睿尔曼 RM65-B 机器人 ROS 使用说明书 V1.6

睿尔曼智能科技（北京）有限公司

文件修订记录：

版本号	时间	备注
V1.0	2021-8-18	拟制
V1.5	2022-6-25	增加 MoveJ_P、切换工具坐标系、切换工作坐标系、查询机械臂当前状态功能中使用的话题以及消息描述；增加 MoveJ、MoveJ_P、MoveL 的编程示例
V1.6	2022-8-10	增加查询六维力数据、六维力数据清零、自动设置六维力重心参数、手动标定六维力数据、停止标定力传感器重心、开始复合模式拖动示教、力位混合控制、结束力位混合控制、开启透传力位混合控制补偿模式、透传力位混合补偿、关闭透传力位混合补偿模式的话题以及以上功能执行状态的返回话题

目录

1.package 介绍

序号	名称	作用
1	rm_65_description	RM65-B 机器人描述功能包，其中有创建好的机器人模型和配置文件 rm_65.urdf.xacro：不带手爪的 RM65-B 机器人模型文件
2	rm_65_moveit_config	使用 Setup Assistant 工具根据机械臂 URDF 模型 rm_65.urdf.xacro 创建生成的一个 MoveIt!配置的功能包，它包含了大部分 MoveIt!启动所需的配置文件和启动文件，以及包含一个简单的演示 demo
3	rm_gazebo	gazebo 仿真机器人所用到参数和文件配置
4	rm_65_demo	MoveIt!编程示例，包括场景规划、避障规划和 pick and place
5	rm_msgs	RM65-B 所用到的所有控制消息和状态消息
6	rm_control	机器人控制器，将 Moveit 规划的机械臂轨迹，通过三次样条插值细分，按照 20ms 的控制周期发给 rm_driver 节点，周期可调，但是应大于 10ms
7	rm_driver	（1）与机械臂通过以太网口建立 socket 连接，机器人 IP 地址：192.168.1.18，请保证上位机的 IP 在同一局域网内，使用 ROS 控制机械臂，请务必确认机械臂处于以太网口通信模式； （2）订阅各 topic 数据，更新 RVIZ 内机械臂各关节角度
8	rm_bringup	启动机器人，运行对应的 launch 文件后，可自动运行 rm_driver，rm_control 和 moveit 交互 RVIZ 界面，直接在仿真界面中拖拽机器人即可控制真实机器人运动

2.环境要求

- 系统：Ubuntu 18.04 或 Ubuntu 20.04
- ROS：melodic 或 noetic

■ 其余软件要求：Moveit!已安装；Gazebo 可用；ros_control 插件可用

3.源码安装编译

新建名称为 ws_rmrobot 的工作空间，执行如下命令：

```
mkdir -p ~/ws_rmrobot/src

cd ~/ws_rmrobot/src/
```

然后将提供的 rm_robot 源码包拷贝到 ws_rmrobot 工作空间的 src 目录下或将源码包拷贝到自己创建的其他工作空间的 src 目录下：

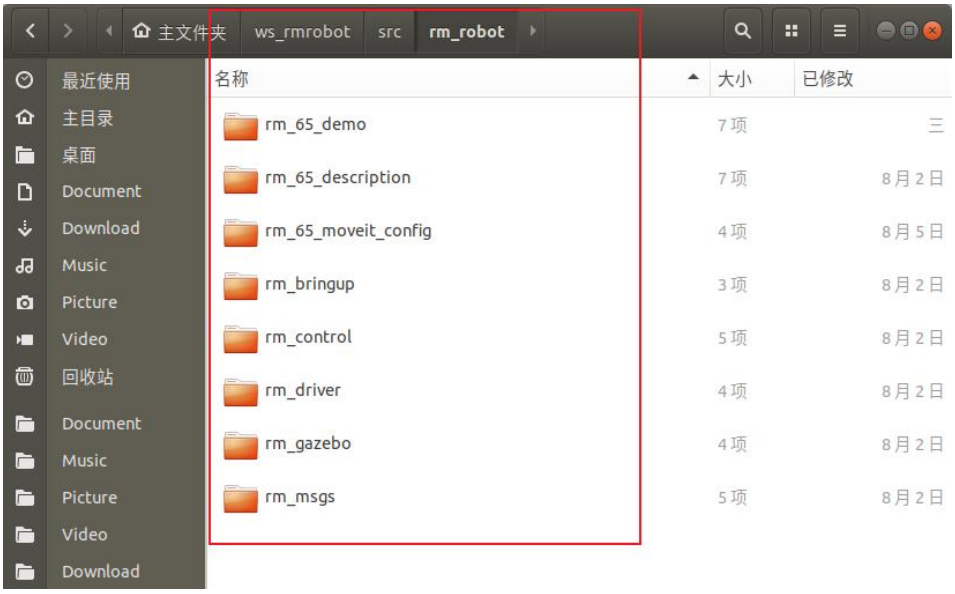

图 3-1 将 rm_robot 源码包拷贝到 ws_rmrobot/src 下

通过 rosdep 安装源码包依赖，执行以下命令（若为 ROS Noetic 版本则将命令中的 melodic 改为 noetic）：

```
rosdep install -y --from-paths . --ignore-src --rosdistro melodic -r
```

使用 catkin 工具配置工作空间并进行编译，执行如下命令：

```
cd ~/ws_rmrobot

catkin init
```

```
catkin build rm_msgs

catkin build
```

编译完成如下图所示：

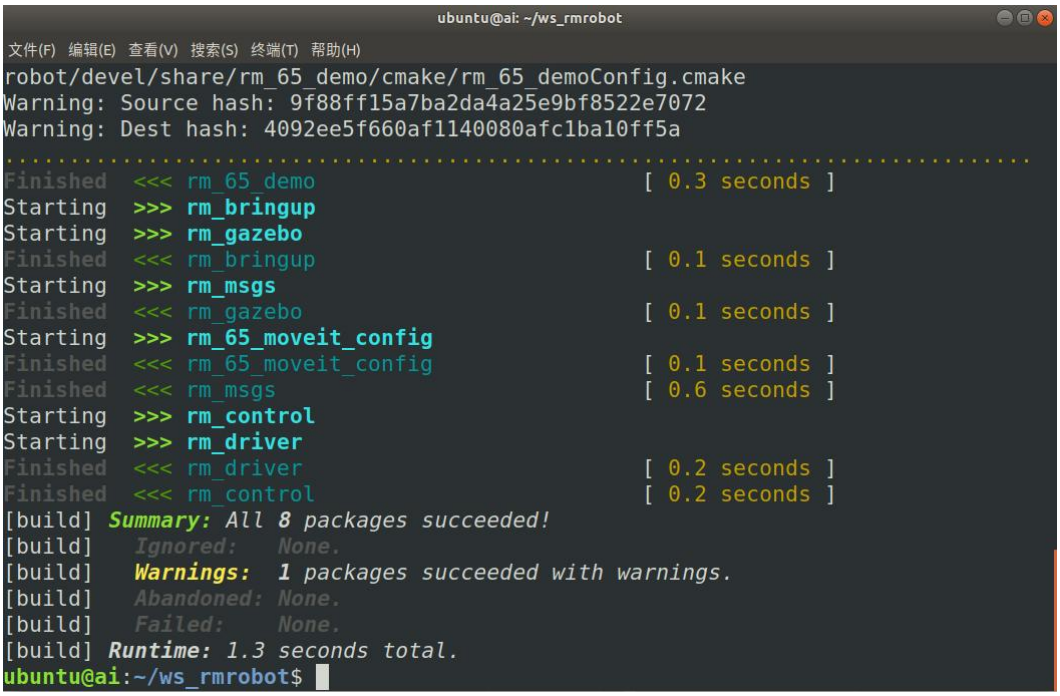

图 3-2 rm_robot 源码包编译成功

4.在 rviz 中显示机械臂模型

4.1 机器人描述功能包

在 rm_robot 源码包中包含了 rm_65_description 功能包，其中有创建好的机器人模型和配置文件。

rm_65_description 功能包中主要包含 urdf、meshes、launch 和 config 四个文件夹。

- urdf：用于存放机器人模型的 URDF 或 xacro 文件。
- meshes：用于放置 URDF 中引用的模型渲染文件。
- launch：用于存放相关启动文件。

- config：用于保存 rviz 的配置文件。

4.2 在 rviz 中显示模型

在 rm_65_description 功能包 launch 文件夹中已经创建用于显示 rm_65 模型的 launch 文件 rm_65_description/launch/display_rm65.launch。

打开终端进入工作空间执行以下命令运行该 launch 文件：

```
cd ~/ws_rmrobot

source devel/setup.bash

roslaunch rm_65_description display.launch
```

如果一切正常，可以在打开的 rviz 中看到如图 4-1 所示的机器人模型。

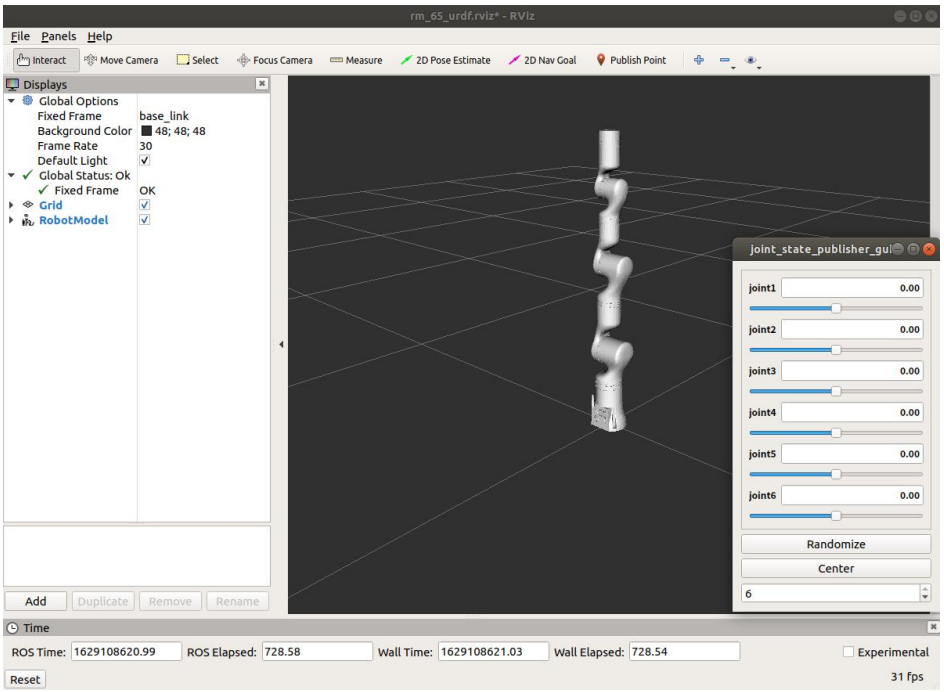

图 4-1 在 rviz 中显示 RM65-B 机械臂模型

运行成功后，不仅启动了 rviz，而且出现了一个名为“joint_state_publisher”的 UI。这是因为我们在启动文件中启动了 joint_state_publisher 节点，该节点可以发布每个 joint（除 fixed 类型）的状态，而且可以通过 UI 对 joint 进行控制。所以在控制界面中用鼠标滑动控制条，rviz 中对应的机械臂关节就会转动。

如果 rviz 中未显示模型，则手动修改“Fixed Frame”为“base_link”，然

后点击左侧下方的 Add 按钮在弹出的界面中找到“RobotModel”添加即可，如图 4-2~4-3 所示：

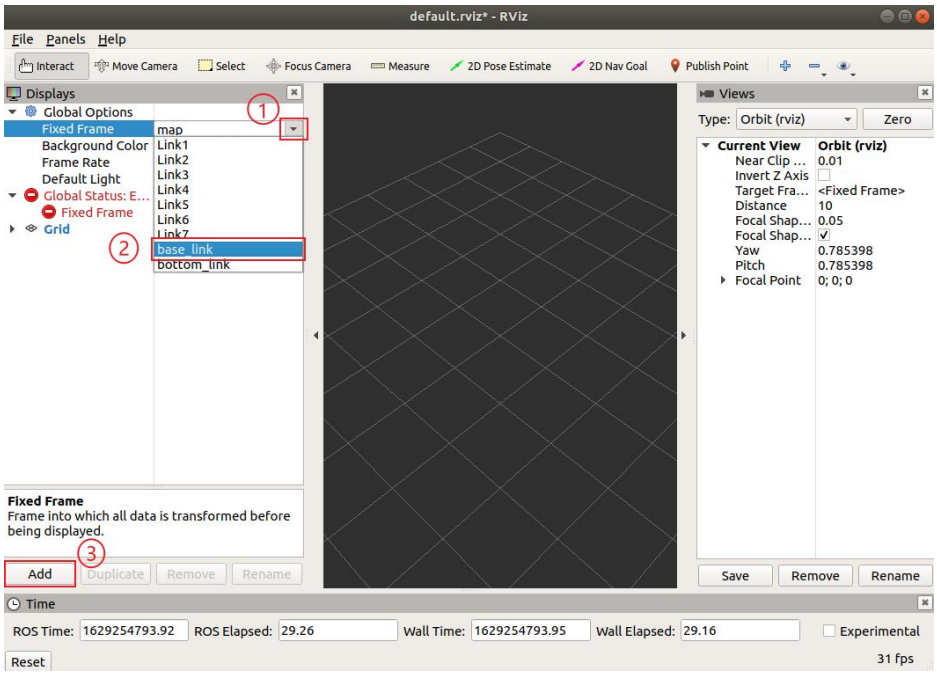

图 4-2 修改 FixedFrame 为 base_link

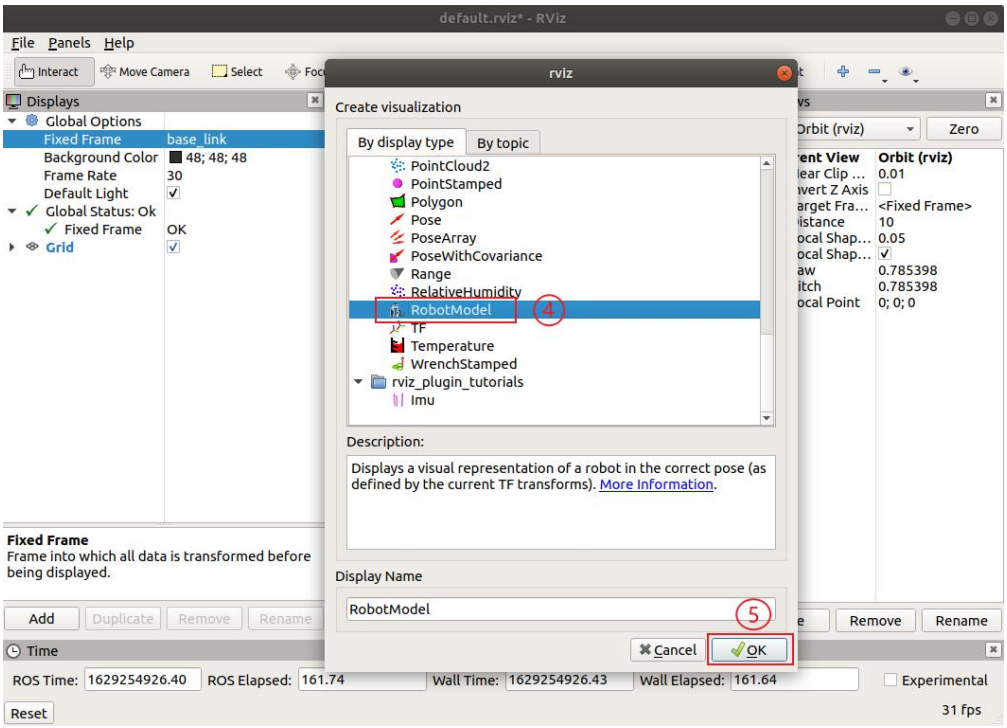

图 4-3 rviz 中添加 RobotModel

5.启动 MoveIt!

5.1MoveIt!简介

在实现机械臂的自主抓取中机械臂的运动规划是其中最重要的一部分，其中包含运动学正逆解算、碰撞检测、环境感知和动作规划等。常见机械臂的运动规划大都采用的是 ROS 系统提供的 MoveIt! 规划。

MoveIt! 是 ROS 系统中集合了与移动操作相关的组件包的运动规划库。它包含了运动规划中所需要的大部分功能，同时其提供友好的配置和调试界面便于完成机器人在 ROS 系统上的初始化及调试。

官方网站：http://moveit.ros.org/，上边有 MoveIt！的教程和 API 说明。

5.2 安装 MoveIt!

MoveIt!需要安装才能使用，如果未安装，请执行如下命令进行安装（若为 ROS Noetic 版本则将命令中的 melodic 改为 noetic）。

```
sudo apt install ros-melodic-moveit

sudo apt install ros-melodic-moveit-*
```

5.3 运行 RM65-B 机械臂的 MoveIt!演示 demo

在 rm_robot 源码包中包含了 rm_65_moveit_config 功能包，它是使用 Setup Assistant 工具根据机械臂 URDF 模型创建生成的一个 MoveIt!配置的功能包，它包含了大部分 MoveIt!启动所需的配置文件和启动文件，以及包含一个简单的演示 demo。

打开终端进入工作空间执行以下命令运行 RM65-B 机械臂的 MoveIt!演示 demo：

```
cd ~/ws_rmrobot

source devel/setup.bash

roslaunch rm_65_moveit_config demo.launch
```

启动成功后，可以看到如图 5-1 所示的界面。

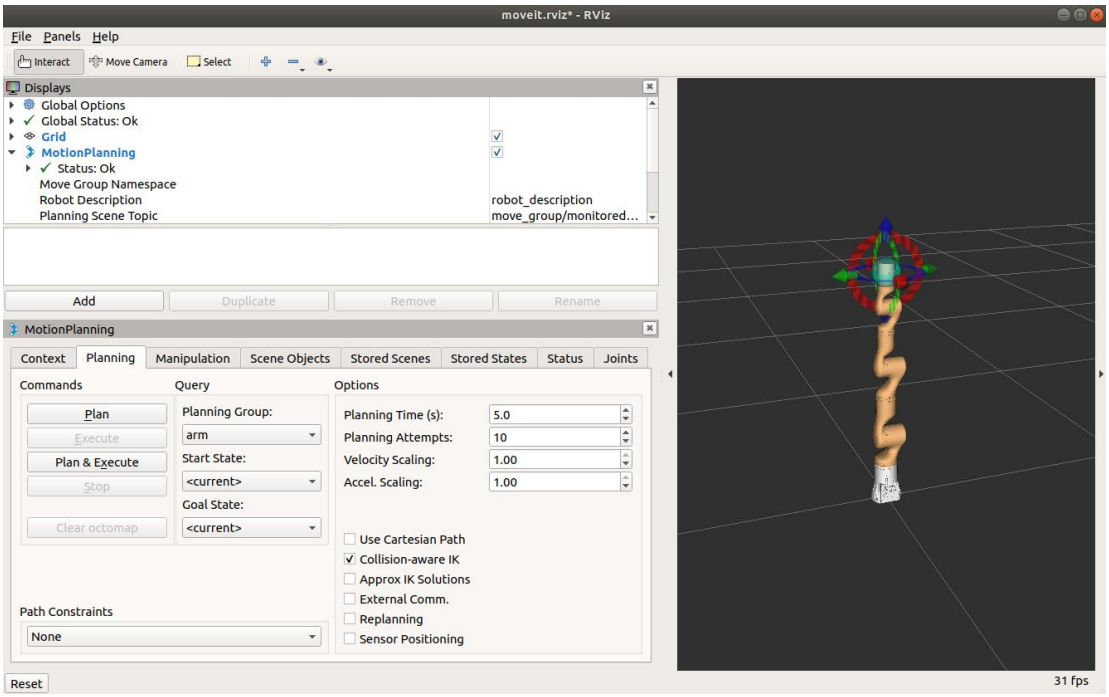

图 5-1 MoveIt! demo 的启动界面

这个界面在 rviz 的基础上加入了 MoveIt!插件，通过左下角的插件窗口可以配置 MoveIt!的相关功能，控制机械臂完成运动规划。例如通过 MoveIt!插件，可以控制机械臂完成拖动规划、随机目标规划、初始位姿更新、碰撞检测等功能。

5.3.1 拖动规划

拖动机械臂的前端，可以改变机械臂的姿态。然后在 Planning 标签页中点击“Plan & Execute”按钮，MoveIt!开始规划路径，并且控制机器人向目标位置移动，从右侧界面可以看到机器人运动的全部过程（见图 5-2）。

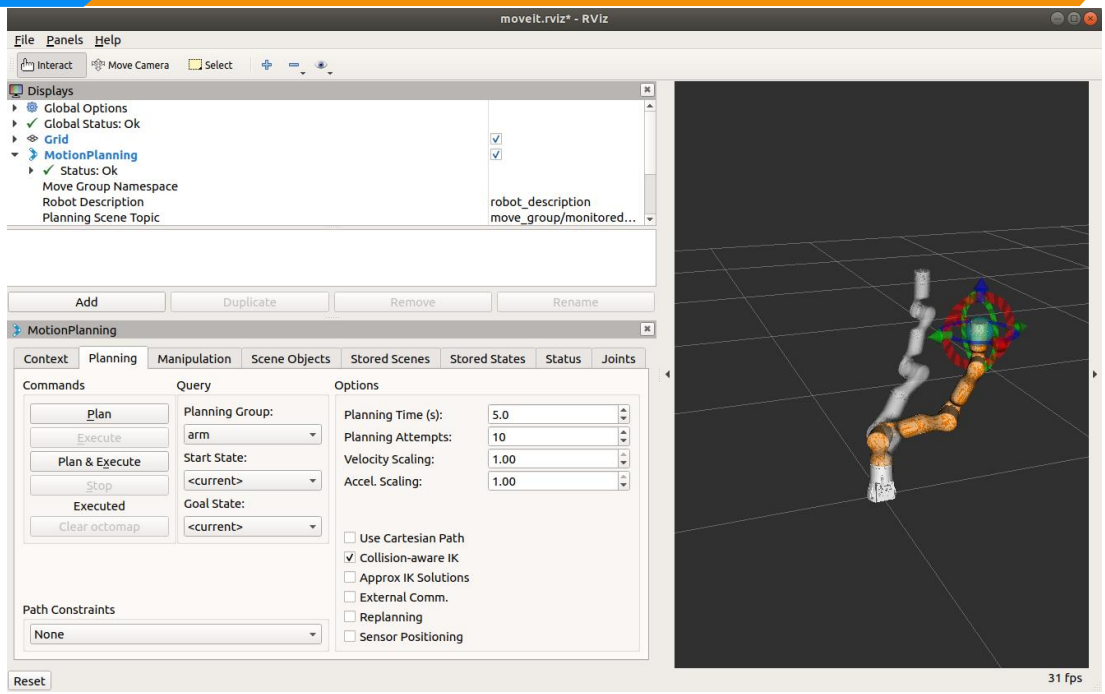

图 5-2 拖动规划的运动效果

5.3.2 选择目标姿态规划

在 Planning 标签页中点击 Goal State 的下拉列表可以选择机械臂的目标姿态，然后点击“Plan & Execute”按钮，MoveIt!开始规划路径，并且控制机器人向目标位置移动，从右侧界面可以看到机器人运动的全部过程（见图 5-3）。

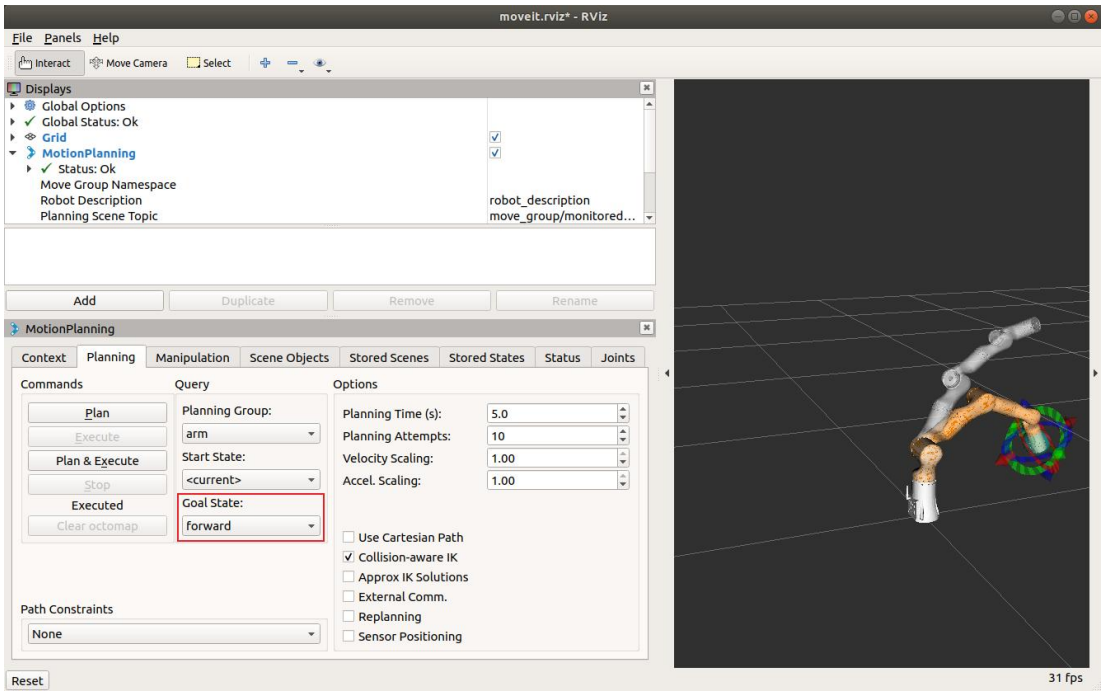

图 5-3 选择目标姿态规划的运动效果

6.使用 MoveIt!控制 Gazebo 中的机械臂

6.1 概述

MoveIt!与 Gazebo 的联合仿真，其主要思路为搭建 ros_control 和 MoveIt!的桥梁。先在 MoveIt!端配置关节和传感器接口 yaml 文件，将其加载到 rviz 端；再在机器人端配置 ros_control 和接口 yaml 文件，将机器人加载到 Gazebo。最后同时启动加载有 ros_control 的 Gazebo 和加载有 MoveIt 的 rviz，达到联合仿真的目的。

6.2 配置说明

6.2.1 MoveIt!端的配置

1）控制器配置文件 controllers_gazebo.yaml

controllers_gazebo.yaml 在 rm_65_moveit_config/config 目录下，代码如下：

```
controller_manager_ns: controller_manager
controller_list:
  - name: arm/arm_joint_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
```

```
    - joint5
    - joint6
```

2）修改启动文件 rm_65_moveit_controller_manager.launch.xml

修改 rm_65_moveit_config 功能包中的 rm_65_moveit_controller_manager.launch.xml，配置加载刚才创建的 controllers_gazebo.yaml 文件到参数服务器，代码如下：

```
<launch>
<arg name="execution_type" default="FollowJointTrajectory" />

<!-- loads moveit_controller_manager on the parameter server which is taken as ar
gument
if no argument is passed, moveit_simple_controller_manager will be set -->
<arg name="moveit_controller_manager" default="moveit_simple_controller_manager/
MoveItSimpleControllerManager" />
<param name="moveit_controller_manager" value="$(arg moveit_controller_manager)
"/>

<!-- load controller_list -->
<arg name="use_controller_manager" default="true" />
<param name="use_controller_manager" value="$(arg use_controller_manager)" />

<!-- loads ros_controllers to the param server -->
```

```
<!-- <rosparam file="$(find rm_65_moveit_config)/config/ros_controllers.yaml"/> -->

<!-- <rosparam file="$(find rm_65_moveit_config)/config/controllers.yaml"/> -->

<rosparam file="$(find rm_65_moveit_config)/config/controllers_gazebo.yaml"/>

</launch>
```

3）**启动文件** moveit_planning_execution.launch

moveit_planning_execution.launch **是在** rm_65_moveit_config/launch **目录下新创建的启动文件，用以加载** MoveIt **和** rviz**，最后运行关节状态发布器，用以向** Gazebo **中发布指令，代码如下：**

```
<launch>

 # The planning and execution components of MoveIt! configured to

 # publish the current configuration of the robot (simulated or real)

 # and the current state of the world as seen by the planner

 <include file="$(find rm_65_moveit_config)/launch/move_group.launch">

  <arg name="publish_monitored_planning_scene" value="true" />



 </include>

 # The visualization component of MoveIt!

 <include file="$(find rm_65_moveit_config)/launch/moveit_rviz.launch"/>



  <!-- We do not have a robot connected, so publish fake joint states -->

  <node name="joint_state_publisher" pkg="joint_state_publisher"

type="joint_state_publisher">
```

```
    <param name="/use_gui" value="false"/>
    <rosparam param="/source_list">[/arm/joint_states]</rosparam>
  </node>
</launch>
```

6.2.2 机器人端的配置

- 关节轨迹控制器

MoveIt!完成运动规划后的输出接口是一个命名为“FollowJointTrajectory”的 action，其中包含了一系列规划好的路径点轨迹，这些信息可以通过 ros_control 为我们提供的一个名为“Joint Trajectory Controller”的控制器插件转换成 Gazebo 中机器人需要的 joint 位置。

1）配置文件 rm_65_trajectory_control.yaml

“FollowJointTrajectory”控制器的使用首先需要一个配置文件对机械臂六个轴的轨迹控制配置控制参数，即 rm_gazebo/config 目录下的 rm_65_trajectory_control.yaml 配置文件，代码如下：

```
arm:
  arm_joint_controller:
    type: "position_controllers/JointTrajectoryController"
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6
```

```
    gains:
      joint1:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      joint2:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      joint3:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      joint4:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      joint5:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
      joint6:    {p: 1000.0, i: 0.0, d: 0.1, i_clamp: 0.0}
```

2）控制器启动文件

通过一个launch文件加载Joint Trajectory Controller，即rm_gazebo/launch目录下的 arm_65_trajectory_controller.launch 文件，代码如下：

```
<launch>
    <rosparam file="$(find rm_gazebo)/config/rm_65_trajectory_control.yaml" command="load"/>
    <node name="arm_controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
          output="screen" ns="/arm" args="arm_joint_controller"/>
</launch>
```

- 关节状态控制器

关节状态控制器是一个可选插件，主要作用是发布机器人的关节状态和 TF 变换，否则在 rviz 的 Fixed Frame 设置中看不到下拉列表中的坐标系选项，只能手动输入，但是依然可以正常使用。

1）关节状态控制器配置文件 arm_gazebo_joint_states.yaml

arm_gazebo_joint_states.yaml 配置文件在 rm_gazebo/config 目录下，代码如下：

```
arm:

  # Publish all joint states ----------------------------------

  joint_state_controller:

    type: joint_state_controller/JointStateController

    publish_rate: 50
```

2）启动文件 arm_gazebo_states.launch 加载参数

arm_gazebo_states.launch 文件在 rm_gazebo/launch 目录下，代码如下：

```
<launch>

    <!-- 将关节控制器的配置参数加载到参数服务器中 -->

    <rosparam file="$(find rm_gazebo)/config/arm_gazebo_joint_states.yaml" comma
nd="load"/>



    <node name="joint_controller_spawner" pkg="controller_manager" type="spawne
r" respawn="false"

          output="screen" ns="/arm" args="joint_state_controller" />



    <!-- 运行 robot_state_publisher 节点，发布 tf  -->

    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_st
ate_publisher"

        respawn="false" output="screen">
```

```
        <remap from="/joint_states" to="/arm/joint_states" />
    </node>
</launch>
```

6.2.3 启动文件 arm_65_bringup_moveit.launch

顶层启动文件 arm_65_bringup_moveit.launch 在 rm_gazebo/launch 目录下，用以启动 Gazebo，并且加载所有的控制器，最后启动 MoveIt!，代码如下：

```
<launch>
    <!-- Launch Gazebo  -->
    <include file="$(find rm_gazebo)/launch/arm_world.launch" />

    <!-- ros_control arm launch file -->
    <include file="$(find rm_gazebo)/launch/arm_gazebo_states.launch" />

    <!-- ros_control trajectory control dof arm launch file -->
    <include file="$(find rm_gazebo)/launch/arm_65_trajectory_controller.launch" />
    <!-- moveit launch file -->
    <include file="$(find
rm_65_moveit_config)/launch/moveit_planning_execution.launch" />
</launch>
```

6.3 运行效果

启动节点前，需要确保 rm_65_moveit_config/launch/rm_65_moveit_controller_manager.launch.xml 文件中配置加载到参数服务器的是 controllers_gazebo.yaml 文件，见图 6-1：

rm_65_moveit_controller_manager.launch.xml ×

home > ubuntu > ws_rmrobot > src > rm_robot > rm_65_moveit_config > launch > rm_65_moveit_controller_manager.launch.xml

```
<launch>
  <arg name="execution_type" default="FollowJointTrajectory" />

  <!-- loads moveit_controller_manager on the parameter server which is taken as argument
   if no argument is passed, moveit_simple_controller_manager will be set -->
  <arg name="moveit_controller_manager" default="moveit_simple_controller_manager/MoveItSimpleControllerManager" />
  <!-- <arg name="moveit_controller_manager" default="moveit_fake_controller_manager/MoveItFakeControllerManager" /> -->
  <param name="moveit_controller_manager" value="$(arg moveit_controller_manager)"/>

  <!-- load controller_list -->
  <arg name="use_controller_manager" default="true" />
  <param name="use_controller_manager" value="$(arg use_controller_manager)" />

  <!-- loads ros_controllers to the param server -->
  <!-- <rosparam file="$(find rm_65_moveit_config)/config/ros_controllers.yaml"/> -->
  <!-- <rosparam file="$(find rm_65_moveit_config)/config/controllers.yaml"/> -->
  <rosparam file="$(find rm_65_moveit_config)/config/controllers_gazebo.yaml"/>
</launch>
```

图 6-1 rm_65_moveit_controller_manager.launch.xml 加载 controllers_gazebo.yaml

执行以下命令运行 MoveIt!和 Gazebo：

```
cd ~/ws_rmrobot

source devel/setup.bash

roslaunch rm_gazebo arm_65_bringup_moveit.launch
```

启动后打开的 Gazebo 如图 6-2 所示：

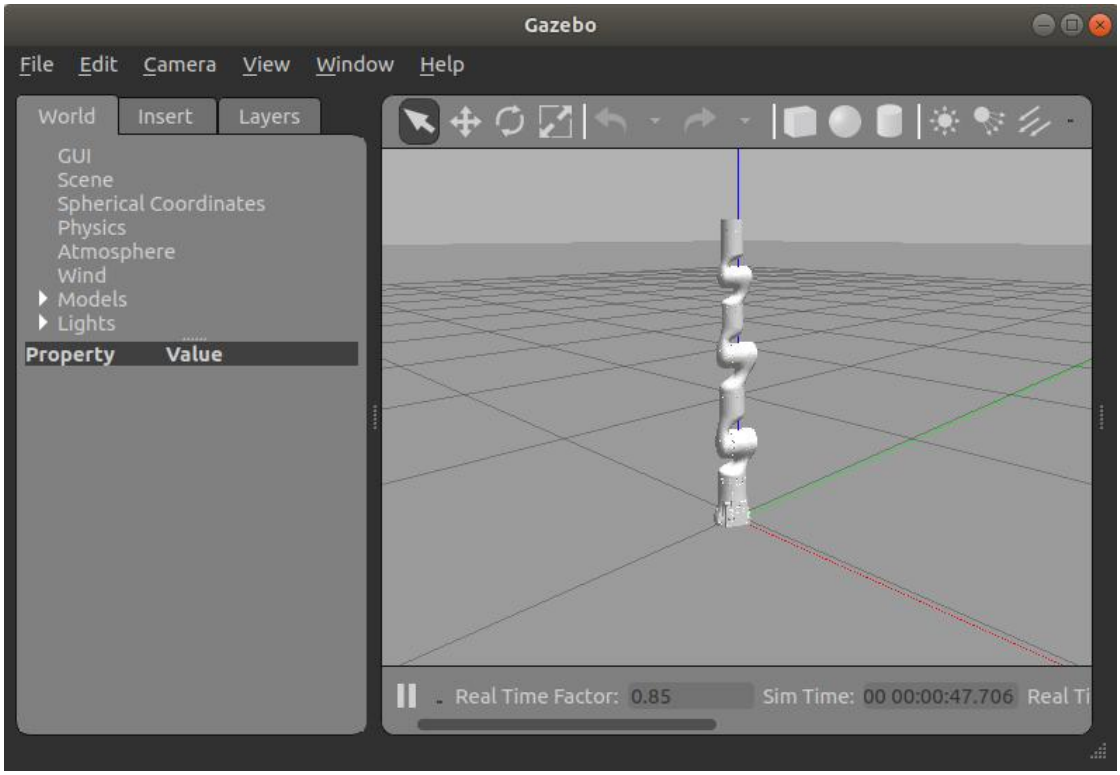

图 6-2 RM65-B 机器人在 Gazebo 中显示效果

启动后打开的 rviz 如图 6-3 所示：

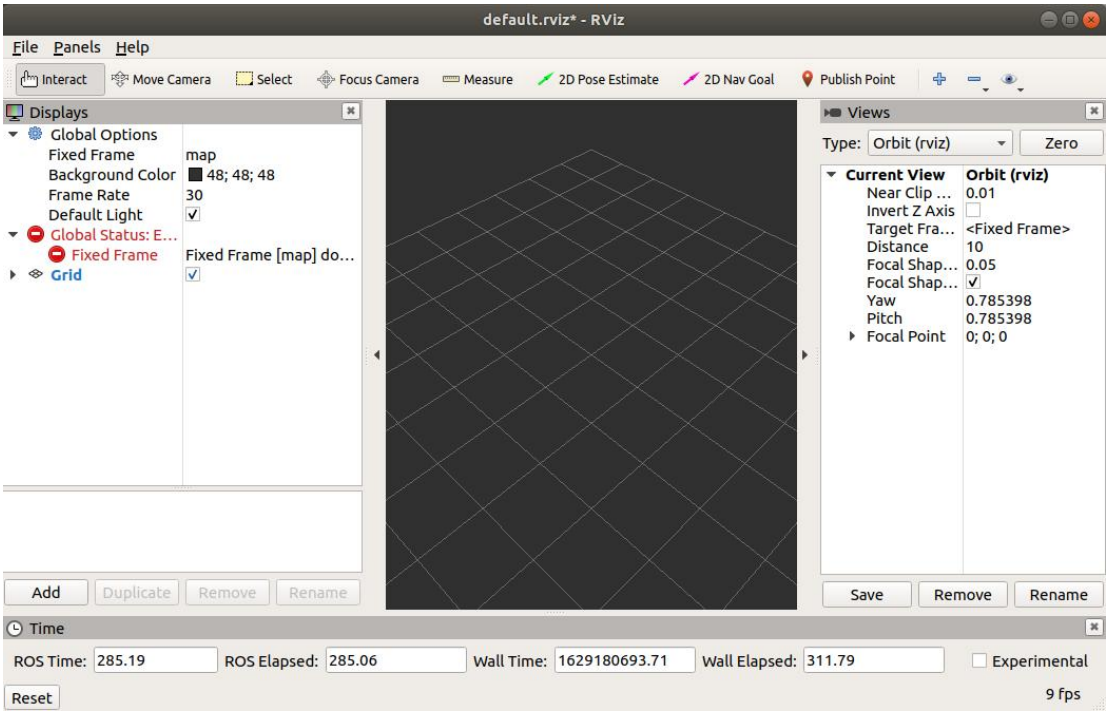

图 6-3 rviz 启动界面

在打开的 rviz 中，暂时还看不到任何机器人模型，甚至还会提示一些错误，接下来进行简单配置解决这些问题。

首先将“Fixed Frame”修改成“base_link”；然后点击左侧插件列表栏中的“Add”按钮，将 MotionPlanning 加入控制窗口。此时应该可以看到机器人出现在右侧主界面中，Gazebo 中机器人的姿态应该和 rvize 中的显示一致。操作如图 6-4~6-6 所示：

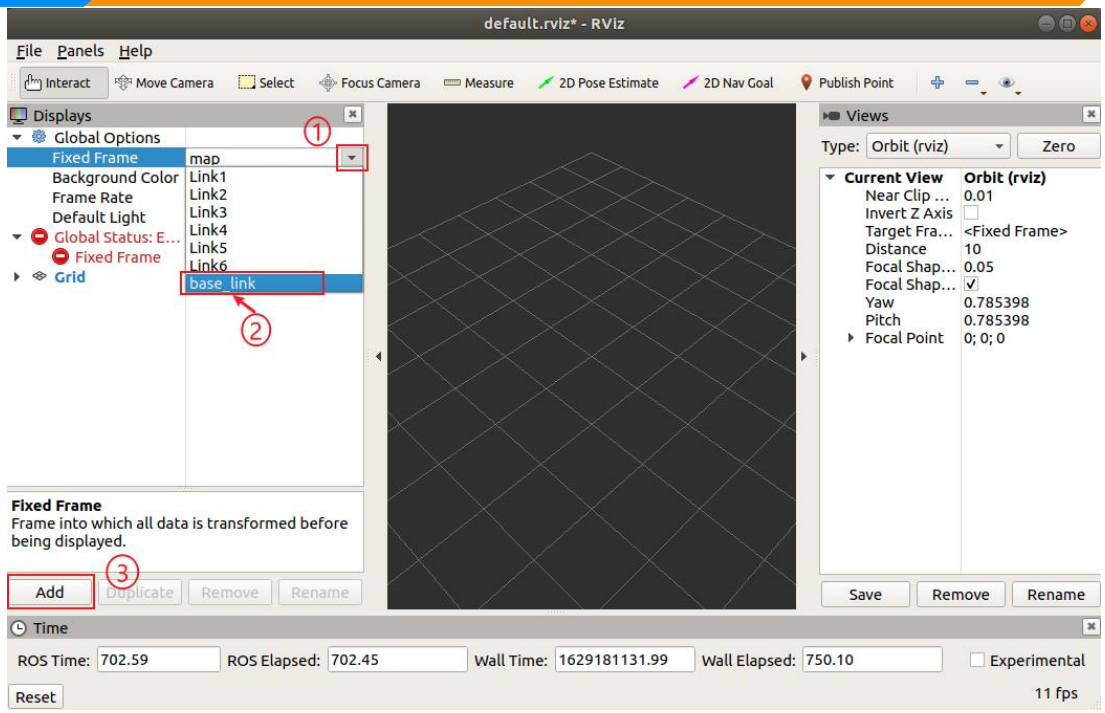

图 6-4 在 rviz 中将“Fixed Frame”修改成“base_link”

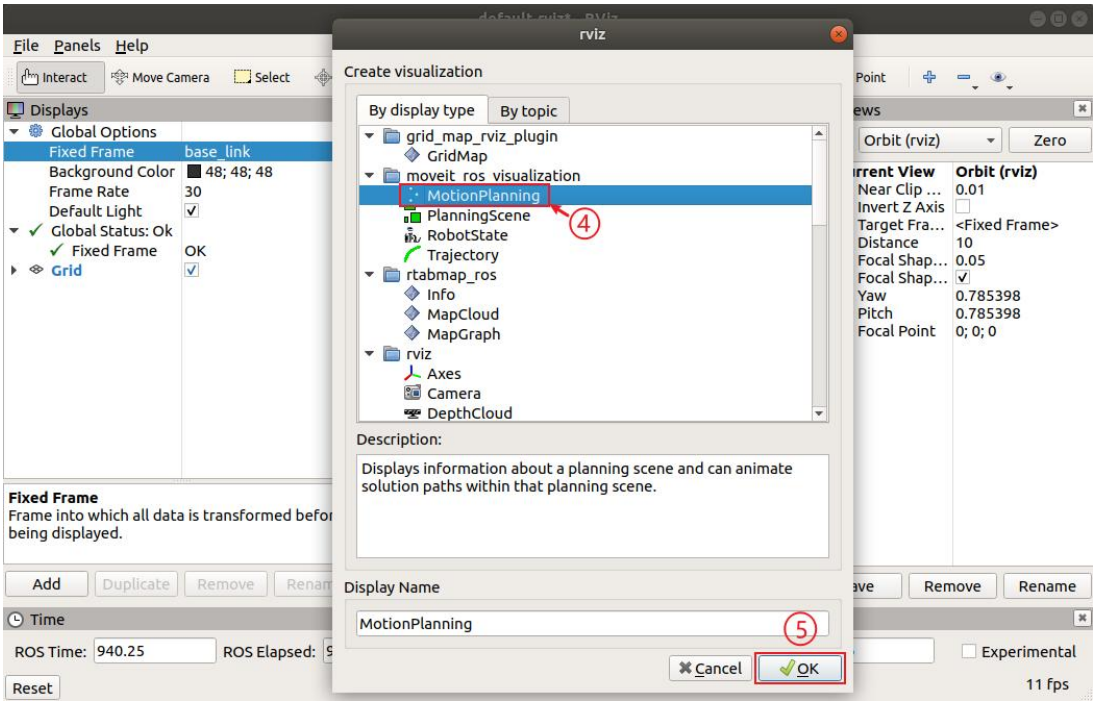

图 6-5 rviz 中将 MotionPlanning 加入控制窗口

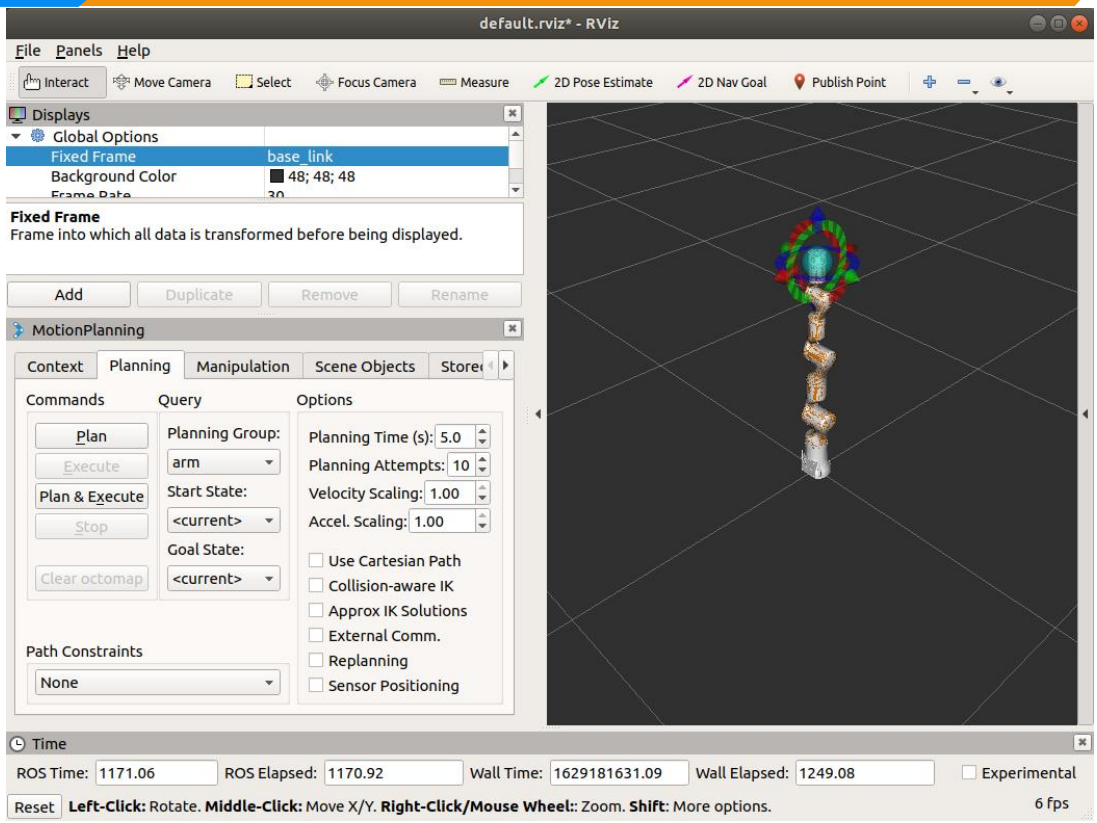

图 6-6 rviz 中显示 RM65-B 机器人

接下来使用 MoveIt!规划运动的几种方式就可以控制 Gazebo 中的机器人了，例如图 6-7 拖动机器人末端到一个位置，然后点击“Plan & Execute”按钮，可以看到 rviz 中机器人开始规划执行并且可以看到 Gazebo 中的机器人开始运动且与 rviz 中的机器人模型保持一致，如图 6-8 所示。

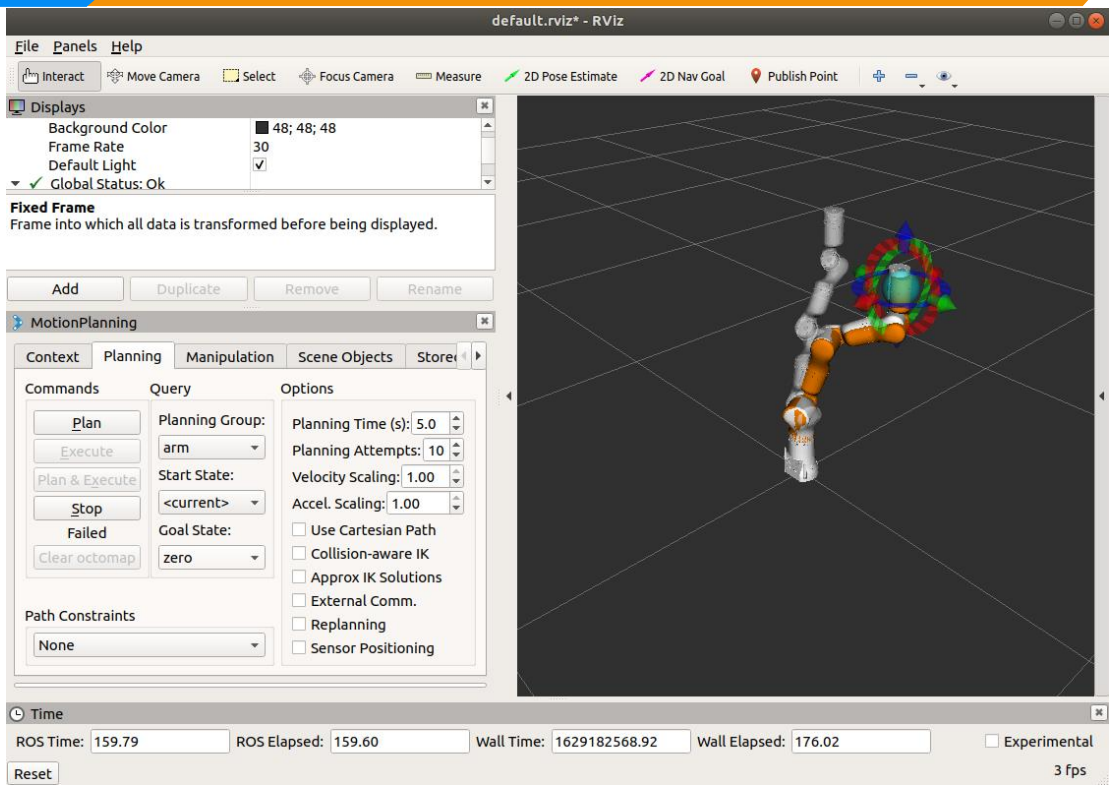

图 6-7 使用 MoveIt!拖动规划执行

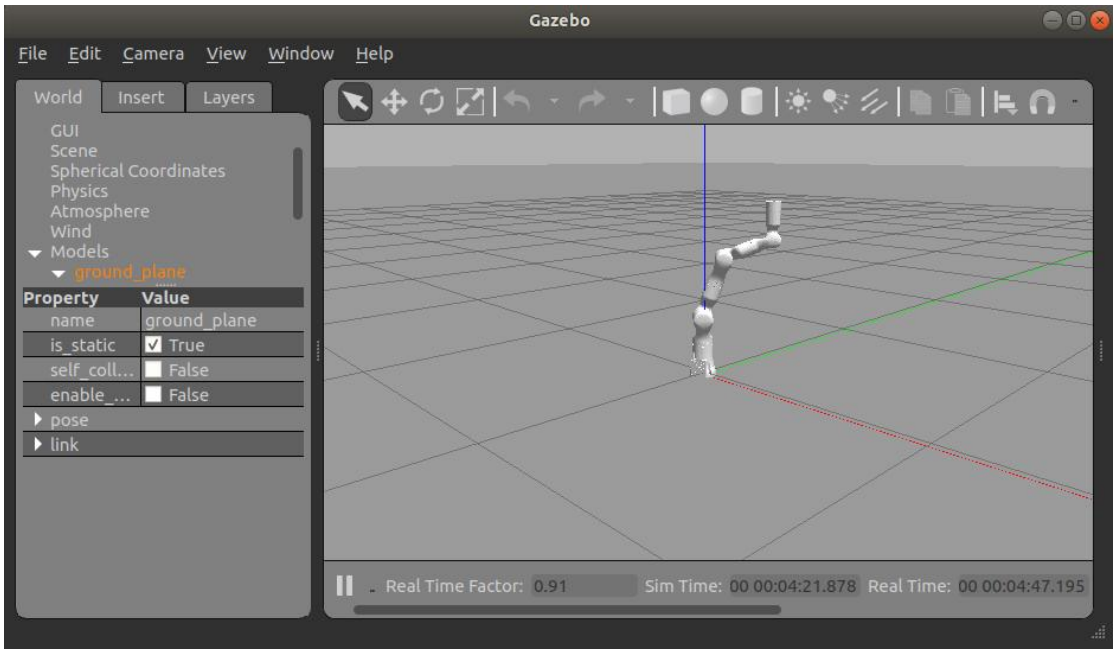

图 6-8 Gazebo 中机器人按 rviz 规划同步执行效果

7.使用 MoveIt!控制真实机械臂

7.1 概述

MoveIt!完成运动规划后的输出接口是一个命名为“FollowJointTrajectory”的 action，其中包含了一系列规划好的路径点轨迹，与使用 MoveIt!控制 Gazebo 中的机械臂不同的是，虚拟机械臂有 gazebo 的 ros_control 插件自动帮我们获取了 follow_joint_trajectory 的动作 action 信息，而现在到真实机器人，需要我们自己编程添加一个 server 订阅这个 action 并处理然后控制真实机器人。

7.1.1 机器人控制器 rm_control

机器人控制器 rm_control 功能包，通过添加一个 server 订阅 MoveIt!完成运动规划后输出的 action 信息，然后将 Moveit 规划的机械臂轨迹，通过三次样条插值细分，按照 20ms 的控制周期发给 rm_driver 节点。具体实现见源码。

7.1.2 机器人驱动功能包 rm_driver

rm_driver 功能包，实现与机械臂通过以太网口建立 socket 连接，订阅和发布机器人的各 topic 信息，将 rm_control 处理后发来的机械臂运行路径点轨迹通过 socket 发送至真实机械臂实现对真实机械臂的控制，同时接收机械臂返回的信息处理后通过 topic 发布至 move_group 完成 rviz 中机器人的同步。具体实现见源码。

关于 rm_driver 和 rm_control 节点订阅发布的信息见表 7-1：

表 7-1 rm_driver 和 rm_control 节点订阅发布 topic 信息

节点名称	分类	Topic 名称	使用 msg 类型	功能
rm_driver（驱动机器人节点）	subscribe	/rm_driver/MoveJ_Cmd	rm_msgs::MoveJ	获取 MoveJ 规划指令，并下发给机械臂
		/rm_driver/MoveJ_P_Cmd	rm_msgs::MoveJ_P	获取 MoveJ_P 规划指令，并下发给机械臂
		/rm_driver/MoveL_Cmd	rm_msgs::MoveL	获取 MoveL 规划指令，并下发给机械臂
		/rm_driver/MoveC_Cmd	rm_msgs::MoveC	获取 MoveC 规划指令，并下发给机械臂

		/rm_driver/ChangeToolName_Cmd	rm_msgs::ChangeTool_Name	获取切换工具坐标系指令，并下发给机械臂
		/rm_driver/ChangeWorkFrame_Cmd	rm_msgs::ChangeWorkFrame_Name	获取切换工作坐标系指令，并下发给机械臂
		/rm_driver/GetArmState_Cmd	rm_msgs::Arm_Current_State	获取反馈机械臂状态指令，并下发给机械臂
		/rm_driver/JointPos	rm_msgs::JointPos	获取角度透传指令，该指令控制器不规划，直接下发给关节运行，该指令与当前角度不可超过 10 度
		/rm_driver/Arm_Digital_Output	rm_msgs::Arm_Digital_Output	获取对机械臂控制器数字 IO 输出控制，并下发给机械臂
		/rm_driver/Arm_Analogl_Output	rm_msgs::Arm_Analog_Output	获取对机械臂控制器模拟 IO 输出控制，并下发给机械臂
		/rm_driver/Tool_Digital_Output	rm_msgs::Tool_Digital_Output	获取对机械臂末端数字 IO 输出控制，并下发给机械臂
		/rm_driver/Tool_Analogl_Output	rm_msgs::Tool_Analog_Output	获取对机械臂末端模拟 IO 输出控制，并下发给机械臂
		/rm_driver/IO_Update	rm_msgs::IO_Update	获取 IO 输入更新指令，并下发给机械臂
		/rm_driver/Gripper_Pick	rm_msgs::Gripper_Pick	获取对手爪力控夹取的指令，并下发机械臂
		/rm_driver/Gripper_Set	rm_msgs::Gripper_Set	获取对手爪开口度设置的指令，并下发机械臂
		/rm_driver/Emergency_Stop	rm_msgs::Stop	获取急停指令，下发给机械臂
		/rm_driver/StartMultiDragTeach_Cmd	rm_msgs::Start_Multi_Drag_Teach	获取复合模式拖动示教指令，并下发给机械臂
		/rm_driver/SetForcePosition_Cmd	rm_msgs::Set_Force_Position	获取力位混合控制指令，并下发给机械臂
		/rm_driver/StopForcePostion_Cmd	std_msgs::Empty	获取关闭力位混合控制指令，并下发给机械臂

		/rm_driver/StartForcePositionMove_Cmd	std_msgs::Empty	开启底层力位混合控制模块补偿模式。在下发透传轨迹前必须下发该指令开启功能
		/rm_driver/ForcePositionMovePose_Cmd	rm_msgs::Force_Position_Move_Pose	下发目标位姿，使用机械臂底层力位混合控制模块通过一维力传感器或者六维力传感器实现力位补偿。
		/rm_driver/ForcePositionMoveJoint_Cmd	rm_msgs::Force_Position_Move_Joint	下发目标角度，使用机械臂底层力位混合控制模块通过一维力传感器或者六维力传感器实现力位补偿。
		/rm_driver/StopForcePositionMove_Cmd	std_msgs::Empty	关闭底层力位混合控制模块补偿模式。在完成透传轨迹后必须下发该指令关闭功能
		/rm_driver/GetSixForce_Cmd	std_msgs::Empty	获取六维力的控制指令，并下发机械臂
		/rm_driver/ClearForceData_Cmd	std_msgs::Empty	清空六维力的控制指令，并下发机械臂
		/rm_driver/SetForceSensor_Cmd	std_msgs::Empty	自动设置六维力重心参数的控制指令，并下发机械臂
		/rm_driver/ManualSetForcePose_Cmd	rm_msgs::Manual_Set_Force_Pose	手动标定六维力数据的控制指令，并下发机械臂
		/rm_driver/StopSetForceSensor_Cmd	std_msgs::Empty	停止标定力传感器重心的控制指令，并下发机械臂
	publish	/joint_states	sensor_msgs::JointState	机械臂关节的实际角度
		/rm_driver/Arm_IO_State	rm_msgs::Arm_IO_State	机械臂控制器 IO 输入状态
		/rm_driver/Tool_IO_State	rm_msgs::Tool_IO_State	机械臂末端 IO 输入状态
		/rm_driver/Plan_State	rm_msgs::Plan_State	机械臂轨迹规划状态
		/rm_driver/GetSixForce	rm_msgs::Six_Force	返回当前机械臂的六维力
		/rm_driver/Force_Position_State	rm_msgs::Force_Position_State	返回当前各关节角度和所使用力控方式的力或力矩

		/rm_driver/StartMultiDragTeach_result	std_msgs::Bool	返回开启复合模式拖动示教的执行结果
		/rm_driver/SetForcePosition_result	std_msgs::Bool	返回设置力位混合控制的执行结果
		/rm_driver/StopForcePostion_result	std_msgs::Bool	返回结束力位混合控制的执行结果
		/rm_driver/ClearForceData_result	std_msgs::Bool	返回六维力数据清零的执行结果
		/rm_driver/ForceSensorSet_result	std_msgs::Bool	返回自动设置六维力重心参数、手动标定六维力数据的执行结果
		/rm_driver/StopSetForceSensor_result	std_msgs::Bool	返回停止标定力传感器重心的执行结果
		/rm_driver/StartForcePositionMove_result	std_msgs::Bool	返回开启透传力位混合控制补偿模式的执行结果
		/rm_driver/StopForcePositionMove_result	std_msgs::Bool	设置透传力位混合补偿失败后的返回
		/rm_driver/Force_Position_Move_result	std_msgs::Bool	返回关闭透传力位混合控制补偿模式的执行结果
rm_control（机器人控制器）	publish	/rm_driver/JointPos	rm_msgs::JointPos	将 moveit 轨迹规划点，通过三次样条插值细分，细分周期 10ms，然后按照该周期将插值后的轨迹点发送给 rm_driver，进而直接控制机械臂运动。

7.1.3 rm_msgs 功能包

rm_msgs 功能包定义了 RM65-B 机器人所用到的所有控制消息和状态消息，rm_control 和 rm_driver 都需要使用。

具体消息类型见表 7-2：

表 7-2 rm_msgs 功能包定义的消息类型

消息类型	功能	组成	类型	定义
Arm_Analog_Output	控制器模拟 IO 输出	num	uint8	模拟 IO 输出通道号，1~4
		voltage	float32	输出电压，0~10V
Arm_Digital_Output	控制器数字 IO 输出	num	uint8	数字 IO 输出通道号，1~4
		state	bool	输出状态，false-低电平，true-高电平
Arm_IO_State	控制器输入 IO 状态	Arm_Digital_Input	bool[3]	数字 IO 输入状态，通道 1~3
		Arm_Analog_Input	float32[4]	模拟 IO 输入状态，通道 1~4
Gripper_Pick	手爪力控夹取	speed	uint16	开合速度，范围：1~1000
		force	uint16	夹持力，范围：1~1000
JointPos	关节角度	joint	float32[6]	6 个关节角度
MoveC	圆弧规划指令	Mid_Pose	geometry_msgs/Pose	圆弧规划中间点
		End_Pose	geometry_msgs/Pose	圆弧规划终点
		speed	float32	速度系数，0~1
MoveJ	关节空间规划指令	joint	float32[6]	目标位置关节弧度
		speed	float32	速度系数，0~1
MoveJ_P	关节空间规划指令	Pose	geometry_msgs/Pose	目标位姿
		speed	float32	速度系数，0~1
MoveL	直线规划指令	Pose	geometry_msgs/Pose	目标位姿
		speed	float32	速度系数，0~1
Plan_State	轨迹规划返回状态	state	bool	false-规划失败，true-到达目标点
Stop	机械臂急停指令	state	bool	true-机械臂急停
Tool_Analog_Output	机械臂工具端模拟 IO 输出指令	voltage	float32	模拟 IO 输出电压，范围：0~10V
Tool_Digital_Output	机械臂工具端数字 IO 输出指令	num	uint8	数字 IO 输出通道号，1~2
		state	bool	输出状态，false-低电平，true-高电平
Tool_IO_State	机械臂工具端 IO 输	Tool_Digital_Input	bool[2]	数字 IO 输入状态，通道 1~2

	入状态	Tool_Analog_Input	float32	模拟 IO 输入状态
Gripper_Set	设置手爪开口范围	position	uint16	目标位置，范围：1~1000，代表位置：0~70mm
Joint_Enable	关节使能控制	joint_num	uint8	关节序号，范围：1~6
		state	bool	true-上使能 false-掉使能
IO_Update	查询机械臂 IO 输入状态	type	uint8	0x01-控制器所有 IO 输入状态，0x02-末端工具所有 IO 输入 下发该指令后，通过 Arm_IO_State/Tool_IO_State 获取状态
ChangeTool_Name	切换机械臂工具坐标系	toolname	string	发送当前已建立的工具坐标系名称，若该工具坐标系不存在，则切换失败
ChangeWorkFrame_Name	切换机械臂工作坐标系	WorkFrame_name	string	发送当前已建立的工作坐标系名称，若该工作坐标系不存在，则切换失败
Arm_Current_State	获取机械臂当前状态	joint	float32[6]	当前位置关节弧度
		Pose	float32[6]	当前位置机械臂位姿
		arm_ree	uint16	机械臂错误代码
		sys_err	uint16	控制器错误代码
Start_Multi_Drag_Teach	开始复合模式拖动示教	mode	uint8	mode:拖动示教模式。0-电流环模式，1-使用末端六维力，只动位置，2-使用末端六维力，只 动姿态，3-使用末端六维力，位置和姿态同时动。
Set_Force_Position	机械臂力位混合控制参数	sensor	uint8	Sensor:传感器，0-NULL,1-一维力，2-六维力 Mode：1-基坐标系 Z 轴力控；
		mode	uint8	

		N	int16	2-作用面法线力控；3-作用面径向力控 N:力的大小 load：坐标系选项 0-基坐标系，1-工具坐标系
		load	uint8	
Force_Position_State	机械臂力位混合设置成功后的返回参数	joint	float32[6]	当前各关节角度和所使用力控方式的力或力矩
		force	float32	
Six_Force	查询机械臂六维力的返回参数	force	float32[6]	机械臂六维力数值
Manual_Set_Force_Pose	手动标定六维力的参数	joint	int16[6]	机械臂各关节角度

7.2 运行演示

在运行前，先将 RM65-B 机器人上电，等待 30 秒让机器人完成初始化；

确保上位机 IP 与机械臂在同一局域网内，然后可以在终端执行：

```
ping 192.168.1.18
```

如果能 ping 通则可以继续以下操作；

修改 rm_65_moveit_config/launch/rm_65_moveit_controller_manager.launch.xml，配置加载 controllers.yaml 文件到参数服务器，代码如下：

```
<launch>

<arg name="execution_type" default="FollowJointTrajectory" />


<!-- loads moveit_controller_manager on the parameter server which is taken as ar

gument

if no argument is passed, moveit_simple_controller_manager will be set -->

<arg name="moveit_controller_manager" default="moveit_simple_controller_manager/
```

```
MoveItSimpleControllerManager" />

<param name="moveit_controller_manager" value="$(arg moveit_controller_manager)

"/>



<!-- load controller_list -->

<arg name="use_controller_manager" default="true" />

<param name="use_controller_manager" value="$(arg use_controller_manager)" />



<!-- loads ros_controllers to the param server -->

<!-- <rosparam file="$(find rm_65_moveit_config)/config/ros_controllers.yaml"/> -->

<rosparam file="$(find rm_65_moveit_config)/config/controllers.yaml"/>

<!-- <rosparam file="$(find rm_65_moveit_config)/config/controllers_gazebo.yaml"/> -

->

</launch>
```

打开终端，执行以下命令先运行 rm_control 节点：

```
cd ~/ws_rmrobot/

source devel/setup.bash

roslaunch rm_control rm_control.launch
```

然后再打开一个新的终端，执行以下命令启动 rm_driver 节点和加载 MoveIt! 的 rviz：

```
cd ~/ws_rmrobot/

source devel/setup.bash
```

```
roslaunch rm_bringup rm_robot.launch
```

运行成功后在 rviz 中可以看到机器人模型与真实机械臂的状态保持一致，如图 7-1 和 7-2 所示：

图 7-1 真实机械臂状态

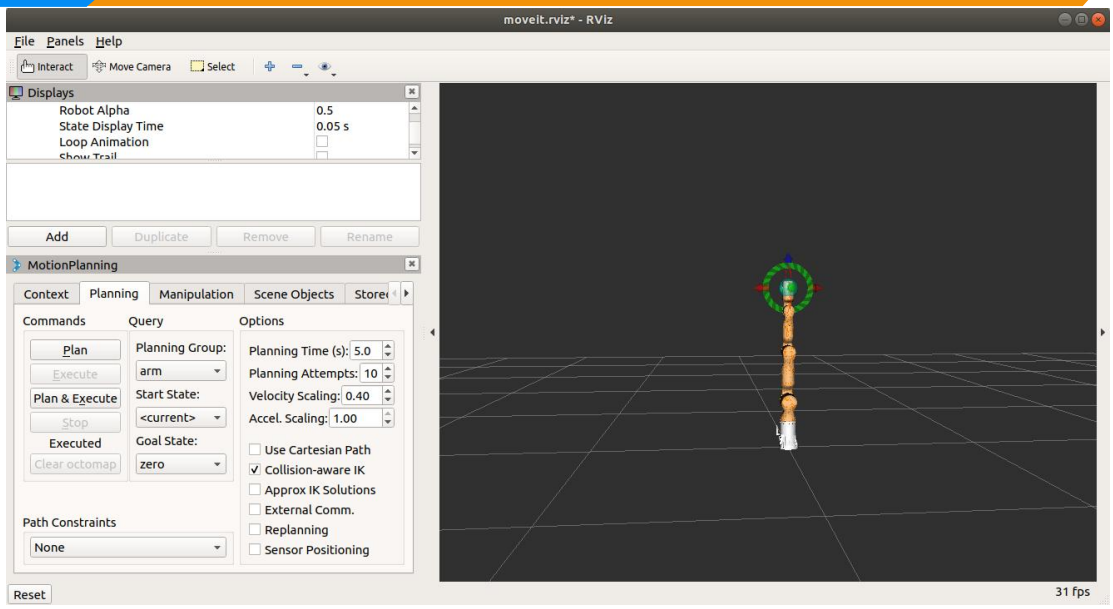

图 7-2 rviz 中机械臂模型状态与真实机械臂保持一致

在 rviz 中进行拖动规划然后点击“Plan & Execute”按钮，可以看到真实机械臂会按照 rviz 中 MoveIt!规划的路径运动，如图 7-3 和 7-4 所示。

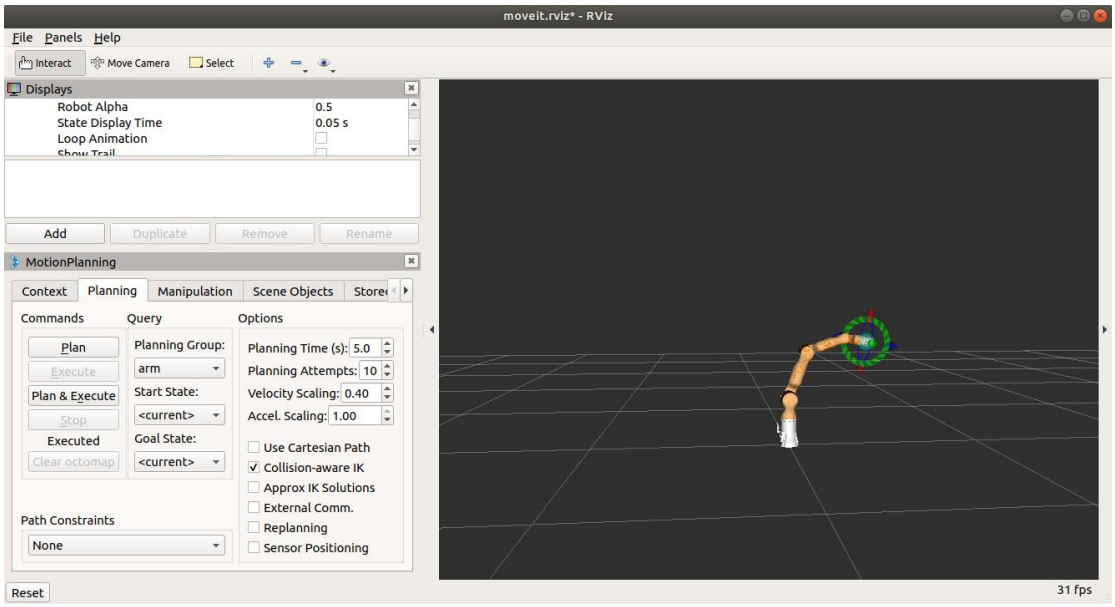

图 7-3 rviz 中机械臂模型规划路径运行到指定位置

图 7-4 真实机械臂按照 rviz 中 MoveIt!规划的路径运动到相同位置

8. MoveIt!编程示例---场景规划

8.1 概述

在实际应用中，MoveIt! GUI 提供的功能有限，很多实现需要在代码中完成。MoveIt!的 move_group 也提供了丰富的 C++和 Python 的编程 API，可以帮助我们完成更多运动控制的相关功能。

8.2 实现功能

1）在场景中添加和移除物体

2）为机器人附加和拆卸物体

该程序源代码：rm_65_demo/src/planning_scene_ros_api_demo.cpp，具体实现见注释。

8.3 运行演示

首先打开终端，执行以下命令运行 MoveIt!演示 demo：

```
cd ~/ws_rmrobot/

source devel/setup.bash

roslaunch rm_65_moveit_config demo.launch
```

rviz 启动后如图 8-1 所示：

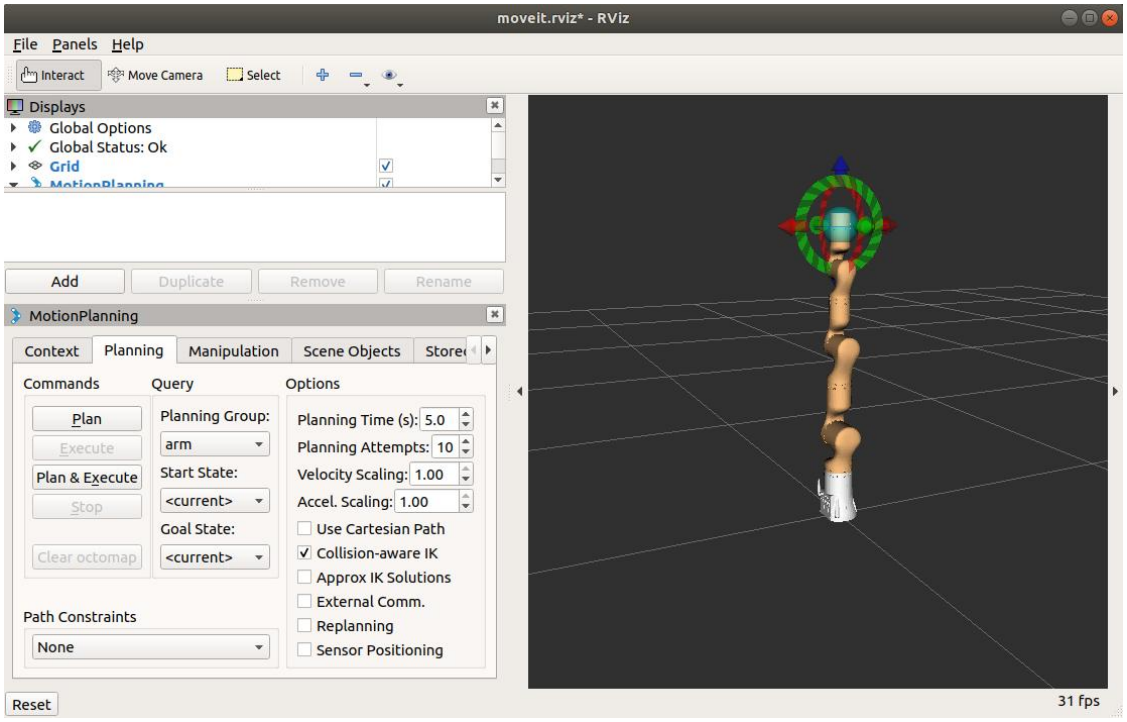

图 8-1 MoveIt!演示 demo 界面

因为本示例程序使用 MoveItVisualTools 插件控制程序运行，所以需要在 rviz 中添加 RvizVisualToolsGui 插件，添加操作如图 8-2~8-4 所示：

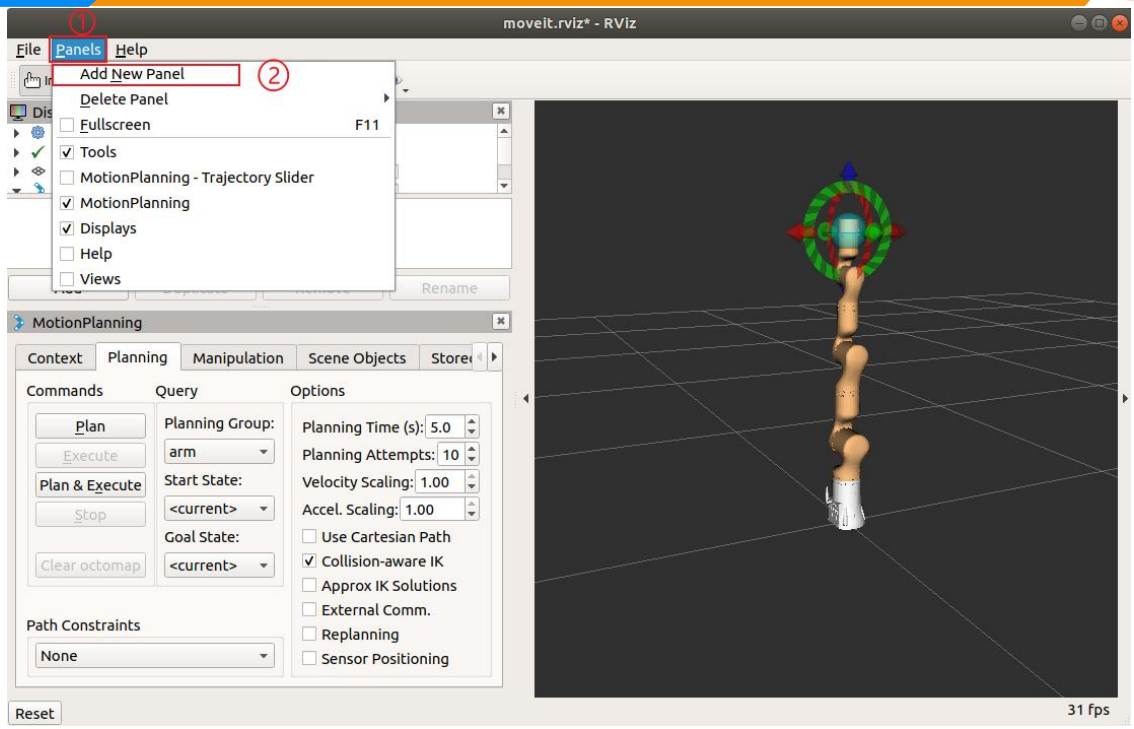

图 8-2 rviz 中 Add New Panel

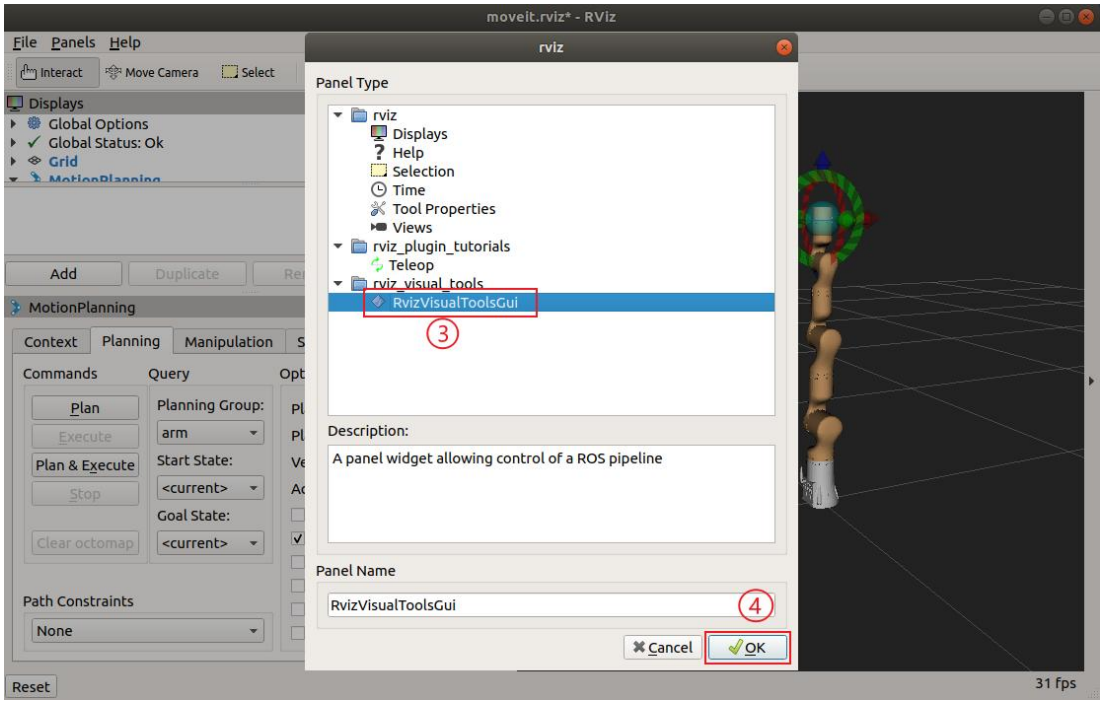

图 8-3 rviz 中添加 RvizVisualToolsGui

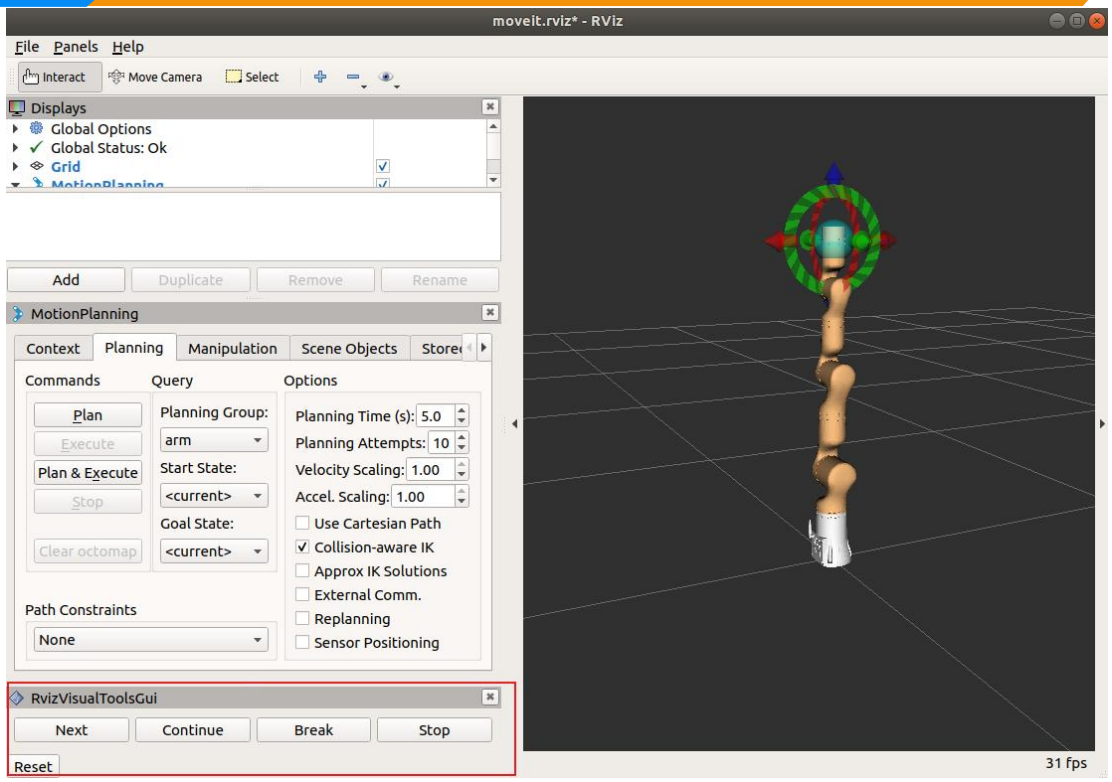

图 8-4 rviz 中添加 RvizVisualToolsGui 完成

打开一个新的终端，执行以下命令启动场景规划节点：

```
cd ~/ws_rmrobot/

source devel/setup.bash

roslaunch rm_65_demo planning_scene_ros_api_demo.launch
```

场景规划节点启动完成后提示点击 rviz 中 RvizVisualToolsGui 面板中的 Next 按钮开始运行程序，如图 8-5 所示：

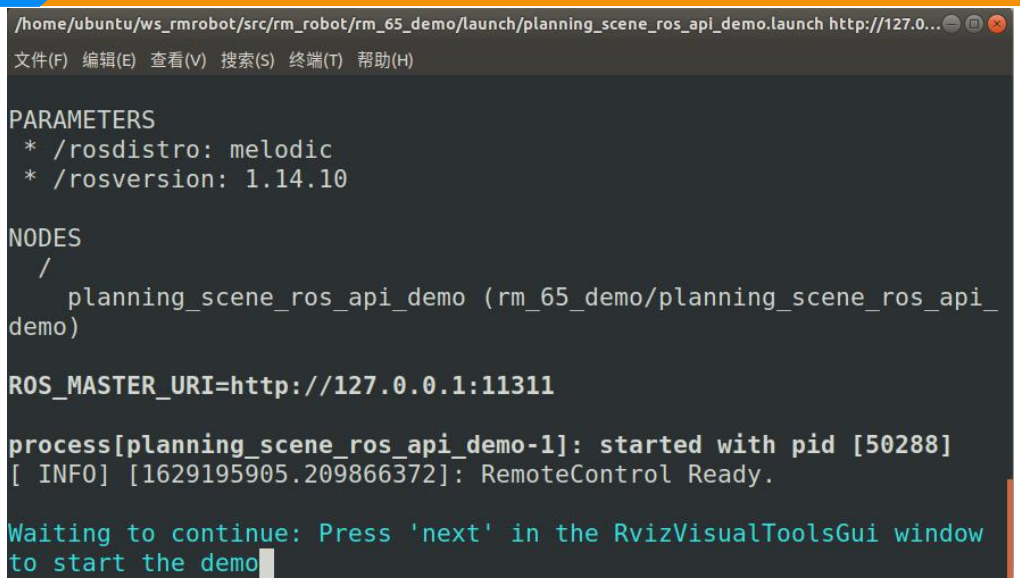

图 8-5 场景规划节点启动完成等待交互运行

在 rviz 中点击 RvizVisualToolsGui 面板中的 Next 按钮可以看到，在机器人的末端位置添加了一个绿色物体，说明在场景中添加物体成功，如图 8-6 所示：

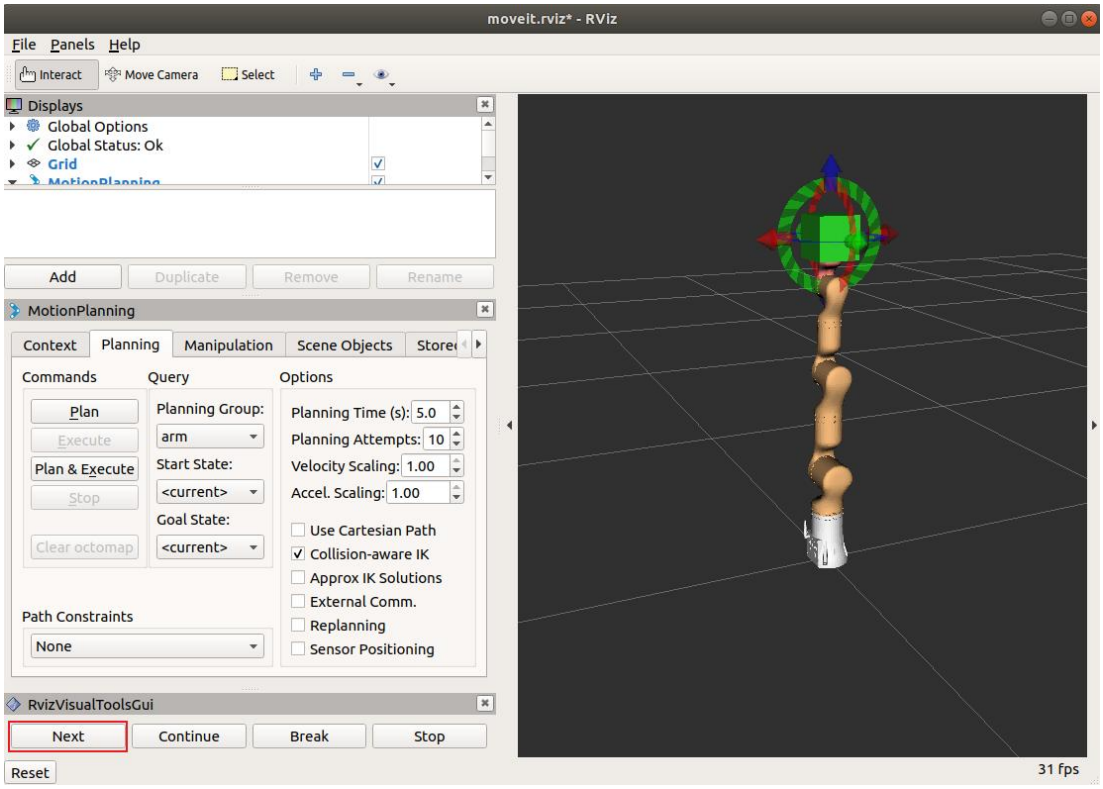

图 8-6 场景添加物体

再次点击 Next 按钮，可以看到物体改变了颜色，此时代表该物体被附加到了机器人上，在进行运动规划时会被视为是机器人的一部分检测碰撞，如图 8-7

所示：

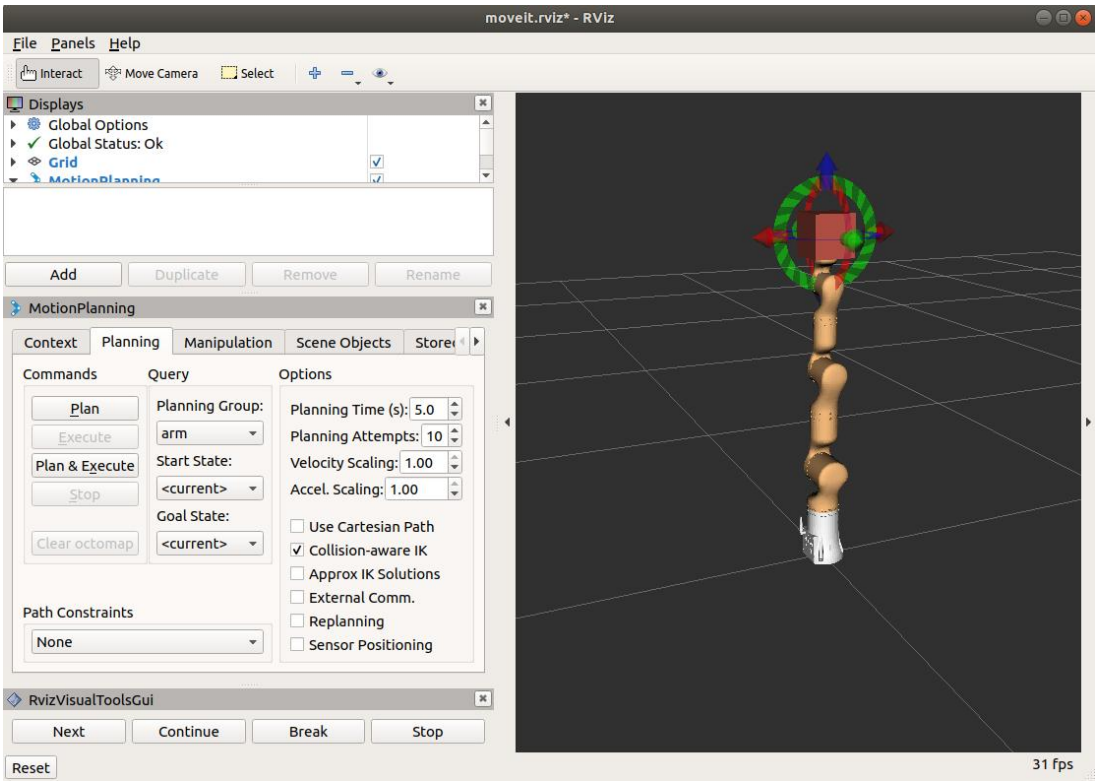

图 8-6 物体被附加到机器人上

再次点击 Next 按钮，可以看到物体又变为了绿色，表示附着的物体从机器人上被解除附着，如图 8-7 所示：

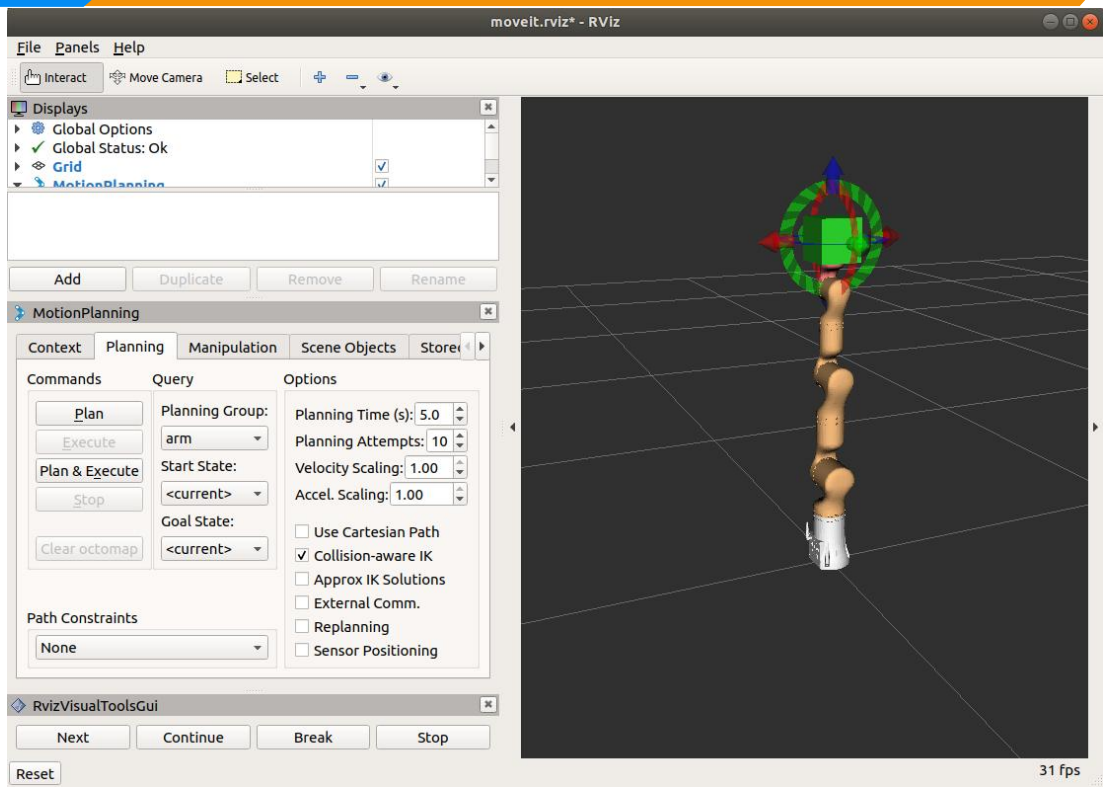

图 8-7 附着物体被解除附着

再次点击 Next 按钮，可以看到物体消失了，说明物体从场景中被移除了，如图 8-8 所示：

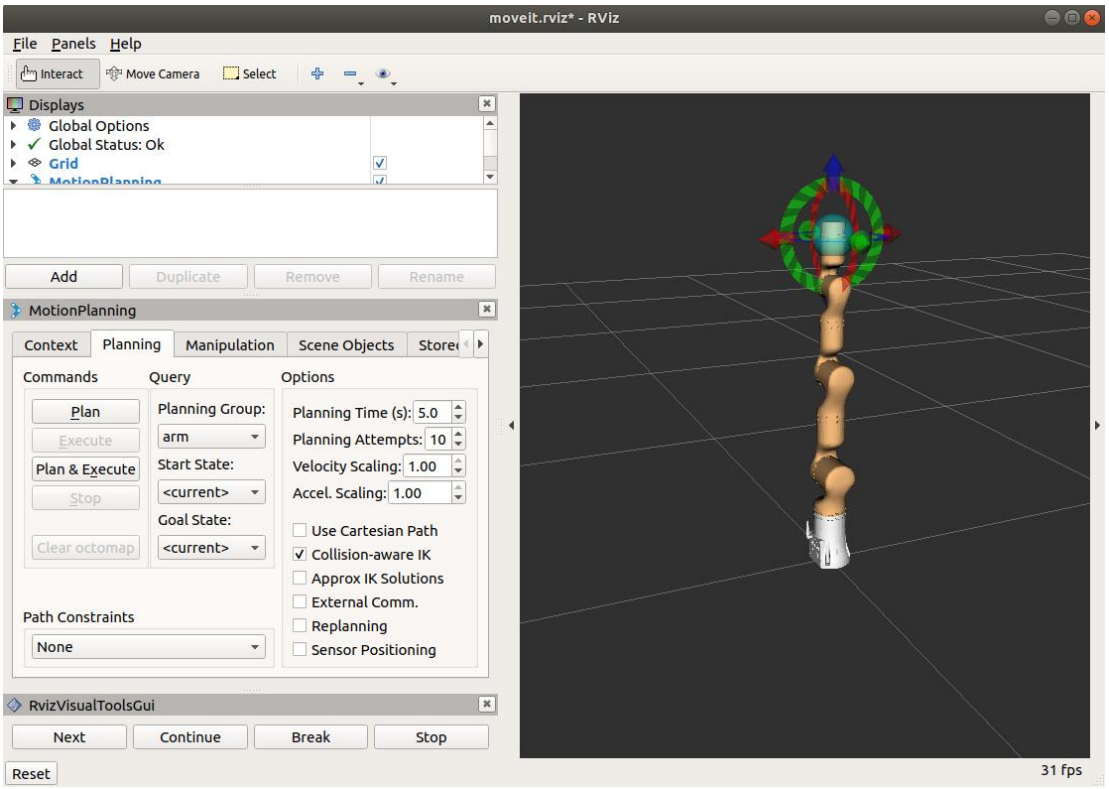

图 8-8 物体从场景中移除

再次点击 Next 按钮，场景规划节点运行结束并退出，如图 8-9 所示：

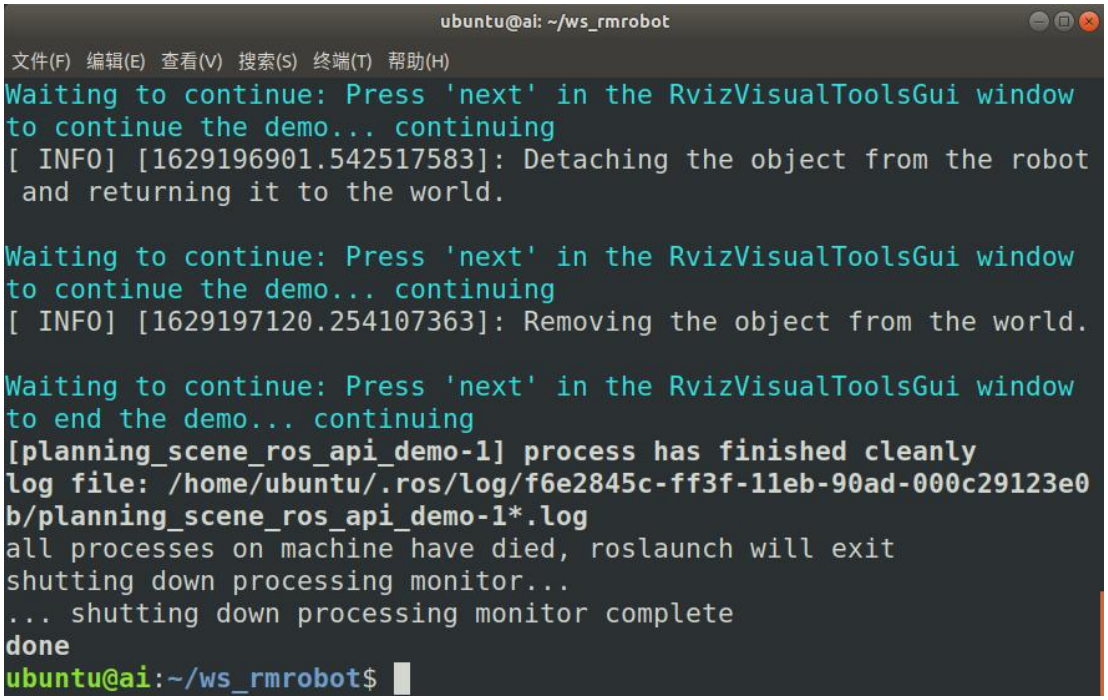

图 8-9 场景规划节点运行结束退出

9. MoveIt!编程示例---避障规划

9.1 概述

本示例源码为 Python 代码，调用了 MoveIt!的 move_group 提供的 Python API 实现，源代码位置：rm_65_demo/scripts/moveit_obstacles_demo.py，具体实现见代码注释。

9.2 实现功能

首先让机器人运动到已经配置好的“zero”位姿，然后在场景中添加一个 table 物体表示障碍物，再然后让机器人自动避开障碍物运动到“forward”位姿，最后再让机器人自动避开障碍物回到“zero”位姿。

9.3 运行演示

首先打开终端，执行以下命令运行 MoveIt!演示 demo：

```
cd ~/ws_rmrobot/

source devel/setup.bash

roslaunch rm_65_moveit_config demo.launch
```

rviz 启动后如图 9-1 所示：

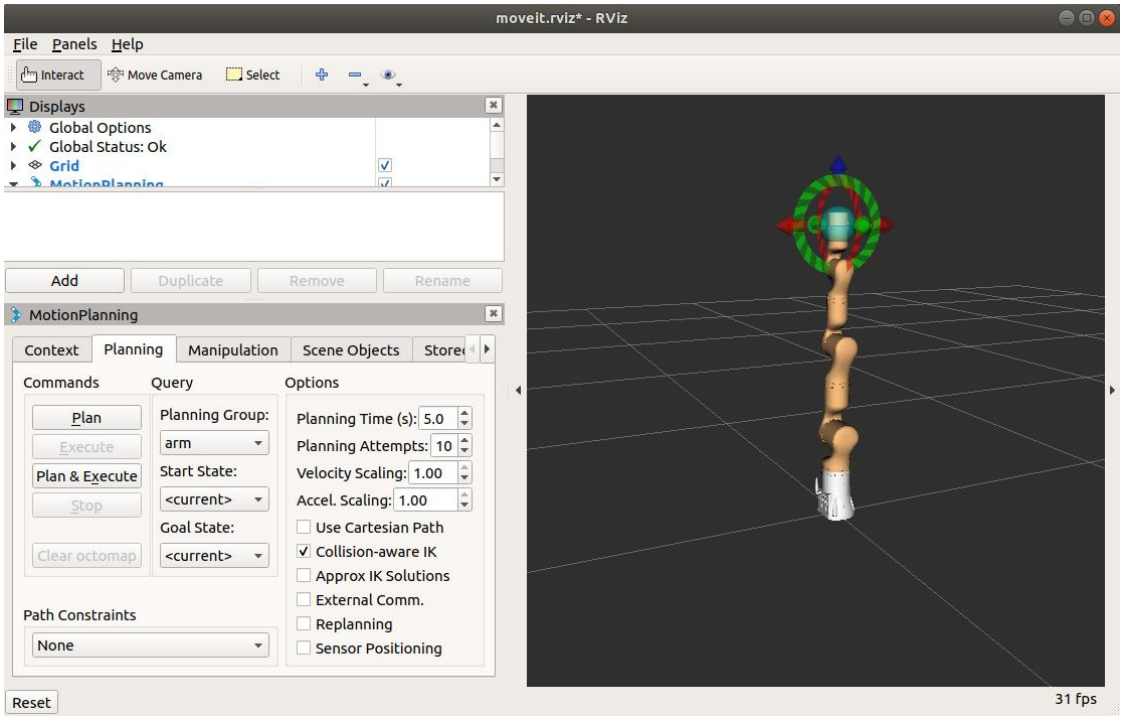

图 9-1 MoveIt!演示 demo 界面

再打开一个新的终端，执行以下命令启动避障规划节点：

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_65_demo moveit_obstacles_demo.py
```

节点运行后，在 rviz 中可以看到场景中添加了一个 table 物体，然后机器人自动避开 table 运行到 forward 位姿，最后从 forward 位姿自动避开 table 回到 zero 位姿，如图 9-2 所示：

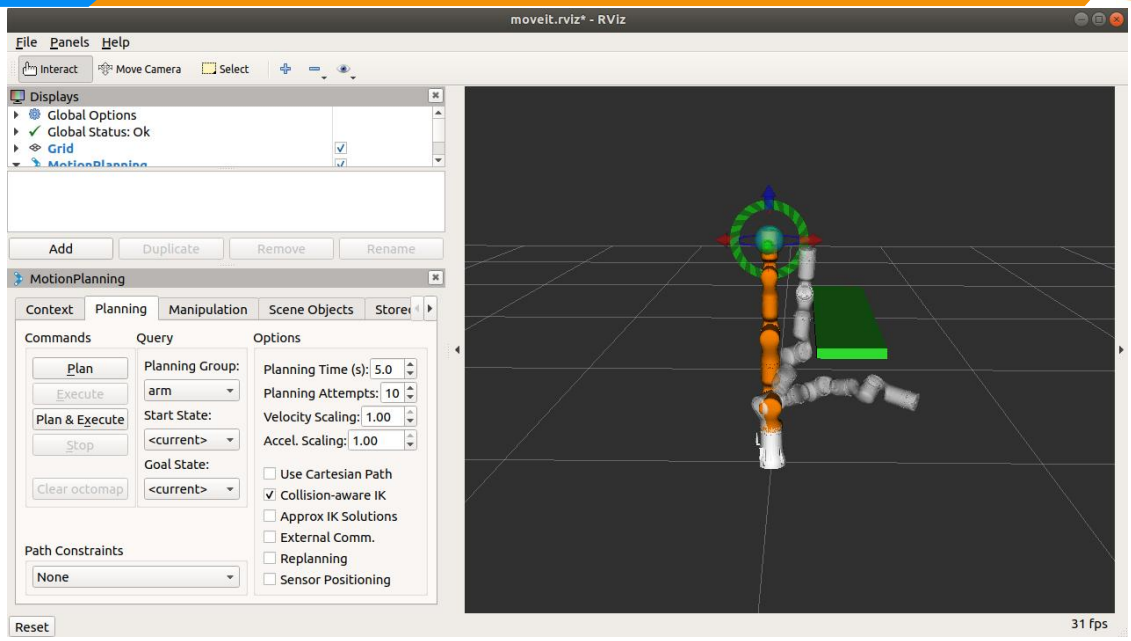

图 9-2 机器人自主避障效果

10.MoveIt!编程示例---pick and place

10.1 概述

本示例源码为 C++代码，调用了 MoveIt!的 move_group 提供的 C++ API 实现，源代码位置：rm_65_demo/src/pick_place_demo.cpp，具体实现见代码注释。

10.2 实现功能

模拟机械臂自动抓取物体然后进行运动规划将物体放置到指定位置。

10.3 运行演示

首先打开终端，执行以下命令运行 MoveIt!演示 demo：

```
cd ~/ws_rmrobot/

source devel/setup.bash

roslaunch rm_65_moveit_config demo.launch
```

rviz 启动后如图 10-1 所示：

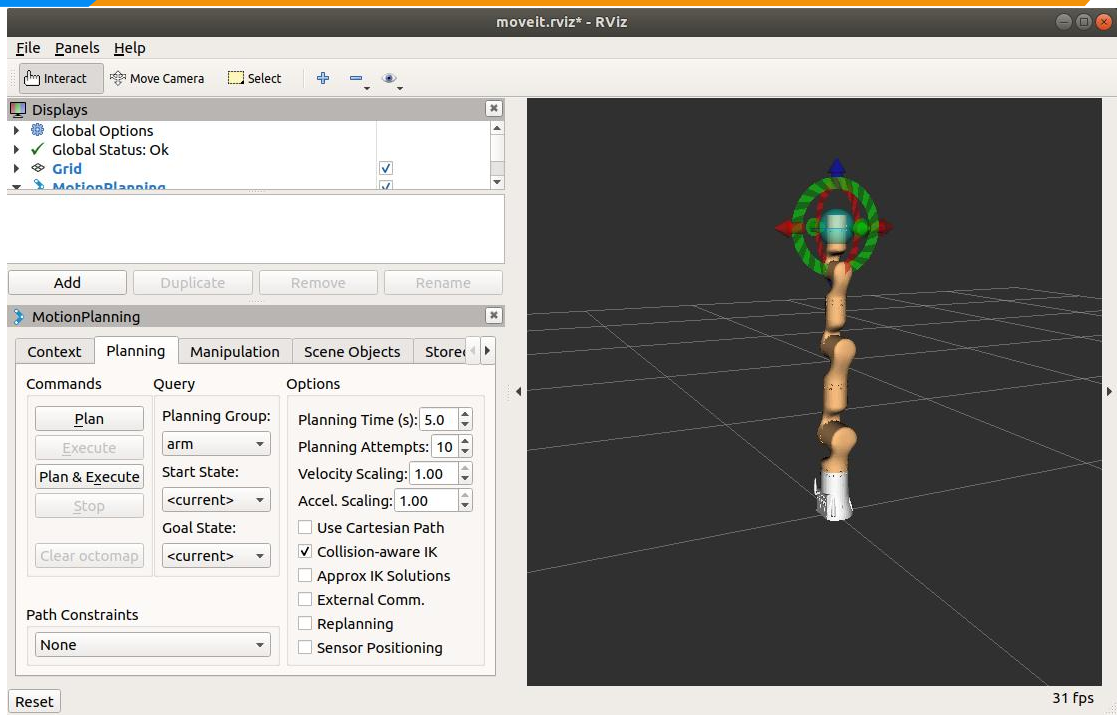

图 10-1 MoveIt!演示 demo 界面

再打开一个新的终端，执行以下命令运行 pick_place_demo 节点：

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_65_demo pick_place_demo
```

节点运行后，在 rviz 中可以看到场景中添加了三个物体，分别代表两个桌子和一个抓取的目标物，然后机器人运动到目标物的位置将目标物附着到机器人上模拟抓取物体，接着进行运动规划将目标物体放置到另一个桌子上然后解除附着，最后机器人返回 zero 姿态，如图 10-2 所示：

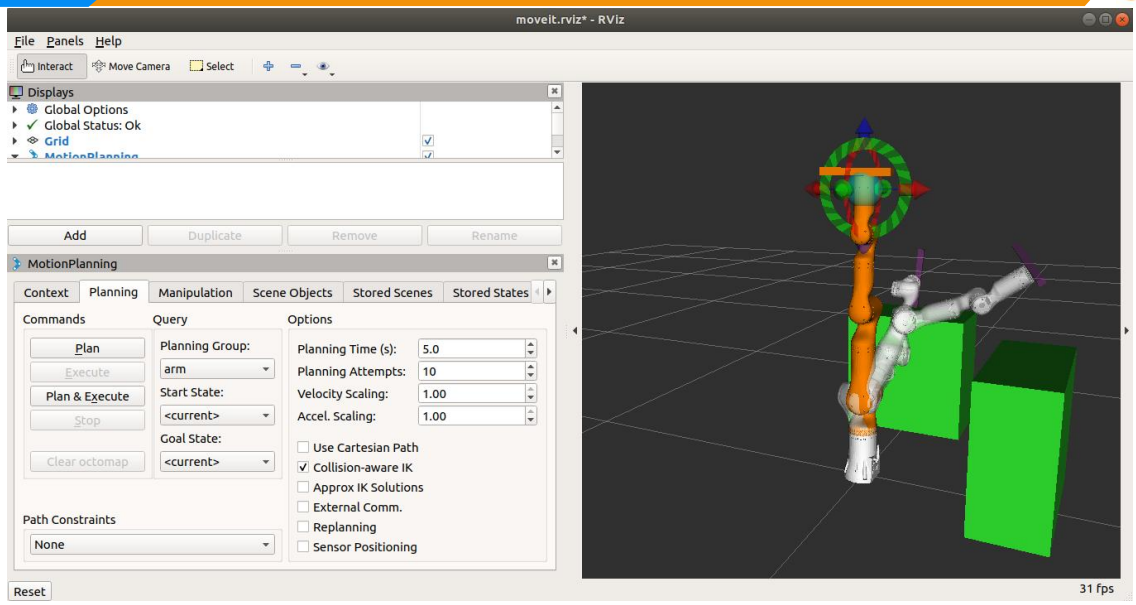

图 10-2 pick and place 运行效果

11.指令驱动机器人编程示例---MoveJ 指令

11.1 概述

ROS 节点 api_moveJ_demo 通过话题/rm_driver/MoveJ_Cmd 将 MoveJ 指令发送给机械臂。机械臂接收 MoveJ 指令后通过内部集成算法完成路径规划并控制机械臂运动。本示例源码为 C++代码，具体实现见代码注释。源代码位置：rm_65_demo/src/api_moveJ_demo.cpp。

11.2 实现功能

自定义 ROS 节点 api_moveJ_demo，该节点发布/rm_driver/MoveJ_Cmd 为话题的 rm_msgs::MoveJ 类型消息（消息内容详见表 7-2），该消息包含机械臂各个关节的角度信息以及运行速度信息。机器人驱动节点 rm_driver 通过订阅话题/rm_driver/MoveJ_Cmd，获取 rm_msgs::MoveJ 内的信息。rm_driver 解析 rm_msgs::MoveJ 消息内容后，将该内容通过 JSON 协议封装并通过 Socket 发给机械臂。机械臂接收到 MoveJ 指令以及机械臂关节角度后，通过机械臂内部集成路径规划算法规划路径，并控制各关节转动相应角度。

11.3 运行演示

首先打开终端，执行以下命令运行 roscore：

```
rocore
```

然后打开一个新终端，执行以下命令运行机器人驱动节点 rm_driver：

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_driver rm_driver
```

节点 rm_driver 启动后，如图 11-1 所示。

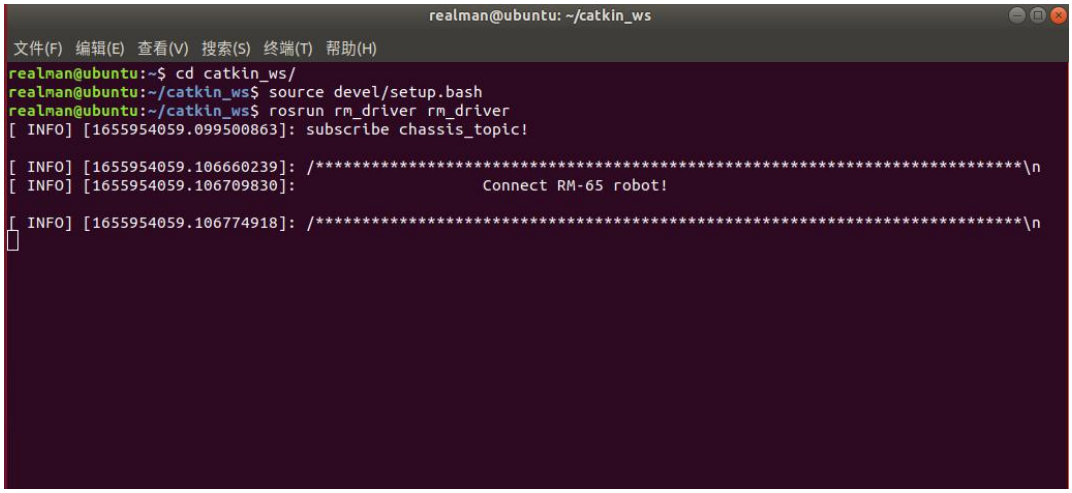

图 11-1 节点 rm_driver 启动成功界面

再打开一个新终端执行以下命令，运行 api_moveJ_demo 节点，控制机械臂各关节转动指定角度。

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_65_demo api_moveJ_demo
```

运行成功后，机械臂各个关节转动指定角度，由初始位姿运动到指定位姿。如图 11-2 与图 11-3 所示。

图 11-2 机械臂初始位姿

图 11-3 执行 MoveJ 指令后的机械臂位姿

12.指令驱动机器人编程示例---MoveJ_P 指令

12.1 概述

ROS 节点 api_moveJ_demo 通过话题/rm_driver/MoveJ_P_Cmd 将 MoveJ_P 指令发送给机械臂。机械臂接收 MoveJ_P 指令后通过内部集成算法完成路径规划并控制机械臂运动。本示例源码为 C++代码，具体实现见代码注释。源代码位置：rm_65_demo/src/api_moveJ_P_demo.cpp。

12.2 实现功能

自定义 ROS 节点 api_moveJ_P_demo，该节点发布/rm_driver/MoveJ_P_Cmd 为话题的 rm_msgs::MoveJ_P 类型消息（消息内容详见表 7-2），该消息包含机械臂末端的目标位姿以及运行速度信息。机器人驱动节点 rm_driver 通过订阅话题/rm_driver/MoveJ_P_Cmd，获取 rm_msgs::MoveJ_P 内的信息。rm_driver 解析 rm_msgs::MoveJ_P 消息内容后，

将该内容通过 JSON 协议封装并通过 Socket 发给机械臂。机械臂接收到 MoveJ_P 指令以及机械臂末端位姿后，通过机械臂内部集成路径规划算法规划路径，并控制机械臂末端到达指定位姿。该指令中的目标位姿必须是机械臂末端法兰中心基于基坐标系的位姿，用户在使用该指令前务必确保，否则目标位姿会出错！与 MoveJ 指令不同的是，MoveJ_P 指令通过机械臂末端位姿控制机械臂运动。

12.3 运行演示

首先打开终端，执行以下命令运行 roscore：

```
rocore
```

然后打开一个新终端，执行以下命令运行机器人驱动节点 rm_driver：

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_driver rm_driver
```

节点 rm_driver 启动后，如图 12-1 所示。

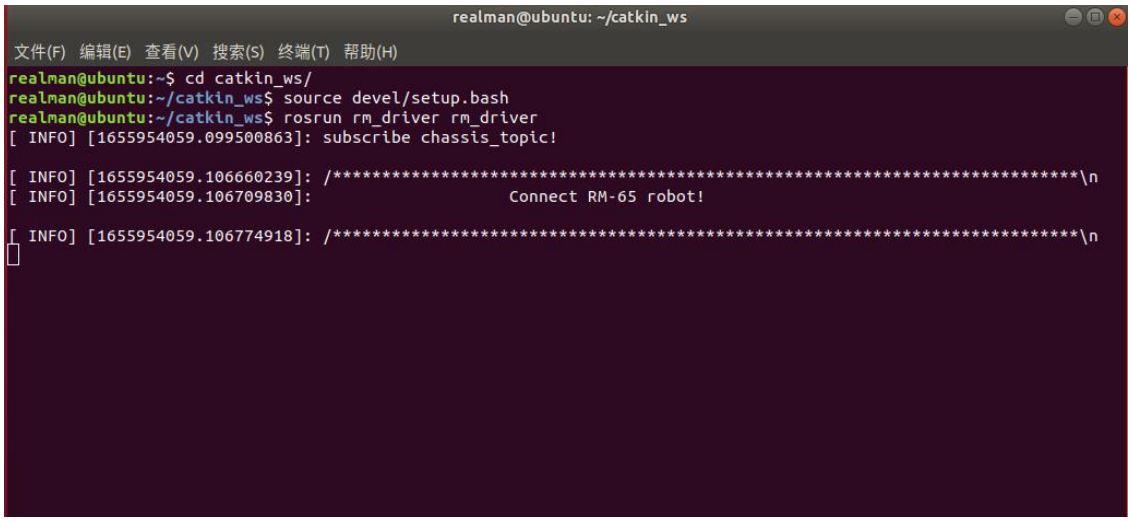

图 12-1 节点 rm_driver 启动成功界面

再打开一个新终端执行以下命令，运行 api_moveJ_P_demo 节点，控制机械臂末端到达指定位姿。

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_65_demo api_moveJ_P_demo
```

运行成功后，机械臂末端由初始位姿运动到指定位姿。如图 12-2 与图 12-3 所示。

图 12-2 机械臂初始位姿

图 12-3 执行 MoveJ_P 指令后的机械臂位姿

13.指令驱动机器人编程示例---MoveL 指令

13.1 概述

ROS 节点 api_moveL_demo 通过话题/rm_driver/MoveL_Cmd 将 MoveL 指令发送给机械臂，机械臂接收 MoveL 指令后通过内部集成算法完成路径规划并控制机械臂末端由初始位姿沿直线运动到指定位姿。本示例源码为 C++代码，具体实现见代码注释。源代码位置：rm_65_demo/src/api_moveL_demo.cpp。

13.2 实现功能

自定义 ROS 节点 api_moveL_demo，该节点发布/rm_driver/MoveL_Cmd 为话题的 rm_msgs::MoveL 类型消息（消息内容详见表 7-2），该消息包含机械臂末端的目标位姿以及运行速度信息。机器人驱动节点 rm_driver 通过订阅话题 /rm_driver/MoveL_Cmd，获取 rm_msgs::MoveL 内的信息，rm_driver 解析 rm_msgs::MoveL 消息内容后，将该内容通过 JSON 协议封装并通过 Socket 发给机械臂。机械臂接收到 MoveL 指令以及机械臂末端位姿后，通过机械臂内部集成路径规划算法规划路径，并控制机械臂末端沿直线到达指定位姿。

13.3 运行演示

首先打开终端，执行以下命令运行 roscore：

```
rocore
```

然后打开一个新终端，执行以下命令运行机器人驱动节点 rm_driver：

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_driver rm_driver
```

节点 rm_driver 启动后，如图 13-1 所示。

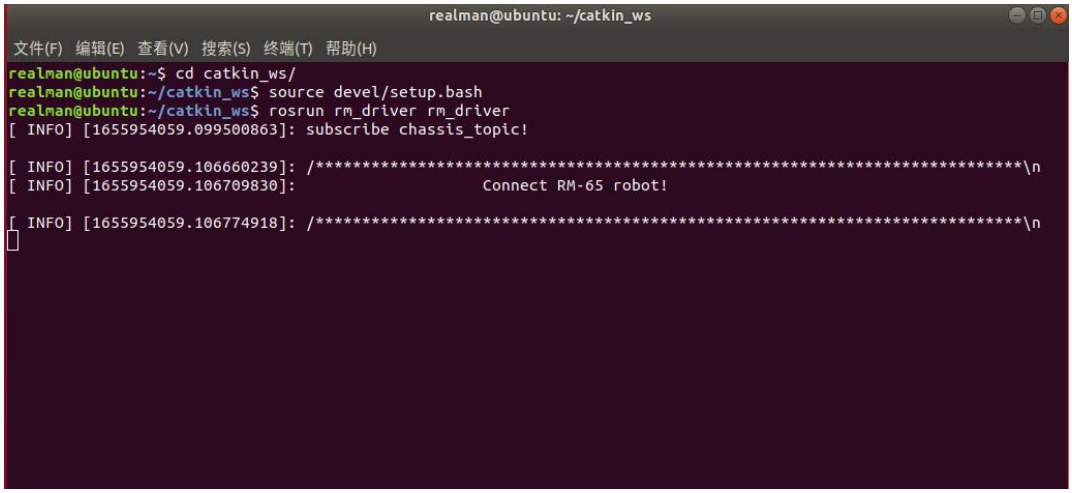

图 13-1 节点 rm_driver 启动成功界面

再打开一个新终端执行以下命令，运行 api_moveL_demo 节点，控制机械臂末端沿直线到达指定位姿。为避免机械臂路径规划过程中出现不可达目标点，该过程中，首先通过 MoveJ_P 控制机械臂由初始位姿运动至过渡位姿（该位姿可保证机械臂执行 MoveL 时机械臂末端位姿可达），然后通过 MoveL 命令控制机械臂末端由过渡位姿沿直线运动至目标位姿。

```
cd ~/ws_rmrobot/

source devel/setup.bash

rosrun rm_65_demo api_moveL_demo
```

运行成功后，机械臂末端由初始位姿沿直线运动到指定位姿。如图 13-2、图 13-3 与图 13-4 所示。

图 13-2 机械臂初始位姿

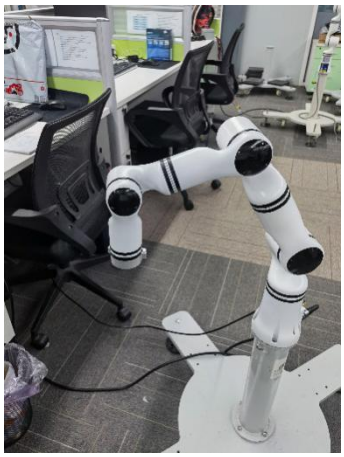

图 13-3 执行 MoveL 指令时机械臂的过渡位姿

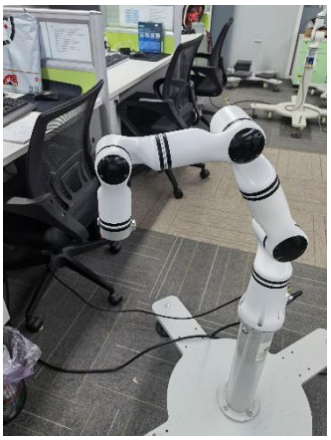

图 13-4 执行完 MoveL 指令后的机械臂位姿