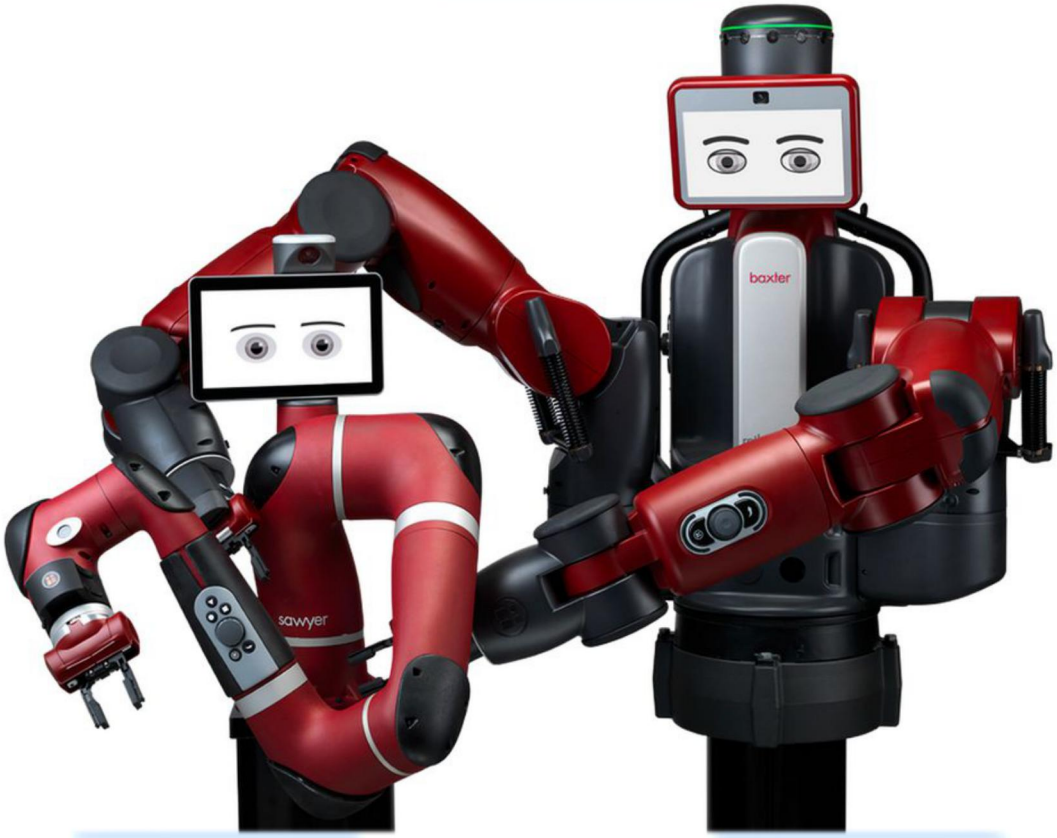
瑞森可智能协作机器人
SAWYER 开发指南



湖南瑞森可机器人科技有限公司

2020 年版

地址： 长沙市岳麓区桐梓坡西路 348 号科力远园区
国家工程中心二楼

电话： 0731-89872400

瑞森可

前言

湖南瑞森可机器人科技有限公司是一家专业从事智能机器人技术研究与应用服务的解决方案提供商。

公司智能机器人产品及应用解决方案已经成功服务于全球 20 多个国家的 500 多个客户和合作伙伴，并形成了具有自主知识产权的多项核心技术。公司秉持“创新价值、开放共享”的合作理念，与麻省理工、GE、丰田、清华大学、中科院等全球知名高校、企业和科研机构展开合作，逐渐形成覆盖生产制造、教育普及、科学研究、集成应用、金融服务等多层级多形态协同发展的“瑞森可+”智能机器人产业合作生态。

公司已经拥有美国原 Rethink Robotics（全球协作机器人鼻祖）旗下智能双臂协同机器人 Baxter 完整的品牌及专利技术，并与德国最大的机器人与自动化公司之一——Hahn Group 建立长期战略协作，共同承接和发展 Rethink 品牌和技术。产品配置了可用于先进机器人学领域研究开发的软件开发工具包（SDK），定位于为全球用户提供机器人研究、开发与应用平台。

本指南为基于 Sawyer 平台开展科研教学的用户提供基础的学习指南。指南主要介绍了 ROS 基础、MoveIt 运动规划、Sawyer SDK 配置与应用、Sawyer 应用案例等内容。

第一章：ROS 基础。介绍了 Python 入门、Linux 基本命令、ROS 简介，ROS 安装，ROS 基础，ROS 仿真。

第二章：MoveIt 运动规划。介绍 MoveIt 运动规划的基本原理和系统构成，运动规划接口，Sawyer 运动规划配置和单臂运动规划示例。

第三章：Sawyer SDK。介绍如何使 Sawyer 进入研究版，如何进行官网界面及下载 SDK，工作环境搭建，Sawyer 的 SDK 简单示例详解。

第四章：Sawyer 应用案例。分析一些实际的应用案例，介绍 ROS 和 Sawyer 综合应用。

附录一：Sawyer 硬件参数。介绍 Sawyer 机械臂及其他硬件参数。

本指南在编写过程中，参阅了国内外相应资料，在此向原作者表示感谢。

目录

第一章 ROS 基础

1.1 Python 入门

本节将简要介绍如何用 Python 语言编写简单的脚本程序。确保学员能够使用 Python 控制 Sawyer。

1.1.1 什么是 Python

Python 是著名的“龟叔”Guido van rossum 在 1989 年圣诞节期间，为了打发无聊的圣诞节而编写的一个编程语言。

现在，全世界差不多有 600 多种编程语言，但流行的编程语言也就那么 20 来种。

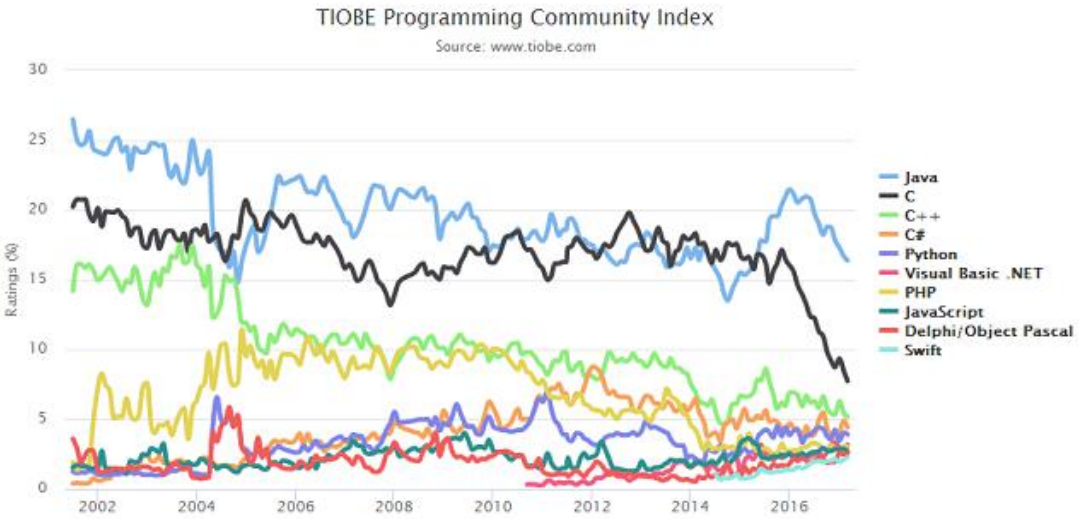


图 1-1 近 10 年主流语言指数（TIOBE）

据 TIOBE2017 年的统计，Python 在所有编程语言中位列第 5，可见还是非常流行的一种语言。不同于 C 语言，Python 是一种解释型语言，执行时机器会逐行将其翻译成 CPU 能理解的机器码，因此比较耗时（但在现代计算机里，这种差距显得不那么重要了）。另一个好消息是，Ubuntu 里已经自带了 Python（包含 2.x 和 3.x 两个版本），因此无需安装了。

1.1.2 编写第一个 Python 程序

Python 支持两种方法写程序：交互式命令行和文件。下面就以这两种方式写一个最简单的 Python 程序。

1）交互式命令行

打开系统终端，敲入命令 python 进入 Python 交互程序

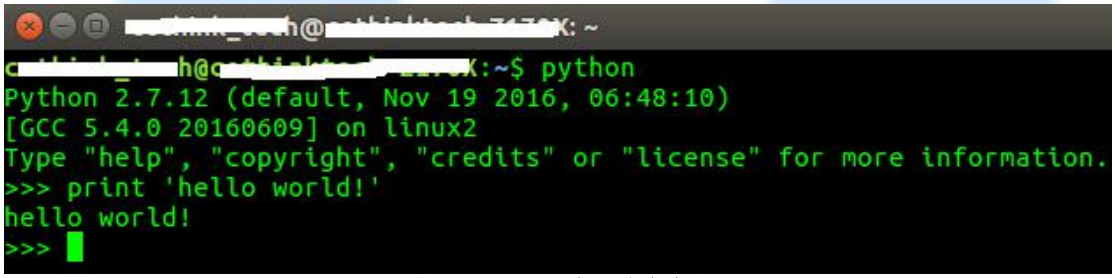


图 1-2 Python 交互式命令行

2）文本代码

新建一个空白文档，命名为 hello.py，打开后输入：

图 1-3 Python 文本方式编程

然后打开终端，敲入命令：

```
$ python hello.py
```

这段代码实现了使用 Python 的 print 函数向屏幕输出字符串。一般来说考虑到程序的重用，我们都使用文本的方式来编写代码。

1.1.3 Python 的基本数据类型和逻辑语句

与 C 语言不同，Python 是动态语言，无需对变量做类型定义。Python 支持传统的数据类型：整数、浮点数、字符串、布尔值和空值，此外 Python 还内置了特别的数据类型：列表、元组、字典和集合。

表 1-1 Python 数据类型

数据类型	整数	浮点数	字符串	布尔值	空值	列表	元组	字典	集合
示例	-10	1.23	‘cothink’	True	None	[1, ‘a’]	(1, ‘a’)	{‘a’:1}	Set([1,2])

Python 对条件判断和循环的支持与其他语言也大同小异：

```
# if 条件判断
 if <条件判断 1>:
      <执行 1>
elif <条件判断 2>:
      <执行 2>
elif <条件判断 3>:
      <执行 3>
else:
      <执行 4>
```

循环结构有两种，一种是 for 循环：

```
# for 循环
names = ['Michael', 'Bob', 'Tracy']
for name in names:
      print name
```

一种是 while 循环：

```
# while 循环
sum = 0
n = 99
while n > 0:
      sum = sum + n
      n = n - 2
print sum
```

1.1.4 理解 Python 程序及逻辑含义

在编程语言的学习中，阅读代码是非常重要的方式。从中可以熟悉该语言的基本用法，更

可以学习到别人的一些小技巧。这一节将给出一段 Sawyer 自带的 Python 脚本，请逐句分析代码，然后给出整个程序要实现的功能，最后上机测试其是否符合预期。

```
#!/usr/bin/env python

import argparse
import random

import rospy

import intera_interface

from intera_interface import CHECK_VERSION


class Wobbler(object):

    def __init__(self):
        """
        'Wobbles' the head
        """
        self._done = False
        self._head = intera_interface.Head()

        # verify robot is enabled
        print("Getting robot state... ")
        self._rs = intera_interface.RobotEnable(CHECK_VERSION)
        self._init_state = self._rs.state().enabled
        print("Enabling robot... ")
        self._rs.enable()
        print("Running. Ctrl-c to quit")

    def clean_shutdown(self):
        """
        Exits example cleanly by moving head to neutral position and
        maintaining start state
        """
        print("\nExiting example...")
        if self._done:
            self.set_neutral()

    def set_neutral(self):
        """
        Sets the head back into a neutral pose
        """
        self._head.set_pan(0.0)

    def wobble(self):
        self.set_neutral()
        """
        Performs the wobbling
        """
        command_rate = rospy.Rate(1)
        control_rate = rospy.Rate(100)
        start = rospy.get_time()
        while not rospy.is_shutdown() and (rospy.get_time() - start < 10.0):
            angle = random.uniform(-2.0, 0.95)
```

```
                while (not rospy.is_shutdown() and
                       not (abs(self._head.pan() - angle) <=
                            intera_interface.HEAD_PAN_ANGLE_TOLERANCE)):
                    self._head.set_pan(angle, speed=0.3, timeout=0)
                    control_rate.sleep()
                command_rate.sleep()

            self._done = True
            rospy.signal_shutdown("Example finished.")


def main():
    """RSDK Head Example: Wobbler
    Nods the head and pans side-to-side towards random angles.
    Demonstrates the use of the intera_interface.Head class.
    """
    arg_fmt = argparse.RawDescriptionHelpFormatter
    parser = argparse.ArgumentParser(formatter_class=arg_fmt,
                                     description=main.__doc__)
    parser.parse_args(rospy.myargv()[1:])

    print("Initializing node... ")
    rospy.init_node("rsdk_head_wobbler")

    wobbler = Wobbler()
    rospy.on_shutdown(wobbler.clean_shutdown)
    print("Wobbling... ")
    wobbler.wobble()
    print("Done.")

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

1.2 Linux 基本命令

Ubuntu（乌班图）是一个以桌面应用为主的 Linux 操作系统，Ubuntu 基于 Debian GNU/Linux，支持 x86、amd64（即 x64）和 ppc 架构，由全球化的专业开发团队（Canonical Ltd）打造的开源 GNU/Linux 操作系统。

下面介绍 Linux 系统一些常用命令。

1.2.1 文件和目录

1）ls：显示文件或目录

-l 列出文件详细信息 l（list）

-a 列出当前目录下所有文件及目录，包括隐藏的 a（all）

以易读的方式显示文件大小(显示为 MB,GB...)：

```
$ ls -lh
-rw-r----- 1 ramesh team-dev 8.9M Jun 12 15:27 arch-linux.txt.gz
```

以最后修改时间升序列出文件：

```
$ ls -ltr
```

在文件名后面显示文件类型：

```
$ ls -F
```

2）mkdir：创建目录

-p 创建目录，若无父目录，则创建 p(parent)

在 home 目录下创建一个名为 test 的目录：

```
$ mkdir ~/test
```

使用-p 选项可以创建一个路径上所有不存在的目录：

```
$ mkdir -p dir1/dir2/dir3/dir4/
```

3）cd：切换目录

切换到工作区间 catkin_ws 文件夹中：

```
~$ cd catkin_ws
~/catkin_ws$
```

4）cp：复制

复制文件 1 到文件 2，并保持文件的权限、属主和时间戳：

```
$ cp -p file1 file2
```

复制 file1 到 file2，如果 file2 存在会提示是否覆盖：

```
$ cp -i file1 file2
```

5）mv：移动或重命名

将文件名 file1 重命名为 file2，如果 file2 存在则提示是否覆盖：

```
$ mv -i file1 file2
```

注意如果使用-f 选项则不会进行提示。

-v 会输出重命名的过程，当文件名中包含通配符时，这个选项会非常方便：

```
$ mv -v file1 file2
```

6）rm：删除文件

-r 递归删除，可删除子目录及文件

-f 强制删除

删除文件前先确认：

```
$ rm -i filename.txt
```

在文件名中使用 shell 的元字符会非常有用。删除文件前先打印文件名并进行确认：

```
$ rm -i file*
```

递归删除文件夹下所有文件，并删除该文件夹：

```
$ rm -r example
```

7）rmdir：删除空目录

删除一个叫做 dir1 的目录：

```
$ rmdir dir1
```

8）pwd：显示当前目录

显示 catkin_ws 的当前目录：

```
~/catkin_ws$ pwd
/home/cothink/catkin_ws
```

9）ln：创建链接文件

创建一个指向文件或目录的软链接：

```
$ ln -s file1 lnk1
```

创建一个指向文件或目录的物理链接：

```
$ ln file1 lnk1
```

10）locate：定位文件

locate 命名可以显示某个指定文件（或一组文件）的路径，它会使用由 updatedb 创建的数据库。

下面的命令会显示系统中所有包含 crontab 字符串的文件：

```
$ locate crontab
/etc/anacrontab
/etc/crontab
/usr/bin/crontab
/usr/share/doc/cron/examples/crontab2english.pl.gz
/usr/share/man/man1/crontab.1.gz
/usr/share/man/man5/anacrontab.5.gz
/usr/share/man/man5/crontab.5.gz
/usr/share/vim/vim72/syntax/crontab.vim
```

11）whatis：显示命令描述信息

wathis 显示 ls 命令的描述信息：

```
$ whatis ls
ls              (1)    - list directory contents

$ whatis ifconfig
ifconfig      (8)              - configure a network interface
```

1.2.2 文本处理

1）cat：查看文件内容

你可以一次查看多个文件的内容，下面的命令会先打印 file1 的内容，然后打印 file2 的内容：

```
$ cat file1 file2
```

-n 命令可以在每行的前面加上行号：

```
$ cat -n /etc/logrotate.conf
     /var/log/btmp {
          missingok
          monthly
          create 0660 root utmp
          rotate 1
     }
```

2）more, less：分页显示文本文件内容

查看一个长文件的内容：

```
$ more file1
```

类似于 more 命令，less 允许在文件中和正向操作一样的反向操作:

```
$ less file1
```

3) head, tail: 显示文件头、尾内容

查看一个文件的前两行：

```
$ head -2 file1
```

查看一个文件的最后两行：

```
$ tail -2 file1
```

4) find: 在文件系统中搜索某文件

查找指定文件名的文件(不区分大小写)：

```
$ find -iname "MyProgram.c"
```

对找到的文件执行某个命令:

```
$ find -iname "MyProgram.c" -exec md5sum {} \;
```

查找 home 目录下的所有空文件：

```
$ find ~ -empty
```

5) wc: 统计文本中行数、字数、字符数

统计 test.py 文本中的字数：

```
$ wc test.py
    54    165    1499    test.py
```

6) grep: 在文本文件中查找某个字符串

在文件中查找字符串(不区分大小写)：

```
$ grep -i "the" demo_file
```

输出成功匹配的行，以及该行之后的三行：

```
$ grep -A 3 -i "example" demo_text
```

在一个文件夹中递归查询包含指定字符串的文件：

```
$ grep -r "ramesh" *
```

显示所有名称中包含 "httpd" 字样的 rpm 包：

```
$ rpm -qa | grep httpd
```

7) vim: 编译文档

打开文件并跳到第 10 行：

```
$ vim +10 filename.txt
```

打开文件跳到第一个匹配的行：

```
$ vim +/search-term filename.txt
```

以只读模式打开文件：

```
$ vim -R /etc/passwd
```

1.2.3 系统管理

1) who：显示在线登陆用户

```
$ who
cothink : 0          2018-05-20 15:42(:0)
```

用 whoami 显示当前操作用户：

```
$ whoami
cothink
```

2) hostname：显示主机名

显示当前主机名：

```
$ hostname
cothinkTab3
```

3) uname：显示系统信息

uname 可以显示一些重要的系统信息，例如内核名称、主机名、内核版本号、处理器类型之类的信息：

```
$ uname -a
Linux john-laptop 2.6.32-24-generic #41-Ubuntu SMP Thu Aug 19 01:12:52 UTC 2010 i686 GNU/Linux
```

4) top：动态显示当前耗费资源最多进程信息

top 命令会显示当前系统中占用资源最多的一些进程（默认以 CPU 占用率排序）：

```
$ top
top - 14:34:02 up 12 min, 2 users, load average: 0.06, 0.24, 0.21
Tasks: 199 total, 2 running, 198 sleeping, 0 stoped, 0 bombie
%Cpu(s): 5.7 us …
…
```

如果只想显示某个特定用户的进程，可以使用-u 选项：

```
$ top -u oracle
```

5) ps：显示瞬间进程状态

ps 命令用于显示正在运行中的进程的信息，ps 命令有很多选项，这里只列出了几个，查看当前正在运行的所有进程：

```
$ ps -ef | more
```

以树状结构显示当前正在运行的进程，H 选项表示显示进程的层次结构：

```
$ ps -efH | more
```

6) du：查看目录大小

带有单位显示目录信息：

```
$ du -h /home
```

估算目录 dir1 已经使用的磁盘空间：

```
$ du -sh dir1
```

以容量大小为依据依次显示文件和目录的大小：

```
$ du -sk * | sort -rn
```

7) df：查看磁盘大小

显示文件系统的磁盘使用情况，默认情况下 df -k 将以字节为单位输出磁盘的使用量：

```
$ df -k
Filesystem            1K-blocks      Used Available Use% Mounted on
/dev/sda1             29530400   3233104  24797232  12% /
/dev/sda2            120367992  50171596  64082060  44% /home
```

使用-h 选项可以以更符合阅读习惯的方式显示磁盘使用量：

```
$ df -h
Filesystem            Size   Used  Avail Capacity  iused    ifree %iused  Mounted on
/dev/disk0s2          232Gi   84Gi  148Gi    37% 21998562 38864868   36%   /
devfs                 187Ki  187Ki    0Bi   100%     648        0  100%   /dev
map -hosts              0Bi    0Bi    0Bi   100%       0        0  100%   /net
map auto_home           0Bi    0Bi    0Bi   100%       0        0  100%   /home
/dev/disk0s4          466Gi   45Gi  421Gi    10%  112774 440997174    0%
/Volumes/BOOTCAMP
//app@izenesoft.cn/public  2.7Ti  1.3Ti  1.4Ti  48%  0 18446744073709551615  0%
/Volumes/public
```

使用-T 选项显示文件系统类型：

```
$ df -T
Filesystem     Type    1K-blocks      Used Available Use% Mounted on
/dev/sda1      ext4     29530400   3233120  24797216  12% /
/dev/sda2      ext4    120367992  50171596  64082060  44% /home
```

8) ifconfig：查看网络情况

ifconfig 用于查看和配置 Linux 系统的网络接口。

查看所有网络接口及其状态：

```
$ ifconfig -a
```

使用 up 和 down 命令启动或停止某个接口：

```
$ ifconfig eth0 up
$ ifconfig eth0 down
```

9) ping：测试网络连通

ping 一个远程主机，只发 5 个数据包：

```
$ ping -c 5 gmail.com
```

10) man：查看手册页面

显示某个命令的 man 页面：

```
$ man crontab
```

有些命令可能会有多个 man 页面，每个 man 页面对应一种命令类型：

```
$ man SECTION-NUMBER commandname
```

man 页面一般可以分为 8 种命令类型

1、用户命令
2、系统调用
3、c 库函数
4、设备与网络接口
5、文件格式

6、游戏与屏保
7、环境、表、宏
8、系统管理员命令和后台运行命令

例如，我们执行 whatis crontab，你可以看到 crontab 有两个命令类型 1 和 5，所以我们可以通过下面的命令查看命令类型 5 的 man 页面：

```
$ whatis crontab
crontab (1)              - maintain crontab files for individual users (V3)
crontab (5)              - tables for driving cron

$ man 5 crontab
```

11) clear：清屏

清空屏幕：

```
$ clear
```

12) alias：对命令重命名

例如

```
$ alias showmeit="ps -aux"
```

另外解除使用

```
$ unaliax showmeit
```

13) kill：杀死进程

kill 用于终止一个进程。一般我们会先用 ps -ef 查找某个进程得到它的进程号，然后再使用 kill -9 进程号终止该进程。你还可以使用 killall、pkill、xkill 来终止进程：

```
$ ps -ef | grep vim
ramesh      7243    7222    9 22:43 pts/2       00:00:00 vim

$ kill -9 7243
```

1.2.4 打包压缩

1) gzip：压缩文件

压缩一个叫做 'file1'的文件：

```
$ gzip file1
```

最大程度压缩：

```
$ gzip -9 file1
```

2) tar：打包压缩

-c	归档文件
-x	压缩文件
-z	gzip 压缩文件
-j	bzip2 压缩文件
-v	显示压缩或解压缩过程 v(view)
-f	使用档名

例如只打包，不压缩：

```
$ tar -cvf /home/abc.tar /home/abc
```

打包，并用 gzip 压缩：

```
$ tar -zcvf /home/abc.tar.gz /home/abc
```

打包，并用 bzip2 压缩：

```
$ tar -jcvf /home/abc.tar.bz2 /home/abc
```

当然，如果想解压缩，就直接替换上面的命令 tar -cvf / tar -zcvf / tar -jcvf 中的“c” 换成“x” 就可以了。

3) rar：打包压缩

创建一个叫做 file1.rar 的包：

```
$ rar a file1.rar test_file
```

同时压缩'file1', 'file2'以及目录'dir1'：

```
$ rar a file1.rar file1 file2 dir1
```

解压 rar 包：

```
$ rar x file1.rar
或：
$ unrar x file1.rar
```

4) 其他压缩命令

压缩一个叫做 'file1' 的文件：

```
$ bzip2 file1
```

解压一个叫做 'file1.bz2'的文件：

```
$ bunzip2 file1.bz2
```

解压一个叫做 'file1.gz'的文件：

```
$ gunzip file1.gz
```

创建一个 zip 格式的压缩包：

```
$ zip file1.zip file1
```

几个文件和目录同时压缩成一个 zip 格式的压缩包：

```
$ zip -r file1.zip file1 file2 dir1
```

解压一个 zip 格式压缩包：

```
$ unzip file1.zip
```

1.2.5 关机与重启

1) shutdown

关闭系统并立即关机：

```
$ shutdown -h now
```

10 分钟后关机：

```
$ shutdown -h +10
```

重启：

```
$ shutdown -r now
```

重启期间强制进行系统检查：

```
$ shutdown -Fr now
```

2) 其他命令

系统重启：

```
$ reboot
```

系统注销：

```
$ logout
```

1.3 ROS 简介与安装

ROS 起源于 2007 年斯坦福大学人工智能实验室与机器人技术公司 Willow Garage 之间合作的个人机器人项目(Personal Robots Program)，2008 年之后就由 Willow Garage 来进行推动。ROS 是一种开源的机器人操作系统，或者说次级操作系统。它能提供类似操作系统所提供的功能，例如硬件抽象描述、底层驱动程序管理、共用功能的执行、程序间的消息传递、程序发行包管理，它也提供一些工具程序和库用于获取、建立、编写和运行多机整合的程序。

图 1-4 ROS 应用公司

1.3.1 ROS 简介

ROS 是一种分布式的处理框架，这使得可执行文件能够被独立设计，并且在运行时松散耦合。ROS 可以分成两层，下层是上述的操作系统层，上层则是广大用户群贡献的能够实现不同功能的功能包，例如机械臂运动规划、自主导航定位、传感器插件、仿真工具等。

ROS 的主要特点为：

1、精简与集成：ROS 建立的系统具有模块化的特点，各模块中的代码可以单独编译，而且编译使用的 CMake 工具可以很容易地实现精简的理念。ROS 集成了很多现在已经存在的开源库，例如 OpenCV、OpenRAVE、Player 等。

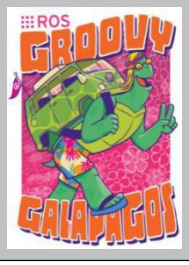
2、支持多种编程语言：为了满足不同编程者的需求，ROS 采用了语言中立性的框架结构，即不依赖某一种编程语言。ROS 支持的语言包括 C++、Python、Octave 和 LISP 等

3、免费并且开源：ROS 遵循 BSD 协议，对个人、商业应用以及修改都是免费的。

4、点对点设计：ROS 的点对点设计以及服务和节点管理器等机制可以分散由计算机视觉和语音识别等功能带来的实时计算压力。

从 2009 年 Willow Garage 开源 ROS 以来，ROS 已经发行了 10 多个版本。目前常用的是 Indigo 和 Kinetic，本指南选择 Indigo 版本作为 Sawyer 的 ROS 系统进行配置和讲解。

表 1-2 截止目前 ROS 历次发行版

版本	发布日期	海报	图标	EOL 日期
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame **(Recommended)**	May 23rd, 2016			May, 2021
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	Sep 4th, 2013			May, 2015
ROS GroovyGalapagos	December 31,2012			July, 2014

ROS Fuerte Turtle	April 23, 2012			--
ROS Electric Emys	August 30, 2011			--
ROS Diamondback	March 2, 2011			--
ROS C Turtle	August 2, 2010			--
ROS Box Turtle	March 2, 2010			--

1.3.2 Ubuntu 安装

Ubuntu 系统的安装方法有很多，在这里使用 U 盘作为启动盘来安装。

首先前往官网（http://www.ubuntu.org.cn/index_kylin）下载桌面版 Ubuntu14.04 的 ISO 文件。下载 ISO 烧录工具（http://unetbootin.github.io/）UNetbootin 并安装。

图 1-5. UNetbootin 烧录 Ubuntu

插入一个大于 2G 的 U 盘，打开 UNetbootin，设置好 Ubuntu 光盘的路径和 U 盘路径，点击确定，等待其烧录完成就可以了。

完成后将 U 盘插入待装机器，开机优先选择 U 盘启动。如图 1-6 所示，选择“Install Ubuntu”一项，然后按照提示一步一步安装即可。

图 1-6. 从 U 盘安装 Ubuntu

1.3.3 ROS 安装与配置

ROS 的安装与配置分为以下 6 个步骤。

1） 添加 sources.list

目的是为了配置 Ubuntu，使其能安装来自 packages.ros.org 的软件。

推荐使用国内的镜像以加快下载速度：

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

2） 添加 Keys

```
$ sudo apt-key adv --keyserver
hkp://pool.sks-keyservers.net --recv-key 421C365B-D9FF1F717815A3895523BAEEB01FA116
```

3） 安装

```
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full
```

4） 初始化 rosdep

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

5） 环境设置

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

6） 安装 rosinstall

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

至此，Indigo 版本的 ros 已经安装完成。

1.4 ROS 基础

ROS 可以运行在 Linux、Unix 和 Android 平台上（图 1-7），目前官方对 Linux 下的 Ubuntu 系统支持最好。

图 1-7. ROS 官方对操作系统平台的支持

1.4.1 ROS 基本概念

ROS 由不同的功能包组成，每个功能包提供一类问题的解决方案。ROS 在内部通过消息和服务的方式实现不同程序（节点）的通信。

功能包集(stack)：
功能包集是实现某种功能的多个功能包的集合，可提供更高级的功能。每一个功能包集都带有相关版本号，是 ROS 软件发布的主要形式。

功能包(package)：
功能包是 ROS 中组织软件的主要形式，可以编写代码并进行编译、执行等操作。一个功能包一般包含程序文件、编译描述文件、配置文件等。

节点(node)：
节点是一个可执行文件，多个节点可实现复杂的功能。程序文件只有转换为可执行文件，才可以在 ROS 中运行。

主题(Topic)和服务(Service)：
节点之间的通信方式主要包括主题和服务两种。主题只能实现节点之间的单向通信，而服务是双向通信，包括请求(request)和响应(response)。

消息(msg)：
消息指的是通信的具体内容，每一个消息都有一个固定的数据结构，支持标准的原始数据类型，包括整形、浮点型、布尔型等。

1.4.1.1 节点和主题

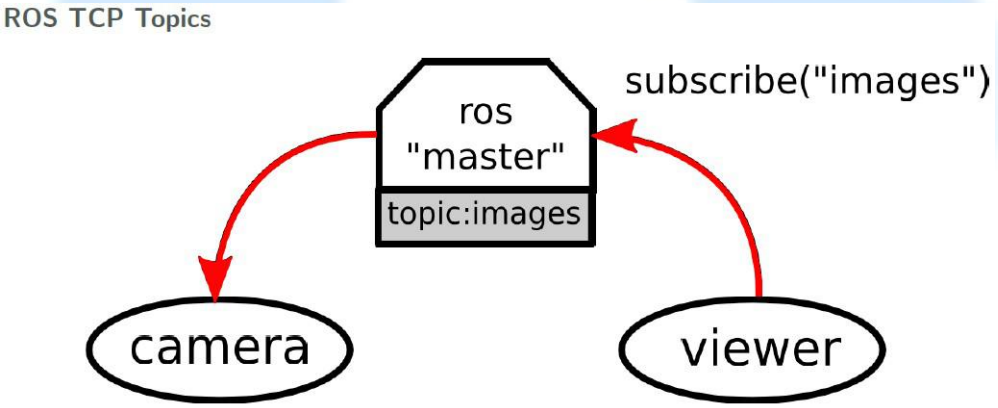


图 1-8 ros 节点通讯图

Nodes: 节点是 ROS 的基本可执行单元，如图 1-8 中的 camera,viewer；

Topois: topic 即主题，各 node 可以向 topic 发送消息，或者监听 topic 消息；
Messages: topic 消息类型，如 Float64，Vector；
Master: 为节点提供命名和注册服务，所以 node 都要向 master 注册；
当 nodes 通过 master 注册后可以直接用 TCP 协议通讯，如图 1-9 所示。

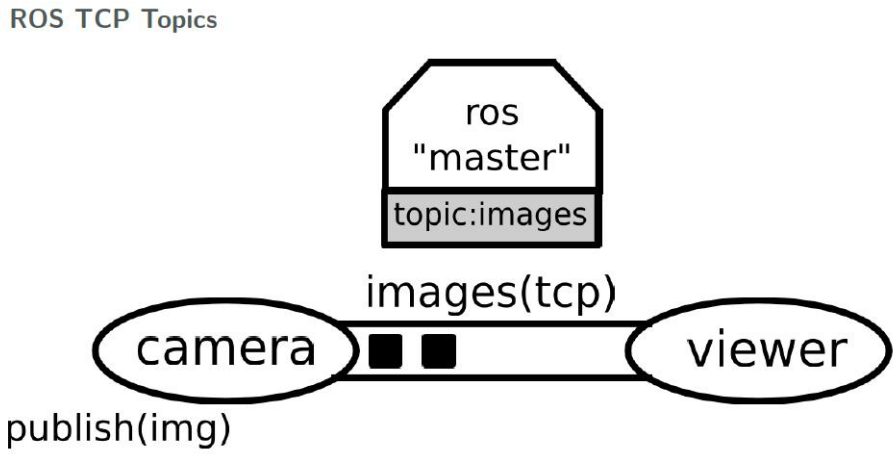


图 1-9 node 直接通讯图

每个 topic 可对应多个 publisher 和 subscriber，如图 1-10，camera-node 发布 image-topic, viewer-node 和 viewer_too-node 均监听于 image-topic。

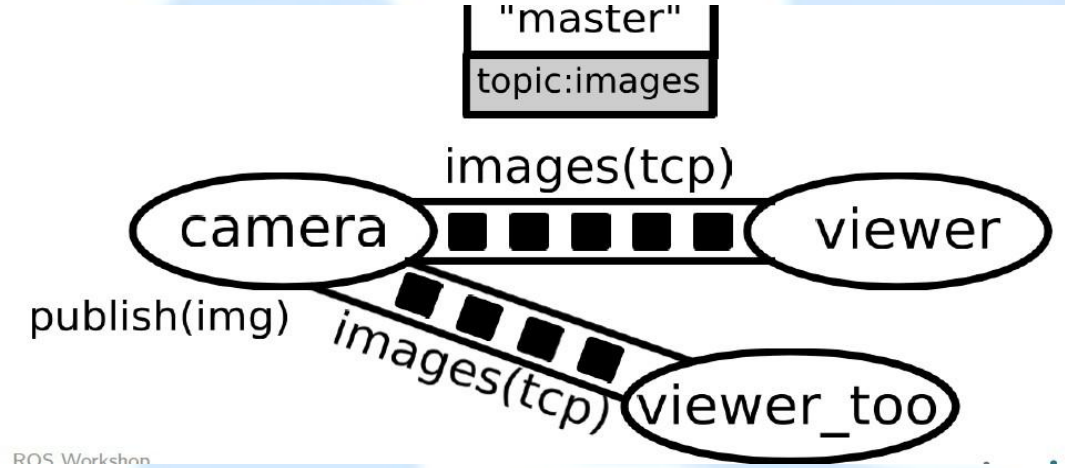


图 1-10 一个 topic 对应多个 publisher 和 subscriber

1.4.1.2 服务和参数

Nodes 之间也可以通过 Service 进行通讯。Service server 提供服务，Service client 通过 XML RPC 调用 Service server 提供的服务，如图 1-11 所示。

Action 和 Service 很相似，对于有时间要求的任务，比如移动机械臂到某个位置，我们想知道机械臂移动到某个位置，执行的结果。Action 可以提供反馈和执行结果，相当于实时的增强型 service。

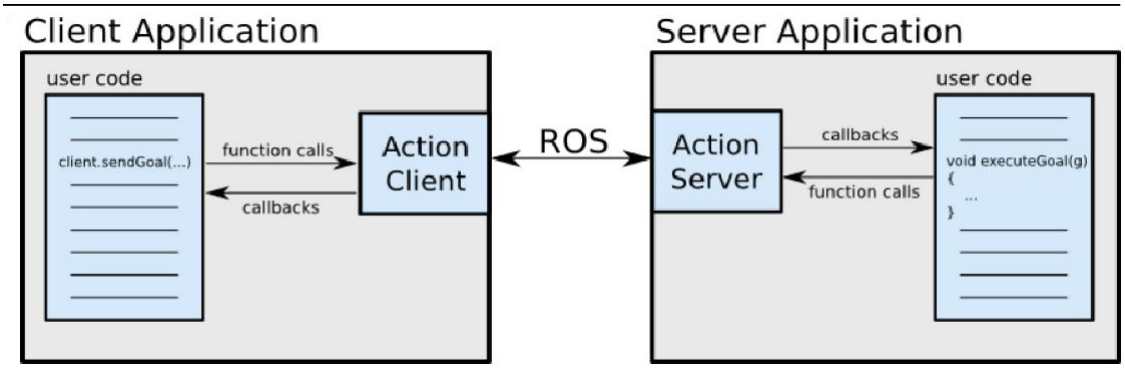


图 1-11　action 工作方式

1.4.2 ROS 基本命令

ROS 有一些特定的操作命令，下面做详细的命令说明。

1.4.2.1 ROS 基本功能包命令

1) rospack(ros+**pack(age))**

rospack 用于获取功能包的相关信息，其命令格式为：

```
$ rospack <command> [package_name]
```

例如：

rospack help，输出 rospack 命令的使用方法。

rospack find [package_name]，查找功能包，返回的是功能包的绝对路径。

rospack depends [package_name]，输出功能包的所有依赖项。

2) rosstack(ros+stack)

rosstack 获取功能包集的相关信息。例如：

rosstack help，输出 rosstack 的帮助信息，即 rosstack 命令的使用方法。

rosstack find [stack_name]，查找功能包集，返回功能包集的绝对路径。

rosstack depends [stack_name]，输出功能包集的所有依赖项。

3) rosls(ros+**ls)**

rosls 是 rosbash 套件的一部分，它可通过功能包的名称列出其包含的文件/文件夹，而不必使用绝对路径，命令格式为：

```
$ rosls [package_name/[subdir]]。
```

4) roscd(ros+cd)

roscd 改变当前目录到指定的功能包/功能包集，命令格式为：

```
$ roscd [package_name /[subdir]]
```

5) catkin_create_pkg

创建 ros 功能包，其命令格式为：

```
$ catkin_create_pkg [package_name] [depend1] [depend2] [depend3]...
```

其中[package_name]后面的部分是新建功能包所依赖的功能包，其作用相当于一个程序文件所包含的头文件。一个功能包可以有多个依赖功能包。

6) catkin**_make**

catkin_make 的作用是编译 ros 功能包。

7) Tab 自动完成输入

当要输入一个完整的软件包名称时会比较繁琐。比如 roscpp_tutorials 是个相当长的名称，输入需要时间，幸运的是，ROS 工具支持 Tab 键自动补全完成输入的功能。

输入：

```
$ roscd roscpp_tut<<<现在请按 TAB 键>>>
```

当按 Tab 键后，命令行中应该会自动补充剩余部分：

```
$ roscd roscpp_tutorials/
```

这应该有用，因为 roscpp_tutorials 是当前唯一一个名称以 roscpp_tut 作为开头的 ROS 软件包。

现在尝试输入：

```
$ roscd tur<<<现在请按 TAB 键>>>
```

按 Tab 键后，命令应该会尽可能地自动补充完整：

```
$ roscd turtle
```

但是，在这种情况下有多个软件包是以 turtle 开头，当再次按 Tab 键后应该会列出所有以 turtle 开头的 ROS 软件包：

```
turtle_actionlib/turtlesim/turtle_tf/
```

这时在命令行中你应该仍然只看到：

```
$roscd turtle
```

现在在 turtle 后面输入 s 然后按 Tab 键：

```
$ roscd turtles<<<请按 TAB 键>>>
```

1.4.2.2 ROS 中的核心命令

1) roscore(ros+core)

对于 ros 1.0，在运行 ros 节点之前，必须先运行 roscore 命令。

2) rosrun (ros+run)

rosrun 运行 ros 节点，命令格式为：

```
$ rosrun [package_name] [node_name]
```

3) rosnode(ros+node)

rosnode 可以显示正在运行的 ros 节点信息。rosnode 的常用命令有如下几种：

rosnode list 列出当前正在运行的节点。

rosnode info [node_name] 输出节点的信息。

rosnode kill [node_name] 结束一个正在运行的节点，例如 rosnode kill -a,结束所有的节点。

4) rosmsg/rossrv(ros+msg/ros+srv)

rosmsg 显示消息数据结构的定义，rossrv 显示服务数据结构的定义。我们可以运行 rosmsg -h (rossrv -h) 查看 rosmsg(rossrv)命令的使用方法。以 rosmsg 为例，运行 rosmsg –h，可以看到 rosmsg 有如下几种命令：

rosmsg show 显示消息中各个变量的定义

rosmsg list	列出 ros 中所有的消息
rosmsg md5	显示消息的 md5 值
rosmsg package	列出功能包中的所有消息
rosmsg packages	列出具有某个消息的所有功能包

5) rostopic(ros+topic)

rostopic 查看节点的主题**信息**，运行 rostopic -h 可以获得该命令的使用方法，如下：

```
rostopic bw     display bandwidth used by topic
rostopic echo   print messages to screen
rostopic find   find topics by type
rostopic hz     display publishing rate of topic
rostopic info   print information about active topic
rostopic list   list active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

- ✧ rostopic bw [topic] 显示主题的带宽；
- ✧ rostopic echo [topic] 输出主题发布的数据；
- ✧ rostopic find 通过类型查找主题；
- ✧ rostopic hz [topic] 输出主题发布的频率；
- ✧ rostopic info [topic] 输出主题的基本信息，包括消息类型、发布节点和订阅节点；
- ✧ rostopic list 输出当前活动的主题；
- ✧ rostopic pub 发布数据到主题；
- ✧ rostopic type [topic] 输出主题的消息类型。

6) rosservice

rosservice 是一个应用于服务的命令。运行命令 rosservice -h，可以获得 rosservice 的帮助信息。rosservice 有如下几种使用方法：

rosservice type	输出服务的类型
rosservice find	通过服务类型查找服务
rosservice uri	输出服务 rosRPC uri
rosservice list	输出当前活动服务的信息
rosservice call	请求服务

7) rosparam

rosparam 可用来保存和设置 ROS 参数服务器(Parameter Server)中的数据。参数服务器可以存储整型、浮点型、布尔型、字典及列表。rosparam 的语法格式采用了 YAML 语言。YAML 语言中：1 表示整型，1.0 表示浮点型，one 表示字符串，true 表示布尔型， [1, 2, 3]表示整型列表，{a: b, c: d}表示字典。 rosparam 的使用方法如下：

rosparam set [param_name]	设置参数
rosparam get [param_name]	获取参数
rosparam load [file_name] [namespace]	从一个文件中加载参数
rosparam dump [file_name] [namespace]	写参数到一个文件
rosparam delete	删除一个参数
rosparam list	列出参数的名称

8) roslaunch

roslaunch 可以按照.launch 文件的描述方式启动节点，其常用方法为：

```
$ roslaunch [package] [filename.launch]
```

9) rosbag

.bag 是 ROS 中用来存储消息数据的文件格式。rosbag 命令可用来处理*.bag* 文件，它的功能包括：记录(record)、总结(info)、回放(play)、检查(check)、修复(fix)等。

1.4.2.3 ROS 操作命令归纳

ROS 的命令可以分为文件系统操作、节点操作、日志及节点主题检查和图形界面工具这四类。

表 1-3 ros 文件系统操作指令

rospack	检查 ros 包
rospack profile	修复路径和 pluginlib 问题
roscd	切换目录
rospd/rosd	ros 的 Pushd 等价
rosls	列出包的信息
rosed	打开 ros 文件
roscp	文件复制
rosdep	安装系统依赖
roswtf	显示正在运行的 ros 系统的警告和错误
catkin_create_pkg	创建包
wstool	管理工作空间的软件库
catkin_make	编译工作空间代码
rqt_dep	显示软件包结构和依赖

具体的一些用法：

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rospd [package[/subdir] | +N | -N]
$ rosd
$ rosls [package[/subdir]]
$ rosed [package] [file]
$ roscp [package] [file] [destination]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ catkin_create_pkg [package name] [depend1]..[dependN]
$ wstool [init | set | update]
$ catkin_make
$ rqt_dep [options]
```

如表 1-4 为节点的操作指令：

表 1-4 节点操作指令

roscore	启动 ros 主节点及参数服务等
rosrun	运行可执行文件
roslaunch	启动 launch 文件

具体的一些用法：

```
roscore：
$ roscore

rosrun:
$ rosrun package_name executable_name
$ rosrun turtlesim turtlesim.node

roslaunch:
$ roslaunch package_name file_name.launch
$ roslaunch -p 1234 package_name file_name.launch
$ roslaunch --local package_name file_name.launch
```

表 1-5 日志及节点主题检查

rosbag record	记录特定 topic 的数据到 bag 文件
rosbag play	回放 bag 文件
rosbag compress	压缩一个或多个 bag 文件
rosbag decompress	解压缩 bag 文件
rosbag filter	过滤 bag 文件内容
rosmsg show	显示消息定义
rosmsg list	显示全部消息名字
rosmsg md5	显示消息 MD5 值
rosmsg package	列出包中的所有消息
rosmsg packages	列出包含消息的所有包
rosnode ping	测试节点连接
rosnode list	列出所有活动节点
rosnode info	输出节点信息
rosnode machine	列出某机器人上运行的节点
rosnode kill	终止一个节点
rostopic bw	topic 带宽
rostopic echo	显示 topic 内容
rostopic find	根据类型寻找 topic
rostopic hz	topic 发布频率
rostopic info	topic 信息
rostopic list	所有 topic 列表
rostopic pub	向 topic 发布消息
rostopic type	topic 类型
rosparam set	设置参数值
rosparam get	获取参数值
rosparam load	装载参数文件
rosparam dump	将参数写入到文件
rosparam delete	删除参数
rosparam list	所有参数列表
roservice list	列出所有可用服务

roservice node	打印提供服务的节点
roservice call	调用服务
roservice args	列出服务所需参数
roservice type	打印服务类型
roservice uri	打印服务 rosRPC uri
roservice find	根据服务类型找到服务

具体的一些用法：

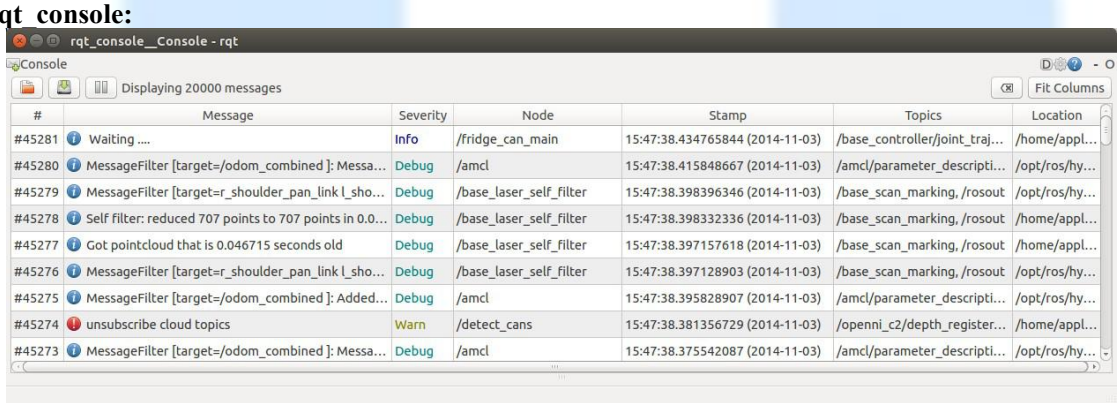
```
记录：
$ rosbag record topic1 topic2
回放：
$ rosbag play -a demo log.bag
多个同时回放：
$ rosbag play demo1.bag demo2.bag
```

表 1-6 图形界面工具

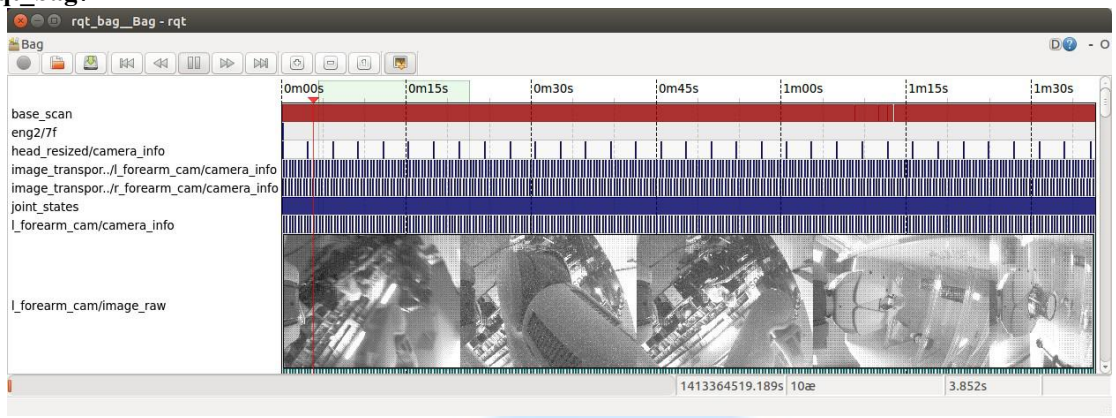
rqt_console	显示 rosout 上的消息
rqt_bag	记录、回放 bag 文件
rqt_logger_level	设置日志等级，如警告、错误
rqt_topic	查看发布的 topics
rqt_msg/srv	显示消息或服务定义
rqt_publisher	向 topic 发送消息
rqt_service_caller	调用服务
rqt_graph	节点关联图
rqt_dep	软件包依赖关系
rqt_reconfigure	参数动态配置
rqt_shell	指令输入窗口
tf_echo	打印坐标系间的坐标变换
view_frames	查看系统坐标变换树
rqt_plot	画出 topic 数据图形
rqt_image_view	显示图像

示例：

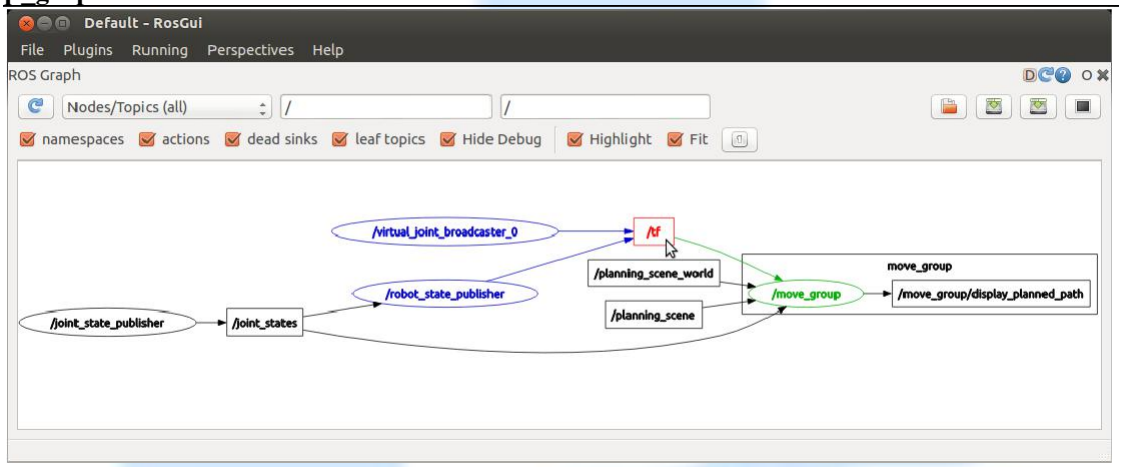
rqt_console:



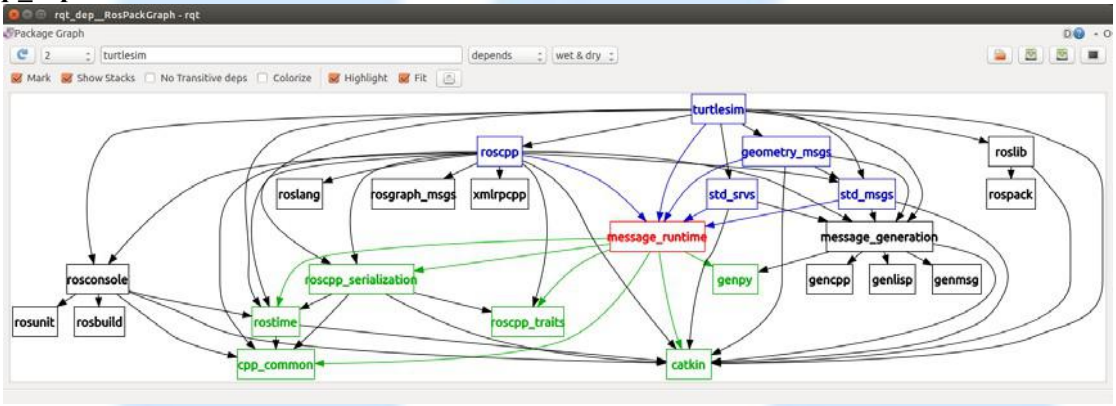
rqt_bag：



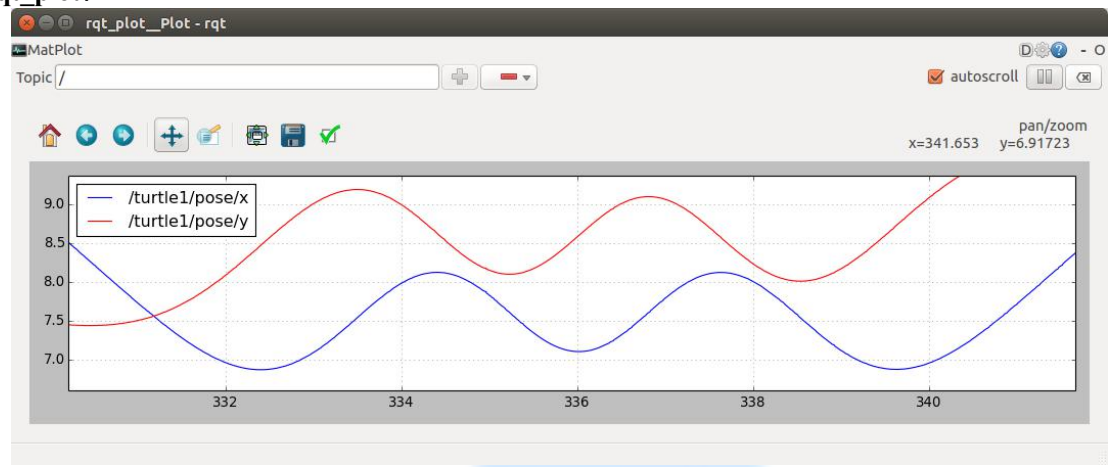
rqt_graph:



rqt_dep：



rqt_plot：



rqt_image_view:

1.4.3 创建 ROS 程序

首先需要创建一个 ROS 工作区间，然后可以通过（catkin_create_pkg)工具来自动创建 ros package；

1.4.3.1 创建 ROS 工作空间

1）命名工作区间

打开 Gnome Terminal,可以通过鼠标点击左上角图标或者搜索“terminal”,亦可通过组合快捷键“Ctrl + Alt + T” 打开：

```
$ mkdir -p ~/ros_ws/src
# ros_ws (short for ros Workspace)
```

其中~/ros_ws/src 为 ROS 开发环境所在路径地址，~代表用户 Home 目录路径。此目录用户可以自定义，推荐与官方 wiki 上写的一样，这样之后看 wiki 也方便些，注意 $ 符号后面的为 terminal 终端所应输入的命令，#符号后为提示语，请不要输入。

2）Source ROS

```
$ source /opt/ros/indigo/setup.bash
```

注意：在每次打开终端时，都先要运行上述命令才能运行 ROS 相关命令，为了避免这一繁

琐过程，可以事先在.bashrc 文件（该目录在当前系统的 Home 目录下）中添加这条命令，这样当你每次登录后，系统自动帮你执行这些命令配置好开发环境。

3）编译与安装

这一步要在创建的工作空间 ros_ws 中进行，所以先要改变工作目录路径，进入 ros_ws。

```
$ cd ~/ros_ws
$ catkin_make
$ catkin_make install
```

1.4.3.2 创建 ROS 程序包

直接使用(catkin_create_pkg)工具来自动创建 ROS 包，命令如下：

```
$ cd ~/ros_ws/src
$ catkin_create_pkg   my_first_pkg   rospy
$ cd ~/ros_ws
$ catkin_make
$ catkin_make install
```

这样就创建了第一个 ROS 程序包。此时，通过以下命令进入到 my_first_pkg 程序包内，将会看到一个 src 文件夹、CMakeLists.txt 文件和 package.xml 文件。

```
$ cd ~/ros_ws/src/my_first_pkg
```

1.4.3.3 编写 Publisher 与 Subscriber

节点是 ROS 里的可执行单元，可以连接到 ROS 网络中去，这里我们创建一个消息发布节点（“talker”），它将持续的向 ROS 系统发布消息。

同时创建一个名为 listener 的 Subscriber 节点接收 talker 节点发布的字符串消息，并在屏幕上打印。

1）Publisher 代码

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String
def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10)   #10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try :
```

```
        talker()
    except rospy.rosInterruptException:
pass
```

2)Publisher 代码解释

```
#!/usr/bin/env python
```

每一个 python 节点文件的顶部都应该放上这么一段，声明用哪个 Python 解释器执行这个脚本。

```
import rospy
from std_msgs.msg import String
```

导入 rospy 包，从 std_msgs 包中引入 String 类型的消息。

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

创建消息发布器，初始化 ROS 节点。

```
rate = rospy.Rate(10)  #10h
```

设定消息发布频率为 10Hz。

```
while not rospy.is_shutdown():
    hello_str = "hello world %s"  % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

主循环，当 ROS 没有关闭时，持续发布‘helloworld’的字符串消息。

```
try :
  talker()
except rospy.rosInterruptException:
  pass
```

异常捕获，Ctrl+C 结束程序。

3)Subscriber 代码

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
    # In ROS, nodes are uniquely named. If two nodes with the same
    # node are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
```

```
    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

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

打开终端，为代码添加可执行权限：

```
$ chmod +x listener.py
```

4)Subscriber 代码解释

```
rospy.init_node('listener', anonymous=True)
rospy.Subscriber("chatter", String, callback)
# spin() simply keeps python from exiting until this node is stopped
rospy.spin()
```

声明节点要监听的 topic 名字及消息类型，设置回调函数，当新消息到来时自动调用回调函数，对传入的消息做相应处理。

rospy.spin()保持节点处于运行状态，不同于 roscpp 中的 spin()，Python 中的 subscriber 有自己的线程，spin()不影响回调函数。

1.4.3.4 编写 Service 与 Client

这里我们将创建一个名为（“add_two_ints_server”）的 Service，它接收传递过来的两个整数，返回两数之和。

1)Service 代码

```
#!/usr/bin/env python
from beginner_tutorials.srv import *
import rospy
def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a +req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__== " __main __":
   add_two_ints_server()
```

1）Service 代码解释：

同样我们首先通过 rospy 初始化节点。

```
s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
```

然后声明一个新的 service，名字为“add_two_ints”，类型是 AddTwoInts。然后把服务请求

中的参数传递到 handle_add_two_ints，该函数求处理请求后返回请求处理结果。同样，这里的 rospy.spin()只是将程序挂起，没有实际作用。

2）Client 代码

```
#!/usr/bin/env python
import sys
import rospy
from beginner_tutorials.srv import *
def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__== " __main__ ":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

4)Client 代码解释

Client 没必要初始化 ROS 节点，我们通过 wait_for_service 等待 Service 端服务启动，服务启动之前，此函数将一直处于等待状态。

```
rospy.wait_for_service('add_two_ints')
```

然后通过 ServiceProxy 创建 Client 对象。

```
add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
```

最后直接调用该对象，并将两个整数作为参数，返回值存到 resp1 中。

```
resp1 = add_two_ints(x, y)
return resp1.sum
```

1.5 ROS 仿真

RViz 和 Gazebo 是 ROS 开发中常用的 3D 显示和仿真工具。RViz 对数据进行三维可视化并进行交互操作，还常用于机器人的运动规划，包括机械臂运动规划、移动机器人路径规划等。Gazebo 是一款流行的机器人的仿真软件，它基于物理引擎 ODE。除 Gazebo 外，也有其他仿真

软件如 ArbotiX 和 Stage。 本章将介绍 RViz 和 Gazebo 的使用方法以及如何利用这两个工具进行仿真。

1.5.1 RViz 仿真工具

RViz 是 Willow Garage 开发的 3D 数据可视化环境。它将坐标和四元数等数据表示成三维图像，使得机器人编程的调试工作变得更容易。在 RViz 中可以同时查看二维图像、深度云、激光扫描数据、点云、坐标系、机器人实时姿态、里程计、路径、地图等丰富的数据类型。此外，RViz 为编程人员提供了可视化标记(visualization markers)，你可以通过编程让 RViz 显示一些三维标记，如箭头、坐标系，你甚至能在 RViz 中操作这些标记来输入位姿数据，这意味着只要编写好相应的控制程序，你就可以使用 RViz 来操控机器人。

1.5.1.1 安装 RViz

ROS wiki 页面：http://wiki.ros.org/rviz

官网提供的 ROS 安装选项有很多，如果选择直接用 ros-<distro>-desktop-full 就会把 RViz 自动包含进去。若需单独安装 RViz，安装命令：

```
$ sudo apt-get install ros-indigo-rviz
```

本节的 RViz 实例使用仿真器 ArbotiX。ArbotiX 可以模拟移动机器人，但不具备任何物理特性，安装命令：

```
$ sudo apt-get install ros-indigo-arbotix-*
```

1.5.1.2 RViz 应用方法

在一个终端中运行 roscore 命令启动 ros master，然后在另一个终端中启动 RViz，运行命令：

```
$ rosrun rviz rviz
```

RViz 启动后的界面如图 1-12 所示。此时没有加载机器人模型。

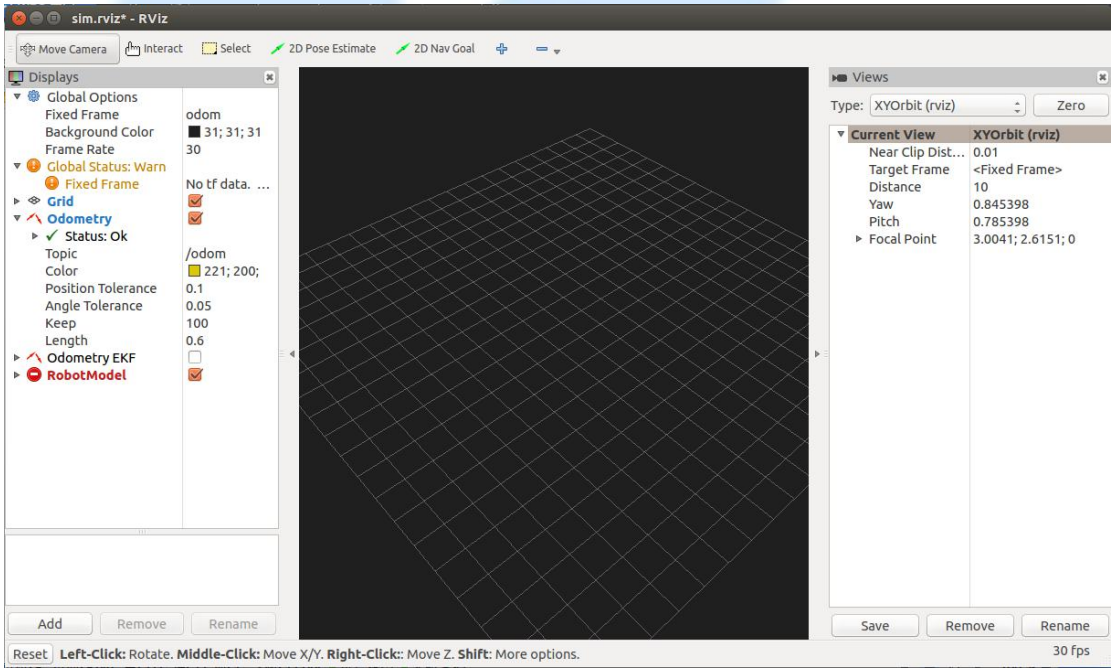


图 1-12 RViz 启动界面

从图 1-12 中可以看出，RViz 主要包括以下几部分：

1) 工具(Tool)栏：

RViz 的工具栏位于窗口上方，包含若干操作按钮。如图 1-13 所示。

图 1-13 RViz 的工具栏

Interact：旋转/平移 3D 显示窗口中的对象。

Move Camera：调整观察的视角，与交互的功能类似。在这模式下，尝试分别按住鼠标左、右键或中键在 3D 窗口拖动。

Select:选择 3D 显示窗口中的部分，选中的效果如图 1-14 所示, 为了方便查看，将背景颜色调浅。可以在 Displays->Global Options->Background Color 处修改背景颜色。（下图使用的是 turtlebot_gazebo 包中的 turtlebot 模型，黄色箭头为 Odometry。要使用 turtlebot 仿真，请访问 ros 官网的页面 http://wiki.ros.org/Robots/TurtleBot 查看使用教程。）

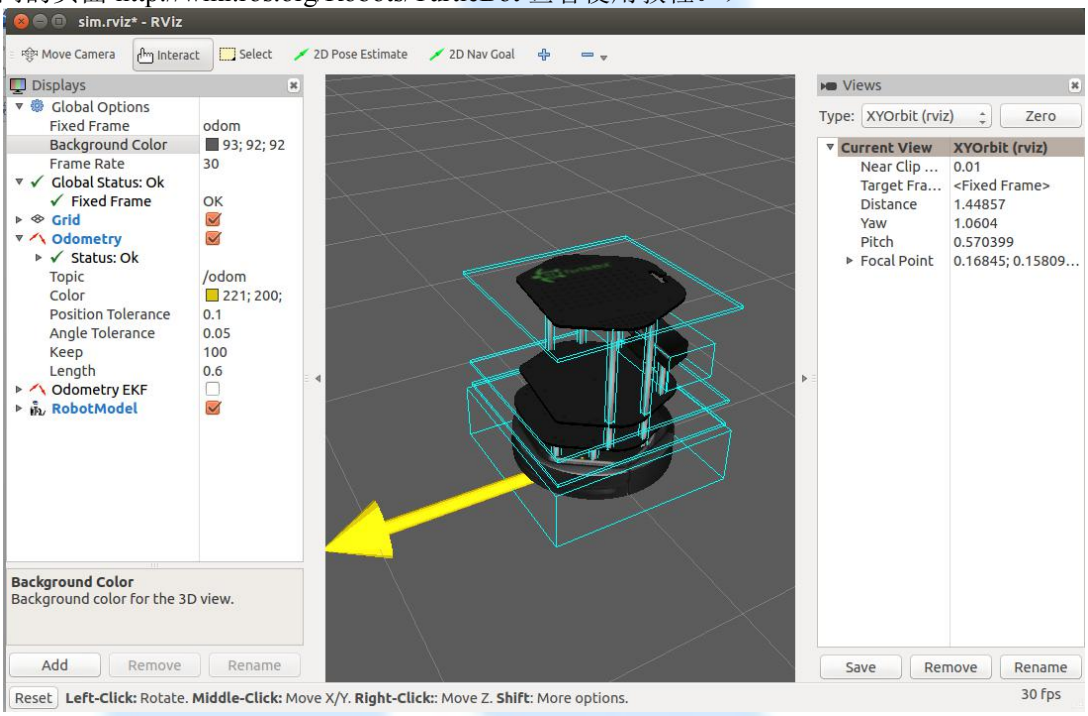


图 1-14 RViz 中选中机器人的效果

Measure

2D Pose Estimate：用户可采用该工具设置移动机器人的初始位姿来启动定位系统，在地面上点击定位位置并拖拽选中的方向即可。

2D Nav Goal：为移动机器人设置目标位姿，在地面上点击定位位置并往任意方向拖拽即可。

+ /-：点击‘+’按键可以添加新的工具，点击'-'按键可以删除工具。

2) 3D 显示窗口

3) 显示(Displays)栏

显示栏主要显示全局选项(Global Options)、网格(Grid)、机器人模型(RobotModel)等部分的设置以及状态信息，如图 1-15 所示。

显示栏的选项可以通过点击右边的值来输入。右边的勾选框用于隐藏/显示数据。有的选项会提供下拉菜单，一般用于选择 tf frame（坐标系）或者 topic。

左边的选项名旁如果带有小三角，可点击小三角来展开和折叠次级选项。

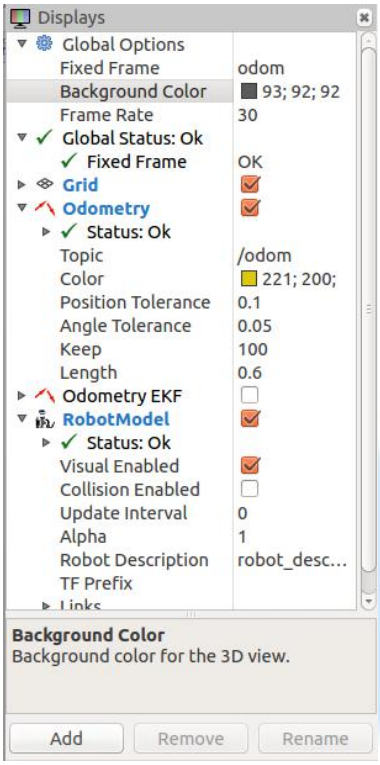


图 1-15　RViz 中的显示栏

(1) 全局选项(Global Options)：该项中最重要的是设置固定坐标系(Fixed Frame)，固定坐标系设置的不恰当会影响数据和机器人模型的显示，如图 1-16 和 1-17 所示

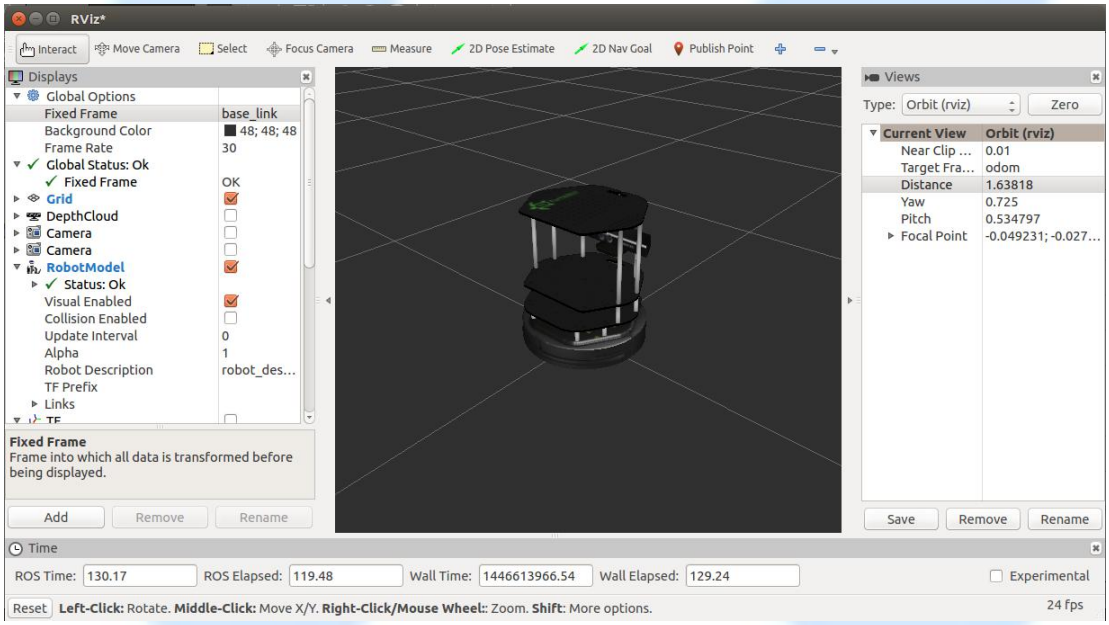


图 1-16 机器人正常状态

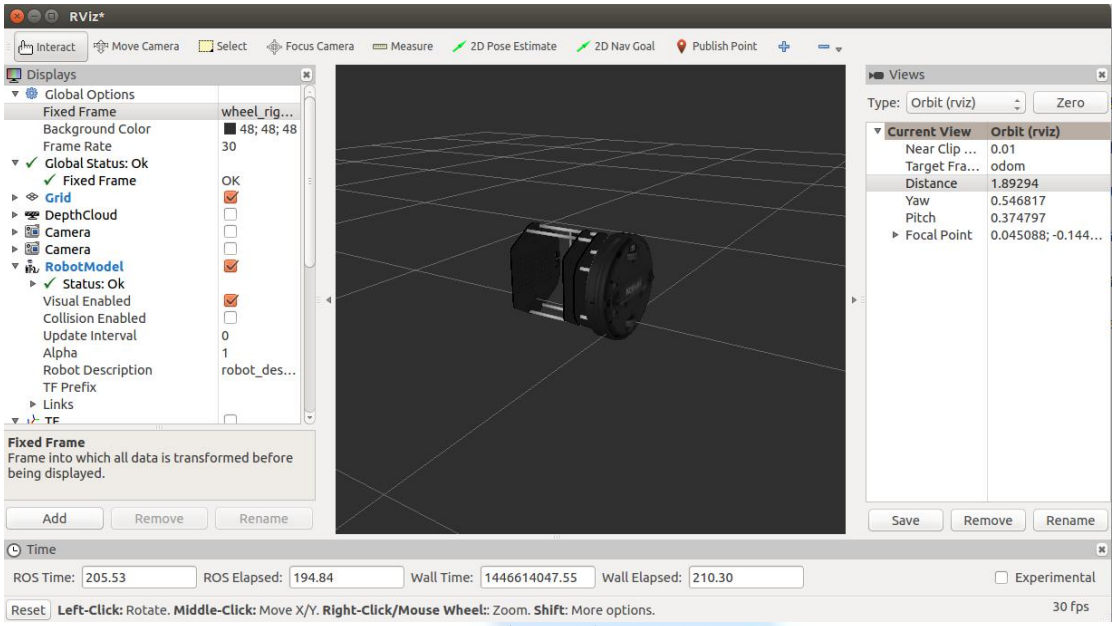


图 1-17 机器人“倒下”状态

(2) 网格(Grid)：该项包括网格的参考坐标系、状态、数目、大小、颜色等信息。

(3) 机器人模型(RobotModel)：该项包括机器人模型的状态、允许可见 (Visual Enabled)、允许碰撞(Collision Enabled)、连接件(Links)等信息。

研究人员可以根据自己的需求添加新的显示项目。点击显示栏下面的“Add” 按键，弹出如图 1-18 所示的对话框。图 1-18 添加了坐标轴(Axes)和 TF 坐标系(TF frame)。TF(transform)是 ros 里表示关节空间位置的重要工具。

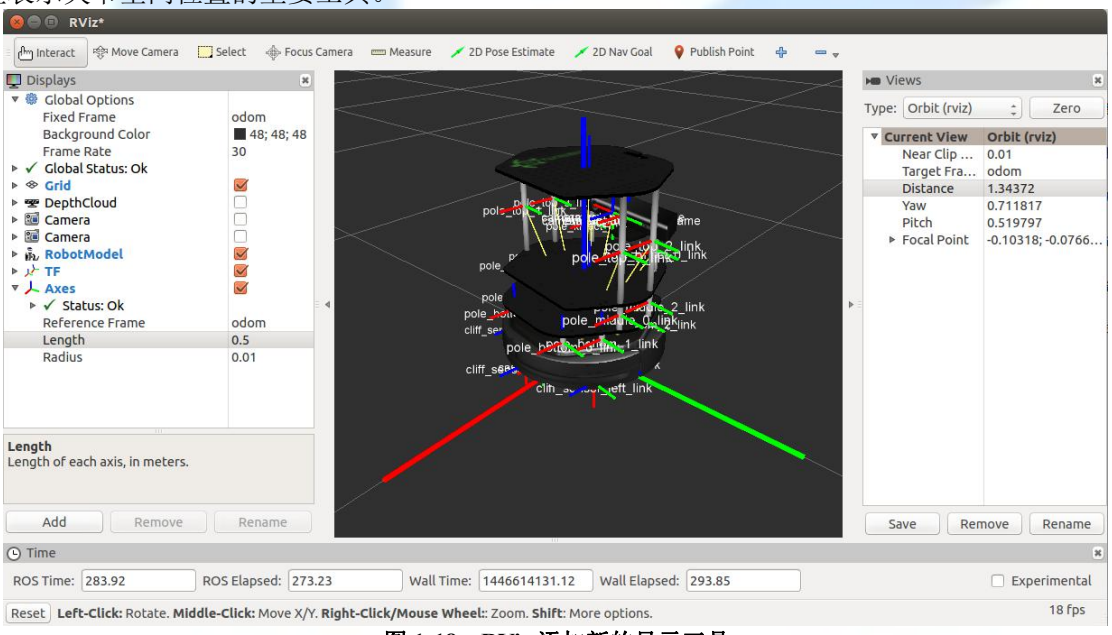


图 1-18 RViz 添加新的显示工具

4) 视图(Views)栏

在视图栏中选择不同类型的摄像机，如图 1-19 所示，可根据不同的需求实现对机器人的多角度观察。

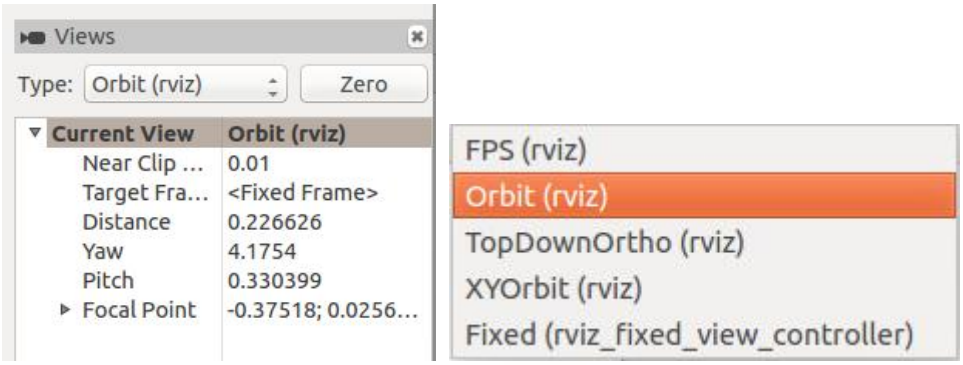


图 1-19　RViz 的视图栏

这些摄像机都需要设置目标坐标系作为(Target Frame)，以此为参考坐标系，不同摄像机可以进行不同自由度的姿态修正，姿态修正可以用鼠标移动摄像机，也可以修改视图栏的参数直接设置。有时候 RViz 会出现一些小 bug 导致摄像机在修改后距离坐标原点太远，这时候可以直接修改参数（如图 1-19 的 Distance）使其回到正常距离。

在修改 RViz 配置后，可以在菜单栏选择 File->Save Config 保存默认设置，也可以另存为(Save Config As)一份（建议使用*.rviz 后缀），以后在启动 RViz 时，使用如下命令加载：

```
$ rosrun rviz rviz -d <文件名>
```

(1) 轨道摄像机(Orbital Camera)：rviz 的默认配置。有一个焦点标记 Target Frame 的位置，旋转时摄像机环绕焦点，平移时焦点移动，摄像机和焦点相对位置不变。当移动摄像机时，焦点显示为一个小的圆盘，如图 1-20 所示。

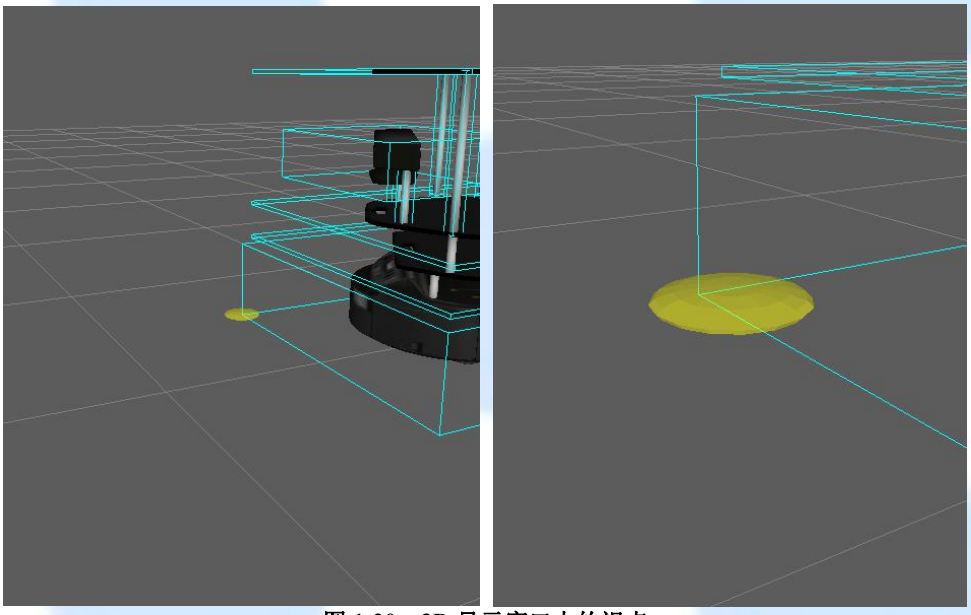


图 1-20　3D 显示窗口中的视点

轨道摄像机的控制方式为：
鼠标左键：点击并拖拽，实现绕视点的旋转
鼠标中键：点击并拖拽，实现视点的平移
鼠标右键：点击并拖拽实现对视点的放大/缩小
鼠标滚轮：实现对视点的放大/缩小

(2) 第一人称摄像机(FPS, First-Person)：
此时相当于手持摄像机进行控制。该摄像机的控制方式为：
鼠标左键：点击并拖拽实现绕视点的旋转

鼠标中键：点击并拖拽，实现在摄像机向上/向右向量形成的平面内平移
鼠标右键：点击并拖拽实现沿摄像机前向向量的移动
鼠标滚轮：向前/后移动

(3) 垂直正交摄像机(Top-down Orthographic)
该摄像机只能沿着 Z 轴(机器人坐标系)向下看，即只能看到机器人的上面。控制方式为：
鼠标左键：点击并拖拽实现绕 Z 轴的旋转
鼠标中键：点击并拖拽，实现在 XY 平面内平移
鼠标右键：点击并拖拽实现对图像的放大
鼠标滚轮：放大图像

(4) 平面轨道摄像机(XY Orbit)
和轨道摄像机类似，但焦点圆盘固定在 XY 平面上。

(5) 固定摄像机(Fixed)
固定在 Target Frame 上，不可移动。

(6) 复位(Reset)键
实现对机器人模型的复位，使其回到最初状态。读者可以根据需要在 Panels 菜单中添加/去掉某部分，如图 1-21 所示。

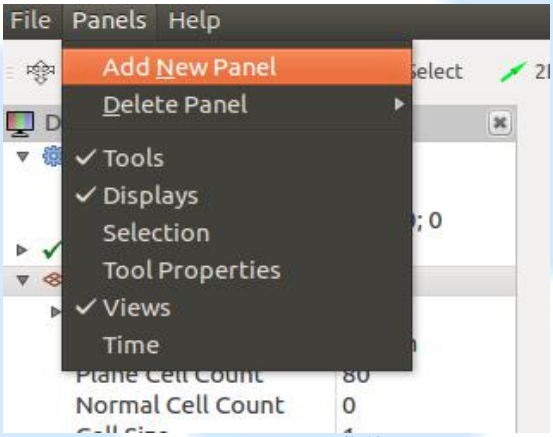


图 1-21 Panels 菜单

1.5.1.3 RViz 仿真实例

下面将通过一个小例子重点介绍如何创建机器人的模型文件，如何设置 RViz 以及如何控制 RViz 中的机器人，使读者能够初步掌握 RViz 的使用方法。本例使用的仿真器为 ArbotiX。

1) 创建功能包

```
$ source ~/catkin_workspace/devel/setup.bash
$ cd ~/catkin_workspace/src
$ catkin_create_pkg my_robot_sim rospy
$ cd ..
$ catkin_make
```

2) 创建模型文件

在 my_robot_sim 功能包中创建一个名为 urdf 的文件夹，用于存放机器人模型文件。

(1) 主体文件

在 urdf 文件夹中，新建名为 my_robot_body.xacro 的文件。该文件的内容如下：

```
1  <?xml version="1.0"?>
2  <robot>
```

```
3    <!--predefined variables-->
4   <property name="joint_damping" value= "0.1" />
5   <property name="joint_friction" value= "0.0" />

6   <!--robot links-->
7   <link name="world"/>
8   <link name="base_link">
9       <inertial>
10          <mass value="1.0" />
11          <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
12      </inertial>
13      <visual>
14          <geometry>
15              <box size="0.07 0.05 0.2"/>
16          </geometry>
17  <origin rpy="0 0 0" xyz="0 0 0.1"/>
18          <material name="bule">
19  <color rgba="0 0 0.8 1"/>
20          </material>
21      </visual>
22      <collision>
23          <geometry>
24              <box size="0.07 0.05 0.2" />
25          </geometry>
26          <origin rpy="0 0 0" xyz="0 0 0.1"/>
27      </collision>
28  </link>
29  <link name="second_link">
30      <inertial>
31          <mass value="1.0" />
32          <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
33      </inertial>
34      <visual>
35          <geometry>
36              <box size="0.07 0.05 0.3"/>
37          </geometry>
38  <origin rpy="0 0 0" xyz="0 0.025 0.11"/>
39          <material name="green">
40  <color rgba="0 1 0 1"/>
41          </material>
42      </visual>
43      <collision>
44          <geometry>
45              <box size="0.07 0.05 0.3" />
46          </geometry>
47          <origin rpy="0 0 0" xyz="0 0.025 0.11"/>
48      </collision>
49  </link>

50  <!--robot joints-->
51  <joint name="myrobot_joint1" type="fixed">
52      <parent link="world"/>
53      <child link="base_link"/>
54      <origin rpy="0 0 0" xyz="0.0 0.0 0.0"/>
55  </joint>

56  <joint name="myrobot_joint2" type="continuous">
```

```
57          <axis xyz="0 1 0"/>
58          <parent link="base_link"/>
59          <child link="second_link"/>
60          <origin rpy="0 0.5 0" xyz="0 0.025 0.15"/>
61          <limit effort="1000" velocity="1000"/>
62          <joint_properties damping="${joint_damping}" friction="${joint_friction}"/>
63    </joint>
64</robot>
```

代码解释：

机器人模型文件的主要包括预定义、连接杆(<link>)、关节(<joint>)。

<link>建立机器人单个部件的模型。其中 name 是<link>必需的，其他各项可以根据需要进行选择，<collision>的形状可以与<visual>不同，当<visual>的形状很复杂时，<collision>采用简单的形状可以提高碰撞检测的实时性。

注意，在 xacro 文件和 urdf 文件里均不能有中文，即便是中文注释。xacro 解析器目前不能支持含有非 ASCII 码字符的文件。一旦 xacro.py 解析出错，尝试运行以下命令查看详细信息:

```
$ rosrun xacro xacro.py <文件目录/文件名>
```

<joint>的作用是将机器人的各个部件连接起来。

(2) 主文件

主文件的作用是将多个机器人主体文件中的模型组装成一个完整的机器人模型。

在 urdf 文件夹中，新建名为 my_robot.urdf.xacro 的文件。该文件的内容如下：

```
<?xml version="1.0"?>
<robot name="myrobot"
      xmlns:xi="http://www.w3.org/2001/XInclude"
      xmlns:gazebo="http://playerstage.sourceforge.net/gazebo/xmlschema/#gz"
      xmlns:model="http://playerstage.sourceforge.net/gazebo/xmlschema/#model"
      xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
      xmlns:body="http://playerstage.sourceforge.net/gazebo/xmlschema/#body"
      xmlns:geom="http://playerstage.sourceforge.net/gazebo/xmlschema/#geom"
      xmlns:joint="http://playerstage.sourceforge.net/gazebo/xmlschema/#joint"
      xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
      xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
      xmlns:rendering="http://playerstage.sourceforge.net/gazebo/xmlschema/#rendering"
      xmlns:renderable="http://playerstage.sourceforge.net/gazebo/xmlschema/#renderable"
      xmlns:physics="http://playerstage.sourceforge.net/gazebo/xmlschema/#physics"
      xmlns:xacro="http://ros.org/wiki/xacro">

<xacro:include filename="$(find my_robot_test)/urdf/myrobot.transmissions. xacro" />
</robot>
```

(3) 创建 launch 文件

在 my_robot_sim 功能包中创建一个 launch 文件夹，在其中新建一个名为 my_robot_rviz.launch 的文件，内容如下：

```
1<launch>
2      <param name="/use_sim_time" value="false" />
3      <!-- Load the URDF/Xacro model of our robot -->
4      <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find my_robot_test)/urdf/
         my_robot.urdf.xacro'" />
```

```
5     <param name="robot_description" command="$(arg urdf_file)" />
6     <arg name="gui" default="false" />
7     <param name="use_gui" value="$(arg gui)"/>
8     <node name="arbotix" pkg="arbotix_python" type="arbotix_driver" output="screen">
9           <rosparam file="$(find my_robot_test)/config/myrobot_arbotix.yaml"
            command="load"/>
10            <param name="sim" value="true"/>
11    </node>
12    <node name="joint_state_publisher" pkg="joint_state_publisher"
            type="joint_state_publisher">
13            <param name="use_gui" value="$(arg gui)"/>
14    </node>
15    <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher">
16          <param name="publish_frequency" type="double" value="20.0" />
17    </node>
18    <node name="rviz" pkg="rviz" type="rviz" />
19</launch>
```

(4) 测试

在终端中运行下面的**命令：**

```
$ roslaunch my_robot_sim my_robot_rviz.launch
```

运行结果如图 1-22 所示。

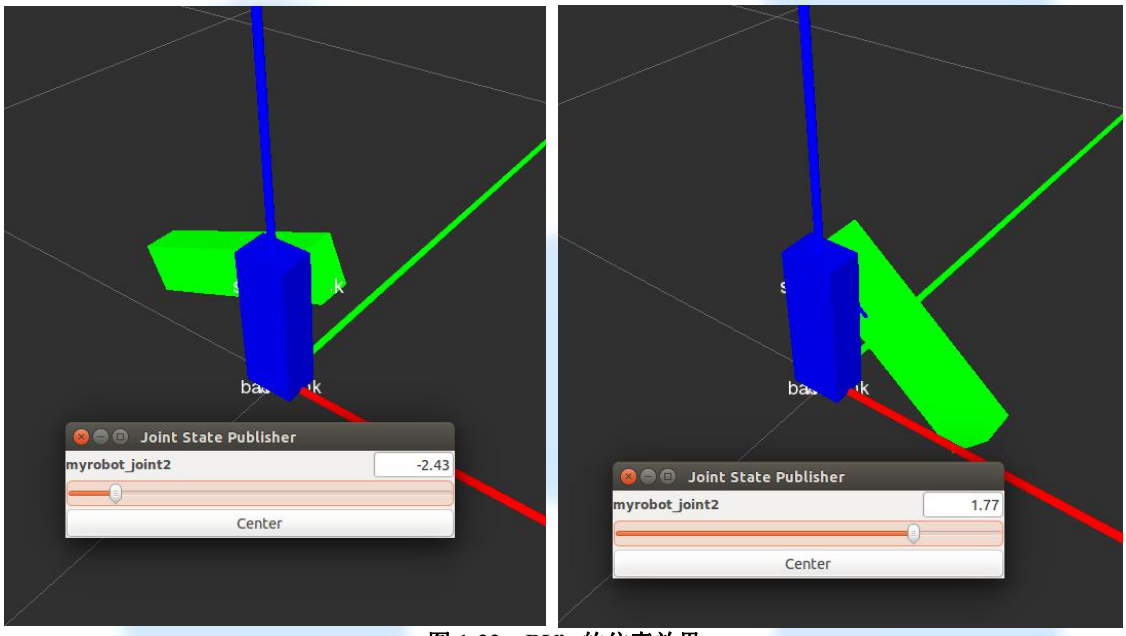


图 1-22　RViz 的仿真效果

1.5.2 Gazebo 仿真工具

1.5.2.1 Gazebo 使用方法

Gazebo 是 ROS 中的物理仿真环境，可模拟机器人以及环境中很多物理特性。图 1-23 是采用 Gazebo 进行机器人仿真的效果图。在 Indigo 下，Gazebo 的版本为 2.x，最新的 ros Jade 使用的是 Gazebo 5.x 。

Gazebo 的官网教程在:http://gazebosim.org/tutorials/

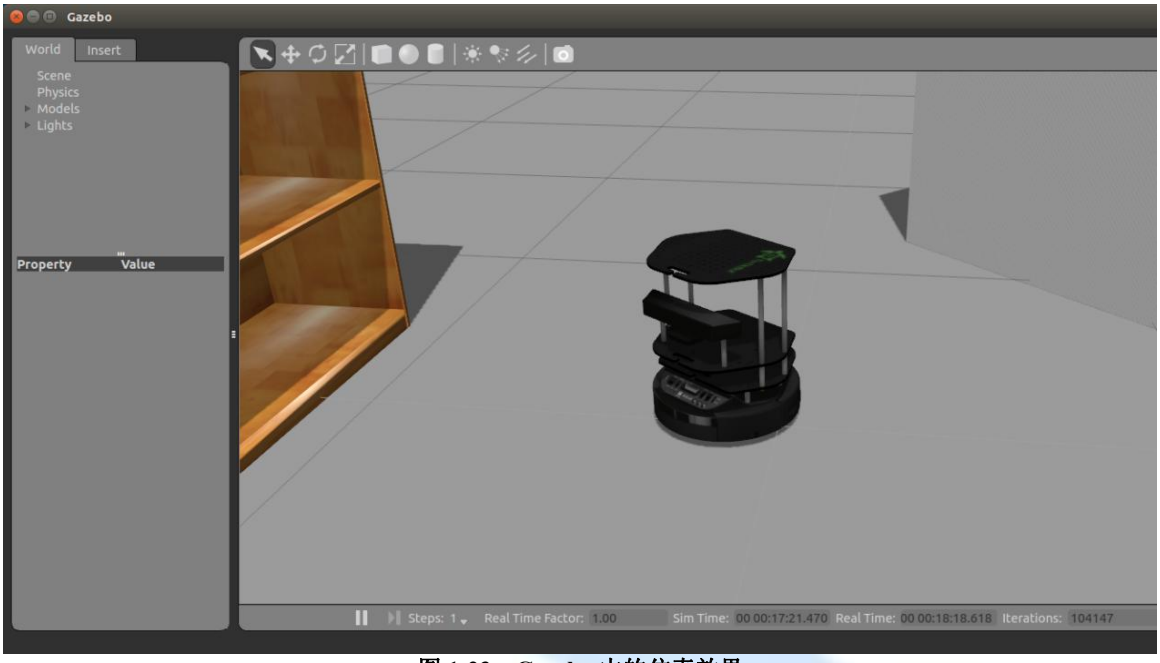


图 1-23　Gazebo 中的仿真效果

图 1-23 同样使用了 turtlebot_gazebo，须自行安装 turtlebot 系列包。运行命令:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

从图中可以看出 Gazebo 软件主要包括以下几部分：

1) 工具栏

gazebo 的工具栏如图 1-24 所示：

图 1-24 Gazebo 的工具栏

(1) 

用来添加光源调整仿真环境中的光照，分别为点光源、有向光源、平行光。当仿真环境亮度比较低时时，可以添加光线以便观察，如图 1-25 所示。

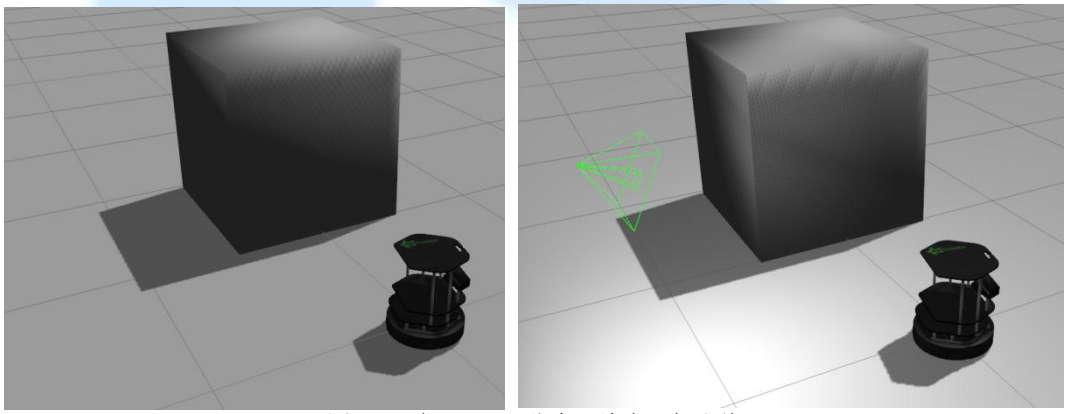


图 1-25 在 Gazebo 仿真环境中添加光线

(2) 

添加标准模型：长方体、圆柱体、球。

(3) 

在前三个图标下时，你都可以调整观察的角度和位置，点击鼠标左键并拖拽实现对视点的平移；点击鼠标中键并拖拽实现绕视点的旋转；点击鼠标右键并拖拽实现对视点的放大或缩小；鼠标滚轮也可以实现对视点的放大或缩小。可以实现对单个物体的移动，可以实现对单个物体的旋转。可以缩放物体的大小。在平移、旋转或缩放物体时，需要先选中目标物体(鼠标左击)，如图 3.14 所示。本例中的 Gazebo 对于物体缩放仅支持简单形状的缩放。若尝试缩放复杂的机器人模型，打开 Gazebo 的终端会输出”Scaling is currently limited to simple shapes.“的警告信息。

(4)  ：场景快照。拍下快照后场景左上角会显示图片保存位置。

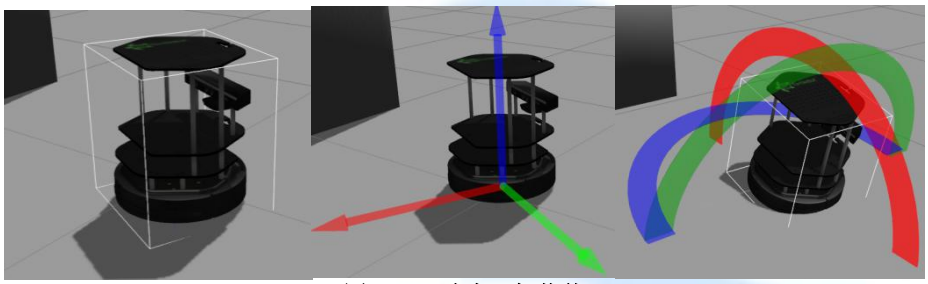


图 1-26　选中目标物体

2) World 栏

点击左侧的“World”标签，可以看到如图 1-27 所示的界面。该界面主要显示 Gazebo 仿真环境的设置情况，包括场景(Scene)、物理参数(Physics)、模型(Models)、光线(Lights)。

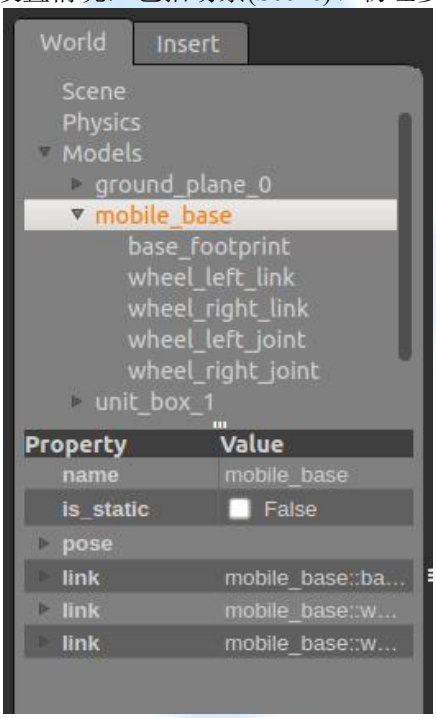


图 1-27 Gazebo 中的“World”标签

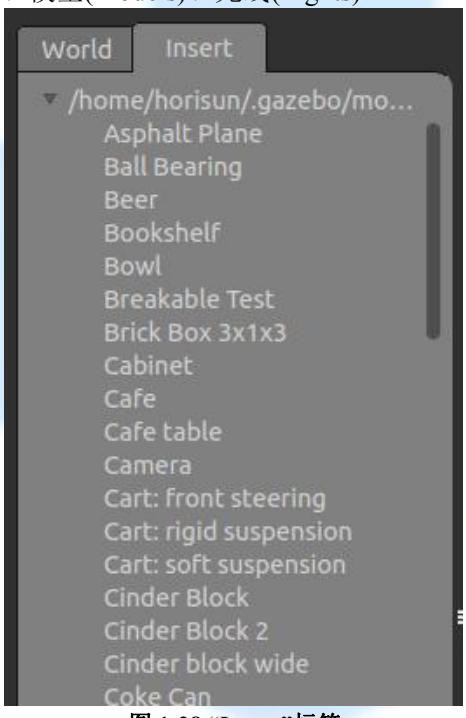


图 1-28 “Insert”标签

3) Insert 栏

点击 gazebo 左侧的“Insert”标签，如图 1-28 所示。该界面提供已有的模型列表，可以方便开发人员向仿真环境中添加复杂的模型。例如点击 Insert 栏中的 PR2 可以向仿真环境中添加 PR2 机器人的模型，如图 1-29 所示。在修改场景后，用 File->Save World 保存当前场景，保存格式为 SDF。

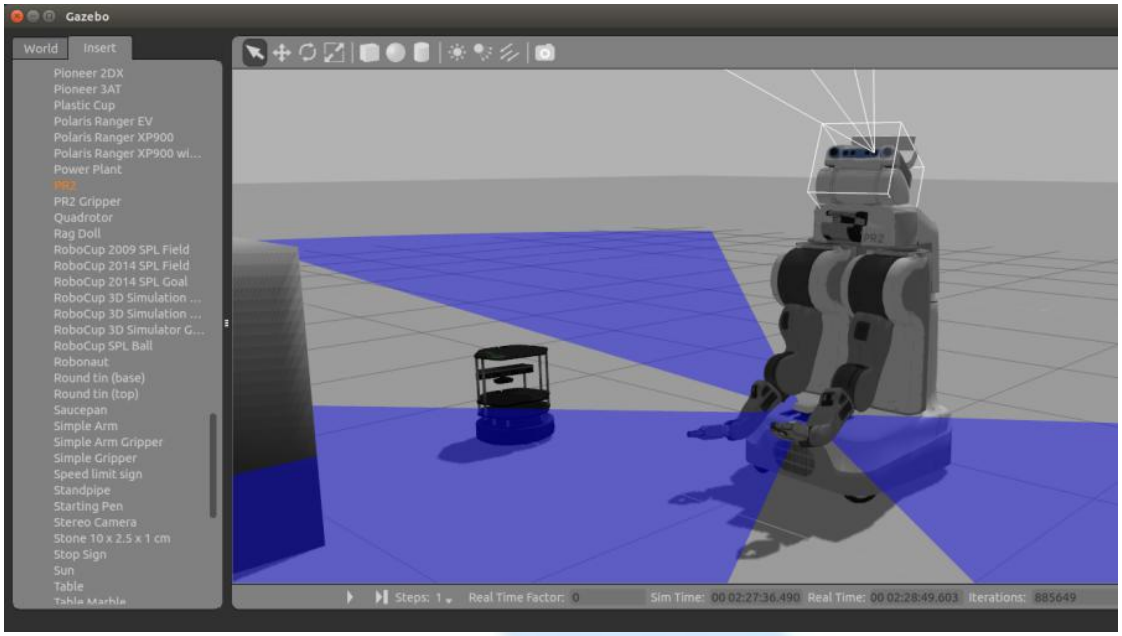


图 1-29 添加 PR2 机器人

4) 菜单栏

Gazebo 窗口的**菜单**栏包括 File、Edit、View、Window、Help。注意 Ubuntu 系统默认设置程序窗口的菜单栏位于屏幕顶部，将鼠标指针移动到屏幕左上角才可以看到。

Edit 菜单包括 Reset Model Poses、Reset World、Building Editor。

(2) View 菜单如图 1-30 所示，可查看仿真模型的 Collisions、Contacts、Joints 等信息，图 1-31 显示的是机器人各个关节的坐标系。

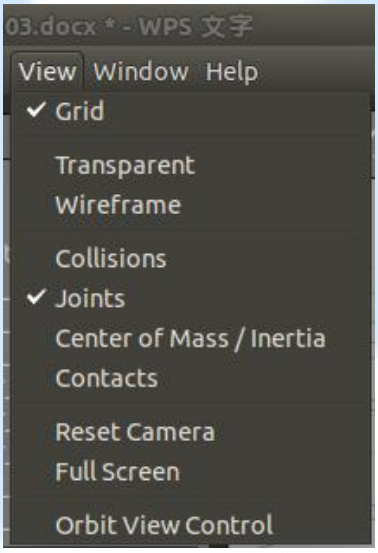


图 1-30 View 菜单

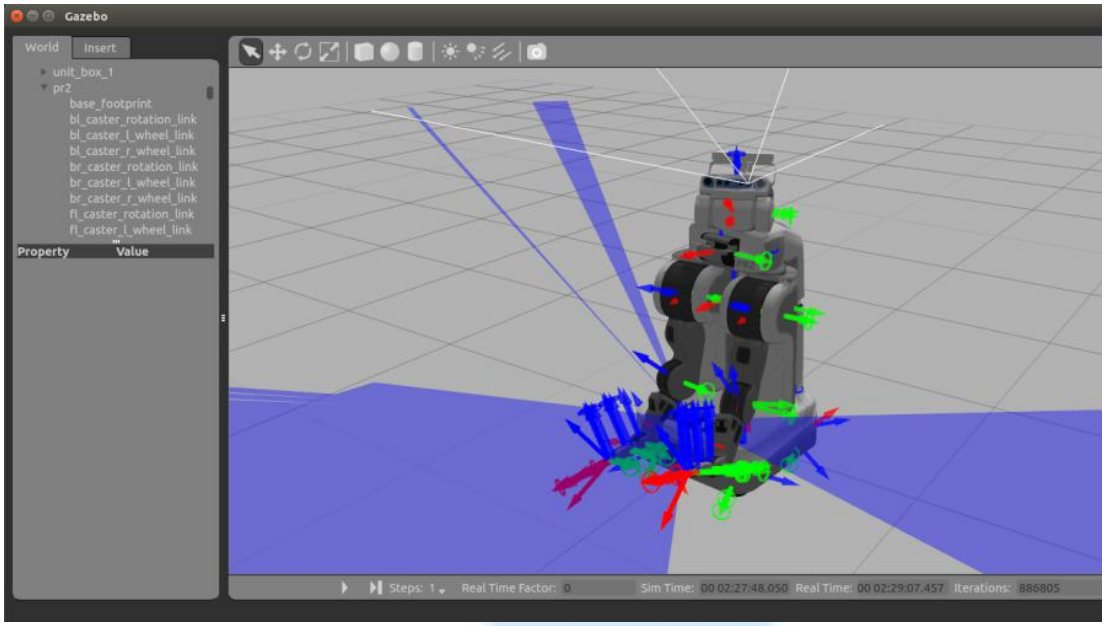


图 1-31 显示关节坐标系

(3) Window 菜单包括 Topic Visualization 和 Log Data 功能。

1.5.2.2 Gazebo 仿真模型

1) 建立仿真模型

前面已经介绍机器人模型文件的建立方法。Gazebo 中的机器人模型还需添加纹理信息的代码，例如：

```
<gazebo reference="base_link">
<material>Gazebo/Blue</material>
</gazebo>
```

2) 建立驱动器和关节的关系

<transmission>是对机器人模型描述的扩展，用来描述驱动器和关节的关系。在 my_robot_sim/urdf 文件夹中新建文件 my_robot.transmissions.xacro，文件的内容如下：

```
1 <?xml version="1.0"?>
2 <robot>
3     <transmission name="myrobot_trans" type="pr2_mechanism_model/SimpleTransmission">
4     <joint name="myrobot_joint2"/>
5     <actuator name="myrobot_motor"/>
6     <mechanicalReduction>1</mechanicalReduction>
7     <motorTorqueConstant>1</motorTorqueConstant>
8     </transmission>
9 </robot>
```

为了将<transmission>添加到机器人模型中，需要在 my_robot.urdf.xacro 文件中添加如下的代码：

```
<xacro:include filename="$(find my_robot_test)/urdf/my_robot.transmissions.xacro" />
```

3) 建立仿真环境

在 my_robot_sim 文件夹中创建一个名为 gazebo 的文件夹，并添加一个名为 my_robot_gazebo.world 的文件，该文件的内容如下：

```
<?xml version="1.0"?>
<gazebo version="1.0">
<world name="default">
    <scene>
      <ambient rgba="0.5 0.5 0.5 1"/>
      <background rgba="0.5 0.5 0.5 1"/>
      <shadows enabled="false"/>
      <grid enabled="true"/>
    </scene>
    <physics type="ode" update_rate="1000">
      <gravity xyz="0 0 -9.8"/>
      <ode>
        <solver type="quick" min_step_size="0.001" iters="10" sor="1.3"/>
        <constraints cfm="0.0" erp="0.1" contact_max_correcting_vel="10.0"
          contact_surface_layer="0.001"/>
      </ode>
    </physics>
    <!-- Ground Plane -->
    <model name="gplane" static="true">
      <origin pose="0 0 0 0 0 0"/>
      <link name="body">
        <collision name="gplane_coll" laser_retro="2000.0">
          <geometry>
            <plane normal="0 0 1"/>
          </geometry>
          <surface>
            <contact>
              <ode kp="1000000.0" kd="1"/>
            </contact>
          </surface>
        </collision>
        <visual name="gplane_vis" cast_shadows="false">
          <geometry>
            <plane normal="0 0 1"/>
          </geometry>
          <material script="Gazebo/Grey"/>
        </visual>
      </link>
    </model>
    <model name="object1">
      <origin pose="0.3 0 0.1 0 0 0"/>
      <link name="body1">
        <inertial mass="1">
          <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1"/>
        </inertial>
        <collision name="body1_Surface">
          <geometry>
            <cylinder length="0.2" radius="0.1"/>
          </geometry>
          <surface>
            <friction>
              <ode>
                <mu>0.799000</mu>
                  <mu2>0.799000</mu2>
                  <fdir1>0.000000 0.000000 0.000000</fdir1>
                  <slip1>0.000000</slip1>
                  <slip2>0.000000</slip2>
```

```
59                  </ode>
60               </friction>
61               <bounce>
62                  <restitution_coefficient>0.000000</restitution_coefficient>
63                  <threshold>100000.000000</threshold>
64               </bounce>
65               <contact>
66                  <ode>
67                     <soft_cfm>0.000000</soft_cfm>
68                     <soft_erp>0.200000</soft_erp>
69                     <kp>100000.000000</kp>
70                     <kd>1.000000</kd>
71                     <max_vel>100.000000</max_vel>
72                     <min_depth>0.001000</min_depth>
73                  </ode>
74               </contact>
75            </surface>
76         </collision>
77         <visual name="body1_Visual">
78            <geometry>
79               <cylinder length="0.2" radius="0.1"/>
80            </geometry>
81            <material>
82               <script>Gazebo/Rocky</script>
83            </material>
84         </visual>
85      </link>
86   </model>
87   <light name="my_light" type="directional" cast_shadows="false">
88      <origin pose="0 0 30 0 0 0"/>
89      <diffuse rgba=".9 .9 .9 1"/>
90      <specular rgba=".1 .1 .1 1"/>
91      <attenuation range="20"/>
92      <direction xyz="0 0 1"/>
93   </light>
94   <light name='point_white' type='point' cast_shadows='0'>
95      <origin pose='-2.000000 -2.000000 5.000000 0.000000 -0.000000 0.000000'/>
96      <diffuse rgba='0.500000 0.500000 0.500000 1.000000'/>
97      <specular rgba='0.100000 0.100000 0.100000 1.000000'/>
98      <attenuation range='10.000000' linear='0.100000' constant='0.200000'
        quadratic='0.000000'/>
99      <direction xyz='0.000000 0.000000 -1.000000'/>
100   </light>
101 </world>
102 </gazebo>
```

代码解释：

4-9 行：设置场景的外表。

10-17 行：设置物理引擎的类型和属性。

19-86 行：创建仿真模型(机器人工作场景中的模型)。

87-100 行：设置仿真环境中的光源。

1.5.2.3 Gazebo 仿真实例

在 my_robot_sim/launch 文件夹中添加一个 launch 文件：my_robot_gazebo.launch，该文件的内容如下：

```
1 <launch>
2     <!-- load world -->
3     <node name="gazebo" pkg="gazebo" type="gazebo" args="$(find my_robot_sim)/gazebo
      /my_robot_gazebo.world " output="screen" respawn="false" />
4
5     <!-- load robot -->
6     <param name="/use_sim_time" value="true"/>
7     <param name="robot_description" command="$(find xacro)/xacro.py '$(find my_robot_sim)
      /urdf/my_robot.urdf.xacro'" />
8     <node name="spawn_myrobot" pkg="gazebo" type="spawn_model" args="-urdf –param
        robot_description -model myrobot -paused=true" respawn="false" output="screen" />
9
10     <!-- start gui -->
11     <node name="gazebo_gui" pkg="gazebo" type="gui" respawn="false" output="screen" />
12 </launch>
```

代码解释：

第 3,4 行加载 world 文件到 gazebo 中，配置仿真环境。

第 7-9 行将机器人模型加载到 gazebo 中。

在终端中运行 my_robot_gazebo.launch， 命令为：

```
$ roslaunch my_robot_sim my_robot_gazebo.launch
```

运行结果如图 1-32 所示。如果机器人没有加载，运行：

```
$ rosrun gazebo_ros spawn_model -urdf -param robot_description -model myrobot -paused=true
```

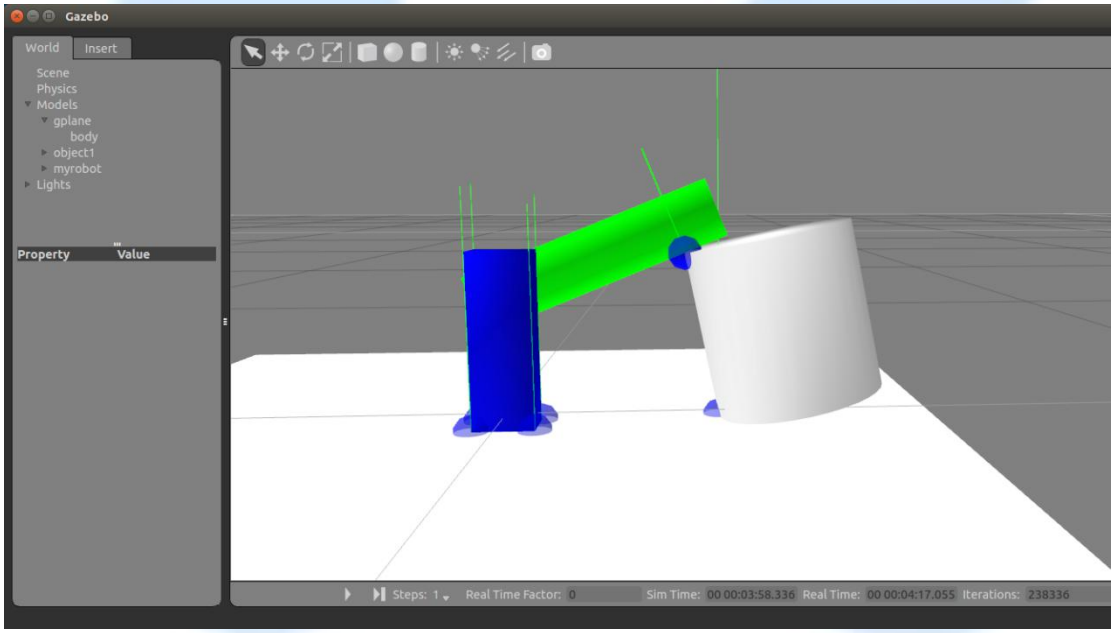


图 1-32 Gazebo 仿真

如果在启动 Gazebo 的窗口使用 Ctrl+C 结束程序，有时会出现其进程 gzserver 和 gzclient 驻留在后台没关闭的情况，导致下一次仿真无法启动成功。对于这种情况，使用如下命令终止进程：

```
$ killall -9 gzserver
$ killall -9 gzclient
```

第二章 MoveIt 运动规划

2.1 MoveIt 构成

2.1.1 系统构架

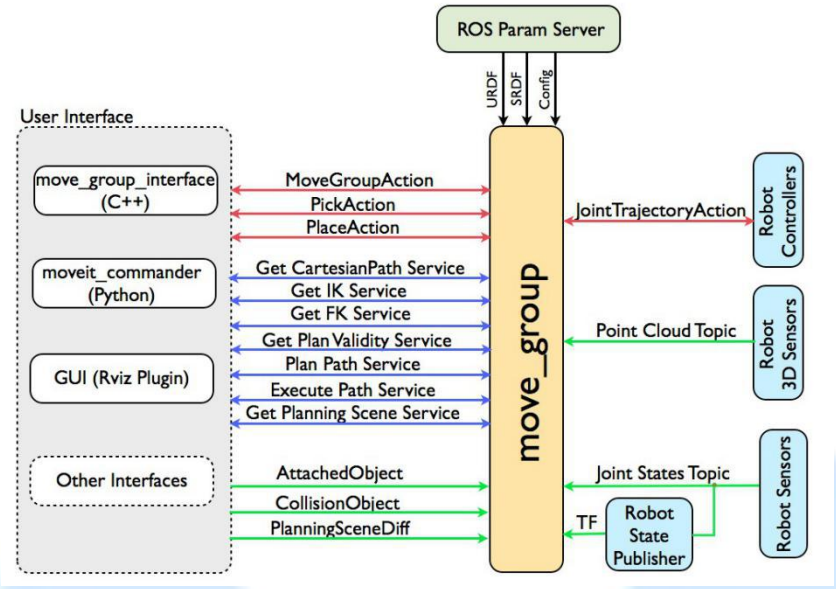


图 2-1 系统架构图

1）move_group 节点

上图是一个高层级的系统架构图，move_group 节点作为一个集成节点，通过加载不同的插件实现各种各样的功能。

2）用户接口

用户可以通过三种方式和 move_group 交互，C++、Python 和图形界面。

3）配置

move_group 作为 ROS 节点需要各种配置参数才能工作，这些参数包括：

URDF-move_group 寻找 robot_description 以获得机器人模型；

SRDF-move_group 通过寻找 robot_semantic_description 参数获得机器人运动链，碰撞检测矩阵等参数；

Moveit 配置-Moveit 通过参数服务器获得机器人关节转角限制，运动学描述，运动规划，感知等参数，这些配置文件默认位于 Moveit 配置包的 config 目录下。

2.1.2 机器人接口

move_group 通过 topic，action 和机器人交互，它需要获取机器人的当前状态信息，比如关节位置、点云等机器人传感器数据，也会向机器人的运动控制器发送轨迹指令。

1）关节状态信息

move_group 监听 joint_states 以获取当前机器人的状态信息，比如由此获得关节转角。

move_group 可以同时监听多个发布到此 topic 的发布器，可能每个发布器只发布机器人部分关节的信息。

2）坐标变换信息

move_group 通过 ROS 和 TF 库获得机器人的坐标变换信息。据此 move_group 可以获得机器人各部分的姿态。只因 Moveit 只监听 TF，因此要保证 robot_state_publisher 处于运行状态。

3）控制器接口

move_group 通过 FollowJointTrajectoryAction 接口和机器人控制器进行交互，运行在机器人上的 action server 提供此接口，move_group 本身不提供此接口。move_group 启动时会初始化一个 action client 和机器人上的 action server 进行通讯。

4）规划场景

move_group 使用规划场景监视器维护一个规划场景，它不仅表示当前的环境状态，也包含机器人当前的状态信息。机器人的状态信息里包括机器人当前携带了什么物体，这些物体被认为是和机器人是一个整体，规划场景部分会详细介绍这些功能。

5）可扩展能力

move_group 被设计为很容易扩展，单个功能，比如 pick place，运动学，运动规划等实际上是以插件的形式提供的，这些插件共用一个基类。插件可以通过一系列的 ROS 参数及插件库进行配置。绝大部分用户不必去配置 move_group 插件，它们通常在 launch 文件中已自动配置。

2.1.3 运动规划

1）运动规划插件

MoveIt 通过插件接口和运动规划器交互，这样 MoveIt 很容易使用不同类型的规划器，使之更有扩展性。同运动规划器交互的接口是通过 ros action 或 service 实现的，默认的规划器是 OMPL。

2）运动规划请求

运动规划请求声明了用户想要规划器做些什么，比如说，用户可以请求规划器将机械臂移动到不同的位置（在关节空间或笛卡尔空间）。默认会执行碰撞检测功能（含自身碰撞检测）。您也可以把一个物体附加到末端执行器上或机器人的其他部位，比如当机器人拿起一个物体时，这样运动规划时，也会把该物体考虑进去。您也可以声明一些运动约束条件，MoveIt 提供的内置的规划约束是运动学约束，包括：

位置约束-把连杆位置约束在特定区域内；

方位约束-约束连杆的 roll，pitch，yaw 等绕轴的旋转；

可视性约束-约束连杆上的一个点处于特定传感器的视锥范围内；

关节约束-约束关节转角最小最大值；

用户声明的约束-用户自定义的其他约束。

3）运动规划结果

运动规划将会产生对应于规划请求的运动轨迹，该轨迹可以让机器人运动到指定位置。注意的是，由 move_group 产生的轨迹不仅仅是一条路径，move_group 会根据速度和加速度限制产生满足关节约束的轨迹。

4）运动规划流程

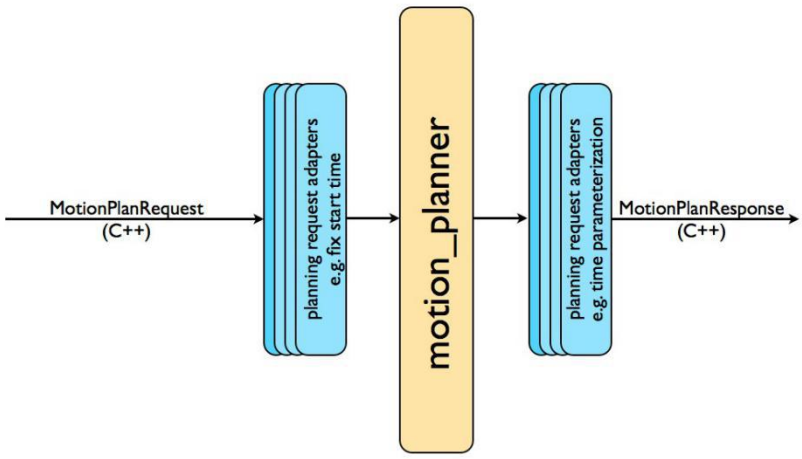


图 2-2 运动规划流程

上图表示了运动规划器和规划请求适配器的系统工作。规划请求适配器可以对规划请求进行预处理，也可以对规划响应进行后期处理。预处理在有些情况下是很有用的，比如机器人的初始状态略微超出了关节转角限制。有些操作也需要后期处理，比如把规划的路径转变成基于时间参数的轨迹。MoveIt 提供了一系列的规划适配器用于完成一些特定功能。

初始状态适配器：此适配器让机器人初始状态处于关节限制的范围之内。比如当机器人的关节限制没有配置好，机器人一个或多个关节超出关节限制，此时运动规划器将无法求解。此适配器可以将初始状态设置在关节转角限制范围内，当然这不是非常正确的方法，比如，某些情况下关节真的超出限制很多。有一个参数可以设置偏离关节限制到达多少时，才采用此适配器进行修复。

工作空间适配器：声明机器人的工作空间，默认是 10m x 10m x 10m 的立方体空间。此工作空间只在规划请求中不含这些限制时声明。

初始状态碰撞适配器：此适配器用于机器人初始状态处于碰撞中时，当机器人的初始状态处于碰撞时，可以通过 jiggle_factor 参数让机器人初始状态进行微小移动，另外一个参数可以声明在放弃之前的随机扰动次数。

初始状态路径约束适配器：此适配器工作在初始状态不遵循路径约束，它将会规划一条机器人当前状态到满足路径约束位置的路径。新的位置将作为规划的初始状态。

时间参数化：运动规划器通常只产生运动学上的路径，这个路径一般不符合速度和加速度约束。此适配器将根据速度和加速度的约束对规划的路径进行时间参数化。

5）OMPL

OMPL（Open Motion Planning Library）是一个开源的运动规划库，主要实现了一些随机规划算法。MoveIt 直接集成了 OMPL 并将它作为默认的运动规划器。OMPL 中的规划器是抽象的，它没有机器人的概念。MoveIt 将它配置为后端，用它解决机器人问题。

2.1.4 规划场景

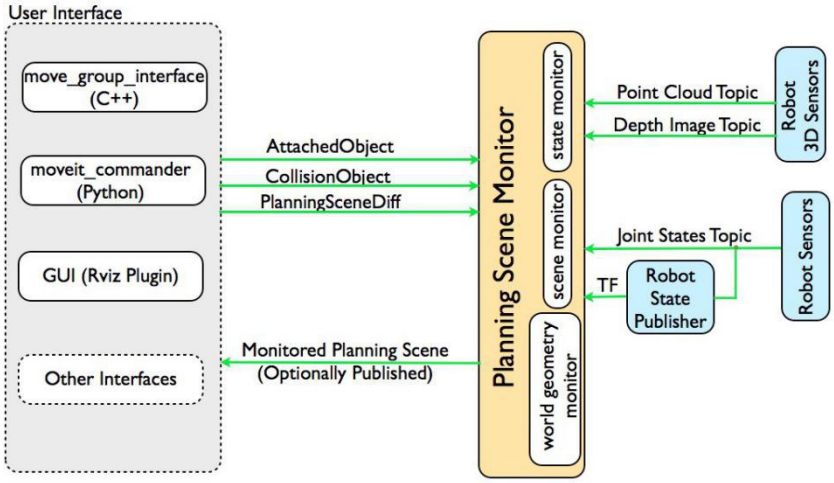


图 2-3 规划场景图

规划场景用于表示机器人所处的环境及机器人自身状态，它由 move_group 内部的场景监视器维护管理，场景监视器主要监听：

状态信息；joint_states topic

传感器信息；使用下述的空间几何监视器

空间几何信息：来自于用户输入

1）空间几何监视器

空间几何监视器根据机器人上的传感器数据和用户输入信息建立机器人周围的空间几何模型。它用下述的 occupancy_map_monitor 建立机器人当前环境的三维模型，并通过 planning_scene 完善该模型。

2）三维感知

MoveIt 的三维感知是由 occupancy_map_monitor 处理的，occupancy_map_monitor 使用插件的机制处理不同类型的传感器数据，MoveIt 默认支持两种数据输入：

点云：由 point_cloud_occupancy_map 插件处理；

深度图像：由 depth_image_occupancy_map 插件处理。

您也可以自己添加插件。

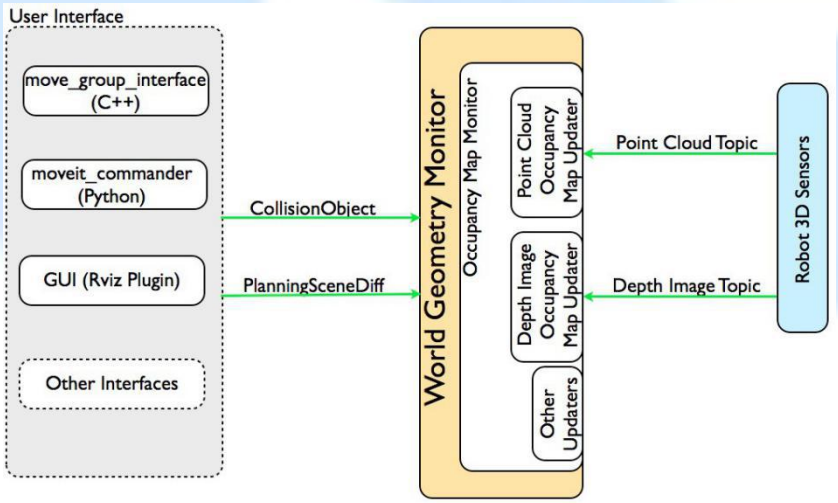


图 2-4 三维感知图

2.1.5 运动学求解与碰撞检测

1）运动学插件

MoveIt 运动学求解也是使用插件机制，可以让用户自己写运动学解算算法。正运动学及雅克比集成在 RobotState 类中，默认的逆运动学求解插件是基于雅克比的数值求解器。它在生成运动规划包时已经自动配置好了。

2）IKFast 插件

通常，我们会实现自己的运动学求解器，比如 PR2 就有自己的运动学求解器。一个常用的做法是使用 IKFast 软件包产生自己机器人运动学求解器的 C++代码。

3）碰撞检测

MoveIt 的碰撞检测功能集成在 planning scene 的 CollisionWorld 中，默认情况下用户不必关心碰撞检测。MoveIt 的碰撞检测功能是通过 FCL 实现的。

4）碰撞物体

MoveIt 支持几种不同类型的物体，包括：

1.网格；

2.基本形状，比如圆球、圆柱、立方体；

3.Octomap，Octomap 对象的物体可以直接用于碰撞检测。

5）FCL

FCL（Flexible Collision Library）是 MoveIt 所用的主要的碰撞检测库。

6）碰撞矩阵（ACM）

碰撞检测是一项很耗费电脑资源的任务，通常占用了运动规划 90%左右的计算开销。ACM 通过二进制编码找到需要进行碰撞检测的两个物体，在 ACM 中如果两个物体的值为 1，表示这两个物体间不需要进行碰撞检测，可能因为这两个物体离的很远不可能发生碰撞。

2.1.6 轨迹处理

规划器通常只产生所谓的路径，此路径不包含时间信息。MoveIt 内有轨迹处理方案，可以根据最大速度、最大加速度的限制，将路径转变成包含时间信息的轨迹。关节限制的信息在 joint_linits.yaml 中声明。

2.2 MoveIt 接口

2.2.1 配置 Sawyer 运动规划包

MoveIt 提供了一个图像界面，您可以通过此图形界面配置您机器人的运动链等，以和 MoveIt 交互。它最主要的功能是为机器人生成一个 SRDF（Semantic Robot Description Format）文件。另外，它也会生成诸如控制器管理器，运动学配置的配置文件和一系列的 launch 文件。我们将以 Sawyer 为例，学习如何配置运动规划包。

1）启动设置助手

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

设置助手启动时有两个选项，配置已有的 MoveIt 包和建立新的 MoveIt 包。这里我们选择建立新的 MoveIt 包。

建立新的规划包时要加载 URDF 文件，选中 Sawyer 的 URDF 文件所在路径，比如../sawyer_description/urdf/sawyer.urdf，然后点击 Load Files 按钮加载。

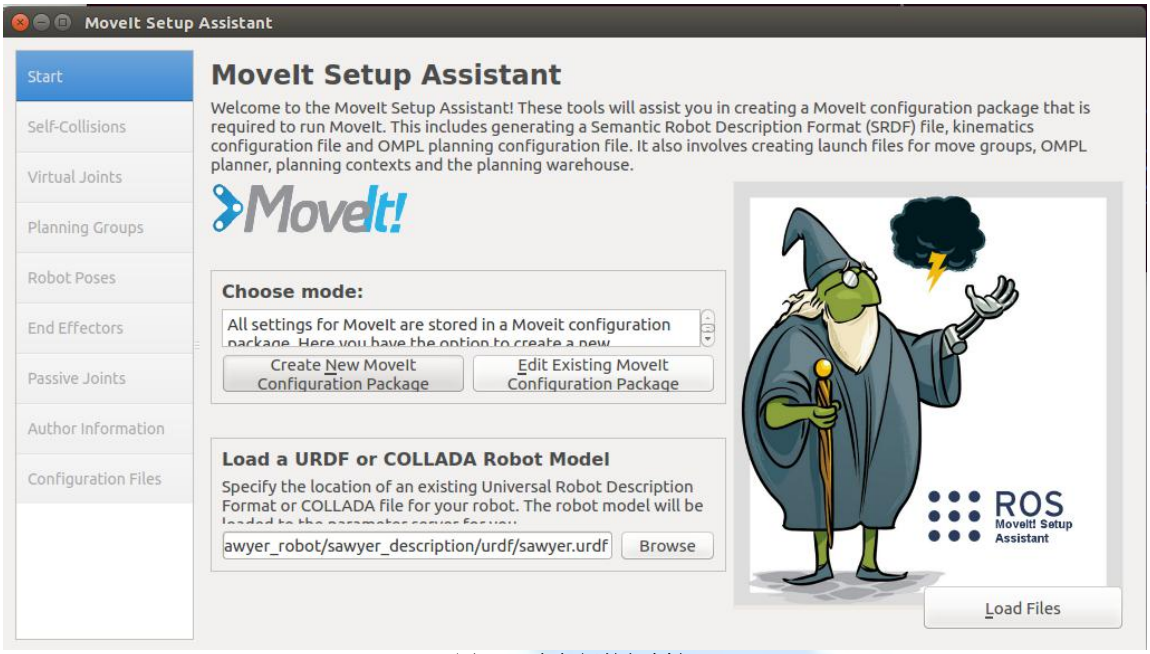


图 2-5 建立新的规划包

装载成功后会显示 Sawyer 的三维模型：

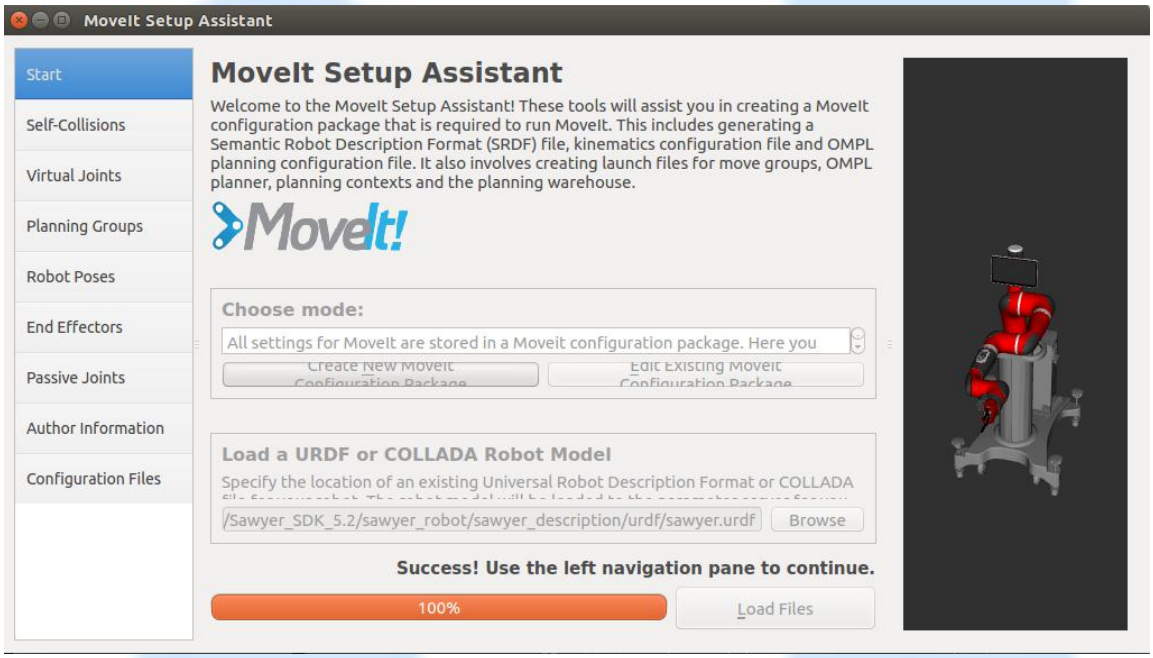


图 2-6 Sawyer 三维模型

2)生成自我碰撞检测矩阵

为了降低碰撞检测的开销，可以预先生成机器人不同连杆之间的碰撞关系，一些不会发生碰撞的连杆可以禁用碰撞检测。那些总是碰撞、从不碰撞、相邻的连杆之间的碰撞检测通常会被取消。采样密度声明了机器人随机自我碰撞检测的采样次数，更高的密度需要较长的计算时间，但能更好的给出碰撞检测结果。默认的是采样 10000 次，碰撞检测是并行的。

选中 Self_Collisions 面板，选择采样密度，点击 Generate Collision Matrix，等待几十秒，则可以生成碰撞检测矩阵。

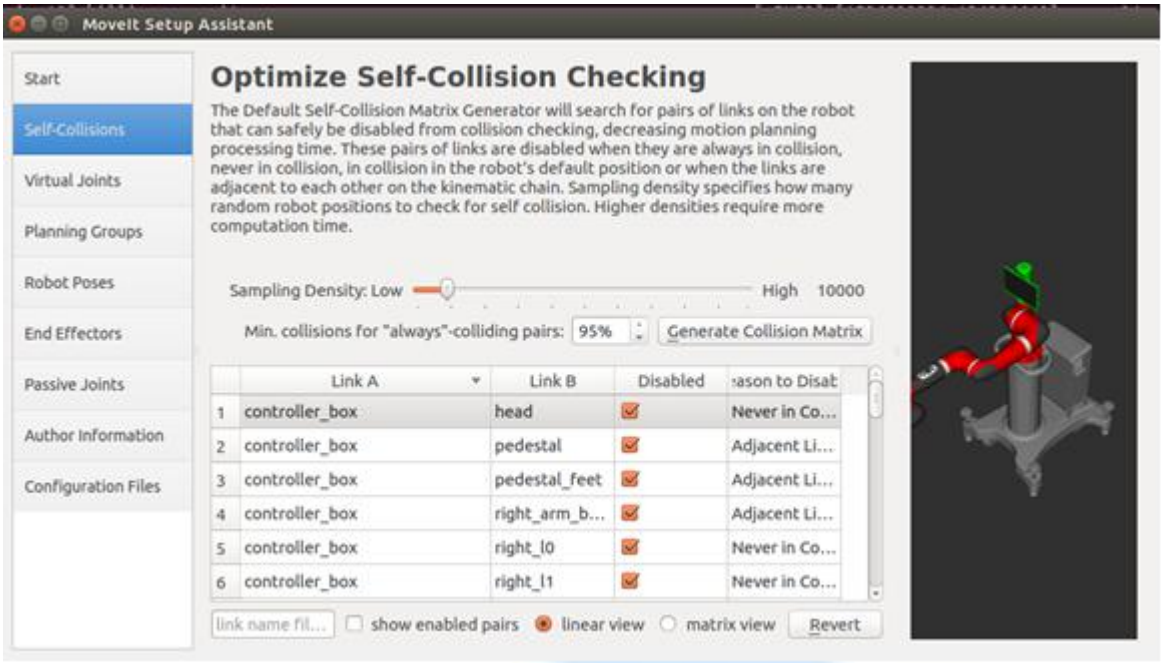


图 2-7　生成碰撞检测矩阵

3）添加虚拟关节

虚拟关节表示机器人和世界的关系，比如带移动平台的机器人，可以在地面运动，它和地面之间可以设置一个平面类型的虚拟关节。Sawyer 的虚拟关节可以设置为固定，或其他类型，不会对 Sawyer 产生实质影响，这里添加了一个浮动虚拟关节。

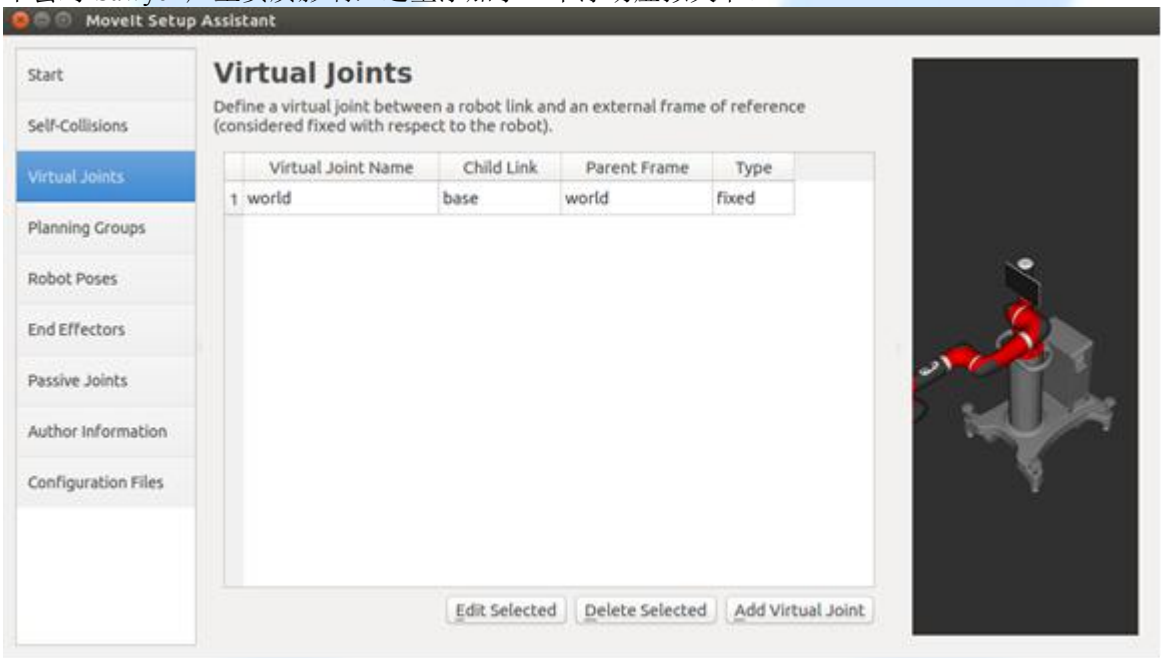


图 2-8　添加浮动虚拟关节

4）配置运动规划组

运动规划组是运动规划时的所需要用到的运动链，可以是一组关节，也可以配置为从机械臂第一个连杆到最后一个连杆，也可以是一些子规划组。运动规划组配置时一般要选择机械臂相应的关节，给规划组配置运动求解器、规划时间、规划次数等。

我们为 Sawyer 设置了一个规划组，right_arm。

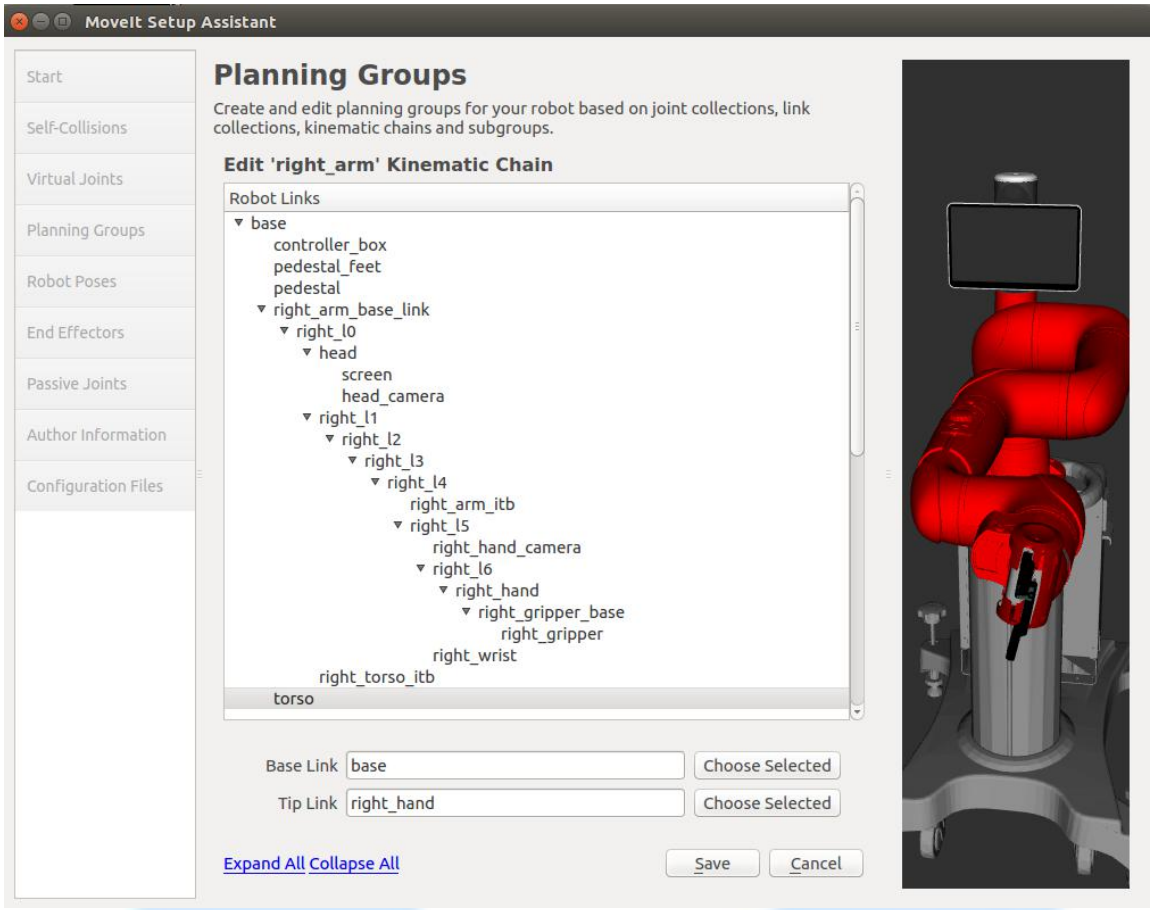


图 2-9 设置 right_arm 规划组步骤 1

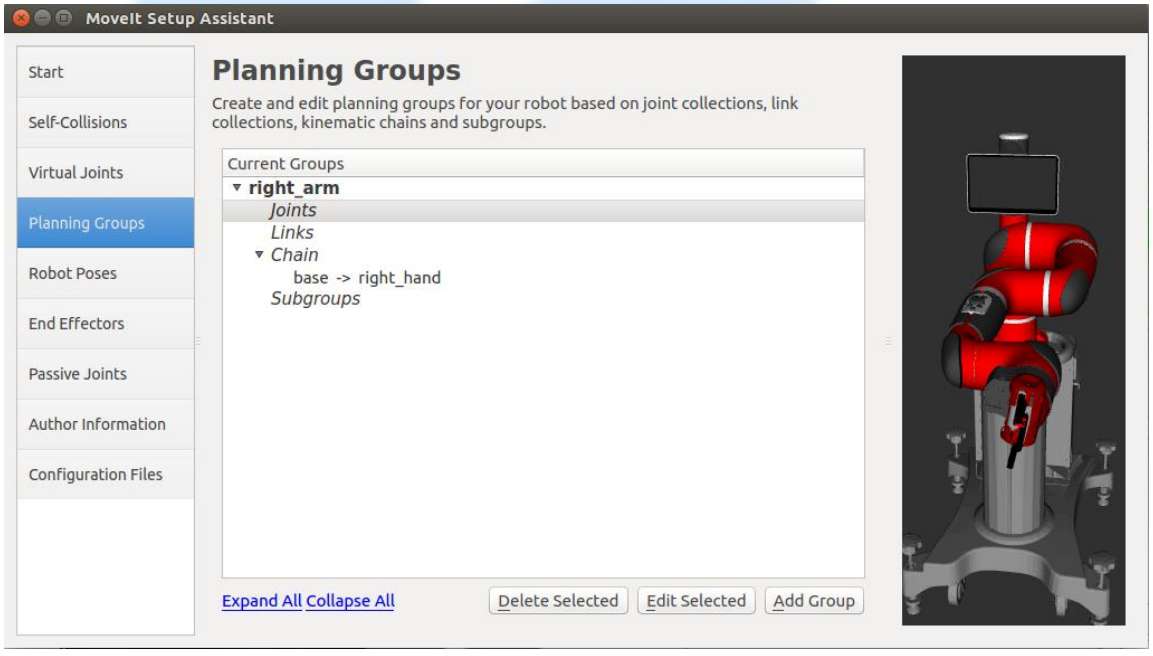


图 2-10 设置 right_arm 规划组步骤 2

5）定义机器人姿态

可以给机器人预先定义一些姿态，方便运动规划时直接调用，比如为 Sawyer 的手臂设置关

节零旋转姿态。

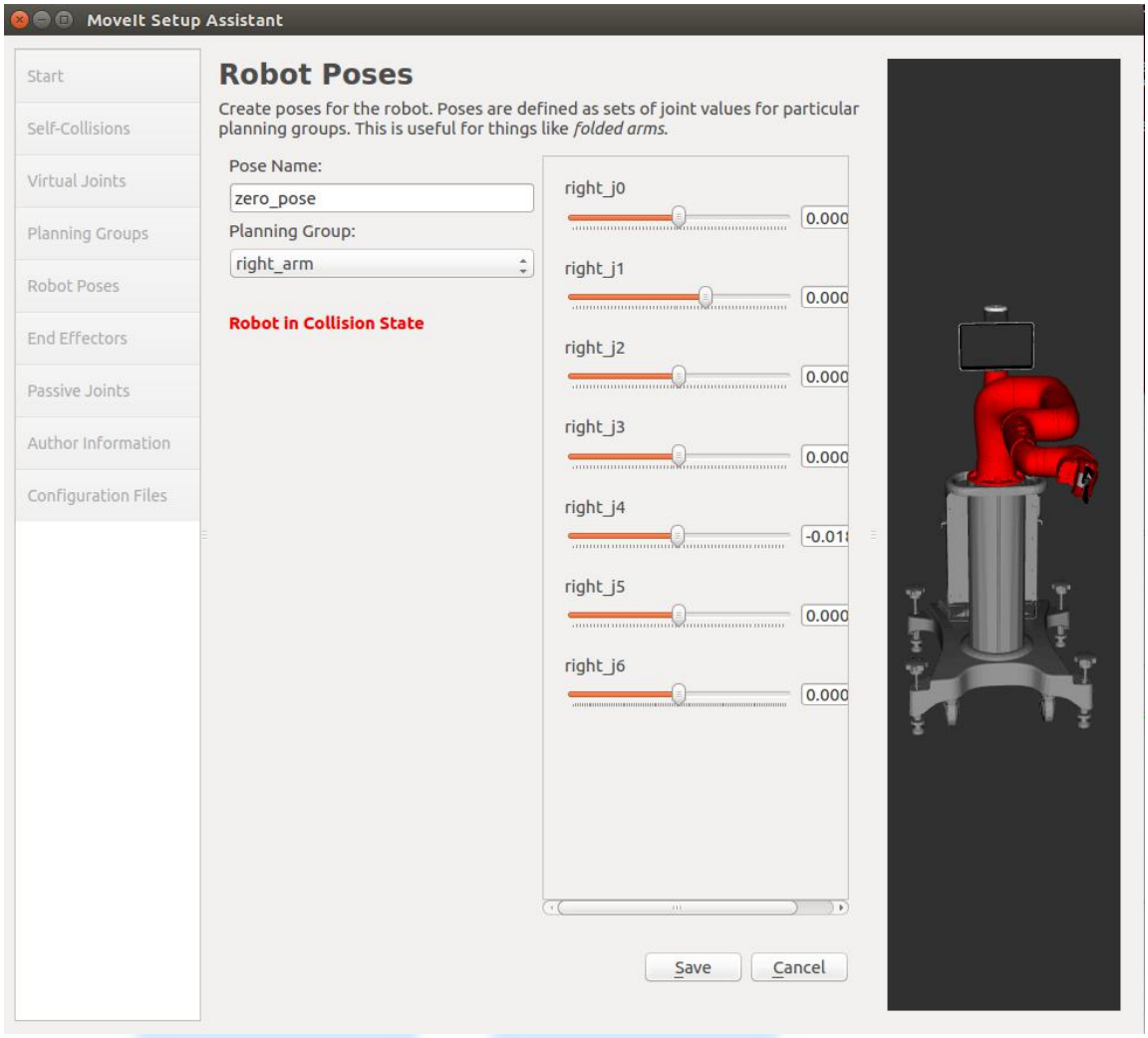


图 2-11 设置关节零旋转姿态

6）生成配置文件

Sawyer 机器人没有被动关节，因此我们直接生成规划配置文件。选择您的工作空间，填入规划包的名字，一般是 *robotname*_moveit_config 点击 Generate Package 即可为我们的机器人生成一系列的配置和启动文件。

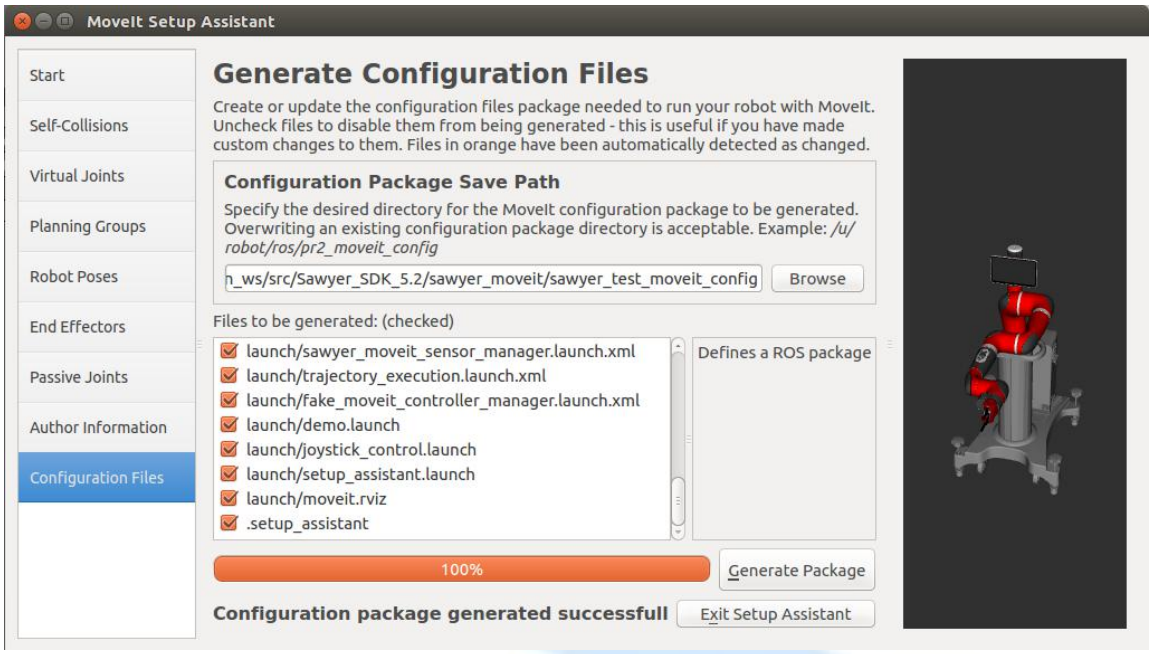


图 2-12 配置和启动文件

7）验证规划包

启动：

```
$ roslaunch sawyer_test_moveit_config demo.launch
```

如下图所示，您将看到 Sawyer 机械臂末端有彩色的圆圈和箭头，代表机械臂末端的平动和转动自由度，至此，Sawyer 运动规划包已经配置完成了。

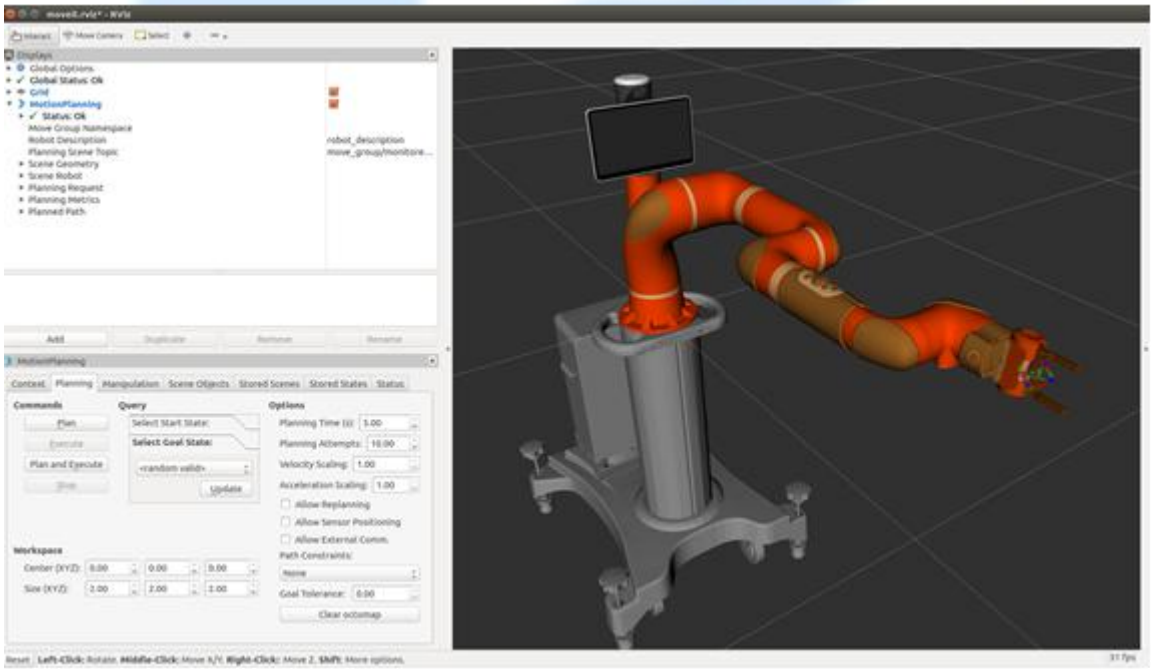


图 2-13 Sawyer 运动规划包配置完成

2.2.2 move_group 接口

和 MoveIt 交互式主要是通过 move_group 的 C++、Python API 或 ros action，service 以及 topic。move_group 接口提供了大量针对运动规划的操作，比如设置关节目标，设置机械臂末端位置，创建运动规划，移动机器人等。

这里以 Python 接口为例，介绍如何使用 MoveIt 的功能。

1）导入 Python 模块 import sys

```
import copy
import rospy
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
```

2）初始化 moveit_commander 及 rospy

```
print "============ Starting tutorial setup"
moveit_commander.roscpp_initialize(sys.argv)
rospy.init_node('move_group_python_interface_tutorial', anonymous=True)
```

3）实例化一个 RobotCommander 对象，用于和机器人交互

```
robot = moveit_commander.RobotCommander()
```

4）实例化一个 PlanningSceneInterface 对象，它代表了机器人周围的环境

```
scene = moveit_commander.PlanningSceneInterface()
```

5）实例化一个 MoveGroupCommander 对象

该对象代表了一个运动规划组，在这里我们实例化左臂。此接口可用于在左臂上进行路径规划，并执行规划的路径。

```
group = moveit_commander.MoveGroupCommander("left_arm")
```

6）在 Rviz 中显示规划的路径

```
display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path',
                                               moveit_msgs.msg.DisplayTrajectory)
print "============ Waiting for RVIZ..."
rospy.sleep(10)
print "============ Starting tutorial "
```

7）获得基本信息

获得机器人参考坐标系：

```
print "============ Reference frame: %s" % group.get_planning_frame()
```

打印终端效应器的名字：

```
print "============ Reference frame: %s" %group.get_end_effector_link()
```

获得机器人所有的规划组：

```
print "============ Robot Groups:"
print robot.get_group_names()
```

获得机器人当前状态：

```
print "============ Printing robot state"
print robot.get_current_state()
print "============"
```

8）规划到目标点

设置目标点：

```
print "============ Generating plan 1"
pose_target = geometry_msgs.msg.Pose()
pose_target.orientation.w = 1.0
pose_target.position.x = 0.7
pose_target.position.y = -0.05
pose_target.position.z = 1.1
group.set_pose_target(pose_target)
```

执行规划命令。只请求规划路径，不实际执行，规划成功后会在 Rviz 中显示出来：

```
plan1 = group.plan()
print "============ Waiting while RVIZ displays plan1..."
rospy.sleep(5)
```

发布规划好的路径，以方便在 Rviz 中查看：

```
print "============ Visualizing plan1"
display_trajectory = moveit_msgs.msg.DisplayTrajectory()

display_trajectory.trajectory_start = robot.get_current_state()
display_trajectory.trajectory.append(plan1) display_trajectory_publisher.publish(display_trajectory);

print "============ Waiting while plan1 is visualized (again)..."
rospy.sleep(5)
```

9）将机器人移动到目标点

这个过程和上面的过程类似，我们调用的是 go()函数，这个函数有个 wait 参数，设置为 True 时，程序会阻塞直到运动完成为止，失败则直接返回。

```
# Uncomment below line when working with a real robot
# group.go(wait=True)
```

10）关节空间运动规划

除了指定机械臂的末端位置外，也可以设置关节转角值。

首先清除之前设置的目标值：

```
group.clear_pose_targets()
```

获得当前关节转角值：

```
group_variable_values = group.get_current_joint_values()
print "============= Joint values: ", group_variable_values
```

修改其中的一个关节值，执行规划命令：

```
group_variable_values[0] = 1.0
group.set_joint_value_target(group_variable_values)
plan2 = group.plan()

print "============= Waiting while RVIZ displays plan2..."
rospy.sleep(5)
```

11）笛卡尔路径规划

可以声明一系列的路点让机器人沿这些路点形成的路径运动：

```
waypoints = []

# start with the current pose
waypoints.append(group.get_current_pose().pose)

# first orient gripper and move forward (+x)
wpose = geometry_msgs.msg.Pose()
wpose.orientation.w = 1.0
wpose.position.x = waypoints[0].position.x + 0.1
wpose.position.y = waypoints[0].position.y
wpose.position.z = waypoints[0].position.z
waypoints.append(copy.deepcopy(wpose))

# second move down
wpose.position.z -= 0.10
waypoints.append(copy.deepcopy(wpose))

# third move to the side
wpose.position.y += 0.05
waypoints.append(copy.deepcopy(wpose))
```

如果想让机械臂末端沿路径以 1cm 的间距插值，我们可以设置 jump_threshold 的值 0.01，这里设置为 0.0，禁用它。

```
(plan3, fraction) = group.compute_cartesian_path(
                                   waypoints,   # waypoints to follow
                                   0.01,        # eef_step
                                   0.0)         # jump_threshold
```

```
print "============ Waiting while RVIZ displays plan3..."
rospy.sleep(5)
```

2.2.3 planning scene 接口

这节我们介绍如何通过 planning scene 接口向规划场景中添加或移除物体。

1）设定 Publisher

可以通过 ROS 特定的 diffs topic 和规划场景交互，一个规划场景的 diff 指的是当前规划场景和新的规划场景之间的差异。

```
ros::Publisher planning_scene_diff_publisher = node_handle.advertise
                                        <moveit_msgs::PlanningScene>("planning_scene", 1);
while(planning_scene_diff_publisher.getNumSubscribers() < 1)
{
    ros::WallDuration sleep_t(0.5); sleep_t.sleep();
}
```

2）定义要附加的物体

定义要添加到规划场景或要从当前场景中移除的物体，以及要附加到机器人上的物体。

```
moveit_msgs::AttachedCollisionObject attached_object;
attached_object.link_name = "r_wrist_roll_link";

/* The header must contain a valid TF frame*/
attached_object.object.header.frame_id = "r_wrist_roll_link";

/* The id of the object */
attached_object.object.id = "box";

/* A default pose */
geometry_msgs::Pose pose;
pose.orientation.w = 1.0;

/* Define a box to be attached */
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.1;
primitive.dimensions[1] = 0.1;
primitive.dimensions[2] = 0.1;

attached_object.object.primitives.push_back(primitive);
attached_object.object.primitive_poses.push_back(pose);
```

上面代码中，为了将物体附件到机器人上，需要加一个 ADD 的操作：

```
attached_object.object.operation = attached_object.object.ADD;
```

3）将物体添加到场景中

通过将物体添加到场景中的物体碰撞集中实现添加物体，只使用 attached_object 消息的 object 域。

```
ROS_INFO("Adding the object into the world at the location of the right wrist.");
moveit_msgs::PlanningScene planning_scene;
planning_scene.world.collision_objects.push_back(attached_object.object);
planning_scene.is_diff = true;
planning_scene_diff_publisher.publish(planning_scene);
sleep_time.sleep();
```

4）场景的同步及异步更新

move_group 使用场景 diff 时，有两种不同的机制：

通过 rosservice 发送 diff 请求时，场景会阻塞知道 diff 已经应用到场景了（同步更新）；

通过 topic 发送 diff 消息，消息会继续发送而不管 diff 是否已经应用到场景了（异步更新）。

大部分的教程使用的是后一个机制（发布 diff 消息后，设置一定的休眠时间），可以用下面的调用服务的方法较好的替代 topic 的方式。

```
ros::ServiceClient    planning_scene_diff_client = node_handle.serviceClient<moveit_msgs::
                                                   ApplyPlanningScene> ("apply_plann ing_scene");
planning_scene_diff_client.waitForExistence();
moveit_msgs::ApplyPlanningScene srv;
srv.request.scene = planning_scene;
planning_scene_diff_client.call(srv);
```

5）将物体附加到机器人上

当机器人拿起一个物体时，我们需要把该物体附加到机器人上，这样执行运动规划时，规划器会把该物体作为机器人的一部分考虑进去。

将物体附加到机器人上有两个步骤，一是把它从环境中移除，再是把它附加到机器人上。

```
/* First, define the REMOVE object message*/
moveit_msgs::CollisionObject remove_object;
remove_object.id = "box";
remove_object.header.frame_id = "odom_combined";
remove_object.operation = remove_object.REMOVE;
```

要先清除 diff 消息中的已附加物体和碰撞物体。

```
/* Carry out the REMOVE + ATTACH operation */
ROS_INFO("Attaching the object to the right wrist and removing it from the world.");
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(remove_object);
planning_scene.robot_state.attached_collision_objects.push_back(attache d_object);
```

```
planning_scene_diff_publisher.publish(planning_scene);
sleep_time.sleep();
```

6）解除机器人附加的物体

解除附加到机器人上的物体时同样也需要两步，一是从机器人上解除附加，再是把物体插入到环境中。

```
/* First, define the DETACH object message*/
moveit_msgs::AttachedCollisionObject detach_object;
detach_object.object.id = "box";
detach_object.link_name = "r_wrist_roll_link";
detach_object.object.operation = attached_object.object.REMOVE;
```

同样要先清除 diff 消息中已附加的物体和碰撞物体。

```
/* Carry out the DETACH + ADD operation */
ROS_INFO("Detaching the object from the robot and returning it to the world.");
planning_scene.robot_state.attached_collision_objects.clear();
planning_scene.robot_state.attached_collision_objects.push_back(detach_ object);
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(attached_object.object);
planning_scene_diff_publisher.publish(planning_scene);
sleep_time.sleep();
```

7）将物体从规划场景中移除

将物体从规划场景中移除时，只需要用到之前定义的移除物体的消息，同样注意，清除 diff 消息中含已附加物体和碰撞物体的部分，以免误操作这些物体。

```
ROS_INFO("Removing the object from the world.");
planning_scene.robot_state.attached_collision_objects.clear();
planning_scene.world.collision_objects.clear();
planning_scene.world.collision_objects.push_back(remove_object);
planning_scene_diff_publisher.publish(planning_scene);
```

第三章 Sawyer SDK

3.1 安装与配置

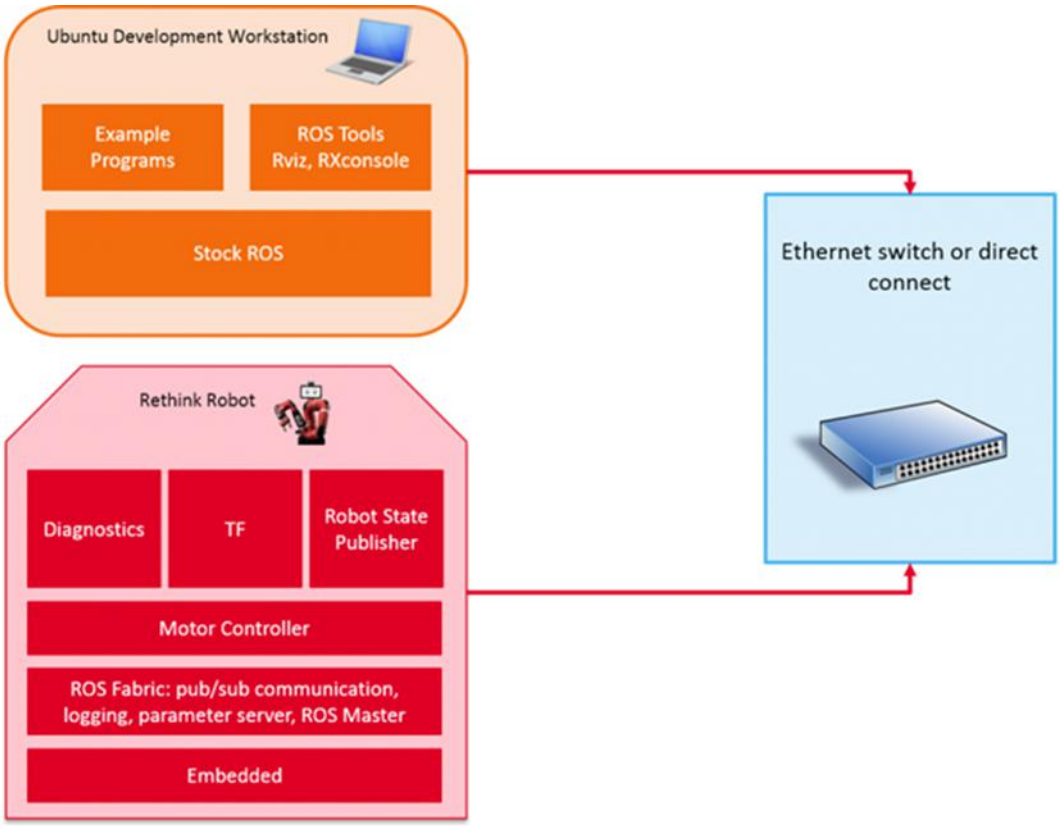


图 3-1 系统布局图

SDK 通过 ROS (Robot Operating System)与机器人交互。Sawyer 机器人提供一个独立的 ROS Master, 这样任何工作站电脑（位于这个 ROS 主机网络中）能够经由 ROS APIs 连接和控制 Sawyer 机器人。

3.1.1 下载 Sawyer SDK

1、Sawyer 研究版的官网地址为：sdk.rethinkrobotics.com/intera/Main_Page。

2、Sawyer 研究版的 github 地址为：https://github.com/RethinkRobotics。点击 Sawyer_robot 进入介绍页，里面有 Sawyer 研究版的 SDK 官网链接。

3、也可以在搜索栏直接输入 github RethinkRobotics，找到相应搜索项进入网站

4、下载 SDK：进入 https://github.com/RethinkRobotics/intera_sdk，点击 Clone or download 绿色按钮，选择 Download ZIP。

或者在终端切入到工作区间的 src 文件夹通过以下命令来获取。

```
$ cd ~/ros_ws/src
$ wstool init .
$ git clone https://github.com/RethinkRobotics/sawyer_robot.git
```

```
$ wstool merge sawyer_robot/sawyer_robot.rosinstall
$ wstool update
$ source /opt/ros/indigo/setup.bash
$ cd ~/ros_ws
$ catkin_make
```

5、API 介绍网址：http://rethinkrobotics.github.io/intera_sdk_docs

3.1.2 工作环境搭建

1）用无线实现 Sawyer 与电脑连接

首先是将 Sawyer 连接到路由器，然后电脑连接路由器的 wifi 就实现了网络上的连接。然后将 intera.sh 里的 robot_hostname 改为 Sawyer 控制器下方的标识，一般为"021605CP000xx.local"。

```
$ cp ~/ros_ws/src/intera_sdk/intera.sh ~/ros_ws
$ cd ~/ros_ws
$ gedit intera.sh
```

修改之后，进入工作区间输入：

```
$ cd ~/ros_ws
$ ./intera.sh
```

便能与 Sawyer 相连了。

2）用网线实现 Sawyer 与电脑直连

首先需要将 Sawyer 的网络模式设置为自动获取地址，即进入 Sawyer 的切换系统界面，然后点击 configuration 配置，选中 dynamic ip,保存后返回主界面重启机器。

然后确保电脑与 Sawyer 是用网线相连的，通过命令

```
$ sudo ufw disable
```

关闭防火墙，关闭 Enable Networking，查询网线链接的网口，若为 eth1，通过命令

```
$ ifconfig eth1
```

查看网口状态，正常的话通过命令

```
$ sudo avahi-autoipd eth1
```

分配 ip 给 eth1 网口，如果出现以下信息，则为分配成功：

```
Found user 'avahi-autoipd' (UID 104) and group 'avahi-autoipd' (GID 111).
Successfully called chroot().
Successfully dropped root privileges.
Starting with address 169.254.10.85
Callout BIND, address 169.254.8.16 on interface eth0
Successfully claimed IP address 169.254.10.85
```

不要关闭该终端，使其在后台运行。

打开一个新终端，输入命令

```
$ avahi-browse -a -r
```

查看是否有 Sawyer 的 hostname，以及给 Sawyer 分配的 address 是否与电脑的 address 在同一个网段。

使用命令

```
$ping '#sawyer_hostname'.local
```

查看是否能 ping 通，如果 ping 通了就说明连接成功了。

然后就可以输入

```
$ ./intera.sh
```

进入 Sawyer 的环境控制 Sawyer 了。

3.1.3 Hello Robot

通过命令

```
# Move to root of our catkin workspace
$ cd ~/ros_ws
$ source /opt/ros/indigo/setup.bash
$ catkin_make
# Source intera.sh script
$ ./intera.sh
# Identify the ROS Master location
$ env | grep ROS_MASTER_URI
```

查看 Sawyer 的 robot_hostname ；

使用命令

```
$ ping 021605CP00006.local
```

查看能否连通；

```
cothink@cothinkTab3:~$ ping 021605CP00006.local
PING 021605CP00006.local (192.168.8.100) 56(84) bytes of data.
64 bytes from 192.168.8.100: icmp_seq=1 ttl=64 time=4.47 ms
64 bytes from 192.168.8.100: icmp_seq=2 ttl=64 time=9.43 ms
64 bytes from 192.168.8.100: icmp_seq=3 ttl=64 time=2.35 ms
64 bytes from 192.168.8.100: icmp_seq=4 ttl=64 time=2.55 ms
64 bytes from 192.168.8.100: icmp_seq=5 ttl=64 time=2.40 ms
64 bytes from 192.168.8.100: icmp_seq=6 ttl=64 time=2.27 ms
```

图 3-2　ping 通示意图

通过命令

```
$ rostopic echo /robot/joint_states
```

查看是否能打印关节信息；
使用命令

```
$ rosrun intera_interface enable_robot.py -e
```

如果显示 Robot Enabled 表示已经使动。
使用命令

```
$ rosrun intera_examples joint_torque_springs.py
```

使机器人处于一个自然的状态。

测试一个例子去了解如何移动 Sawyer 的关节：
首先要保证 Sawyer 已经使动，最好输入一次命令

```
$ rosrun intera_interface enable_robot.py -e
```

确保使动。
在终端上依次输入：

```
$ python
# 进入到 python 的运行界面
# 导入 rospy 环境
>>> import rospy
# 导入 Sawyer 的 intera_interface 环境
>>> import intera_interface
# 初始化 ROS 节点
>>> rospy.init_node('Hello_Sawyer')
# 连接 Sawyer 的右臂
>>> limb = intera_interface.Limb('right')
# 得到右臂目前关节点的位置
>>> angles = limb.joint_angles()
# 打印当前的关节位置信息
>>> print angles

# 移动到 Sawyer 的最自然的位置
>>> limb.move_to_neutral()
# 获取最自然位置的关节点信息
>>> angles = limb.joint_angles()
# 打印关节点信息
>>> print angles

# 将关节点全部设置为 0.0 的状态
>>> angles['right_j0']=0.0
>>> angles['right_j1']=0.0
>>> angles['right_j2']=0.0
>>> angles['right_j3']=0.0
>>> angles['right_j4']=0.0
```

```
>>> angles['right_j5']=0.0
>>> angles['right_j6']=0.0
# 打印关节点信息
>>> print angles
# 使 Sawyer 的右臂移动到该关节点
>>> limb.move_to_joint_positions(angles)

# 使 Sawyer 跟大家摆手打招呼
# 保存第一个动作的关节点位置
>>> wave_1 = {'right_j6': -1.5126, 'right_j5': -0.3438, 'right_j4': 1.5126, 'right_j3': -1.3833, 'right_j2': 0.03726, 'right_j1': 0.3526, 'right_j0': -0.4259}
# 保存第二个动作的关节点位置
>>> wave_2 = {'right_j6': -1.5101, 'right_j5': -0.3806, 'right_j4': 1.5103, 'right_j3': -1.4038, 'right_j2': -0.2609, 'right_j1': 0.3940, 'right_j0': -0.4281}
# 连续运行这两个动作 3 次
>>> for _move in range(3):
...         limb.move_to_joint_positions(wave_1)
...         rospy.sleep(0.5)
...         limb.move_to_joint_positions(wave_2)
...         rospy.sleep(0.5)

# 退出 python 界面
>>> quit()
```

(a) 自然状态

(b)关节点都为 0.0 的状态

图 3-3 Sawyer 关节状态

从这里获取到的信息是：

要与 Sawyer 的手臂关节联系，首先要连接右臂 limb = intera_interface.Limb('right')；获取关节角度的接口是 angles = limb.joint_angles()；关节点的单独赋值方式是 angles['right_j0']=0.0；对所有关节点整体赋值的方式是 wave_1 = {'right_j6': -1.5126, 'right_j5': -0.3438, 'right_j4': 1.5126,

'right_j3': -1.3833, 'right_j2': 0.03726, 'right_j1': 0.3526, 'right_j0': -0.4259}；将 Sawyer 的右臂移动到指定的关节点的接口是 limb.move_to_joint_positions(wave_1)；Sawyer 自带的移动到自然位置的接口是 limb.move_to_neutral()；要使 Sawyer 连续地做某些动作，就设定多个动作并在一个循环里循环调用这些动作。

3.2 Sawyer 应用程序接口

Sawyer 应用程序接口，提供了用于和 Sawyer 机器人交互的 PythonAPI，它由一系列的封装了 ROS 通讯的类组成，可以直接通过此 PythonAPI 控制机器人不同的接口。

3.2.1 可用应用程序接口

模块层级：

intera_control
intera_control.pid
intera_dataflow
intera_dataflow.signals
intera_dataflow.wait_for'
intera_dataflow.weakrefset
intera_interface
intera_interface.camera
intera_interface.cfg
intera_interface.cfg.SawyerPositionFFJointTrajectoryActionServerConfig
intera_interface.cfg.SawyerPositionJointTrajectoryActionServerConfig
intera_interface.cfg.SawyerVelocityJointTrajectoryActionServerConfig
intera_interface.cuff
intera_interface.digital_io
intera_interface.gripper
intera_interface.head
intera_interface.head_display
intera_interface.lights
intera_interface.limb
intera_interface.navigator
intera_interface.robot_enable
intera_interface.robot_params
intera_interface.settings
intera_io
intera_io.io_command
intera_io.io_interface

详细函数和变量可参考：

https://rethinkrobotics.github.io/intera_sdk_docs/5.1.0/intera_interface/html/index.html .

3.2.2 机械臂接口

此接口针对机械臂操作，可通过下述方式初始化：

```
from intera_interface import Limb
right_arm = Limb('right')
```

该接口包含下述功能：
查询关节状态；
切换控制模式；
发送关节命令（位置、速度、力矩）。

3.2.3 末端执行器接口

此接口针对末端执行器操作：

```
from intera_interface import Gripper
right_gripper = Gripper('right')
```

主要功能如下：
向夹持器发送打开或闭合命令；
查询夹持器状态和属性；
夹持器插拔式时做出相应反应；
夹持器标定；
控制夹持器的各种动作（速度、移动力、抓取力、死区、真空阈值等）。

3.2.4 导航按钮接口

此接口对应于 Sawyer 的导航按钮的接口：

```
from intera_interface import Navigator
right_arm_navigator = Navigator('right')
right_torso_navigator = Navigator('torso_right')
```

主要功能如下：
查询转轮的状态；
对转轮和按钮动作做出相应反应；
控制导航按钮的灯。

3.2.5 相机接口

此接口对应于 Sawyer 的相机接口：

```
from intera_interface import Cameras
cameras = Cameras('head_camera')
```

主要功能如下：
打开或关闭相机；
切换相机：head_camera, right_hand_camera；
获得相机设置参数（帧速、曝光度、增益、白平衡等）。

3.2.6 头部接口

此接口用于控制 Sawyer 头部运动：

```
from intera_interface import Head
<component name> = Head()
```

可用命令：

on_head_state：对用于状态改变；
blocked：检查头部是否被阻挡；
pan, pan_mode, panning：状态值；
set_pan：设置摇头。

3.2.7 其他设置

SDK 包含机器人参数设置：
机械臂关节转角精度；
头部平动关节旋转精度；
SDK 版本；
夹持器版本和 SDK 版本兼容设置。

3.3 Sawyer SDK 示例

3.3.1 内置相机图像处理

1、命令操作：

使用命令

```
$ rosrun intera_examples camera_display.py
```

来启动该节点，另外还有 3 个参数：

◆ -c or --camera：这是选择摄像头的选项，默认是 head_camera，还可选 right_hand_camera；

◆ -r or --raw：使用原始图像或是经 opencv 标定矫正后的图像，默认是使用经 opencv 标定矫正过图像，如果使用-r 则选择原始图像；

◆ -e or --edge：是否使用 opencv 图像算子对图像进行灰度化和平滑处理，默认不处理，如果使用-e 则进行图像平滑处理。

比如用命令：

```
$ rosrun intera_examples camera_display.py
```

则表示启动头部摄像头，并使用经 opencv 矫正过的图像，不对图像进行平滑处理

使用：

```
$ rosrun intera_examples camera_display.py -c right_hand_camera -r -e
```

则表示使用手部摄像头，并使用有畸变的原图，对图像进行平滑处理。

2、代码详解：

大致过程是用代码段：

```
rp = intera_interface.RobotParams()
valid_cameras = rp.get_camera_names()
```

获取相机参数，采用代码段：

```
camera = intera_interface.Cameras()
if not camera.verify_camera_exists(args.camera):
    rospy.logerr("Invalid camera name, exiting the example.")
    return
```

```
camera.start_streaming(args.camera)
```

上述代码获得相机图像后，再通过回调 camera.set_callback(...)函数来对图像做处理等。Sawyer 没有设置图像大小的参数，默认的图像大小经测试为：臂端摄像头的图像尺寸为 752x480；头部摄像头获取的图像尺寸为 1280x800。

3.3.2 显示屏背景图处理

1、命令操作：

使用命令

```
$ rosrun intera_examples head_display_image.py -f <the path to image file to send>
```

来启动节点。

usage: head_display_image.py [-h] [-f FILE [FILE ...]] [-l] [-r RATE]

◆-f <图像路径>为必要参数外，还有两个参数可选；

◆-l or --loop : 这是播放次数的选项，默认是一次，如果选择-l 则会持续显示该图片；

◆-r or --rate : 图片显示的频率，默认是 1.0；

其中屏幕的尺寸为 1024x600，显示图片是以左上角为起点平铺。

2、代码详解：

大致显示图片的代码段为：

```
head_display = intera_interface.HeadDisplay()
head_display.display_image(args.file, args.loop, args.rate)
```

根据代码可以推测，如果想要多次在屏幕上显示图片，就可以修改 args.file 的文件目录，在一个循环里进行显示，如果想要在屏幕中间显示图片，就得对图像用 opencv 进行处理，做一幅 1024x600 的底片，将图片移到底片中间再进行显示。

3.3.3 夹爪操控

3.3.3.1 键盘操控夹爪

1、命令操作：

通过命令

```
$ rosrun intera_examples gripper_keyboard.py
```

来用键盘控制 Sawyer 的夹爪,节点有两个参数：

◆-h --提示帮助

◆-l --可选，默认的是 right

启动后输入?号，会出现键盘按键的对应操作：

key bindings:

Esc: Quit --退出节点

?: Help --帮助显示信息

c: calibrate --表示标定夹爪，在打开 Sawyer 后只需要标定一次，而如果输入一次 r 之后，就表示重启了手抓，需要输入 c 就会执行一次正常的标定。

q: close --关闭夹爪到最小的位置
h: decrease holding force --减轻夹取物品的力度
u: decrease position --缩小夹爪间的距离
j: increase holding force --增加夹取的力度
i: increase position --增加夹爪手指间的距离
o: open --打开夹爪
r: reboot --重启夹爪控制，之后如果没有通过 c 命令标定夹爪，执行其他操作都无用
+: set 100% velocity --设置夹爪最大的速率
-: set 30% velocity --设置%30 的夹取速度
s: stop --停止夹爪的动作，需要重新输入 r 然后再输入 c 之后才能再次控制夹爪。

2、代码详解：

先是获取控制电动手抓的接口：

```
gripper = intera_interface.Gripper(limb)
```

然后就是用各种键盘键控制手抓：

```
bindings = {
        # key: (function, args, description)
        'r': (gripper.reboot, [], "reboot"),
        'c': (gripper.calibrate, [], "calibrate"),
        'q': (gripper.close, [], "close"),
        'o': (gripper.open, [], "open"),
        '+': (gripper.set_velocity, [gripper.MAX_VELOCITY], "set 100% velocity"),
        '-': (gripper.set_velocity, [thirty_percent_velocity], "set 30% velocity"),
        's': (gripper.stop, [], "stop"),
        'h': (offset_holding, [-(gripper.MAX_FORCE / num_steps)], "decrease holding force"),
        'j': (offset_holding, [gripper.MAX_FORCE / num_steps], "increase holding force"),
        'u': (offset_position, [-(gripper.MAX_POSITION / num_steps)], "decrease position"),
        'i': (offset_position, [gripper.MAX_POSITION / num_steps], "increase position"),
    }
```

3.3.3.2 摇杆操控夹爪

1、命令操作：

通过命令

```
$ rosrun intera_examples gripper_joystick.py
```

来通过遥感控制 Sawyer 的夹爪，有 4 个参数：

- ◆-h --提示帮助
- ◆-l --可选，默认 right

必选参数：

- ◆-j --必选{xbox,logitech,ps3}，选择遥感类型

2、手柄和摇杆的驱动安装

1）在 ubuntu 中安装相应驱动

首先先安装几个测试软件，在软件中心搜索 joystick，把图 3-4 中三个软件都安装了：

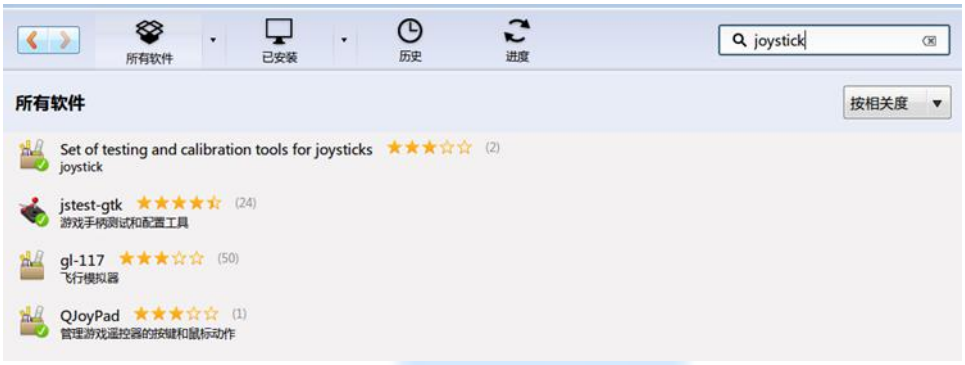


图 3-4 所需驱动图

接下来安装相关驱动，如果是使用 xbox360 专用手柄：

```
$ sudo add-apt-repository ppa:grumbel/ppa
$ sudo apt-get update
$ sudo apt-get install xboxdrv
```

如果是 ps3 手柄：

```
$ sudo apt-get install joy2key
```

以上驱动等安装完成，就可以测试手柄；最好的方式是将这些全部都安装好，这样无论是何种手柄都可以识别。

安装完驱动后，照正常步骤测试一下驱动。

先将手柄通过 USB 或者无线适配器连接到电脑，通过命令

```
$ ls -l /dev/input
```

查看手柄是否连接到电脑，默认情况下游戏手柄接口名称为 js0，如果看到有 js0 的端口，就说明连接成功。

再输入：

```
$ ls -l /dev/input/js0
```

输出为：

```
crw-rw-r--+ 1 root root 13, 0 10 月  5 09:03 /dev/input/js0
```

可知 js0 默认的所有者和群组都是 root，即需要 root 权限才能操作 js0。此处需要更改 js0 的 root 群组为 dialout 群组，通过命令

```
$ sudo chgrp dialout /dev/input/js0
```

查看更改后所属群组，输入：

```
$ ls -l /dev/input/js0
```

输出：

```
crw-rw-r--+ 1 root dialout 13, 0 10 月  5 09:03 /dev/input/js0
```

表明已经更改成功。

测试手柄信号，通过命令

```
$ sudo jstest /dev/input/js0
```

也可以看到手柄的所有信号端口的信息，操作手柄时，相应的信息就会变化。现在在软件中测试手柄，打开 jstest-gtk，动一动各个键位摇杆，是否都正常。

图 3-5　jstest-gtk 测试

操作手柄时，可以看到图 3-5 中相应的条码移动。

2）安装 ros 系统手柄相关的包和驱动

通过命令：

```
$ sudo apt-get install ros-indigo-joy
$ sudo apt-get install ros-indigo-joystick-drivers
$ rosdep install joy
$ rosmake joy
```

将这些驱动安装完成之后，就可以在 ros 中使用手柄了。运行 joy_node 节点：

```
$ rosrun joy joy_node
```

查看消息：

```
$ rostopic echo joy
```

现在按下手柄按键即可在信息中看到相应的变化。

如果这中间没出任何问题，就可以打开 baxter 的模拟器，并且使用手柄操作模拟器了。

使用命令：

```
$ roslaunch baxter_gazebo baxter_world.launch
```

开启 baxter 的模拟器，再使用命令：

```
$ roslaunch baxter_examples joint_position_joystick.launch joystick:=<joystick_type>
```

如：

```
$ roslaunch baxter_examples joint_position_joystick.launch joystick:=xbox
```

来启动控制 xbox 手柄的节点。两个节点都开启后，就可以控制 baxter 的模拟器了。

使用命令：

```
$ roslaunch  baxter_gazebo  baxter_world.launch
```

开启 baxter 的模拟器，再使用：

```
$ rosrun  joy  joy_node
```

开启手柄的按键和按钮，以便被调用，使用：

```
$ rosrun  baxter_examples  joint_position_joystick.py
```

来启动手柄与 baxter 的连接。开启这三个节点这后，便可用手柄控制 baxter 的模拟器了。

3、saitick 摇杆的驱动安装

通过命令安装驱动：

```
$ sudo  apt-get  install  git-core
$ sudo  apt-get  install  libx52pro0  libx52pro-dev
```

安装依赖包：

```
$ cd  ~/catkin_ws/src
$ git  clone https://github.com/Hacks4ros/h4r_x52_joyext.git
$ cd  ~/catkin_ws
$ catkin_make
```

安装完成之后，也可以测试摇杆是否能在 ubuntu 和 ros 中使用。

连接摇杆和电脑，测试摇杆信号，通过命令

```
$ sudo jstest /dev/input/js0
```

也可以看到摇杆的所有信号端口的信息，操作摇杆时，相应的信息就会变化。

运行 joy_node 节点：

```
$ rosrun joy joy_node
```

查看消息：

```
$ rostopic echo joy
```

会看到操作摇杆的按钮时，相应的 axes 和 buttons 信号会变化。

使用摇杆控制机器需要另外写代码，通过古风月的博客例程测试（ros 探索总结（九）——操作杆控制），摇杆在 ROS 上是可以控制机器的，如图。

摇杆介绍网址：http://hacks4ros.github.io/h4r_x52_joyext/doc/doxygen/index.html

图 3-6　摇杆图

3.3.3.3 腕部操控夹爪

1、命令操作：

通过命令

```
$ rosrun intera_examples gripper_cuff_control.py
```

来通过 Sawyer 电动夹爪的按钮来控制夹爪的开闭。

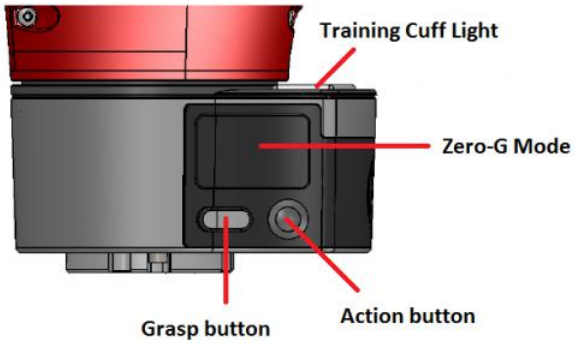


图 3-7 Sawyer 按钮示意图

节点有 4 个参数：

◆-h, --help --帮助提示

◆-g {['right']} --可选参数，默认为 right

◆-n, --no-lights --可选参数，输入-n 表示不开启与图 3-8 中 Zero-G 键相连的亮灯(图 3-8 中 Training Cuff Light)，即按住手臂的零重力按钮(图 3-8 中 Zero-G)时蓝灯或红灯不亮

◆-v, --verbose --可选参数，输入-v 则打印 debug 信息

启动后，按住长型的 Dash(Grasp button)键则关闭夹爪，长按住圆形的 Circle(Action button)按钮，则开启夹爪。

2、代码详解：

获取 Cuff 按钮、亮灯和夹爪的控制接口：

```
self._cuff = Cuff(limb)
self._lights = Lights()
self._gripper = Gripper(limb)
```

3.3.4 头部显示屏转动

1、命令操作：

通过节点

```
$ rosrun intera_examples head_wobbler.py
```

测试头部的运动。

2、代码详解：

代码部分通过 head = intera_interface.Head()来控制头部关节；通过 head.set_pan(angle)来设置头部关节的位置。

3.3.5 IK 服务

采用两种逆向运动学 IK 解算方式，分别是简单的 SolvePositionIK 和高阶的 SolvePositionIK-Request。

3.3.6 关节控制

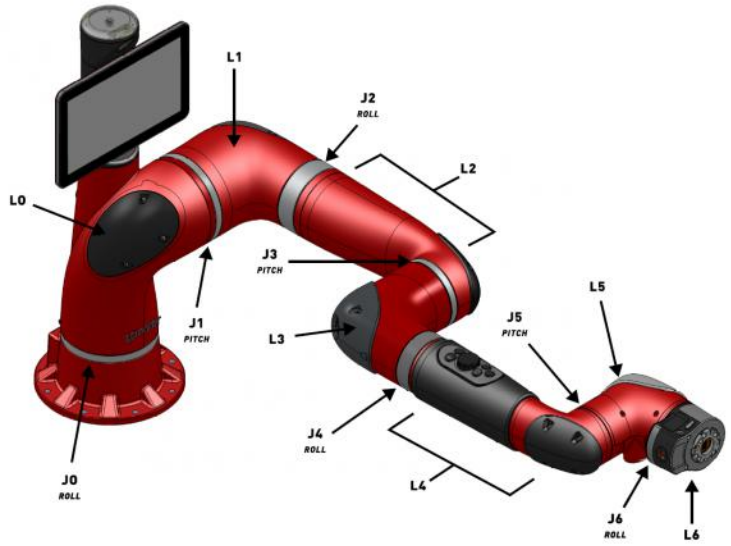


图 3-8 Sawyer 关节点示意图

Sawyer 总共有 7 个关节，编号为 J0, J1, J2, J3, J4, J5, J6，其中 J1, J3, J5 为 pitch 抓取关节点，J0, J2, J4, J6 为 roll 旋转关节点。

3.3.6.1 键盘操控关节运动

1、命令操作：

通过节点：

```
$ rosrun intera_examples joint_position_keyboard.py
```

通过键盘控制 Sawyer 关节点的运动，数字从 1 到 9，字母都是第一行从 Q 到 I，关闭进退都是数字字母上下对应的，1 和 Q 对应同一个关节，...，8 和 I 对应同一个关节，9 是单独表示夹爪的标定，数字增加，字母减少。

输入信息：

◆-h, --help --可选，帮助信息

◆-l {right}, --可选，选择手臂，Sawyer 默认右手臂

输入节点命令后，按下？号出现键盘控制提示：

key bindings:

Esc: Quit --退出按钮

?: Help --帮助按钮

9: right gripper calibrate --夹爪标定

8: right gripper close --夹爪关闭

i: right gripper open --夹爪开启

q: right_j0 decrease --底座关节 j0 顺时针转 350 度

1: right_j0 increase --底座关节 j0 逆时针转 350 度

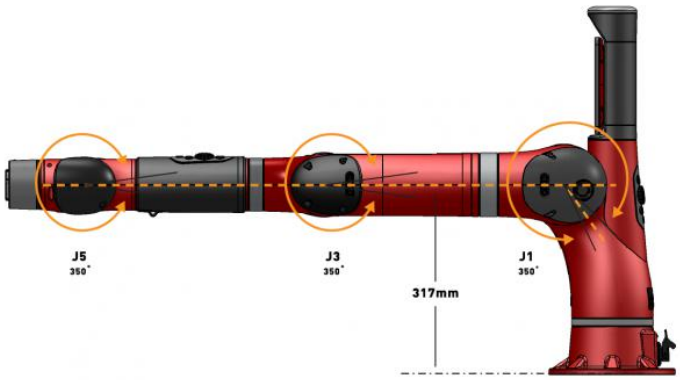
w: right_j1 decrease --肩关节 j1 顺时针转动 350 度

2: right_j1 increase --肩关节 j1 逆时针转动 350 度

e: right_j2 decrease --上手臂关节 j2 顺时针转动 350 度

3: right_j2 increase --上手臂关节 j2 逆时针转动 350 度

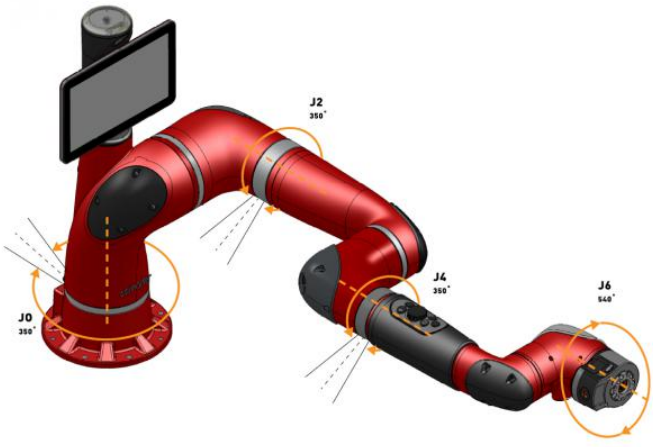
r: right_j3 decrease --肘关节 j3 顺时针转动 350 度
4: right_j3 increase --肘关节 j3 逆时针转动 350 度
t: right_j4 decrease --下手臂关节 j4 顺时针转动 350 度
5: right_j4 increase --下手臂关节 j4 逆时针转动 350 度
y: right_j5 decrease --环关节 j5 顺时针转动 350 度
6: right_j5 increase --环关节 j5 逆时针转动 350 度
u: right_j6 decrease --腕关节 j6 顺时针转动 540 度
7: right_j6 increase --腕关节 j6 逆时针转动 540 度



Pitch Joints

Joint Name	Range (Degrees)
J1	350
J3	350
J5	350

图 3-9 pitch 关节点及最大旋转角度



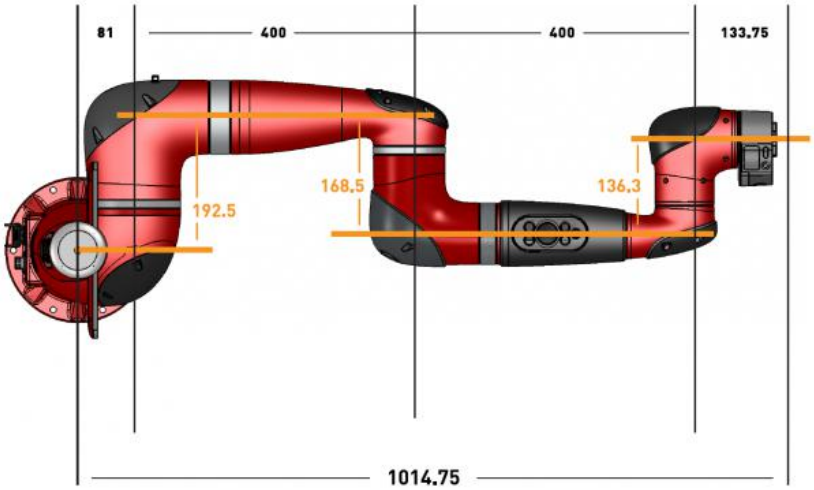
Roll Joints

Joint Name	Range (Degrees)
J0	350
J2	350
J4	350
J6	540

图 3-10 roll 关节点及最大旋转角度

其中关节分类为：Roll 旋转作用关节包含了 j0,j2,j4,j6 关节，Pitch 抓取作用关节包含 j1,j3,j5

关节。



Link Lengths

Link Name	Length
L0	81mm
L1	192.5mm
L2	400mm
L3	168.5mm
L4	400mm
L5	136.3mm
L6	133.75mm

图 3-11 手臂长度

2、代码详解：

代码部分需要控制 intera_**interface** 的两个类：

```
gripper = intera_interface.Gripper('right')
limb = intera_interface.Limb('right')
```

通过 gripper.open(), gripper.close(), gripper.calibrate()来控制夹爪的闭合。

通过 limb.joint_angle(joint_name), limb.set_joint_positions(joint_command)来控制关节的运动。

3.3.6.2 遥杆操控关节运动

1、命令操作：

首先确保已经安装了 joystick 软件和相应的手柄驱动工具，如果没有安装，请参照 3.3.3.2 节的相关内容进行安装，以便能正常驱动手柄。

(1)通过节点

```
$ rosrun intera_examples joint_position_joystick.py
```

来通过手柄来控制关节运动。节点有 4 个参数，输入信息：

◆-h, --help --可选，帮助信息

◆-l {right} --可选，选择手臂，Sawyer 默认右手臂

必选参数：

◆-j {xbox,logitech,ps3} --必选，支持 3 种遥感连接，xbox 手柄，logitech 摇杆，ps3 手柄。

(2)同时通过节点使用

```
$ rosrun joy joy_node
```

开启手柄的按键和按钮，以便被调用。通过以上两个节点就可以利用手柄和摇杆控制机器臂了。

(3)或者使用命令

```
$ roslaunch intera_examples joint_position_joystick.launch joystick:=<joystick_type>
```

来直接启动调用手柄控制机器臂的运动；例如用命令

```
$ roslaunch intera_examples joint_position_joystick.launch    joystick:=xbox
```

来启动控制 xbox 手柄的节点。

2、代码详解：

获取 Sawyer 的关节和夹爪接口：

```
limb = intera_interface.Limb(side)
gripper = intera_interface.Gripper(side)
```

获取手柄的控制接口：

```
jhi = lambda s: joystick.stick_value(s) > 0
jlo = lambda s: joystick.stick_value(s) < 0
bdn = joystick.button_down
bup = joystick.button_up
```

一一映射：

```
bindings = [
            ((jlo, ['leftStickHorz']), (set_j, [limb_cmd, limb, joints, 0, 0.1]),
              lambda: "Increase " + joints[0]),
```

3.3.6.3 控制关节到达路径点

1、命令操作

通过节点

```
$ rosrun intera_examples joint_position_waypoints.py
```

来记录动作以及播放已记录的动作。有 3 个参数，输入信息为：

- ◆ -h --可选，帮助信息
- ◆ -s --可选，速度信息，0.0 - 1.0,默认 0.3
- ◆ -a --可选，精度，默认 0.0

比如运行命令

```
$ rosrun intera_examples joint_position_waypoints.py -s 0.1 -a 0.1
```

表示以 0.1 的速率和 0.1 的精度运行。此时执行一个连续动作，并按下 Sawyer 手臂上的 OK 键，便会将这个连续动作记录；多次记录动作，会将前面已记录的动作覆盖，默认为记录一次动作；当终端上出现至少一个 Waypoint Recorded 时，按下 Rethink 按钮 Sawyer 便会循环执行上一次记录的动作，直至按下 ctrl+c 终止。

2、代码详解：

获取相关接口：

```
self._limb = intera_interface.Limb(self._arm)
self._navigator_io = intera_interface.Navigator(self._arm)
```

相应的控制函数：

```
ok_id = self._navigator.register_callback(self._record_waypoint, 'right_button_ok')
show_id = self._navigator.register_callback(self._stop_recording, 'right_button_show')
self._waypoints.append(self._limb.joint_angles())
self._limb.move_to_joint_positions(waypoint, timeout=20.0, threshold=self._accuracy)
```

3.3.6.4 控制关节的力矩弹簧

1、命令操作：

通过节点：

```
$ rosrun intera_examples joint_torque_springs.py
```

实现力矩的控制，运行完命令之后，Sawyer 会先运行到 neutral 姿态，进入力矩的空档，此时 Sawyer 是处于零重力状态的，可以用手去推动，但是会有 Sawyer 会带有一点回力。

输入参数有 2 个：

- ◆-h, --help --可选，帮助信息
- ◆-l {right} --可选，选择手臂，Sawyer 默认右手臂

2、代码详解：

获取相关接口：

```
self._limb = intera_interface.Limb(limb)
self._rs = intera_interface.RobotEnable(CHECK_VERSION)
self._rs.enable()
```

相应的控制代码段：

```
#     create our command dict
cmd = dict()
#     record current angles/velocities
cur_pos = self._limb.joint_angles()
```

```
cur_vel = self._limb.joint_velocities()

#     calculate current forces
for joint in self._start_angles.keys():
   #     spring portion
   cmd[joint] = self._springs[joint] * (self._start_angles[joint] -cur_pos[joint])
   #     damping portion
   cmd[joint] -= self._damping[joint] * cur_vel[joint]
   #     command new joint torques
self._limb.set_joint_torques(cmd)
```

3.3.6.5 记录关节运动

1、命令操作：

通过节点

```
$ rosrun intera_examples joint_recorder.py
```

来记录一个动作，有 3 个参数：

- ◆-h, --help --帮助信息
- ◆-r RECORDRATE, --记录速递，缺省 100

必选参数:

- ◆-f FILENAME --必选参数，保存记录文件的路径

比如通过命令

```
$ rosrun intera_examples joint_recorder.py -f test_recorder
```

就会记录你控制 Sawyer 执行的连续动作并保存在 test_recorder 文件中，如果不指定路径，就默认保存在工作区间 catkin_ws 文件夹下，如果指定绝对路径，就保存在绝对路径下。如果文件已经在相同的路径下保存了一份，再次记录就会覆盖之前的内容。在记录动作的过程中，可以按下 dash 按钮打开夹爪，长按下 circle 按钮关闭夹爪。记录的文件内容格式为：

```
time,right_j0,right_j1,right_j2,right_j3,right_j4,right_j5,right_j6,right_gripper
0.2791,0.3556,0.54189,...
```

2、代码详解：

获取相关接口：

```
self._limb = intera_interface.Limb(limb)
self._gripper = intera_interface.Gripper(side)
self._cuff = intera_interface.Cuff(side)
```

相应的控制代码段：

```
if self._filename:
    joints_right = self._limb_right.joint_names()
```

```
    with open(self._filename, 'w') as f:
        f.write('time,')
    temp_str = '' if self._gripper else '\n'
    f.write(','.join([j for j in joints_right]) + ',' + temp_str)
    if self._gripper:
        f.write(self.gripper_name+'\n')
    while not self.done():
        if self._gripper:
            if self._cuff.upper_button():
                self._gripper.open()
            elif self._cuff.lower_button():
                self._gripper.close()
        angles_right = [self._limb_right.joint_angle(j) for j in joints_right]
        f.write("%f," % (self._time_stamp(),))
        f.write(','.join([str(x) for x in angles_right]) + ',' + temp_str)
        if self._gripper:
            f.write(str(self._gripper.get_position()) + '\n')
        self._rate.sleep()
```

3.3.6.6 重演关节运动轨迹

1、命令操作：

通过节点

```
$ rosrun intera_examples joint_position_file_playback.py
```

来执行已记录的关节运动，有 4 个参数：

- ◆-h --可选参数，帮助信息
- ◆-l --可选参数，选择手臂，默认为右臂
- ◆-n --可选参数，循环播放次数，默认为 1 次

必选参数：

- ◆-f --必选参数，已记录关节运动的路径文件名

比如使用命令

```
$ rosrun intera_examples joint_position_file_playback.py -f test_recorder -n 3
```

循环执行 3 次 test_recorder 文件里保存的运动轨迹。

2、代码详解：

获取相关接口：

```
self._limb = intera_interface.Limb(limb)
self._gripper = intera_interface.Gripper(side)
```

相应的控制代码段：

```
with open(filename, 'r') as f:
    lines = f.readlines()
keys = lines[0].rstrip().split(',')
...
limb_interface.move_to_joint_positions(cmd_start)
...
if len(limb_cmd):
      limb_interface.set_joint_positions(limb_cmd)
if has_gripper and gripper.name in cmd:
      gripper.set_position(cmd[gripper.name])
```

3.3.6.7 关节轨迹运行服务测试

首先通过节点：

```
$ rosrun intera_interface joint_trajectory_action_server.py
```

来启动关节控制器，有 4 个参数：

◆-h, --help --可选参数，帮助信息

◆-l {right} --可选参数，选择手臂，默认右臂

◆-r RATE, --可选参数，速率，默认 100

◆-m {position_w_id,position,velocity} --可选参数，轨迹执行的控制模式，默认 position_w_id

代码部分为：

```
if mode == 'velocity':
   config_name = ".join([robot_name,"VelocityJointTrajectoryAction-ServerConfig"])"
elif mode == 'position':
   config_name = ".join([robot_name,"PositionJointTrajectoryAction-ServerConfig"])"
else:
   config_name = ".join([robot_name,"PositionFFJointTrajectoryAction-ServerConfig"])"
```

然后通过节点

```
$ rosrun intera_examples joint_trajectory_file_playback.py
```

来执行已经记录的轨迹运动，有 4 个参数：

◆-h, --help --可选参数，帮助信息

◆-l {right} --可选参数，选择手臂，默认右臂

◆-n --loops --可选参数，循环播放次数，默认为 1 次

必选参数：

◆-f --file --必选参数，已记录关节运动的路径文件名

测试结果为：

如果通过命令

```
$ rosrun intera_interface joint_trajectory_action_server.py -m position_w_id
```

来启动服务器，在另一个终端通过命令

```
$ rosrun intera_examples joint_trajectory_file_playback.py -f test_recorder
```

来执行运动轨迹时，则提示一些错误信息，说某些关节点位置超出错误门限，比如：Exceeded Error Threshold on right_j4:0.20672507,多测试几次的错误信息，基本都是某个关节点超过 0.2 以上就是会出错，因此，position_w_id 模式下的运动轨迹应该时小幅度运动时的精细动作的轨迹模式，某些关节超出范围便会不执行。

如果使用命令

```
$ rosrun intera_interface joint_trajectory_action_server.py -m position
```

来启动服务器，在另一个终端通过命令

```
$ rosrun intera_examples joint_trajectory_file_playback.py -f test_recorder
```

来执行 test_recorder 文件保存的运动轨迹，在这种模式下，运动轨迹基本全部回放，和录制时的动作一样；

如果通过命令

```
$ rosrun intera_interface joint_trajectory_action_server.py -m velocity
```

来启动服务器， 在另外一个终端通过命令

```
$ rosrun intera_examples joint_trajectory_file_playback.py -f test_recorder
```

来执行运动轨迹时，只会将动作大致执行，并没有将关节运动记录点全部回放。

3.3.7 头部灯光闪烁测试

1、命令操作：

通过节点

```
$ rosrun intera_examples lights_blink.py
```

来测试头顶的亮灯情况，进行两次的绿灯亮和灭的显示，2 个输入参数：

- ◆-h, --help --可选，帮助信息
- ◆-l {right} --可选，选择手臂的头顶灯，Sawyer 默认 head_green_light

2、代码详解：

代码部分，通过

```
from intera_interface import Lights
```

导入 Lights 类，通过语句

```
l = Lights()
```

来连接头顶灯接口，通过语句：

```
l.set_light_state('head_green_light', True)
```

控制头顶灯开，通过语句：

```
l.set_light_state('head_green_light', Faulse)
```

控制灯灭。

3.3.8 MoveIt 测试

命令操作：

测试 Sawyer moveit 下载过程：

```
= Make sure to update your sources =
  $ sudo apt-get update
= Install MoveIt! =
$ sudo apt-get install catkin-indigo-moveit
$ cd ~/catkin_ws/
$ ./intera.sh
$ cd ~/catkin_ws/src
$ wstool merge https://raw.githubusercontent.com/Rethin kRobotics/ sawyer_moveit/master/
sawyer_moveit.catkininstall

$ wstool update
$ cd ~/catkin_ws/
$ catkin_make
```

再运行命令

```
$ rosrun intera_interface joint_trajectory_action_server.py
```

启动轨迹服务器，在另外一个终端启动

```
$ roslaunch sawyer_moveit_config sawyer_moveit.launch electric_gripper:=true
```

来运行 MoveIt 节点，此处 electric_gripper:=true 表示带电动夹爪，在 rviz 里会增加对对电动夹爪的控制，如果没有这一项，就表示没有带夹爪。

第四章 Sawyer 应用案例

本章介绍研究型 Sawyer 的几个基础应用案例，例如人脸识别与跟踪、模仿人手臂动作进行手臂跟随以及拖动示教演示。

4.1 人脸识别与跟踪

本例展示 Sawyer 可以识别人脸，并实时跟踪人脸的方位，进而使 Sawyer 头部跟随人脸运动。首先调用 Sawyer 的头部摄像头获取图像，对图像进行分析，通过算法分析图像中是否含有人脸。

在你运行这个程序节点(node)前，请确认你的电脑已经正确与 Sawyer 建立了连接。然后请在终端中运行：

```
$ roslaunch face_tracker_sawyer face_follower.launch
```

当你运行上面所说的 launch 文件时，它按顺序启动了三个程序节点(node)。

第一个被启动的程序节点 (node) 打开了 Sawyer 头部位置的摄像头 (head_camera) 。当 Sawyer 没有发布头部摄像头视频图像消息到指定 ros topic 话题 "/io/internal_camera/head_camera/camera_info"上时，需要运行这个节点，之后可手动将其关闭。

第二个程序节点(node) 运行了 rbx1 （ROS By Example 书籍第一册）代码集中的 face_tracker2 。我们的代码需要该程序节点(node) 进行人脸识别的计算。这意味着你需要先将 rbx1 的代码集下载到你本地的工作空间(workspace) 中。

第三个程序节点 (node) 就是我们编写的用于跟踪脸部的，其中使用到了 PD 算法。

Face_follower.launch 代码内容：

```
<launch>

  <node name="camera_display" pkg="intera_examples" type="camera_display.py" args="-c
head_camera" respawn="false"/>

  <node pkg="rbx1_vision" name="face_tracker2" type="face_tracker2.py" output="screen">

    <remap from="input_rgb_image" to="/io/internal_camera/head_camera/image_raw" />
    <remap from="input_depth_image" to="/camera/depth/image_raw" />

    <rosparam>
      use_depth_for_tracking: False
      min_keypoints: 20
      abs_min_keypoints: 6
      add_keypoint_distance: 10
      std_err_xy: 2.5
      pct_err_z: 1.5
```

```
        max_mse: 10000
        add_keypoints_interval: 1
        drop_keypoints_interval: 1
        show_text: True
        show_features: True
        show_add_drop: False
        feature_size: 1
        expand_roi: 1.02
        gf_maxCorners: 200
        gf_qualityLevel: 0.02
        gf_minDistance: 7
        gf_blockSize: 10
        gf_useHarrisDetector: False
        gf_k: 0.04
        haar_scaleFactor: 1.3
        haar_minNeighbors: 3
        haar_minSize: 30
        haar_maxSize: 150
    </rosparam>

    <param name="cascade_1" value="$(find
rbx1_vision)/data/haar_detectors/haarcascade_frontalface_alt2.xml" />
    <param name="cascade_2" value="$(find
rbx1_vision)/data/haar_detectors/haarcascade_frontalface_alt.xml" />
    <param name="cascade_3" value="$(find
rbx1_vision)/data/haar_detectors/haarcascade_profileface.xml" />

  </node>

  <node pkg="face_tracker_sawyer" name="face_follower" type="face_follower.py" output="screen">

    <remap from="camera_info" to="/io/internal_camera/head_camera/camera_info" />

    <rosparam>
        rate: 10
        max_speed: 1
        min_speed: 0.1
        dist_gain: 0.045
        speed_Kp: 300
        speed_Kd: 50
        x_threshold: 0.01
    </rosparam>

  </node>
```

```
</launch>
```

Face Tracker 2 代码链接：

https://github.com/pirobot/rbx1/blob/indigo-devel/rbx1_vision/nodes/face_tracker2.py

Face Follower 代码解析：

```
#!/usr/bin/env python
import roslib
import rospy
from sensor_msgs.msg import RegionOfInterest, CameraInfo
from intera_interface import head

class FaceFollower():
    def __init__(self):
        rospy.init_node("face_follower_sawyer")

        # Set the shutdown function
        rospy.on_shutdown(self.shutdown)

        # The tate of updating the robot's motion
        self.rate = rospy.get_param("~rate", 10)
        r = rospy.Rate(self.rate)

        # The maximum rotation speed (speed range 0-100)
        self.max_speed = rospy.get_param("~max_speed", 1)

        # The minimum rotation speed (speed range 0-100)
        self.min_speed = rospy.get_param("~min_speed", 0.05)

        # Use PD Algorithm to control the robot
        self.dist_gain = rospy.get_param("~dist_gain", 2.0)
        self.speed_Kp = rospy.get_param("~speed_Kp", 2.0)
        self.speed_Kd = rospy.get_param("~speed_Kd", 0)

        # Only do some action if the displacement exceeds the threshold
        # This can reduce shaking (in percentage format)
        self.x_threshold = rospy.get_param("~x_threshold", 0.01)

        # Intialize the head movement command
        self.head_ = head.Head()
        self.pan_distance_ = 0
```

```
        self.pan_speed_ = 0.1
        self.pre_pan_speed_ = 0
        rospy.loginfo("Turning the head to the exactly middle...")
        self.head_.set_pan(0, 0.1, 10.0)

        # Get the image width and height from the camera_info topic
        self.image_width = 0
        self.image_height = 0

        # Set flag to indicate when the ROI stops updating
        self.target_visible = False

        # Wait for the camera_info topic to become available
        rospy.loginfo("Waiting for camera_info topic...")
        rospy.wait_for_message("camera_info", CameraInfo)

        # Subscribe the camera_info topic to get the image width and height
        rospy.Subscriber("camera_info", CameraInfo, self.get_camera_info)

        # Wait until we actually have the camera data
        while self.image_width == 0 or self.image_height == 0:
            rospy.sleep(1)

        # Subscribe to the ROI topic and set the callback to update the robot's motion
        rospy.Subscriber('roi', RegionOfInterest, self.PD_cal)

        # Wait until ROI is detected
        rospy.wait_for_message('roi', RegionOfInterest)
        rospy.loginfo("ROI messages detected. Starting follower...")

        # Begin the following loop
        while not rospy.is_shutdown():
            # If the target is not visible, stop the robot
            if not self.target_visible:
                rospy.loginfo("No visible target!")
            else:
                # Reset the flag to False by default
                self.target_visible = False
                # Output data for debug
                rospy.loginfo("Following the targrt! Moving head to location %f in speed %d",
self.pan_distance_, self.pan_speed_)

            self.head_.set_pan(self.pan_distance_, self.pan_speed_, 0.1)
```

```
            # Sleep for 1/self.rate seconds
            r.sleep()

    def PD_cal(self, msg):
        # If the ROI has a width or height of 0, we have lost the target
        if msg.width == 0 or msg.height == 0:
            return

        # If the ROI stops updating this, the robot will stay the same
        self.target_visible = True

        # Compute the displacement of the ROI from the center of the image
        target_offset_x = msg.x_offset + msg.width / 2 - self.image_width / 2

        try:
            percent_offset_x = float(target_offset_x) / (float(self.image_width) / 2.0)
        except:
            percent_offset_x = 0

        # Rotate the head only if the displacement of the target exceeds the threshold
        if abs(percent_offset_x) > self.x_threshold:
            # Calculate the pan distance by using P gain only
            self.pan_distance_ -= self.dist_gain * percent_offset_x
            # Calculate the pan speed by using PD Algorithm
            try:
                speed = self.speed_Kp * abs(percent_offset_x) - self.speed_Kd * (self.pan_speed_ -
self.pre_pan_speed_)
                if speed < 0:
                    direction = -1
                else:
                    direction = 1
                # Constrain the speed in range [min_speed, max_speed]
                self.pan_speed_ = abs(direction * max(self.min_speed, min(self.max_speed,
abs(speed))))
                # Store the speed as a previous one for next calculation
                self.pre_pan_speed_ = self.pan_speed_
            except:
                self.pan_speed_ = self.min_speed
                self.pre_pan_speed_ = self.min_speed
            #rospy.loginfo("percent_offset_x=%f,speed=%f,pre_pan_speed_=%d", percent_offset_x,
speed, self.pre_pan_speed_)
        else:
            # Otherwise stop the robot slowly. Here cannot set pan_distance_ to 0!
            self.pan_speed_ = 0.1
```

```
    def get_camera_info(self, msg):
        self.image_width = msg.width
        self.image_height = msg.height

    def shutdown(self):
        rospy.loginfo("Stopping the robot...")
        rospy.sleep(1)

if __name__ == '__main__':
    try:
        FaceFollower()
        rospy.spin()
    except rospy.rosInterruptException:
        rospy.loginfo("Face following node terminated.")
```

4.2 手臂跟随

本例展示的是 Sawyer 模仿人手臂的动作。首先通过 Kinect 采集到深度图像，然后使用 openni_tracker 去获取人体双臂的 tf 坐标变换，并将坐标变换转变成角度值，控制 Sawyer 机器人相应关节运动到指定角度，由于 Sawyer 关节设计与 Baxter 不同，而且只有单臂（右臂），并没有明显类人的肩关节和肘关节等，因此 Sawyer 跟随动作可能不太精确。本例仅控制 Sawyer 前四个关节，即 J0、J1、J2 和 J3。

Lauch 文件：

```
<launch>
  <!-- Start Sawyer Arm Tracking Based on Kinect -->

  <node pkg="openni_tracker" name="openni_tracker" type="openni_tracker" output="screen">
      <param name="fixed_frame" value="openni_depth_frame" />
  </node>

  <!-- Fire up teleoperate_tweaked-->
  <node name="teleoperate_tweaked" pkg="sawyer_teleoperation" type="teleoperate_sawyer.py"
output="screen"/>

  <!-- Fire up rviz with skeleton_frames-->
  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find rbx1_vision)/skeleton_frames.rviz"/>

</launch>
```

Sawyer 跟随代码：

```
#!/usr/bin/env python
```

```
import argparse
import sys
import rospy
import tf2_ros
import math
import numpy as np

import intera_interface
from intera_interface import CHECK_VERSION

BASE_FRAME = 'openni_depth_frame'
FRAMES = [
            'torso',
            'left_shoulder',
            'left_elbow',
            'left_hand',
            'right_shoulder',
            'right_elbow',
            'right_hand',
            ]

TEST_JOINT_ANGLES = dict()

TEST_JOINT_ANGLES['left'] = dict()

TEST_JOINT_ANGLES['right'] = dict()

TEST_JOINT_ANGLES['left']['left_s0'] = 0.0
TEST_JOINT_ANGLES['left']['left_s1'] = 0.0
TEST_JOINT_ANGLES['left']['left_e0'] = 0.0
TEST_JOINT_ANGLES['left']['left_e1'] = 0.0
TEST_JOINT_ANGLES['left']['left_w0'] = 0.0
TEST_JOINT_ANGLES['left']['left_w1'] = 0.0
TEST_JOINT_ANGLES['left']['left_w2'] = 0.0

TEST_JOINT_ANGLES['right']['right_s0'] = 0.0
TEST_JOINT_ANGLES['right']['right_s1'] = 0.0
TEST_JOINT_ANGLES['right']['right_e0'] = 0.0
TEST_JOINT_ANGLES['right']['right_e1'] = math.pi
TEST_JOINT_ANGLES['right']['right_w0'] = 0.0
TEST_JOINT_ANGLES['right']['right_w1'] = 0.0
TEST_JOINT_ANGLES['right']['right_w2'] = 0.0

#根据 openni_tracker 骨架识别发布的 TF 树坐标变换求解 Sawyer J0 至 J4 关节夹角
```

```
def get_joint_angles(user, tfBuffer, test, mirrored):
    """
    @param line: the line described in a list to process
    @param names: joint name keys
    """
    joint_angles = dict()
    joint_angles['left'] = dict()
    joint_angles['right'] = dict()

    frames = get_frame_positions(user, tfBuffer)
    if frames is None and not test:
        # if there's a problem with tracking, don't move
        return None

    if test:
        # use the hardcoded angles for debugging
        joint_angles = TEST_JOINT_ANGLES
    else:
        # do the math to find joint angles!
        reh = frames['right_hand'] - frames['right_elbow']
        res = frames['right_shoulder'] - frames['right_elbow']
        leh = frames['left_hand'] - frames['left_elbow']
        les = frames['left_shoulder'] - frames['left_elbow']
        rse = np.negative(res)
        lse = np.negative(les)

        # find down vector and normals
        nt = np.cross(frames['right_shoulder'] - frames['torso'], frames['left_shoulder'] - frames['torso'])
        d = np.cross(nt, frames['right_shoulder'] - frames['left_shoulder'])
        rns = np.cross(d, rse)
        lns = np.cross(d, lse)
        lne = np.cross(leh, les)
        rne = np.cross(reh, res)

        # normalize vectors
        reh = reh / np.linalg.norm(reh)
        res = res / np.linalg.norm(res)
        leh = leh / np.linalg.norm(leh)
        les = les / np.linalg.norm(les)
        rse = rse / np.linalg.norm(rse)
        lse = lse / np.linalg.norm(lse)
        nt = nt / np.linalg.norm(nt)
        d = d / np.linalg.norm(d)
        rns = rns / np.linalg.norm(rns)
```

```
        lns = lns / np.linalg.norm(lns)
        lne = lne / np.linalg.norm(lne)
        rne = rne / np.linalg.norm(rne)

        # do the math to find joint angles 通过矩阵数学运算得出夹角，Sawyer 仅限右臂
        joint_angles['left']['left_j0'] = np.arccos(np.dot(nt,lns)) - math.pi/5.0 # was + 0.0
        joint_angles['left']['left_j1'] = np.arccos(np.dot(d, lse)) - math.pi/2.0
        joint_angles['left']['left_j2'] = np.arccos(np.dot(nt, lne))
        joint_angles['left']['left_j3'] = math.pi - np.arccos(np.dot(leh, les))
        joint_angles['left']['left_j4'] = 0.0
        joint_angles['left']['left_j5'] = 0.0
        joint_angles['left']['left_j6'] = 0.0
        joint_angles['right']['right_j0'] = np.arccos(np.dot(nt,rns)) - math.pi*7.0/8.0 # was - pi
        joint_angles['right']['right_j1'] = np.arccos(np.dot(d, rse)) - math.pi/2.0
        joint_angles['right']['right_j2'] = np.arccos(np.dot(nt, rne)) - math.pi
        joint_angles['right']['right_j3'] = math.pi - np.arccos(np.dot(reh, res))
        joint_angles['right']['right_j4'] = 0.0
        joint_angles['right']['right_j5'] = 0.0
        joint_angles['right']['right_j6'] = 0.0

    if mirrored:
        return joint_angles
    else:
        # this is the default option, where the teleoperator's
        # left/right arm corresponds to the robot's left/right arm
        unmirrored = dict()
        unmirrored['left'] = dict()
        unmirrored['right'] = dict()
        unmirrored['left']['left_j0'] = joint_angles['right']['right_j0']
        unmirrored['left']['left_j1'] = joint_angles['right']['right_j1']
        unmirrored['left']['left_j2'] = joint_angles['right']['right_j2']
        unmirrored['left']['left_j3'] = joint_angles['right']['right_j3']
        unmirrored['left']['left_j4'] = 0.0
        unmirrored['left']['left_j5'] = 0.0
        unmirrored['left']['left_j6'] = 0.0
        unmirrored['right']['right_j0'] = joint_angles['left']['left_j0']
        unmirrored['right']['right_j1'] = joint_angles['left']['left_j1']
        unmirrored['right']['right_j2'] = joint_angles['left']['left_j2']
        unmirrored['right']['right_j3'] = joint_angles['left']['left_j3']
        unmirrored['right']['right_j4'] = 0.0
        unmirrored['right']['right_j5'] = 0.0
        unmirrored['right']['right_j6'] = 0.0
        return unmirrored
```

```
# 查找坐标系转换 TF 树，得到各坐标系位置
def get_frame_positions(user, tfBuffer):
    frame_positions = dict()
    try:
        for frame in FRAMES:
            transformation= tfBuffer.lookup_transform(BASE_FRAME, "%s_%d" % (frame, user),
rospy.Time())
            translation = transformation.transform.translation
            pos = np.array([translation.x, translation.y, translation.z])
            frame_positions[frame] = pos
    except (tf2_ros.LookupException, tf2_ros.ConnectivityException, tf2_ros.ExtrapolationException) as
e:
        print "Problem with kinect tracking."
        print e
        return None
    return frame_positions

def teleoperate(rate, user,test, mirrored):
    """
    Teleoperates the robot based on tf2 frames.
    @param rate: rate at which to sample joint positions in ms
    """

    rate = rospy.Rate(rate)
    # TODO: make these attributes of a class
    tfBuffer = tf2_ros.Buffer()
    listener = tf2_ros.TransformListener(tfBuffer)

    #left = intera_interface.Limb('left')
    right = intera_interface.Limb('right')

    while not rospy.is_shutdown():

        rate.sleep()
        joint_angles = get_joint_angles(user, tfBuffer, test, mirrored)
        print joint_angles
        if joint_angles is not None:
            #left.set_joint_position_speed(.7)
            right.set_joint_position_speed(.5)
            #left.set_joint_positions(joint_angles['left'])
            right.set_joint_positions(joint_angles['right'])

            print "updated positions"
    print "rospy shutdown, exiting loop."
```

```
    return True

def main():
    """
    Note: This version of simply drives the joints towards the next position at
    each time stamp. Because it uses Position Control it will not attempt to
    adjust the movement speed to hit set points "on time".
    """
    arg_fmt = argparse.RawDescriptionHelpFormatter
    parser = argparse.ArgumentParser(formatter_class=arg_fmt,
                                     description=main.__doc__)
    parser.add_argument(
        '-r', '--rate', type=int, default=10,
        help='rate to sample the joint positions'
    )

    parser.add_argument(
        '-t', '--test', type=bool, default=False,
        help='use hardcoded test joint angles'
    )
    parser.add_argument(
        '-u', '--user', type=int, default=1,
        help='kinect user number to use'
    )

    parser.add_argument(
        '-m', '--mirrored', type=bool, default=False,
        help='mirror the teleoperators movements'
    )
    args = parser.parse_args(rospy.myargv()[1:])

    print("Initializing node... ")
    rospy.init_node("teleoperation")
    print("Getting robot state... ")
    rs = intera_interface.RobotEnable(CHECK_VERSION)
    init_state = rs.state().enabled

    def clean_shutdown():
        print("\nExiting...")
        if not init_state:
            print("Disabling robot...")
            rs.disable()
    rospy.on_shutdown(clean_shutdown)
```

```
    print("Enabling robot... ")
    rs.enable()

    teleoperate(args.rate,    args.user, args.test, args.mirrored)

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

4.3 拖动示教

本例展示的是利用研究型 Sawyer 进行人机交互，拖动 Sawyer 手臂进行任务示教，按下指定按钮记录任务中的路径点、抓取与放置状态，最后按下完成按钮结束示教过程，此时程序将示教的任务保存为文件，接着程序会执行刚刚示教的任务两次。下次重新启动程序时，可以选择运行上次任务（读取任务文件）或者新建拖动示教任务。

代码解释:

```
#!/usr/bin/python
# -*- coding:utf-8 -*-
import collections
from copy import deepcopy
import rospy
import tf
import cv2
import cv_bridge
import rospkg
import tf

from geometry_msgs.msg import (
    PoseStamped,
    Pose,
    Point,
    Quaternion,
)

from std_msgs.msg import Header

from sensor_msgs.msg import (
    Image,
    JointState,
)

import intera_interface
```

```
from intera_core_msgs.srv import (
    SolvePositionIK,
    SolvePositionIKRequest,
)

class RecordPlay(object):
    def __init__(self, limb):
        self._rp = rospkg.rosPack()
        self._images = (self._rp.get_path('sawyer_record_play') +
                        '/share/images')
        self._path = self._rp.get_path('sawyer_record_play') + '/config/'
        self._side = limb
        self._limb = intera_interface.Limb(limb)

        dash_io = intera_interface.DigitalIO(limb + '_upper_button')
        circle_io = intera_interface.DigitalIO(limb + '_lower_button')
        navi_ok_io=intera_interface.DigitalIO(limb+ '_button_ok')


        self.record_jp = dict()#记录路径点，关节轨迹中的一点，一组 7 关节夹角值
        self.gripper_status=dict()#记录路径点对应的夹爪状态

        self.finished=False
        #counter for how many locations recorded in run time
        self.locationsTotal=0
        self._gripper = intera_interface.Gripper(limb)

        self._gripper.calibrate()
        self._gripper.set_holding_force(100.0)
        print "Gripper calibrated!"

        ik_srv = "ExternalTools/" + limb + "/PositionKinematicsNode/IKService"
        self._iksvc = rospy.ServiceProxy(ik_srv, SolvePositionIK)
        self._ikreq = SolvePositionIKRequest()

        # 等待按钮信号事件，执行相应任务
        circle_io.state_changed.connect(self._recordLocation)#记录路径点
        dash_io.state_changed.connect(self._toggleGripper)#夹爪开合切换
        navi_ok_io.state_changed.connect(self._finishRecording)#结束任务


    #夹爪开合切换
    def _toggleGripper(self,value):
        if value:
```

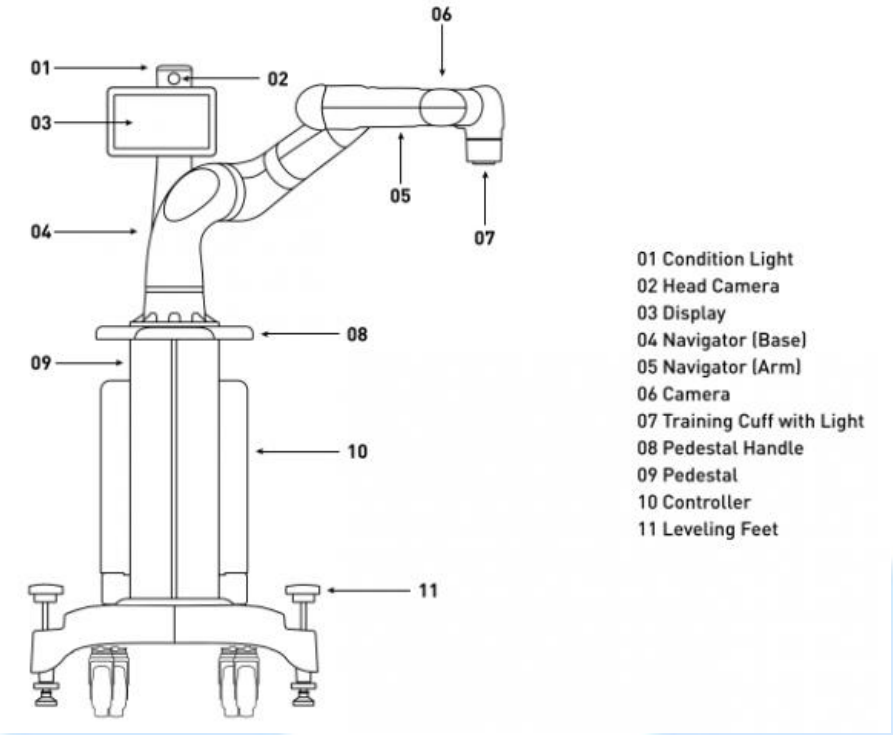
```
            if self._gripper.get_position()==0.0 or self._gripper.is_gripping()==True :
                self._gripper.open()
                print 'gripper opened'
            else:
                self._gripper.close()
                print 'gripper closed'

    #结束任务
    def _finishRecording(self,value):
        if value:
            self.finished=True
            filename = self._path + self._side + '_poses.config'
            self._save_file(filename)

    #记录路径点
    def _recordLocation(self,value):
        if value:
            self.locationsTotal +=1
            locationMsg = "Recording location " +str(self.locationsTotal )
            print locationMsg
            self.record_jp[self.locationsTotal] = self._limb.joint_angles()
            self.gripper_status[self.locationsTotal] = self._gripper.get_position()
            locationRecordedMsg = "Location " +str(self.locationsTotal ) + " recorded
successfullly!\n\n"
            print locationRecordedMsg
            print ("Move Sawyer's %s arm into a location you want\n"
                   "- press dash button to toggle gripper open and close\n"
                   "- press Circle button to confirm\n"
                   "- press OK button from arm's Navigation button group to end
Teaching/Recording\n"
                   % (self._side,))

    #读取任务文件
    def _read_file(self, file):
        with open(file, 'r') as f:
            for line in f:
                split = line.split('=')
                location = split[0]
                prefix = location.split("_")[0]
                position = split[1]
                if '_location' in location:
                    self.record_jp[int(prefix)] = eval(position)
                elif '_gripper' in location:
                    self.gripper_status[int(prefix)] = eval(position)
```

```
    #保存任务文件
    def _save_file(self, file):
        print "Saving your positions to file!"
        f = open(file, 'w')

        for prefix in range(1,self.locationsTotal+1):
            f.write(str(prefix) + '_location=' + str(self.record_jp[prefix]) + '\n')
            f.write(str(prefix) + '_gripper=' +
                    str(self.gripper_status[prefix]) + '\n')
        f.close()

    #终端交互提示与示教过程
    def get_locations(self):
        good_input = False
        while not good_input:
            self.read_file = raw_input("Would you like to use the previously "
                                       "found pick and place locations (y/n)?")
            if self.read_file != 'y' and self.read_file != 'n':
                print "You must answer 'y' or 'n'"
            elif self.read_file == 'y':
                filename = self._path + self._side + '_poses.config'
                self._read_file(filename)
                good_input = True
            else:
                print ("Move Sawyer's %s arm into a location you want\n"
                       "- press dash button to toggle gripper open and close\n"
                       "- press Circle button to confirm\n"
                       "- press OK button from arm's Navigation button group to end
Teaching/Recording\n"
                       % (self._side,))
                while not self.finished and not rospy.is_shutdown():
                    rospy.sleep(0.1)#多线程，等待结束任务信号按钮被触发
                    #waiting for all locations to be recorded
                print ("All location recorded!")

                good_input = True

    #移动至指定关节点，并根据条件打开与关闭夹爪
    def move_ToLocation(self,jp,gripperStatus):
        self._limb.set_joint_position_speed(0.2)
        self._limb.move_to_joint_positions(jp)
        self._gripper.set_position(gripperStatus)
```

```
    #运行/播放任务
    def play_procedure(self):
        for prefix in range(1,len(self.record_jp)+1):
            print "Location " + str(prefix-1) + " ----->> Location " + str(prefix) +"\n"
            self.move_ToLocation(self.record_jp[prefix],self.gripper_status[prefix])
            print "Arrived at Location "+ str(prefix) + " successfully!\n"

def main():
    rospy.init_node("rsdk_sawyer_record_play")

    rs = intera_interface.RobotEnable()
    print("Enabling robot... ")
    rs.enable()

    pp = RecordPlay('right')
    pp.get_locations()
    pp._gripper.open()

    #how many times do you want to play the recording procedure
    play_loops = 2
    play_count= 0
    while play_count<play_loops and not rospy.is_shutdown():
        pp.play_procedure()
        play_count+=1

    if rospy.is_shutdown():
        pp._gripper.open()

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

附录一 Sawyer 硬件参数

1）Sawyer 整体结构



图附-1 Sawyer 整体结构图

图附-1 中：

01—表示头顶亮灯，可以根据亮灯推测机器人的状态，Sawyer 开机时会头顶灯会显示为绿色，关机时头顶灯灭；

02—为头部摄像头，头部摄像头可观测周围环境，视野范围较广；

03—为屏幕显示，屏幕上会显示一些状态图片等；

04—为机体上的按钮区域；

05—为手臂上的按钮区域，按钮区域图片如图附-2 所示，包括 6 个功能键：Back button--返回按钮，Rethink button--主菜单按钮，Selector knob--选择按钮，Circle button--圆圈零重力按钮，Square button--方形按钮，X button--错误按钮；

06—表示手臂摄像头，用于观测放置于工作台上的物品，以便正确抓取；

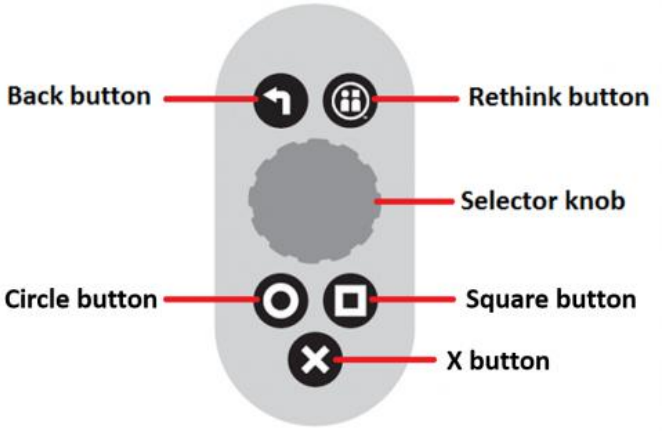
07—为带亮灯的 Cuff 零重力按钮；

08—为 Sawyer 手臂的基座螺丝安装柄；

09—为安装基座柱子；

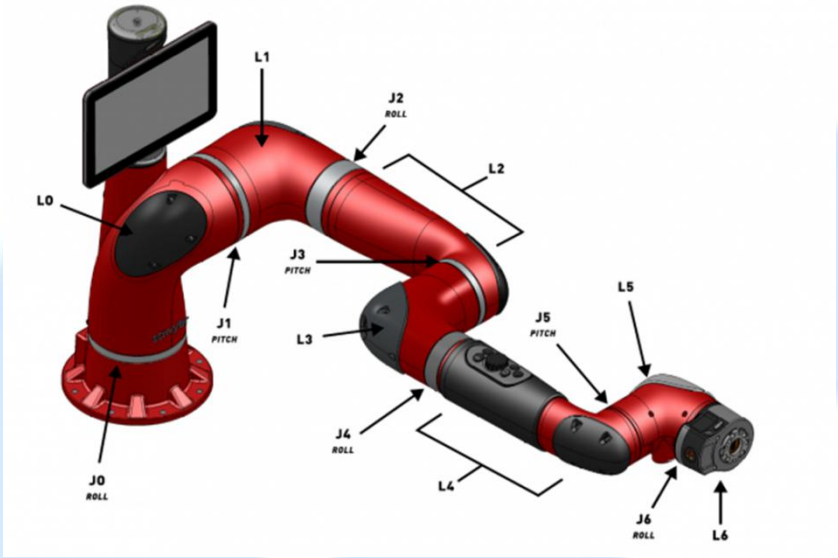
10—为控制器机箱；

11—为调节脚，用于固定；



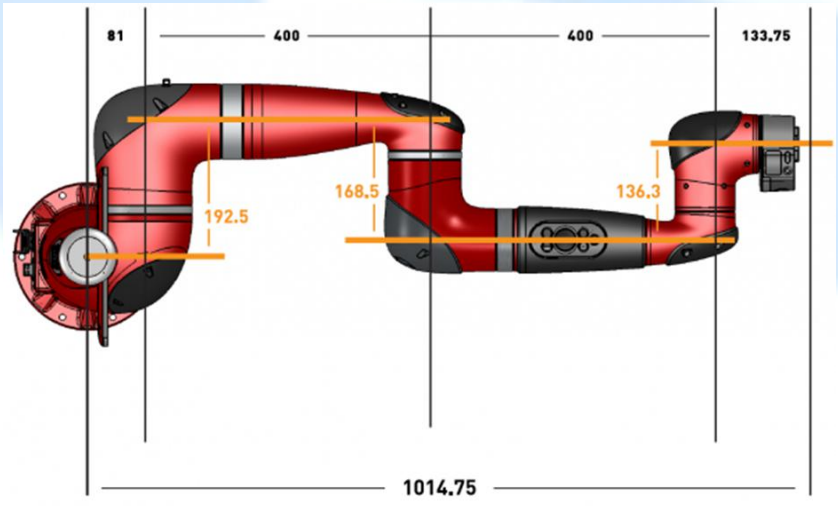
图附-2 Navigator 图片

2）Sawyer 关节与连杆

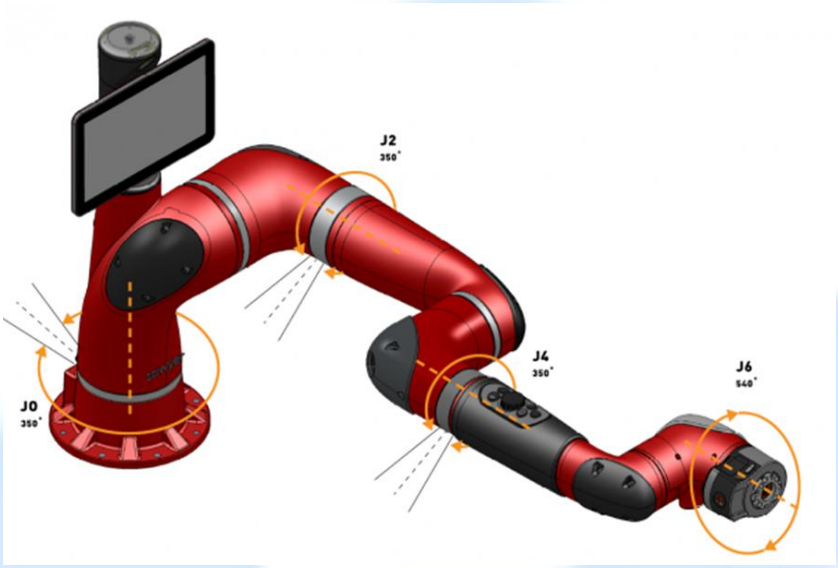


Sawyer 关节分类

Roll 回转	Pitch 俯仰
J0, J2, J4, J6	J1, J3, J5

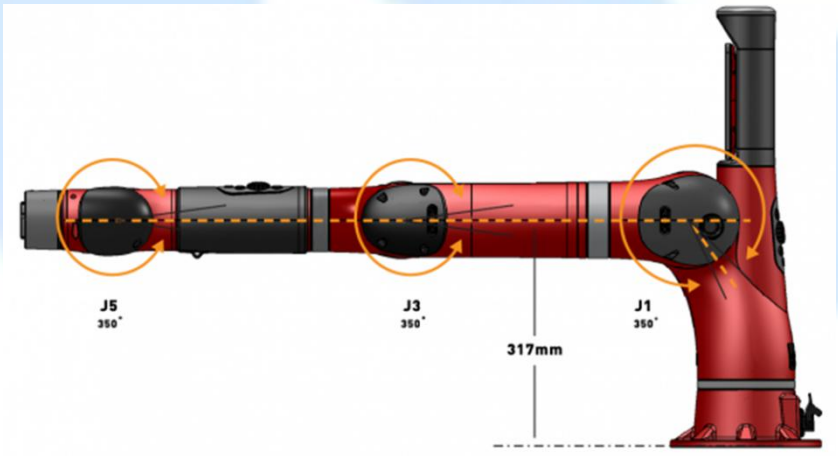


Sawyer 连杆长度	
L0	81mm
L1	192.5mm
L2	400mm
L3	168.5mm
L4	400mm
L5	136.3mm
L6	133.75mm
L0	81mm



回转关节

关节名称	转动范围(度)
J0	350
J2	350
J4	350
J6	540



俯仰关节

关节名称	转动范围（度）
J1	350

J3	350
J5	350

3）Sawyer 技术参数

基本参数	
本体自重	19Kg
臂展范围	1260mm
自由度	7
有效负载	4Kg
防护等级	IP54
重复定位精度	±0.1mm
性能参数	
固有安全性	功率和力度受限的柔性机械臂，带有串联弹性驱动器和内置传感器
嵌入式力觉传感器	每一个关节都有高分辨率的力度传感器
操作系统	自主研发的 Intera 软件平台（可升级）
电源	标准 220V 电源
嵌入式视觉	腕部 COGNEX 摄像头一个，顶部视觉摄像头一个
头部摄像头	
摄像头分辨率	1280×800
镜头类型	广角
色彩	RGB
腕部摄像头	
摄像头分辨率	752×480
镜头焦距	3.7mm
镜头类型	广角
色彩	Grayscale 灰度
其他特征	全局快门，同步内置闪光灯

湖南瑞森可机器人科技有限公司

官方网址：http://www.cothinkrobotics.com

联系方式：0731-89872400

企业邮箱：info@cothinkrobotics.com

地　　址：湖南省长沙市高新区桐梓坡 348 号国家工程中心