机械臂 ROS 使用部分

版本号:1.0

作者:吴雪铭

联系方式:微信 K15837821186,QQ 1362385699

1 ROS 基础介绍

ROS 机器人系统主要有消息、服务、动作三种通讯方式，此外还有一个特殊的 tf 机器人姿态树传输整个网络机器人的位置姿态信息 ,同时按照机器人位置运动学的原理进行位姿数据的共享以及计算。

1.1　消息方式

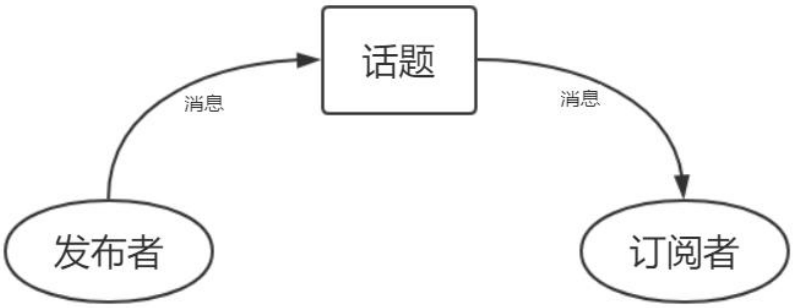


图 3.5 消息

如图 3.5 所示 ,消息通讯是一种类似于生产消费者模型(生产者-产品-消费者)的通讯方式，话题是一个 ROS 系统，在全局网络中都能够访问到，专门存放特定消息数据结构的、有一定长度的队列缓存。这个缓存地址以及命名，又称话题名。节点管理器负责提供话题名和网络程序端口地址的映射解析功能。一般来说，发布者定时向话题中放入最新的数据，而订阅者往往通过回调函数检查是否有新的消息数据到达。一般来说，传感器的驱动程序往往采用这种方式定时向整个 ROS 系统提供可用的数据，如超声波、里程计、图像、点云、惯性导航传感器等等。例如，移动机器人的底盘控制的驱动往往是作为一个订阅者，接收 ROS 系统中其他节点对它的控制消息，并将控制消息通过种种方法转化成对直流电机转动的控制。话题适合于有持续性的、有时间规律的场景。

1.2 服务方式

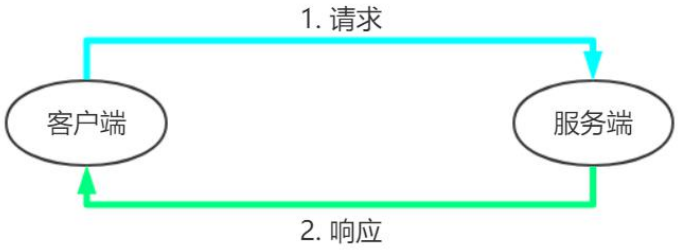


图 3.6 服务

如图 3.6 所示，服务通讯是一种类似于于传统网络应用通讯 CS(客户端-服务端)模式，不过通讯的数据结构分请求与响应两部分，它们是互相独立的数据结构类型。客户端发送请求结构的数据到服务端，ROS 服务端在接收到 ROS 客户端的请求数据后，进行数据的处理，往往最后给客户端一个数据响应，至此服务结束。服务与话题类似，服务端的名字和网络端口，也由节点管理器中提供映射解析功能。和一些长时间持续运行的话题不同，通常情况下 ROS 服务通讯处理的是临时的、短暂的、单次的任务，ROS 中服务器和客户端并不总是保持连接状态，因此这种通讯方式具备使用时间方面的灵活性，提高了服务的效率。

1.3　动作方式。

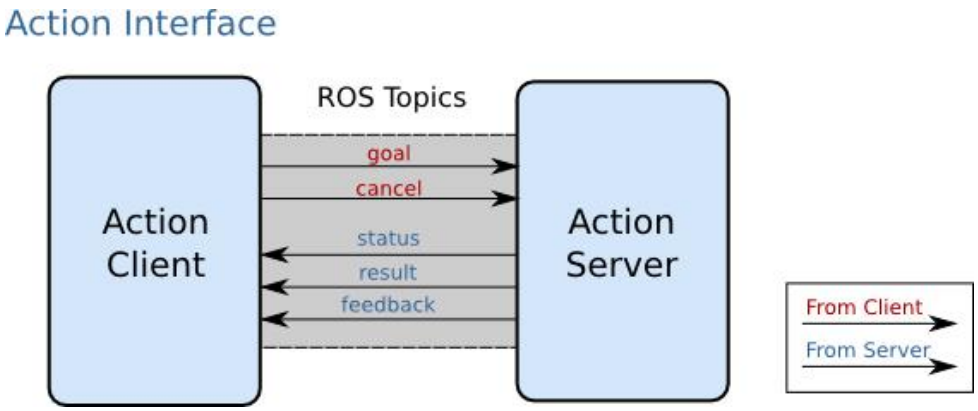


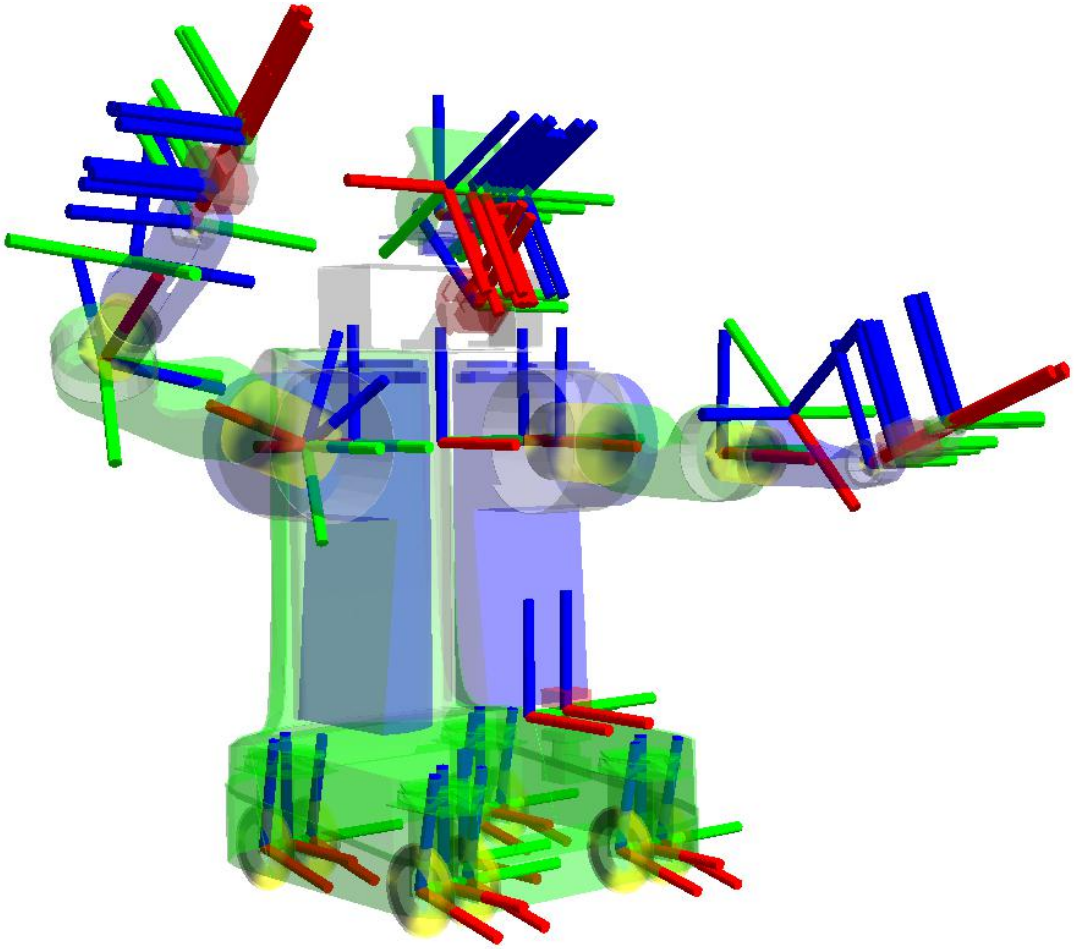
图 3.7 动作

如图 3.7 所示，ROS 动作通讯方式比较复杂。从逻辑层面看，ROS 动作编程类似于服务，它拥有动作客户端以及动作服务器，但是从实现方式上说，ROS 动作编程却是基于一组消息通讯实现的。动作通讯编程设计之初，就是为了一些需要在执行过程中时间较长、执行过程中也需要产生反馈的场景进行应用。比如，自主导航、以及机器人运动控制等应用。本文描述的毕业设计，就是使用 Action 动作编程进行实现的。

动作的数据结构里面，包含了 3 个子数据结构分别为：goal、result、feedback，这三个数据结构对应动作通讯组下的 3 个话题，而余下的 2 个话题 cancel 和 status 则是用于告诉服务端，取消对 goal 的执行和反馈服务端当前的状态，其中这个状态又类似于安卓开发中程序的生命周期状态，由于过于复杂这里不再解释，详情可见官方文档 wiki.ROS.org/actionlib/DetailedDescription。动作机制是由客户端向服务端发送一个 goal 数据结构，服务端进行数据处理，处理过程中向客户端反馈 feedback 数据结构，当数据处理完毕后，服务端会向客户端反馈 result 数据结构，来告诉客户端要求的 goal 最终执行的情况。status 数据结构包括成功、失败、占用、不可达、取消等 10 余种数据选项。cancel 数据结构比较简单，它一旦发布数据，则告诉服务端，取消当前 goal 的执行，并提前结束动作通讯。

1.4　机器人姿态树。

机器人姿态树程序存在的方式比较特别，通常由专门的姿态维护组件进行维护，为机器人提供符合机器人位置运动学原理的位姿数据。机器人不同的部分就像一片片树叶一样，由树枝将它们在空间位置和逻辑关系中彼此联系起来。用户既可以往这棵树上挂自己东西，也可以通过这个工具或者不同部分进行相关坐标的转换，比如笔者曾参与过的消防机器人项目中，就是通过机器人姿态树，将摄像头坐标系下火焰的空间位置，转化成水枪坐标系下火焰的空间位置，从而实现灭火的目的。



2　机械臂简介

2.1　机械臂产业链介绍

传统工业机械臂主要由电机、减速器、控制器以及传感器等组成，主要进行生产自动化。其中世界公认的机器人四大家族是安川、发那科、ABB、库卡。此外，这些年还出现了一些新兴的机器人品牌如优傲、傲波、新松等等。近些年伴随着人工智能技术的发展，许多机器人还用于物流、救援、医疗、服务等领域，从而牵扯到众多上下游许多不同产业链的发展，其中包括机电技术、机械工程技术、微电子技术、人工智能技术以及传感器制造技术等等。从这个层面上来讲，机器人技术的发展将会各行各业都产生巨大的影响。

图 1.1 通用机器人 6 轴机械臂 ur10e

图 1.2 安川 7 轴机械臂 SIA5D

2.2 机械臂关节运动原理

从机械臂的关节类型上划分主要分为旋转、伸缩和滑动三种关节，其中最为常用的则是旋转关节。能够执行旋转动作的电器则有很多，如仿生学模仿人类肌肉牵动的多种牵引结构，还有传统使用的各类电机。其中牵引结构中又主要包括使用铁丝、气压、液压作为动力来源，机械臂的各类电机又主要包括步进电机、伺服电机、舵机、无刷电机、以及近些年来新兴机器人关节模组中的无框电机等等。它们的运动原理都有所类似，这里以常见的桌面机器人上的步进电机为例，来讲解其中的运动原理。

常规的步进电机有两相和四相步进电机，由于本设计采用两相步进电机，两相步进电机的每一相分正负两极，所以常见二项四线步进电机，如图 3.1.1 所示，有绿、黑、红、蓝四根线分别对应 1a、2a、1b、2b，此处按照此类型的步进电机进行介绍。

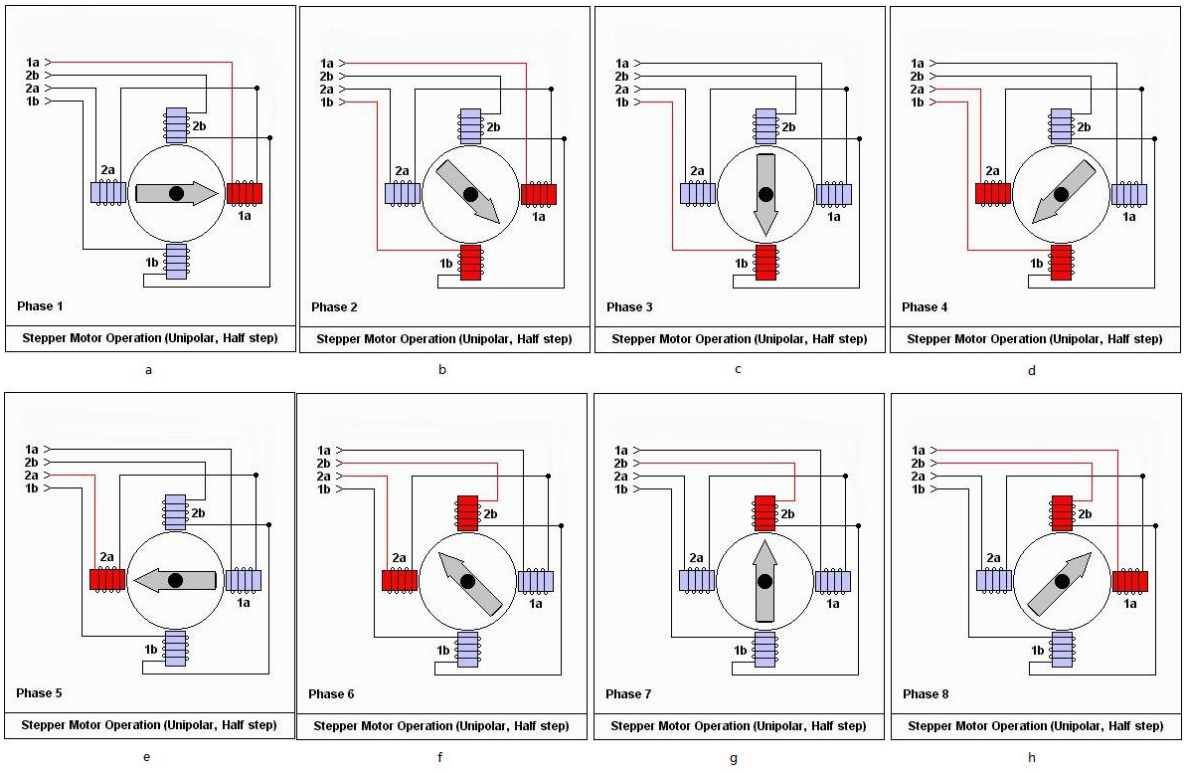


图 3.2 步进电机工作过程

如图 3.2 中 a 到 h 所示，红色线代表高电位，蓝色线代表低电位，中间的箭头代表了转

子，是拥有磁性的。电流从 1a 流入，则电流必定经过 2a 流出，1b 与 2b 同理。因此，通电时，同一时间同一相的两极生成的磁性必然是相反的，而在没有通电时，这两极都没有磁性。在四根线依次通电的组合顺序下，步进电机就会按照一个方向进行旋转，如果按照这个顺序的倒序输入电流时，步进电机将会按照相反的方向进行旋转，如果当这个组合停留在某个状态时，那么步进电机的转子就会稳定在一个位置，从而在负载不超过其最大力矩的情况下，转子的位置不会发生改变，即机械臂会处于一个静止状态，因此步进电机的准确性、稳定性非常适合作为轻量级桌面机械臂的关节设备。

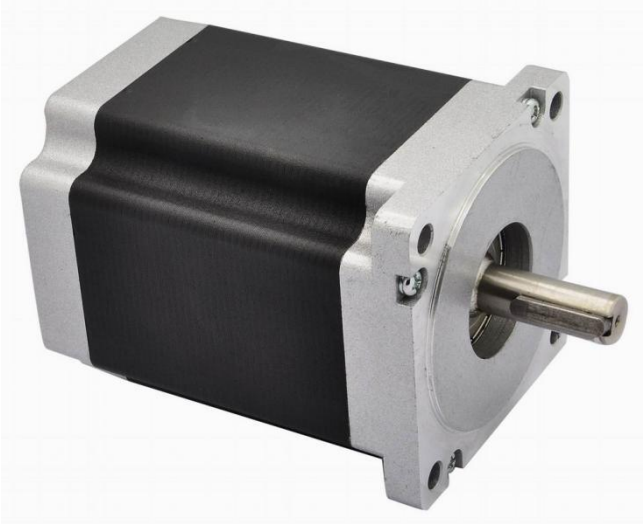


图 3.2 步进电机

2.3 机械臂驱动器使用方法

通常机械臂都采用微控制器作为控制中枢，而具体的控制工作则交由专用于电机驱动的模块，即电机驱动器。这些驱动器彼此独立，各自控制不同的电机。

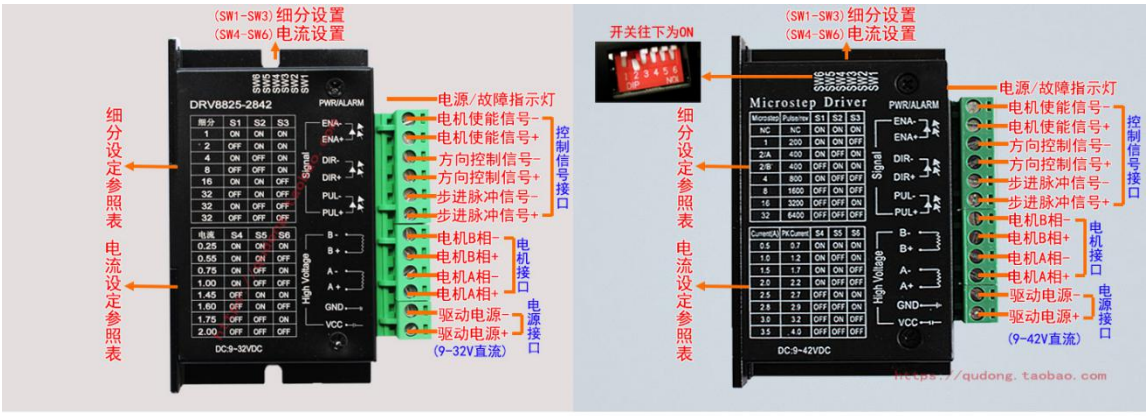


图 3.3 步进电机驱动器

驱动器和步进电机的连接方式即 A+、A-、B+、B-分别与绿、黑、红、蓝四根线一一对应。驱动装置可以分配给 4 根线上不同的电流，从而控制 4 个电磁铁的磁场强度，最后产生合力朝向一个特定的位置，这些电流的控制越精确，步进电机的运动精度就越高，电流的控制可以通过细分设置开关进行调整。需要注意的是，细分精度越高则步进电机的最大速度就会越低。驱动器会根据控制信号来确定当前四根线的电流分配状况，从而调整机械臂关节到相应的角度。这就实现了信息技术与电器技术之间的转换。

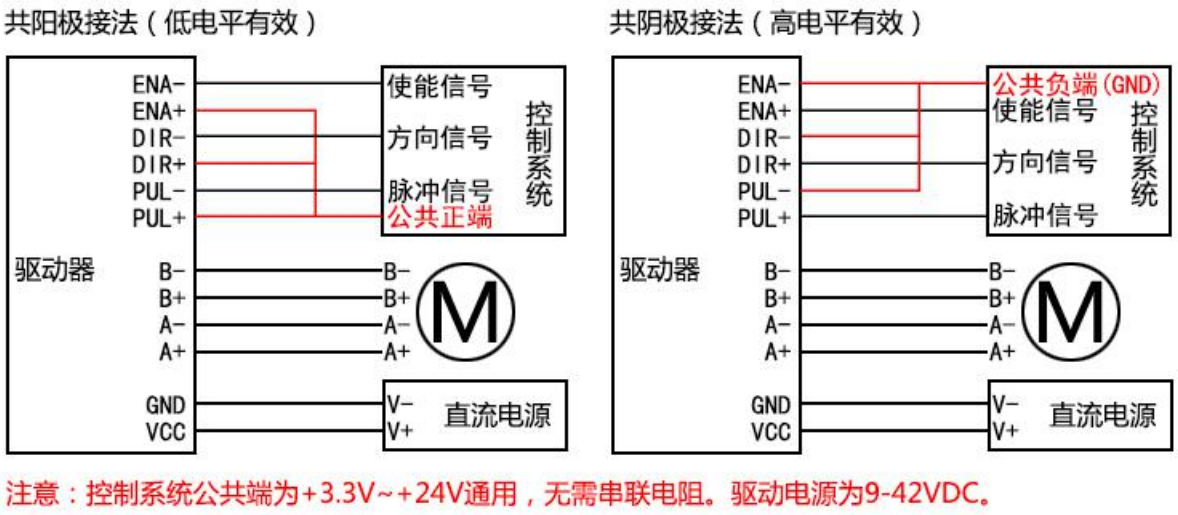


图 3.4 TB6600 接线图

控制器和驱动器的交互，属于信号技术对电器设备的控制。如图 3.4 中控制信号接口所示，主要使用使能、方向和脉冲三种类型的信号，其中正负端是二极管单向导通的，可以为用户提供低电压触发或者高电压触发的选项。

这里以共阴极接法为例，当给使能端高电平时，驱动器会给电机通上电流，步进电机转子将会固定于当前位置。反之，低电平时，步进电机将会处于失力状态。方向信号决定了步进电机接下来是进行顺时针或者逆时针运动。脉冲信号，决定了步进电机的转角步数，当步进电机处于通电状态并且方向信号不改变时，步进电机将会按照脉冲信号的数量，来进行单步运动。需要注意，脉冲信号能接受的频率都有一个范围，如果超过这个频率的最大数值，步进电机将会出现失步现象。通常为了方便控制，所有驱动器公用一个使能端引脚，来决定机械臂是否处于失力状态。

2.4 机械臂部署设计

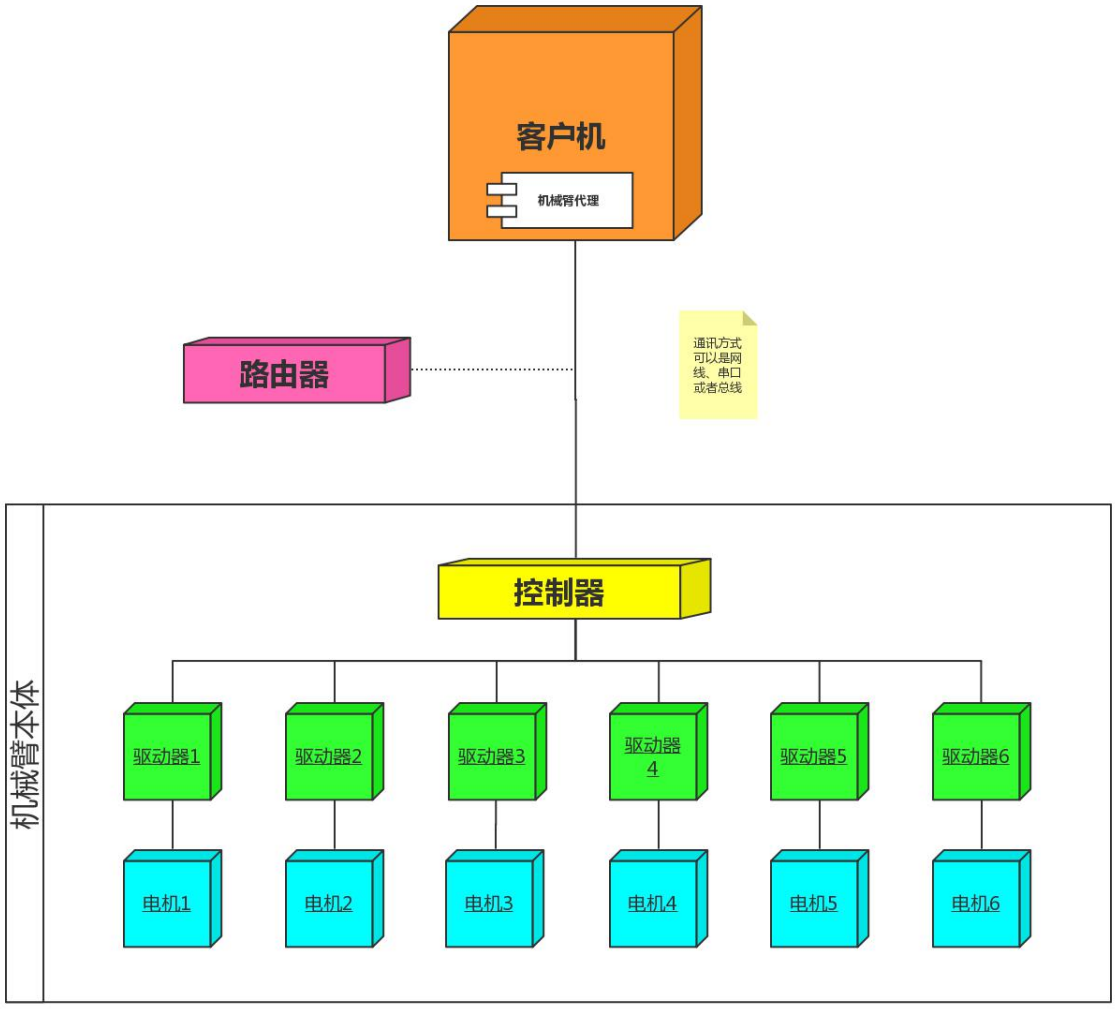


图 2.4 部署图

随着人工智能技术的发展，传统使用示教器来控制机械臂的方式已经很难满足人们对机器人智能化的需求。因此客户机——控制器结构的模型逐渐兴起。

这里以常见的 6 轴桌面机械臂为例，给出图 2.4 部署图。用户使用客户机与机械臂本体的控制器进行交互，而控制器则连接多个驱动器，每个驱动器再通过具体的电压电流调控，可以控制步进电机的运动，进而使得机械臂进行不同的运动。

3 MoveIt 简介

MoveIt 是 ROS 机器人操作系统中，专为控制机械臂控制而设计的软件框架，在机器人领域有着举足轻重的地位。MoveIt 可以帮助不同领域的开发者们快速建立起对机械臂的控制和使用。

3.1 MoveIt 框架介绍

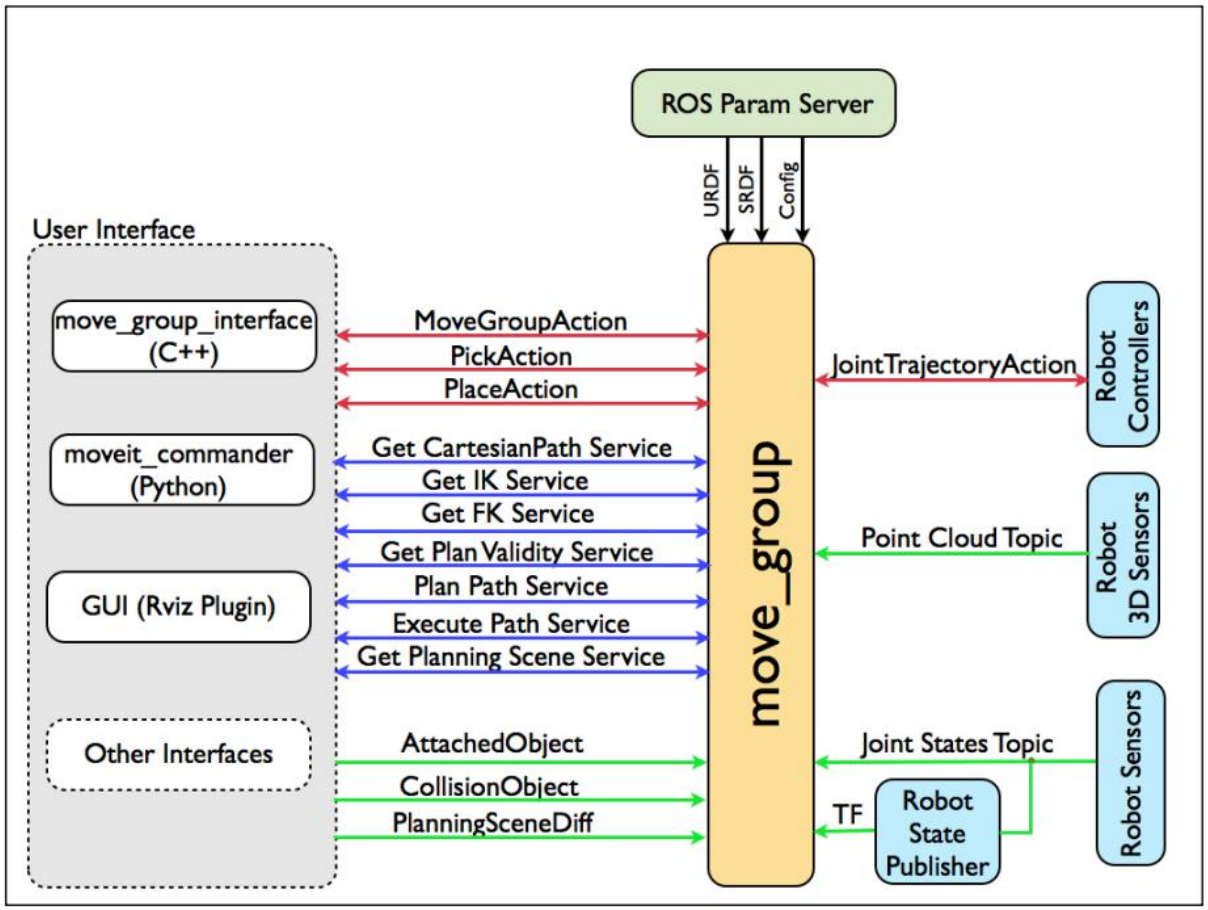


图 3.8　MoveIt 接口结构

如图 3.8 所示，MoveIt 架构较为复杂，但是在机械臂控制中包括用户编程接口、机械臂算法插件接口、机器人驱动接口、碰撞检测接口、点云 3D 传感器接口等。

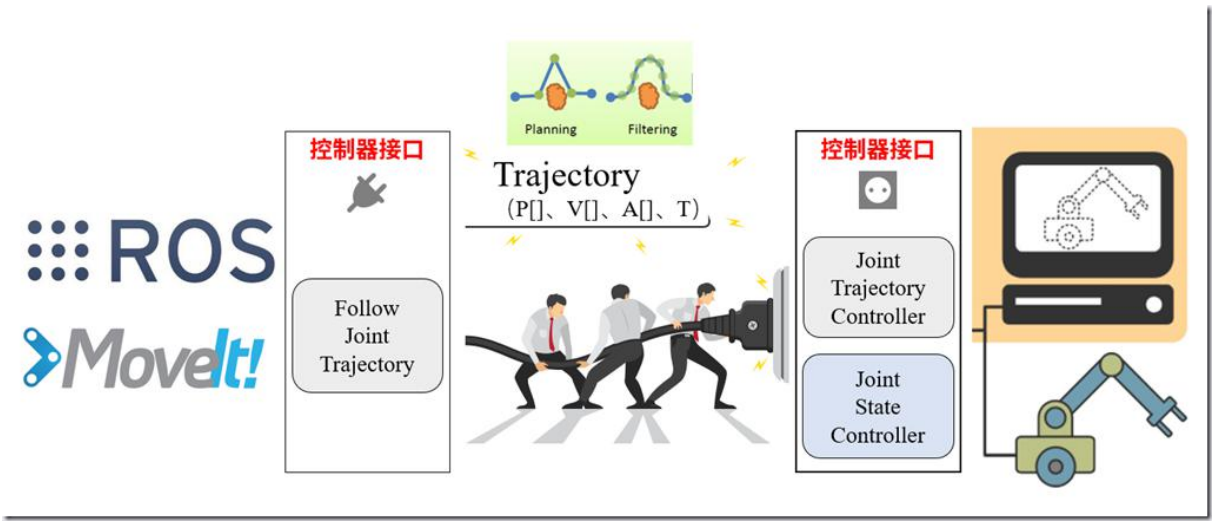


图 3.14　MoveIt 接口关系

如图 3.14 所示，MoveIt 通过三个接口的配合从而操控真实/虚拟机器人。在 ros_cotrol 框架下，用户只要配置好 FollowJointTrajectory 到 JointTrajectoryController 以及 JointStateController 之间的关系，便可以通过 MoveIt 编程接口来操控机械臂。其中关键的问题是如何处理好 Trajectory 中，P[]、V[]、A[]、T，即位置、速度、加速度、时间等数据信息。

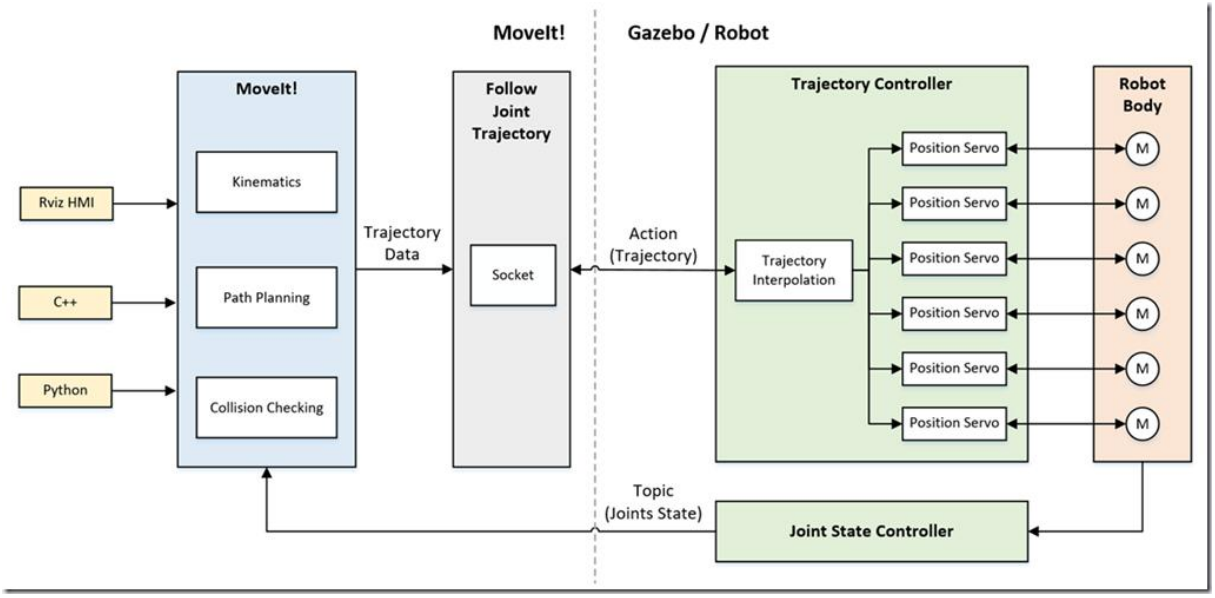


图 3.15　MoveIt 驱动框图

MoveIt 为方便用户理解，提供了 MoveIt 如何操纵机械臂的驱动框图。其中由 MoveIt！，

即 move_group 节点接收到算法或用户传来的轨迹数据，然后进行将轨迹数据传送给机械臂控制器，从而驱动关节伺服，最终控制机械臂本体进行运动。在此同时，由 JointStateController 关节状态控制器将机械臂的状态信息反馈给 MoveIt，从而在上层做到闭环控制器。

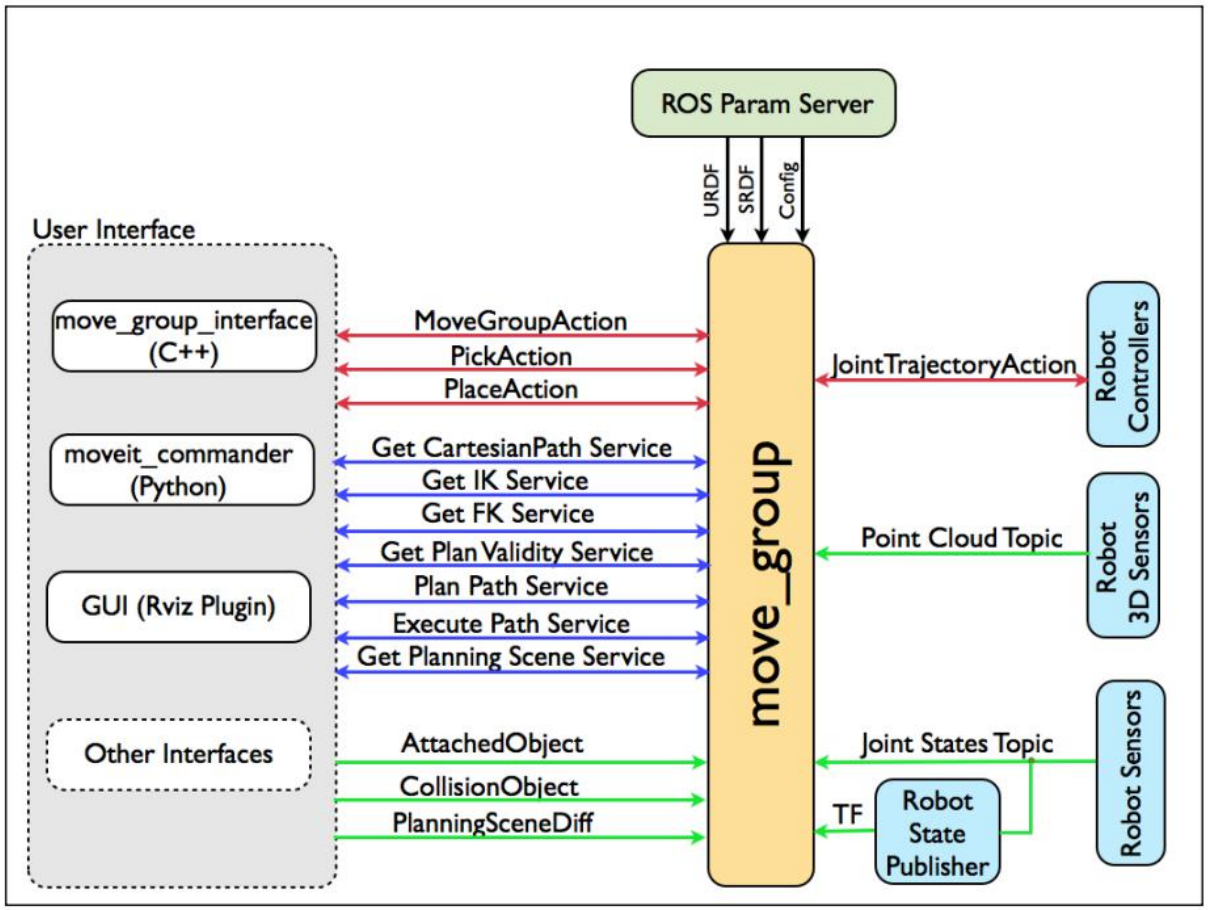


图 3.8 MoveIt 接口结构

在之前的 3.1 节中介绍过 MoveIt 接口结构，其中就包括灰色虚线框内的 User Interface，即用户接口。这些接口包括编程接口、命令控制器、可视化插件等等。

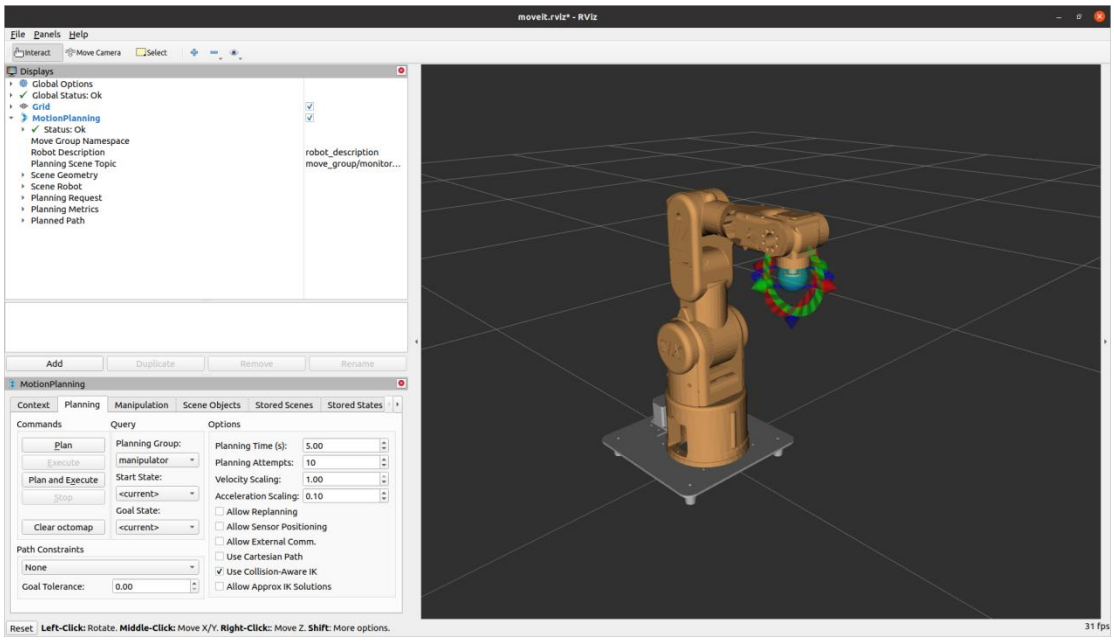


图 3.16　MoveIt 可视化界面

用户可以在 rviz 可视化界面中对机械臂进行控制器和查看。这种方式非常直观，有利于用户快速入门。首先，用户可以通过拖动蓝色小球、6 个方向上的箭头以及不同颜色的圆环，来对机械运动的最终位置进行设置。然后，用户可以点击左下角的 Plan 按键进行机械臂的路径规划，如果规划成功，可以点击 Execute 按键来命令机械臂进行运动。如果用户希望能机械臂能够直接进行运动的话，可以直接点击 Plan and Execute 按键进行控制。此外用户还可以通过选择左下角第二列第三个按键 Goal State 选项，来给机械臂设置一些特殊的位置进行运动。

4 MoveIt 机器人编程

4.1 MoveIt 机器人编程介绍

在介绍本节前，请思考一个问题，如果给机器人下达一个“请为我拿过来一杯水”的指令，机器人该如何如何完成这样一个任务呢?

图 4.1 PR2 机器人

在许多场景中，机器人需要拥有面临在没有人的操控下的自主判断和运动能力，这些判断和识别往往比较复杂。因此大部分的识别判断能力往往是有许多第三方人工智能模块赋予，他们通常采用深度学习，神经网络等技术来为机器人提供其强大的感官能力。而运动能力则有 MoveIt 的编程接口所提供。这里我们先讲一个具体使用的 MoveIt 的例子，然后介绍 MoveIt 的编程接口。

我们这里假设有一个比较简单的场景，即机器人固定在地面上，这样当用户通过 AI 技术获取到机器人需要抓取的物体后，在确定物体的空间坐标和朝向，最终计算出机械臂末端夹爪抓取物体的位置和姿态(即位姿)后，启动抓取装置进行物体抓取，然后运动到要放置物体的位姿后，松开抓取装置进行物体放置。假如机器人是可移动机器人的话，则额外需要考

虑一些 slam，环境构建，自主导航等无人驾驶技术。此外在距离被抓物体多远的情况下，机器人将停止移动，从而启动机械臂进行物体抓取，这些技术都是需要进行进一步的完善和融合。我们可以将这些人工智能的 API 和 MoveIt 的 API 写入一个文件中，从而逐渐实现机器人的智能化。

通过上述问题，笔者希望读者知道在许多场景下，人们往往没有时间通过任何示教器和可视化界面对机器人进行控制来去做一些智能化的任务，取而代之的是使用程序使机器人进行机器人的自主控制。这样读者就应该了解到 MoveIt 编程接口学习的重要性了。

这些接口主要包括机器人的关节运动，笛卡尔空间运动，障碍物添加以及自主避障运动。

本章节仅对一些必要且关键的代码进行讲解。由于大量的人工智能项目采用 python 语言进行开发，这里仅对 python 代码进行分析。本节代码均可以从 xxxx 地址下载。

4.2 MoveIt 机器人通用编程接口

在 MoveIt 编程中有一些常用的代码接口需要介绍，这些在接口几乎在所有操作中都需要使用。这里给大家进行讲解。用户可以参考 MoveIt 官方 API 说明，链接为

http://docs.ros.org/en/noetic/api/moveit_commander/html/

4.2.1

```
import rospy, sys
import moveit_commander
from geometry_msgs.msg import Pose
```

如果我们观察官方的示例代码，就会发现几乎每个头文件都需要引入 rospy、 sys、moveit_commander、Pose 等头文件和内容。其中 rospy 为 ros 的 python 接口 ,sys 是 python 中获取系统特定参数的模块 ,而 moveit_commander 则是操作 moveit 机械臂的重要接口 ,Pose 是描述机械臂末端位姿的数据结构。作为 moveit 操作的核心 moveit_commander 应当重点学习。

4.2.2

```
# 初始化 move_group 的 API
moveit_commander.roscpp_initialize(sys.argv)
# 初始化 ROS 节点
rospy.init_node('moveit_ik_demo')
# 初始化需要使用 move group 控制的机械臂中的 arm group
arm = moveit_commander.MoveGroupCommander('manipulator')
```

首先，我们对 moveit_commander 这个管理者进行初始化，由于 moveit 底层都是 c++所编写，所以调用 roscpp_initialize 来初始化 move_group 的 API，虽然可以使用 roscpp_initialize 的参数为节点命名，但是这种方式并不常用。rospy.init_node 则是初始化 ROS 节点，并在 ROS 网络中命名为 moveit_ik_demo，这种方式是非常推荐的方法。最后一行，则是创建一个我们需要使用的 move_group 实例 arm，来作为机械臂运动的命令管

理器，其参数就是 move_group 的名字。不同的名字将会生成不同机械臂的命令管理器，这种方式为多臂机器人的操控提供了极大的便利。这些名字在 moveit 配置助手里面进行设置，笔者已经将其配置工作为读者做好，并且为了读者进行扩展学习，名字命名为 manipulator。此后，用户只需要对实例 arm 进行操作，便可以进行对机器人本体的操作。

4.2.3

```
# 获取终端 link 的名称
end_effector_link = arm.get_end_effector_link()
# 设置目标位置所使用的参考坐标系
reference_frame = 'link0'
arm.set_pose_reference_frame(reference_frame)
```

这几句代码确定了机器人那些关节可以运动，其参考坐标系又是哪里。这些步骤分别调用实例 arm 的不同接口而完成。需要注意，以后对机器人运动目标位姿的设置，都是以坐标系 link0 作为参考坐标进行设置。如果用户使用机器人正解计算(moveit_fk_demo.py)，这几句代码便可以省去。

4.2.4

```
# 设置机械臂运动的允许误差值
arm.set_goal_position_tolerance(0.01)
arm.set_goal_orientation_tolerance(0.01)
# 设置允许的最大速度和加速度
arm.set_max_acceleration_scaling_factor(0.02)
arm.set_max_velocity_scaling_factor(0.02)
```

这些代码设置了机器人运行过程中的容忍的误差、最大加速度以及最大速度等参数。误差包括位置和姿态，这个参数在机器人逆向运动学中非常重要

```
# 设置机械臂运动的允许误差值
arm.set_goal_joint_tolerance(0.01)
```

此外在机器人正向运动学解算时(moveit_fk_demo.py)，会有各个关节角度的误差，这时我们便需要使用 set_goal_joint_tolerance 参数 ,来代替 set_goal_position_tolerance 和 set_goal_orientation_tolerance。

4.2.5

```
# 当运动规划失败后，允许重新规划
arm.allow_replanning(True)
```

在机器人逆向运动学中，由于机械结构和算法策略问题，常常需要进行多次运算才能获取合适的运动路径。

4.2.6

```
# 控制机械臂运动到预设位态
arm.set_named_target('zero')
arm.go()
```

机器人可以规划到预设的位姿的路径，并进行执行。通过 arm.set_named_target('zero')将机械臂的运行目标 ,设置到预设的位姿上。通过 arm.go() 进行路径规划和运动执行。

4.2.7

```
# 关闭并退出 moveit
moveit_commander.roscpp_shutdown()
moveit_commander.os._exit(0)
```

当用户不再使用机械臂时，可以调用这 2 行代码，来进行相关资源的释放。

4.3 MoveIt 正向运动

正向运动学就是用户直接给出机械臂各个关节的运动角度，从而控制机械臂进行运动。读者可以参考 moveit_fk_demo.py 进行学习。这里仅对和正向运动学有关的代码进行分析。

```
# 设置机械臂的目标位置，使用六轴的位置数据进行描述（单位：弧度）
joint_positions = [0.391410, -0.676384, -0.376217, 0.0, 1.052834, 0.454125]
arm.set_joint_value_target(joint_positions)
# 控制机械臂完成运动
arm.go()
rospy.sleep(1)
```

首先，这里对一个一维度数组进行赋值，数组大小应和机械臂的关节数量保持一致，需要注意在 ROS 中，角度的单位都是弧度，即 3.1415926 弧度便对应 180 度。然后通过 arm.set_joint_value_target(joint_positions)来获取用户设置的角度信息。最终通过 arm.go()来使得机械臂进行运动。

4.4 MoveIt 逆向运动

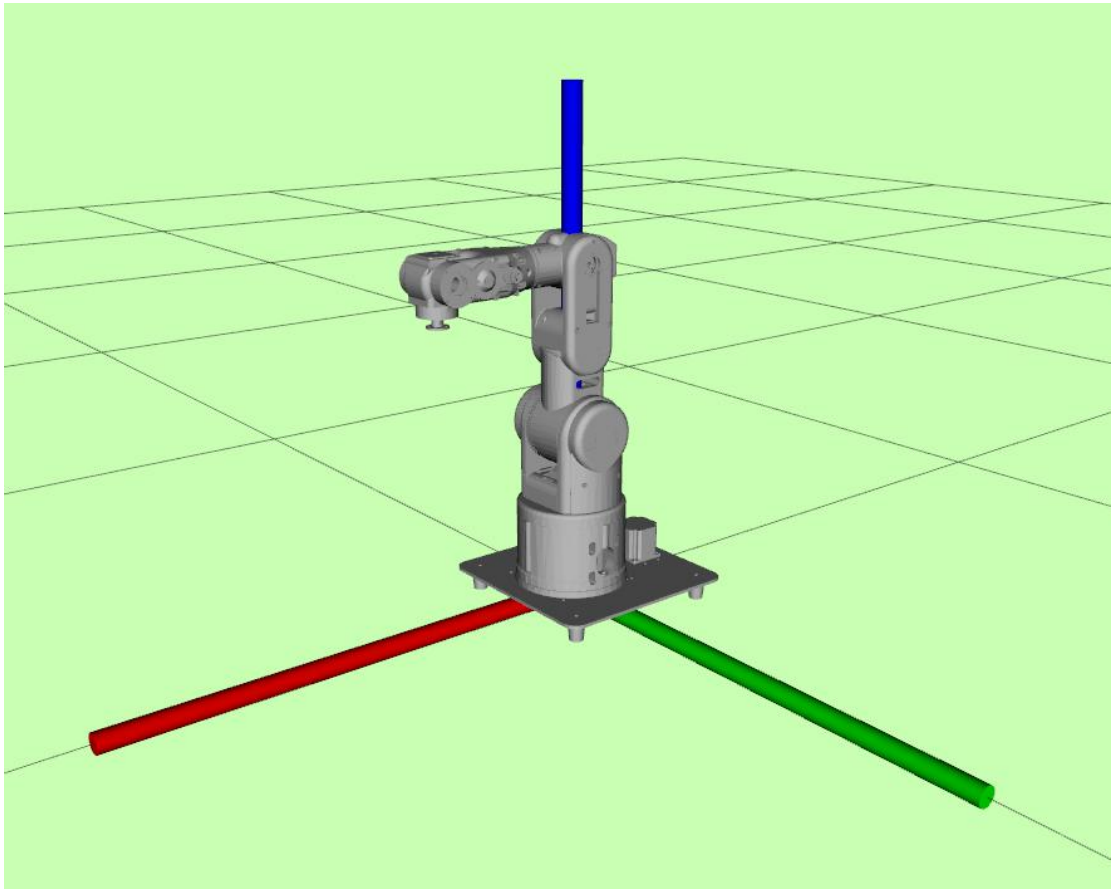
逆向运动学，与正向运动学那种用户直接给出机械臂各个关节运动角度。读者可以参考 moveit_ik_demo.py 进行学习。这里仅对和逆向运动学有关的代码进行分析。

```
# 设置机械臂工作空间中的目标位姿，位置使用 x、y、z 坐标描述，
# 姿态使用四元数描述，基于 link0 坐标系
target_pose = PoseStamped()
target_pose.header.frame_id = reference_frame
target_pose.header.stamp = rospy.Time.now()
target_pose.pose.position.x = 0.2
target_pose.pose.position.y = 0.0
target_pose.pose.position.z = 0.20
target_pose.pose.orientation.x = 0.0
target_pose.pose.orientation.y = 0.70692
target_pose.pose.orientation.z = 0.0
target_pose.pose.orientation.w = 0.70729
# 设置机器臂当前的状态作为运动初始状态
arm.set_start_state_to_current_state()
# 设置机械臂终端运动的目标位姿
arm.set_pose_target(target_pose, end_effector_link)
# 规划运动路径
traj = arm.plan()
# 按照规划的运动路径控制机械臂运动
arm.execute(traj)
rospy.sleep(1)
```

首先 target_pose = PoseStamped()创建一个带有时间戳的目标点实例。

第二行的 target_pose.header.frame_id = reference_frame 通常没有实际意义，但是通常设置为运动物体的基坐标，即机械臂的基坐标 link0。

target_pose.header.stamp = rospy.Time.now()获取当前时间作为时间戳信息。



```
target_pose.pose.position.x = 0.2
target_pose.pose.position.y = 0.0
target_pose.pose.position.z = 0.20
```

这三行代码确定了机器人运动目标点的坐标，即位置信息。这个坐标信息，是以坐标系 link0 作为坐标原点进行计算的。

```
target_pose.pose.orientation.x = 0.0
target_pose.pose.orientation.y = 0.70692
target_pose.pose.orientation.z = 0.0
target_pose.pose.orientation.w = 0.70729
```

这四行代码确定了机器人运动目标点的姿态，即姿态信息。这个姿态信息，是以坐标系 link0 作为坐标原点进行计算的。此姿态信息是以四元数的方式进行设置。用户可以通过运行 rosrun robot_demo euler_orientation 启动此程序，进行从 xyz 欧拉角到 wyzw 四元数的转换。

```
# 设置机器臂当前的状态作为运动初始状态
arm.set_start_state_to_current_state()
# 设置机械臂终端运动的目标位姿
arm.set_pose_target(target_pose, end_effector_link)
# 规划运动路径
traj = arm.plan()
```

通过 arm.set_start_state_to_current_state()来将机械臂当前位姿作为轨迹的起点，随后通过 arm.set_pose_target(target_pose, end_effector_link)来设置末端关节在轨迹终点的运动位姿。最后调用 traj = arm.plan()来获取 MoveIt 为此次运动而规划的轨迹，并保存于 traj 当中。

```
# 按照规划的运动路径控制机械臂运动
arm.execute(traj)
```

当得到一条规划好的轨迹后，调用 arm.execute(traj)，MoveIt 便会向机械臂传递命令，从而进行运动。

4.5 MoveIt 笛卡尔运动

图 4.5 大盘鸡

在生活当中，我们往往有这样的需求，移动某个物体时，总需要这个物品保持一个相对稳定的姿态，比如端盘子、端茶杯、运送装有物体的开口容器等等，这时我们便需要用到笛卡儿运动。笛卡尔运动，正式点讲就是我们需要两位姿之间的光滑路径，而它同时涉及位置及姿态的变化。为了使得机器人能胜任这些特殊任务，MoveIt 为用户提供相关接口来进行笛卡尔运动的轨迹规划。读者可以参考 moveit_cartesian_demo.py 进行学习。这里仅仅讲解核心代码。

```
# 获取当前位姿数据作为机械臂运动的起始位姿
start_pose = arm.get_current_pose(end_effector_link).pose
```

与之前机器人逆向运动中 arm.set_start_state_to_current_state()不同，这里通过 arm.get_current_pose(end_effector_link).pose 来获得机械臂上某个特定关节的位姿信息。

```
# 初始化路点列表
waypoints = []
```

用户需要一个列表(数组)来存储轨迹上不同的轨迹点。

```
# 设置路点数据，并加入路点列表
wpose = deepcopy(start_pose)
wpose.position.z -= 0.2
waypoints.append(deepcopy(wpose))
```

用户需要将符合其特定数据结构的点(如 strat_pose)放入到路点列表(数组)中。通过对函数的查找，可以发现路径点的格式就是 geometry_msgs/Pose。

通过在终端运行 rosmsg show geometry_msgs/Pose 命令，可以看到其具体的定义

```
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

接下来需要调用笛卡尔路径规划的 API，来规划出轨迹

```
fraction = 0.0   #路径规划覆盖率
maxtries = 100   #最大尝试规划次数
attempts = 0     #已经尝试规划次数

# 设置机器臂当前的状态作为运动初始状态
arm.set_start_state_to_current_state()

# 尝试规划一条笛卡尔空间下的路径，依次通过所有路点
while fraction < 1.0 and attempts < maxtries:
    (plan, fraction) = arm.compute_cartesian_path (
                            waypoints,   # waypoint poses，路点列表
                            0.01,        # eef_step，终端步进值
                            0.0,         # jump_threshold，跳跃阈值
                            True)        # avoid_collisions，避障规划
```

首先通过 arm.set_start_state_to_current_state()来将机械臂当前位姿作为轨迹的起点。而(plan, fraction) = arm.compute_cartesian_path (waypoints, 0.01, 0.0, True)则根据路点列表 waypoints，而计算出一条符合笛卡尔运动的轨迹 plan。eff_step 终端步进参数确定了轨迹的疏密程度，这里设置为 0.01 米有一个路径点。jump_threshold 则和具体的解算算法有关，如果对相关算法不是特别了解，这里设置为 0.0 即可。avoid_collisions 则是避障选项，由于笛卡尔运动轨迹规划中可能会出现一些障碍物位于某两个路径点当中，所以如果用户希望规划出的笛卡尔运动轨迹不和障碍物发生碰撞的话，就可以将此参数设置为 True。

```
arm.execute(plan)
```

当拥有笛卡尔运动轨迹，即 plan 后，调用 arm.execute(plan)来使机械臂对轨迹进行执行。