ROS 与 navigation 教程

说明：

- 介绍如何为机器人整合导航包，实现有效控制和自主导航等功能

目录：

- ROS 与 navigation 教程-carrot_planner 介绍
- ROS 与 navigation 教程-teb_local_planner 介绍
- ROS 与 navigation 教程-dwa_local_planner（DWA）介绍
- ROS 与 navigation 教程-nav_core 介绍
- ROS 与 navigation 教程-robot_pose_ekf 介绍
- ROS 与 navigation 教程-amcl 介绍
- ROS 与 navigation 教程-move_slow_and_clear 介绍
- ROS 与 navigation 教程-clear_costmap_recovery 介绍
- ROS 与 navigation 教程-rotate_recovery 介绍
- ROS 与 navigation 教程-costmap_2d 介绍
- ROS 与 navigation 教程-costmap_2d-range_sensor_layer 介绍
- ROS 与 navigation 教程-costmap_2d-social_navigation_layers 介绍
- ROS 与 navigation 教程-costmap_2d-staticmap 介绍
- ROS 与 navigation 教程-costmap_2d-inflation 介绍
- ROS 与 navigation 教程-obstacle 层介绍
- ROS 与 navigation 教程-Configuring Layered Costmaps

参考:

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

ROS 与 navigation 教程-设置机器人使用 TF

说明：

- 介绍如何配置机器人让其使用 TF
- 注意：本教程假设您已完成 ROS 教程和 tf 基础教程。
- 本教程的代码在 robot_setup_tf_tutorial 包中可用

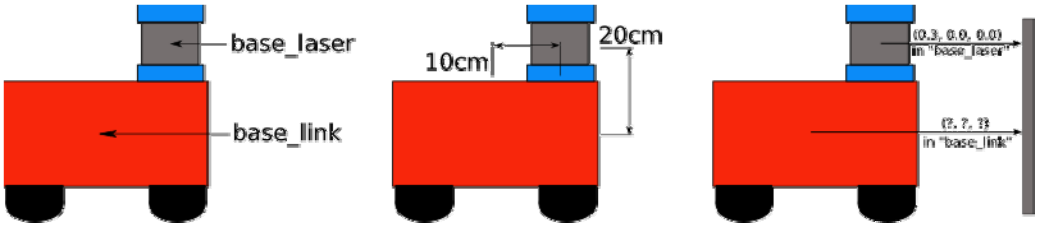
步骤：

(1)变换配置

许多 ROS 包需要使用 tf 软件库发布机器人的变换树。在抽象层，变换树根据不同坐标系之间的平移和旋转来定义偏移量。为了使这更具体，考虑一个简单的机器人的例子，它具有安装在其顶部的单个激光器的移动基座。在提及机器人时，我们定义两个坐标系：一个对应于机器人底座的中心点，一个坐标系安装在基座顶部的激光中心点。我们还给他们名字，以供参考。我们将调用附加到移动基础“base_link”的坐标系（对于导航，它重要的是将其放置在机器人的旋转中心），我们也会调用附加到激光“base_laser”的坐标系。

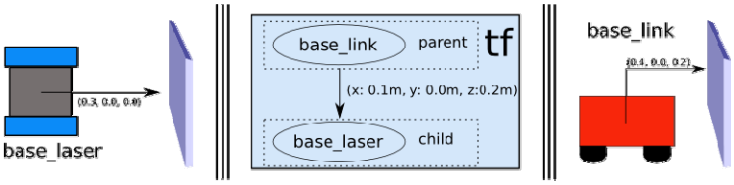
有关框架命名约定，请参见 REP 105

在这一点上，我们假设激光器的一些数据以与激光中心点的距离的形式来表达。换句话说，我们在“base_laser”坐标系中有一些数据。现在假设我们想要获取这些数据并使用它来帮助移动基地避免活动空间的障碍。要做到这一点，我们需要一种将我们从“base_laser”坐标系转换成“base_link”坐标系的激光扫描的方法。实质上，我们需要定义“base_laser”和“base_link”坐标系之间的关系。



在定义这种关系时，假设我们知道激光器在移动基座的中心点之上 10 厘米和向前 20 厘米处安装。这给了我们一个将“base_link”坐标系与“base_laser”坐标系相关联的平移值和偏移值。具体来说，我们知道要从“base_link”坐标系到“base_laser”坐标系的数据，我们必须应用（x：0.1m，y：0.0m，z：0.2m）的转换，并从“ base_laser“坐标系到”base_link“坐标系，我们必须应用相反的变换（x：-0.1m，y：0.0m，z：-0.20m）。我们可以选择自己管理这种关系，这意味着在必要时存储和应用帧之间的适当变换，但随着坐标系的数量增加，这成为一个真正的痛苦。幸运的是，我们不用自己做这个工作。相反，我们将使用 tf 定义“base_link”和“base_laser”之间的关系，并让它管理两个坐标系之间的变换。

要使用 tf 定义和存储“base_link”和“base_laser”坐标系之间的关系，我们需要将它们添加到一个变换树中。在概念上，变换树中的每个节点对应于坐标系，每个方向对应于需要应用从当前节点移动到其子节点的变换。Tf 使用树结构来确保只有一个遍历将任何两个坐标系连接在一起，并假设树中的所有方向都是从父节点引导到子节点。



要为我们的简单示例创建一个变换树，我们将创建两个节点，一个用于“base_link”坐标系，一个用于“base_laser”坐标系。为了创建它们之间的方向，我们首先需要决定哪个节点是父节点，哪些节点是子节点。记住，这种区别很重要，因为 tf 假定所有转换都从父对象移动到子对象。让我们选择“base_link”坐标系作为父节点，因为随着其他部分/传感器被添加到机器人，通过遍历“base_link”框架，它们将最有意义地与“base_laser”坐标系相关。这意味着连接“base_link”和“base_laser”的变换应为（x：0.1m，y：0.0m，z：0.2m）。使用这个变换树设置，将在“base_laser”坐标系中接收到的激光扫描转换为“base_link”坐标系就像调用 tf 库一样简单。我们的机器人可以使用这些信息来理解“base_link”框架中的激光扫描，并安全地计划避开环境中的障碍物。

(2)编写代码

希望上面的例子有助于在概念层面了解 tf。现在，我们必须使用变换树并用代码创建它。在这个例子中，假设你已熟悉 ROS，因此如果任何术语或概念不熟悉，请确保查看 ROS 文档。

假设我们有一个高级别的任务描述在“base_laser”坐标系中获取点并将其转换为“base_link”坐标系。我们需要做的第一件事是创建一个负责在系统中发布转换的节点。接下来，我们必须创建一个节点来监听通过 ROS 发布的变换数据，并应用它来转换一个点。我们将首先为源代码创建一个包，然后我们给它一个简单的名字，如“robot_setup_tf”我们将依赖于 roscpp，tf 和 geometry_msgs。

```
$ cd %TOP_DIR_YOUR_CATKIN_WS%/src
$ catkin_create_pkg robot_setup_tf roscpp tf geometry_msgs
```

请注意，您必须运行上面的命令（例如~/ros，您可能已为之前的教程创建）。

在 fuerte，groovy 和 hydro 中的替代方法：在 navigation_tutorials 包中有一个标准的 robot_setup_tf_tutorial 包。您可能希望通过以下方式安装 (%YOUR_ROS_DISTRO% can be { fuerte, groovy } .)：

```
$ sudo apt-get install ros-%YOUR_ROS_DISTRO%-navigation-tutorials
```

(3)广播变换

现在我们有了我们的软件包，我们需要创建一个节点来完成在 ROS 上广播 base_laser → base_link 变换的工作。在刚刚创建的 robot_setup_tf 包中，启动您喜欢的编辑器，并将以下代码粘贴到 src/tf_broadcaster.cpp 文件中。

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "robot_tf_publisher");
  ros::NodeHandle n;

  ros::Rate r(100);

  tf::TransformBroadcaster broadcaster;

  while(n.ok()){
    broadcaster.sendTransform(
      tf::StampedTransform(
        tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.2)),
        ros::Time::now(),"base_link", "base_laser"));
    r.sleep();
  }
}
```

现在，我们来更详细看看发布 base_link → base_laser 变换相关的代码

```
#include <tf/transform_broadcaster.h>
```

tf 包提供了一个 tf::TransformBroadcaster 的实现，以帮助使发布变换的任务更容易。要使用 TransformBroadcaster，我们需要包含 tf/transform_broadcaster.h 头文件。

```
tf::TransformBroadcaster broadcaster;
```

在这里，我们创建一个 TransformBroadcaster 对象，我们稍后将通过网络发送 base_link → base_laser 变换。

```
broadcaster.sendTransform(
  tf::StampedTransform(
    tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1, 0.0, 0.2)),
    ros::Time::now(),"base_link", "base_laser"));
```

这是实际工作完成的地方。使用 TransformBroadcaster 发送变换需要五个参数。首先，我们传递旋转变换，它由 btQuaternion 指定，需要在两个坐标系之间进行任何旋转。在这种情况下，我们不要施加旋转，所以我们发送一个由俯仰，滚动和偏航值等于零的 btQuaternion。第二，btVector3 可以应用到任何变换的，所以我们创建一个 btVector3 对应的激光的 x 偏移 10 厘米和 Z 偏移距离机器人基座 20 厘米。第三，我们需要给出发布的时间戳，我们将用 ros::Time::now()。第四，我们需要传递我们创建的链接的父节点的名称，在这种情况下是“base_link”。第五，我们需要传递我们创建的链接的子节点的名称，在这种情况下是“base_laser”。

(4)使用变换

以上，我们创建了一个通过 ROS 发布 base_laser → base_link 变换的节点。现在，我们要编写一个节点，将使用该变换在“base_laser”坐标系中取一点，并将其转换为“base_link”坐标系中的一个点。我们将首先将下面的代码粘贴到一个文件中，并进行更详细的解释。在 robot_setup_tf 包中，创建一个名为 src/tf_listener.cpp 的文件，并粘贴以下内容：

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>

void transformPoint(const tf::TransformListener& listener){
  //we'll create a point in the base_laser frame that we'd like to transform to the base_l
ink frame
  geometry_msgs::PointStamped laser_point;
  laser_point.header.frame_id = "base_laser";

  //we'll just use the most recent transform available for our simple example
  laser_point.header.stamp = ros::Time();

  //just an arbitrary point in space
  laser_point.point.x = 1.0;
  laser_point.point.y = 0.2;
  laser_point.point.z = 0.0;

  try{
    geometry_msgs::PointStamped base_point;
    listener.transformPoint("base_link", laser_point, base_point);
```

```
    ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f, %.2f, %.2f) at time
 %.2f",
        laser_point.point.x, laser_point.point.y, laser_point.point.z,
        base_point.point.x, base_point.point.y, base_point.point.z, base_point.header.sta
mp.toSec());
  }
  catch(tf::TransformException& ex){
    ROS_ERROR("Received an exception trying to transform a point from \"base_laser\" to \"
base_link\": %s", ex.what());
  }
}

int main(int argc, char** argv){
  ros::init(argc, argv, "robot_tf_listener");
  ros::NodeHandle n;

  tf::TransformListener listener(ros::Duration(10));

  //we'll transform a point once every second
  ros::Timer timer = n.createTimer(ros::Duration(1.0), boost::bind(&transformPoint, boos
t::ref(listener)));

  ros::spin();

}
```

现在我们将一步一步解释

```
#include <tf/transform_listener.h>
```

在这里，我们包含 tf/transform_listener.h 头文件，我们需要创建一个 tf::TransformListener。一个 TransformListener 对象自动订阅了 ROS 变换消息主题和管理所有将在网络中广播的变换数据。

```
void transformPoint(const tf::TransformListener& listener){
```

我们将创建一个函数，给定一个 TransformListener，在“base_laser”坐标系中取一点，并将其转换为“base_link”坐标系。这个函数将作为在我们程序的 main()中创建的 ros::Timer 的回调，并且每秒都会触发。

```
 //we'll create a point in the base_laser frame that we'd like to transform to the base_li
nk frame
  geometry_msgs::PointStamped laser_point;
  laser_point.header.frame_id = "base_laser";
```

```
  //we'll just use the most recent transform available for our simple example
  laser_point.header.stamp = ros::Time();

  //just an arbitrary point in space
  laser_point.point.x = 1.0;
  laser_point.point.y = 0.2;
  laser_point.point.z = 0.0;
```

在这里，我们将创建一个 geometry_msgs::PointStamped 的点。消息名称结尾处的“Stamped”只是意味着它包含一个头，允许我们将时间戳和 frame_id 与消息相关联。我们将 laser_point 消息的 Stamp 字段设置为 ros::Time()，它是一个特殊的时间值，允许我们向 TransformListener 询问最新的可用变换。对于标题的 frame_id 字段，我们将把它设置为“base_laser”，因为我们在“base_laser”坐标系中创建一个点。最后，我们将设置一些数据，例如 x：1.0，y：0.2 和 z：0.0 的点。

```
try{
    geometry_msgs::PointStamped base_point;
    listener.transformPoint("base_link", laser_point, base_point);

    ROS_INFO("base_laser: (%.2f, %.2f. %.2f) -----> base_link: (%.2f, %.2f, %.2f) at time
 %.2f",
        laser_point.point.x, laser_point.point.y, laser_point.point.z,
        base_point.point.x, base_point.point.y, base_point.point.z, base_point.header.sta
mp.toSec());
  }
```

现在我们将“base_laser”坐标系中的点转换成“base_link”坐标系的点。为此，我们将使用 TransformListener 对象，调用 transformPoint()，并使用三个参数：我们要将点转换为（“base_link”在我们的例子中）坐标系的名称，我们正在转换的点，以及存储变换点。所以在调用 transformPoint()之后，base_point 保存与 laser_point 相同的信息，包含变换后的信息。

catch(tf::TransformException& ex){

ROS_ERROR("Received an exception trying to transform a point from "base_laser" to "base_link": %s",

ex.what());

}

如果由于某种原因 base_laser → base_link 变换不可用（也许 tf_broadcaster 没有运行），当我们调用 transformPoint()时，TransformListener 可能会引发异常。为了确保我们优雅地处理这个问题，我们将捕获异常，并为用户打印一个错误。

(5)编译代码

现在我们已经写了一个例子，我们需要构建它。打开由 roscreate-pkg 自动生成的 CMakeLists.txt 文件，并将以下行添加到文件的底部。

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)
add_executable(tf_listener src/tf_listener.cpp)
target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})
target_link_libraries(tf_listener ${catkin_LIBRARIES})
```

接下来，确保保存文件并构建包。

```
$ cd %TOP_DIR_YOUR_CATKIN_WS%
$ catkin_make
```

(6)运行代码

让我们来看看 ROS 实际发生了什么。对于本节，您将需要打开三个终端。

- 新开终端，运行 roscore。

```
roscore
```

- 新开终端，运行 tf_broadcaster

```
rosrun robot_setup_tf tf_broadcaster
```

- 新开终端，运行 tf_listener，将我们的模拟点从“base_laser”坐标系变换为“base_link”坐标系一个点

```
rosrun robot_setup_tf tf_listener
```

- 如果一切正常，您应该看到以下输出显示一个点从“base_laser”坐标系转换到“base_link”坐标系。

```
[ INFO] 1248138528.200823000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.2
0, 0.20) at time 1248138528.19
[ INFO] 1248138529.200820000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.2
0, 0.20) at time 1248138529.19
[ INFO] 1248138530.200821000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.2
0, 0.20) at time 1248138530.19
```

```
[ INFO] 1248138531.200835000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.2
0, 0.20) at time 1248138531.19
[ INFO] 1248138532.200849000: base_laser: (1.00, 0.20. 0.00) -----> base_link: (1.10, 0.2
0, 0.20) at time 1248138532.19
```

恭喜，您刚刚为平面激光器编写了一个成功的变换广播。下一步是将我们用于此示例的 PointStamped 替换为 ROS 中的传感器流。

ROS 与 navigation 教程-基本导航调整指南

说明：

- 介绍如何调整机器人上的 ROS 导航包

步骤：

(1) 机器人导航需要那些准备?

在调整新机器人上的导航包时遇到的大部分问题都在本地规划器调谐参数之外的区域。机器人的里程计，定位，传感器以及有效运行导航的其他先决条件常常会出错。所以，我做的第一件事是确保机器人本身正在准备好导航。这包括三个组件检查：距离传感器，里程计和定位。

- 距离传感器

如果机器人没有从其距离传感器（例如激光器）获取信息，那么导航将不起作用。我将确保我可以在 rviz 中查看传感器信息，它看起来相对正确，并以预期的速度进入。

- 里程计

通常我会很难使机器人正确定位。它将不断迷失，我会花费大量的时间调试 AMCL 的参数，发现真正的罪魁祸首是机器人的测距。因此，我总是运行两个健全检查，以确保我相信机器人的里程计。

第一个测试检查角速度是否合理。我打开 rviz，将坐标系设置为“odom”，显示机器人提供的激光扫描，将该主题的衰减时间设置为高（类似 20 秒），并执行原地旋转。然后，我看看扫描出来的边线在随后的旋转中如何相互匹配。理想情况下，每次扫描将刚好落在相互的顶端，会重叠在一起，但是有些旋转漂移是预期的，所以我只是确保扫描之间误差，不会超过一度或两度以上。

第二个测试检查线速度是否合理。机器人放置在与距离墙壁几米远地方，然后以上面相同的方式设置 rviz。接着我将驱动机器人向墙壁前进，从 rviz 中聚合的激光扫描看看扫描出来边线的厚度。理想情况下，墙体应该看起来像一个扫描，但我只是确保它的厚度不超过几厘米。如果显示扫描边线的分散在半米以上，但有些可能是错误的测距。

其他的测试角速和线速方法：

1. 线速标定：http://www.ncnynl.com/archives/201701/1217.html
2. 角速标定：http://www.ncnynl.com/archives/201701/1218.html
3. 线速标定：http://www.ncnynl.com/archives/201707/1812.html
4. 角速标定：http://www.ncnynl.com/archives/201707/1813.html

- 定位

假设里程计和激光扫描仪都能合理地执行，建图和调整 AMCL 通常并不会太糟糕。首先，我将运行 gmapping 或 karto，并操纵机器人来生成地图。然后，我将使用该地图与 AMCL，并确保机器人保持定位。如果我运行的机器人的距离不是很好，我会用 AMCL 的测距模型参数玩一下。对整个系统的一个很好的测试是确保激光扫描和地图可以在 rviz 的“地图”坐标系中可视化，并且激光扫描与环境地图很好地匹配。

(2)代价地图

- 一旦我的机器人满足导航的先决条件，我想确保代价地图的设置和配置正确。
- 我会在如何配置 ROS 导航教程和 costmap_2d 文档详细说明，但是我会给出一些我经常做的事情的提示。
- 如下建议可用来调整代价地图：
  - 确保根据传感器实际发布的速率为每个观测源设置 expected_update_rate 参数。我通常在这里给出相当的容忍度，把检查的期限提高到我期望的两倍，但是当传感器低于预期速率时，它很容易从导航中接收警告。
  - 为系统适当设置 transform_tolerance 参数。查看使用 tf 从“base_link”坐标系到“map”坐标系的转换的预期延迟。我通常使用 tf_monitor 查看系统的延迟，并将参数保守地设置为关闭。另外，如果 tf_monitor 报告的延迟足够大，我可能会随时看看导致延迟的原因。这有时会导致我发现关于给定机器人的变换如何发布的问题。

- 在缺乏处理能力的机器人上，我会考虑关闭 map_update_rate 参数。然而，在这样做时，我考虑到这将导致传感器数据快速进入代价地图的延迟，这反过来会降低机器人对障碍物的反应速度。
- 该 publish_frequency 参数是在 rviz 可视化 costmap 有用。然而，特别是对于大型全局地图，该参数可能导致事情运行缓慢。在生产系统中，我考虑降低成本图发布的速度，当我需要可视化非常大的地图时，我确定设置速度真的很低。
- 是否对代价地图使用 voxel_grid 或 costmap 模型的决定在很大程度上取决于机器人具有的传感器套件。调整代价地图为基于 3D-based 代价地图更多是涉及未知空间的考虑。如果我正在使用的机器人只有一个平面激光，我总是使用 costmap 模型的地图。
- 有时，它只能在里程坐标系中运行导航。要做到这一点，我发现最容易做的事情就是我的复制 local_costmap_params.yaml 文件覆盖 global_costmap_params.yaml 文件并更改了地图宽度和高度比如 10 米。如果需要独立的定位性能来调整导航，这是一个简单易行的方法

- 我倾向于根据机器人的尺寸和处理能力选择我使用的地图的分辨率。在具有很多处理能力并需要适应狭窄空间的机器人，如 PR2，我会使用细粒度的地图...将分辨率设置为 0.025 米。对于像 roomba 这样的东西，我可能会以高达 0.1 米的分辨率去降低计算量。
- Rviz 是验证代价地图正常工作的好方法。我通常从代价地图中查看障碍物数据，并确保在操纵杆控制下驱动机器人时，它与地图和激光扫描相一致。这是对传感器数据以合理的方式进入代价地图的合理检查。如果我决定用机器人跟踪未知的空间，主要是这些机器人正在使用 voxel_grid 模型的代价地图，我一定要看看未知的空间可视化，看看未知的空间被以合理的方式清除。是否正确地从代价地图中清除障碍物的一个很好的检查方法是简单地走在机器人的前面，看看它是否都成功地看到我，并清除我。

- 当导航包仅运行 costmap 时，检查系统负载是一个好主意。这意味着提高 move_base 节点，但不会发送目标点并查看负载。如果计算机在这个时候陷入僵局，我知道如果我想要运行规划器的话，我需要做一些 CPU 参数调整。

(3) 局部规划器

如果通过代价地图的操作令人满意，我将继续调整局部规划器参数。在具有合理加速度限制的机器人上，我通常使用 dwa_local_planner，对于那些具有较低加速度限制的机器人，并且可以从每个步骤考虑到加速限制的，我将使用 base_local_planner。调整 dwa_local_planner 比调整 base_local_planner 更为愉快，因为它的参数是动态可配置的。为导航包添加 dynamic_reconfigure 也已经在计划中。针对规划器的提示：

- 对于给定的机器人最重要的是，正确设置了加速度限制参数。如果这些参数关闭，只能期望来自机器人的次优行为。如果我不知道机器人的加速度极限是什么，我会花点时间写出一个脚本，让电机以最大平移和旋转速度命令运行一段时间，看看报告的里程计速度（假设里程会给出合理的估计），并从中得出加速度极限。合理设置这些参数可以节省很多时间。
- 如果我正在使用的机器人具有低加速度限制，我确保我正在运行 base_local_planner，其中 dwa 设置为 false。将 dwa 设置为 true 后，我还将确保根据处理能力将 vx_samples 参数更新为 8 到 15 之间的值。这将允许在展示中生成非圆形曲线。
- 如果我正在使用的机器人的定位并不是很好，我将确保将目标公差参数设置得比我想象的要高一些。如果机器人具有较高的最小旋转速度以避免在目标点的振荡，那么我也将提高旋转公差。
- 如果我使用低分辨率的 CPU 原因，我有时会提高 sim_granularity 参数，以节省一些周期。
- 我实际上很少发现自己改变了 path_distance_bias 和 goal_distance_bias（对于 base_local_planner 这些参数被称为 pdist_scale 和 gdist_scale）参数在计划者上非常多。当我这样做时，通常是因为我试图限制本地规划器自由，让计划的路径与除 NavFn 以外的全局规划器合作。将 path_distance_bias 参数增大，将使机器人更紧密地跟随路径，同时以快速向目标向前移动。如果这个值设置得太高，机器人将会拒绝移动，因为移动的代价大于停留在路径上的位置。简单来说就是让实际移动更接近全局路径还是本地路径。
- 如果我想以聪明的方式介绍代价函数，我将确保将 meter_scoring 参数设置为 true。这使得它在代理函数中的距离以米而不是单元格，也意味着我可以调整一个地图分辨率的代价函数，并且当我移动到他人时期望合理的行为。此外，您现在可以通过将 publish_cost_grid 参数设置为 true 来显示本地计划程序在 rviz 中生成的代价函数。（这不管怎么说，从来没有把它放入文档，我会很快到那个时候）。考虑到以米为单位的代价函数，我可以计算出移动 1 米的成本与目标平衡的成本的折衷。这倾向于给我一个如何调整东西的体面的想法。
- 轨迹从其端点得分。这意味着将 sim_time 参数设置为不同的值可能对机器人的行为有很大的影响。我通常将此参数设置在 1-2 秒之间，其中设置更高可以导致稍微平滑的轨迹，确保乘以 sim_period 的最小速度小于目标的两倍。否则，机器人将倾向于在其目标位置的范围之外的位置旋转，而不是朝向目标移动。

- 精确的轨迹模拟也取决于距离测距的合理速度估计。这来自于 dwa_local_planner 和 base_local_planner 都使用这种速度估计以及机器人的加速度限制来确定规划周期的可行速度空间。虽然来自测距的速度估计不一定是完美的，但确保其至少接近得到最佳行为是重要的。

ROS 导航调整指南

如果您的机器人正在导航准备就绪，而您即将通过优化机器人的导航行为的过程，此处是由郑开宇创建的 ROS 导航调整指南。它讨论了包括设置速度和加速度的组件，全局规划器，本地规划器（特别是 DWA 本地规划），代价地图，AMCL（定位）， recovery behaviors 等。这是一个用于改进 SCITOS G5 机器人导航系统的视频演示，基于本指南中提出的想法。本指南绝对不完整（如 AMCL 部分需要更多讨论），并可能包含错误。如有问题，随时向这个 Github Repository 提交问题或提出请求！

参考：

- https://github.com/zkytony/ROSNavigationGuide/blob/master/main.pdf

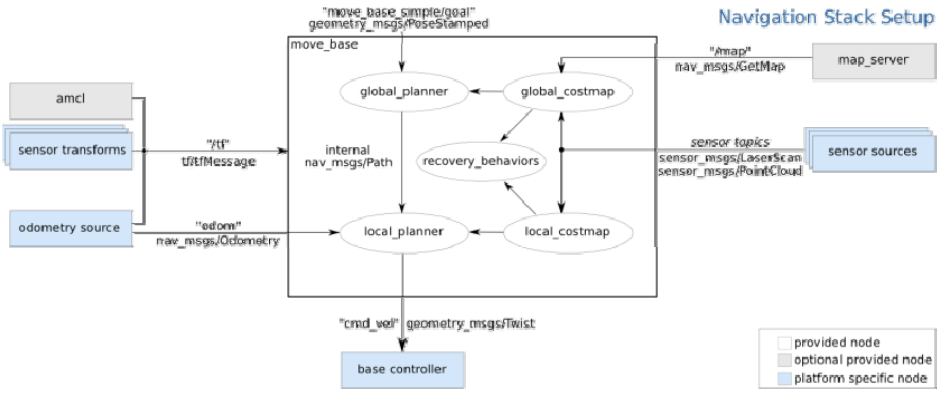
ROS 与 navigation 教程-安装和配置导航包

说明：

- 介绍如何在机器人整合导航包。包括使用 TF 发送变换，发布里程信息，发布传感器数据以及基本导航包配置。

(1)机器人配置：

- 机器人配置图



为使用导航包需要机器人以特定方式配置。上图显示了此配置的概述。白色组件是已实现的必需组件，灰色组件是已实现的可选组件，必须为每个机器人平台创建蓝色组件。以下部分提供了导航包的先决条件以及如何满足每个要求的说明。

- ROS

导航堆栈假设机器人正在使用 ROS。有关如何在机器人上安装 ROS 的说明，请参阅 ROS 文档。

- 变换配置（其他变换）

导航包要求机器人使用 tf 发布关于坐标系之间的关系的信息。有关设置此配置的详细教程，请参见：变换配置

- 传感器信息（传感器源）

导航包使用来自传感器的信息来避开地图上的障碍物，它假定这些传感器在 ROS 上发布 sensor_msgs/LaserScan 或 sensor_msgs/PointCloud 消息。有关通过 ROS 发布这些消息的信息，请参阅“ROS 发布传感器流”教程。此外，还有一些传感器已经有相关处理的 ROS 驱动程序。

- 支持的传感器及其相应驱动程序的链接如下所示：
  - 符合 SCIP2.2 标准的 Hokuyo 激光设备以及 Hokuyo 型号 04LX，30LX- urg_node
  - SICK LMS2xx 激光器 - sicktoolbox_wrapper

- 里程信息（里程计源）

导航包需要使用 TF 和 nav_msgs/Odometry 消息来发布里程信息。有关发布测距信息的教程可以在这里找到：在 ROS 上发布测距信息。

- odometry 支持的平台及其相应驱动程序的链接如下所示：
  - Videre Erratic: erratic_player
  - PR2: pr2_mechanism_controllers

- 基本控制器（基本控制器）

导航包假设它向机器人的“cmd_vel”话题，发送 geometry_msgs/Twist 类型的消息速度命令。这意味着必须有一个订阅“cmd_vel”话题的节点能够执行 (vx, vy, vtheta) <==> (cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z) 的速度并将其转换为电机命令以发送到移动基站。

- 支持的基础控制平台及其相应驱动程序的链接如下：
  - Videre Erratic: erratic_player

- PR2: pr2_mechanism_controllers

- 建图（map_server）

导航包不需要地图操作，但为了本教程的目的，我们假设您有一个。有关创建您的环境地图的详细信息，请参阅构建地图教程。

(2)导航包设置

假定上述机器人设置的所有要求已经得到满足。这意味着机器人必须使用 TF 发布坐标系信息，接收所有能被导航包使用的传感器 sensor_msgs/LaserScan 或 sensor_msgs/PointCloud 的消息，并使用 tf 和 nav_msgs/Odometry 发布里程消息，同时也发布速度命令到移动基座。如果您的机器人中没有满足这些要求，请参阅上面的“机器人配置”部分，了解完成它们的说明。

- 创建包

本教程的第一步是创建一个包，其中我们将存储导航包的所有配置和启动文件。此软件包将依赖于用于满足上述“机器人配置”部分以及包含导航包高级接口的 move_base 软件包的任何软件包。因此，创建命令为：

```
catkin_create_pkg my_robot_name_2dnav move_base my_tf_configuration_dep my_odom_configura
tion_dep my_sensor_configuration_dep
```

此命令将创建一个具有必要依赖关系的包，以便在机器人上运行导航包。

- 创建机器人配置启动文件

现在我们有一个工作空间用于我们所有的配置和启动文件，我们将创建一个 roslaunch 文件，它将启动所有硬件和发布机器人需要 TF 变换。启动您最喜欢的编辑器，并将以下代码段粘贴到名为 my_robot_configuration.launch 的文件中。当然，您应该随意将文本“my_robot”替换为实际机器人的名称。我们还必须对启动文件进行类似的更改，如下所述，因此请确保您阅读本节的其余部分。

```
<launch>
  <node pkg="sensor_node_pkg" type="sensor_node_type" name="sensor_node_name" output="scr
een">
    <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="trans
form_configuration_name" output="screen">
    <param name="transform_configuration_param" value="param_value" />
  </node>
</launch>
```

现在我们有一个启动文件的模板，但我们需要填写我们的特定机器人。我们将在下面的每一部分中详细介绍需要做的更改

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

在本节中，我们将介绍机器人用于导航的任何传感器。将“sensor_node_pkg”替换为传感器的 ROS 驱动程序的包名称，传感器驱动程序类型“sensor_node_type”，传感器节点所需名称的“sensor_node_name”以及带有任何参数的“sensor_param”您的节点可能需要。请注意，如果您有多个传感器用于将信息发送到导航包，则应在此处启动所有传感器。

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

在本部分中，我们将为基座启动里程计。再次，您需要将 pkg，type，name 和 param 规范替换为与实际启动的节点名称。

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

在本节中，我们将为机器人启动变换配置。再次，您需要将 pkg，type，name 和 param 规范替换为与实际启动的节点名称。

- 代价地图配置(local_costmap) & (global_costmap)

导航包使用两个成本地图来存储有关地图的障碍物信息。一个成本地图用于全局规划，意味着在整个环境中创建长期计划，另一个用于本地规划和避障。有一些配置选项，我们希望两个代价地图都使用，一些我

们要在每种代价地图上单独设置。因此，下面有三个部分用于代价地图的配置：常用配置选项，全局配置选项和本地配置选项。

注意：以下部分仅涵盖了代价地图的基本配置选项。有关各种选项的文档，请参阅 costmap_2d 文档

- 通用配置 (local_costmap) & (global_costmap)

导航包使用代价地图存储有关障碍物的信息。为了正确执行此操作，我们需要为 costmaps 指定应该监听传感器主题和更新频率。我们创建一个名为 costmap_common_params.yaml 的文件，如下所示，并填写：

```
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, ma
rking: true, clearing: true}

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

让我们将上面的文件分解说明。

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

这些参数设置了代价地图中的障碍物信息的阈值。 "obstacle_range" 参数确定最大范围传感器读数，这将导致障碍物被放入代价地图中。在这里，我们把它设置在 2.5 米，这意味着机器人只会更新其地图包含距离移动基座 2.5 米以内的障碍物的信息。“raytrace_range”参数确定了用于清除指定范围外的空间。将其设置为 3.0 米，这意味着机器人将尝试清除 3 米外的空间。

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

在这里，我们设置机器人的占用面积或机器人的半径是圆形。在指定占地面积的情况下，假设机器人的中心处于（0.0,0.0），并且支持顺时针和逆时针规格。我们还将设定代价地图的膨胀半径。膨胀半径应设定

为在代价地图的与障碍物保持安全的最大距离。例如，将膨胀半径设定为 0.55 米，意味着机器人针对相同的障碍物获取的所有路径都保持距离障碍物 0.55 米或更远。

```
observation_sources: laser_scan_sensor point_cloud_sensor
```

“observe_sources”传感器源参数定义把信息传递给代价地图的传感器列表，以空格分隔每个传感器，每个传感器分别定义。

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

设置上述传感器源 observation_sources 的传感器信息。以 laser_scan_sensor 作为例子，应将“frame_name”参数设置为传感器坐标系的名称，“data_type”参数应根据话题使用的消息设置为 LaserScan 或 PointCloud，并将“topic_name”设置为传感器发布数据的话题。“marking”和“clearing”参数确定传感器是否向代价地图添加障碍物信息或从代价地图中清除障碍物信息，或者同时都做。

- 全局代价地图配置（global_costmap）

我们将在下面创建一个文件，用于存储特定于全局代价地图的配置选项。打开一个带有 global_costmap_params.yaml 文件的编辑器，并粘贴到以下文本中：

```
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应该运行的坐标系，在这种情况下，我们将选择/map 坐标系。“robot_base_frame”参数定义了代价地图应该为机器人的基座的坐标系。“update_frequency”参数决定了代价地图更新的频率（以 Hz 为单位）。该“static_map”参数确定是否由 map_server 提供的地图服务来进行代价地图的初始化。如果您没有使用现有的地图或地图服务器，请将 static_map 参数设置为 false。

- 本地代价地图配置（local_costmap）

我们将在下面创建一个文件，它将存储特定于本地代价地图的配置选项。打开一个带有文件 local_costmap_params.yaml 的编辑器并粘贴到以下文本中：

```
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”和“static_map”参数与上述“全局代价地图配置”部分中描述的相同。“publish_frequency”参数确定代价地图发布可视化信息的速率（以 Hz 为单位）。将“rolling_window”参数设置为 true 意味着当机器人移动时，保持机器人在本地代价地图中心。“宽度”，“高度”和“分辨率”参数设置本地代价地图的宽度（米），高度（米）和分辨率（米/单元格）。请注意，这个网格的分辨率与静态地图的分辨率不同，但大多数时候我们倾向设置为相同值。

- 完整的配置选项

此最低配置应该可以运行，但是有关可用于代价地图的配置选项的更多详细信息，请参阅 costmap_2d 文档。

- 基本本地规划器配置/base_local_planner

base_local_planner 负责把上层规划器计算的速度指令发送给移动基座。我们需要根据我们的机器人的规格来设置一些配置选项，以便开始运行。打开一个名为 base_local_planner_params.yaml 的文件，并粘贴以下文本：

```
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
```

上述参数的第一部分定义了机器人的速度限制。第二部分定义了机器人的加速度限制。

注意：本节仅介绍 TrajectoryPlanner 的基本配置选项。有关各种选项的文档，请参阅 base_local_planner 文档。

- 创建导航包的启动文件

现在我们已经有了所有的配置文件，我们需要把所有的东西放在导航包的启动文件中。打开一个带有 move_base.launch 文件的编辑器并粘贴以下文本：

```
<launch>
  <master auto="start"/>

  <!-- Run the map server -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find my_map_package)
/my_map.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="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load
" ns="global_costmap" />
    <rosparam file="$(find my_robot_name_2dnav)/costmap_common_params.yaml" command="load
" ns="local_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>
```

您需要对此文件进行的唯一更改是将地图服务器更改为指向已创建的地图，如果您有差分驱动器机器人，则将“amcl_omni.launch”更改为“amcl_diff.launch”。有关创建地图的教程，请参阅构建地图。

- AMCL 配置（amcl）

AMCL 有许多配置选项会影响本地化的性能。有关 AMCL 的更多信息，请参阅 amcl 文档。

- 运行导航包

现在我们已经完成了所有设置，我们可以运行导航包。为此，我们需要打开机器人的两个终端。在一个终端中，我们将启动 my_robot_configuration.launch 文件，另一个终端将启动我们刚创建的 move_base.launch 文件。

- 终端 1，执行

```
roslaunch my_robot_name_2dnav my_robot_configuration.launch
```

- 终端 2，执行

```
roslaunch my_robot_name_2dnav move_base.launch
```

恭喜，导航包现在应该正在运行。有关通过图形界面将目标发送到导航包的信息，请参阅 rviz 和导航教程。

如果要使用代码将目标发送到导航包，请参阅“ 发送简单导航目标”教程。

- 故障排除

对于运行导航堆栈时遇到的常见问题，请参阅导航堆栈疑难解答页面。

参考资料

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

ROS 与 navigation 教程-结合 RVIZ 与导航包

说明：

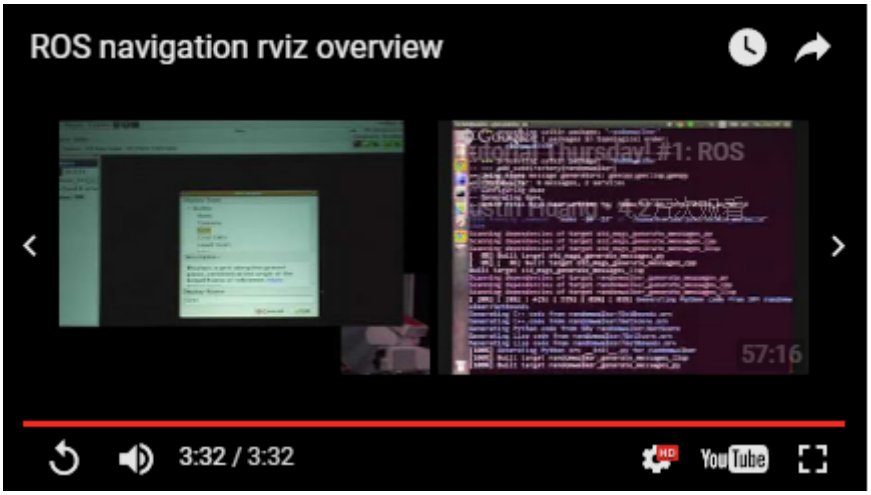
- 介绍如何结合 rviz 来使用导航包，包括初始化位态，将目标发送给机器人以及查看导航包可视化信息

概观：

- rviz 是一个强大的可视化工具，可以用于许多不同的目的。本教程至少对 rviz 进行了一些熟悉，可以在这里找到 rviz 文档。
- 设置 rviz

以下视频显示如何设置 rviz 以使用导航包。这包括为 amcl 定位系统设置机器人的姿态，显示导航包提供的所有可视化信息，以及使用 rviz 将目标发送到导航包。navstack 发布的每个可视化话题的讨论可以在下面找到。

- 截图：



- 视频：(此为 youtbe 视频，请翻墙查看)

- 2D 导航目标/2D Nav Goal
  - 话题：move_base_simple/goal
  - 类型：geometry_msgs/PoseStamped
  - 说明：允许用户设置机器人目标位姿，并发送到导航包。

- 2D 姿势估计/2D Pose Estimate
  - 话题：initialpose
  - 类型： geometry_msgs/PoseWithCovarianceStamped
  - 说明：允许用户设置机器人的初始化位姿。

- 静态地图/Static Map
  - 话题：map
  - 类型：nav_msgs/GetMap
  - 类型：nav_msgs/OccupancyGrid
  - 说明：显示由 map_server 提供的静态地图（如果存在）

- 粒子云/Particle Cloud
  - 话题：particlecloud
  - 类型：geometry_msgs/PoseArray

- 说明：显示机器人定位系统中使用的粒子云。粒子云的分布代表了定位系统对机器人姿态评估的不确定性。一个非常分散的粒子云层反映了很高的不确定性，而一个浓缩的粒子云代表着低的不确定性。换句话说越集中定位越准。

- 机器人本体/Robot Footprint
  - 话题：local_costmap/robot_footprint
  - 类型：geometry_msgs/PolygonStamped
  - 说明：显示机器人的占用空间

- 障碍/Obstacles
  - 话题：local_costmap/obstacles
  - 类型：nav_msgs/GridCells
  - 说明：显示导航包在其本地代价地图中看到的障碍。为了使机器人避免碰撞，机器人本体绝不应与包含障碍物的单元相交。

- 障碍膨胀区/Inflated Obstacles
  - 话题：local_costmap/inflated_obstacles
  - 类型：nav_msgs/GridCells
  - 说明：在导航包的代价地图中显示由机器人与障碍区之间一个内切半径范围的空间。为了机器人避免碰撞，机器人的中心点不应与障碍膨胀区重叠。

- 未知空间/Unknown Space
  - 话题：local_costmap/unknown_space
  - 类型：nav_msgs/GridCells
  - 说明：显示导航包的 costmap_2d 中包含的任何未知空间。

- 全局规划/Global Plan
  - 话题：TrajectoryPlannerROS/global_plan
  - 类型：nav_msgs/Path
  - 说明：代价地图中显示达到目标的全局规划路径

- 本地规划/Local Plan
  - 话题：TrajectoryPlannerROS/local_plan
  - 类型：nav_msgs/Path
  - 说明：代价地图中显示达到目标本地规划的路径。
- 规划器规划/Planner Plan
  - 话题：NavfnROS/plan
  - 类型：nav_msgs/Path
  - 说明：显示由全局规划器计算出机器人的完整规划。
- 当前目标/Current Goal
  - 话题：current_goal
  - 类型：geometry_msgs/PoseStamped
  - 说明：显示导航包尝试实现的目标位势

ROS 与 navigation 教程-发布里程计消息

说明：

- 介绍导航包发布里程信息的示例。
- 它涵盖了通过 ROS 发布 nav_msgs/Odometry 消息，以及从“odom”坐标系到 tf 上的“base_link”坐标系的变换。

(1)发布里程信息

导航包使用 TF 来确定机器人在地图中的位置，并将传感器数据与静态地图相关联。然而，TF 不提供关于机器人的速度的任何信息。因此，导航包要求任何里程源都通过 ROS 发布 TF 变换和 nav_msgs/Odometry

消息。本教程解释了 nav_msgs/Odometry 消息，并提供了用于发布消息并通过 ROS 和 tf 进行变换的示例代码。

(2)nav_msgs/Odometry 消息

nav_msgs/Odometry 消息存储了在自由空间中机器人的位置和速度的估计值：

```
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given by header.frame_id.
# The twist in this message should be specified in the coordinate frame given by the child_frame_id
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

该消息中的 pose 对应于机器人在坐标系中的估计位置以及用于该位态估计的确定性的可选协方差。该消息中的 twist 对应于机器人在子坐标系中的速度，通常是移动基站的坐标系，以及用于该速度估计的确定性的可选协方差。

(3)使用 tf 发布 Odometry 变换

如变换配置教程中所述，“TF”软件库负责管理与机器人相关的变换树的各坐标系之间的关系。因此，任何里程源都必须发布有关其管理的坐标系的信息。

(4)编写代码

在本节中，我们将编写一些用于通过 ROS 发布 nav_msgs/Odometry 消息的示例代码，并使用 tf 进行变换，这种变换只是在一个循环中驱动的仿真的机器人。我们首先将代码全部显示出来，下面是一个一个一个的解释。

将依赖关系添加到包的 manifest.xml 中

```
<depend package="tf"/>
<depend package="nav_msgs"/>
```

示例代码：

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
```

```
int main(int argc, char** argv){
  ros::init(argc, argv, "odometry_publisher");

  ros::NodeHandle n;
  ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);
  tf::TransformBroadcaster odom_broadcaster;

  double x = 0.0;
  double y = 0.0;
  double th = 0.0;

  double vx = 0.1;
  double vy = -0.1;
  double vth = 0.1;

  ros::Time current_time, last_time;
  current_time = ros::Time::now();
  last_time = ros::Time::now();

  ros::Rate r(1.0);
  while(n.ok()){

    ros::spinOnce();               // check for incoming messages
    current_time = ros::Time::now();

    //compute odometry in a typical way given the velocities of the robot
    double dt = (current_time - last_time).toSec();
    double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
    double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
    double delta_th = vth * dt;

    x += delta_x;
    y += delta_y;
    th += delta_th;

    //since all odometry is 6DOF we'll need a quaternion created from yaw
    geometry_msgs::Quaternion odom_quat = tf::createQuaternionMsgFromYaw(th);

    //first, we'll publish the transform over tf
    geometry_msgs::TransformStamped odom_trans;
    odom_trans.header.stamp = current_time;
    odom_trans.header.frame_id = "odom";
```

```
    odom_trans.child_frame_id = "base_link";

    odom_trans.transform.translation.x = x;
    odom_trans.transform.translation.y = y;
    odom_trans.transform.translation.z = 0.0;
    odom_trans.transform.rotation = odom_quat;

    //send the transform
    odom_broadcaster.sendTransform(odom_trans);

    //next, we'll publish the odometry message over ROS
    nav_msgs::Odometry odom;
    odom.header.stamp = current_time;
    odom.header.frame_id = "odom";

    //set the position
    odom.pose.pose.position.x = x;
    odom.pose.pose.position.y = y;
    odom.pose.pose.position.z = 0.0;
    odom.pose.pose.orientation = odom_quat;

    //set the velocity
    odom.child_frame_id = "base_link";
    odom.twist.twist.linear.x = vx;
    odom.twist.twist.linear.y = vy;
    odom.twist.twist.angular.z = vth;

    //publish the message
    odom_pub.publish(odom);

    last_time = current_time;
    r.sleep();
  }
}
```

让我们详细分解代码的重要部分

```
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
```

由于我们将一个从“odom”坐标系变换到“base_link”坐标系和 nav_msgs/Odometry 消息，我们需要包括相关的头文件。

```
ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);
tf::TransformBroadcaster odom_broadcaster;
```

我们需要创建一个 ros::Publisher 和一个 tf::TransformBroadcaster 来分别使用 ROS 和 tf 发送消息。

```
double x = 0.0;
double y = 0.0;
double th = 0.0;
```

我们假设机器人起始于“odom”坐标系的起点

```
double vx = 0.1;
double vy = -0.1;
double vth = 0.1;
```

这里我们将设置一些速度，这将导致“base_link”坐标系在“odom”坐标系中以 x 方向 0.1m/s 速度，y 方向为 -0.1m/s 速度，th 方向为 0.1rad/s 弧度移动。这或多或少会导致我们的仿真机器人兜一圈。

```
ros::Rate r(1.0);
```

在本例中，我们将以 1Hz 的速率发布里程信息，使得显示更简洁些，大多数系统将以更高的速率发布里程信息。

```
//compute odometry in a typical way given the velocities of the robot
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;
```

在这里，我们将根据我们设定的恒定速度更新我们的里程信息。当然，真正的里程系统会整合计算速度。

```
//since all odometry is 6DOF we'll need a quaternion created from yaw
geometry_msgs::Quaternion odom_quat = tf::createQuaternionMsgFromYaw(th);
```

我们通常会尝试在我们的系统中使用所有消息的 3D 版本，以允许 2D 和 3D 组件在适当的情况下协同工作，并将消息数量保持在最低限度。因此，有必要将我们的 yaw(偏航值)变为四元数。幸运的是，tf 提供的功能允许从 yaw(偏航值)容易地创建四元数，并且可以方便地获取四元数的偏航值。

```
//first, we'll publish the transform over tf
geometry_msgs::TransformStamped odom_trans;
odom_trans.header.stamp = current_time;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_link";
```

在这里，我们将创建一个TransformStamped消息，通过tf发送。我们要发布在current_time时刻的从"odom"坐标系到“base_link”坐标系的变换。因此，我们将相应地设置消息头和 child_frame_id，确保使用“odom”作为父坐标系，将“base_link”作为子坐标系。

```
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = y;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = odom_quat;

//send the transform
odom_broadcaster.sendTransform(odom_trans);
```

将我们的里程数据中填入变换消息中，然后使用 TransformBroadcaster 发送变换。

```
//next, we'll publish the odometry message over ROS
nav_msgs::Odometry odom;
odom.header.stamp = current_time;
odom.header.frame_id = "odom";
```

我们还需要发布 nav_msgs/Odometry 消息，以便导航包可以从中获取速度信息。我们将消息的 header 设置为 current_time 和“odom”坐标系。

```
//set the position
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.position.z = 0.0;
odom.pose.pose.orientation = odom_quat;

//set the velocity
odom.child_frame_id = "base_link";
odom.twist.twist.linear.x = vx;
```

```
odom.twist.twist.linear.y = vy;
odom.twist.twist.angular.z = vth;
```

这将使用里程数据填充消息，并发送出去。我们将消息的 child_frame_id 设置为“base_link”坐标系，因为我们要发送速度信息到这个坐标系。

ROS 与 navigation 教程-发布传感器数据

说明：

- 介绍如何通过 ROS 发送两种类型的传感器数据， 分别是 sensor_msgs/LaserScan 消息和 sensor_msgs/PointCloud 消息。

发布传感器数据

通过 ROS 从传感器正确发布数据对于导航包安全运行非常重要。如果导航包没有收到来自机器人传感器的信息，那么它将被盲目地驾驶，并且很有可能会撞到东西。有许多传感器可为导航包提供信息：激光器，相机，声纳，红外线，碰撞传感器等。但是，当前的导航包只接受使用 sensor_msgs/LaserScan 消息类型或 sensor_msgs/PointCloud 消息类型。以下教程将提供两种类型消息的典型设置和使用示例。

ROS 消息头

像其他类型消息一样， sensor_msgs/LaserScan 和 sensor_msgs/PointCloud 消息包含 TF 坐标系和时间关联的消息。为了标准化此信息的发送方式， Header 消息类型用作所有消息中的字段

Header 类型中有三个字段如下所示。seq 字段对应于标识符的消息由发布者发送时候自动增加。stamp 字段存储与数据相关联的时间信息。以激光为例，stamp 可能对应于扫描发生时的时间。frame_id 字段存储与数据相关联的 TF 坐标系的消息。以激光为例，应该是设置为进行扫描的坐标系。

```
#Standard metadata for higher-level flow data types
#sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
```

```
time stamp
#Frame this data is associated with
# 0: no frame
# 1: global frame
string frame_id
```

发布激光消息

- LaserScan 消息

对于具有激光扫描仪的机器人，ROS 在 sensor_msgs 包提供特殊的消息类型，叫 LaserScan，用于保存有关扫描的信息。主要从扫描仪返回的数据可以格式化成这种消息类型，就可以使代码更容易处理。在谈论如何生成和发布这些消息之前，让我们来看看消息规范：

```
#
# Laser scans angles are measured counter clockwise, with 0 facing forward
# (along the x-axis) of the device frame
#

Header header
float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]
float32 time_increment   # time between measurements [seconds]
float32 scan_time        # time between scans [seconds]
float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]
float32[] ranges         # range data [m] (Note: values < range_min or > range_max should
be discarded)
float32[] intensities    # intensity data [device-specific units]
```

上面的注释就能明白字段表达的意思，为了更直观认识，写一个简单的激光扫描发布器来进行演示。

编写代码发布激光/LaserScan 消息

通过 ROS 发布 LaserScan 消息是相当简单的。我们将首先提供下面的示例代码，然后逐行解释代码。

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "laser_scan_publisher");
```

```
  ros::NodeHandle n;
  ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>("scan", 50);

  unsigned int num_readings = 100;
  double laser_frequency = 40;
  double ranges[num_readings];
  double intensities[num_readings];

  int count = 0;
  ros::Rate r(1.0);
  while(n.ok()){
    //generate some fake data for our laser scan
    for(unsigned int i = 0; i < num_readings; ++i){
      ranges[i] = count;
      intensities[i] = 100 + count;
    }
    ros::Time scan_time = ros::Time::now();

    //populate the LaserScan message
    sensor_msgs::LaserScan scan;
    scan.header.stamp = scan_time;
    scan.header.frame_id = "laser_frame";
    scan.angle_min = -1.57;
    scan.angle_max = 1.57;
    scan.angle_increment = 3.14 / num_readings;
    scan.time_increment = (1 / laser_frequency) / (num_readings);
    scan.range_min = 0.0;
    scan.range_max = 100.0;

    scan.ranges.resize(num_readings);
    scan.intensities.resize(num_readings);
    for(unsigned int i = 0; i < num_readings; ++i){
      scan.ranges[i] = ranges[i];
      scan.intensities[i] = intensities[i];
    }

    scan_pub.publish(scan);
    ++count;
    r.sleep();
  }
}
```

下面一一进行解释

```
#include <sensor_msgs/LaserScan.h>
```

这里包含 sensor_msgs/LaserScan 的头文件

```
ros::Publisher scan_pub = n.advertise<sensor_msgs::LaserScan>("scan", 50);
```

此代码创建一个 ros::Publisher，用于使用 ROS 发送 LaserScan 消息

```
unsigned int num_readings = 100;
double laser_frequency = 40;
double ranges[num_readings];
double intensities[num_readings];
```

这里设置消息的长度，便于填充一些虚拟数据。一个真正的应用程序将从他们的激光扫描仪中获取数据

```
//generate some fake data for our laser scan
for(unsigned int i = 0; i < num_readings; ++i){
  ranges[i] = count;
  intensities[i] = 100 + count;
}
ros::Time scan_time = ros::Time::now();
```

用每秒增加 1 的值填充虚拟激光数据

```
//populate the LaserScan message
sensor_msgs::LaserScan scan;
scan.header.stamp = scan_time;
scan.header.frame_id = "laser_frame";
scan.angle_min = -1.57;
scan.angle_max = 1.57;
scan.angle_increment = 3.14 / num_readings;
scan.time_increment = (1 / laser_frequency) / (num_readings);
scan.range_min = 0.0;
scan.range_max = 100.0;

scan.ranges.resize(num_readings);
scan.intensities.resize(num_readings);
for(unsigned int i = 0; i < num_readings; ++i){
  scan.ranges[i] = ranges[i];
  scan.intensities[i] = intensities[i];
}
```

创建一个 scan_msgs::LaserScan 消息，并填写我们生成的数据，并准备发送

```
scan_pub.publish(scan);
```

通过 ROS 发布消息

通过 ROS 发布点云/PointClouds

- PointCloud 消息

为了存储和共享关于空间上多个点的数据，ROS 提供了一个 sensor_msgs/PointCloud 消息。如下所示，此消息旨在支持三维点阵列以及作为通道存储的任何相关数据。例如，PointCloud 可以发送一个“强度”通道，保存有关云中每个点的强度值的信息。我们将在下一部分中探索使用 ROS 发送 PointCloud 的示例。

```
#This message holds a collection of 3d points, plus optional additional information about
each point.
#Each Point32 should be interpreted as a 3d point in the frame given in the header

Header header
geometry_msgs/Point32[] points  #Array of 3d points
ChannelFloat32[] channels       #Each channel should have the same number of elements as p
oints array, and the data in each channel should correspond 1:1 with each point
```

- 编写代码发布 PointCloud 消息

用 ROS 发布 PointCloud 是非常简单的。我们将在下面详细介绍一个简单的例子，然后详细讨论重要的部分。

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "point_cloud_publisher");

  ros::NodeHandle n;
  ros::Publisher cloud_pub = n.advertise<sensor_msgs::PointCloud>("cloud", 50);

  unsigned int num_points = 100;

  int count = 0;
  ros::Rate r(1.0);
  while(n.ok()){
```

```
    sensor_msgs::PointCloud cloud;
    cloud.header.stamp = ros::Time::now();
    cloud.header.frame_id = "sensor_frame";

    cloud.points.resize(num_points);

    //we'll also add an intensity channel to the cloud
    cloud.channels.resize(1);
    cloud.channels[0].name = "intensities";
    cloud.channels[0].values.resize(num_points);

    //generate some fake data for our point cloud
    for(unsigned int i = 0; i < num_points; ++i){
      cloud.points[i].x = 1 + count;
      cloud.points[i].y = 2 + count;
      cloud.points[i].z = 3 + count;
      cloud.channels[0].values[i] = 100 + count;
    }

    cloud_pub.publish(cloud);
    ++count;
    r.sleep();
  }
}
```

下面进行逐行解释

```
#include <sensor_msgs/PointCloud.h>
```

包含 sensor_msgs/PointCloud 消息头

```
ros::Publisher cloud_pub = n.advertise<sensor_msgs::PointCloud>("cloud", 50);
```

创建 ros::Publisher, 用于发送 PointCloud 消息

```
sensor_msgs::PointCloud cloud;
cloud.header.stamp = ros::Time::now();
cloud.header.frame_id = "sensor_frame";
```

填写 PointCloud 消息的 header，分别是发布信息的相关坐标系和时间戳

```
cloud.points.resize(num_points);
```

设置点云中的点数，以便我们可以用虚拟数据填充它们

```
//we'll also add an intensity channel to the cloud
cloud.channels.resize(1);
cloud.channels[0].name = "intensities";
cloud.channels[0].values.resize(num_points);
```

向 PointCloud 添加一个称为“强度”的频道，并将其大小设置为与云中我们将拥有的点数相匹配。

```
//generate some fake data for our point cloud
for(unsigned int i = 0; i < num_points; ++i){
  cloud.points[i].x = 1 + count;
  cloud.points[i].y = 2 + count;
  cloud.points[i].z = 3 + count;
  cloud.channels[0].values[i] = 100 + count;
}
```

用一些虚拟数据填充 PointCloud 消息。另外，用伪数据填充强度信道。

```
cloud_pub.publish(cloud);
```

使用 ROS 发布 PointCloud

ROS 与 navigation 教程-编写自定义全局路径规划

- 介绍在 ROS 中编写和使用全局路径规划器的步骤，提供从编写路径计划程序类开始直到将其部署为插件的所有步骤。

简介：

首先要知道的是，为了向 ROS 添加一个新的全局路径规划器，新的路径规划器必须遵守 nav_core 包中定义的 nav_core::BaseGlobalPlanner C ++接口。一旦编写了全局路径规划器，它必须作为插件添加到 ROS 中，以便它可以被 move_base 包使用。

我将使用 Turtlebot 作为机器人的一个例子来部署新的路径规划器。有关如何将真正的 GA 规划程序集成为 ROS 插件的教程，参考在 ROS 中添加遗传算法全局路径规划器作为插件。此链接中还提供了视频教程

(1)编写路径规划类

- Class Header/类头文件

第一步是为遵循 nav_core::BaseGlobalPlanner 的路径规划器编写一个新类。在 carrot_planner.h 中可以找到一个类似的例子作为参考。为此，您需要创建一个将在本例中调用的 global_planner.h 头文件。我将介绍添加插件的最小代码，这是添加任何全局规划程序的必要和常见步骤。

最小头文件 global_planner.h 定义如下：

```
/** include the libraries you need in your planner here */
 /** for global path planner interface */
 #include <ros/ros.h>
 #include <costmap_2d/costmap_2d_ros.h>
 #include <costmap_2d/costmap_2d.h>
 #include <nav_core/base_global_planner.h>
 #include <geometry_msgs/PoseStamped.h>
 #include <angles/angles.h>
 #include <base_local_planner/world_model.h>
 #include <base_local_planner/costmap_model.h>

 using std::string;

 #ifndef GLOBAL_PLANNER_CPP
 #define GLOBAL_PLANNER_CPP

 namespace global_planner {

 class GlobalPlanner : public nav_core::BaseGlobalPlanner {
 public:

  GlobalPlanner();
  GlobalPlanner(std::string name, costmap_2d::Costmap2DROS* costmap_ros);

  /** overridden classes from interface nav_core::BaseGlobalPlanner **/
  void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
  bool makePlan(const geometry_msgs::PoseStamped& start,
                const geometry_msgs::PoseStamped& goal,
                std::vector<geometry_msgs::PoseStamped>& plan
               );
  };
 };
```

```
#endif
```

解释头文件的不同部分

```
#include <ros/ros.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>
#include <nav_core/base_global_planner.h>
#include <geometry_msgs/PoseStamped.h>
#include <angles/angles.h>
#include <base_local_planner/world_model.h>
#include <base_local_planner/costmap_model.h>
```

需要包含路径规划器所需的核心 ROS 库。路径规划器需要使用<costmap_2d/costmap_2d_ros.h>和 <costmap_2d/costmap_2d.h>，而 costmap_2d::Costmap2D 类作为输入的地图类。当定义为插件时，路径规划器类将自动访问此地图。没有必要订阅 costmap2d 来获取 ROS 的代价地图。

<nav_core/base_global_planner.h> 用于导入接口 nav_core::BaseGlobalPlanner，这是插件必须要继承的父类。

```
namespace global_planner {

class GlobalPlanner : public nav_core::BaseGlobalPlanner {
```

为您的类定义命名空间是一个很好的做法，尽管不是必需的。在这里，我们将为 GlobalPlanner 类定义命名空间为 global_planner。命名空间用于定义对类的完整引用，如 global_planner::GlobalPlanner。然后定义类 GlobalPlanner 并继承接口 nav_core::BaseGlobalPlanner。在 nav_core::BaseGlobalPlanner 中定义的所有方法都必须在新类 GlobalPlanner 中总重写。

```
public:

  GlobalPlanner();
  GlobalPlanner(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
   /** overriden classes from interface nav_core::BaseGlobalPlanner **/
  void initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros);
  bool makePlan(const geometry_msgs::PoseStamped& start,
                const geometry_msgs::PoseStamped& goal,
                std::vector<geometry_msgs::PoseStamped>& plan
               );
```

构造函数 GlobalPlanner(std::string name, costmap_2d::Costmap2DROS* costmap_ros)用于初始化代价地图，即将用于规划的代价地图（costmap_ros）以及规划器的名称（name）。默认构造函数 GlobalPlanner()即使用默认值作为初始化。方法 initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros)是 BaseGlobalPlanner 的初始化函数，用于初始化 costmap，参数为用于规划的代价地图（costmap_ros）和规划器的名称(name)。

对于 carrot_planner 的具体情况，初始化方法实现如下：

```
void CarrotPlanner::initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros){
    if(!initialized_){
      costmap_ros_ = costmap_ros; //initialize the costmap_ros_ attribute to the paramete
r.
      costmap_ = costmap_ros_->getCostmap(); //get the costmap_ from costmap_ros_

     // initialize other planner parameters
      ros::NodeHandle private_nh("~/" + name);
      private_nh.param("step_size", step_size_, costmap_->getResolution());
      private_nh.param("min_dist_from_robot", min_dist_from_robot_, 0.10);
      world_model_ = new base_local_planner::CostmapModel(*costmap_);

      initialized_ = true;
    }
    else
      ROS_WARN("This planner has already been initialized... doing nothing");
  }
```

接着，bool makePlan（start，goal，plan）的方法必须要重写。最终规划将存储在方法的参数 std::vector<geometry_msgs::PoseStamped>& plan 中。该规划将通过插件自动发布为话题。carrot_planner 的 makePlan 方法的参考。

- Class Implementation/类实现

接下来，我将介绍在实施全局规划插件时要考虑的主要问题。我将不会描述完整的路径规划算法。为了说明的目的，我将使用 makePlan（）方法的路径规划的例子（以后可以改进）。这是全局规划（global_planner.cpp）的最小代码实现，它总是生成一个静态计划。

```
#include <pluginlib/class_list_macros.h>
#include "global_planner.h"
```

```
//register this planner as a BaseGlobalPlanner plugin
PLUGINLIB_EXPORT_CLASS(global_planner::GlobalPlanner, nav_core::BaseGlobalPlanner)

using namespace std;

//Default Constructor
namespace global_planner {

GlobalPlanner::GlobalPlanner (){

}

GlobalPlanner::GlobalPlanner(std::string name, costmap_2d::Costmap2DROS* costmap_ros){
  initialize(name, costmap_ros);
}


void GlobalPlanner::initialize(std::string name, costmap_2d::Costmap2DROS* costmap_ros){

}

bool GlobalPlanner::makePlan(const geometry_msgs::PoseStamped& start, const geometry_msg
s::PoseStamped& goal,  std::vector<geometry_msgs::PoseStamped>& plan ){

   plan.push_back(start);
  for (int i=0; i<20; i++){
    geometry_msgs::PoseStamped new_goal = goal;
    tf::Quaternion goal_quat = tf::createQuaternionFromYaw(1.54);

     new_goal.pose.position.x = -2.5+(0.05*i);
     new_goal.pose.position.y = -3.5+(0.05*i);

     new_goal.pose.orientation.x = goal_quat.x();
     new_goal.pose.orientation.y = goal_quat.y();
     new_goal.pose.orientation.z = goal_quat.z();
     new_goal.pose.orientation.w = goal_quat.w();

  plan.push_back(new_goal);
  }
  plan.push_back(goal);
 return true;
}
```

```
};
```

可以根据计划人员的要求和规格实施构造函数。在这个具体的说明性例子中不考虑它们的实现。有几件重要的事情要考虑：

1.注册规划器作为 BaseGlobalPlanner 插件：这是通过指令 PLUGINLIB_EXPORT_CLASS(global_planner::GlobalPlanner, nav_core::BaseGlobalPlanner)完成的。为此，必须包含库 #include <pluginlib/class_list_macros.h>

2.makePlan()方法实现：在开始和目标参数用来获取初始位置和目标位置。在该说明性实现中，以起始位置（plan.push_back(start)）作为开始的规划变量。然后，在 for 循环中，将在规划中静态插入 20 个新的虚拟位置，然后将目标位置作为最后一个位置插入规划。然后，此规划路径将发送到 move_base 全局规划模块，该模块将通过 ROS 话题 nav_msgs/Path 进行发布，然后将由本地规划模块接收。

现在，您的全局规划类已经完成，您已准备好进行第二步，即为全局规划类创建插件，将其集成到 move_base 包的全局规划模块 nav_core::BaseGlobalPlanner 中。

- 编译

要编译上面创建的全局规划库，必须将其添加到 CMakeLists.txt 中。添加代码：

```
add_library(global_planner_lib src/path_planner/global_planner/global_planner.cpp)
```

然后在终端中运行 catkin_make，在 catkin 工作空间目录中生成二进制文件。这将在 lib 目录中创建库文件 ~/catkin_ws/devel/lib/libglobal_planner_lib。观察到“lib”附加到 CMakeLists.txt 中声明的库名 global_planner_lib

(2)编写你的插件

基本上，重要的是按照插件描述页面中所述创建一个新插件所需的所有步骤。有五个步骤：

- 插件注册

首先，您需要通过导出将全局规划类注册为插件。为了允许一个类被动态加载，它必须被标记为一个导出的类。这是通过特殊宏 PLUGINLIB_EXPORT_CLASS 完成的。这个宏可以放在构成插件库的任何源（.cpp）文件中，通常放在导出类的.cpp 文件的末尾。以上已经在 global_planner.cpp 中完成了该指令

```
PLUGINLIB_EXPORT_CLASS(global_planner::GlobalPlanner, nav_core::BaseGlobalPlanner)
```

这将使 global_planner::GlobalPlanner 类注册为 move_base 包中 nav_core::BaseGlobalPlanner 类的插件。

- 插件描述文件

第二步是在描述文件中描述该插件。插件描述文件是一个 XML 文件，用于以机器可读格式存储有关插件的所有重要信息。它包含有关插件库的信息，插件的名称，插件的类型等。以我们的全局规划类为例，您需要创建一个新文件并将其保存在您的包中的某个位置（在我们的 case_ global_planner 包），并给它一个名称，例如 global_planner_plugin.xml。插件描述文件（global_planner_plugin.xml）的内容将如下所示：

```
<library path="lib/libglobal_planner_lib">
  <class name="global_planner/GlobalPlanner" type="global_planner::GlobalPlanner" base_class_type="nav_core::BaseGlobalPlanner">
    <description>This is a global planner plugin by iroboapp project.</description>
  </class>
</library>
```

在第一行 `<library path="lib/libglobal_planner_lib">`我们指定插件库的路径。我们的例子，路径是 lib/libglobal_planner_lib，其中 lib 是目录中的文件夹 ~/catkin_ws/devel/（参见上面的编译部分）。在此行中，我们首先指定我们将在 move_base 启动文件中使用的 global_planner 插件的名称作为参数，在此行中， 它指定了要在 nav_core 中使用的全局规划器。通常使用命名空间（global_planner）后跟斜杠，然后使用类（ GlobalPlanner）的名称来指定插件的名称。如果不指定名称，那么名称将等于该类型，在这种情况下将是 global_planner::GlobalPlanner。它建议指定名称以避免混淆。

该类型指定了实现插件的类的名称，在我们的例子中是 global_planner::GlobalPlanner，而 base_class_type 指定了实现插件的基类的名称，在我们的例子中是 nav_core::BaseGlobalPlanner。所述 `<description>` 标记提供关于插件的简要说明。有关插件描述文件及其相关标签/属性的详细说明，请参阅以下文档。

为什么我们需要这个文件？除了代码宏，我们需要这个文件，以允许 ROS 系统自动发现，加载。插件描述文件还包含重要信息，如插件的描述，不适合加在宏定义里。

- 注册插件到 ROS 包系统

为了让 pluginlib 在所有 ROS 程序包上查询系统上的所有可用插件，每个程序包都必须明确指定其导出的插件，哪些程序包库包含这些插件。插件提供者必须在其导出标记块中的 package.xml 中指向其插件描述文件。请注意，如果您有其他出口，他们都必须在同一个导出字段中。在我们的全局规划示例中，相关行将如下所示：

```
<export>
<nav_core plugin="${prefix}/global_planner_plugin.xml" />
</export>
```

在${PREFIX}/自动确定 global_planner_plugin.xml 文件完整的路径。有关导出插件的详细讨论，请参阅以下文档。

注意：为了使上述导出命令正常工作，提供程序包必须直接依赖于包含插件接口的程序包，这在全局规划程序的情况下是 nav_core。因此，global_planner 软件包在其 package.xml 中必须包含以下内容：

```
<build_depend>nav_core</build_depend>
<run_depend>nav_core</run_depend>
```

这将告诉编译器关于 nav_core 包的依赖性

- 在 ROS 包系统中，查询可用插件

可以通过 rospack 查询 ROS 包系统，以查看任何给定的包可用的插件。例如：

```
rospack plugins --attrib=plugin nav_core
```

这将返回从 nav_core 包导出的所有插件。这是一个执行的例子：

```
akoubaa@anis-vbox:~/catkin_ws$ rospack plugins --attrib=plugin nav_core
rotate_recovery /opt/ros/hydro/share/rotate_recovery/rotate_plugin.xml
navfn /opt/ros/hydro/share/navfn/bgp_plugin.xml
base_local_planner /opt/ros/hydro/share/base_local_planner/blp_plugin.xml
move_slow_and_clear /opt/ros/hydro/share/move_slow_and_clear/recovery_plugin.xml
global_planner /home/akoubaa/catkin_ws/src/global_planner/global_planner_plugin.xml
dwa_local_planner /opt/ros/hydro/share/dwa_local_planner/blp_plugin.xml
clear_costmap_recovery /opt/ros/hydro/share/clear_costmap_recovery/ccr_plugin.xml
carrot_planner /opt/ros/hydro/share/carrot_planner/bgp_plugin.xml
```

请注意，我们的插件现在可以在 global_planner 软件包下使用，并且在 /home/akoubaa/catkin_ws/src/global_planner/global_planner_plugin.xml 文件中指定。您还可以观察 nav_core 软件包中已经存在的其他插件，其中包括 carrot_planner/CarrotPlanner 和 navfn，其实现 Dijkstra 算法。

现在，你的插件就可以使用了。

(3)在 Turtlebot 上运行插件

在 Turtlebot 上运行您的规划器有几个步骤。首先，您需要将包含您的全局规划程序（在我们的示例中为 global_planner）的软件包复制到 Turtlebot 的 catkin 工作区（例如 catkin_ws）。然后，您需要运行 catkin_make 将您的插件导出到您的 turtlebot ROS 环境中。其次，您需要对 move_base 配置进行一些修改，以指定要使用的新计划。为此，请按照以下三个步骤：

1.在 ROS Hydro 版本中，转到此文件夹/opt/ros/hydro/share/turtlebot_navigation/launch/includes，其他版本方法通用。

```
$ roscd turtlebot_navigation/
$ cd launch/includes/
```

2.打开文件 move_base.launch.xml（您可能需要 sudo 打开并能够保存），并将新的规划器添加为全局规划器的参数，如下所示：

```
 ......
 <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
 <param name="base_global_planner" value="global_planner/GlobalPlanner"/>
 ....
```

保存并关闭 move_base.launch.xml。需要注意的是，规划器的名称是 global_planner/GlobalPlanner，跟指定在 global_planner_plugin.xml 文件名称一致。现在，您已准备好使用新的计划。

3.你现在必须启动你的 Turtlebot。您需要启动 minimal.launch，3dsensor.launch，amcl.launch.xml 和 move_base.launch.xml。以下是可用于此目的的启动文件的示例。

```
<launch>
 <include file="$(find turtlebot_bringup)/launch/minimal.launch"></include>

   <include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
     <arg name="rgb_processing" value="false" />
     <arg name="depth_registration" value="false" />
     <arg name="depth_processing" value="false" />
     <arg name="scan_topic" value="/scan" />
   </include>

   <!-- Map server -->
   <arg name="map_file" default="your_map_folder/your_map_file.yaml"/>
   <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

   <!-- Localization -->
```

```
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find turtlebot_navigation)/launch/includes/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Move base -->
  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>

</launch>
```

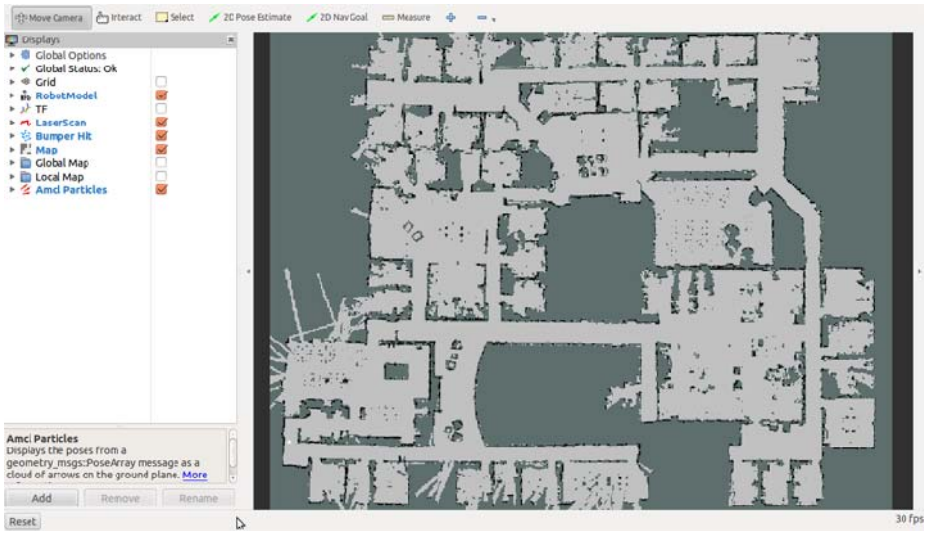
请注意，在使用此启动文件启动您的 turtlebot 时，将会考虑在 move_base.launch.xml 文件中所做的更改。

(4)用 RVIZ 测试计划

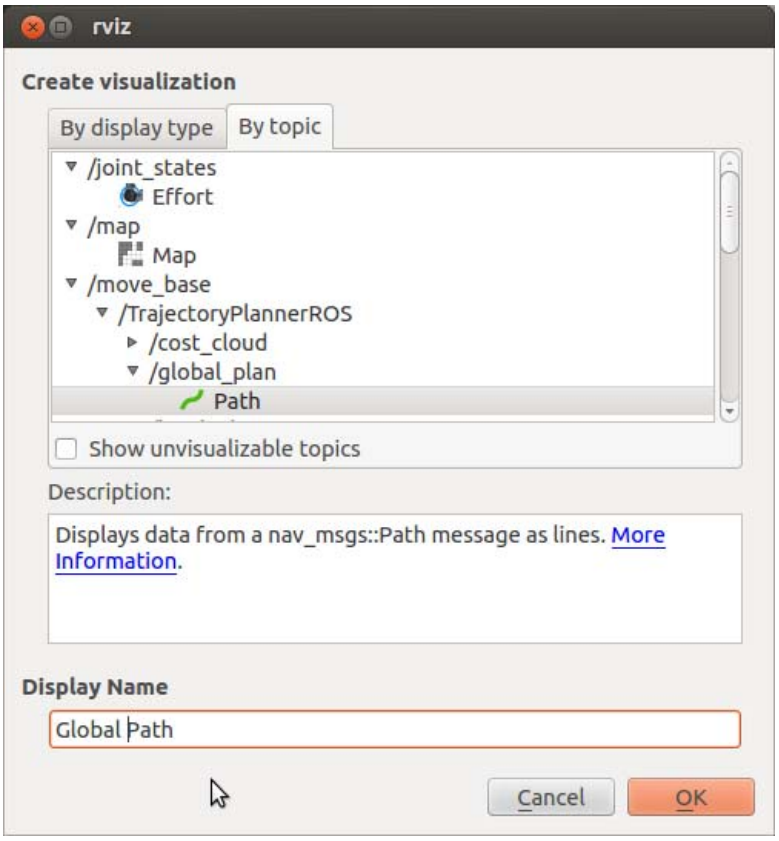
在你启动 Turtlebot 之后，你可以使用这个命令启动 rviz（在新的终端）

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

效果图：

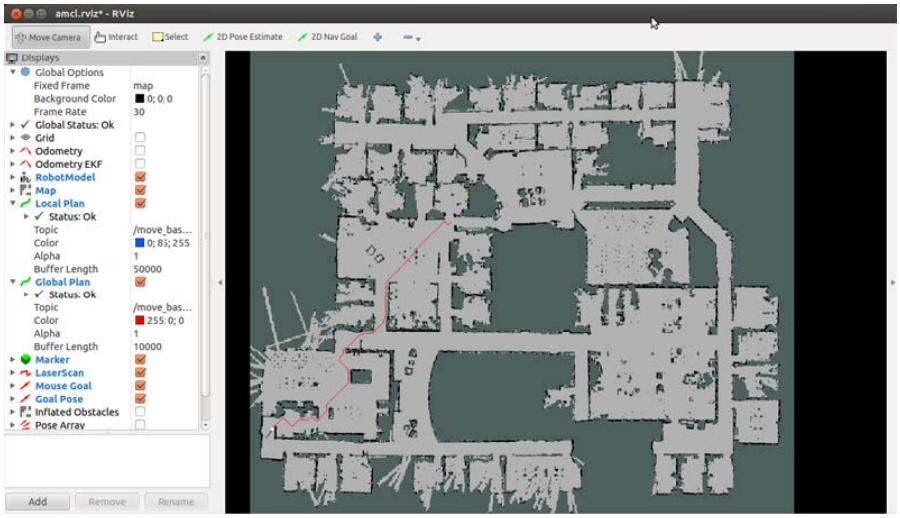


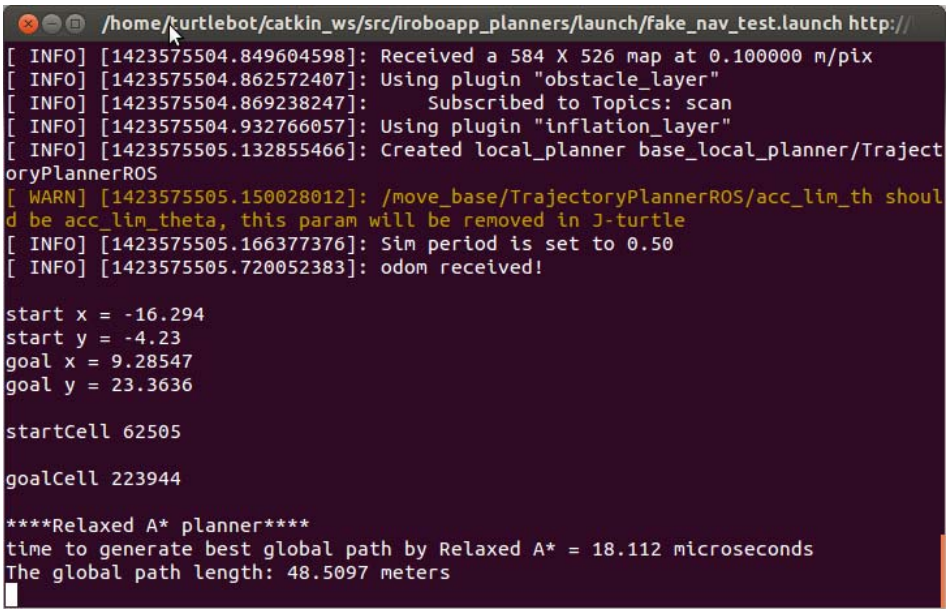
您现在可以通过点击按钮"Add"（在底部）添加要在 rviz 中显示的所有信息。您将看到以下窗口：



例如，如果要显示全局路径，请单击"By topic"，在“move_base”类别下选择"/global_plan" ，然后选择"Path"，如果需要，请添加显示名称，然后单击“OK”。您可以以相同的方式添加本地路径。

现在点击 “2D nav goal”按钮（在顶部），并选择目标位置。你现在可以看到你的机器人走向目标。





测试地图：

willow_garage_map.pgm

willow_garage_map.yaml

参考：

- http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS

ROS 与 navigation 教程-stage 仿真

说明：

- 介绍如何在 stage 仿真器下使用 navigation 包

代码：

- https://github.com/ros-planning/navigation_tutorials

简介：

- 这个代码包含 launch 文件，用于在 stage 中运行 navigation 包

- launch/move_base_amcl_10cm.launch: 在分辨率为 10CM 的地图运行导航包并带有 amcl

- launch/move_base_amcl_5cm.launch: 在分辨率为 5CM 的地图运行导航包并带有 amcl
- launch/move_base_amcl_2.5cm.launch: 在分辨率为 2.5CM 的地图运行导航包并带有 amcl
- launch/move_base_fake_localization_10cm.launch:在分辨率为 10CM 的地图运行导航包并带有 fake_localization
- launch/move_base_fake_localization_5cm.launch:在分辨率为 5CM 的地图运行导航包并带有 fake_localization
- launch/move_base_fake_localization_2.5cm.launch: 在分辨率为 10CM 的地图运行导航包并带有 fake_localization
- launch/move_base_multi_robot.launch: 运行导航包并带有多个机器人
- launch/move_base_gmapping_5cm.launch: 在分辨率为 5CM 的地图运行导航包并带有 gmapping

参考：

- http://wiki.ros.org/navigation_stage

ROS 与 navigation 教程-示例-激光发布(C++)

说明：

- 介绍如何实际编程发布激光数据

代码：

- 参考 github 仓库

安装：

- 新建目录

```
$ mkdir -p ~/catkin_ws/src
```

- 下载并编译

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-planning/navigation_tutorials
```

```
$ cd ..
$ catkin_make
```

测试

- 新终端，执行 roscore

```
$ roscore
```

- 新终端，执行

```
$ rosrun laser_scan_publisher_tutorial laser_scan_publisher
```

ROS 与 navigation 教程-示例-里程发布(C++)

说明：

- 介绍如何实际编程发布里程信息

代码：

- 参考 github 仓库

安装：

- 新建目录

```
$ mkdir -p ~/catkin_ws/src
```

- 下载并编译

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-planning/navigation_tutorials
$ cd ..
$ catkin_make
```

测试

- 新终端，执行 roscore

```
$ roscore
```

- 新终端，执行

```
$ rosrun odometry_publisher_tutorial odometry_publisher
```

ROS 与 navigation 教程-示例-点云发布(C++)

说明：

- 介绍如何实际编程发布点云数据

代码：

- 参考 github 仓库

安装：

- 新建目录

```
$ mkdir -p ~/catkin_ws/src
```

- 下载并编译

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-planning/navigation_tutorials
$ cd ..
$ catkin_make
```

测试

- 新终端，执行 roscore

```
$ roscore
```

- 新终端，执行

```
$ rosrun point_cloud_publisher_tutorial point_cloud_publisher
```

ROS 与 navigation 教程-示例-机器人 TF 设置(C++)

说明：

- 介绍如何设置机器人的 TF，实现底座到激光的变换

代码：

- 参考 github 仓库

安装：

- 新建目录

```
$ mkdir -p ~/catkin_ws/src
```

- 下载并编译

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-planning/navigation_tutorials
$ cd ..
$ catkin_make
```

测试

- 新终端，执行 roscore

```
$ roscore
```

- 新终端，执行发布 TF

```
$ rosrun robot_setup_tf_tutorial tf_broadcaster
```

- 新终端，执行接收 TF

```
$ rosrun robot_setup_tf_tutorial tf_listener
```

ROS 与 navigation 教程-示例-导航目标设置(C++)

说明：

- 介绍如何实现指定目标点导航

代码：

- 参考 github 仓库

安装：

- 新建目录

```
$ mkdir -p ~/catkin_ws/src
```

- 下载并编译

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/ros-planning/navigation_tutorials
$ cd ..
$ catkin_make
```

测试

- 新终端，执行 roscore

```
$ roscore
```

- 修改源码 simple_navigation_goals.cpp

```
rosed simple_navigation_goals_tutorial simple_navigation_goals.cpp
```

- 替换里面的坐标点

```
goal.target_pose.pose.position.x = 2.0;
goal.target_pose.pose.position.y = 0.2;
```

- 新终端，执行驱动

```
$ rosrun simple_navigation_goals_tutorial simple_navigation_goals
```

ROS 与 navigation 教程-turtlebot-整合导航包简明指南

说明：

- 介绍为 turtlebot 整合导航包简明指南

关键文件：

- turtlebot 应用导航包的规则也跟其他机器人一样，由 launch 和 yaml 文件组成。包含在 turtlebot_navigation 软件包，分别位于 luanch 和 param 目录。

移动基座

- TurtleBot 导航行为是由 move_base 生成的，他们拥有全局和本地的代价地图，因此可以创建全局和本地计划。它的行为是在 param/move_base 的 yaml 文件中定义的，三个用于代价地图，base_local_planner_params.yaml 是针对本地规划的。costmap_2d 配置是相当棘手的，在大多数情况下是由 cpu 使用和性能之间的平衡需求驱动的，所以我们不在此提及。

规划器

- 改变速度限制：
  - max_vel_x：最大线速度; TurtleBot 1 的绝对限制为 0.5，对于 TurtleBot 2 为 0.7
  - min_vel_x：最小线速度; 也许您需要在重载时增加，以便机器人以最低速度能克服摩擦
  - max_rotational_vel：最大角速度; TurtleBot 2 的绝对限制是 3.14（TurboBot 1 不确定）
  - min_in_place_rotational_vel：与最小线速度相同作用

- Goal tolerance/目标容差
- 这两个参数允许您在达到目标时使 TurtleBot 或多或少准确。小心：非常低的值可以使机器人在目标周围移动，而不会达到它！
  - yaw_goal_tolerance
  - xy_goal_tolerance

- Cost computing biases/代价计算偏倚
- 这三个参数定义了 TurtleBot 遵循其全局规划的偏好
  - path_distance_bias：增加使机器人更紧密地遵循计划，即偏向更符合全局规划的路径
  - goal_distance_bias：增加，使机器人轨迹更平滑，更有效率，即偏向更符合本地规划的路径

- occdist_scale：增加使机器人更害怕撞上障碍物

Amcl (localization)/定位

- TurtleBot 定位由 amcl 提供。您可以通过修改 launch/includes/_amcl.launch 文件中的参数来调整此算法。
- 但这不是基本的东西，所以超出了本教程的范围。

Gmapping (map building)/建图

- TurtleBot 地图使用 gmapping 进行构建。您可以通过修改 launch/includes/_gmapping.launch 文件中的参数来调整此算法。
- 但是，这再次超出了本教程的范围。

参考：

- 参考官网 wiki

ROS 与 navigation 教程-turtlebot-SLAM 地图构建

说明：

- 介绍 turtlebot 如何通过 gmapping 算法实现构建地图
- 目的是演示如何构建地图，使得机器人能记住周边环境，通过生成的地图，机器人能实现自主导航。

步骤：

- 确保已经启动机器人(参考最小启动 luanch)和配置好网络环境（参考网络环境配置）
- 默认的导航参数是由 turtlebot_navigation 包提供，应该能适合大部分情况。如果不可是，请参考配置篇
- 主机，新终端，启动 turtlebot

```
$ roslaunch turtlebot_bringup minimal.launch
```

- 主机，新终端，启动 gmapping

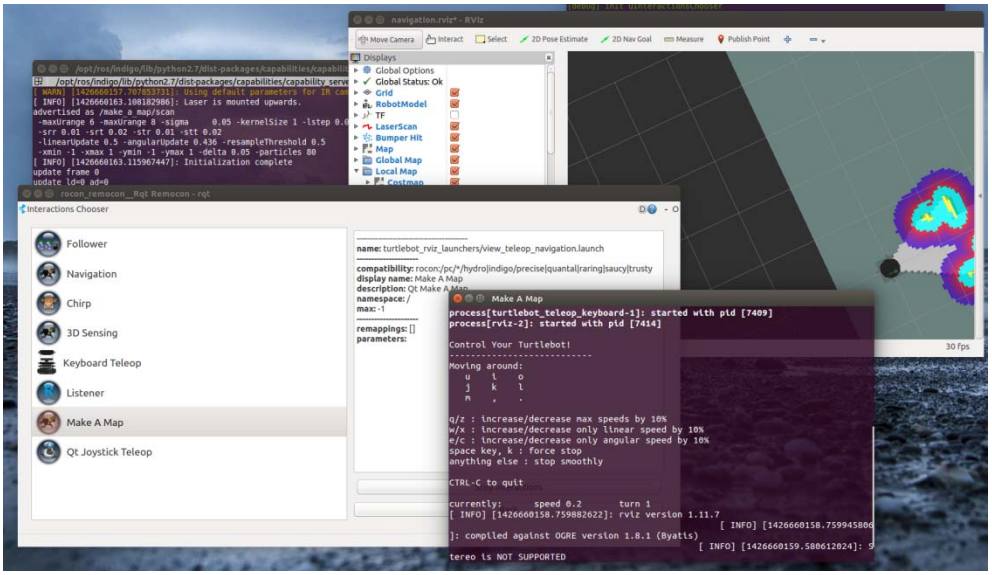
```
$ roslaunch turtlebot_navigation gmapping_demo.launch
```

- 从机，新终端， 启动 rviz

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

如果遇到问题，请检查 TurtleBot Bringup, PC Bringup, 和 Network Configuration

- 控制机器人在周围走动进行建图，完成保存地图
  - 使用键盘等控制，可以参考遥控教程
- 或者使用 Remocon ，新终端，执行命令 `Remocon`
  - 从弹出的窗口选择 PC Pairing/Make A Map
  - 它会启动 rviz 和键盘
  - 然后控制机器人在周围走动进行建图
- 效果图：



- 保存地图：

```
rosrun map_server map_saver -f /tmp/my_map
```

- 可以使用上述命令建图，可以指定地图路径及名称， my_map 是地图名称
- 注意：保存地图之前，不要关闭 gmapping 启动的终端。

ROS 与 navigation 教程-turtlebot-现有地图的自主导航

说明：

- 介绍如何完成建图之后，实现自主导航

步骤：

- 主机，新终端， 启动 turtlebot

```
$ roslaunch turtlebot_bringup minimal.launch
```

- 主机，新终端， 启动指定地图

```
$ export TURTLEBOT_MAP_FILE=/tmp/my_map.yaml
```

- 例如 `export TURTLEBOT_MAP_FILE:=`rospack find turtlebot_navigation`/maps/willow-2010-02-18-0.10.yaml`
- 主机，新终端，启动 amcl

```
$ roslaunch turtlebot_navigation amcl_demo.launch
```

- 从机，新终端， 启动 rviz

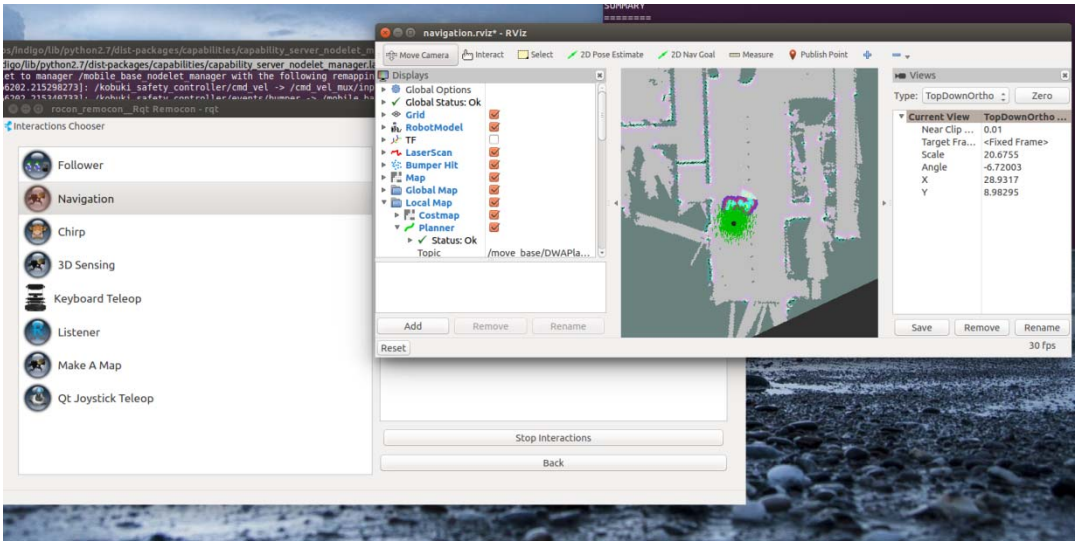
```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

使用 Remocon

- 新终端，执行 `Remocon`
- 指定地图：

```
export TURTLEBOT_MAP_FILE=/tmp/my_map.yaml
```

- 选择 PC Pairing/Navigation
- 效果图：



在 rviz 中操作

(1)定位

- 刚启动，turtlebot 并不知道自己在地图的什么位置，所以需要先初始化定位：
  - 点击"2D Pose Estimate" 按钮
  - 然后观察 turtlebot 机器人在地图中的大概位置，在该位置按住鼠标左键，并拖出一个箭头，箭头代表 turtlebot 的方向，此方向应该跟 turtlebot 的前方方向一致。
- 你会看到在 turtlebot 周边有很多小箭头围绕，这些小箭头就是评估机器人目前的相对位置，同时激光边缘应该跟墙面是重合的，如果不一致就重复定位操作，确保定位正确。
- 注意：在 rviz 中要设置 global options 的 fixed frame 为 "map"，再使用"2D Pose Estimate"

(2) 遥控

- 遥控可以与导航包一起使用，操控优先于自主导航命令。在进行自主导航之前，可以先进行遥控，可以让 turtlebot 的定位箭头进行聚合，让定位更加准确。

(3) 发送导航目标

- 当做好定位，就可以开始进行导航
- 发送目标：
  - 点击"2D Nav Goal" 按钮
  - 点住地图某点，并拖出一个箭头，再松开，即可指定目标点
- 设置成功之后，机器人即自动导航到目标点，如果中间要停止，就发送当前位置给机器人。

ROS 与 navigation 教程-map_server 介绍

说明：

- 介绍 map_server 包的地图格式和命令行工具

概要

- map_server 包提供了一个 map_server ROS 节点，该节点通过 ROS 服务器方式提供地图数据。
- 该包还提供了 map_saver 命令行 utility，使用该工具可将动态创建的地图保存成文件。

代码库

- 参考链接: https://github.com/ros-planning/navigation.git

地图格式

- 该包中的工具使用过的地图会被存储在两个文件中。
- 一个是 YAML 格式的文件描述地图 meta-data 并命名 image 文件。
- 另一个 image 文件用来编码 occupancy data。

Image 格式

- Image 描述了地图上每个单元在相应像素的颜色中的占用状态。
- 白色像素表示自由，黑色像素格表示占用，两种颜色之间的单元表示未知。
- 彩色和灰度图像都适合，但大多数地图是灰度图像（尽管它们存储的好像是以彩色的形式），YAML 文件的阈值划分为 3 类；阈值是在 map_server 内部完成的。
- 比较阈值参数时，图像单元占用概率的计算如下：occ = (255 - color_avg) / 255.0 （color_avg 是用 8 位数表示的来自于所有通道的平均值）。
- 如果图像是 24 位颜色，一个单元的颜色 0x0a0a0a 有一个 0.96 的概率，这是一个完全占用((255-(0*16+10))/255.0=0.96)。
- 如果像素颜色是 0xeeeeee，则占用概率是 0.07，这意味着几乎没有被占用。

- 当 ROS 消息通信时，占用度被表示为范围为[0-100]的整数，0 的意思完全是自由的，100 的意思完全占用，特殊值-1 完全未知。
- Image data 通过 SDL_Image 库读取，取决于 sdl_image 在特定平台上提供的内容，且支持多种格式。
- 一般来说，Image data 支持大多数流行的图像格式。
- 一个必须要注意的例外是 PNG 格式在 OS X 平台上不能被支持。

YAML 格式

- 以下是 YAML 格式的一个示例：

```
image: testmap.png
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

需要的字段:

- image:指定包含 occupancy data 的 image 文件路径; 可以是绝对路径，也可以是相对于 YAML 文件的对象路径 。
- resolution:地图分辨率，单位是 meters/pixel 。
- origin:图中左下角像素的二维位姿，如（x，y，yaw），yaw 逆时针旋转(yaw=0 表示没有旋转)。系统的很多部分现在忽略 yaw 值。
- occupied_thresh:像素占用率大于这个阈值则认为完全占用。
- free_thresh:像素占用率比该阈值小被则认为完全自由。
- negate:无论白色或黑色，占用或自由，语义应该是颠倒的（阈值的解释不受影响）。
- negate : Whether the white/black free/occupied semantics should be reversed (interpretation of thresholds is unaffected)

命令行工具

(1)Map_server

- map_server 是一个 ROS 节点，可以从磁盘读取地图并使用 ROS service 提供地图。
- 目前实现的 map_server 可将地图中的颜色值转化成三种占用值：自由 (0), 占用 (100), 和 未知 (-1).

- 这个工具的未来版本可能会使用 0 和 100 之间的值来表达更细致的占用度。
- 命令语法

```
$ map_server <map.yaml>
```

- 示例

```
$ rosrun map_server map_server mymap.yaml
```

- 注意：map data 可以通过指定 topic 或者 service 来提取。service 的方式最后可能要被废弃。
- 发布的主题

```
[map_metadata (nav_msgs/MapMetaData)][3]
```

- 通过指定话题获取地图的 metadata

```
[map (nav_msgs/OccupancyGrid)][4]
```

- 通过指定话题获取地图
- 服务

```
static_map ([nav_msgs/GetMap][5])
```

- 通过该服务来获取地图
- 参数

```
~frame_id (string, default: "map")
```

- 设置已发布的地图的坐标系（The frame to set in the header of the published map. ）

(2)map_saver

- map_saver 可以把地图保存到磁盘。 例如：从 SLAM mapping 服务中保存.
- 命令语法

```
$ rosrun map_server map_saver [-f mapname]
```

- Map_saver 获取地图数据，并把它写到 map.pgm 和 map.yaml。
- 使用-f 选项为指定地图的存放目录和名称。

- 命令示例

```
$ rosrun map_server map_saver -f /home/xxx/map/zhizaokongjian
```

- /home/xxx/map/为地图目录路径，zhizaokongjian 为地图名称，生成后得到 zhizaokongjian.yaml 和 zhizaokongjian.pgm 两个文件
- 订阅话题

map (nav_msgs/OccupancyGrid)

- 通过指定话题来获取地图

参考资料：

- http://wiki.ros.org/map_server
- http://blog.csdn.net/x_r_su/article/details/53392272
- http://blog.csdn.net/w383117613/article/details/46860075

ROS 与 navigation 教程-move_base 介绍

说明：

- 简单介绍 move_base 包的概念

概要

- move_base 包为一个 ation 提供了实现的途径（详情参见 actionlib 包)。假如在地图内给定一个目标，这个操作将尝试控制一个移动基座到达这个目标。
- move_base 节点将全局路径和局部路径规划程序链接在一起，以完成其全局导航任务。
- 它支持连接符合 nav_core 包中指定的 nav_core::BaseGlobalPlanner 接口的任一全局路径规划器和符合 nav_core::BaseLocalPlanner 接口的任一局部路径规划器。
- move_base 节点还维护两个 costmaps，一个用于全局路径规划器，一个用于局部路径规划器（参见 costmap_2d 软件包），用于完成导航任务。

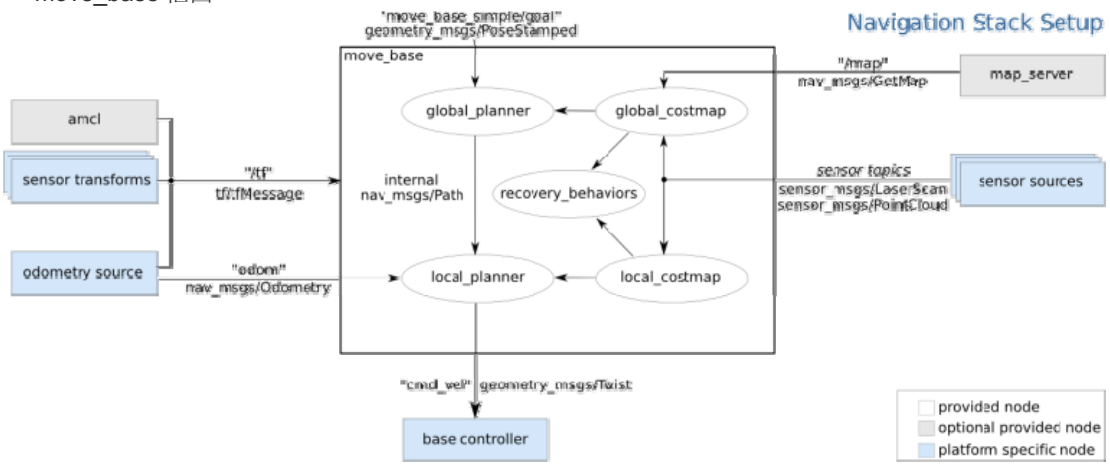
代码库

- 参考链接：https://github.com/ros-planning/navigation

- 注意：move_base 包允许您使用导航功能包集将机器人移动到所需的位置。 如果您只想在里程计坐标系中驱动 PR2 机器人，您可能对此教程感兴趣：pr2_controllers/Tutorials/Using the base controller with odometry and transform information

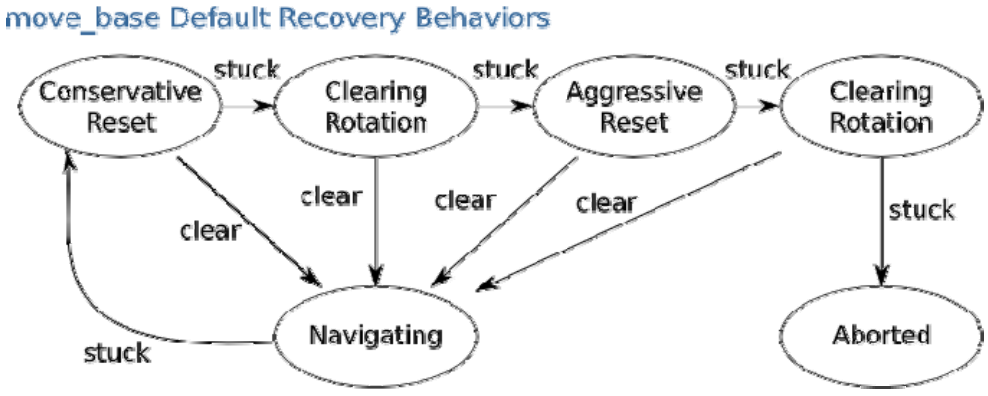
节点

- 该包提供了作为导航功能包集的主要组件的 move_base ROS 节点。 此节点及其配置选项的详细说明如下。
- move_base 框图



- move_base 节点提供了一个 ROS 接口，用于配置，运行和与机器人上的导航功能包集进行交互。
- move_base 节点的高级视图及其与其他组件的交互如上所示。蓝色部分基于机器人平台而变化，所有平台都能提供可选择的灰色部分和不可少的白色部分。
- 有关 move_base 节点和导航功能包集配置的更多信息，请参阅 navigation setup and configuration 教程。

(1) move_base 节点进行流程



- 在机器人上正确运行 move_base 节点（详情请看 navigation stack documentation），来让机器人尝试到达在使用者允许的误差范围内目标位置。
- 在没有动态障碍的情况下，move_base 节点最终运行结果要么是接近了规划的目标，要么是给用户返回失败信号。
- 当机器人认为它是卡住了，move_base 节点可以选择执行修复操作。默认情况下，move_base 节点采用如下所述行动来清理出的周围空间：
- 首先，用户指定区域以外的障碍物将从机器人的地图中清除。如果失败，机器人会更激进的将其可以就地旋转的矩形框外面的障碍物全部移除。
- 紧接着下一个就地旋转操作会被继续执行。如果所有这些操作都失效，机器人会认为目标设定不恰当并通知用户放弃操作。
- 这些修复操作可以通过 recovery_behaviors 参数来配置，也可以 recovery_behavior_enabled 参数 disable。

(2) move_base action API

- move_base 节点实现了 SimpleActionServer（请参阅 actionlib 文档），接收带有 geometry_msgs/PoseStamped 消息的目标信息。
- 虽然可以使用 ROS 直接与 move_base 节点通信，但推荐使用 SimpleActionClient 发送目标信息与其通信，这样可以追踪目标的执行状态。

(3) Action Subscribed Topics

```
move_base/goal ([move_base_msgs/MoveBaseActionGoal][10])
```

- move_base 的搜索目标

```
move_base/cancel ([actionlib_msgs/GoalID][11])
```

- 取消特定目标的请求

Action Published Topics

```
move_base/feedback ([move_base_msgs/MoveBaseActionFeedback][12])
```

- 反馈含有基座当前位置信息

```
move_base/status ([actionlib_msgs/GoalStatusArray][13])
```

- 提供有关发送到 move_base 操作的目标的状态信息。

```
move_base/result ([move_base_msgs/MoveBaseActionResult][14])
```

- move_base 操作的结果为空。

(4) Subscribed Topics

```
move_base_simple/goal ([geometry_msgs/PoseStamped][15])
```

- 对于不需要追踪目标的执行状态的用户，提供一个非 action 接口

(5) Published Topics

```
cmd_vel ([geometry_msgs/Twist][16])
```

- 移动基座执行速度命令

(6) Services

~make_plan (nav_msgs/GetPlan)

- 允许用户从 move_base 获取给定目标的路径规划，但不会执行该路径规划。

```
~clear_unknown_space ([std_srvs/Empty][18])
```

- 允许用户直接清除机器人周围的未知空间。非常适合于 costmap 停了很长时间后，在一个全新地方重新启动时候采用。
- Available in versions from 1.1.0-groovy

```
~clear_costmaps ([std_srvs/Empty][19])
```

- 允许用户命令 move_base 节点清除 costmap 中的障碍。
- 这可能会导致机器人装上物体，因此请谨慎使用。
- New in 1.3.1

(7) 参数

```
~base_global_planner (string, default: "navfn/NavfnROS" For 1.1+ series)
```

- 指定用于 move_base 的全局路径规划器的插件名称，该插件必须继承自此插件 nav_core 包中指定的 nav_core::BaseGlobalPlanner 接口。(1.0 series default: "NavfnROS")
- 有关插件的更多详细信息，请参阅 pluginlib 文档。

```
~base_local_planner (string, default: "base_local_planner/TrajectoryPlannerROS" For 1.1+ series)
```

- 指定用于 move_base 的局部路径规划器的插件名称，该插件必须继承自此插件 nav_core 包中指定的 nav_core::BaseGlobalPlanner 接口。(1.0 series default: "TrajectoryPlannerROS")

```
~recovery_behaviors (list, default: [{name: conservative_reset, type: clear_costmap_recovery/ClearCostmapRecovery}, {name: rotate_recovery, type: rotate_recovery/RotateRecovery}, {name: aggressive_reset, type: clear_costmap_recovery/ClearCostmapRecovery}] For 1.1+ series)
```

- 指定用于 move_base 的修复操作插件列表，当 move_base 不能找到有效的计划的时候，将按照这里指定的顺序执行这些操作。每个操作执行完成后，move_base 都会尝试生成有效的计划。如果该计划生成成功，move_base 会继续正常运行。否则，下一个修复操作启动执行。
- 这些修复操作插件必须继承自 nav_core::RecoveryBehavior 接口。(1.0 series default: [{name: conservative_reset, type: ClearCostmapRecovery}, {name: rotate_recovery, type: RotateRecovery}, {name: aggressive_reset, type: ClearCostmapRecovery}])
- 注意: 对于默认参数, aggressive_reset 行动将清理出一个 4 * ~/local_costmap/circumscribed_radius 的范围

```
~controller_frequency (double, default: 20.0)
```

- 以 Hz 为单位的速率运行控制循环并向基座发送速度命令。

```
~planner_patience (double, default: 5.0)
```

- 在空间清理操作执行前，路径规划器等待多长时间（秒）用来找出一个有效规划。
- How long the planner will wait in seconds in an attempt to find a valid plan before space-clearing operations are performed.

```
~controller_patience (double, default: 15.0)
```

- 在空间清理操作执行前，控制器会等待多长时间（秒）用来找出一个有效控制。

- How long the controller will wait in seconds without receiving a valid control before space-clearing operations are performed.

```
~conservative_reset_dist (double, default: 3.0)
```

- 当在地图中清理出空间时候，距离机器人几米远的障碍将会从 costmap 清除。
- 注意：该参数仅用于 move_base 使用了默认参数的情况。

```
~recovery_behavior_enabled (bool, default: true)
```

- 是否启用 move_base 修复机制来清理出空间

```
~clearing_rotation_allowed (bool, default: true)
```

- 决定做清理空间操作时候，机器人是否会采用原地旋转。
- 注意：该参数仅用于 move_base 使用了默认参数的情况，这意味着用户尚未将 recovery_behaviors 参数设置为任何自定义形式。
- Determines whether or not the robot will attempt an in-place rotation when attempting to clear out space. Note: This parameter is only used when the default recovery behaviors are in use, meaning the user has not set the recovery_behaviors parameter to anything custom.

```
~shutdown_costmaps (bool, default: false)
```

- 当 move_base 进入 inactive 状态时候，决定是否停用节点的 costmap

```
~oscillation_timeout (double, default: 0.0)
```

- 执行修复操作之前，允许的震荡时间是几秒。 值 0 意味着永不超时。New in navigation 1.3.1

```
~oscillation_distance (double, default: 0.5)
```

- 机器人需要移动多少距离才算作没有震荡。移动完毕后重置定时器计入参数~oscillation_timeout。New in navigation 1.3.1

```
~planner_frequency (double, default: 0.0)
```

- 全局路径规划器 loop 速率。如果设置这个为 0.0，当收到新目标点或者局部路径规划器报告路径不通时候全局路径规划器才启动。New in navigation 1.6.0

```
~max_planning_retries (int32_t, default: -1)
```

(8) Component APIs

- move_base 节点中使用的组件基本上自己都有对应 ROS APIs。
- 基于参数~base_global_planner, ~base_local_planner, and ~recovery_behaviors 的不同设置，使用的组件可能不同。
- 以下是 move_base 用到的默认组件的链接地址：
  - costmap_2d costmap_2d 包
  - nav_core nav_core::BaseGlobalPlanner 和 nav_core::BaseLocalPlanner 接口
  - base_local_planner 基座局部路径规划器
  - navfn 全局路径规划器
  - clear_costmap_recovery 用于清除 costmap 的修复操作
  - rotate_recovery 旋转修复操作

参考资料：

- http://wiki.ros.org/move_base
- http://blog.csdn.net/x_r_su/article/details/53376245

ROS 与 navigation 教程-move_base_msgs 介绍

说明：

- 介绍 move_base_msgs 的概念

概要

- move_base_msgs 能够保留 move_base 包的操作描述和相关消息

代码库

- 参考文献：https://github.com/ros-planning/navigation.git

Overview

- 这个包包含用于与 move_base 节点进行通信的消息。
- 这些消息是基于 MoveBase.action 操作规范自动生成的。

MoveBase.action

```
geometry_msgs/PoseStamped target_pose
---
---
geometry_msgs/PoseStamped base_position
```

- 该 target_pose 是导航功能包集试图实现的目标。
- 作为反馈给出的 base_position 是 tf 报告基座在地图上的当前位置。
- 对于 move_base 节点，当尝试实现目标时，target_pose 会被投影到 XY 平面中，Z 轴朝上。

ROS 与 navigation 教程-fake_localization 介绍

说明：

- 简单介绍 fake_localization 的概念

概述

- fake_localization 是一个 ROS 节点，用来简单的转发 odometry 信息。
- fake_localization 包提供了一个单一的 ROS 节点————fake_localization, 用来替代定位系统，并且提供了 amcl 定位算法 ROS API 的子集。
- 该节点在仿真中被频繁使用，是一种不需要大量计算资源就能进行定位的方式。

Nodes

(1) Subscribed Topics

```
base_pose_ground_truth ([nav_msgs/Odometry][1])
```

- 仿真器发布的机器人位置信息。

```
initialpose ([geometry_msgs/PoseWithCovarianceStamped][2])
```

- 允许使用像 rviz 或 nav_view 这样的工具来设置 fake_localization 的自定义位姿。
- Allows for setting the pose of fake_localization using tools like rviz or nav_view to give a custom offset from the ground truth source being published.

(2) Published Topics

```
amcl_pose ([geometry_msgs/PoseWithCovarianceStamped][3])
```

- 通过仿真器报告的位姿。

```
particlecloud ([geometry_msgs/PoseArray][4])
```

- 在 rviz 和 nav_view 中使用，来可视化机器人位姿的粒子云。

(3) Parameters

```
~odom_frame_id (string, default: "odom")
```

- 里程计坐标系名字。

```
~delta_x (double, default: 0.0)
```

- 地图坐标系与仿真器坐标系原点在 x 轴方向的偏移。

```
~delta_y (double, default: 0.0)
```

- 地图坐标系与仿真器坐标系原点在 y 轴方向的偏移。

```
~delta_yaw (double, default: 0.0)
```

- 地图坐标系与仿真器坐标系原点在 yaw 偏航角的偏移。

```
~global_frame_id (string, default: /map)
```

- 指定使用 tf 发布 global_frame_id→odom_frame_id 转换的坐标系。

```
~base_frame_id (string, default: base_link)
```

- 机器人基座坐标系。 New in 1.1.3

(4) Provided tf Transforms

```
/map → <value of odom_frame_id parameter>
```

- 使用 tf 发布仿真器传递过来的坐标系转换

参考资料

- http://wiki.ros.org/fake_localization
- http://blog.csdn.net/x_r_su/article/details/53395760

ROS 与 navigation 教程-voxel_grid 介绍

说明：

- 介绍 voxel_grid 的概念

概述

- voxel_grid 实现里高效的 3D voxel grid。
- 在单元格状态下，占用网格可以支持的 3 种不同表示形式：标记，自由或未知。
- 由于依赖按位与和或的整数运算的底层实现，体素网格仅支持每个体素列 16 个不同的级别。
- 然而，这种限制产生了与标准 2D 结构相当的光栅跟踪和单位标记性能，但其明显快于大多数的 3D 结构。

代码库

- 参考资料： https://github.com/ros-planning/navigation.git

ROS 与 navigation 教程-global_planner 介绍

说明：

- 介绍 global_planner 不同参数化实例和相关的 API

概述

- global_planner 是一个路径规划器节点。
- 这个包为导航实现了一种快速，内插值的全局路径规划器， 继承了 nav_core 包中 nav_core::BaseGlobalPlanner 接口，该实现相比 navfn 使用更加灵活。

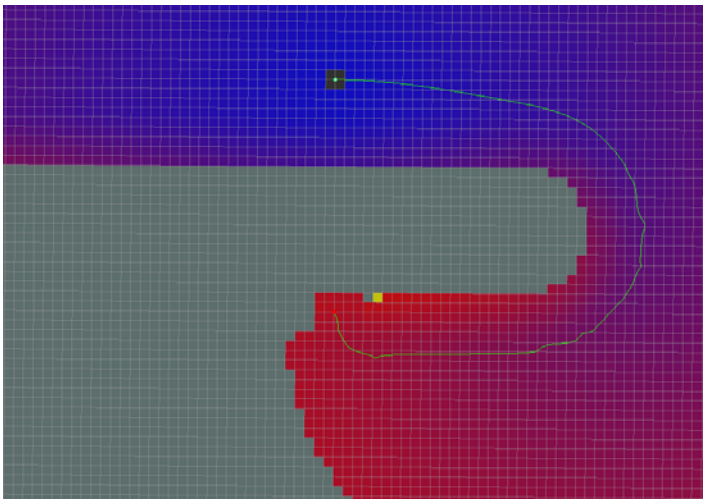
代码库

- 参考资料：https://github.com/ros-planning/navigation

不同参数化实例

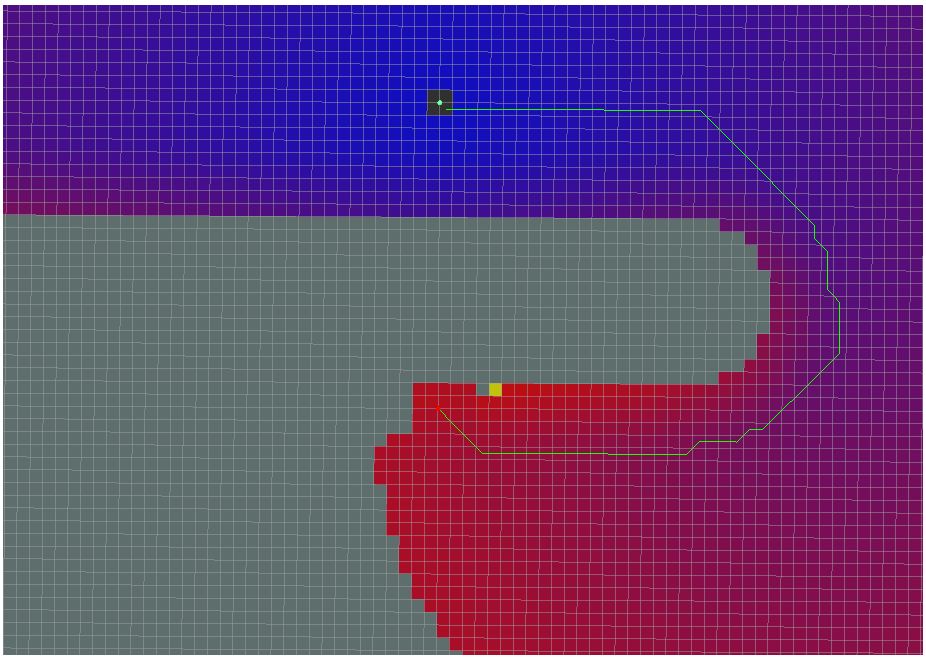
（1）标准行为示例

- 所有参数使用默认值



（2）栅格路径示例

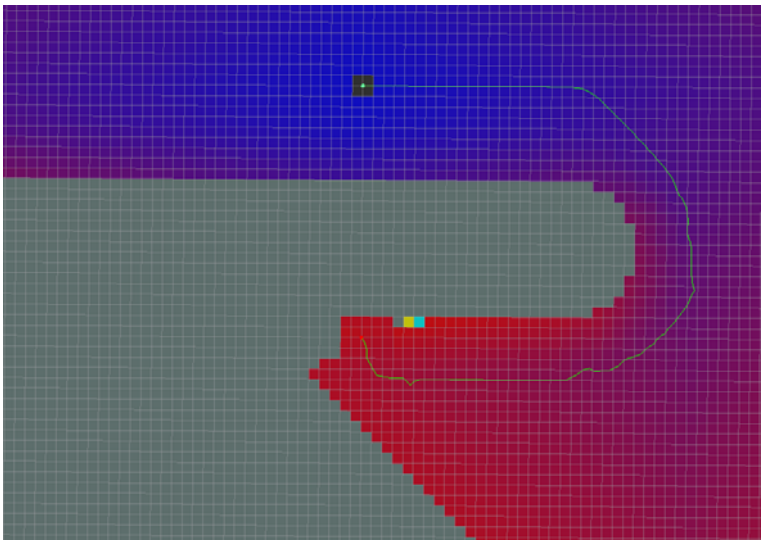
- use_grid_path=True



- 路径遵循栅格边界。

(3) Simple Potential Calculation

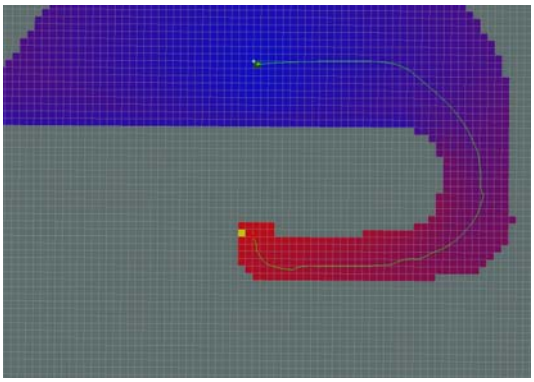
- use_quadratic=False



- Slightly different calculation for the potential. Note that the original potential calculation from navfn is a quadratic approximation. Of what, the maintainer of this package has no idea.

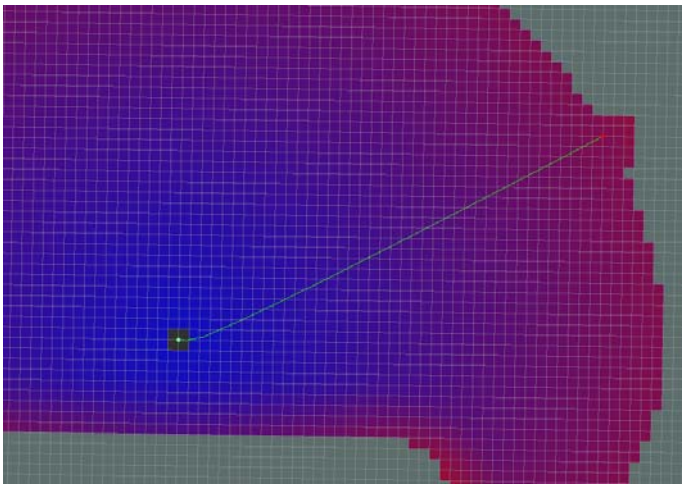
(4) A* Path

- use_dijkstra=False

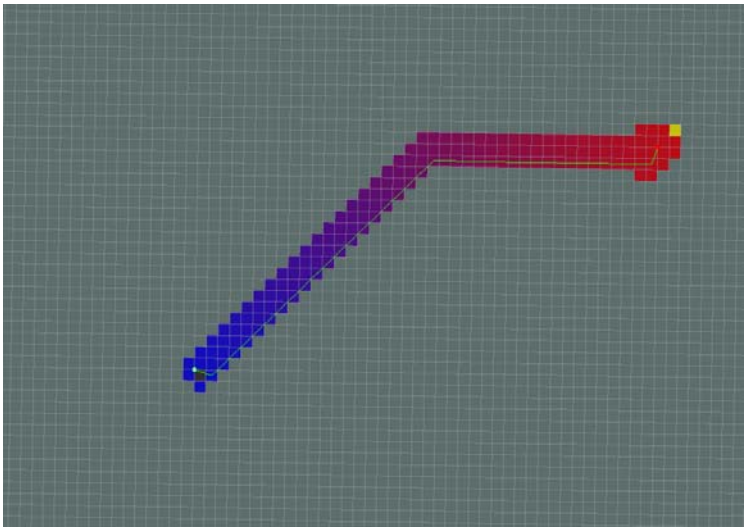


- Note that a lot less of the potential has been calculated (indicated by the colored areas). This is indeed faster than using Dijkstra's, but has the effect of not necessarily producing the same paths. Another thing to note is that in this implementation of A*, the potentials are computed using 4-connected grid squares, while the path found by tracing the potential gradient from the goal back to the start uses the same grid in an 8-connected fashion. Thus, the actual path found may not be fully optimal in an 8-connected sense. (Also, no visited-state set is tracked while computing potentials, as in a more typical A* implementation, because such is unnecessary for 4-connected grids). To see the differences between the behavior of Dijkstra's and the behavior of A*, consider the following example.

(4.1) Dijkstra's

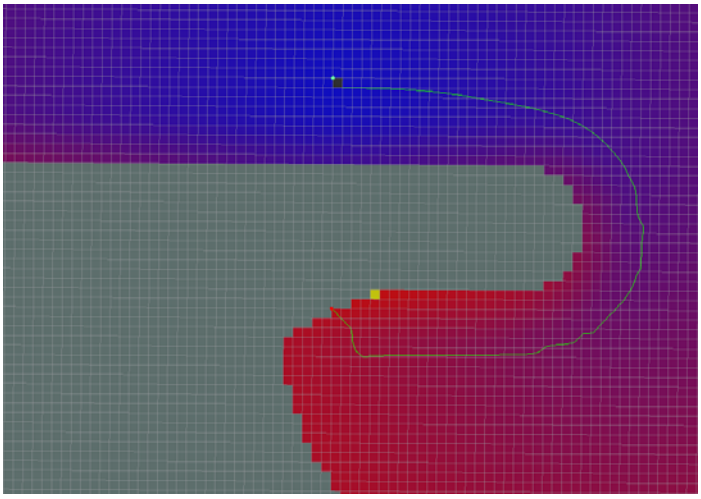


(4.2) A*



(5) Old Navfn Behavior

- old_navfn_behavior=True For reproducing paths just like NavFn did.



- 注意
  - 地图上路径的起始位置与实际的起始位置不匹配
  - 路径的最末端沿着栅格线移动。
  - 所有坐标稍微偏移了一半的网格单元格

ROS API

(1) Published Topics

```
~<name>/plan ([nav_msgs/Path][11])
```

- The last plan computed, published every time the planner computes a new path, and used primarily for visualization purposes.

(2)Parameters

```
~<name>/allow_unknown (bool, default: true)
```

- 指定是否允许路径规划器在未知空间创建路径规划。
- 注意：如果您使用带有体素或障碍层的分层 costmap_2d costmap，还必须将该层的 track_unknown_space 参数设置为 true，否则会将所有未知空间转换为可用空间。
- Specifies whether or not to allow the planner to create plans that traverse unknown space. NOTE: if you are using a layered costmap_2d costmap with a voxel or obstacle layer, you must also set the track_unknown_space param for that layer to be true, or it will convert all your unknown space to free space (which planner will then happily go right through).

```
~<name>/default_tolerance (double, default: 0.0)
```

- 路径规划器目标点的公差范围。
- 在默认公差范围内，路径规划器将尝试创建一个尽可能接近到达指定的目标的路径规划。

```
~<name>/visualize_potential (bool, default: false)
```

- 指定是否通过可视化 PointCloud2 计算的潜在区域。

```
~<name>/use_dijkstra (bool, default: true)
```

- 如果为 ture，则使用 dijkstra 算法。 否则使用 A *算法。

```
~<name>/use_quadratic (bool, default: true)
```

- If true, use the quadratic approximation of the potential. Otherwise, use a simpler calculation.

```
~<name>/use_grid_path (bool, default: false)
```

- 如果为 true，沿着栅格边界创建路径。 否则，使用梯度下降的方法。

```
~<name>/old_navfn_behavior (bool, default: false)
```

- 如果你想要 global_planner 准确反映 navfn 的行为，此项设置为 true。

参考资料

- http://wiki.ros.org/global_planner?distro=lunar
- http://blog.csdn.net/x_r_su/article/details/53391462

ROS 与 navigation 教程-base_local_planner 介绍

说明：

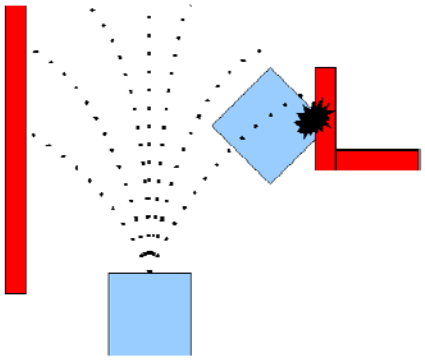
- 介绍 base_local_planner 的概念及相关接口

代码库

- 参考链接：https://github.com/ros-planning/navigation

概述

- 这个包使用 Trajectory Rollout and Dynamic Window approaches 来做平面上运动的机器人局部导航，控制器基于给定的路径规划和 costmap 生成速度命令后发送给移动基座。
- 该包适用于全向移动和非全向移动机器人，机器人轮廓可以表示为凸多边形或者圆。
- 这个包进行了 ROS 封装，继承了 BaseLocalPlanner 接口，且可以在启动文件中设置 ROS 参数。
- base_local_planner 包提供了驱动底座在平面移动的控制器，控制器可以连接路径规划器和机器人基座。
- 为了让机器人从起始位置到达目标位置，路径规划器使用地图创建运动轨迹。
- 向目标移动的路上，路径规划器至少需要在地图上的机器人周围创建一个可以表示成栅格地图的评价函数。
- This value function encodes the costs of traversing through the grid cells.
- 该控制器任务就是用这个评价函数确定发送速度和角度（dx，dy，dtheta velocities）给机器人基座。



- Trajectory Rollout and Dynamic Window Approach (DWA)算法基本理念如下:
  - 采样机器人当前的状态。Discretely sample in the robot's control space (dx,dy,dtheta)
  - 用采样的离散点做前向模拟，基于机器人当前状态，预测如果使用空间采样点的速度运动一段时间可能出现的情况。
  - 评价前向模拟的每条轨迹，评价标准包括（接近障碍，接近目标，接近全局路径和速度）。丢弃不合法的轨迹（如可能碰到障碍物的轨迹）。
  - 根据打分，选择最优路径，并将其对应速度发送给基座。
  - 重复上面步骤。
- DWA 与 Trajectory Rollout 的区别主要是在机器人的控制空间采样差异。Trajectory Rollout 采样点来源于整个前向模拟阶段所有可用速度集合，而 DWA 采样点仅仅来源于一个模拟步骤中的可用速度集合。这意味着相比之下 DWA 是一种更加有效算法，因为其使用了更小采样空间；然而对于低加速度的机器人来说可能 Trajectory Rollout 更好，因为 DWA 不能对常加速度做前向模拟。
- 在实践中，我们经过多次实验发现 2 种方法性能基本相当，这样的话我们推荐使用效率更高的 DWA 算法。
- 一些有用的参考链接:
  - 在 LAGR 机器人上使用的 Trajectory Rollout 算法的讨论：Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain ".
  - The Dynamic Window Approach to local control： D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance".
  - An previous system that takes a similar approach to control：Alonzo Kelly. "An Intelligent Predictive Controller for Autonomous Vehicles".

(1) Map Grid

- 为了有效地评价轨迹，使用了地图栅格。
- 每个控制周期，都会在机器人周围创建栅格（大小为局部 costmap）区域，并且全局路径会被映射到这个区域上。 这意味着有一定栅格将被标记为到路径点距离为 0，到目标距离为 0。接着利用传播算法可以将剩下的标记，记录它们到各自最近的标记为 0 的点的距离。
- 然后，利用地图栅格进行轨迹评价。

- 全局目标一般不会出现在地图栅格标记的小区域内，所以为接近目标进行轨迹评价时，这个目标应该是个局部目标，这意味着该小区域内第一个路径点一定是该区域外还有其连续点的那个点。 该小区域的尺寸大小由 move_base 确定。

震荡抑制

- 当在 x, y, or theta 维度出现震荡后，正负值会连续出现。因此，为了抑制震荡影响，当机器人在某方向移动时，对下一个周期的与其相反方向标记为无效，直到机器人从标记震荡的位置处离开一定距离。

通用局部路径规划

- 从 ROS groovy 版本开始有了一种新的局部路径规划包（DWA）。 这种实现更模块化，其可复用的代码使客制化一种局部路径规划器时候更加容易。 base_local_planner 基础代码已经被扩展，增加了几个头文件和新的类。
- 局部规划器的工作原则是在每一个控制周期搜索最佳的局部路径规划。首先局部规划器会生成一些候选轨迹。其次在检测生成的轨迹是否会与障碍物碰撞。如果没有，规划器就会评价且比较选择出最好的轨迹。
- 很显然，由于机器人外形（和制动器）以及应用领域的差异，这种工作原则的实例化会不尽相同。

以下的类和接口遵照通用局部路径规划器工作原则允许进行不同实例化。可以以 dwa_local_planner 为模板加入自己的代价函数或者轨迹生成器，来创建自定义的局部路径规划器。

(1) TrajectorySampleGenerator

- 该接口描述了一种可以生成很多轨迹发生器，每调用一次 nextTrajectory()就会返回一个新的轨迹。
- SimpleTrajectoryGenerator 类可以使用 trajectory rollout 或 DWA 原理来生成概述中描述的轨迹。

(2) TrajectoryCostFunction

- 这个接口包含了最重要的函数 scoreTrajectory(Trajectory &traj)，该函数输入轨迹后会输出轨迹评价分数。
- 如果输出负分数意味着轨迹无效；输出分数为正值，对于 cost 函数来说值越小越好。
- 每个 cost 函数有一个比例因子，与其它 cost 函数比较时候，通过调节比例因子，cost 函数影响可以被改变。
- base_local_planner 包附带了一些在 PR2 上使用过的 cost 函数，如下所述。

(3) SimpleScoredSamplingPlanner

- 这是轨迹搜索的一种简单实现，利用了 TrajectorySampleGenerator 产生的轨迹和一系列 TrajectoryCostFunction。 它会一直调用 nextTrajectory()直到发生器停止生成轨迹。对于每一个生成的轨迹，将会把列表中的 cost 函数都循环调用，并把 cost 函数返回的正值，负值丢弃。
- 利用 cost 函数的比例因子， 最佳轨迹就是 cost 函数加权求和后最好的的那条轨迹。

(4) Helper classes

(4.1) LocalPlannerUtil

- 该帮助接口提供了通用的接口可供所有规划器使用。它管理当前的全局规划，当前的运动约束，以及当前的 cost 地图（感知障碍物的局部 cost 地图）。

(4.2) OdometryHelperRos

- 该类为机器人提供 odometry 信息。

(4.3) LatchedStopRotateController

- 理想情况下，局部路径规划器可以让机器人准确停到它应该停止地方。然而在现实中，由于传感器噪声和执行器的不稳定性，机器人会接近到达目标，但其会继续移动，这不是我们想要的结果。
- Ideally a local planner will make a robot stop exactly where it should. In practice however, due to sensor noise and actuator uncertainty, it may happen that the robot approaches the target spot but moves on. This can lead to undesired robot behavior of oscilatting on a spot.
- LatchedStopRotateController 是一个不错的控制器，当机器人足够靠近目标时可以迅速启用。
- 然后，控制器将执行完全停止和原地旋转朝向目标方向的操作，无论在停止后的机器人位置是否超出目标位置公差范围。
- The LatchedStopRotateController is a Controller that can be used as soon as the robot is close enough to the goal. The Controller will then just perform a full stop and a rotation on the spot towards the goal orientation, regardless of whether the robot position after the full stop leads the robot outside the goal position tolerance.

(5) Cost Functions

(5.1) ObstacleCostFunction

- 该 cost 函数类基于感知到的障碍物评价轨迹。如果轨迹经过障碍物则返回负值，其它返回 0。

(5.2) MapGridCostFunction

- 该 cost 函数类基于轨迹与全局路径或者目标点的接近程度来评价轨迹。它尝试对所有轨迹使用相同的到某个目标或者路径距离的预计算图来优化计算速度。
- 在 dwa_local_planner 中，因目的不同（为让轨迹尽可能接近全局路径，为让机器人朝着局部目标前进，还有为让机器人的头保持指向局部目标），该 cost 函数具体实现也会不尽相同。因该 cost 函数使用了试探机制，因此如果使用了不合适的参数可能给出的结果很差甚至根本不能工作。

(5.3) OscillationCostFunction

- 该 cost 函数类用以减少一定程度的震荡。虽然该 cost 函数能有效防止震荡，但是如果使用不合适的参数也可能会得不到一些好的解决方案。

(5.4) PreferForwardCostFunction

- 该 cost 函数类适用于类似 PR2 那种在机器人前方有很好传感器（如 tilting laser）布置的机器人。
- 该 cost 函数鼓励前向运动，惩罚后向或者其它周围方向性运动。但是这种特性在某些领域机器人可能并不是所期望的，所以仅适合于特定应用的机器人。

TrajectoryPlannerROS

- base_local_planner::TrajectoryPlannerROS 是对 base_local_planner::TrajectoryPlanner 的 ROS 封装。
- 它在初始化时确定的 ROS 命名空间内运行，该接口继承了 nav_core 包的 nav_core::BaseLocalPlanner 接口。
- 如下是 base_local_planner::TrajectoryPlannerROS 的一个应用案例:

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <base_local_planner/trajectory_planner_ros.h>

...

tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS costmap("my_costmap", tf);

base_local_planner::TrajectoryPlannerROS tp;
tp.initialize("my_trajectory_planner", &tf, &costmap);
```

(1) API Stability

- C++ API 是稳定的。

- ROS API 也是稳定的。

(1.1) Published Topics

```
~<name>/global_plan ([nav_msgs/Path][6])
```

- 表示的是局部路径规划器目前正在跟随的全局路径规划中的一部分，主要用于可视化。

```
~<name>/local_plan ([nav_msgs/Pat][7]h)
```

- 表示的是上一个周期局部规划或者轨迹得分最高者，主要用于可视化。

```
~<name>/cost_cloud ([sensor_msgs/PointCloud2][8])
```

- 用于表示规划的 cost 栅格，也是用于可视化目的。参数 publish_cost_grid_pc 用来使用或者关闭该可视化。New in navigation 1.4.0

(1.2) Subscribed Topics

```
odom ([nav_msgs/Odometry][9])
```

- 该 Odometry 信息用于向局部规划器提供当前机器人的速度。
- 这个里程消息中的速度信息被假定使用的坐标系与 TrajectoryPlannerROS 对象内 cost 地图的 robot_base_frame 参数指定的坐标系相同。
- 有关该参数详细介绍请参照 costmap_2d 包。

(2) Parameters

- 有许多 ROS 参数可以用来自定义 base_local_planner::TrajectoryPlannerROS 的行为。
- 这些参数分为几类：机器人配置，目标公差，前向仿真，轨迹评分，防振和全局计划。
- These parameters are grouped into several categories: robot configuration, goal tolerance, forward simulation, trajectory scoring, oscillation prevention, and global plan.

(2.1) Robot Configuration Parameters

```
~<name>/acc_lim_x (double, default: 2.5)
```

- 机器人在 x 方向的最大加速度，单位 meters/sec^2 。

```
~<name>/acc_lim_y (double, default: 2.5)
```

- 机器人在 y 方向的最大加速度，单位 meters/sec^2 。

```
~<name>/acc_lim_theta (double, default: 3.2)
```

- 机器人的最大角加速度，单位 radians/sec^2 。

```
~<name>/max_vel_x (double, default: 0.5)
```

- 基座允许的最大线速度，单位 meters/sec 。

```
~<name>/min_vel_x (double, default: 0.1)
```

- 基座允许的最小线速度，单位 meters/sec 。
- 设置的最小速度需要保证基座能够克服摩擦。

```
~<name>/max_vel_theta (double, default: 1.0)
```

- 基座允许的最大角速度，单位 radians/sec 。

```
~<name>/min_vel_theta (double, default: -1.0)
```

- 基座允许的最小角速度，单位 radians/sec 。

```
~<name>/min_in_place_vel_theta (double, default: 0.4)
```

- 原地旋转时，基座允许的最小角速度，单位 radians/sec 。

```
~<name>/backup_vel (double, default: -0.1)
```

- DEPRECATED (use escape_vel):Speed used for backing up during escapes in meters/sec. Note that it must be negative in order for the robot to actually reverse. A positive speed will cause the robot to move forward while attempting to escape.

```
~<name>/escape_vel (double, default: -0.1)
```

- 表示机器人的逃离速度，即背向相反方向行走速度，单位为 meters/sec 。
- 该值必需设为负值，但若设置为正值，机器人会在执行逃离操作时向前移动。

```
~<name>/holonomic_robot (bool, default: true)
```

- 确定是否为全向轮或非全向轮机器人生成速度指令。

- 对于全向轮机器人，可以向基座发出施加速度命令。 对于非全向轮机器人，不会发出速度指令。

以下参数仅在 holonomic_robot 设置为 true 时使用：

```
~<name>/y_vels (list, default: [-0.3, -0.1, 0.1, 0.3])
```

- The strafing velocities that a holonomic robot will consider in meters/sec

（2.2）目标公差参数（Goal Tolerance Parameters）

```
~<name>/yaw_goal_tolerance (double, default: 0.05)
```

- The tolerance in radians for the controller in yaw/rotation when achieving its goal

```
 ~<name>/xy_goal_tolerance (double, default: 0.10)
```

- The tolerance in meters for the controller in the x & y distance when achieving a goal

```
~<name>/latch_xy_goal_tolerance (bool, default: false)
```

- If goal tolerance is latched, if the robot ever reaches the goal xy location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so. - New in navigation 1.3.1

（2.3）前向仿真参数（Forward Simulation Parameters）

```
~<name>/sim_time (double, default: 1.0)
```

- 前向模拟轨迹的时间，单位为 seconds 。
- The amount of time to forward-simulate trajectories in seconds.

```
~<name>/sim_granularity (double, default: 0.025)
```

- 在给定轨迹上的点之间的步长，单位为 meters 。
- The step size, in meters, to take between points on a given trajectory

```
~<name>/angular_sim_granularity (double, default: ~<name>/sim_granularity)
```

- 给定角度轨迹的弧长，单位为 radians 。
- The step size, in radians, to take between angular samples on a given trajectory. - New in navigation 1.3.1

```
~<name>/vx_samples (integer, default: 3)
```

- x 方向速度的样本数。
- The number of samples to use when exploring the x velocity space .

```
~<name>/vtheta_samples (integer, default: 20)
```

- 角速度的样本数。
- The number of samples to use when exploring the theta velocity space

```
~<name>/controller_frequency (double, default: 20.0)
```

- 调用控制器的频率，单位为 Hz 。
- Uses searchParam to read the parameter from parent namespaces if not set in the namespace of the controller. For use with move_base, this means that you only need to set its "controller_frequency" parameter and can safely leave this one unset.

（2.4）轨迹评分参数（Trajectory Scoring Parameters）

```
   cost =
  pdist_scale * (distance to path from the endpoint of the trajectory in map cells or mete
rs depending on the meter_scoring parameter)
  + gdist_scale * (distance to local goal from the endpoint of the trajectory in map cells
 or meters depending on the meter_scoring parameter)
  + occdist_scale * (maximum obstacle cost along the trajectory in obstacle cost (0-254))
```

- 该 cost 函数使用如上公式给每条轨迹评分

```
~<name>/meter_scoring (bool, default:false)
```

- 默认 false 情况下，用单元格作为打分的单位，反之，则用米。
- Whether the gdist_scale and pdist_scale parameters should assume that goal_distance and path_distance are expressed in units of meters or cells. Cells are assumed by default.

```
~<name>/pdist_scale (double, default: 0.6)
```

- The weighting for how much the controller should stay close to the path it was given.

```
~<name>/gdist_scale (double, default: 0.8)
```

- The weighting for how much the controller should attempt to reach its local goal, also controls speed.

```
~<name>/occdist_scale (double, default: 0.01)
```

- The weighting for how much the controller should attempt to avoid obstacles .

```
~<name>/heading_lookahead (double, default: 0.325)
```

- How far to look ahead in meters when scoring different in-place-rotation trajectories .

```
~<name>/heading_scoring (bool, default: false)
```

- 是否根据机器人前进路径的距离进行评分。
- Whether to score based on the robot's heading to the path or its distance from the path.

```
~<name>/heading_scoring_timestep (double, default: 0.8)
```

- How far to look ahead in time in seconds along the simulated trajectory when using heading scoring .

```
~<name>/dwa (bool, default: true)
```

- 是否使用 Dynamic Window Approach (DWA)，或者是否使用 Trajectory Rollout。
- Whether to use the Dynamic Window Approach (DWA)_ or whether to use Trajectory Rollout (NOTE: In our experience DWA worked as well as Trajectory Rollout and is computationally less expensive. It is possible that robots with extremely poor acceleration limits could gain from running Trajectory Rollout, but we recommend trying DWA first.)

```
~<name>/publish_cost_grid_pc (bool, default: false)
```

- Whether or not to publish the cost grid that the planner will use when planning. When true, a sensor_msgs/PointCloud2 will be available on the ~/cost_cloud topic. Each point cloud represents the cost grid and has a field for each individual scoring function component as well as the overall cost for each cell, taking the scoring parameters into account. New in navigation 1.4.0

```
~<name>/global_frame_id (string, default: odom)
```

- 设置 cost_cloud 工作坐标系，应该与局部 cost 地图的 global 坐标系一致。New in navigation 1.4.0

(2.5) Oscillation Prevention Parameters

```
~<name>/oscillation_reset_dist (double, default: 0.05)
```

- 机器人必须运动多少米远后才能复位震荡标记。

(2.6) Global Plan Parameters

```
~<name>/prune_plan (bool, default: true)
```

- 定义当机器人是否边沿着路径移动时，边抹去已经走过的路径规划。 设置为 true 时，表示当机器人移动 1 米后，将 1 米之前的 global 路径点一个一个清除。（包括全局的 global path 和局部的 global path）
- Defines whether or not to eat up the plan as the robot moves along the path. If set to true, points will fall off the end of the plan once the robot moves 1 meter past them.

TrajectoryPlanner

- base_local_planner::TrajectoryPlanner 实现了 DWA and Trajectory Rollout 算法。
- 如果要在 ROS 中想使用 base_local_planner::TrajectoryPlanner，请使用 TrajectoryPlannerROS wrapper。

(1) API Stability

- 虽然 C++ API 是稳定的。 但是仍然推荐使用 TrajectoryPlannerROS wrapper 而不是 base_local_planner::TrajectoryPlanner。

参考文章

- http://wiki.ros.org/base_local_planner
- http://blog.csdn.net/x_r_su/article/details/53380545
- http://blog.sina.com.cn/s/blog_135e04d0a0102zpl1.html#cmt_3076623

ROS 与 navigation 教程-carrot_planner 介绍

说明：

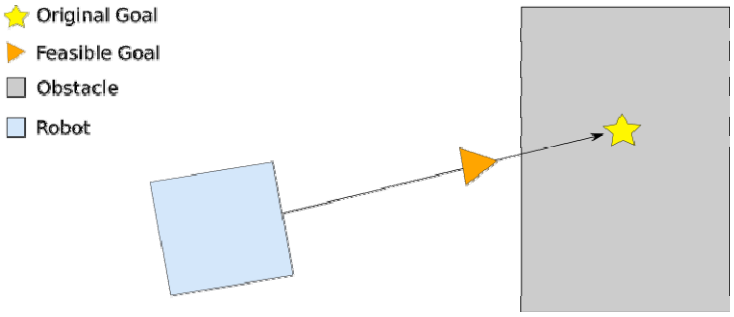
- 简单介绍 nav-carrot_planner 的概念和部分参数的含义。

代码库

- https://github.com/ros-planning/navigation

概述

- 这个规划器试图找到适当的位置来使机器人跟随。
- It does this by moving back along the vector between the robot and the goal point.
- carrot_planner :: CarrotPlanner 是一个简单的全局路径规划器，其继承里 nav_core :: BaseGlobalPlanner 接口，可以用作 move_base 节点的全局路径规划器的插件。
- 规划器获取用户指定的目标位置，检查用户指定的位置是否在障碍区中。
- 如果用户指定的位置在障碍区中，规划器就会在机器人与指定位置的已规划路径中寻找一个在障碍区外的可行的目标位置，然后再将此目标位置发送给局部规划器或者控制器。
- carrot planner 允许机器人机器人尽可能靠近用户指定的目标点。



CarrotPlanner

- carrot_planner :: CarrotPlanner 对象是通过 ROS 的封装来实现的它的功能。
- 它在初始化时指定的 ROS 命名空间内运行。

carrot_planner :: CarrotPlanner 对象的创建示例：

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <carrot_planner/carrot_planner.h>

...
tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS costmap("my_costmap", tf);

carrot_planner::CarrotPlanner cp;
cp.initialize("my_carrot_planner", &costmap);
```

(1) API Stability

- C++ API 是稳定的。
- ROS API 也是稳定的。

(2) ROS Parameters

```
~<name>/step_size (double, default: Resolution of the associated costmap)
```

- The size steps to take backward in meters along the vector between the robot and the user-specified goal point when attempting to find a valid goal for the local planner.

```
~<name>/min_dist_from_robot (double, default: 0.10)
```

- The minimum distance from the robot in meters at which a goal point will be sent to the local planner.

(3) C++ API

- The C++ carrot_planner::CarrotPlanner class 继承里 nav_core :: BaseGlobalPlanner 接口,。（详细参阅：CarrotPlanner Documentation.）

参考资料

- http://wiki.ros.org/carrot_planner

ROS 与 navigation 教程-teb_local_planner 介绍

说明：

- 介绍了 teb_local_planner 的概念和参数

参考代码

- https://github.com/rst-tu-dortmund/teb_local_planner

概要

- teb_local_planner 包是 2D 导航功能包中 base_local_planner 的插件实现。
- Timed Elastic Band 基础方法在轨迹执行时间，障碍物分离以及运行时遵守 kinodynamic 约束方面，局部优化了机器人的轨迹。
- 该软件包实现了一种在线优化的局部轨迹规划器，作为 ROS 导航包的插件用于导航和控制移动机器人。

- 在运行时，优化由全局路径规划器生成的初始轨迹，以便最小化轨迹执行时间（时间最优目标），与障碍物分离，并遵守诸如满足最大速度和加速度的动力学约束。
- 目前的实现符合非全向轮机器人（差速驱动机器人和四驱驱动机器人）的运动学。 （在 Kinetic 版本开始，该包支持全向轮机器人）
- 通过求解 sparse scalarized multi-objective 优化问题有效地获得了最优轨迹。 用户可以为优化问题设置权重，以便在目标冲突的情况下指定操作。
- “Timed-Elastic-Band”方法在：
  - Rösmann C., Feiten W., Wösch T., Hoffmann F. and Bertram. T.: Trajectory modification considering dynamic constraints of autonomous robots. Proc. 7th German Conference on Robotics, Germany, Munich, 2012, pp 74–79.
  - Rösmann C., Feiten W., Wösch T., Hoffmann F. and Bertram. T.: Efficient trajectory optimization using a sparse model. Proc. IEEE European Conference on Mobile Robots, Spain, Barcelona, 2013, pp. 138–143.
- 局部路径规划器如 Timed Elastic Band 经常被卡住在局部最优轨迹上，因为他们无法跨越障碍物。所以它进行了扩展，并行优化不同拓扑的可通过轨迹的子集。局部规划器能够切换到候选集中的当前全局最优轨迹，同时通过利用同源/同伦类的概念获得不同的拓扑。
- 将会很快提供更多有关此扩展的信息。 相关文章已发表： Rösmann C., Hoffmann F. and Bertram T.: Planning of Multiple Robot Trajectories in Distinctive Topologies, Proc. IEEE European Conference on Mobile Robots, UK, Lincoln, Sept. 2015
- Use GitHub to report bugs or submit feature requests. View active issues

Video

- 以下视频介绍了该软件包的功能，并显示了仿真和真实机器人情况下的示例。
- youtube 视频链接：
- 以下视频介绍了 0.2 版本的功能(支持四驱驱动机器人和 costmap 转换):
- youtube 视频链接：

Node API

(1) Topics

(1.1) Published Topics

```
~<name>/global_plan ([nav_msgs/Path][4])
```

- Global plan that the local planner is currently attempting to follow.
- 主要用于可视化。

```
~<name>/local_plan ([nav_msgs/Path][5])
```

- teb_local_planner 优化和跟踪的局部路径规划或轨迹。
- 主要用于可视化。

```
~<name>/teb_poses ([geometry_msgs/PoseArray][6])
```

- 当前局部路径规划的（discrete pose list）离散位姿列表（SE2）。
- 主要用于可视化。

```
~<name>/teb_markers ([visualization_msgs/Marker][7])
```

- teb_local_planner 通过具有不同命名空间的标记来提供规划场景的附加信息。
- 命名空间 PointObstacles 和 PolyObstacles：可视化在优化时当前考虑的所有点和多边形障碍物。
- 可视化替代拓扑中所有找到的和优化的轨迹（仅当启用并行计划时）。
- 将会发布更多的信息，如优化足迹模型。

```
~<name>/teb_feedback ([teb_local_planner/FeedbackMsg][10])
```

- 反馈消息含有包括速度曲线和时间信息以及障碍物列表的已规划轨迹。
- 主要用于评估和调试。
- 需启用 ~/publish_feedback 参数。

(1.2) Subscribed Topics

```
~<name>/odom ([nav_msgs/Odometry][11])
```

- 给局部路径规划器提供机器人当前速度的里程计信息。
- 可以通过重新映射或修改参数～ / odom_topic 来更改此话题。

```
~<name>/obstacles ([teb_local_planner/ObstacleMsg][12])
```

- 提供自定义的障碍物如点，线或多边形（另外或可代替 costmap 的障碍）。

(2) Parameters

- teb_local_planner 包允许用户设置参数以便自定义机器人的行为。
- 这些参数分为几类：机器人配置，目标公差，轨迹配置，障碍，优化，特殊拓扑中的规划和其他参数。
- 其中一些参数和 base_local_planner 包中的参数是一样的，许多（但不是全部）参数可以在运行时使用 rqt_reconfigure 进行修改。

(2.1) Robot Configuration Parameters

```
~<name>/acc_lim_x (double, default: 0.5)
```

- 机器人的最大线加速度，单位为 meters/sec^2 。

```
~<name>/acc_lim_theta (double, default: 0.5)
```

- 机器人的最大角加速度，单位为 radians/sec^2 。

```
~<name>/max_vel_x (double, default: 0.4)
```

- 机器人的最大平移速度，单位为 meters/sec

```
~<name>/max_vel_x_backwards (double, default: 0.2)
```

- 当向后移动时，机器人的最大绝对平移速度，单位为 meters/sec 。
- See optimization parameter weight_kinematics_forward_drive.

```
~<name>/max_vel_theta (double, default: 0.3)
```

- 机器人的最大角速度，单位为 radians/sec 。

以下参数适用于四驱驱动机器人：

```
~<name>/min_turning_radius (double, default: 0.0)
```

- 四驱驱动机器人的最小转向半径（差速驱动机器人设置为零）。

```
~<name>/wheelbase (double, default: 1.0)
```

- 后驱动轴与前驱动轴之间的距离。

- The value might be negative for back-wheeled robots（仅当～ / cmd_angle_instead_rotvel 设置为true 时才需要）。

```
~<name>/cmd_angle_instead_rotvel (bool, default: false)
```

- Substitute the rotational velocity in the commanded velocity message by the corresponding steering angle [-pi/2,pi/2]. Note, changing the semantics of yaw rate depending on the application is not preferable. Here, it just complies with the inputs required by the stage simulator. Datatypes in ackermann_msgs are more appropriate, but are not supported by move_base. The local planner is not intended to send commands by itself.

以下参数适用于全向轮机器人:

- Note, reduce ~/weight_kinematics_nh significantly in order to adjust the tradeoff between usual driving and strafing.

```
~<name>/max_vel_y (double, default: 0.0)
```

- Maximum strafing velocity of the robot (should be zero for non-holonomic robots!)

```
~<name>/acc_lim_y (double, default: 0.5)
```

- Maximum strafing acceleration of the robot

以下是与用于优化的足迹模型相关的参数：

- (see Tutorial Obstacle Avoidance and Robot Footprint Model). New in version 0.3

```
~<name>/footprint_model/type (string, default: "point")
```

- 指定用于优化的机器人足迹模型的类型。
- 有如“点”，“圆形”，“线”，“two_circles”和“多边形”的不同类型。
- 但其模型的类型对所需的计算时间有显著的影响。

```
~<name>/footprint_model/radius (double, default: 0.2)
```

- 该参数是仅用于“圆形“类型的模型，其包含了圆的半径。
- 圆的中心位于机器人的旋转轴（axis of rotation）上。

```
~<name>/footprint_model/line_start (double[2], default: [-0.3, 0.0])
```

- 该参数是仅用于“线“类型的模型，其包含了里线段的起始坐标。

```
~<name>/footprint_model/front_offset (double, default: 0.2)
```

- 该参数是仅用于“two_circles“类型的模型，其描述了前圆（front circle）的中心沿机器人的 x 轴移动了多少。假设机器人的旋转轴（axis of rotation）位于[0,0]处。

```
~<name>/footprint_model/front_radius (double, default: 0.2)
```

- 该参数是仅用于“two_circles“类型的模型，其包含了前圆（front circle）的半径。

```
~<name>/footprint_model/rear_offset (double, default: 0.2)
```

- 该参数是仅用于“two_circles“类型的模型，其描述了后圆（rear circle）的中心沿机器人的负 x 轴移动多少。假设机器人的旋转轴（axis of rotation）位于[0,0]处。

```
~<name>/footprint_model/rear_radius (double, default: 0.2)
```

- 该参数是仅用于“two_circles“类型的模型，其包含了后圆（rear circle）的半径。

```
~<name>/footprint_model/vertices (double[], default: [ [0.25,-0.05], [...], ...])
```

- 该参数是仅用于“多边形“类型的模型，其包含了多边形顶点列表（每个 2d 坐标）。
- 多边形总是闭合的：不要在最后重复第一个顶点。

(2.2) Goal Tolerance Parameters

```
~<name>/xy_goal_tolerance (double, default: 0.2)
```

- 允许的机器人到目标位置的最终欧氏距离（euclidean distance），单位为 meters 。

```
~<name>/yaw_goal_tolerance (double, default: 0.2)
```

- Allowed final orientation error in radians

```
~<name>/free_goal_vel (bool, default: false)
```

- 去除目标速度的约束，让机器人可以以最大速度到达目标。
- Remove the goal velocity constraint such that the robot can arrive at the goal with maximum speed.

(2.3) Trajectory Configuration Parameters

```
~<name>/dt_ref (double, default: 0.3)
```

- Desired temporal resolution of the trajectory (the trajectory is not fixed to dt_ref since the temporal resolution is part of the optimization, but the trajectory will be resized between iterations if dt_ref +-dt_hysteresis is violated.

```
~<name>/dt_hysteresis (double, default: 0.1)
```

- Hysteresis for automatic resizing depending on the current temporal resolution, usually approx. 10% of dt_ref is recommended .

```
~<name>/min_samples (int, default: 3)
```

- 最小样本数（始终大于 2）

```
~<name>/global_plan_overwrite_orientation (bool, default: true)
```

- 覆盖由全局规划器提供的局部子目标的方向（因为它们通常仅提供 2D 路径）

```
~<name>/global_plan_viapoint_sep (double, default: -0.1 (disabled))
```

- 如果为正值，则通过点（via-points ）从全局计划（路径跟踪模式）展开。
- 该值确定参考路径的分辨率(沿着全局计划的每两个连续通过点之间的最小间隔,if negative: disabled)。
- 可以参考参数 weight_viapoint 来调整大小。New in version 0.4.
- If positive, via-points are extrected from the global plan (path-following mode). The value determines the resolution of the reference path (min. separation between each two consecutive via-points along the global plan, if negative: disabled). Refer to parameter weight_viapoint for adjusting the intensity. New in version 0.4

```
~<name>/max_global_plan_lookahead_dist (double, default: 3.0)
```

- Specify the maximum length (cumulative Euclidean distances) of the subset of the global plan taken into account for optimization. The actual length is than determined by the logical conjunction of the local costmap size and this maximum bound. Set to zero or negative in order to deactivate this limitation.

```
~<name>/force_reinit_new_goal_dist (double, default: 1.0)
```

- Reinitialize the trajectory if a previous goal is updated with a separation of more than the specified value in meters (skip hot-starting)

```
~<name>/feasibility_check_no_poses (int, default: 4)
```

- Specify up to which pose on the predicted plan the feasibility should be checked each sampling interval.

```
~<name>/publish_feedback (bool, default: false)
```

- 发布包含完整轨迹和动态障碍的列表的规划器反馈（应仅用于测试或调试启用）。 查看上面发布者的列表。

```
~<name>/shrink_horizon_backup (bool, default: true)
```

- 允许规划器在自动检测到问题(e.g. infeasibility)的情况下临时缩小 horizon（50％）。参考参数 shrink_horizon_min_duration。
- Allows the planner to shrink the horizon temporary (50%) in case of automatically detected issues (e.g. infeasibility). Also see parameter shrink_horizon_min_duration.

```
~<name>/allow_init_with_backwards_motion (bool, default: false)
```

- If true, underlying trajectories might be initialized with backwards motions in case the goal is behind the start within the local costmap (this is only recommended if the robot is equipped with rear sensors).

```
~<name>/exact_arc_length (bool, default: false
```

- 如果为真，则规划器在速度，加速度和角速度计算中使用确切的弧长度（ - >增加 cpu 运算时间），否则使用 Euclidean approximation。

```
~<name>/shrink_horizon_min_duration (double, default: 10.0)
```

- Specify minimum duration for the reduced horizon in case an infeasible trajectory is detected (refer to parameter shrink_horizon_backup in order to activate the reduced horizon mode).

(2.4) Obstacle Parameters

```
~<name>/min_obstacle_dist (double, default: 0.5)
```

- 与障碍的最小期望距离，单位 meters 。

```
~<name>/include_costmap_obstacles (bool, default: true)
```

- 指定应否考虑到局部 costmap 的障碍，被标记为障碍物的每个单元格被认为是点障碍物（point-obstacle）。
- 不要选择非常小的 costmap 分辨率，因为它增加了计算时间。
- 在未来的版本中，这种情况将会得到解决，并为动态障碍提供额外的 API。

```
~<name>/costmap_obstacles_behind_robot_dist (double, default: 1.0)
```

- Limit the occupied local costmap obstacles taken into account for planning behind the robot (specify distance in meters).

```
~<name>/obstacle_poses_affected (int, default: 30)
```

- Each obstacle position is attached to the closest pose on the trajectory in order to keep a distance. Additional neighbors can be taken into account as well.
- 注意：在将来的版本中可能会删除该参数，因为障碍物关联算法已经在 *kinetic+* 中被修改。
- 可以参考参数 legacy_obstacle_association 的描述。

```
~<name>/inflation_dist (double, default: pre kinetic: 0.0, kinetic+: 0.6)
```

- Buffer zone around obstacles with non-zero penalty costs (should be larger than min_obstacle_dist in order to take effect).
- 可以参考参数 weight weight_inflation 。

```
<name>/legacy_obstacle_association (bool, default: false)
```

- 连接轨迹位姿与优化障碍的策略已被修改（参考 changelog）。
- 参数设置为 true，既是切换到旧的的策略。 旧战略：对于每个障碍，找到最近的 TEB 位姿; 新策略：对于每个 teb 位姿，只找到"relevant" obstacles。

```
~<name>/obstacle_association_force_inclusion_factor (double, default: 1.5)
```

- 在优化过程中，非传统障碍物关联策略试图仅将相关障碍与离散轨迹相连接。 但是，特定距离内的所有障碍都被迫被包含（作为 min_obstacle_dist 的倍数）。
- 例如。 选择 2.0 以便在 2.0 * min_obstacle_dist 的半径范围内强制考虑障碍。 [只有在参数 legacy_obstacle_association 为 false 时才使用此参数]

```
~<name>/obstacle_association_cutoff_factor (double, default: 5)
```

- 请参考参数 barriers_association_force_inclusion_factor，但超出[value] * min_obstacle_dist 的倍数时，优化过程中将忽略所有障碍。
- 首先处理参数 barrier_association_force_inclusion_factor。 [只有参数 legacy_obstacle_association 为 false 时才使用此参数]

以下参数适用于需要 costmap_converter 插件（参见教程）的情况下：

```
~<name>/costmap_converter_plugin (string, default: "")
```

- 定义插件名称，用于将 costmap 的单元格转换成点/线/多边形。
- 若设置为空字符，则视为禁用转换，将所有点视为点障碍。

```
~<name>/costmap_converter_spin_thread (bool, default: true)
```

- 如果为 true，则 costmap 转换器将以不同的线程调用其回调队列。

```
~<name>/costmap_converter_rate (double, default: 5.0)
```

- 定义 costmap_converter 插件处理当前 costmap 的频率（该值不高于 costmap 更新率）,单位为 Hz 。

(2.5) Optimization Parameters

```
~<name>/no_inner_iterations (int, default: 5)
```

- 在每个外循环迭代中调用的实际求解器迭代次数。
- 参考参数 no_outer_iterations。
- Number of actual solver iterations called in each outerloop iteration. See param no_outer_iterations.

~/no_outer_iterations (int, default: 4)

- 每个外循环自动根据期望的时间分辨率 dt_ref 调整轨迹大小，并调用内部优化器（执行参数 no_inner_iterations）。 因此，每个规划周期中求解器迭代的总数是两个值的乘积。
- Each outerloop iteration automatically resizes the trajectory according to the desired temporal resolution dt_ref and invokes the internal optimizer (that performs no_inner_iterations). The total number of solver iterations in each planning cycle is therefore the product of both values.

```
~<name>/penalty_epsilon (double, default: 0.1)
```

- 为硬约束近似的惩罚函数添加一个小的安全范围。
- Add a small safety margin to penalty functions for hard-constraint approximations .

```
~<name>/weight_max_vel_x (double, default: 2.0)
```

- 满足最大允许平移速度的优化权重。

```
~<name>/weight_max_vel_theta (double, default: 1.0)
```

- 满足最大允许角速度的优化权重。

```
~<name>/weight_acc_lim_x (double, default: 1.0)
```

- 满足最大允许平移加速度的优化权重。

```
~<name>/weight_acc_lim_theta (double, default: 1.0)
```

- 满足最大允许角加速度的优化权重。

```
~<name>/weight_kinematics_nh (double, default: 1000.0)
```

- 用于满足 non-holonomic 运动学的优化权重（由于运动学方程构成等式约束，该参数必须很高）
- Optimization weight for satisfying the non-holonomic kinematics (this parameter must be high since the kinematics equation constitutes an equality constraint, even a value of 1000 does not imply a bad matrix condition due to small 'raw' cost values in comparison to other costs).

```
~<name>/weight_kinematics_forward_drive (double, default: 1.0)
```

- 强制机器人仅选择正向（正的平移速度）的优化权重。
- 重量小的机器人仍然可以向后移动。

```
~<name>/weight_kinematics_turning_radius (double, default: 1.0)
```

- 采用最小转向半径的优化权重（仅适用于四驱驱动机器人）

```
~<name>/weight_optimaltime (double, default: 1.0)
```

- Optimization weight for contracting the trajectory with regard to transition/execution time .

```
~<name>/weight_obstacle (double, default: 50.0)
```

- 保持与障碍物的最小距离的优化权重。

```
~<name>/weight_viapoint (double, default: 1.0)
```

- 最小化到通过点的距离(resp. reference path)的优化权重。New in version 0.4.

```
~<name>/weight_inflation (double, default: 0.1)
```

- 膨胀区惩罚的优化权重（值应该很小）。

```
~<name>/weight_adapt_factor (double, default: 2.0)
```

- Some special weights (currently weight_obstacle) are repeatedly scaled by this factor in each outer TEB iteration (weight_new = weight_old*factor). Increasing weights iteratively instead of setting a huge value a-priori leads to better numerical conditions of the underlying optimization problem.

(2.6) Parallel Planning in distinctive Topologies

```
~<name>/enable_homotopy_class_planning (bool, default: true)
```

- 在不同的拓扑里激活并行规划（因为一次优化多个轨迹，所以需要占用更多的 CPU 资源）
- Activate parallel planning in distinctive topologies (requires much more CPU resources, since multiple trajectories are optimized at once)

```
~<name>/enable_multithreading (bool, default: true)
```

- 激活多个线程，以便在不同的线程中规划每个轨迹。

```
~<name>/max_number_classes (int, default: 4)
```

- 指定考虑到的不同轨迹的最大数量（限制计算量）。

```
~<name>/selection_cost_hysteresis (double, default: 1.0)
```

- Specify how much trajectory cost must a new candidate have w.r.t. a previously selected trajectory in order to be selected (selection if new_cost < old_cost*factor).

```
~<name>/selection_obst_cost_scale (double, default: 100.0)
```

- Extra scaling of obstacle cost terms just for selecting the 'best' candidate.

```
~<name>/selection_viapoint_cost_scale (double, default: 1.0)
```

- Extra scaling of via-point cost terms just for selecting the 'best' candidate. New in version 0.4.

```
~<name>/selection_alternative_time_cost (bool, default: false)
```

- If true, time cost (sum of squared time differences) is replaced by the total transition time (sum of time differences).

```
~<name>/roadmap_graph_no_samples (int, default: 15)
```

- 指定为创建路线图而生成的样本数。

```
~<name>/roadmap_graph_area_width (double, default: 6)
```

- 指定该区域的宽度。
- 在开始和目标之间的矩形区域中采样随机关键点/航点(keypoints/waypoints)。

```
~<name>/h_signature_prescaler (double, default: 1.0)
```

- 缩放用于区分同伦类的内部参数（H-signature）。
- 警告：只能减少此参数，如果在局部 costmap 中遇到太多障碍物的情况，请勿选择极低值，否则无法将障碍物彼此区分开（0.2 < value <= 1）。

```
~<name>/h_signature_threshold (double, default: 0.1)
```

- Two H-signatures are assumed to be equal, if both the difference of real parts and complex parts are below the specified threshold.

```
~<name>/obstacle_heading_threshold (double, default: 1.0)
```

- Specify the value of the scalar product between obstacle heading and goal heading in order to take them (obstacles) into account for exploration.

```
~<name>/visualize_hc_graph (bool, default: false)
```

- 可视化创建的图形，用于探索不同的轨迹（在 rviz 中检查标记消息）

```
~<name>/viapoints_all_candidates (bool, default: true)
```

- If true, all trajectories of different topologies are attached to the set of via-points, otherwise only the trajectory sharing the same topology as the initial/global plan is connected with them (no effect on test_optim_node). New in version 0.4

(2.7) Miscellaneous Parameters

```
~<name>/odom_topic (string, default: "odom")
```

- odometry 消息的主题名称，由机器人驱动程序或仿真器提供。

```
~<name>/map_frame (string, default: "odom")
```

- 全局规划坐标系（在静态地图的情况下，此参数通常必须更改为“/ map”。）。

Roadmap

- 目前为将来所计划的一些功能和改进。 欢迎大家来贡献出自己的一分力！
  - 在不可避免的障碍（例如障碍物十分靠近目标位置）的情况下，增加和改进安全性能。
  - 实施适当的逃离行为。
  - 改进或解决规划器在多个局部最佳解决方案之间的选择问题(不是在拓扑学基础上，而是由于出现干扰等)。

参考资料

- http://wiki.ros.org/teb_local_planner

ROS 与 navigation 教程-dwa_local_planner（DWA）介绍

说明：

- 介绍 dwa_local_planner（DWA）的概念和相关知识。

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- 该包使用 DWA（Dynamic Window Approach）算法实现了平面上移动机器人局部导航功能。
- 给定一个全局路径规划和代价地图，局部路径规划器生成的速度命令发送到移动基座。
- 该包支持任何 footprint 表示为凸多边形或圆形的机器人，同时将其配置公开为可在启动文件中设置的 ROS 参数。 该规划器的参数也可重新动态配置。
- 该包的 ROS 封装继承了 BaseLocalPlanner 接口。
- dwa_local_planner 包提供了一个驱动平面中移动基座的控制器，其将路径规划器和机器人连接到一起。
- 移动过程中，路径规划器会在机器人周围创建可以表示为栅格地图的评价函数。其中控制器的主要任务就是利用评价函数确定发送给基座的速度（dx,dy,dtheta）。

- DWA（Dynamic Window Approach）算法的基本思路如下：
  - 在机器人控制空间（dx，dy，dta）中离散采样。
  - 对于每个采样速度，从机器人的当前状态执行正向移动模拟，以预测如果在一些（短）时间段内应用采样速度将会发生什么。
  - 从以下方面来评价正向移动模拟产生的每个轨迹：接近障碍物，接近目标，接近全局路径和速度。 放弃非法轨迹（与障碍物相撞的轨迹）。
  - 选择最高得分的轨迹，并将相关速度发送到移动基座。
  - 冲洗并重复。
- 参考链接：
  - D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". The Dynamic Window Approach to local control.
  - Alonzo Kelly. "An Intelligent Predictive Controller for Autonomous Vehicles".An previous system that takes a similar approach to control.
  - Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain ".Discussion of the Trajectory Rollout algorithm in use on the LAGR robot.

DWAPlannerROS

- dwa_local_planner::DWAPlannerROS 对象是 dwa_local_planner::DWAPlanner 对象的 ROS 封装，在初始化时指定的 ROS 命名空间使用，继承了 nav_core::BaseLocalPlanner 接口。

创建 dwa_local_planner::DWAPlannerROS 对象的例子：

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <dwa_local_planner/dwa_planner_ros.h>

...

tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS costmap("my_costmap", tf);

dwa_local_planner::DWAPlannerROS dp;
dp.initialize("my_dwa_planner", &tf, &costmap);
```

(1) API Stability

- C++ API 是稳定的。
- ROS API 是稳定的。

(1.1) Published Topics

```
~<name>/global_plan ([nav_msgs/Path][6])
```

- 局部路径规划器当前正在跟踪的全局路径规划的一部分。主要用于可视化。

```
~<name>/local_plan ([nav_msgs/Path][7])
```

- 上一个周期中得分最高的局部路径规划或轨迹。 主要用于可视化。

(1.2) Subscribed Topics

```
odom ([nav_msgs/Odometry][8])
```

- 提供机器人当前速度的里程计信息到局部路径规划器。假定该消息中的速度信息使用与 TrajectoryPlannerROS 对象中包含的代价地图的 robot_base_frame 相同的坐标系。 有关 robot_base_frame 参数的信息，请参阅 costmap_2d 包。

(2) Parameters

- 可以通过设置 ROS 参数来定制 dwa_local_planner :: DWAPlannerROS 的行为。
- 这些参数分为几类：机器人配置，目标公差，前向仿真，轨迹评分，防振和全局计划。

- 这些参数中的大多数也可以使用 dynamic_reconfigure 进行更改，以便于在正在运行的系统中调整本地路径规划器。。

(2.1) Robot Configuration Parameters

```
~<name>/acc_lim_x (double, default: 2.5)
```

- 机器人在 x 方向的加速度极限，单位为 meters/sec^2 。

```
~<name>/acc_lim_y (double, default: 2.5)
```

- 机器人在 y 方向的速度极限，单位为 meters/sec^2 。

```
~<name>/acc_lim_th (double, default: 3.2)
```

- 机器人的角加速度极限，单位为 radians/sec^2 。

```
~<name>/max_trans_vel (double, default: 0.55)
```

- 机器人最大平移速度的绝对值，单位为 m/s 。

```
~<name>/min_trans_vel (double, default: 0.1)
```

- 机器人最小平移速度的绝对值，单位为 m/s 。

```
~<name>/max_vel_x (double, default: 0.55)
```

- 机器人在 x 方向到最大速度，单位为 m/s 。

```
~<name>/min_vel_x (double, default: 0.0)
```

- 机器人在 x 方向到最小速度，单位为 m/s 。
- 若设置为负数，则机器人向后移动。

```
~<name>/max_vel_y (double, default: 0.1)
```

- 机器人在 y 方向到最大速度，单位为 m/s 。

```
~<name>/min_vel_y (double, default: -0.1)
```

- 机器人在 y 方向到最小速度，单位为 m/s 。

```
~<name>/max_rot_vel (double, default: 1.0)
```

- 机器人的最大角速度的绝对值，单位为 rad/s 。

```
~<name>/min_rot_vel (double, default: 0.4)
```

- 机器人的最小角速度的绝对值，单位为 rad/s 。

(2.2) Goal Tolerance Parameters

```
~<name>/yaw_goal_tolerance (double, default: 0.05)
```

- The tolerance in radians for the controller in yaw/rotation when achieving its goal .

```
~<name>/xy_goal_tolerance (double, default: 0.10)
```

- The tolerance in meters for the controller in the x & y distance when achieving a goal .

```
~<name>/latch_xy_goal_tolerance (bool, default: false)
```

- 如果锁定目标公差且机器人到达目标 xy 位置，机器人将简单地旋转到位，即使它在目标公差的范围内结束。
- If goal tolerance is latched, if the robot ever reaches the goal xy location it will simply rotate in place, even if it ends up outside the goal tolerance while it is doing so.

(2.2) Forward Simulation Parameters

```
~<name>/sim_time (double, default: 1.7)
```

- 前向模拟轨迹的时间，单位为 seconds 。

```
~<name>/sim_granularity (double, default: 0.025)
```

- 给定轨迹上的点之间的间隔尺寸，单位为 meters 。

```
~<name>/vx_samples (integer, default: 3)
```

- x 方向速度的样本数。
- The number of samples to use when exploring the x velocity space .

```
~<name>/vy_samples (integer, default: 10)
```

- y 方向速度的样本数。
- The number of samples to use when exploring the y velocity space .

```
~<name>/vth_samples (integer, default: 20)
```

- 角速度的样本数。
- The number of samples to use when exploring the theta velocity space .

```
~<name>/controller_frequency (double, default: 20.0)
```

- 调用该控制器的频率。
- 当用 move_base 时，可以只设置 controller_frequency 参数，可以放心的忽略它。
- Uses searchParam to read the parameter from parent namespaces if not set in the namespace of the controller. For use with move_base, this means that you only need to set its "controller_frequency" parameter and can safely leave this one unset.

(2.4) Trajectory Scoring Parameters

```
     cost =
  pdist_scale * (distance to path from the endpoint of the trajectory in map cells or mete
rs depending on the meter_scoring parameter)
  + gdist_scale * (distance to local goal from the endpoint of the trajectory in map cells
 or meters depending on the meter_scoring parameter)
  + occdist_scale * (maximum obstacle cost along the trajectory in obstacle cost (0-254))
```

- 该 cost 函数使用如上公式给每条轨迹评分

```
~<name>/path_distance_bias (double, default: 32.0)
```

- 控制器靠近给定路径的权重。
- The weighting for how much the controller should stay close to the path it was given .

```
~<name>/goal_distance_bias (double, default: 24.0)
```

- The weighting for how much the controller should attempt to reach its local goal, also controls speed .

```
~<name>/occdist_scale (double, default: 0.01)
```

- 控制器尝试避免障碍物的权重。
- The weighting for how much the controller should attempt to avoid obstacles .

```
~<name>/forward_point_distance (double, default: 0.325)
```

- The distance from the center point of the robot to place an additional scoring point, in meters .

```
~<name>/stop_time_buffer (double, default: 0.2)
```

- The amount of time that the robot must stop before a collision in order for a trajectory to be considered valid in seconds .

```
~<name>/scaling_speed (double, default: 0.25)
```

- The absolute value of the velocity at which to start scaling the robot's footprint, in m/s.

```
~<name>/max_scaling_factor (double, default: 0.2)
```

- The maximum factor to scale the robot's footprint by .

(2.5) Oscillation Prevention Parameters

```
~<name>/oscillation_reset_dist (double, default: 0.05)
```

- 机器人必须运动多少米远后才能复位震荡标记。

(2.6) Global Plan Parameters

```
~<name>/prune_plan (bool, default: true)
```

- 定义当机器人是否边沿着路径移动时，边抹去已经走过的路径规划。 设置为 true 时，表示当机器人移动 1 米后，将 1 米之前的全局路径点一个一个清除。（包括全局的 global path 和局部的 global path）

(3) C++ API

- 有关 base_local_planner::TrajectoryPlannerROS 类的 C ++ API 文档，请参阅：DWAPlannerROS C++ API

DWAPlanner

- dwa_local_planner::DWAPlanner 实现了 DWA 和 Trajectory Rollout 的实现。要在 ROS 中使用 dwa_local_planner::DWAPlanner，请使用进行了 ROS 封装的格式，不推荐直接使用它。

(1) API Stability

- C ++ API 是稳定的。
- 但是，建议您使用 DWAPlannerROS，而不是使用 dwa_local_planner::TrajectoryPlanner。

(2) C++ API

- 有关 dwa_local_planner :: DWAPlanner 类中的 C ++ API 文档，请参阅：DWAPlanner C++ API。

参考资料

- wiki.ros.org/dwa_local_planner
- blog.csdn.net/x_r_su/article/details/53393872
- blog.csdn.net/dxuehui/article/details/39376009

ROS 与 navigation 教程-nav_core 介绍

说明:

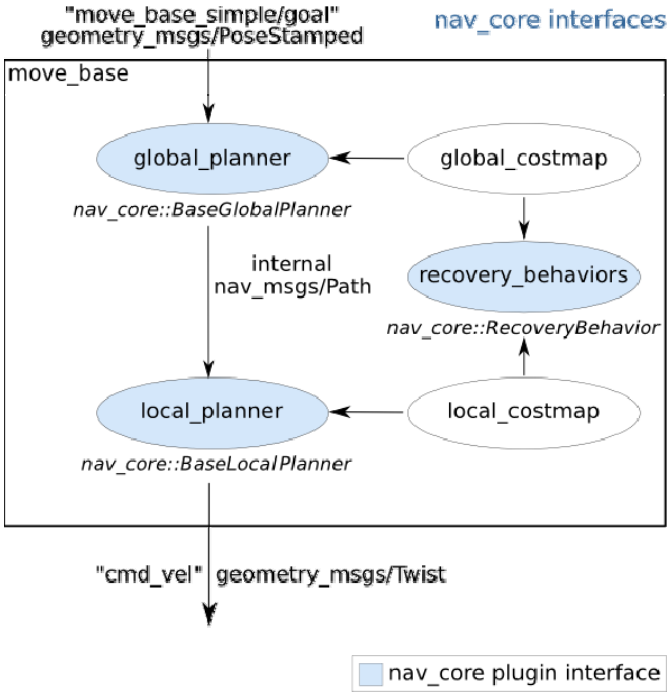
- 介绍了 nave_core 的概念和其相关知识

代码库

- 参考链接: https://github.com/ros-planning/navigation.git

概要

- nav_core 包包含了导航功能包的关键接口。
- 所有规划器和修复行为机制可以以插件方式在 move_base node 中使用，且必须继承这些接口。



BaseGlobalPlanner

- nav_core::BaseGlobalPlanner 支持了供导航中全局路径规划器使用的接口。
- move_base node 中使用的所有全局路径规划器插件都必须继承这个接口。
- 目前使用了 nav_core::BaseGlobalPlanner 接口的全局规划器有：
  - o global_planner：一个快速内插值全局规划器，是对 navfn 的更灵活的替代(pluginlib name: "global_planner/GlobalPlanner")
  - o navfn 一个使用 navigation 函数计算机器人路径且基于栅格的全局规划器(pluginlib name: "navfn/NavfnROS")
  - o carrot_planner：一个简单的全局路径规划器，作用是尽可能地移动机器人到用户指定的目标位置，即使目标在障碍中也会靠近目标。 (pluginlib name: "carrot_planner/CarrotPlanner")

(1) API Stability

- C++ API 是稳定的。

(2) BaseGlobalPlanner C++ API

- 有关 nav_core :: BaseGlobalPlanner 的 C ++ API 的文档可以在这里找到：BaseGlobalPlanner documentation 。

BaseLocalPlanner

- nav_core::BaseLocalPlanner 给导航中使用局部路径规划器提供接口。
- move_base 中所有局部规划器插件必须继承该接口。
- 目前 nav_core::BaseLocalPlanner 接口包括：
  - base_local_planner： 提供用 Dynamic Window and Trajectory Rollout approaches 来做局部规划控制
  - eband_local_planner： Implements the Elastic Band method on the SE2 manifold .
  - teb_local_planner：实现用于在线轨迹优化的 Timed-Elastic-Band 方法。

(1) API Stability

- C++ API 是稳定的。

(2) BaseLocalPlanner C++ API

- 有关 nav_core::BaseGlobalPlanner 的 C ++ API 的文档可以在这里找到： BaseLocalPlanner Documentation.

RecoveryBehavior

- nav_core::RecoveryBehavior 提供了导航中修复机制接口。
- move_base 中所有修复机制插件必须继承该接口。
- 目前使用了 nav_core::RecoveryBehavior 接口的主要有：
  - clear_costmap_recovery：将用户定义的某个范围外的代价地图回退到静态地图。
  - rotate_recovery： 执行 360 度旋转来清理出空间。

(1) API Stability

- C++ API 是稳定的。

(2) RecoveryBehavior C++ API

- 有关 nav_core::RecoveryBehavior 的 C ++ API 的文档可以在这里找到： RecoveryBehavior Documentation

参考资料

- http://wiki.ros.org/nav_core
- http://blog.csdn.net/x_r_su/article/details/53379676

ROS 与 navigation 教程-navfn 介绍

代码库

- 参考链接：https://github.com/ros-planning/navigation.git

概要

- navfn 提供了一个快速内插值的导航功能，用于为移动基座创建路径规划。
- 规划器假设机器人为圆形并利用代价地图来进行操作，以从栅格的起点到终点找到代价最小的路径规划。
- 导航功能使用 Dijkstra 算法计算，未来亦会加入 A *算法。
- navfn 提供了 ROS 封装接口供调用，也继承了 nav_core::BaseGlobalPlanner 接口。
- navfn::NavfnROS 对象在 move_base 中被作为全局路径规划器插件使用。

NavfnROS

- navfn::NavfnROS 是 navfn::NavFn 的 ROS 封装，导出了可用的 ROS 接口。
- 其可以在初始化时指定的 ROS 命名空间使用，继承了 nav_core::BaseGlobalPlanner 接口。
- 以下是创建 navfn::NavfnROS 对象的一个例子:

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <navfn/navfn_ros.h>

...

tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS costmap("my_costmap", tf);

navfn::NavfnROS navfn;
navfn.initialize("my_navfn_planner", &costmap);
```

(1) API Stability

- ROS API 是稳定的。
- C++ API 是稳定的。

(2) ROS API

(2.1) Published Topics

```
~<name>/plan ([nav_msgs/Path][2])
```

- 通过 navfn 计算的上一个最新规划，规划器每次都会计算一条新路径并发布。
- 主要用于可视化。

(2.2) Parameters

```
~<name>/allow_unknown (bool, default: true)
```

- 指定是否允许 navfn 在 unknown 空间创建路径规划。
- 注意：如果你使用带有体素或障碍层的分层 costmap_2d 代价地图，那么需将该图层的 track_unknown_space 参数设置为 true，否则所有未知空间将转换为自由空间(which navfn will then happily go right through)。

```
~<name>/planner_window_x (double, default: 0.0)
```

- 指定可选窗口的 x 大小以限定规划器工作空间。
- 其有利于限定 NavFn 在大型代价地图的小窗口下工作。

```
~<name>/planner_window_y (double, default: 0.0)
```

- 指定可选窗口的 y 大小以限定规划器工作空间。
- 其有利于限定 NavFn 在大型代价地图的小窗口下工作。

```
~<name>/default_tolerance (double, default: 0.0)
```

- 定义路径规划器目标点的公差范围。
- NavFn 将试图创建尽可能接近指定目标的路径规划，但不会超过 default_tolerance 。New in navigation 1.3.0

```
~<name>/visualize_potential (bool, default: false)
```

- 指定是否通过 PointCloud2 来可视化由 navfn 计算的潜在区域。 New in navigation 1.3.1.

(2.3) C++ API

- navfn::NavfnROS 继承了 nav_core::BaseGlobalPlanner 接口。
- 详细资料参阅 NavfnROS C++ Documentation

NavFN

- navfn::NavFn 额提供了上述导航功能，且可以免费使用，但是请记住我们不保证其 API 是稳定的。

(1) API Stability

- C++ API 是不稳定的。
- It is subject to change at any time.

(2) C++ API

- 关于 navfn :: NavFn 类的 C ++ API 的文档可参阅：NavFn Documentation 。
- 再次强调，C ++ API 不能保证稳定。

Reference

- Rationale of Dijkstra being used in navfn instead of A*

参考资料

- http://wiki.ros.org/navfn
- http://blog.csdn.net/x_r_su/article/details/53407105

ROS 与 navigation 教程-robot_pose_ekf 介绍

说明：

- 介绍了 robot_pose_ekf 的概念及相关知识

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- Robot Pose EKF 包用于评估机器人的 3D 位姿，基于来自不同来源的位姿测量信息。
- 它使用带有 6D（3D position and 3D orientation）模型信息的扩展卡尔曼滤波器来整合来自轮式里程计，IMU 传感器和视觉里程计的数据信息。
- 基本思路就是用松耦合方式融合不同传感器信息实现位姿估计。

How to use the Robot Pose EKF

(1) Configuration

- EKF 节点的默认启动文件位于 robot_pose_ekf 包目录中。 启动文件包含多个可配置参数：
  - freq: 滤波器更新和发布频率。注意：频率高仅仅意味着一段时间可以获得更多机器人位姿信息，但是并不表示可以提高每次位姿评估的精度。
  - sensor_timeout: 当传感器停止向滤波器发送信息时，滤波器在没有传感器的情况下等待多长时间才重新开始工作。
  - odom_used, imu_used, vo_used: 确认是否输入。
- 在启动文件中修改配置，例如：

```
    <launch>
  <node pkg="robot_pose_ekf" type="robot_pose_ekf" name="robot_pose_ekf">
    <param name="output_frame" value="odom"/>
    <param name="freq" value="30.0"/>
    <param name="sensor_timeout" value="1.0"/>
    <param name="odom_used" value="true"/>
    <param name="imu_used" value="true"/>
    <param name="vo_used" value="true"/>
    <param name="debug" value="false"/>
    <param name="self_diagnose" value="false"/>
  </node>
</launch>
```

(2) Running

- 编译

```
$ rosdep install robot_pose_ekf

$ roscd robot_pose_ekf

$ rosmake
```

- 运行

```
$ roslaunch robot_pose_ekf.launch
```

Nodes

(1) robot_pose_ekf

- robot_pose_ekf 包实现扩展卡尔曼滤波器，用于确定机器人位姿。

(1.1) Subscribed Topics

```
odom ([nav_msgs/Odometry][2])
```

- 2D pose (用于轮式里程计)：其包含机器人在地面中的位置（position）和方位（orientation）以及该位姿的协方差。 发送此 2D 位姿的消息实际上表示 3D 位姿，但 z，pitch 和 roll 分量被简单忽略了。

```
imu_data ([sensor_msgs/Imu][3])
```

- 3D orientation (用于 IMU):3D 方位提供机器人基座相对于地图坐标系的 Roll, Pitch and Yaw 偏角。Roll and Pitch 角是绝对角度（因为 IMU 使用了重力参考），而 YAW 角是相对角度。协方差矩阵指定的方位测量的不确定度。当仅仅收到这个主题消息时，机器人位姿 ekf 还不会启动，因为它还需要来自主题'vo'或者'odom'的消息。

```
vo ([nav_msgs/Odometry][4])
```

- 3D pose (用于视觉里程计):3D 位置表示机器人的完整位置和方位以及该位姿的协方差。当用传感器只测量部分 3D 位姿(e.g. the wheel odometry only measures a 2D pose)时候，可以给还未真正开始测量的部分 3D 位姿先简单指定一个大的协方差。

robot_pose_ekf 节点不要求所有三个传感器源一直可用。每个源给出位态估计和协方差。源以不同的速率和延迟进行操作。 源可以随着时间的推移出现并消失，节点将自动检测并使用可用的传感器。假如要添加您自己的传感器到输入源中，请查看 the Adding a GPS sensor tutorial。

(1.2) Published Topics

```
robot_pose_ekf/odom_combined ([geometry_msgs/PoseWithCovarianceStamped][6])
```

- 滤波器输出 (评估的 3D 机器人位姿)。

(1.3) Provided tf Transforms

- odom_combined → base_footprint

How Robot Pose EKF works

(1) Pose interpretation

- 向滤波器节点发送信息的所有传感器源都有自己的参考坐标系，并且随着时间推移都可能出现漂移现象。
- 因此，不同传感器发送的绝对位姿不能相互比较。
- 节点使用每个传感器的相对位姿差来更新扩展卡尔曼滤波器。

(2) Covariance interpretation

- 随着机器人的移动，其在参考坐标系中的位姿的不确定性越来越大。同时随着时间的推移，协方差将会无限增长。因此，在位姿本身上公布协方差是没有意义的，而传感器源则会随时间的变化来公布协方差，即速度的协方差。
- Note that using observations of the world (e.g. measuring the distance to a known wall) will reduce the uncertainty on the robot pose; this however is localization, not odometry.

(3) Timing

- 想象一下，机器人姿势过滤器最后在时间 t_0 更新。 在每个传感器的至少一次测量到达时间晚于 t_0 的时间戳之后，节点将不会更新机器人姿态滤波器。
- 当例如 收到有关时间戳 t_1> t_0 的 odom 主题的消息，并且在时间戳 t_2> t_1> t_0 的 imu_data 主题上，过滤器现在将更新为关于所有传感器的信息可用的最新时间，在这种情况下为时间 T_1。 直接给出 t_1 处的 odom 姿态，并且通过在 t_0 和 t_2 之间的 imu 姿态的线性插值来获得 t_1 上的 imu 姿态。在 t_0 和 t_1 之间，使用 odom 和 imu 的相对姿势来更新机器人姿势过滤器。

- 上图显示了 PR2 机器人从给定的初始位置（绿点）开始移动并返回到初始位置时的实验结果。 完美的 odometry x-y 曲线图显示精确的闭环曲线图。 蓝线显示来自轮式里程计的输入，蓝点是估计的最终位置。 红线显示的是 robot_pose_ekf 的输出，其整合了轮式里程计和 imu 的信息与红点的估计结束位置。

Package Status

(1) Stability

- 该包的基础部分代码已经被充分测试且稳定较长时间。
- 但是 ROS API 一直在随着消息类型的变化而被升级。
- 在未来的版本中，ROS API 可能会再次更改为简化的单主题界面（参见下面的路线图）。

(2) Roadmap

- 目前该滤波器被设计用于在 PR2 机器人上使用的三个传感器信号(wheel odometry, imu and vo) 。下一步计划是让该包更加通用可以监听更多传感器源，所有发布均使用消息 (nav_msgs/Odometry) 。每个源将在“里程计”消息中设置 3D 位姿的协方差，来指定其实际测量的 3D 位姿的哪一部分。
- 增加速度到扩展卡尔曼滤波器状态中。

Tutorials

- robot_pose_ekf/Tutorials/AddingGpsSensor

参考资料

- http://wiki.ros.org/robot_pose_ekf
- http://blog.csdn.net/x_r_su/article/details/53406078

ROS 与 navigation 教程-robot_localization 介绍

说明：

- 介绍 robot_localization 的概念

代码库

- 参考链接：https://github.com/cra-ros-pkg/robot_localization.git
- Use GitHub to report bugs or submit feature requests. View active issues

概要

- Provides nonlinear state estimation through sensor fusion of an arbitrary number of sensors.
- 详细资料请参阅 hosted on docs.ros.org 。

参考资料

- http://wiki.ros.org/robot_localization

ROS 与 navigation 教程-amcl 介绍

说明：

- 介绍 amcl 的概念和相关知识。

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- amcl 是一种机器人在 2D 中移动的概率定位系统。 它实现了自适应（或 KLD 采样）蒙特卡罗定位方法（如 Dieter Fox 所述），该方法使用粒子滤波器来针对已知地图跟踪机器人的位姿。

Algorithms

- 在《Probabilistic Robotics》这书中，提及了许多的参数和算法。推荐读者阅读该书来获得更多相关信息。
- 其中用到该书的算法有 sample_motion_model_odometry, beam_range_finder_model, likelihood_field_range_finder_model, Augmented_MCL, and KLD_Sampling_MCL。
- 基于目前现有的实现的话， 这个 node 仅能使用激光扫描和扫描地图来工作，但是可以进行扩展以使用其它传感器数据。

Example

- 在 base_scan topic 上使用激光数据定位：

```
amcl scan:=base_scan
```

Nodes

(1) amcl

- amcl 节点接收 laser-based 地图，激光扫描和 tf 变换信息，并且输出位姿评估。
- 启动时，amcl 根据提供的参数初始化其粒子滤波器。
- 注意：如果没有设置参数，初始滤波器状态将是以（0,0,0）为中心的适中大小的粒子云。

(1.1) Subscribed Topics

```
scan ([sensor_msgs/LaserScan][2])
```

- 激光扫描。

```
tf ([tf/tfMessage][3])
```

- tf 变换信息。

```
initialpose ([geometry_msgs/PoseWithCovarianceStamped][4])
```

- 用来初始化粒子滤波器的均值和协方差。

```
map ([nav_msgs/OccupancyGrid][5])
```

- 当设置 use_map_topic 参数时，AMCL 会订阅此话题来获取用于 laser-based 定位的地图。New in navigation 1.4.2.

(1.2) Published Topics

```
amcl_pose ([geometry_msgs/PoseWithCovarianceStamped][6])
```

- 机器人在地图上的带有协方差的预估位姿。

```
particlecloud ([geometry_msgs/PoseArray][7])
```

- 粒子滤波器维护的预估位姿集合。

```
tf ([tf/tfMessage][8])
```

- 发布里程（可以通过~odom_frame_id 参数重新映射）到地图的变换。

(1.3) Services

```
global_localization ([std_srvs/Empty][9])
```

- 初始化全局定位，其中所有粒子随机分布在地图中的自由空间中。

```
request_nomotion_update ([std_srvs/Empty][10])
```

- 用于手动执行更新和发布更新的粒子。

(1.4) Services Called

```
static_map ([nav_msgs/GetMap][11])
```

- amcl 调用此服务来获取 laser-based 定位的地图；从该服务获取地图的启动模块。
- amcl calls this service to retrieve the map that is used for laser-based localization; startup blocks on getting the map from this service.

(1.5) Parameters

ROS 参数可以使用三种类型来配置 amcl 节点：verall filter, laser model, 和 odometery model。

Overall filter 参数

```
~min_particles (int, default: 100)
```

- 允许的最少粒子数。

```
~max_particles (int, default: 5000)
```

- 允许的最多粒子数。

```
~kld_err (double, default: 0.01)
```

- 实际分布与估计分布之间的最大误差。

```
~kld_z (double, default: 0.99)
```

- Upper standard normal quantile for (1 - p), where p is the probability that the error on the estimated distrubition will be less than kld_err.

```
~update_min_d (double, default: 0.2 meters)
```

- 执行更新过滤器操作之前需要进行平移运动。

```
~update_min_a (double, default: π/6.0 radians)
```

- 执行更新滤波器操作之前需要进行旋转运动。

```
~resample_interval (int, default: 2)
```

- 重新采样前需要的滤波器到更新次数。

```
~transform_tolerance (double, default: 0.1 seconds)
```

- Time with which to post-date the transform that is published, to indicate that this transform is valid into the future.

```
~recovery_alpha_slow (double, default: 0.0 (disabled))
```

- slow average weight 滤波器的指数衰减率，用于决定何时通过添加随机位姿进行恢复。

- 建议设置为 0.001。
- Exponential decay rate for the slow average weight filter, used in deciding when to recover by adding random poses. A good value might be 0.001.

```
~recovery_alpha_fast (double, default: 0.0 (disabled))
```

- fast average weight 滤波器的指数衰减率，用于决定何时通过添加随机位姿进行恢复。
- 建议设置为 0.1。
- Exponential decay rate for the fast average weight filter, used in deciding when to recover by adding random poses. A good value might be 0.1.

```
~initial_pose_x (double, default: 0.0 meters)
```

- 初始位姿 mean（x），用于初始化高斯分布的滤波器。

```
~initial_pose_y (double, default: 0.0 meters)
```

- 初始位姿 mean（y），用于初始化高斯分布的滤波器。

```
~initial_pose_a (double, default: 0.0 radians)
```

- 初始位姿 mean（yaw），用于初始化高斯分布的滤波器。

~initial_cov_xx (double, default: 0.5*0.5 meters)

- 初始位姿 covariance (x*x) ，用于初始化高斯分布的滤波器。

```
~initial_cov_yy (double, default: 0.5*0.5 meters)
```

- 初始位姿 covariance (y*y) ，用于初始化高斯分布的滤波器。

```
~initial_cov_aa (double, default: (π/12)
```

- 初始位姿 covariance (yaw*yaw) ，用于初始化高斯分布的滤波器。

```
~gui_publish_rate (double, default: -1.0 Hz)
```

- 指定最大可用多大速率(Hz)扫描并发布用于可视化的路径。
- 若设置为-1.0，则表示为禁用。

```
~save_pose_rate (double, default: 0.5 Hz)
```

- 指定在 ~initial_pose_* and ~initial_cov_*变量存储的上次预估的位姿和协方差到参数服务器的最大速率 (Hz)。保存的位姿会在后面初始化滤波器时候使用。
- 若设置为-1.0，则表示为禁用。

```
~use_map_topic (bool, default: false)
```

- 若为 true，AMCL 将订阅地图话题，而不是进行服务调用来获取其地图。New in navigation 1.4.2.

```
~first_map_only (bool, default: false)
```

- 若为 true，AMCL 将使用订阅到的第一个地图，不会使用每次更新获取的新地图。New in navigation 1.4.2.

Laser model 参数

注意：无论使用什么混合权重，权重加总应该等于 1。beam model 使用了所有的 4 种权重: z_hit, z_short, z_max, 和 z_rand。likelihood_field model 仅仅使用了 2 种: z_hit 和 z_rand。

```
~laser_min_range (double, default: -1.0)
```

- 指定最小的扫描范围。
- 若设置为-1.0，则表示使用已报告的激光的最小范围。

```
~laser_max_range (double, default: -1.0)
```

- 指定最大的扫描范围。
- 若设置为-1.0，则表示使用已报告的激光的最大范围。

```
~laser_max_beams (int, default: 30)
```

- 在更新滤波器时，每次扫描中使用多少个均匀分布的 beam。

```
~laser_z_hit (double, default: 0.95)
```

- 模型的 z_hit 部分的混合权重。

```
~laser_z_short (double, default: 0.1)
```

- 模型的 z_short 部分的混合权重。

```
~laser_z_max (double, default: 0.05)
```

- 模型的 z_max 部分的混合权重。

```
~laser_z_rand (double, default: 0.05)
```

- 模型的 z_rand 部分的混合权重。

```
~laser_sigma_hit (double, default: 0.2 meters)
```

- 在 z_hit 部分模型中使用的高斯模型的标准差。

```
~laser_lambda_short (double, default: 0.1)
```

- 模型中 z_short 部分的指数衰减参数。

```
~laser_likelihood_max_dist (double, default: 2.0 meters)
```

- 在地图上进行障碍物膨胀的最大距离，用于 likelihood_field 模型。

```
~laser_model_type (string, default: "likelihood_field")
```

- 需要用到哪种模型， either beam, likelihood_field, or likelihood_field_prob (same as likelihood_field but incorporates the beamskip feature, if enabled). 。

Odometry model 参数

如果~odom_model_type 参数设置为“diff”，则使用《Probabilistic Robotics》p136 中的 sample_motion_model_odometry 算法; 正如本书中所定义一样,该模型使用 odom_alpha_1 到 odom_alpha_4 的噪声参数。

如果~odom_model_type 参数设置为“omni”，那么使用自定义模型用于 omni-directional 基座，该模型使用 odom_alpha_1 到 odom_alpha_5 的噪声参数。前 4 个参数类似于“diff”模型，第 5 个参数用于捕获机器人在垂直于前进方向的位移（没有旋转）趋势。

```
~odom_model_type (string, default: "diff")
```

- 需要使用哪个模型，either "diff", "omni", "diff-corrected" or "omni-corrected"。

```
~odom_alpha1 (double, default: 0.2)
```

- 基于机器人运动旋转分量，来指定里程旋转估计中预期的噪声。
- Specifies the expected noise in odometry's rotation estimate from the rotational component of the robot's motion.

```
~odom_alpha2 (double, default: 0.2)
```

- 基于机器人运动旋转分量，来指定里程平移估计中预期的噪声。
- Specifies the expected noise in odometry's rotation estimate from translational component of the robot's motion.

```
~odom_alpha3 (double, default: 0.2)
```

- 基于机器人运动平移分量，来指定里程平移估计中预期的噪声。
- Specifies the expected noise in odometry's translation estimate from the translational component of the robot's motion.

```
~odom_alpha4 (double, default: 0.2)
```

- 基于机器人运动平移分量，来指定里程旋转估计中预期的噪声。
- Specifies the expected noise in odometry's translation estimate from the rotational component of the robot's motion.

  ~odom_alpha5 (double, default: 0.2)
- Translation-related noise parameter (only used if model is "omni").

```
~odom_frame_id (string, default:"odom")
```

- 里程计使用的坐标系。

```
~base_frame_id (string, default: "base_link")
```

- 移动基座使用的坐标系。

```
~global_frame_id (string, default:"map")
```

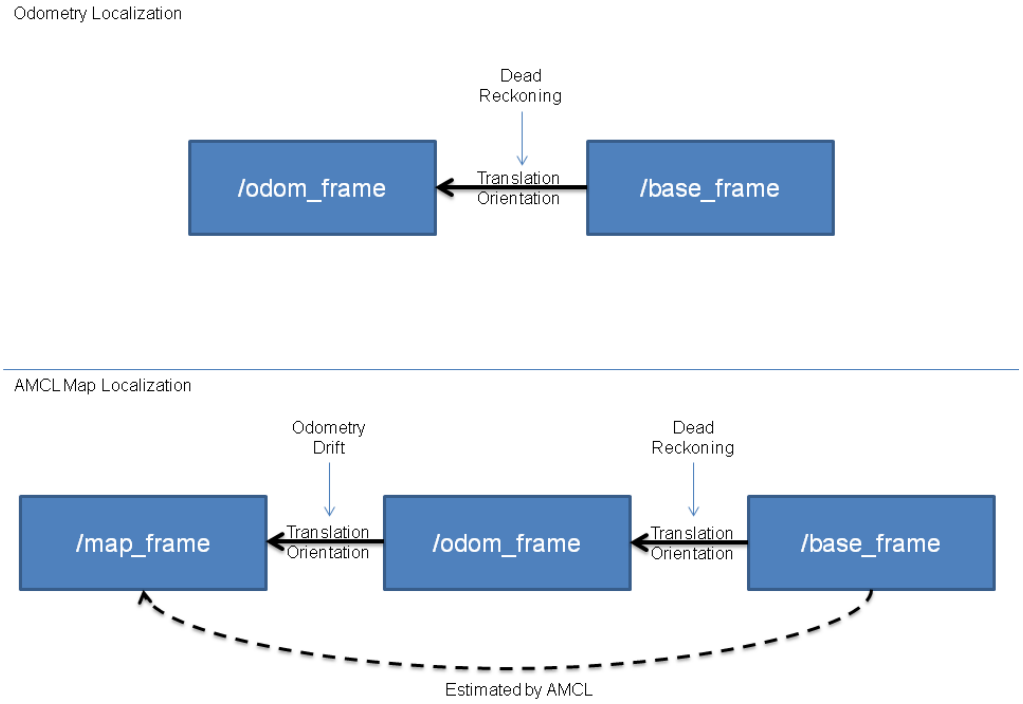
- 由定位系统发布的坐标系的名称。

```
~tf_broadcast (bool, default: true)
```

- 但若设置为 false，则可防止 amcl 在全局坐标系和里程计坐标系之间发布 tf 变换。

(1.6) Transforms

- amcl 要把进来的激光扫描信息转换到里程坐标系 (~odom_frame_id)。因此，tf 树中必定会存在一条从发布激光的坐标系到里程坐标系的路径。
- 实现细节：在接收到第一次激光扫描时，amcl 查找激光坐标系和基座坐标系（~base_frame_id）之间的的变换，并将其永久锁存（latches it forever）。 所以 amcl 不能处理相对于基地移动的激光。
- 下图显示了使用里程计和 amcl 的定位的区别。 在操作期间，amcl 评估了相对于全局坐标系（~global_frame_id）的基座坐标系（~base_frame_id）的变换，但它仅在全局坐标系和里程计坐标系（~odom_frame_id）之间发布变换。 本质上，这种变换解释了使用“Dead Reckoning”进行的累计漂移。AMCL 发布的转换可能带有未来时间戳，但是对 AMCL 来说是有效的。



参考资料

- wiki.ros.org/amcl
- blog.csdn.net/x_r_su/article/details/53396564

ROS 与 navigation 教程-move_slow_and_clear 介绍

说明：

- 介绍 move_slow_and_clear 的概念和相关参数

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- move_slow_and_clear::MoveSlowAndClear 是一种简单的修复机制，用来清除代价地图中信息并且限制机器人移动速度。
- 注意：这种修复技术是非安全技术，机器人可能会撞上东西，它只以用户指定速度移动。
- 这种修复机制仅仅兼容可以通过 dynamic_reconfigure 配置最大速度局部规划器（如 dwa_local_planner）。

MoveSlowAndClear

- move_slow_and_clear::MoveSlowAndClear 导出了 ROS 封装的 C++接口，在初始化时候定义的 ROS 命名空间工作，且继承了 nav_core::RecoveryBehavior 接口。

(1) API Stability

- C++ API 是稳定的。
- ROS API 是稳定的。

(2) ROS Parameters

```
~<name>/clearing_distance (double, default: 0.5)
```

- 清除以机器人为中心多少米半径范围外的障碍。

```
~<name>/limited_trans_speed (double, default: 0.25)
```

- 执行该修复行为时候限定的平移速度是为少，单位为 meters/second 。

```
~<name>/limited_rot_speed (double, default: 0.25)
```

- 执行该修复行为时候限定的角速度是多少,单位为 radians/second 。

```
~<name>/limited_distance (double, default: 0.3)
```

- 在速度限制解除之前机器人必须移动的距离,单位为 meters 。
- The distance in meters the robot must move before the speed restrictions are lifted.

```
~<name>/planner_namespace (string, default: "DWAPlannerROS")
```

- 用于重配置参数的规划器命名空间。
- 具体说， 命名空间内 max_trans_vel 和 max_rot_vel 参数将被重新配置。

(3) C++ API

- C ++ move_slow_and_clear :: MoveSlowAndClear 类继承 nav_core 包中的 nav_core :: RecoveryBehavior 接口。
- 详细资料请参阅 MoveSlowAndClear Documentation`。

参考资料

- http://wiki.ros.org/move_slow_and_clear
- http://blog.csdn.net/x_r_su/article/details/53379310

ROS 与 navigation 教程-clear_costmap_recovery 介绍

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- This package provides a recovery behavior for the navigation stack that attempts to clear space by reverting the costmaps used by the navigation stack to the static map outside of a given area.
- The clear_costmap_recovery::ClearCostmapRecovery is a simple recovery behavior that clears out space in the navigation stack's costmaps by reverting to the static map outside of a given radius away from the robot.
- 其继承了 nav_core 包中接口 nav_core::RecoveryBehavior ，以插件方式在 move_base node 中使用。

ClearCostmapRecovery

- clear_costmap_recovery::ClearCostmapRecovery 对象将其功能以 C++ ROS Wrapper 导出，在初始化时指定的 ROS 命名空间使用。
- 以下是创建 clear_costmap_recovery::ClearCostmapRecovery 对象的例子：

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <clear_costmap_recovery/clear_costmap_recovery.h>

...
tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS global_costmap("global_costmap", tf);
costmap_2d::Costmap2DROS local_costmap("local_costmap", tf);

clear_costmap_recovery::ClearCostmapRecovery ccr;
ccr.initialize("my_clear_costmap_recovery", &tf, &global_costmap, &local_costmap);

ccr.runBehavior();
```

(1) API Stability

- C++ API 是稳定的。
- ROS API 是稳定的。

(2) ROS Parameters

```
~<name>/reset_distance (double, default: 3.0)
```

- The radius away from the robot in meters outside which obstacles will be removed from the costmaps when they are reverted to the static map.

(3) C++ API

- C ++ clear_costmap_recovery :: ClearCostmapRecovery 类继承 nav_core 包中的 nav_core :: RecoveryBehavior 接口。
- 详细资料请参阅 ClearCostmapRecovery Documentation 。

ROS 与 navigation 教程-rotate_recovery 介绍

说明：

- 介绍了 rotate_recovery 的概念及其相关知识。

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- 该包给导航功能包提供了 rotate_recovery::RotateRecovery 修复机制，它尝试让机器人执行 360 度旋转来完成清理导航功能包里的代价地图的空间。
- 同时 rotate_recovery::RotateRecovery 继承了 nav_core::RecoveryBehavior 接口，并以插件方式用于 move_base node。

RotateRecovery

- rotate_recovery::RotateRecovery 将功能以 C++ ROS Wrapper 导出，在初始化时指定的 ROS 命名空间中使用。
- 以下是创建 rotate_recovery::RotateRecovery 对象的示例:

```
#include <tf/transform_listener.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <rotate_recovery/rotate_recovery.h>

...
tf::TransformListener tf(ros::Duration(10));
costmap_2d::Costmap2DROS global_costmap("global_costmap", tf);
costmap_2d::Costmap2DROS local_costmap("local_costmap", tf);

rotate_recovery::RotateRecovery rr;
rr.initialize("my_rotate_recovery", &tf, &global_costmap, &local_costmap);

rr.runBehavior();
```

(1) API Stability

- C++ API 是稳定的。
- ROS API 是稳定的。

(2) ROS Parameters

- rotate_recovery::RotateRecovery 对象假定 move_base node 使用的局部路径规划器是 base_local_planner::TrajectoryPlannerROS，并相应地读取其中的一些参数。
- 其将会独立工作，同时需要用户指定其他参数。

(2.1) RotateRecovery Parameters

```
~<name>/sim_granularity (double, default: 0.017)
```

- 在检查原地旋转是否安全时，检查机器人与障碍物之间的距离，单位为 radians 。
- 默认为 1 度。

```
~<name>/frequency (double, default: 20.0)
```

- 向移动基座发送速度命令的频率，单位为 HZ 。

(2.2) TrajectoryPlannerROS Parameters

- 当使用 base_local_planner :: TrajectoryPlannerROSjubu 局部路径规划器时，这些参数已经设置好的。
- 只有当导航功能包集中的其它局部规划器被 rotate_recovery::RotateRecovery 使用时才有必要设置这些参数。

```
~TrajectoryPlannerROS/yaw_goal_tolerance (double, default: 0.05)
```

- The tolerance in radians for the controller in yaw/rotation when achieving its goal.

```
~TrajectoryPlannerROS/acc_lim_th (double, default: 3.2)
```

- 机器人的角速度极限，单位为 radians/sec^2 。

```
~TrajectoryPlannerROS/max_rotational_vel (double, default: 1.0)
```

- 移动基座允许的最大角速度，单位为 radians/sec 。

```
~TrajectoryPlannerROS/min_in_place_rotational_vel (double, default: 0.4)
```

- 移动基座在执行原地旋转时的最小角速度，单位为 radians/sec 。

(3) C++ API

- C ++ rotate_recovery :: RotateRecovery 类继承 nav_core 包中的 nav_core :: RecoveryBehavior 接口。

- 详细资料请参阅 RotateRecovery documentation。

参考资料

- http://wiki.ros.org/rotate_recovery
- http://blog.csdn.net/x_r_su/article/details/53405850

ROS 与 navigation 教程-costmap_2d 介绍

说明：

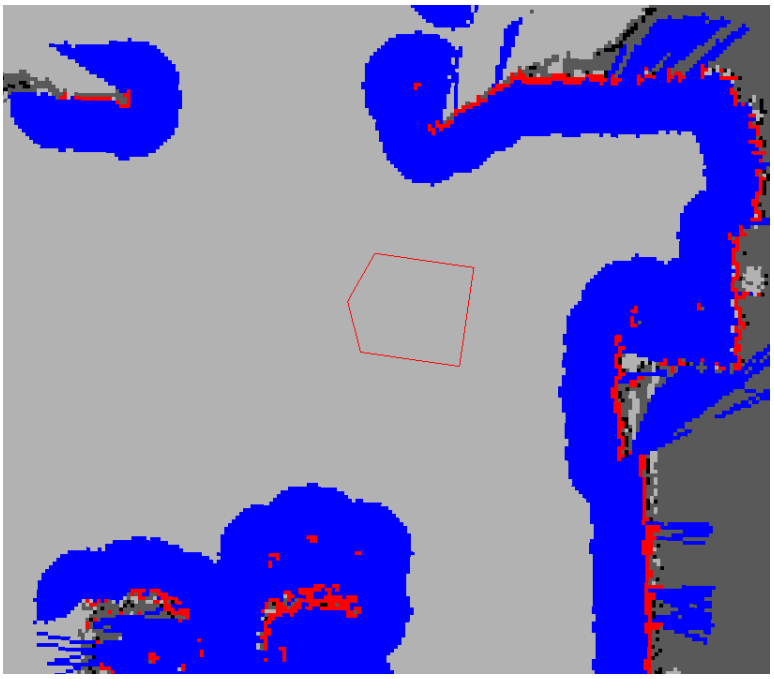
- 介绍 costmap_2d 的概念和相关知识。

代码库

- 参考链接：https://github.com/ros-planning/navigation

概要

- costmap_2d 包提供了一种 2D 代价地图的实现方案，该方案从实际环境中获取传感器数据，构建数据的 2D 或 3D 占用栅格（取决于是否使用基于体素的实现），以及基于占用栅格和用户定义膨胀半径的 2D 代价地图的膨胀代价。
- 该包也支持基于 map_server 初始化代价地图，基于滚动窗口的代价地图，以及基于参数化订阅和配置传感器主题。



- 注意：在上图中，红色单元表示代价地图中的障碍物，蓝色单元表示由机器人内切半径来计算膨胀的障碍物，红色多边形表示机器人的垂直投影（footprint）。 为了避免机器人与障碍碰撞，机器人的垂直投影（footprint）不能与红色单元相交，机器人的中心不能穿过蓝色单元。
- costmap_2d 包提供了一种可配置框架来维护机器人在占用栅格应该如何导航的信息。代价地图使用传感器数据和来自静态地图的信息，通过 costmap_2d :: Costmap2DROS 对象来存储和更新实际环境中的障碍物信息。costmap_2d :: Costmap2DROS 对象为其用户提供纯 2D 接口，这意味着关于障碍的查询只能在列中进行。 例如，在 XY 平面中具有不同 Z 位置的相同位置的表和鞋将导致 costmap_2d :: Costmap2DROS 对象的代价地图中具有相同成本值的相应单元格。这种设计对平面空间进行路径规划是有帮助的。
- 从 Hydro 发布版本开始， 用来写数据到代价地图的底层方法已经完全可配置了。每种功能放置一层中。例如，静态地图是一层，障碍物是另一层。默认情况下，障碍层维护 3D 信息（参阅 voxel_grid）。维护 3D 障碍数据可以使图层更智能地处理标记和清除。
- 该包提供的主要接口是 costmap_2d :: Costmap2DROS，其保留了大部分与 ROS 相关的功能。它包含一个 costmap_2d :: LayeredCostmap，用于跟踪每一层。每一层在 Costmap2DROS 中以插件方式被实例化，并被添加到 LayeredCostmap。每一层可以独立编译，且可使用 C++接口实现对代价地图的随意修改。costmap_2d::Costmap2D 类中实现了用来存储和访问 2D 代价地图的的基本数据结构。

- 有关代价地图如何更新占用栅格的详细信息将会在下面提到，同时提供了页面链接供查看某一层具体如何工作。

Marking and Clearing

- 成本图自动预订 ROS 上的传感器话题，并相应进行更新。每个传感器可以执行标记操作（将障碍物信息插入到成本图中），清除操作（从成本图中删除障碍物信息）或两者都同时执行。标记操作就是索引到数组内修改单元格的代价。然而对于清除操作，每次观测报告都需要传感器源向外发射线。如果存储的障碍物信息是 3D 的，需要将每一列的障碍物信息投影成 2D 后才能放入到代价地图。

Occupied, Free, and Unknown Space

- 虽然代价图中的每个单元格可以具有 255 个不同的代价值之一（见“inflation”部分），但下层数据结构仅需要 3 个值。在该下层结构中的每个单元有三种状态：自由的，占用的或未知的。 每个状态在投影到代价图中后都会有其特定的代价值。每个状态在投影到代价图中后都会有其特定的代价值。 具有一定数量的占用单元格的列（参阅 mark_threshold 参数）会被分配了 costmap_2d :: LETHAL_OBSTACLE 代价值;具有一定数量的未知单元格的列（参见 unknown_threshold 参数）被分配了 costmap_2d :: NO_INFORMATION 代价值，剩余的其他列分配了 costmap_2d :: FREE_SPACE 代价值。

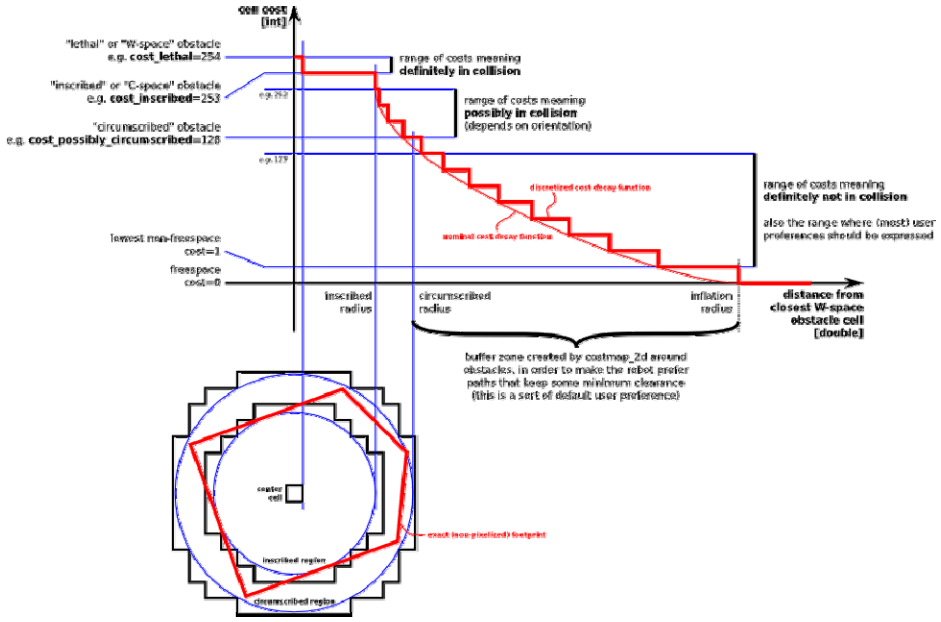
Map Updates

- 代价地图以参数 update_frequency 指定的周期进行地图更新。每个周期获取传感器数据后，代价地图底层占用结构都会执行标记和清除操作，并且将该结构投影到代价图且附上相应代价值。然后，每个代价值为 costmap_2d::LETHAL_OBSTACLE 的单元格都会执行障碍物膨胀操作，即从每个占用单元格向外传播代价值，直到用户定义的膨胀半径为止。

tf

- 为了把来自传感器源的数据插入到代价地图，costmap_2d::Costmap2DROS 要大量使用 tf。 具体来说，它假定由 global_frame 参数，robot_base_frame 参数和传感器源指定的坐标系之间的所有变换都是有联系的且是最新的。transfor_tolerance 参数设置了这些变换之间的最大延迟量（amount of latencylatency）。如果 tf 树没有以期望速度来更新，那么导航功能包将会让机器人停止。

Inflation



- 膨胀代价值随着机器人离障碍物距离增加而减少。
- 为代价地图的代价值定义 5 个与机器人有关的标记：
  - 致命的（"Lethal" cost）：说明该单元格中存在一个实际的障碍。若机器人的中心在该单元格中，机器人必然会跟障碍物相撞。
  - 内切（"Inscribed" cost）：说明该单元格离障碍物的距离小于机器人内切圆半径。若机器人的中心位于等于或高于"Inscribed" cost 的单元格，机器人必然会跟障碍物相撞。
  - 可能外切（"Possibly circumscribed" cost ）：说明一个单元格离障碍物的距离小于机器人外切圆半径，但是大于内切圆半径。若机器人的中心位于等于或高于"Possibly circumscribed" cost 的单元格，机器不一定会跟障碍物相撞，其取决于机器人的方位。
  - 自由空间（"Freespace" ）：没有障碍的空间。
  - 未知（"Unknown"） ： 未知空间。
  - All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.
- The rationale behind these definitions is that we leave it up to planner implementations to care or not about the exact footprint, yet give them enough information that they can incur the cost of tracing out the footprint only in situations where the orientation actually matters.

Map Types

- 初始化 costmap_2d :: Costmap2DROS 对象的方法主要有两种。
- 第一种是使用用户生成的静态地图进行初始化（有关构建地图的文档，请参阅 map_server 包）。在这种情况下，代价地图被初始化来匹配与静态地图提供的宽度，高度和障碍物信息。这种方式通常与定位系统（如 AMCL）结合使用，允许机器人在地图坐标系注册障碍物，并且当基座在环境中移动时可以根据传感器数据更新其代价地图。
- 第二种是给出一个宽度和高度，并将 rolling_window 参数设置为 true。当机器人在移动时，rolling_window 参数将机器人保持在成本图的中心。然而当机器人在给定区域移动得太远时，障碍物信息将会被从地图中丢弃。这种方式的配置最常用于里程计坐标系中，机器人只关注局部区域内的障碍物。

Component API

(1) Costmap2DROS

- costmap_2d::Costmap2DROS 是对 costmap_2d::Costmap2D 的封装，将其功能以 C++ ROS Wrapper 导出，可以在初始化时指定的 ROS 命名空间使用。
- 以下是在指定的 my_costmap 命名空间创建 costmap_2d::Costmap2DROS 对象：

  #include <tf/transform_listener.h>

  #include <costmap_2d/costmap_2d_ros.h>

  ...

  tf::TransformListener tf(ros::Duration(10));

  costmap_2d::Costmap2DROS costmap("my_costmap", tf);
- 如果您直接 rosrun 或 roslaunch 的 costmap_2d 节点，其将在 costmap 命名空间内运行。在这种情况下，所有对下面的名称的引用都应该被替换成 costmap。
- 更常见的情况是通过启动 move_base 节点来运行完整的导航功能包。 这将创建 2 个 costmaps，每个都有自己的命名空间：local_costmap 和 global_costmap。 这就可能需要为每个代价地图设置参数。

(1.1) ROS API

The C++ API has changed as of Hydro.

(1.2)

```
~<name>/footprint ([geometry_msgs/Polygon][7])
```

- 机器人脚印的规格。This replaces the previous parameter specification of the footprint.

(1.3) Published Topics

```
~<name>/grid ([nav_msgs/OccupancyGrid][8])
```

- 代价地图的值。
- The values in the costmap .

```
~<name>/grid_updates ([map_msgs/OccupancyGridUpdate][9])
```

- 代价图的更新区域的值。

```
~<name>/voxel_grid ([costmap_2d/VoxelGrid][10])
```

- Optionally advertised when the underlying occupancy grid uses voxels and the user requests the voxel grid be published.

(1.4) Parameters

- Hydro 和更高版本使用所有 costmap_2d 层的插件。如果您没有提供插件参数，那么初始化代码将假定您的配置为 Hydro，并将使用默认命名空间加载默认的插件集。您的参数将自动移动到新的命名空间。默认的命名空间是 static_layer，barriers_layer 和 inflation_layer。同时要注意一些教程（和书籍）仍然是用 Hydro 前的参数。所以请务必提供一个插件参数。

Plugins

```
~<name>/plugins (sequenc
```

e, default: pre-Hydro behavior)

- 插件说明的顺序，每层一个。每个说明都是具有名称和类型字段的字典。该名称用来定义插件的参数命名空间。
- 有关示例，请参阅 tutorials。

Coordinate frame and tf parameters

```
~<name>/global_frame (string, default: "/map")
```

- The global frame for the costmap to operate in.

```
~<name>/robot_base_frame (string, default: "base_link")
```

- 机器人 base_link 的坐标系名称。

```
~<name>/transform_tolerance (double, default: 0.2)
```

- 指定在 transform（tf）数据中可接受的延迟，单位为 seconds 。
- 此参数可用于在 tf 树中丢失链接的情况，同时仍允许用户在系统中存在一定的延迟。
- 例如，0.2 秒超时的变换可能是可以接受的，但变化为 8 秒过期的转换不是。如果一个 tf 变换超过了 tolerance 限定的时间，机器人将被停止。

Rate parameters

```
~<name>/update_frequency (double, default: 5.0)
```

- 地图更新的频率，单位为 Hz 。

```
~<name>/publish_frequency (double, default: 0.0)
```

- 地图发布到显示信息的频率。

Map management parameters

```
~<name>/rolling_window (bool, default: false)
```

- 是否使用滚动窗口模式的代价地图。
- 如果 static_map 参数设置为 true，则此参数必须设置为 false。

```
~<name>/always_send_full_costmap (bool, default: false)
```

- 假若为 true，则每次更新都会将完整的代价地图发布到 "~/grid"。但若为 false，则仅在 "~/grid" 话题上发布已更改的代价地图的部分。

以下参数可以被某些层重写，即静态地图层。

```
~<name>/width (int, default: 10)
```

- 地图宽度，单位为 meters 。

```
~<name>/height (int, default: 10)
```

- 地图高度，单位 meters 。

```
~<name>/resolution (double, default: 0.05)
```

- 地图分辨率，单位 meters 。

```
~<name>/origin_x (double, default: 0.0)
```

- 全局地图中的 x 原点，单位 meters 。

```
~<name>/origin_y (double, default: 0.0)
```

- 全局地图中的 y 原点，单位 meters 。

(1.5)Required tf Transforms

- (value of global_frame parameter) → (value of robot_base_frame parameter)
- 通常都是由负责里程计或者定位的节点，比如 amcl 节点，来提供这个 tf 变换。

(1.6)C++ API

- 有关 costmap_2d :: Costmap2DROS 类的 C ++levelAPI 文档，请参阅：Costmap2DROS C ++ API。

(2) Layer Specifications

(2.1) Static Map Layer

- 静态地图层代表代价地图的一个很大的不变部分，就像用 SLAM 生成的一样。

(2.2) Obstacle Map Layer

- 障碍层读取传感器数据来追踪障碍物。 ObstacleCostmapPlugin 标记和光线追踪 2D 的障碍，而 VoxelCostmapPlugin 也是用同样的方式追踪 3D 的障碍。

(2.3) Inflation Layer

- 膨胀层是对致命障碍（lethal obstacles）周围边界的扩展（即是障碍物膨胀），以便代价地图更清晰显示机器人的可利用空间。

(2.4) Other Layers

- 其他层可以通过 pluginlib 在 costmap 中实现和使用。
  - Social Costmap Layer
  - Range Sensor Layer

参考资料

- http://wiki.ros.org/costmap_2d/
- http://blog.csdn.net/sonictl/article/details/51518492
- http://blog.csdn.net/x_r_su/article/details/53408528

ROS 与 navigation 教程-costmap_2d-range_sensor_layer 介绍

说明：

- 介绍 range_sensor_layer 的概念以及相关知识。

代码库

- 参考链接：https://github.com/DLu/navigation_layers

概要

- range_sensor_layer 是 costmap_2d 中 LayeredCostmap 的一个插件，使用消息类型是 sensor_msgs/Range ，适用于声呐和红外传感器数据传输。
- Range 消息通过使用概率模型整合到代价图中。
- 在主代价地图（master costmap）中，高于 mark_threshold 的概率的单元格会被标记为致命障碍（lethal obstacles），低于 clear_threshold 的概率的单元格会被标记为自由空间。

API

(1) Subscribed Topics

```
"topics" ([sensor_msgs/Range][2])
```

- 距离传感器数据。

(2) Parameters

```
ns (string, default: "")
```

- 命名空间，用作所有 topic 的前缀。

```
topics (Array of strings, default: ['/sonar'])
```

- 列出要订阅的 Range 话题列表

```
no_readings_timeout (double, default: 0.0)
```

- 如果为 0，则该参数无效。
- 否则如果该层在该参数指定时间内没有收到传感器任何数据，该层会给出告警并被标记为没有数据流。

```
clear_threshold (double, default: .2)
```

- 在主代价地图（master costmap）中，低于 clear_threshold 的概率的单元格会被标记为自由空间。

```
mark_threshold (double, default: .8)
```

- 在主代价地图（master costmap）中，高于 mark_threshold 的概率的单元格会被标记为致命障碍（lethal obstacles）。

```
clear_on_max_reading (bool, default: false)
```

- 是否清除最大距离内的传感器读数。

Usage

- 该层可以通过将以下值添加到代价地图中的插件参数来使用。

```
{name: sonar,  type: "range_sensor_layer::RangeSensorLayer"}
```

参考链接

- http://wiki.ros.org/range_sensor_layer
- http://blog.csdn.net/x_r_su/article/details/53418044

ROS 与 navigation 教程-costmap_2d-social_navigation_layers 介绍

说明：

- 介绍 social_navigation_layers 的概念和相关知识。

代码库

- 参考链接：https://github.com/DLu/navigation_layers

概要

- Plugin-based layers for the navigation stack that implement various social navigation contraints, like proxemic distance.
- 目前有 2 种 social navigation 层， 但是它们共享部分功能（如：都需要订阅人在哪里的消息，都是用高斯分布调节人周围的代价地图）。
- 这种方法的类都继承了 general SocialLayer class。
- 在 costmap configurations 中使用时类型如下:
  - social_navigation_layers::ProxemicLayer
  - social_navigation_layers::PassingLayer

(1) ProxemicLayer/PassingLayer

- 这两种层所用接口相同。

(1.1) Subscribed Topics

```
/people ([people_msgs/People][3])
```

- People to navigate around .

(1.2) Parameters

```
enabled (bool, default: True)
```

- 是否使用该插件。

```
cutoff (double, default: 10.0)
```

- Smallest value to publish on costmap adjustments

```
amplitude (double, default: 77.0)
```

- Amplitude of adjustments at peak

```
covariance (double, default: 0.25)
```

- Covariance of adjustments

```
factor (double, default: 5.0)
```

- Factor with which to scale the velocity .

```
keep_time (double, default: 0.75)
```

- 在清除 leg list 之前暂停。

How the Costmap Changes

(1) Proxemic Layer

- proxemic 层使用上述参数在检测到的人员周围增加高斯代价（gaussian costs）。
- 如果人是静止的，高斯代价分布是圆形。
- 如果认识运动的，代价会在人移动方向上增长，至于代价在人周围增长多远与比例参数 factor 有关。
- The proxemic layer adds gaussian costs all around the detected person, with the parameters specified above. If the person is stationary, the gaussian is perfectly round. However, if the person is moving, then the costs will be increased in the direction of their motion. How far in front of the person the costs are increased is proportional to the factor parameter.

(1.1) Stationary Person

(1.2) Moving Person

(2) Passing Layer

- 这一层的目的是让机器人从行人的一侧通过，因此只在行人一侧增加代价。

参考链接

- http://wiki.ros.org/social_navigation_layers
- http://blog.csdn.net/x_r_su/article/details/53418225

ROS 与 navigation 教程-costmap_2d-staticmap 介绍

说明:

- 介绍 costmap_2d-staticmap 的概念及其相关知识

概要

- 静态地图主要包含来自外部源的不变数据（有关构建地图的资料，请参阅 map_server 包）。

Subscribed Topics

```
"map" ([nav_msgs/OccupancyGrid][2])
```

- 代价地图可以选择使用用户定义的静态地图来进行初始化（参考 static_map 参数）
- 如果该选项被选择，代价地图将使用 service call 从 map_server 获取静态地图。

Parameters

```
unknown_cost_value (int, default: -1)
```

- The value for which a cost should be considered unknown when reading in a map from the map server. If the costmap is not tracking unknown space, costs of this value will be considered occupied. A value of zero also results in this parameter being unused.

```
lethal_cost_threshold (int, default: 100)
```

- The threshold value at which to consider a cost lethal when reading in a map from the map server.

```
map_topic (string, default: "map")
```

- The topic that the costmap subscribes to for the static map. This parameter is useful when you have multiple costmap instances within a single node that you want to use different static maps. - New in navigation 1.3.1

```
first_map_only (bool, default: false)
```

- 只订阅地图上的第一条消息主题，忽略所有后续消息。

```
subscribe_to_updates (bool, default: false)
```

- 除了 map_topic，还订阅 map_topic +“_updates”

```
track_unknown_space (bool, default: true)
```

-如果为 true，则地图消息中的未知值将直接转换到该层。 否则，地图消息中的未知值将在层中转换为 FREE_SPACE。

```
use_maximum (bool, default: false)
```

- Only matters if the static layer is not the bottom layer. If true, only the maximum value will be written to the master costmap.

```
trinary_costmap (bool, default: true)
```

- 如果为 true，则将所有地图消息值转换为 NO_INFORMATION / FREE_SPACE / LETHAL_OBSTACLE（三个值）。
- If false, a full spectrum of intermediate values is possible.

**参考资料*

- http://wiki.ros.org/costmap_2d/hydro/staticmap
- http://blog.csdn.net/x_r_su/article/details/53420130

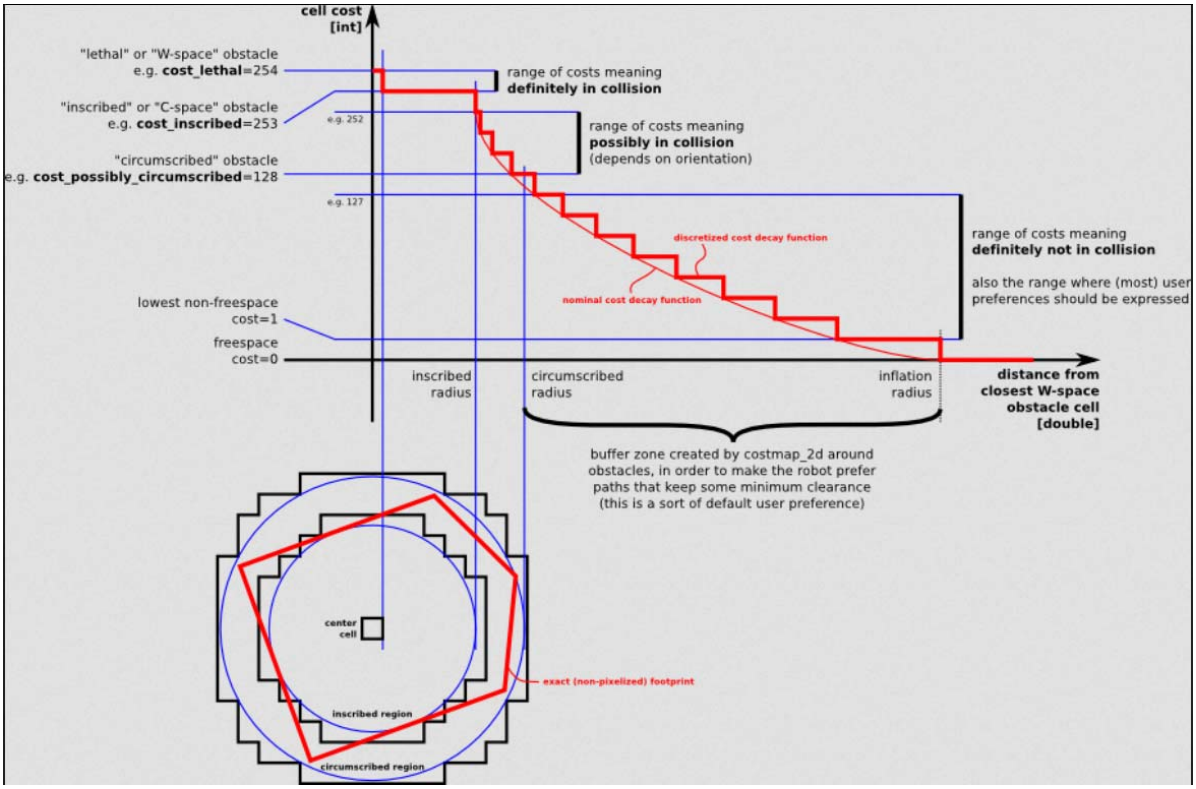
ROS 与 navigation 教程-costmap_2d-inflation 介绍

说明：

- 介绍 costmap_2d-inflation 的概念和相关知识。

Inflation Costmap Plugin

(1) Inflation



- 膨胀代价值随着机器人离障碍物距离增加而减少。
- 为代价地图的代价值定义 5 个与机器人有关的标记：
  - 致命的（"Lethal" cost）：说明该单元格中存在一个实际的障碍。若机器人的中心在该单元格中，机器人必然会跟障碍物相撞。
  - 内切（"Inscribed" cost）：说明该单元格离障碍物的距离小于机器人内切圆半径。若机器人的中心位于等于或高于"Inscribed" cost 的单元格，机器人必然会跟障碍物相撞。
  - 可能外切（"Possibly circumscribed" cost ）：说明一个单元格离障碍物的距离小于机器人外切圆半径，但是大于内切圆半径。若机器人的中心位于等于或高于"Possibly circumscribed" cost 的单元格，机器不一定会跟障碍物相撞，其取决于机器人的方位。
  - 自由空间（"Freespace" ）：没有障碍的空间。
  - 未知（"Unknown"） ： 未知空间。
  - All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

- The rationale behind these definitions is that we leave it up to planner implementations to care or not about the exact footprint, yet give them enough information that they can incur the cost of tracing out the footprint only in situations where the orientation actually matters.

(2) ROS API

(2.1) Parameters

```
~<name>/inflation_radius (double, default: 0.55)
```

- The radius in meters to which the map inflates obstacle cost values.
- 障碍物在地图中向外扩展的膨胀区的半径，单位为 meters 。

```
~<name>/cost_scaling_factor (double, default: 10.0)
```

- 膨胀过程中应用到代价值的比例因子。
- 成本函数的计算方法如下：exp(-1.0 * cost_scaling_factor * (distance_from_obstacle - inscribed_radius)) * (costmap_2d::INSCRIBED_INFLATED_OBSTACLE - 1)，公式中 costmap_2d::INSCRIBED_INFLATED_OBSTACLE 目前指定为 254。
- 注意：由于在公式中将 cost_scaling_factor 乘以负数，所以增大比例因子会降低代价值。
- A scaling factor to apply to cost values during inflation. The cost function is computed as follows for all cells in the costmap further than the inscribed radius distance and closer than the inflation radius distance away from an actual obstacle: exp(-1.0 * cost_scaling_factor * (distance_from_obstacle - inscribed_radius)) * (costmap_2d::INSCRIBED_INFLATED_OBSTACLE - 1), where costmap_2d::INSCRIBED_INFLATED_OBSTACLE is currently 254. NOTE: since the cost_scaling_factor is multiplied by a negative in the formula, increasing the factor will decrease the resulting cost values.

参考资料

- http://wiki.ros.org/costmap_2d/hydro/inflation
- http://blog.csdn.net/sonictl/article/details/51518492
- http://www.cnblogs.com/hong2016/p/6863074.html
- http://blog.csdn.net/x_r_su/article/details/53420209

ROS 与 navigation 教程-obstacle 层介绍

说明：

- 介绍 obstacle 层的概念和相关知识。

概要

- obstacle 和 voxel 层包含了 PointClouds 或者 LaserScans 形式的传感器消息。
- 障碍物层用于二维追踪，体素层用于三维追踪。

Marking and Clearing

- 代价地图自动订阅传感器主题并自动更新。
- 每个传感器用于标记操作（将障碍物信息插入到代价地图中），清除操作（从代价地图中删除障碍物信息）或两者操作都执行。
- 如果使用的是体素层，每一列上的障碍信息需要先进行投影转化成二维之后才能放入代价地图中。

ROS API

(1) Subscribed Topics

```
<point_cloud_topic> ([sensor_msgs/PointCloud][3])
```

- 对于 observe_sources 参数列表中列出的每个“PointCloud”源，其信息用于更新代价地图。

```
<point_cloud2_topic> ([sensor_msgs/PointCloud2][4])
```

- 对于 observe_sources 参数列表中列出的每个“PointCloud2”源，其信息用于更新代价地图。

```
<laser_scan_topic> ([sensor_msgs/LaserScan][5])
```

- 对于 observe_sources 参数列表中列出的每个“LaserScan”源，其信息用于更新代价地图。

  "map" (nav_msgs/OccupancyGrid)

- 代价地图可以从用户生成的静态地图中进行初始化（参考 static_map 参数）。
- 如果选择此选项，则 costmap 将调用 map_server 以获取此地图。

(2) Sensor management parameters

```
~<name>/observation_sources (string, default: "")
```

- 观察源列表以空格分割表示，定义了下面参数中每一个 <source_name> 命名空间。

observe_sources 中的每个 source_name 定义了可设置参数的命名空间：

```
~<name>/<source_name>/topic (string, default: source_name)
```

- 传感器数据来源的话题。
- 默认为该源的名称。

```
~<name>/<source_name>/sensor_frame (string, default: "")
```

- 指定传感器原点坐标系。
- 默认设为空，则从传感器数据读取坐标系。
- 可以从 sensor_msgs / LaserScan，sensor_msgs / PointCloud 和 sensor_msgs / PointCloud2 消息读取该坐标系。

```
~<name>/<source_name>/observation_persistence (double, default: 0.0)
```

- 传感器读数保存多长时间，单位为 seconds 。
- 若设为 0.0，则为保存最新读数信息。

```
~<name>/<source_name>/expected_update_rate (double, default: 0.0)
```

- 读取传感器数据的频率，单位为 seconds 。
- 若设为 0.0，则为不断续地读取传感器数据。
- 当传感器发生故障时，该参数用作故障保护来让导航堆栈不能命令机器人。其应该被设置为比传感器的实际速率稍多的值。
- 如果每 0.05 秒进行一次激光扫描，可以将此参数设置为 0.1 秒，以提供大量的缓冲区并占用一定的系统延迟。

```
~<name>/<source_name>/data_type (string, default: "PointCloud")
```

- 指定主题相关的数据类型，目前可用数据类型有 "PointCloud", "PointCloud2"，和 "LaserScan"。

```
~<name>/<source_name>/clearing (bool, default: false)
```

- 这种观察源是否应用于清除自由空间。

```
~<name>/<source_name>/marking (bool, default: true)
```

- 这种观察源是否应用于标记障碍。

```
~<name>/<source_name>/max_obstacle_height (double, default: 2.0)
```

- 传感器读数的最大有效高度，单位为 meters。
- 通常设置为略高于机器人的高度。将此参数设置为大于全局 max_obstacle_height 参数的值，则为无效；设置为小于全局 max_obstacle_height 的值将过滤掉高于该高度的该传感器的点。

```
~<name>/<source_name>/min_obstacle_height (double, default: 0.0)
```

- 传感器读数的最大有效高度，单位为 meters。
- 传感器读数的最小高度（以米为单位）被认为有效。 这通常设置为地面高度，但可以根据传感器的噪声模型设置更高或更低。

```
~<name>/<source_name>/obstacle_range (double, default: 2.5)
```

-将障碍物插入代价地图的最大范围，单位为 meters 。

```
~<name>/<source_name>/raytrace_range (double, default: 3.0)
```

- 从地图中扫描出障碍物的最大范围，单位为 meters 。

```
~<name>/<source_name>/inf_is_valid (bool, default: false)
```

- 允许使用在“LaserScan”观察消息中的值。
- Inf 值被转换为激光最大范围。
- Allows for Inf values in "LaserScan" observation messages. The Inf values are converted to the laser maximum range.

(3) Global Filtering Parameters

以下参数适用于所有传感器。

```
~<name>/max_obstacle_height (double, default: 2.0)
```

- 插入代价地图中的障碍物的最大高度，单位为 meter 。
- 该参数应设置为略高于机器人的高度。

```
~<name>/obstacle_range (double, default: 2.5)
```

- 距离机器人的默认最大距离，以该距离为半径的范围来为代价地图插入障碍物，单位为 meters 。
- 这可以在每个传感器的基础上进行覆盖。

```
~<name>/raytrace_range (double, default: 3.0)
```

- 从地图中扫描出障碍物的默认范围，单位为 meters 。
- 这可以在每个传感器的基础上进行覆盖。

(4) ObstacleCostmapPlugin

以下参数适用于 ObstacleCostmapPlugin

```
~<name>/track_unknown_space (bool, default: false)
```

- 但若为 flase ，每个像素都有 2 个状态之一：lethal obstacle 或自由；若为 ture ，每个像素都有 3 个状态之一：lethal obstacle，自由或未知。

```
~<name>/footprint_clearing_enabled (bool, default: true)
```

- 但若为 ture ，机器人会将其在移动中经过的空间标记为自由空间。

(5) VoxelCostmapPlugin

以下参数适用于 VoxelCostmapPlugin。

```
~<name>/origin_z (double, default: 0.0)
```

- The z resolution of the map in meters/cell.

```
~<name>/z_resolution (double, default: 0.2)
```

- The z resolution of the map in meters/cell.

```
~<name>/z_voxels (int, default: 10)
```

- The number of voxels to in each vertical column, the height of the grid is z_resolution * z_voxels.

```
~<name>/unknown_threshold (int, default: ~<name>/z_voxels)
```

- 在被认为是“known”的列中，允许存在的未知单元格的数量。
- The number of unknown cells allowed in a column considered to be "known"

```
~<name>/mark_threshold (int, default: 0)
```

- The maximum number of marked cells allowed in a column considered to be "free".

```
~<name>/publish_voxel_map (bool, default: false)
```

- 是否发布底层的体素栅格，其主要用于可视化。

ObservationBuffer

costmap_2d :: ObservationBuffer 用于从传感器接收点云，并使用 tf 将其转换为所需的坐标系，并将其存储直到它们被请求为止。大多数用户使用 costmap_2d::Costmap2DROS 自动创建的 costmap_2d::ObservationBuffer 即可满足需求，但为了某些特殊需求可能需要自己创建 costmap_2d::ObservationBuffer。

(1) API Stability

- C++ API 是稳定的。

(2) C++ API

- 有关 costmap_2d::ObservationBuffer 类的 C++文档，请参阅 ObservationBuffer 。

参考资料

- wiki.ros.org/costmap_2d/hydro/obstacles
- blog.csdn.net/x_r_su/article/details/53420310

ROS 与 navigation 教程-Configuring Layered Costmaps

说明：

- 本教程旨在指导您在代价地图中创建自定义的一组图层。
- This will be accomplished using the costmap_2d_node executable, although the parameters can be ported into move_base.

Step 1: Transforms

- One basic thing that the costmap requires is a transformation from the frame of the costmap to the frame of the robot. For this purpose, one solution is to create a static transform publisher in a launch file.

```
   <node name="static_tf0" pkg="tf" type="static_transform_publisher" args="2 0 0 0 0 0 /
map /base_link 100"/>
```

Step 2: Basic Parameters

- 接下来，需要指定 costmap 的行为。
- 创建一个将被加载到参数空间中的 minimal.yaml 文件。

```
plugins: []
publish_frequency: 1.0
footprint: [[-0.325, -0.325], [-0.325, 0.325], [0.325, 0.325], [0.46, 0.0], [0.325, -0.32
5]]
```

- We start out with no layers for simplicity.
- 为了方便进行调试，publish_frequency 为 1.0Hz，我们指定了一个简单的脚本，因为 costmap 需要定义一些 footprint。
- 这些参数可以通过配置以下启动文件来加载

```
<rosparam file="$(find simple_costmap_configuration)/params/minimal.yaml" command="load"
ns="/costmap_node/costmap" />
```

- 注意：如果您在没有插件参数的情况下运行 costmap，则将采用预先配置的参数配置方式，并根据使用的代价地图自动创建图层。

Step 3: Plugins Specification

- 如果要指定图层，其需要将字典存储在插件数组中。 例如：

```
plugins:
- {name: static_map,       type: "costmap_2d::StaticLayer"}
```

- 假设要发布一个类似以下内容的静态地图。

```
   <node name="map_server" pkg="map_server" type="map_server" args="$(find simple_costma
p_configuration)/realmap.yaml"/>
```

- 可以在数组中添加额外的字典。

```
plugins:
- {name: static_map,       type: "costmap_2d::StaticLayer"}
- {name: obstacles,        type: "costmap_2d::VoxelLayer"}
```

- The namespace for each of the layers is one level down from where the plugins are specified.
- 因此，需要扩展参数文件来指定障碍层的内容。

```
plugins:
- {name: static_map,       type: "costmap_2d::StaticLayer"}
- {name: obstacles,        type: "costmap_2d::VoxelLayer"}
publish_frequency: 1.0
obstacles:
observation_sources: base_scan
base_scan: {data_type: LaserScan, sensor_frame: /base_laser_link,  clearing: true, markin
g: true, topic: /base_scan}
```

参考资料

- wiki.ros.org/costmap_2d/Tutorials/Configuring Layered Costmaps#Step_2:_Basic_Parameters