6.1URDF和SRDF

7.1 场景规划（碰撞检测和关节约束）PlanningScene类实例化

设置

```
robot_model_loader::RobotModelLoader
robot_model_loader("robot_description");
robot_model::RobotModelPtr kinematic_model = robot_model_loader.getModel();
planning_scene::PlanningScene planning_scene(kinematic_model);
```

自我碰撞检测
构造 CollisionRequest对象和 CollisionResult对象

```
collision_detection::CollisionRequest collision_request;
collision_detection::CollisionResult collision_result;
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 1: Current state is " << (collision_result.collision ?
"in" : "not in") << " self collision");
```

检查状态

```
robot_state::RobotState& current_state =
planning_scene.getCurrentStateNonConst();
current_state.setToRandomPositions();
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 2: Current state is " << (collision_result.collision ?
"in" : "not in") << " self collision");
```

检查组

```
collision_request.group_name = "hand";
current_state.setToRandomPositions();
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result);
ROS_INFO_STREAM("Test 3: Current state is " << (collision_result.collision ?
"in" : "not in") << " self collision");
```

修改允许的碰撞矩阵

```
collision_detection::AllowedCollisionMatrix acm =
planning_scene.getAllowedCollisionMatrix();
robot_state::RobotState copied_state = planning_scene.getCurrentState();

collision_detection::CollisionResult::ContactMap::const_iterator it2;
for (it2 = collision_result.contacts.begin(); it2 !=
collision_result.contacts.end(); ++it2)
{
  acm.setEntry(it2->first.first, it2->first.second, true);
}
collision_result.clear();
planning_scene.checkSelfCollision(collision_request, collision_result,
copied_state, acm);
ROS_INFO_STREAM("Test 6: Current state is " << (collision_result.collision ?
"in" : "not in") << " self collision");
```

全碰撞检测

```
collision_result.clear();
planning_scene.checkCollision(collision_request, collision_result,
copied_state, acm);
ROS_INFO_STREAM("Test 7: Current state is " << (collision_result.collision ?
"in" : "not in") << " self collision");
```

约束检查

检查运动学约束

```
std::string end_effector_name = joint_model_group-
>getLinkModelNames().back();

geometry_msgs::PoseStamped desired_pose;
desired_pose.pose.orientation.w = 1.0;
desired_pose.pose.position.x = 0.3;
desired_pose.pose.position.y = -0.185;
desired_pose.pose.position.z = 0.5;
desired_pose.header.frame_id = "panda_link0";
moveit_msgs::Constraints goal_constraint =
    kinematic_constraints::constructGoalConstraints(end_effector_name,
desired_pose);
```

8.1 PlanningScene的ORS API

运行代码
脚本一

> roslaunch panda_moveit_config demo.launch
>
> 脚本二
>
> roslaunch moveit_tutorials planning_scene_ros_api_tutorial.launch

Visualization可视化

```
moveit_visual_tools::MoveItVisualTools visual_tools("panda_link0");
visual_tools.deleteAllMarkers();
```

ROS应用程序接口 规划场景发布者的 ROS API 通过使用“diffs”的主题接口实现的。规划场景差异是当前规划场景（由 move_group 节点维护）与用户所需的新规划场景之间的差异。

发布所需主题

```
ros::Publisher planning_scene_diff_publisher =
node_handle.advertise<moveit_msgs::PlanningScene>("planning_scene", 1);
ros::WallDuration sleep_t(0.5);
while (planning_scene_diff_publisher.getNumSubscribers() < 1)
{
  sleep_t.sleep();
}
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to start
the demo");
```

定义附加的对象消息
可以使用此消息从世界中添加或减去对象，并将该对象附加到机器人。

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

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

注意将对象附加到机器人需要将相应的操作指定为ADD操作

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

由于将物体附加到机器人手上来模拟拾取物体，因此碰撞检查器忽略物体和机器人手之间的碰撞

```
attached_object.touch_links = std::vector<std::string>{ "panda_hand",
"panda_leftfinger", "panda_rightfinger" };
```

将对象添加到环境中
方法是将对象添加到规划场景的“世界”部分中的一组碰撞对象中

```
ROS_INFO("Adding the object into the world at the location of the hand.");
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);
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
//这段代码是用于提示用户在RvizVisualToolsGui窗口中
按下"next"键以开始演示的。
```

同步更新与异步更新
两种独立的机制可用于使用差异与 move_group 节点进行交互：

1.通过 rosservice 调用发送 diff，并阻止直到应用 diff（同步更新）
2.通过话题发送差异，即使差异可能尚未应用也继续（异步更新）

服务客服端service client替换planning_scene_diff_publisher

```
ros::ServiceClient planning_scene_diff_client =
    node_handle.serviceClient<moveit_msgs::ApplyPlanningScene>
("apply_planning_scene");
planning_scene_diff_client.waitForExistence();
```

之后并通过服务调用将差异发送到规划场景

```
moveit_msgs::ApplyPlanningScene srv;
srv.request.scene = planning_scene;
planning_scene_diff_client.call(srv);
```

之后确定应用差异

附加一个对象需要两个操作
从环境中移除原本对象
这里清除碰撞对象列表 将物体附加到机器人上

```
moveit_msgs::CollisionObject remove_object;
remove_object.id = "box";
remove_object.header.frame_id = "panda_link0";
remove_object.operation = remove_object.REMOVE;
```

注意下面有首先清除这些字段来确保 diff 消息不包含其他附加对象或冲突对象

```
ROS_INFO("Attaching the object to the hand 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(attached_obj
ect);
planning_scene_diff_publisher.publish(planning_scene);

visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
```

从机器人上分离物体
从碰撞世界中移除对象只需要使用之前定义的移除对象消息
通过首先清除这些字段来确保 diff 消息不包含其他附加对象或冲突对象

```
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);
```

调试规划场景监视器Planning Scene Monitor
作用是打印出当前系统中的规划场景信息，包括分离和附着的碰撞对象。

```
rosrun moveit_ros_planning moveit_print_planning_scene_info
```

包括物体的位置、姿态、形状等信息，以及它们之间的碰撞关系。通过查看这些信息，可以更好地理解和调试你的机器人运动规划系统。

9.1Motion Planning API运动规划API

MoveIt 中，运动规划器是使用插件基础设施加载的。MoveIt 在运行时加载运动规划器。

运行代码
脚本一

> roslaunch panda_moveit_config demo.launch
> 脚本二
> roslaunch moveit_tutorials motion_planning_api_tutorial.launch

设置开始使用运动计划器。运动规划器在 MoveIt 中设置为插件，使用 ROS pluginlib 接口加载需要使用的任何规划器。在加载规划器之前，我们需要两个对象：RobotModel 和 PlanningScene

```
const std::string PLANNING_GROUP = "panda_arm";
robot_model_loader::RobotModelLoader
robot_model_loader("robot_description");
robot_model::RobotModelPtr robot_model = robot_model_loader.getModel();
```

```
robot_state::RobotStatePtr robot_state(new
robot_state::RobotState(robot_model));
const robot_state::JointModelGroup* joint_model_group = robot_state-
>getJointModelGroup(PLANNING_GROUP);
```

使用RobotModel，我们可以构建一个维护世界状态（包括机器人）的PlanningScene

```
planning_scene::PlanningScenePtr planning_scene(new
planning_scene::PlanningScene(robot_model));
```

配置有效的机器人状态

```
planning_scene-
>getCurrentStateNonConst().setToDefaultValues(joint_model_group, "ready");
```

现在将构建一个加载器来按名称加载计划器。在这里使用 ROS pluginlib 库

```
boost::scoped_ptr<pluginlib::ClassLoader<planning_interface::PlannerManager>
> planner_plugin_loader;
planning_interface::PlannerManagerPtr planner_instance;
std::string planner_plugin_name;
```

我们将从 ROS 参数服务器获取要加载的规划插件的名称，然后加载规划器以确保捕获所有异常

```
if (!node_handle.getParam("planning_plugin", planner_plugin_name))
  ROS_FATAL_STREAM("Could not find planner plugin name");
try
{
  planner_plugin_loader.reset(new
pluginlib::ClassLoader<planning_interface::PlannerManager>(
      "moveit_core", "planning_interface::PlannerManager"));
}
catch (pluginlib::PluginlibException& ex)
{
  ROS_FATAL_STREAM("Exception while creating planning plugin loader " <<
ex.what());
}
try
{
  planner_instance.reset(planner_plugin_loader-
>createUnmanagedInstance(planner_plugin_name));
  if (!planner_instance->initialize(robot_model,
node_handle.getNamespace()))
    ROS_FATAL_STREAM("Could not initialize planner instance");
  ROS_INFO_STREAM("Using planning interface '" << planner_instance-
>getDescription() << "'");
}
catch (pluginlib::PluginlibException& ex)
{
  const std::vector<std::string>& classes = planner_plugin_loader-
>getDeclaredClasses();
  std::stringstream ss;
  for (std::size_t i = 0; i < classes.size(); ++i)
    ss << classes[i] << " ";
  ROS_ERROR_STREAM("Exception while loading planner '" <<
planner_plugin_name << "': " << ex.what() << std::endl
                                                       << "Available
plugins: " << ss.str());
}
```

Visualization可视化

MoveItVisualTools 包提供了许多用于可视化 RViz 中的对象、机器人和轨迹的功能以及调试工具

```
namespace rvt = rviz_visual_tools;
moveit_visual_tools::MoveItVisualTools visual_tools("panda_link0");
visual_tools.loadRobotStatePub("/display_robot_state");
visual_tools.enableBatchPublishing();
visual_tools.deleteAllMarkers();  // clear all old markers
visual_tools.trigger();

/* Remote control is an introspection tool that allows users to step through
a high level script
   via buttons and keyboard shortcuts in RViz */
visual_tools.loadRemoteControl();

/* RViz provides many types of markers, in this demo we will use text,
cylinders, and spheres*/
Eigen::Isometry3d text_pose = Eigen::Isometry3d::Identity();
text_pose.translation().z() = 1.75;
visual_tools.publishText(text_pose, "Motion Planning API Demo", rvt::WHITE,
rvt::XLARGE);

/* Batch publishing is used to reduce the number of messages being sent to
RViz for large visualizations */
visual_tools.trigger();

/* We can also use visual_tools to wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to start
the demo");
```

Pose Goal姿势目标

现在将为 Panda 手臂创建一个运动计划请求，指定末端执行器的所需姿势作为输入

MotionPlanRequest和MotionPlanResponse

```
visual_tools.publishRobotState(planning_scene->getCurrentStateNonConst(),
rviz_visual_tools::GREEN);
visual_tools.trigger();
planning_interface::MotionPlanRequest req;
planning_interface::MotionPlanResponse res;
geometry_msgs::PoseStamped pose;
pose.header.frame_id = "panda_link0";
pose.pose.position.x = 0.3;
pose.pose.position.y = 0.4;
pose.pose.position.z = 0.75;
pose.pose.orientation.w = 1.0;
```

位置公差规定为 0.01 m，方向规定公差为 0.01 弧度

```
std::vector<double> tolerance_pose(3, 0.01);
std::vector<double> tolerance_angle(3, 0.01);
```

我们将使用 kinematic_constraints 包中提供的辅助函数将请求创建为约束

```
moveit_msgs::Constraints pose_goal =
    kinematic_constraints::constructGoalConstraints("panda_link8", pose,
tolerance_pose, tolerance_angle);

req.group_name = PLANNING_GROUP;
req.goal_constraints.push_back(pose_goal);
```

我们现在构建一个封装场景、请求和响应的规划上下文
使用这个规划上下文来调用规划器
getPlanningContext()函数接受三个参数：planning_scene表示机器人的规划场景，req表示路径规划请求，res.error_code_用于存储错误信息

```
planning_interface::PlanningContextPtr context =
    planner_instance->getPlanningContext(planning_scene, req,
res.error_code_);
context->solve(res);
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
```

Visualize the result可视化结果

```
ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>
("/move_group/display_planned_path", 1, true);
moveit_msgs::DisplayTrajectory display_trajectory;

/* Visualize the trajectory */
moveit_msgs::MotionPlanResponse response;
res.getMessage(response);

display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
visual_tools.publishTrajectoryLine(display_trajectory.trajectory.back(),
joint_model_group);
visual_tools.trigger();
display_publisher.publish(display_trajectory);

/* Set the state in the planning scene to the final state of the last plan
*/
robot_state->setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
planning_scene->setCurrentState(*robot_state.get());
```

显示目标状态

```
visual_tools.publishRobotState(planning_scene->getCurrentStateNonConst(),
rviz_visual_tools::GREEN);
visual_tools.publishAxisLabeled(pose.pose, "goal_1");
visual_tools.publishText(text_pose, "Pose Goal (1)", rvt::WHITE,
rvt::XLARGE);
visual_tools.trigger();

/* We can also use visual_tools to wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
```

Joint Space Goals 联合空间目标

```
robot_state::RobotState goal_state(robot_model);
std::vector<double> joint_values = { -1.0, 0.7, 0.7, -1.5, -0.7, 2.0, 0.0 };
goal_state.setJointGroupPositions(joint_model_group, joint_values);
moveit_msgs::Constraints joint_goal =
kinematic_constraints::constructGoalConstraints(goal_state,
joint_model_group);
req.goal_constraints.clear();
req.goal_constraints.push_back(joint_goal);
```

响应规划器并可视化轨迹

```
/* Re-construct the planning context */
context = planner_instance->getPlanningContext(planning_scene, req,
res.error_code_);
/* Call the Planner */
context->solve(res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
/* Visualize the trajectory */
res.getMessage(response);
display_trajectory.trajectory.push_back(response.trajectory);

/* Now you should see two planned trajectories in series*/
visual_tools.publishTrajectoryLine(display_trajectory.trajectory.back(),
joint_model_group);
visual_tools.trigger();
display_publisher.publish(display_trajectory);

/* We will add more goals. But first, set the state in the planning
   scene to the final state of the last plan */
robot_state->setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
planning_scene->setCurrentState(*robot_state.get());
```

显示目标状态

```
visual_tools.publishRobotState(planning_scene->getCurrentStateNonConst(),
rviz_visual_tools::GREEN);
visual_tools.publishAxisLabeled(pose.pose, "goal_2");
visual_tools.publishText(text_pose, "Joint Space Goal (2)", rvt::WHITE,
rvt::XLARGE);
visual_tools.trigger();

/* Wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");

/* Now, we go back to the first goal to prepare for orientation constrained
planning */
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal);
context = planner_instance->getPlanningContext(planning_scene, req,
res.error_code_);
context->solve(res);
res.getMessage(response);

display_trajectory.trajectory.push_back(response.trajectory);
```

```
visual_tools.publishTrajectoryLine(display_trajectory.trajectory.back(),
joint_model_group);
visual_tools.trigger();
display_publisher.publish(display_trajectory);

/* Set the state in the planning scene to the final state of the last plan
*/
robot_state->setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
planning_scene->setCurrentState(*robot_state.get());
```

显示目标状态

```
visual_tools.publishRobotState(planning_scene->getCurrentStateNonConst(),
rviz_visual_tools::GREEN);
visual_tools.trigger();

/* Wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
```

Adding Path Constraints添加路径约束

添加一个新的姿势目标。这次将为运动添加路径约束

```
pose.pose.position.x = 0.32;
pose.pose.position.y = -0.25;
pose.pose.position.z = 0.65;
pose.pose.orientation.w = 1.0;
moveit_msgs::Constraints pose_goal_2 =
    kinematic_constraints::constructGoalConstraints("panda_link8", pose,
tolerance_pose, tolerance_angle);

/* Now, let's try to move to this new pose goal*/
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal_2);

/* But, let's impose a path constraint on the motion.
   Here, we are asking for the end-effector to stay level*/
geometry_msgs::QuaternionStamped quaternion;
quaternion.header.frame_id = "panda_link0";
quaternion.quaternion.w = 1.0;
req.path_constraints =
kinematic_constraints::constructGoalConstraints("panda_link8", quaternion);
```

施加路径约束需要规划器在末端执行器的可能位置空间（机器人的工作空间）中进行推理，因此，我们还需要指定允许的规划体积的界限；注意默认边界由 WorkspaceBounds 请求适配器自动填充

```
req.workspace_parameters.min_corner.x =
req.workspace_parameters.min_corner.y =
    req.workspace_parameters.min_corner.z = -2.0;
req.workspace_parameters.max_corner.x =
req.workspace_parameters.max_corner.y =
    req.workspace_parameters.max_corner.z = 2.0;
```

呼应计划器和可视化所有创建的计划

```
context = planner_instance->getPlanningContext(planning_scene, req,
res.error_code_);
context->solve(res);
res.getMessage(response);
display_trajectory.trajectory.push_back(response.trajectory);
visual_tools.publishTrajectoryLine(display_trajectory.trajectory.back(),
joint_model_group);
visual_tools.trigger();
display_publisher.publish(display_trajectory);

/* Set the state in the planning scene to the final state of the last plan
*/
robot_state->setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
planning_scene->setCurrentState(*robot_state.get());
```

显示目标状态

```
visual_tools.publishRobotState(planning_scene->getCurrentStateNonConst(),
rviz_visual_tools::GREEN);
visual_tools.publishAxisLabeled(pose.pose, "goal_3");
visual_tools.publishText(text_pose, "Orientation Constrained Motion Plan
(3)", rvt::WHITE, rvt::XLARGE);
visual_tools.trigger();
```

http://t.csdnimg.cn/rvuhS

10.1运动规划管道

在 MoveIt 中，运动规划器被设置为规划路径
我们可能想要预处理运动规划请求或后处理规划的路径，使用规划管道，将运动规划器与预处理和后处理阶段链接起来。预处理和后处理阶段称为规划请求适配器，可以通过 ROS 参数服务器的名称进行配置

运行代码
脚本一

```
source ~/ws_moveit/devel/setup.bash
roslaunch panda_moveit_config demo.launch
```

脚本二

```
source ~/ws_moveit/devel/setup.bash
roslaunch moveit_tutorials motion_planning_pipeline_tutorial.launch
```

设置开始使用规划管道，加载规划器之前，需要两个对象：RobotModel 和 PlanningScene
实例化一个RobotModelLoader对象，它将在ROS参数服务器上查找机器人描述并构造一个RobotModel

```
robot_model_loader::RobotModelLoader
robot_model_loader("robot_description");
robot_model::RobotModelPtr robot_model = robot_model_loader.getModel();
```

使用RobotModel，我们可以构建一个维持世界状态（包括机器人）的PlanningScene

```
planning_scene::PlanningScenePtr planning_scene(new
planning_scene::PlanningScene(robot_model));
```

设置 PlanningPipeline 对象，它将使用 ROS 参数服务器来确定请求适配器集和要使用的规划插件

```
planning_pipeline::PlanningPipelinePtr planning_pipeline(
    new planning_pipeline::PlanningPipeline(robot_model, node_handle,
"planning_plugin", "request_adapters"));
```

可视化

```
namespace rvt = rviz_visual_tools;
moveit_visual_tools::MoveItVisualTools visual_tools("panda_link0");
visual_tools.deleteAllMarkers();

/* Remote control is an introspection tool that allows users to step through
a high level script
   via buttons and keyboard shortcuts in RViz */
```

```
visual_tools.loadRemoteControl();

/* RViz provides many types of markers, in this demo we will use text,
cylinders, and spheres*/
Eigen::Isometry3d text_pose = Eigen::Isometry3d::Identity();
text_pose.translation().z() = 1.75;
visual_tools.publishText(text_pose, "Motion Planning Pipeline Demo",
rvt::WHITE, rvt::XLARGE);

/* Batch publishing is used to reduce the number of messages being sent to
RViz for large visualizations */
visual_tools.trigger();

/* Sleep a little to allow time to startup rviz, etc..
   This ensures that visual_tools.prompt() isn't lost in a sea of logs*/
ros::Duration(10).sleep();

/* We can also use visual_tools to wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to start
the demo");
```

姿势目标

Panda 的右臂创建一个运动计划请求，指定末端执行器的所需姿势作为输入

```
planning_interface::MotionPlanRequest req;
planning_interface::MotionPlanResponse res;
geometry_msgs::PoseStamped pose;
pose.header.frame_id = "panda_link0";
pose.pose.position.x = 0.3;
pose.pose.position.y = 0.0;
pose.pose.position.z = 0.75;
pose.pose.orientation.w = 1.0;
```

位置公差规定为 0.01 m，方向规定公差为 0.01 弧度

```
std :: vector <double> tolerance_pose ( 3,0.01 ) ; std :: vector <double>
tolerance_angle ( 3,0.01 ) ;
```

使用kinematic_constraints包中提供的辅助函数将请求创建为约束

```
req.group_name = "panda_arm";
moveit_msgs::Constraints pose_goal =
    kinematic_constraints::constructGoalConstraints("panda_link8", pose,
tolerance_pose, tolerance_angle);
req.goal_constraints.push_back(pose_goal);
```

调用管道并检查规划是否成功

```
planning_pipeline->generatePlan(planning_scene, req, res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
```

可视化结果

```
ros::Publisher display_publisher =
    node_handle.advertise<moveit_msgs::DisplayTrajectory>
("/move_group/display_planned_path", 1, true);
moveit_msgs::DisplayTrajectory display_trajectory;

/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
moveit_msgs::MotionPlanResponse response;
res.getMessage(response);

display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
display_publisher.publish(display_trajectory);

/* Wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
```

关节空间目标

```
/* 首先，将计划场景中的状态设置为上一个计划的最终状态 */
robot_state = planning_scene_monitor::LockedPlanningSceneRO(psm)-
>getCurrentStateUpdated(response.trajectory_start);
robot_state->setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
moveit::core::robotStateToRobotStateMsg(*robot_state, req.start_state);
```

设定一个关节空间目标

```
robot_state::RobotState goal_state(robot_model);
std::vector<double> joint_values = { -1.0, 0.7, 0.7, -1.5, -0.7, 2.0, 0.0 };
goal_state.setJointGroupPositions(joint_model_group, joint_values);
moveit_msgs::Constraints joint_goal =
kinematic_constraints::constructGoalConstraints(goal_state,
joint_model_group);

req.goal_constraints.clear();
req.goal_constraints.push_back(joint_goal);
```

调用管道并可视化轨迹

```
planning_pipeline->generatePlan(planning_scene, req, res);
/* Check that the planning was successful */
if (res.error_code_.val != res.error_code_.SUCCESS)
{
  ROS_ERROR("Could not compute plan successfully");
  return 0;
}
/* Visualize the trajectory */
ROS_INFO("Visualizing the trajectory");
res.getMessage(response);
display_trajectory.trajectory_start = response.trajectory_start;
display_trajectory.trajectory.push_back(response.trajectory);
```

两个串联的计划轨迹

```
display_publisher.publish(display_trajectory);

/* Wait for user input */
visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
continue the demo");
```

使用计划请求适配器
规划请求适配器允许我们指定在规划发生之前或在结果路径 上完成规划之后应该发生的一系列操作

```
/* First, set the state in the planning scene to the final state of the last
plan */
robot_state = planning_scene->getCurrentStateNonConst();
planning_scene->setCurrentState(response.trajectory_start);
robot_state.setJointGroupPositions(joint_model_group,
response.trajectory.joint_trajectory.points.back().positions);
```

将其中一个关节设置为稍微超出其上限

```
const robot_model::JointModel* joint_model = joint_model_group-
>getJointModel("panda_joint3");
const robot_model::JointModel::Bounds& joint_bounds = joint_model-
```

```
>getVariableBounds();
std::vector<double> tmp_values(1, 0.0);
tmp_values[0] = joint_bounds[0].min_position_ - 0.01;
robot_state.setJointPositions(joint_model, tmp_values);
req.goal_constraints.clear();
req.goal_constraints.push_back(pose_goal);
```

呼叫规划器并可视化轨迹

```
  planning_pipeline->generatePlan(planning_scene, req, res);
  if (res.error_code_.val != res.error_code_.SUCCESS)
  {
    ROS_ERROR("Could not compute plan successfully");
    return 0;
  }
  /* Visualize the trajectory */
  ROS_INFO("Visualizing the trajectory");
  res.getMessage(response);
  display_trajectory.trajectory_start = response.trajectory_start;
  display_trajectory.trajectory.push_back(response.trajectory);
  /* Now you should see three planned trajectories in series*/
  display_publisher.publish(display_trajectory);

  /* Wait for user input */
  visual_tools.prompt("Press 'next' in the RvizVisualToolsGui window to
finish the demo");

  ROS_INFO("Done");
  return 0;
```

##可视化碰撞 Roslaunch启动文件以直接从moveit_tutorials运行代码

```
roslaunch moveit_tutorials motion_planning_pipeline_tutorial.launch
```

我们使用InteractiveRobot 对象作为包装器，将 robots_model 与立方体和交互式标记组合在一起。我们还创建一个PlanningScene来进行碰撞检查

```
InteractiveRobot robot;
/* Create a PlanningScene */
g_planning_scene = new planning_scene::PlanningScene(robot.robotModel());
```

将几何体添加到 PlanningScene

```
Eigen::Isometry3d world_cube_pose;
double world_cube_size;
robot.getWorldGeometry(world_cube_pose, world_cube_size);
g_world_cube_shape.reset(new shapes::Box(world_cube_size, world_cube_size,
world_cube_size));
g_planning_scene->getWorldNonConst()->addToObject("world_cube",
g_world_cube_shape, world_cube_pose);
```

碰撞请求

Panda 机器人创建一个碰撞请求

```
collision_detection::CollisionRequest c_req;
collision_detection::CollisionResult c_res;
c_req.group_name = robot.getGroupName();
c_req.contacts = true;
c_req.max_contacts = 100;
c_req.max_contacts_per_pair = 5;
c_req.verbose = false;
```

检查碰撞

检查机器人与其自身或世界之间的碰撞

```
g_planning_scene->checkCollision(c_req, c_res, *robot.robotState());
```

显示碰撞接触点

如果发生碰撞，我们会获取接触点并将显示为标记。getCollisionMarkersFromContacts()是一个辅助函数，它将碰撞接触点添加到 MarkerArray 消息中。如果您想将接触点用于显示以外的其他用途，您可以迭代 c_res.contacts，它是接触点的 std::map

```
if (c_res.collision)
{
  ROS_INFO("COLLIDING contact_point_count=%d", (int)c_res.contact_count);
  if (c_res.contact_count > 0)
  {
    std_msgs::ColorRGBA color;
    color.r = 1.0;
    color.g = 0.0;
    color.b = 1.0;
    color.a = 0.5;
    visualization_msgs::MarkerArray markers;

    /* Get the contact points and display them as markers */
    collision_detection::getCollisionMarkersFromContacts(markers,
"panda_link0", c_res.contacts, color,ros::Duration(),  // remain until
deleted
    0.01);            // radius
    publishMarkers(markers);
  }
}
```