Note1

rviz 加载任意模型

更多介绍可见 笔记：rviz/DisplayTypes/Marker。
在cpp中实现rviz任意模型的加载（支持.dae,.stl,.mesh格式）。

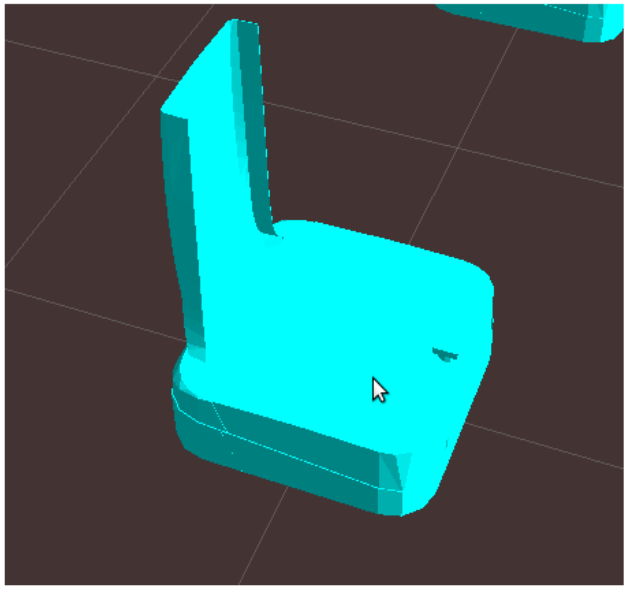
```
// marker publisher
  //http://docs.ros.org/jade/api/visualization_msgs/html/msg/Marker.html
  application.marker_publisher = nh.advertise<visualization_msgs::Marker>(
          application.cfg.MARKER_TOPIC,1);

// publish messages
            marker_publisher.publish(cfg.MARKER_MESSAGE);
```

Marker就是rviz需要显示的信息。
将.obj文件通过3维转化软件转化成.stl，然后放到rospack 可以找到的包内，在代码中通过

```
//set mesh resource
    MARKER_MESSAGE.mesh_resource="package://ur_description/tt.STL";
```

设置模型。
A scale of `(1.0, 1.0, 1.0)` means the mesh will display as the exact size specified in the mesh file.
可以控制模型的缩放。

1Mesh Resource (MESH_RESOURCE=10) [1.1+]



Uses the `mesh_resource` field in the marker. Can be any mesh type supported by rviz (.stl or Ogre .mesh in 1.0, with the addition of COLLADA (.dae) in 1.1). The format is the URI-form used by resource_retriever, including the `package://` syntax.

An example of a mesh is `package://pr2_description/meshes/base_v0/base.dae`

Scale on a mesh is relative. A scale of `(1.0, 1.0, 1.0)` means the mesh will display as the exact size specified in the mesh file. A scale of `(1.0, 1.0, 2.0)` means the mesh will show up twice as tall, but the same width/depth.

If the `mesh_use_embedded_materials` flag is set to true and the mesh is of a type which supports embedded materials (such as COLLADA), the material defined in that file will be used instead of the color defined in the marker.

Note2

UR5 添加 robotiq 2 finger gripper (c-model)

1，现在需要为 demo 中的 ur5 更换 gripper，找到了 ros 官网的 robotiq 的 package，github 下载下来后，在 catkin_ws 目录里的 robotiq 包内，有很多文件夹，其中我需要的是两指的 gripper 模型，即 robotiq_c2_model_visualization 这个包，将其 copy 到 demo_manipulation 下的 src 目录下，即可进行下一步操作。

2，修改 top xacro 文件，即 ur5_collision_avoidance.xacro 文件，top xacro 文件的标志是：

```
<robot name="ur5_collision_avoidance" xmlns:xacro="http://ros.org/wiki/xacro">
```

修改 gripper 模型，我们只需要替换以前 include 的 gripper 模型，修改为：

```
<xacro:include filename="$(find
robotiq_c2_model_visualization)/urdf/robotiq_c2_model_macro.xacro" />
```

3，还需要修改其他跟 gripper 有关的地方：

```
<!-- instantiating arm and gripper -->
  <xacro:ur5_robot prefix="" joint_limited="true"/>
    <xacro:robotiq_c2_model prefix="${arm_prefix}"/>
```

这里 robotiq_c2_model 是 robotiq_c2_model_macro.xacro 文件中提供的 name，prefix 是该文件中指定的需要的参数：

```
12 <robot xmlns:xacro="http://ros.org/wiki/xacro">
13   <xacro:macro name="robotiq_c2_model" params="prefix">
```

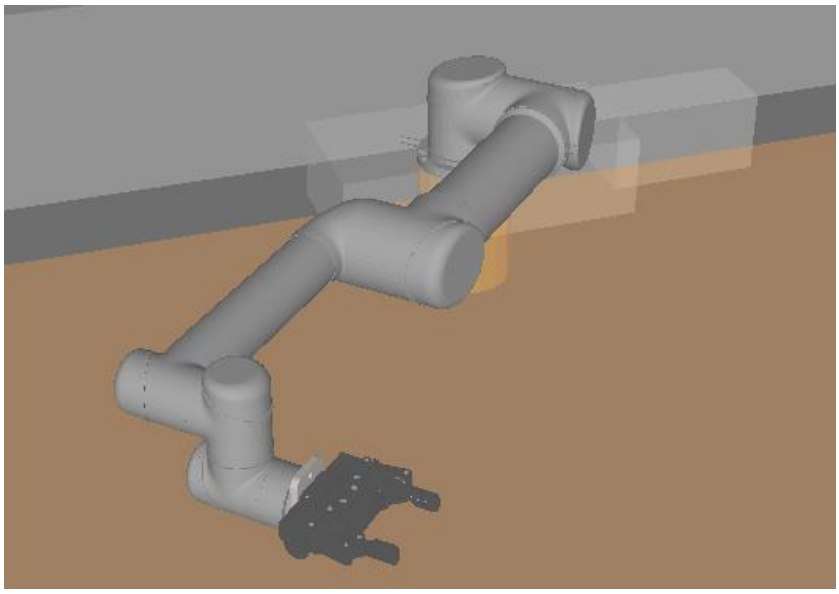
4，robotiq_c2_model.xacro 是一个 top xacro，它是单独使用 gripper 模型的文件，这里告诉了我们 base_link 的名字。

```
 3 <!--
 4     Please note that the base link of the c_model gripper is
 5     "${prefix}robotiq_85_adapter_link"
 6  -->
 7
 8 <robot name="robotiq_c2_model" xmlns:xacro="http://ros.org/wiki/xacro">
 9     <xacro:include filename="$(find robotiq_c2_model_visualization)/urdf/robotiq_c2_model_macro.xacro" />
10     <xacro:robotiq_c2_model prefix=""/>
11 </robot>
```

所以我们还需要修改 joint 的信息：

```
<!-- arm-gripper coupling joint definitions -->
    <joint name="${arm_prefix}to_gripper" type="fixed">
        <parent link="${arm_prefix}ee_link"/>
        <child link="${arm_prefix}robotiq_85_adapter_link"/>
        <origin xyz="0 0 0" rpy="0 0 0"/> <!-- 0 1.57 0 -->
    </joint>
```

5，可以通过 moveit setup assistant 的新建 pkg 中，测试一下生成的模型场景，如果 xacro 文件有错，则会出现 unable to parse 或 unvalid model，下面是效果图：

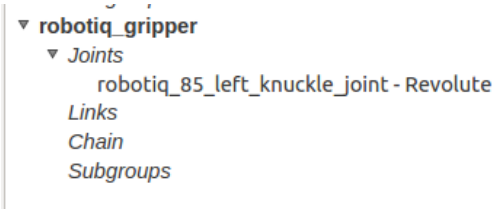


6，进入 assistant 后，我们需要给 gripper 添加一个 planning groups，如何用一个 joint 控制整个 gripper 的 close & open 呢，我们打开 robotiq 的 robotiq_c2_model_macro.xacro 文件查看一下内容，其中发现了以下语法：

```
<joint name="${prefix}robotiq_85_right_knuckle_joint" type="revolute">
  <parent link="${prefix}robotiq_85_base_link"/>
  <child link="${prefix}robotiq_85_right_knuckle_link"/>
  <axis xyz="1 0 0"/>
  <origin rpy="1.5707 -1.5707 0" xyz=".04191 -.0306 0"/>
  <limit lower="0" upper="1.5707" velocity="2.0" effort="1000"/>
  <mimic joint="${prefix}robotiq_85_left_knuckle_joint" multiplier="1"/>
</joint>
```

关于 mimic 的用法，官网有介绍： http://wiki.ros.org/urdf/XML/joint

总之，其他的几个 joint 会根据 robotiq_85_left_knuckle_joint 这个关节做相应的动作，因此我们只需要在 group 里添加这一个关节，即可控制 gripper 了。

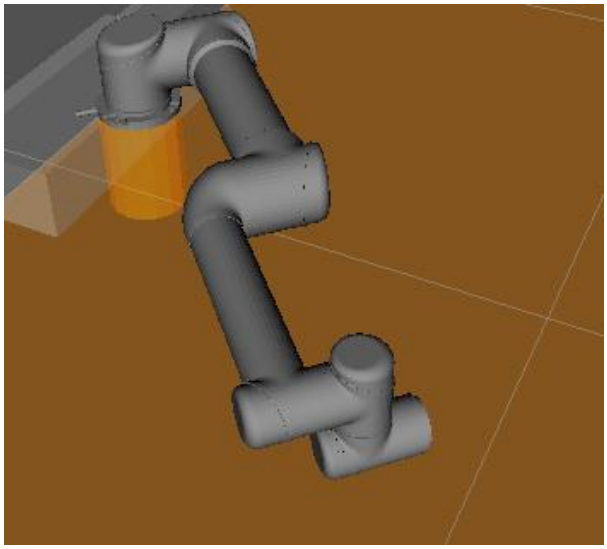


7, important step，update the moveit package:

http://wiki.ros.org/Industrial/Tutorials/Create_a_MoveIt_Pkg_for_an_Industrial_Robot#Update_Configuration_Files

每次在 assistant 中对模型有配置修改，需要重新 update the package，否则运行 moveit_planning_execution.launch 会出现 robot status error。

8，配置完成之后，运行 demo_manipulation 的 ur5_setup，发现 rviz 中不显示 gripper



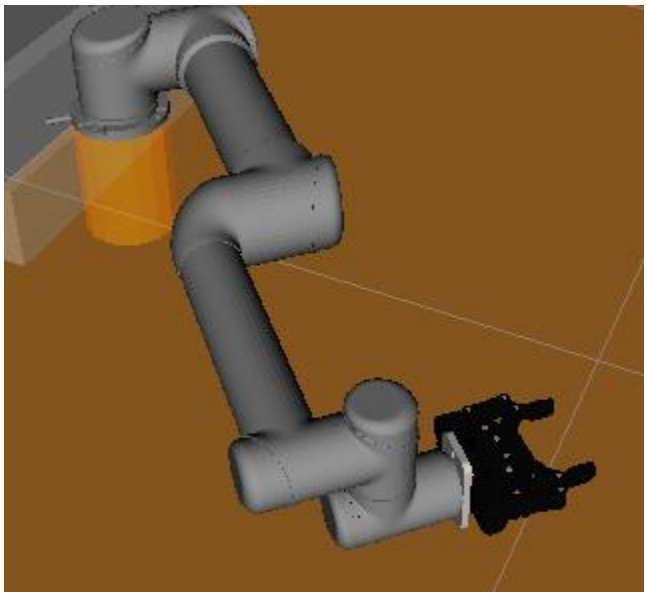
检查了一下 ur5_setup.launch 这个文件，其中加载 rviz 的语句为：

```
<node name="$(anon rviz)" pkg="rviz" type="rviz" respawn="false"
      args="-d $(find collision_avoidance_pick_and_place)/config/ur5/rviz_config.rviz" output="screen" launch-pref
  <rosparam command="load" file="$(find ur5_collision_avoidance_moveit_config)/config/kinematics.yaml"/>
```

所以找到 rviz_config.rviz 这个文件，修改 link 的语句：

```
77      Links:
78        All Links Enabled: false
79        Expand Joint Details: false
80        Expand Link Details: false
81        Expand Tree: false
82        Link Tree Style: Links in Alphabetic Order
83        back_wall2:
```

将这个 false 修改为 true，再运行 ur5_setup.launch 文件，gripper 成功显示。

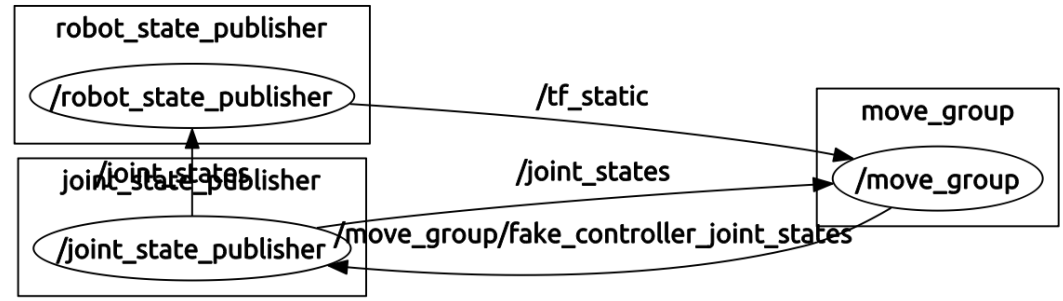


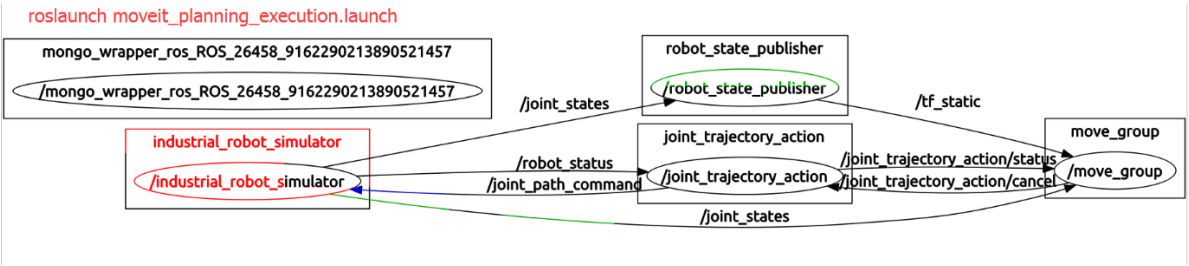
Note3

moveit！ controller ros_control

http://wiki.ros.org/Industrial/Tutorials/Create_a_MoveIt_Pkg_for_an_Industrial_Robot

roslaunch demo.launch





上面两图分别是运行 demo.launch 和 planning_execution.launch 后的节点图，其中 demo 是完全模拟情况下的 moveit 运行情况，joint_state_publisher 发布 robot 的关节信息，这个 demo 是用来检测 moveit PKG 是否成功创建，而如果需要连接到真实 robot 上，还需要创建一个/joint_trajectory_action 节点，它用来向真实的 robot 发布运行轨迹，planning_execution 的作用是模拟真实 robot，这里建立了一个/industrial_robot_simulator 模拟 robot，用来检测 motion plan 的部分。 在 planning_execution 的 rviz 中执行 plan&execute 指令，robot 会出现两次运动(plan&execute）。

在我目前看来： 根据下面这个图里表示的，moveit 是最右边的 user 控制算法部分，而 robot_simulator 和 joint_trajectory_action 一起扮演了最左边的 robot 角色。
那么 move_group 和 robot 之间的通信部分是谁扮演的呢？
move_group 节点中包含了 action_client 的部分， 而 joint_trajectory_action 中包含了 action_server 部分。 而这里的 action_client,action_server 又具体指的什么呢？
从下面的第二张图可以看出来：moveit PKG 中我们配置的 controllers.yaml 文件，就是为了产生 client 的 action 消息，所以 move_group 节点扮演了 action_client 的角色，所以 controllers.yaml 中配置了几个 controller，就会产生几个 action client。 而 joint_trajectory_action 这个节点则是 ros-i core 为我们写的一个 action server，它既能接收 action client 的消息，又能执行 这些控制命令，它能够与 robot 的 interface 连接。
但是有一个问题，这个 action server 只会 joint_trajectory_action 这种 action 类型，所以当我们定义了其他的 controllers.yaml 时，

也就是有其他 action client 时，这些 client 无法接收自己需要的 action 消息， 会出现"action client not connected" 的问题。 如何解决这个问题？

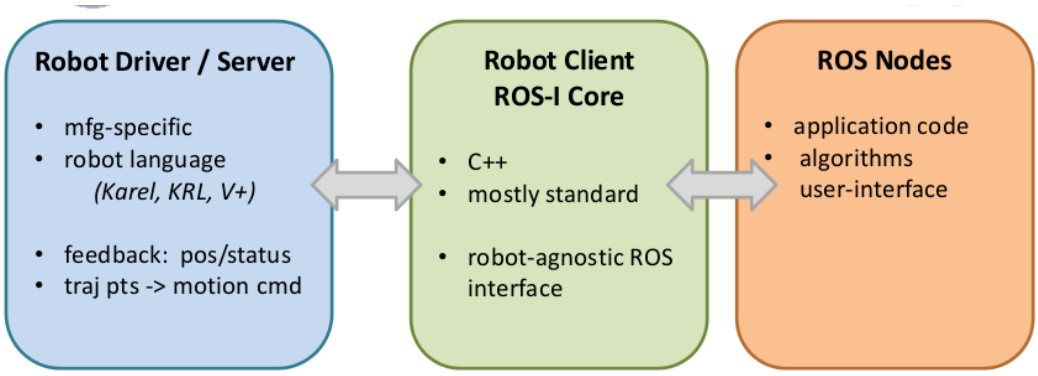
方法： 当做 simulation 的时候，需要启动 robot_simulator 和 joint_trajectory_action 这两个节点，当使用真机的时候，则启动 ur_driver 节点即可。

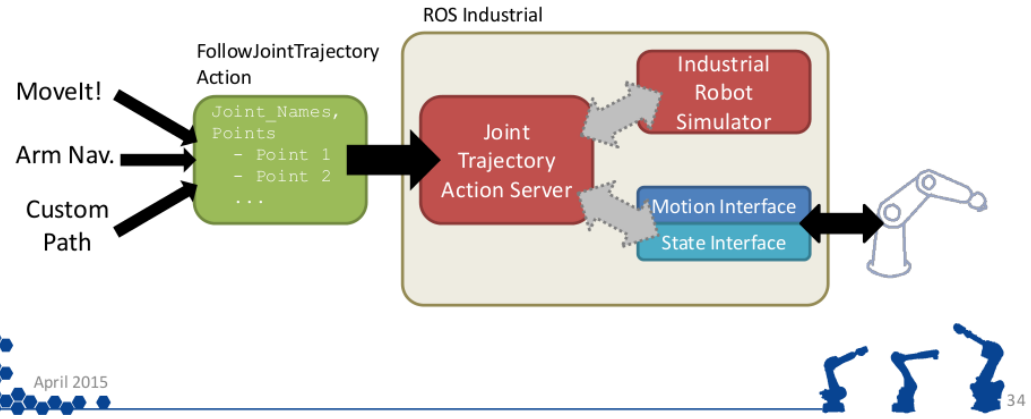
在 simulation 时，我们的一个问题是，系统创建的 joint_trajectory_action 只包含了 server。 当我们有多个 group，例如 ur5+gripper 时，就需要两个不同的 action server，

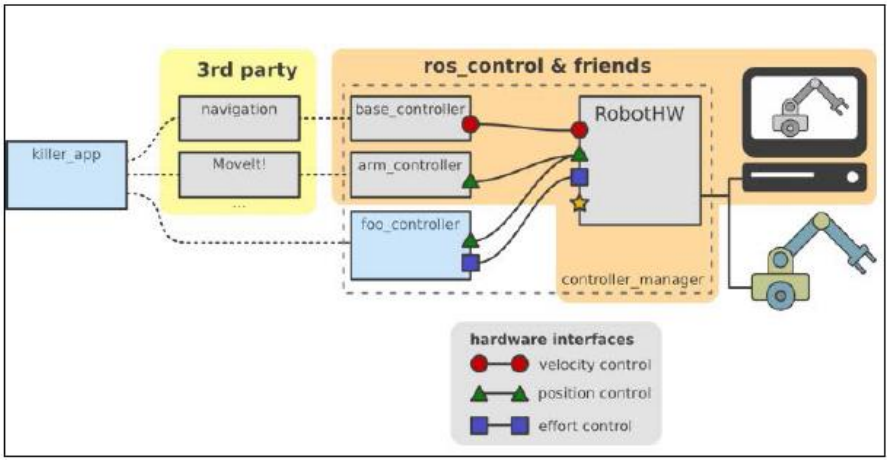
这时我们需要 gazebo 了。我们需要在 gazebo 里模拟真实 robot，用来做为 robot 的反馈。

当使用 gazebo 时，如下面第三个图所示，我们需要根据 urdf 的模型，定制多个 controller，然后这些 controller 一方面驱动 robot 的运动，另一方面作为 action server 与 move_group 相连。

到此为止，我们出现了两种 controller，第一种是开始给 moveit 配置的 controller（作为 action client，与 robot 接口交互），第二种 controller 是为 robot 配置的，用来控制 robot 的运动，并作为 action server 与 move_group 配合。 这里又需要引入 controller manager 和 ros_control 的概念。 controller manager 是用于管理 robot controll 的，ros_control 则是包含了 controller_manager 以及定义了一些 controllers 类型的包。详见：http://wiki.ros.org/ros_control







ps：

"Motion Planning" -> "Plan and Execute" to send trajectory to the sim robot

- you should see the planning scene update twice:
  1. show computed trajectory plan (quick)
  2. show simulated robot "executing" the trajectory (slower)
- you can also confirm/monitor the simulated robot operation at the command line:

```
$ rostopic echo joint_states    (broadcast by simulator)
$ rostopic echo joint_path_command    (inputs to simulator)
```

使用上面的两个指令，可以监控 robot 的状态和运动指令。

在 planning_execution.launch 中，

```
15   <!-- run the robot simulator and action interface nodes -->
16   <group if="$(arg sim)">
17     <include file="$(find industrial_robot_simulator)/launch/robot_interface_simulator.launch" />
18     <rosparam param="initial_joint_state">[1.158, -0.953, 1.906, -1.912, -1.765, 0.0]</rosparam>
19   </group>
20
```

这几条语句，运行了两个节点，即 robot_simulation 和 joint_trajectory_action 节点，

robot_interface_simulator.launch：

```
14   <!-- industrial_robot_simulator: accepts robot commands and reports status -->
15   <node pkg="industrial_robot_simulator" type="industrial_robot_simulator" name="industrial_robot_simulator"/>
16
17   <!-- joint_trajectory_action: provides actionlib interface for high-level robot control -->
18   <node pkg="industrial_robot_client" type="joint_trajectory_action" name="joint_trajectory_action"/>
19
20 </launch>
```

pps：在书中找到了 answer，印证了猜想

> After motion planning, the generated trajectory talks to the controllers in the robot using the `FollowJointTrajectoryAction` interface. This is an action interface in which an action server is run on the robot, and `move_node` initiates an action client which talks to this server and executes the trajectory on the real robot/Gazebo simulator.