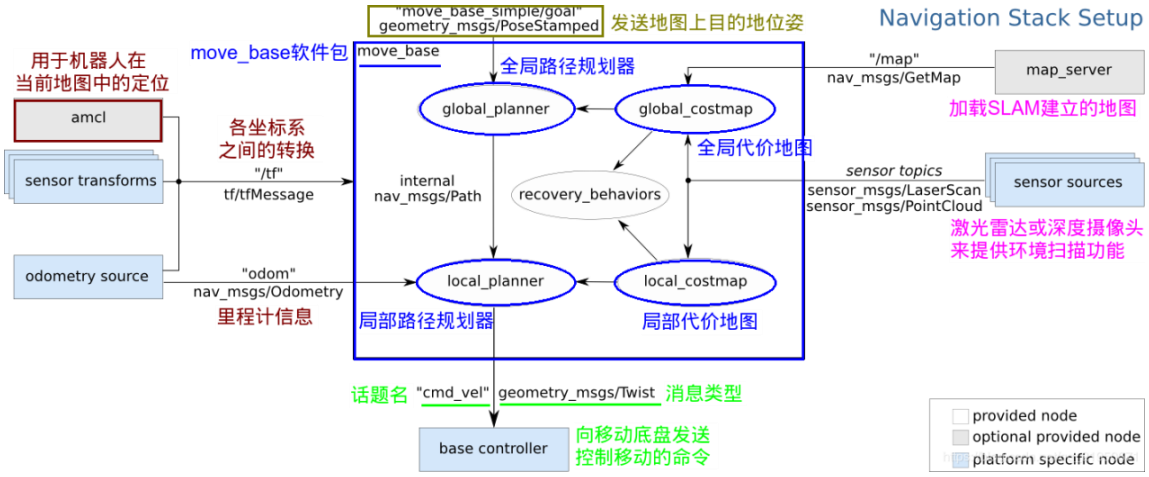
轻舟机器人路径规划及 move_base 包的使用说明

AI 航团队

路径规划是自主导航的核心，整个 navigation 栈都是为绕 move_base 来展开的，所以接下来的重点使用官网的 move_base 包，让它服务于我们的机器人。本教程内容完全来源于官网，示例代码及参数仅供帮助理解该部分内容，并非实际使用，轻舟机器人的代码搭建思路完全参照本教程，具体实现需参见下一篇教程。

官网有更详细资料：http://wiki.ros.org/navigation/Tutorials/RobotSetup

1. 轻舟机器人导航框架解析



move_base 节点位于导航框架正中心，可以理解为一个强大的路径规划器，在实际的导航任务中，只需要启动这一个 node，并且给他提供数据，就可以规划出路径和速度。在 navigation 这个问题的角度，map 是作为已知信息，默认是已经解决的（使用 gmapping 等算法提前将地图建立完成）。

所以，配置机器人，导航功能包集将使其可以运动。**白色**的部分是必须且已实现的组件，**灰色**的部分是可选且已实现的组件，**蓝色**的部分是必须为每一个机器人平台创建的组件。所以配置导航功能包集就转化为为 move_base 提供必须的激光雷达、里程计以及 tf 等信息，使其计算出控制指令，再将控制指令用于底盘控制的过程。

在总体框架图中可以看到，move_base 提供了 ROS 导航的配置、运行、交互接口，它主要包括两个部分：

（1）全局路径规划（global planner）：根据给定的目标位置进行总体路径的规划。

在 ROS 的导航中，首先会通过全局路径规划，计算出机器人到目标位置的全局路线。这一功能默认是 navfn 这个包实现的。base_global _planner 实现方式包括：

parrot_planner 实现了较简单的全局规划算法

Navfn 实现了 Dijkstra 和 A*全局规划算法

global_planner 可以看作 navfn 的改进版

（2）本地实时规划（local planner）：根据附近的障碍物进行躲避路线规划。

本地的实时规划是利用 base_local_planner 包实现的。该包使用 Trajectory Rollout 和 Dynamic Window approaches 算法计算机器人每个周期内应该行驶的速度和角度（dx，dy，dtheta velocities）。

base_local_planner 这个包通过地图数据，通过算法搜索到达目标的多条路经，利用一些评价标准（是否会撞击障碍物，所需要的时间等等）选取最优的路径，并且计算所需要的实时速度和角度。

其中，Trajectory Rollout 和 Dynamic Window approaches 算法的主要思路如下：

（1） 采样机器人当前的状态（dx,dy,dtheta）；

（2） 针对每个采样的速度，计算机器人以该速度行驶一段时间后的状态，得出一条行驶的路线。

（3） 利用一些评价标准为多条路线打分。

（4） 根据打分，选择最优路径。

（5） 重复上面过程。

2. 硬件要求

尽管导航功能包设计得尽可能通用，但是仍然对机器人的硬件有以下三个要求：

（1）导航功能包仅对差分等轮式机器人有效，并且假设机器人可直接使用速度指令进行控制，速度指令的格式为：x 方向速度、y 方向速度、速度向量角度。

（2）导航功能包要求机器人必须安装有激光雷达等二维平面测距设备。

（3）导航功能包以正方型的机器人为模型进行开发，所以对于正方形或者圆形外形的机器人支持度较好，而对于其他外形的机器人来讲，虽然仍然可以正常使用，但是表现则很有可能不佳。

3.机器人配置

简单来说，就是按照 move_base 包的需求，将需要的功能完成即可。其中白色的部分是 ROS 功能包已经完成的部分，不需要我们去实现，灰色的是可选的部分，也由 ROS 完成，在使用中根据需求使用，需要关注的重点部分是蓝色部分，这些需要我们根据输入输出的要求完成相应的功能。

3.1 ROS

首先，请确保你的机器人安装了 ROS 框架。请查阅 ROS documentation 以了解如何在你的机器人上安装 ROS：

http://wiki.ros.org/ROS

3.2、tf 变换(sensortransforms)

导航功能包集需要机器人不断使用 tf 发布有关坐标系之间的关系的信息。详细的配置教程请查阅：tf 配置。

http://wiki.ros.org/cn/navigation/Tutorials/RobotSetup/TF

3.3、传感器信息(sensor sources)

导航功能包集使用来自传感器的信息避开现实环境中的障碍物，它假定这些传感器在 ROS 上不断发布 sensor_msgs/LaserScan 消息或者 sensor_msgs/PointCloud 消息。有关在 ROS 上发布这些消息的教程，请查阅在 ROS 上发布传感器数据流：

http://wiki.ros.org/cn/navigation/Tutorials/RobotSetup/Sensors。

3.4、里程计信息(odometrysource)

导航功能包要求机器人发布 nav_msgs/Odometry 格式的里程计信息，同时也要发布相应的 tf 变换。在 ROS 上发布里程计信息：

http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom

3.5、基座控制器(base controller)

导航功能包集假定它可以通过话题"cmd_vel"发布 geometry_msgs/Twist 类型的消息，这个消息基于机器人的基座坐标系，它传递的是运动命令。这意味着必须有一个节点订阅"cmd_vel"话题，将该话题上的速度命令(vx, vy, vtheta 转换为电机命令(cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z)发送给移动基座。

3.6、地图（map_server）

导航功能包集的配置并不需要有一张地图以供操作，但基于本教程的目的，我们假定你有一张地图。请查阅教程创建一张地图了解在你的系统环境下创建一张地图的细节。

http://wiki.ros.org/slam_gmapping/Tutorials/MappingFromLoggedData

4、导航功能包集的配置

假设上述所有需要的环境都已满足。特别的，机器人必须使用 tf 发布坐标帧，并从所有的传感器接收 sensor_msgs/LaserScan 或者 sensor_msgs/PointCloud 消息用于导航，同时需要使用 tf 和 nav_msgs/Odometry 消息发布导航消息，消息会以命令的形式下发给机器人底座。如果所需要的配置你都没有，请参见机器人配置。

4.1、创建一个功能包

首先，我们创建一个软件包，用来保存我们所有的配置文件和启动文件。这个软件包需要包含所有用于实现机器人配置小节所述以来，就如其以依赖导航功能包集高级接口 move_base 软件包一样。因此，为你的软件包选好位置，执行以下命令:

```
catkin_create_pkg my_robot_name_2dnav move_base my_tf_configuration_dep my_odom_configuration_de
p my_sensor_configuration_dep
```

这个指令会创建一个包含运行导航功能包集所需依赖的软件包。

4.2 创建机器人启动配置文件

现在，我们有了一个存放所有配置文件和启动文件的工作空间，我们会创建一个 roslaunch 文件来启动所有的硬件以及发布机器人所需的 tf。打开你喜欢的编辑器，粘贴以下内容到 my_robot_configuration.launch。你可以自由的将 "my_robot" 改成你的机器人的名字。以后，我们会对 launch 文件做相似的更改，确保你阅读了本节其余内容。

```
<launch>
  <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_node_name" output="screen">
    <param name="sensor_param" value="param_value" />
  </node>

  <node pkg="odom_node_pkg" type="odom_node_type" name="odom_node" output="screen">
    <param name="odom_param" value="param_value" />
  </node>

  <node pkg="transform_configuration_pkg" type="transform_configuration_type" name="transform_configuration_name" output
="screen">
    <param name="transform_configuration_param" value="param_value" />
  </node>
</launch>
```

现在我们有了一个 launch 文件模板，但是，我们需要根据自己的机器人去完善它。以下章节，我们会逐步的改变它。

```
<launch>

   <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_node_name" output="screen">
```

这里，我们会启动机器人运行导航功能包所需的所有传感器。用你的传感器对应的 ROS 驱动包替换"sensor_node_pkg"，用你的传感器类型替换"sensor_node_type"（通常与节点名一致），用你的传感器节点名替换"sensor_node_name"，"sensor_param"包含所有必需的参数。注意，如果你有多个传感器，在这里一起启动它们。

```
</node>
<node pkg="odom_node_pkg" type="odom_node_type" name="odom_node" output="screen">
   <param name="odom_param" value="param_value" />
</node>
```

这里，我们启动基座（底盘）的里程计。同样，替换相应的 pkg, type, name，并根据实际指定相关参数。

```
   <param name="transform_configuration_param" value="param_value" />
</node>
```

这里，我们启动相应的 tf 变换。同样，替换相应的 pkg, type, name，并根据实际指定相关参数。

4.3 配置代价地图 (local_costmap) & (global_costmap)

导航功能包集需要两个代价地图来保存世界中的障碍物信息。一张代价地图用于规划，在整个环境中创建长期的规划，另一个用于局部规划与避障。有一些参数两个地图都需要，而有一些则各不相同。因此，对于代价地图，有三个配置项: common 配置项, global 配置项和 local 配置项。

注意: 接下来的内容只是代价地图的基本配置项。想要查看完整的配置，参看 costmap_2d 文档. http://wiki.ros.org/cn/costmap_2d

（1）共同配置(local_costmap) & (global_costmap)

导航功能包集使用代价地图存储障碍物信息。为了使这个过程更合理，我们需要指出要监听的传感器的话题，以更新数据。我们创建一个名为 costmap_common_params.yaml 的文件，内容如下：

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
inflation_radius: 0.55

observation_sources: laser_scan_sensor point_cloud_sensor

laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan, topic: topic_name, marking:
true, clearing: true}

point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud, topic: topic_name, markin
g: true, clearing: true}
```

现在我们分解以上代码。

```
obstacle_range: 2.5
raytrace_range: 3.0
```

这些参数设置放入代价地图的障碍信息的阈值。 “obstacle_range”参数决定了引入障碍物到代价地图的传感器读书的最大范围。 在这里,我们把它设定为 2.5 米,这意味着机器人只

会更新以其底盘为中心半径 2.5 米内的障碍信息。 “raytrace_range”参数确定的空白区域内光线追踪的范围。设置为 3.0 米意味着机器人将试图根据传感器读数清除其前面 3.0 米远的空间。

```
footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
#robot_radius: ir_of_robot
inflation_radius: 0.55
```

这里我们设置机器人的 footprint 或机器人半径（如果是圆形的）。指定的 footprint 时,机器人的中心被认为是在(0.0,0.0),顺时针和逆时针规范都支持。我们还将设置代价地图膨胀半径。膨胀半径应该设置为障碍物产生代价的最大距离。例如,膨胀半径设定在 0.55 米意味着机器人所有路径与障碍物保持 0.55 米或更的远离（具有同样的成本）。

```
observation_sources: laser_scan_sensor point_cloud_sensor
```

“observation_sources”参数定义了一系列传递空间信息给代价地图的传感器。每个传感器定义在下一行。

```
laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan, topic: topic_name, marking:
true, clearing: true}
```

这一行设置“observation_sources”中提到的传感器。这个例子定义了 laser_scan_sensor。“frame_name”参数应设置为传感器坐标帧的名称，“data_type”参数应设置为 LaserScan 或 PointCloud，这取决于主题使用的消息，“topic_name”应该设置为发布传感器数据的主题的名称。 “marking”和“clearing”参数确定传感器是否用于向代价地图添加障碍物信息，或从代价地图清除障碍信息，或两者都有。

（2）全局配置(global_costmap)

下面我们将创建一个存储特定的全局代价地图配置选项的文件。新建一个文件：global_costmap_params.yaml 并粘贴以下内容：

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
```

“global_frame”参数定义了代价地图运行所在的坐标帧。在这种情况下,我们会选择/map frame 。 “robot_base_frame”参数定义了代价地图参考的的机器地毯的坐标帧。“update_frequency”参数决定了代价地图更新的频率。 “static_map”参数决定代价地图是否根据 map_server 提供的地图初始化。如果你不使用现有的地图，设为 false。

（3）本地配置(local_costmap)

下面我们将创建一个存储特定的本地代价地图配置选项的文件。新建一个文件：localal_costmap_params.yaml 并粘贴以下内容：

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

“global_frame”，“robot_base_frame”，“update_frequency”，“static_map”参数与全局配置意义相同。“publish_frequency”参数决定了代价地图发布可视化信息的频率。将“rolling_window”参数设置为 true，意味着随着机器人在限时世界里移动，代价地图会保持以机器人为中心。“width”、“height”,“resolution”参数分别设置代价地图的宽度(米、)高度(米)和分辨率(米/单元)。 注意，这里的分辨率和你的静态地图的分辨率可能不同，但我们通常把他们设成一样的。

以上是用于启动和运行的最简单的配置，完整的配置选项，更多的细节请参阅 costmap_2d 文档. http://wiki.ros.org/cn/costmap_2d

4.4 Base Local Planner 配置

Base_local_planner 负责根据高层规划计算速度命令并发送给机器人基座。我们需要根据我们的机器人规格配置一些选项使其正常启动与运行。新建一个名为 base_local_planner_params.yaml 的文件，内容如下：

注意：本节只涵盖 TrajectoryPlanner 的基本配置选项。文档的全部选项,请参阅 base_local_planner 文档. http://wiki.ros.org/cn/base_local_planner

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4

  acc_lim_theta: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: true
```

上面的第一部分参数定义机器人的速度限制。第二部分定义了机器人的加速度的限制。

4.5 为导航功能包创建一个 Launch 启动文件

现在我们已经有了所有的配置文件，我么需要在一个启动文件中一起启动他们，创建一个名为 move_base.launch 的文件，内容如下：

```
<launch>

   <master auto="start"/>
 <!-- Run the map server -->
    <node name="map_server" pkg="map_server" type="map_server" args="$(find my_map_package)/my_m
ap.pgm my_map_resolution"/>

 <!--- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_omni.launch" />

   <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load" ns="g
lobal_costmap" />
    <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load" ns="l
ocal_costmap" />
    <rosparam file="$(find my_robot_name_2dnav)/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find my_robot_name_2dnav)/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find my_robot_name_2dnav)/base_local_planner_params.yaml" command="load" /
>
 </node>

</launch>
```

唯一需要修改的地方是更改地图服务器使指向你的已有的地图，并且，如果你有一台差分驱动的机器人，将"amcl_omni.launch"改为"amcl_diff.launch"。对于如何创建一张地图，请

查阅创建一张地图. http://wiki.ros.org/cn/slam_gmapping/Tutorials/MappingFromLoggedData

4.6 AMCL 配置(amcl)

AMCL 有许多配置选项将影响定位的性能。有关 AMCL 的更多信息请参阅 amcl 文档. http://wiki.ros.org/cn/amcl

5、运行导航功能包集

现在我们配置结束，我们可以运行导航功能包了。为此我们需要在机器人上启动两个终端。在一个终端上,我们将启动 my_robot_configuration.launch 文件，在另一个终端上我们将启动我们刚刚创建的 move_base.launch。

终端 1:

```
roslaunch my_robot_configuration.launch
```

终端 2:

```
roslaunch move_base.launch
```

祝贺你,导航功能包集现在应该运行了。关于如何通过图形化界面给导航功能包集发送一个目标信息,请参阅 rviz 和导航教程。

http://wiki.ros.org/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack

如果你想使用代码给导航功能包集发送导航目标,请参阅发送简单导航目标教程。

http://wiki.ros.org/navigation/Tutorials/SendingSimpleGoals