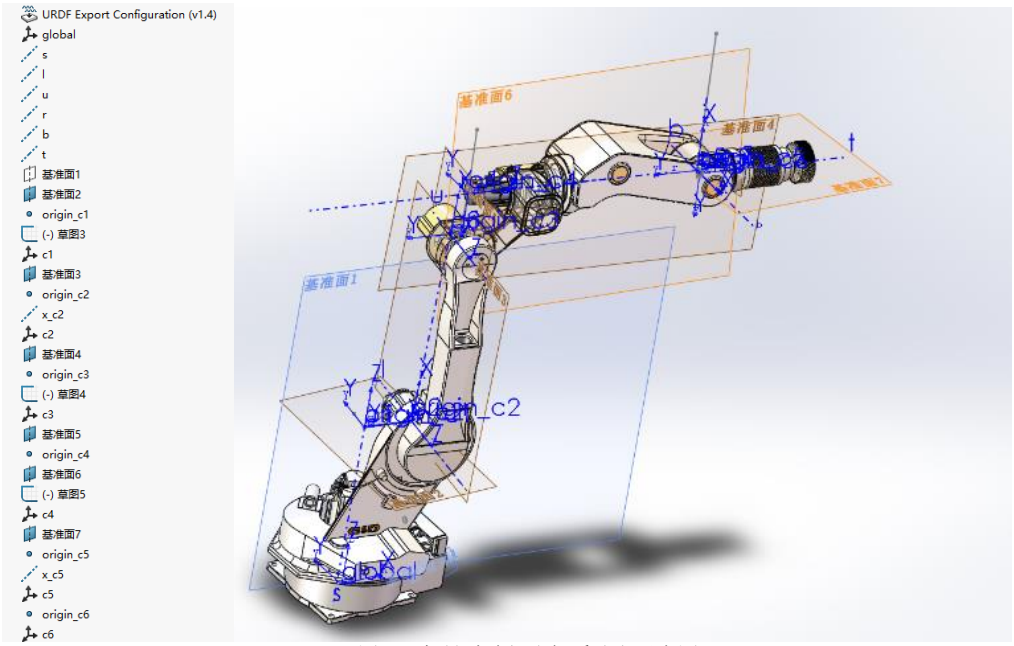


图 10. 机械臂连杆坐标系建立示意图

2.2.3.2 通用机器人描述文件（URDF）的编写

建立好坐标系之后，就可以开始 URDF 文件的编写了，手动编写这个文件需要进行大量的计算，比如相邻连杆坐标系之间的变换矩阵、每个连杆的质量矩阵等，但已经有软件可以代替手工计算。

1） sw_urdf_exporter 插件

插件下载地址：

> http://wiki.ros.org/sw_urdf_exporter

使用这个插件，只需要选择合适的各连杆坐标系和各关节基准轴，设置一些速度限制和表面摩擦系数等属性，就可以从装配体生成机械臂 URDF 文件以及匹配的 STL 模型文件，同时，该插件以 ROS 扩展包的格式存放生成文件，并提供了基础的 GAZEBO、RVIZ 可视化模型的可执行文件。后续工作就是在这个扩展包的基础上添加更多功能。

2） 基本连杆与关节定义

本教程完整文件请参阅 github 仓库，此处仅贴出部分代码进行说明。

基本连杆定义中主要包括质量属性、外观属性和碰撞检测属性三个部分：

```
<link name="00_link">  #连杆名称
<inertial>  #质量属性开始标签
<origin  #重心坐标系相对连杆坐标系的变换，按照先旋转后平移原则。
xyz="0.0651952757401018 -0.00111135577866065 0.0820117972141799"
rpy="0 0 0" /> #欧拉角，r 为绕 x 轴旋转，p 为绕 y 轴旋转，y 为绕 z 轴旋转
<mass value="3" /> #连杆质量/kg,一般设置较小的质量，在 gazebo 中的仿真比较稳定
<inertia  #惯量矩阵（反对称阵）
```

```
ixx="1.9849048937555"
ixy="0.00410475564409846"
ixz="0.106687639386624"
iyy="3.61179659152104"
iyz="0.0107423198174343"
izz="4.99307347614783" />
</inertial>  #质量属性结束标签
<visual>  #外观属性
<origin  #相对于连杆坐标系的变换，一般为零
xyz="0 0 0"
rpy="0 0 0" />
<geometry>  #几何属性，可以为简单几何体，也可以插入网格文件
<mesh filename="package://m_mh24/meshes/base_link.STL" />
</geometry>
  <material name="DarkGrey"/>
</visual>
<collision>  #碰撞检测属性
      <origin  #相对于连杆坐标系的变换
      xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>  #用于碰撞检测的几何模型
          <mesh filename="package://m_mh24/meshes/base_link.STL" />
      </geometry>
  </collision>
</link>  #连杆定义结束
```

基本旋转关节定义：

```
<joint name="s_joint" type="revolute">  #关节名称与关节类型，关节类型可以有四种：fixed、prismatic、revolute、continuous
  <origin  #子连杆坐标系相对父连杆坐标系的变换，按照先旋转后平移原则。
xyz="-0.00549 0.019704 0.62959"
    rpy="0 0 -0.16856" />
  <parent link="00_link" />  #父连杆名
<child link="01_link" />  #子连杆名
<axis xyz="0 0 1" />  #轴向量（子连杆坐标系中），一般所有关节轴向量需统一为（0,0,1）
<limit  #转动限制参数
lower="-3.14"  #最小角度
upper="3.14"  #最大角度
effort="1000"  #最大受力
  velocity="3" />  #最大速度
<dynamics  #力学属性
  damping="1"  #阻尼力
  friction="30"/>  #摩擦力，这个参数很重要，关系到模型能否自然保持静止
</joint>  #关节定义结束
```

定义完所有的基本连杆与关节后，为将机械臂固定到世界坐标系中，还需要增加一个世界连杆与固定关节。

```
<link name="world"/>
<joint name="world_base_joint" type="fixed">
<parent link="world"/>
  <child link="00_link"/>
  <origin xyz="0 0 0" rpy="0 0 0" />
</joint>
```

3） 闭环机构的处理方式

URDF 类型的文件不支持闭环机构，因为它不允许两个不同的父连杆拥有一个共同的子连杆。实际上，在 Gazebo 中使用 URDF 文件时会自动将其转换为 SDF 文件，而 SDF 文件允许这一点。所以若需要使用 URDF 文件描述一个闭环机构，就需要在文件中声明插件，并调用它，从而在 SDF 文件中添加用于形成闭环的关节。具体方法查看如下 Github 仓库。

```
https://github.com/wojiaojiao/pegasus_gazebo_plugins.git
```

4）ROS-Gazebo 标签

URDF 格式是一种 xml 文件格式，最初是 ROS 中用于描述组成机器人的所有组件的文件格式，要在 Gazebo 中使用它，需要添加一些插件，并添加相应仿真属性的标签。

a) ros_gazebo_plugin 控制器标签

gazebo_ros_control 插件的功能是提供 ros 平台与 gazebo 间通信所需的协议，在 urdf 文件中配置所需参数的方式如下：

```
<gazebo>
  <static>false</static>
<plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/m_mh24</robotNamespace>  #名称空间，各处需保持一致
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
    <legacyModeNS>true</legacyModeNS>
  </plugin>
</gazebo>
```

b) 与 ros_gazebo_plugin 匹配的执行元件接口标签

有了通信协议之后要对各个关节处的伺服电机和减速器进行声明，确定控制方式，以下是对旋转关节“s”控制器的声明标签。

```
<transmission name="s_transmission">  #控制器名称
  <type>transmission_interface/SimpleTransmission</type> #控制器接口类型
<joint name="s_joint">  #所控关节名称
    <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>  #控制方式：位置反馈（另外还有力矩反馈和速度反馈两种方式）
  </joint>
  <actuator name="s_motor">  #驱动器名称
    <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>  #控制方式：位置反馈
    <mechanicalReducction>1</mechanicalReducction>  #减速比，减速比越大可以负担的机械臂质量越大
  </actuator>
</transmission>
```

3) Gazebo 标签

a) Gazebo 中模型的外观与 RVIZ 中不一致，若要改变其中模型外观必须要定义模型的材料。材料可以在文件中被预先定义：

```
<material name="DarkGray">
<color rgba="0.3 0.3 0.3 1.0"/>  #颜色标签
</material>
```

然后更改连杆材料属性为预设值：

```
<gazebo reference="00_link">
<material>Gazebo/DarkGray</material>
</gazebo>
```

b) 各种传感器标签

Gazebo 同样提供了一些传感器插件，其中比较常用的有受力和力矩传感器（ft sensor）：

```
<gazebo reference="s_joint">
  <provideFeedback>true</provideFeedback>
</gazebo>

<gazebo>
  <plugin name="ft_sensor" filename="libgazebo_ros_ft_sensor.so">
    <updateRate>100.0</updateRate>
    <topicName>m_mh24/ft_sensor/s_joint</topicName>
    <jointName>s_joint</jointName>
  </plugin>
</gazebo>
```

4) 利用 RVIZ 可视化界面检查模型

编写好 URDF 文件后，可以先用 RVIZ 检查模型是否被正确定义，各个连杆间的变换矩阵是否正确，各个关节的基准轴是否被正确定义。要实现在 RVIZ 中显示模型并调整位姿，需要利用到 ros 中的 launch 文件，在 launch 文件中，可以定义变量，设定变量初值；利用<param>标签改变 ros 参数服务器中参数的值；利用<node>标签同时启动多个 ros 节点。下面是本教程中的 display.launch 文件。

```
#file: display.launch
<?xml version="1.0"?>  #说明文档是 xml 文件，在 Ubuntu 系统中会自动以颜色区分字符
<launch>
  <arg  #变量定义：model
    name="model" />
  <arg  #变量定义：gui
    name="gui"
    default="true" /> #变量初值
  <param  #将变量上载到 ros 参数管理器
    name="robot_description"  #“机器人描述文件”是参数管理器里的重要变量
    textfile="$(find m_mh24)/urdf/m_mh24.urdf" />  #路径为 m_mh24 扩展包里的 urdf 文件夹下的 m_mh24.urdf 文件
  <param
    name="use_gui"  #“gui”是 ros 提供的一个功能，启动后可以通过滑条改变关节状态
    value="$(arg gui)" />  #赋值为变量 gui 的值
  <node  #启动节点
    name="joint_state_publisher"  #节点功能：发布所有可动关节当前状态到/joint_state 话题
    pkg="joint_state_publisher"
    type="joint_state_publisher" />
  <node
    name="robot_state_publisher"  #节点功能：从/joint_state 话题接收关节状态，发布新的机械臂各连杆间变换矩阵到话题/tf 和/tf_static
    pkg="robot_state_publisher"
    type="state_publisher" />
  <node
    name="rviz"  #节点功能：启动 rviz 可视化界面
    pkg="rviz"
    type="rviz"
    args="-d $(find m_mh24)/urdf.rviz" />
</launch>
```

在终端输入命令

```
roslaunch m_mh24 display.launch
#roslaunch 命令用于启动 launch 文件，格式为 roslaunch 扩展包名 launch 文件名
```

执行后界面如图 11：

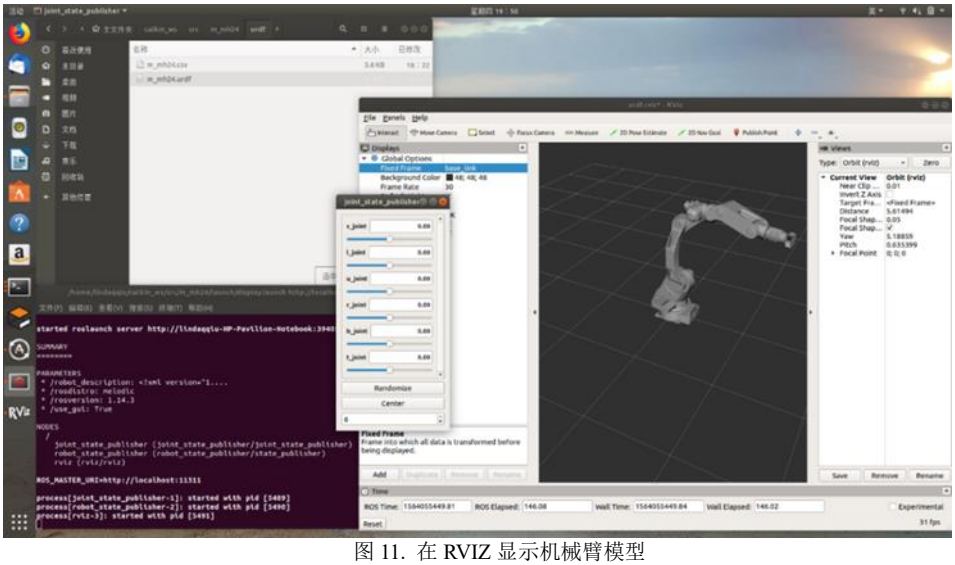


图 11. 在 RVIZ 显示机械臂模型

2.2.4 各关节控制器配置

2.2.4.1 ROS 中的控制器

上一节介绍 ros_gazebo_plugin 标签的时候，其中有一项<hardwareInterface>, 其中使用的是 PositionJointInterface，也就是位置接口，另外还有力矩与速度两种接口。硬件接口的主要功能是读和写特定数据类型的信息，它从控制器读取位置/速度/力矩命令，向硬件写入位置/速度/力矩命令。ROS 中硬件接口的类型按照它向硬件写入的数据类型划分，同一个硬件上可以同时设置多个类型的硬件接口。

负责向硬件接口写入命令的单元称为控制器，ROS 中已经提供了很多种类的控制器，大多数都是基于 PID 控制的。ROS 控制器代码库网址：

```
https://github.com/ros-controls/ros_controllers
```

其中 JointTrajectoryController 能为一组关节提供在关节空间内进行描述的轨迹数据，并且依据所配合硬件接口的不同提供了五个不同的类，如图 12。

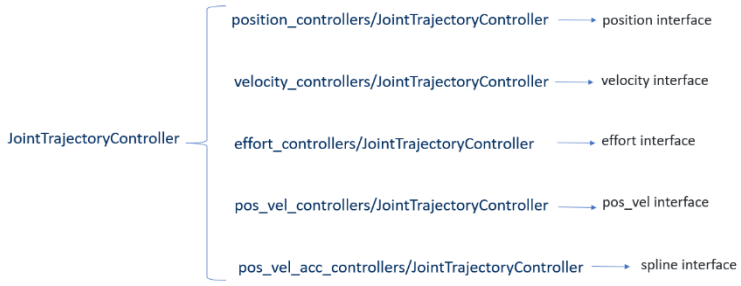


图 12. JointTrajectoryController 的几种类别

本教程中采用的是 position_controller/JointTrajectoryController，其工作的原理是，接收关节当前状态和目标状态信息（位置、速度、加速度），计算出所需位置控制信号，并向硬件接口发送位置控制信号，不断重复这一过程以形成反馈回路。

如图 13 中下半部分描述了此控制器与硬件接口的作用形式，可以看到这一控制器需要和另外两个单元配合才能够完成任务，一个是发送关节状态的 joint_state_publisher,另一个是由 Moveit 提供的发送目标轨迹的 FollowJointTrajectory。后者作用的具体方式是，在 Moveit 求解出机械臂由当前位姿运动到目标位姿期间各关节所走过的轨迹之后，将一组轨迹拆解成一系列平滑连接的目标点，并以一定数据形式通过 FollowJointTrajectory 逐一发送。

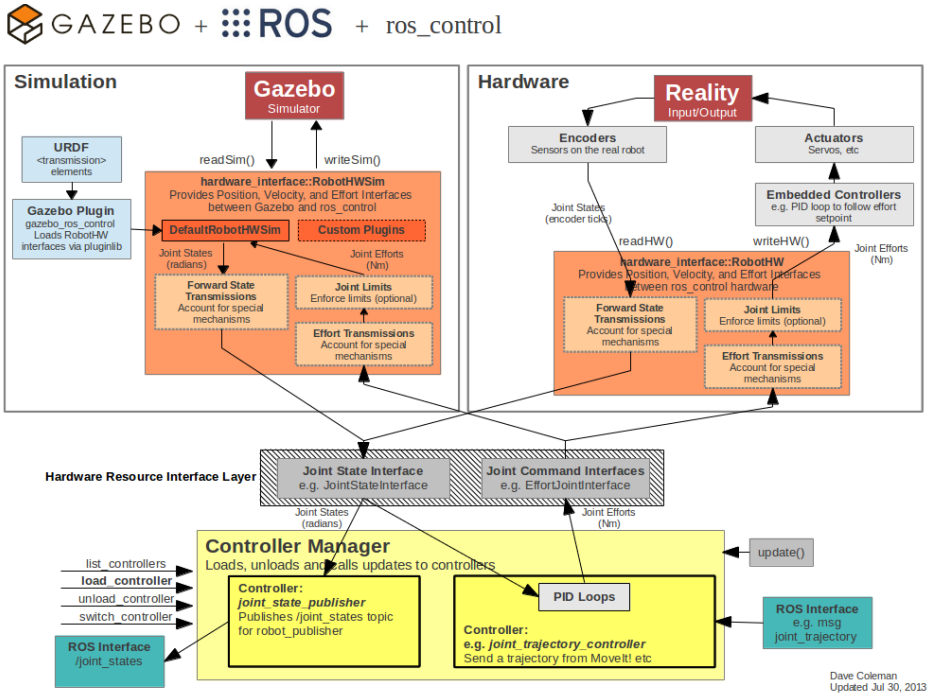


图 13. ROS controller 官方示意图

2.2.5 利用 Moveit 包完成机械臂从起始位姿到目标位姿的轨迹规划

2.2.5.1 Moveit-ROS-Gazebo 三者间的关系

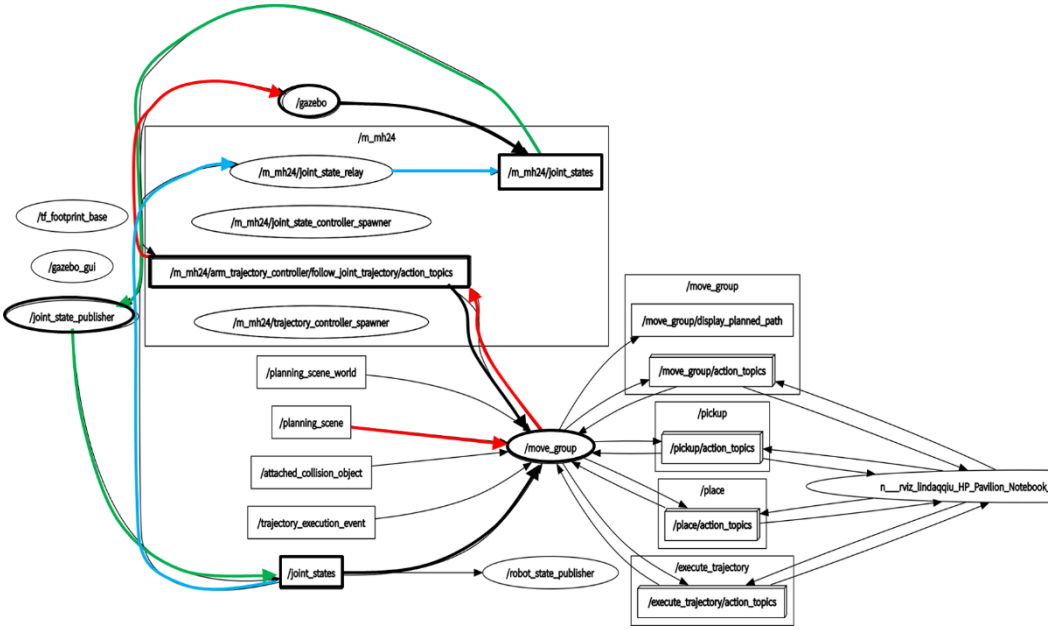


图 14. 本教程中 ROS-Gazebo 仿真平台节点和话题网络图

本教程就是 ROS、Gazebo 和 Moveit 三者结合使用实现了利用改进的 mh_24 型机械臂进行家具打磨的仿真实验平台。这三个组件各自的分工是：Gazebo 中存在一个虚拟的机械臂，它代替了真实物理世界中的机械臂的位置，接受 ROS 平台对它的控制，而 Moveit 接受用户的目标指令，处理从 Gazebo 中传回的传感

器信号，并将规划好的轨迹传给 ROS 平台，ROS 平台将其用于控制 Gazebo 中机械臂。

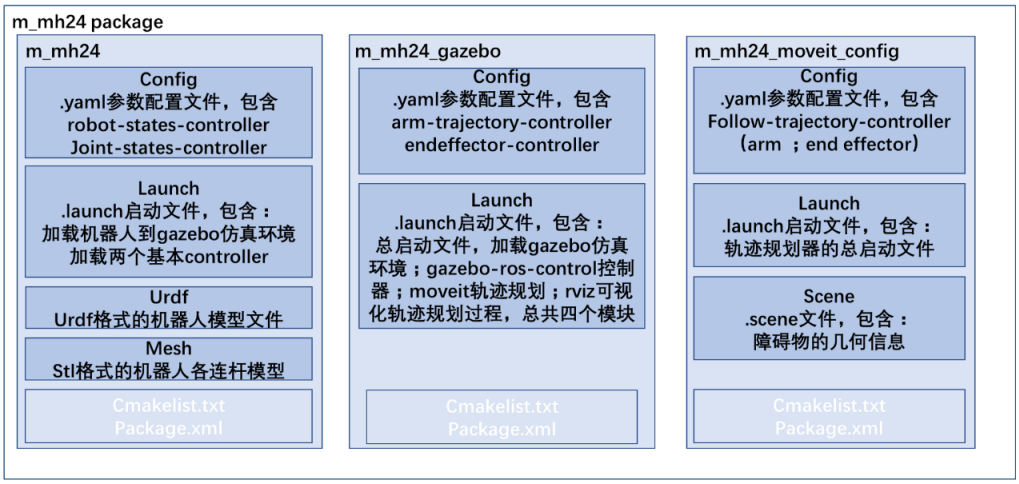
图中是本教程的 ROS 节点与话题网络图，方框中为话题，椭圆框中为节点。红色线演示了指令从用户界面经 Moveit 轨迹规划器传递到 gazebo 中从而控制虚拟机械臂运动的路线。黑色线表示了当前关节状态信息的发布与接收。绿色线与蓝色线表示了当前关节状态信息在 Moveit 和 Gazebo 间交流更新的过程。

节点	/gazebo 运行 Gazebo 物理仿真引擎
节点	/move_group 运行 Moveit 轨迹规划器
节点	/joint_state_publisher 发布各关节当前状态到/joint_states 话题
话题	/joint_states 存放各关节当前状态
话题	/planning_scene 存放用户界面设定的目标点
行动	/m_mh24/arm_tarjectory_controller/follow_joint_trajectory/action_topics arm_tarjectory_controller 与 follow_joint_trajectory 之间进行信息交流的方式，后者向前者发送目标指令，前者给予后者反馈与完成度信号
节点	/m_mh24/joint_state_relay 为解决 gazebo 发布关节状态的话题/m_mh24/joint_states 与 moveit 发布关节状态的话题/joint_states 名字空间不一致的问题，将两个话题内容统一

2.2.5.2 功能包 m_mh24 的搭建过程

1） m_mh24 功能包文件目录

该功能包由三个子功能包组成，即 m_mh24、m_mh24_gazebo 与 m_mh24_moveit_config，图 15 中简要介绍了几个功能包中所包含的模块。图 16 中为该功能包文件目录。



*modified mh24 package: 目标功能是实现用六自由度打磨机械臂打磨一个家具过程的仿真。

图 15. m_mh24 及其子功能包所包含模块简介

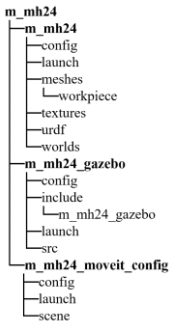


图 16. m_mh24 功能包文件目录树

2） 建立 catkin_ws 工作空间以存放功能包

打开新终端，在其中输入如下命令：

```
mkdir -p ~/catkin_ws/src  #mkdir 是创建文件目录命令，-p 为创建嵌套文件目录
cd ~/catkin_ws/  #进入文件路径
```

```
catkin_make   #catkin 工作空间创建工具
echo "source /home/lindaqqiu/catkin_ws/devel/setup.bash" >> ~/.bashrc
#将“工作空间的 setup 文件路径”写入用户.bashrc 文件
cd ~/catkin_ws/src/
catkin_create_pkg m_mh24 std_msgs rospy roscpp
#创建 ros 功能包 m_mh24，并以 std_msgs、rospy 与 roscpp 作为依赖项
```

3） Gazebo 虚拟世界配置

虚拟世界配置包含三个部分即空世界、机械臂和障碍物。机械臂的配置文件已经由 sw_urdf_exporter 插件生成，并且已经添加好了仿真所需的各种标签。障碍物模型命名为 workpiece，储存在子功能包 m_mh24 的 mesh 文件夹中。

文件路径：……m_mh24\m_mh24\launch\m_mh24_gazebo_with_workpiece.launch

代码片段	解释
<!-- send robot urdf to param server -->	<!-- xxx --> 为注释
<include file="$(find m_mh24)/launch/m_mh24_upload.launch"/>	将机械臂模型发送到参数服务器，便于包中的所有组件使用。
<include file="$(find gazebo_ros)/launch/empty_world.launch">	启动空 gazebo 世界的 launch 文件。 *ros 语法：可在 launch 文件中 include 其它 launch 文件以依次启动这些文件
<arg name="paused" value="$(arg paused)"/>	
<arg name="gui" value="$(arg gazebo_gui)"/>	
</include>	
<!-- push robot_description to factory and spawn robot in gazebo -->	在 gazebo 中放置机械臂模型
<node	
name="spawn_gazebo_model"	
pkg="gazebo_ros"	
type="spawn_model"	
args="-urdf -param robot_description -model robot"	放置位置在原点
respawn="false"	
output="screen" />	
<node	在 gazebo 中放置家具模型
name="spawn_gazebo_model2"	
pkg="gazebo_ros"	
type="spawn_model"	
args="-urdf -file $(find m_mh24)/urdf/workpiece.urdf -model workpiece -x 2.1 -y 0 -z 0"	放置位置在（2.1,0,0）
respawn="false"	
output="screen" />	

配置完成的虚拟世界如图 17 所示。

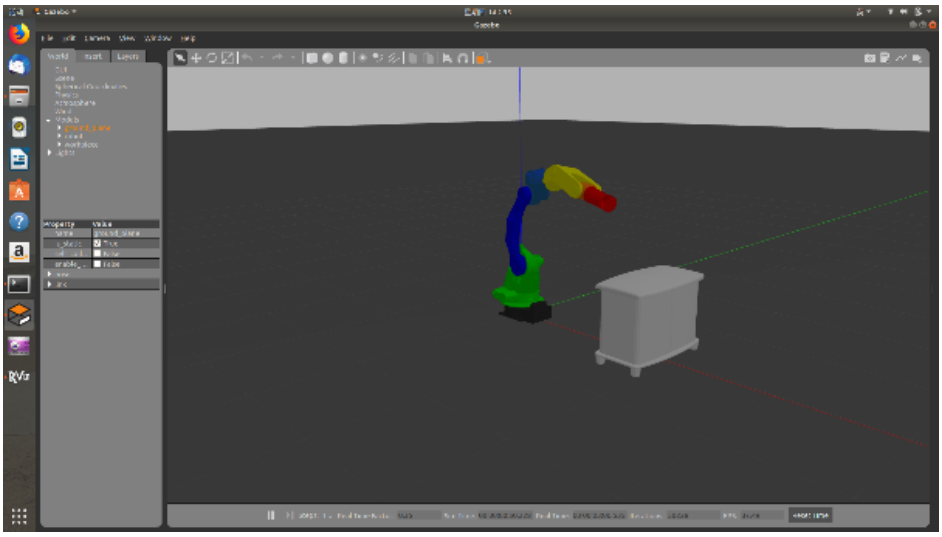


图 17. 配置完成的 Gazebo 虚拟世界

4） 生成 Moveit 功能包

Moveit!提供了 Moveit! SetupAssistant 来生成一个基础的功能包，只需要再在这个基础上进行修改就能够使用。使用教程网址如下：

http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/setup_assistant/setup_assistant_tutorial.html

5） 各控制器配置与启用

a) robot_states_publisher

机械臂状态的发布器目的是为了发布机械臂各连杆之间的变换矩阵，而 movieit 和 rviz 接收此信息以建立机械臂的的坐标系树。

文件路径：……m_mh24\m_mh24\launch\m_mh24_gazebo_with_workpiece.launch

代码片段	解释
<!-- Robot state publisher(publish the tf information to ros parameter service) -->	发布机械臂各连杆变换矩阵到参数服务器
<node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">	
<param name="publish_frequency" type="double" value="50.0" />	发布频率（Hz）
<remap from="/robot_states" to="/m_mh24/robot_states" />	设定发布话题的名字空间
</node>	

b) joint_states_publisher

第一步：编写控制器参数配置文件。

文件路径：……m_mh24\m_mh24\config\joint_state_controller.yaml

代码片段	解释
joint_state_controller:	发布关节状态
type: joint_state_controller/JointStateController	类别
publish_rate: 50	发布频率（Hz）
*.yaml 文件在 ROS 中专门用来配置参数，使用.yaml 文件统一管理参数的好处是便于后期进行参数调整。另外.yaml 文件使用空格缩进划分级别，所以在编写时需要十分注意格式。	

第二步：加载参数并启动控制器。

文件路径：……m_mh24\m_mh24\launch\m_mh24_gazebo_with_workpiece.launch

代码片段	解释
<!-- joint_state_controller(publish joint states to m_mh24/joint_states topic) -->	
<rosparam ns="/m_mh24" file="$(find m_mh24)/config/joint_state_controller.yaml"	上载控制器配置

command="load"/>	参数到 ros 参数服务器
<node	启动控制器
ns="/m_mh24"	名字空间为 m_mh24
name="joint_state_controller_spawner"	
pkg="controller_manager"	
type="spawner"	
respawn="false"	
output="screen"	
args="joint_state_controller"/>	

c) arm_trajectory_controller

类型为 position_controllers/JointTrajectoryController 的 ROS 控制器，其功能请参看之前的内容。

文件路径：……m_mh24\m_mh24_gazebo\config\ joint_trajectory_controller.yaml

代码片段	解释
m_mh24:	名字空间
arm_trajectory_controller:	控制器名称 *可设置多个控制器，并使每个控制器控制一组关节。
type: "position_controllers/JointTrajectoryController"	类型
joints:	所包含关节
- s_joint	
- l_joint	
- u_joint	
- r_joint	
- b_joint	
gains:	PID 参数
s_joint:	
p: 100	
d: 1	
i: 1	
i_clamp: 1	
l_joint:	
p: 100	
d: 1	
i: 1	
i_clamp: 1	
u_joint:	
p: 100	
d: 1	
i: 1	
i_clamp: 1	
r_joint:	
p: 100	
d: 1	
i: 1	
i_clamp: 1	
b_joint:	
p: 100	
d: 1	
i: 1	
i_clamp: 1	
t_joint:	
p: 100	
d: 1	

i: 1	
i_clamp: 1	
constraints:	时间限制：执行超过此时间则返回执行失败消息。
goal_time: 0.6	
stopped_velocity_tolerance: 0.05	
s_joint: {trajectory: 0.1, goal: 0.1}	
l_joint: {trajectory: 0.1, goal: 0.1}	
u_joint: {trajectory: 0.1, goal: 0.1}	
r_joint: {trajectory: 0.1, goal: 0.1}	
b_joint: {trajectory: 0.1, goal: 0.1}	
t_joint: {trajectory: 0.1, goal: 0.1}	
stop_trajectory_duration: 0.5	若电脑性能不足，适当加大此参数
state_publish_rate: 25	状态发布频率
action_monitor_rate: 10	行动完成情况反馈频率

文件路径：……m_mh24\m_mh24\launch\m_mh24_gazebo_with_workpiece.launch

代码片段	解释
<!-- joint trajectory controller -->	
<rosparam file="$(find m_mh24_gazebo)/config/joint_trajectory_controller.yaml" command="load"/>	
<node	
ns="/m_mh24"	
name="trajectory_controller_spawner"	
pkg="controller_manager"	
type="spawner"	
respawn="false"	
output="screen"	
args="arm_trajectory_controller"/>	

d) FollowJointTrajectory

由 Moveit 提供的 FollowJointTrajectory 类型的控制器其作用是向 JointTrajectoryController 类型的控制器发送目标轨迹点处的各关节状态数据。

第一步：编写控制器参数配置文件。

文件路径：……m_mh24\m_mh24_moveit_config\config\controllers.yaml

代码片段	解释
controller_manager_ns: controller_manager	
#action namespace: name/action_ns	#为注释
controller_list:	
- name: m_mh24/arm_trajectory_controller	可有多个不同名的控制器存在，与不同关节轨迹控制器配对使用。 配对方式是 FollowJointTrajectory 类型的控制器作为用户与作为服务器的 trajectory_controller 以行动方式沟通。
action_ns: follow_joint_trajectory	
type: FollowJointTrajectory	
default: true	
joints:	
- s_joint	
- l_joint	
- u_joint	
- r_joint	
- b_joint	
- t_joint	

第二步：上传参数配置到 ROS 参数服务器。

文件路径：……m_mh24\m_mh24_moveit_config\launch\ m_mh24_moveit_controller_manager.launch

代码片段	解释

<!-- loads joint controllers configrations to the parameter server -->	
<rosparam file="$(find m_mh24_moveit_config)/config/controllers.yaml"/>	#为注释

第三步：更改 move_group 节点设置以启动控制器

文件路径：……m_mh24\m_mh24_moveit_config\launch\move_group.launch

代码片段	解释
<!-- move_group settings --> <arg name="allow_trajectory_execution" default="true"/> <arg name="fake_execution" default="false"/>	更改 move_group 节点的设置，确认允许轨迹执行，并禁用虚假执行。（move_group 节点已经包含了启用控制器的功能，所以不需要再加一个节点以启用此控制器了）

e) 编写总启动文件

总启动文件文件路径如下：……m_mh24\m_mh24_gazebo\launch\m_mh24_bringup_moveit.launch

```
代码片段
<launch>
   <!-- startup simulate world with the robot and workpiece -->
  <include file="$(find m_mh24)/launch/m_mh24_gazebo_with_workpiece.launch" />

   <!-- load moveit controller and ros controllers -->
   <include file="$(find m_mh24_moveit_config)/launch/m_mh24_moveit_controller_manager.launch" />

   <!-- launch moveit -->
   <include file="$(find m_mh24_moveit_config)/launch/moveit_planning_execution.launch"/>

   <!-- launch rviz -->
   <include file="$(find m_mh24_moveit_config)/launch/moveit_rviz.launch">
      <param name="config" value="true"/>
   </include>
</launch>
```

f) 编译功能包

打开新终端，输入如下命令

```
cd ~/catkin_ws/
catkin_make   #编译 catkin 工作空间
```

编译完成后，在终端输入以下命令就可以启动仿真程序。

```
roslaunch m_mh24_gazebo m_mh24_bringup_moveit.launch
```

启动仿真程序后出现两个界面，一个 Moveit 界面（图 18）和一个 Gazebo 仿真界面（图 19）。

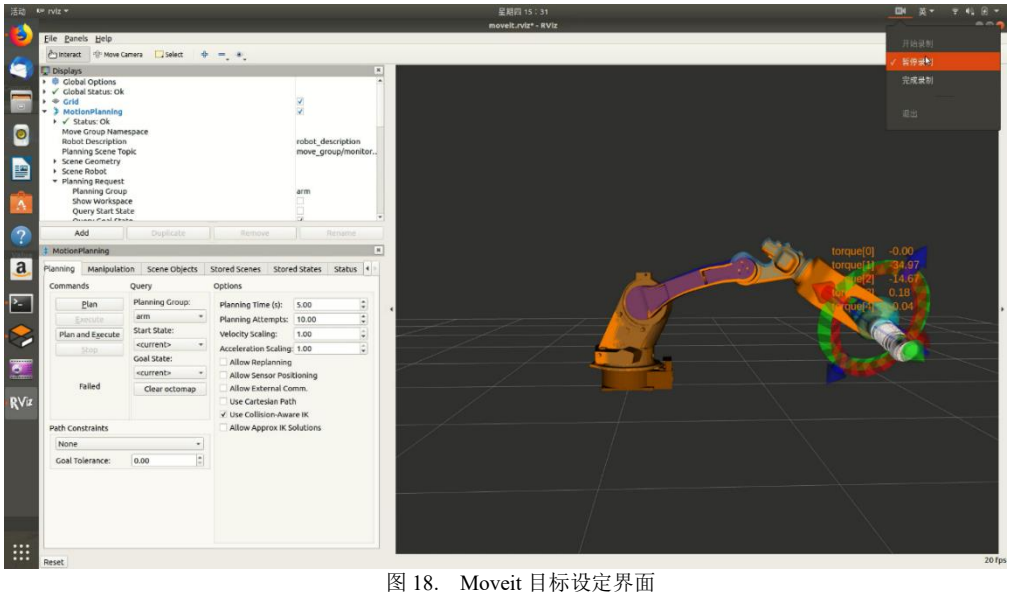


图 18. Moveit 目标设定界面

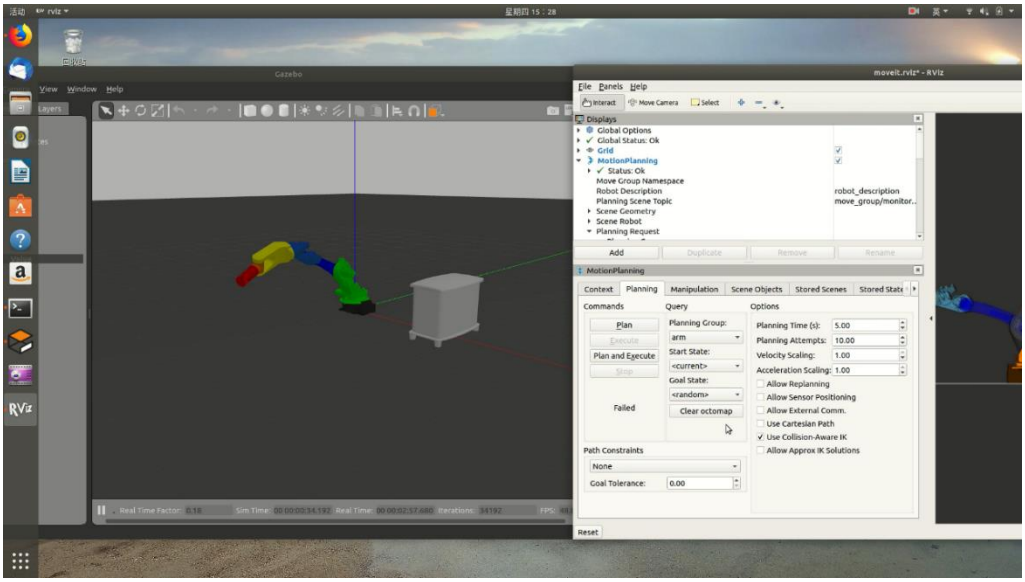


图 19. gazebo 仿真界面

仿真运行实例视频参见网址：

> 百度网盘链接：https://pan.baidu.com/s/1WJx3cYXOpdEbxMX0lzY93A
> 提取码：216l

2.3 在功能包 m_mh24 中添加更多功能

2.3.1 添加 kinect 传感器得到周围环境的点云数据

如果需要在机器人的轨迹规划功能中加入避障功能，首先需要让机器人了解周围的环境，总的来说有两种方法，一种是预先输入周围障碍物的三维信息，一种是通过传感器实时感知周围环境。在 Moveit 中也提供了这两种方法，第一种是直接添加障碍物的 mesh 文件如图 20，第二种是通过传感器获得周围的点云地图或者深度图像，如图 21。

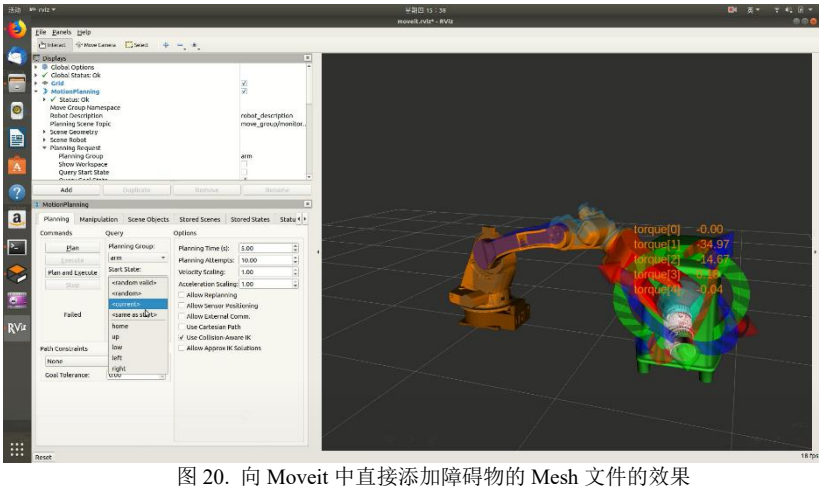


图 20. 向 Moveit 中直接添加障碍物的 Mesh 文件的效果

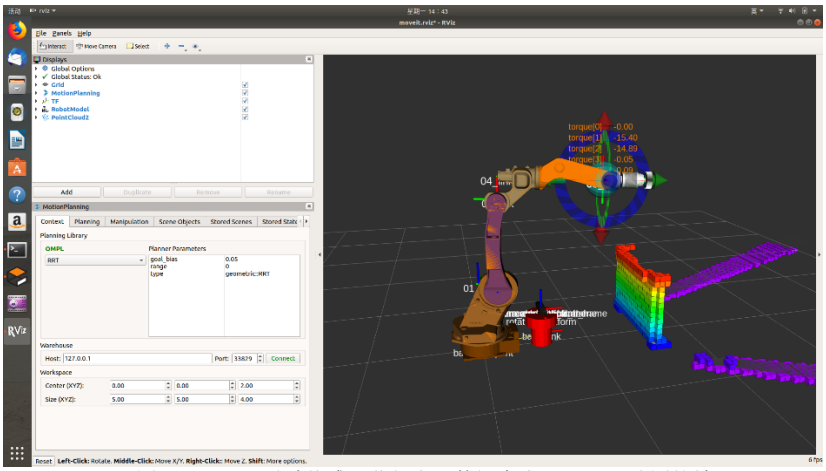


图 21. Moveit 通过传感器获得点云数据生成 Octomap 地图的效果

本教程中采用的传感器模型是 kinect（微软公司的一款三维体感相机），moveit 可将其传回的点云数据处理成 octomap 地图，其形态如图中彩色小方块所示，小方块的边长约小则地图越精细。

使用 kinect 感知周边障碍物并生成三维地图的步骤如下：

第一步：在 Gazebo 虚拟世界中加载传感器模型

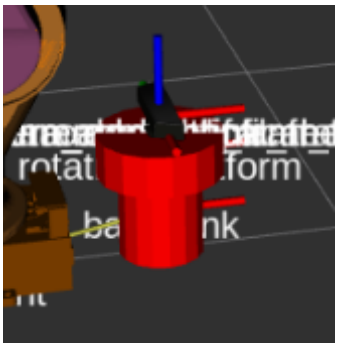


图 22. 放在旋转平台上的 Kinect 传感器模型

本教程中的传感器由三个部分组成，最上方是 kinect 传感器的官方模型，中部是旋转平台，下部是支撑柱。这样设计是为了能旋转传感器以获得 360 度范围内的地图数据。

官方 kinect 模型是从 ROS 提供的经典移动机器人案例 Turtelbot 中提取出来的，为将其与旋转平台的模型连接，这里用到了 xacro 宏文件，以下是文件示例。

文件路径：……m_mh24\m_mh24\urdf\rotating_kinect.urdf.xacro

代码片段	解释
<robot name="rotating_kinect" xmlns:xacro="http://ros.org/wiki/xacro">	包含 kinect 模型宏文件
<xacro:include filename="$(find m_mh24)/urdf/kinect.urdf.xacro"/>	
<xacro:property name="b_h" value="0.2"/>	宏文件中可定义一些常量，便于后期修改
<xacro:property name="p_h" value="0.1"/>	
<!--rotating platform-->	在宏文件中与 urdf 一样，可添加连杆与关节标签
<link name="world"/>	
<joint name="world_joint" type="fixed">	
<origin xyz="0.55 0 ${b_h*0.5}" rpy="0 0 0"/>	
<parent link="world"/>	
<child link="base_link" />	
</joint>	
……	省略部分定义
<link name="rotating_platform">	
<visual>	
<origin xyz="0 0 0" rpy="0 0 0"/>	
<geometry>	
<cylinder length="${p_h}" radius="0.15"/>	
</geometry>	
</visual>	
<collision>	
<origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>	
<geometry>	
<cylinder length="${p_h}" radius="0.15"/>	
</geometry>	
</collision>	
<inertial>	
<mass value="1" />	
<inertia ixx="0.00646" ixy="0.0" ixz="0.0" iyy="0.00646" iyz="0.0" izz="0.01125" />	
</inertial>	
</link>	
<sensor_kinect parent="rotating_platform"/>	将包含的 kinect 宏文件中定义的“sensor_kinect”模型连接上，并传递参数“rotating_platform”为父连杆名称
</robot>	

第二步：为传感器模型加载必要的控制器

此处只有一个关节需要控制，所以关节控制器只需要一个，即 position_controllers/JointPositionController 类型的控制器，此外还需要发布连杆坐标系以及转换矩阵的 robot_state_publisher 和发布实时关节状态的 joint_state_publisher。设置步骤与之前机械臂设置基本相同，不再赘述，但因为此时涉及到同一世界中的多个机器人，在名字空间的设置上有几点需要注意。

Gazebo 插件路径文件：……m_mh24\m_mh24\urdf\rotating_kinect.urdf.xacro

参数配置文件路径：……m_mh24\m_mh24\config\rotating_kinect_control.yaml

控制器启动文件路径：……m_mh24\m_mh24\launch\m_mh24_gazebo_with_workpiece_kinect.launch

代码片段	解释
<launch>	
<include file="$(find m_mh24)/launch/m_mh24_gazebo_with_workpiece.launch"/>	
<group ns="rk">	使用<group>标签为一组参数和节点添加名字空间，添加后设
<!-- spawn robot in gazebo -->	
<param name="robot_description" command="$(find xacro)/xacro.py $(find	

m_mh24)/urdf/rotating_kinect.urdf.xacro" />	置的参数名称、发布的话题等都会被包括到设置的名字空间内
<rosparam file="$(find m_mh24)/config/rotating_kinect_control.yaml" command="load"/>	
<node	
name="spawn_rk_model"	
pkg="gazebo_ros"	
type="spawn_model"	
args="-urdf -param /rk/robot_description -model rotating_kinect"	需要注意的是，不同名字空间内 robot_state_publisher 所发布的 tf 树，最终会被归集到一起。所以所有的连杆和关节（无论是否在同一机器人中）都必须有不同的名字。但可以设置同名的“world”连杆，以将所有连杆坐标系放置在同一个世界树中。
respawn="false"	
output="screen" />	
<node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">	
<param name="publish_frequency" type="double" value="50.0" />	
</node>	
<node pkg="joint_state_publisher" type="joint_state_publisher" name="joint_state_publisher">	
<param name="use_gui" value="true"/>	
</node>	
</group>	
</launch>	

第三步：确定传感器模型基坐标系到机械臂模型极坐标系的转换矩阵

将传感器安置在实物机器人上时，因为依靠测量几何参数所得的传感器坐标并不准确，所以都应该进行标定操作。针对 kinect 的标定操作属于手眼标定的一种，需要得到的数据是从机器人基座到传感器的坐标变换矩阵树。可参考的标定功能包网址如下：

https://github.com/IFL-CAMP/easy_handeye.git
https://github.com/arushk1/visp_hand2eye_calibration.git

得到数据后按以下格式写在 launch 文件中：

代码片段	解释
<!-- publish the transform matrix between the robot world link and the kinect world link -->	
<!-- args="x y z qx qy qz qw frame_id child_frame_id period_in_ms"-->	参数格式
<node pkg="tf" type="static_transform_publisher" name="kinect_tansform" args="0.55 0 0.1 0 0 0 world rk/world 100"/>	

但在本教程中，由于所有的虚拟模型都有确定的 urdf 模型，所以不用再进行标定过程。只需要在机械臂和 kinect 传感器的 urdf 文件中设置同名的“world”连杆即可。

第四步：修改 Moveit 设置文件以接收并处理传感器数据

这一部分主要是使 moveit 的 octomap 生成器订阅 kinect 发布的深度点云数据，并设置一些 octomap 的基本参数。需要用到的文件同样是一个参数配置文件和一个启动文件。

文件路径：……m_mh24\m_mh24_moveit_config\config\sensors_kinect.yaml

代码片段
sensors:
- filtered_cloud_topic: filtered_cloud
max_range: 5.0
max_update_rate: 1.0
padding_offset: 0.1
padding_scale: 1.0
point_cloud_topic: /rk/camera/depth/points
point_subsample: 1
sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater

文件路径：……m_mh24\m_mh24_moveit_config\launch\ m_mh24_moveit_sensor_manager.launch

```
代码片段
<?xml version="1.0"?>
<launch>
   <rosparam command="load" file="$(find m_mh24_moveit_config)/config/sensors_kinect.yaml"/>
   <param name="octomap_frame" type="string" value="world" />
   <param name="octomap_resolution" type="double" value="0.05" />
   <param name="max_range" type="double" value="5.0" />
</launch>
```

至此整个仿真实验平台的框架搭建完成，完成代码和示例视频参见 Github 仓库：

```
https://github.com/lindaqqiu/m_mh24
```

2.3.2 未来目标

如果想在此基础上增加功能，需要了解 moveit 编程方面的内容，以下将对这部分的基本思想做一些说明。

2.3.2.1 深入理解 Moveit 是如何完成其功能的

Moveit 中最重要的节点是 move_group，通过这个节点，Moveit 建起了运动规划库，用户界面，机器人硬件接口和 ROS 参数服务器之间的桥梁。在本教程的进行过程中，已经体验了如何搭建起这个系统，之后需要对其中源码深入学习，才能优化系统和开发各种新功能。主要需要学习的部分是运动规划库和用户界面。运动规划库中包含两个部分，即路径规划求解器和运动学求解器，希望掌握能求解设定多路径途径点时的机械臂轨迹的求解器的方法。在用户界面中主要需要学习的是如何设定一系列路径途径点并传送给路径规划器，比如如何将所需的加工轨迹传递给路径规划器。

1） 利用路径规划求解器求解机械臂末端路径

Moveit 中集成有 OMPL 库，其中包含很多种不同的路径规划求解器（图 23），最常用的一种是 RRT 也就是快速随机树方法。OMPL 库中快速随机树方法的源码如下：

```
    https://bitbucket.org/ompl/ompl/src/default/src/ompl/geometric/planners/rrt/src/RRT.cpp
    https://bitbucket.org/ompl/ompl/src/default/src/ompl/geometric/planners/rrt/RRT.h
```

```
default_planner_config: RRT
planner_configs:
  - SBL
  - EST
  - LBKPIECE
  - BKPIECE
  - KPIECE
  - RRT
  - RRTConnect
  - RRTstar
  - TRRT
  - PRM
  - PRMstar
  - FMT
  - BFMT
  - PDST
  - STRIDE
  - BiTRRT
  - LBTRRT
  - BiEST
  - ProjEST
  - LazyPRM
  - LazyPRMstar
  - SPARS
  - SPARStwo
```

图 23. Moveit 中可用的运动规划器列表

如果需要在 Moveit 中添加自己的规划器，总的来说分为三步，即先从源码安装 Moveit 和 OMPL，在 OMPL 库中对应路径添加自己算法的.cpp 和.h 文件，修改 Moveit 中的代码以使其能识别新规划器的名字，可参考的教程如下：

https://answers.ros.org/question/296238/custom-state-sampler-in-moveit/	英文
https://answers.ros.org/question/298345/different-ways-to-implement-custom-planners-with-moveit/	英文
https://www.zhihu.com/question/293386019/answer/486199063	中文

2） 利用运动学求解器求解各关节运动轨迹

而在运动学求解器方面，最常用的是 KDL（kinematics and dynamitic library）插件（图 24），是由 Orocos KDL 包提供的数字逆运动学求解器。其源码参考如下：

https://github.com/orocos/orocos_kinematics_dynamics.git	
https://github.com/orocos/orocos_kinematics_dynamics/tree/master/orocos_kdl/src	类定义
https://github.com/orocos/orocos_kinematics_dynamics/tree/master/orocos_kdl/examples	运用示例

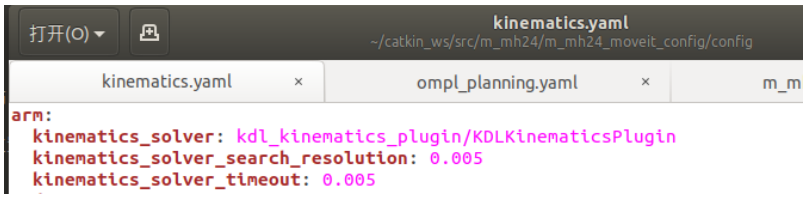


图 24. Moveit 中采用的运动学求解器

3） 实现设定多末端路径途经点的准静态轨迹规划

这一部分的内容是关于用户界面的，这个界面可以是一个图形化窗口，也可以是一个程序，目标是为了向求解器传递机械臂目标位姿的信息。当我们提到目标位姿，首先应该确定的问题是目标位姿将以何种形式被描述，现在其描述方式可以被分为两个大类，其一是关节空间，其二是笛卡尔空间，其区别是使用的坐标系不同，但共同点是都包括位置、速度、加速度三种变量。关节空间，简单来讲就是某关节转动的角度或者移动的距离；笛卡尔空间，就是某关节坐标系在基坐标系里的坐标表示。这样一来，以关节空间描述目标位姿时，很难控制机械臂末端在基坐标系中的实际轨迹，所以当需要按照特定轨迹加工物体时，常采用笛卡尔空间描述。

Moveit 提供了一些函数来帮助实现将多个机械臂末端目标途经点输入给求解器的工作。只需要提供途经点的位置信息（x、y、z、r、p、y 六个元素），接下来 Moveit 会利用 KDL 对途径点进行差补计算，确定点与点之间需要的速度向量，利用雅各比矩阵求运动学逆解，解出各个关节空间内的位置、速度等信息，再交由 OMPL 路径规划库，看其能否找到符合这一系列速度与位置要求的解，如果可以，则驱动机械臂执行这一路径。

下面是 Moveit 提供的设定笛卡尔空间目标轨迹的函数说明网址：

http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/move_group_interface/move_group_interface_tutorial.html#cartesian-paths

其它关于设定目标点的 Moveit 接口的查询地址：

https://github.com/ros-planning/moveit_ros/blob/bc153b0e8ee8c2e9a740b593874cbb95aa6fcd46/planning_interface/move_group_interface/src/move_group.cpp

一个应用此函数实现控制机械臂按圆轨迹运行的 Python 代码示例：

```
https://github.com/ps-micro/PROBOT_Anno/blob/master/probot_demo/scripts/moveit_circle_demo.py
```

一个完整的能完成多目标点设定的图形化用户界面示例：

http://wiki.ros.org/moveit_cartesian_plan_plugin