1. **Gmapping应用条件**

Gmapping是一个比较完善的开源地图构建包，使用激光数据和里程计数据生成二维栅格地图。

ROS中Gmapping分别订阅和发布了哪些主题（topic）？

|  |
| --- |
| entropy\_publisher\_ = private\_nh\_.advertise<std\_msgs::Float64>("entropy", 1, true);  sst\_ = node\_.advertise<nav\_msgs::OccupancyGrid>("map", 1, true);  sstm\_ = node\_.advertise<nav\_msgs::MapMetaData>("map\_metadata", 1, true);  ss\_ = node\_.advertiseService("dynamic\_map", &SlamGMapping::mapCallback, this);  scan\_filter\_sub\_ = new message\_filters::Subscriber<sensor\_msgs::LaserScan>(node\_, "scan", 5);  scan\_filter\_ = new tf::MessageFilter<sensor\_msgs::LaserScan>(\*scan\_filter\_sub\_, tf\_, odom\_frame\_, 5);  scan\_filter\_->registerCallback(boost::bind(&SlamGMapping::laserCallback, this, \_1));    transform\_thread\_ = new boost::thread(boost::bind(&SlamGMapping::publishLoop, this, transform\_publish\_period\_)) |

由上述源码可以看出：

Gmapping的订阅非常简单，只有两种：

1. tf(tf/tfMessage) 用于激光器坐标系，基座坐标系，里程计坐标系之间转换。里程计数据要转化成tf版本的里程计数据才可以使用。
2. Scan(sensor\_msgs/LaserScan) 激光器扫描数据

**scan是激光雷达数据，ros中的depthimage\_to\_laserscan包，这个包可以将kinect发布出来的深度图转换成激光雷达扫描数据。**

tf是ros中必用的部分，说简单一些，tf类中定义了两个刚体之间的旋转与平移矩阵，并且重载了乘法运算符，这样我们就可以通过相乘两个tf来沿着tf树的方向求末段执行器相对世界坐标的位置与方向因此，**gmapping需要的里程计odom便是通过我们自己发布tf树的形式告诉gmapping，**而我们该如何获得这个里程计，这就需要我们自己完成这一部分了，通常做法是在移动平台上安装电机编码器与电子罗盘，在移动平台上的嵌入式

Gmapping发布三个话题：

1. map\_metadata (nav\_msgs/MapMetaData) 地图的描述消息
2. map (nav\_msgs/OccupancyGrid) 地图数据
3. ~entropy (std\_msgs/Float64)，发布机器人姿态分布熵的估计

发布了地图的信息，比例，初始位置等。

同时会发布一个服务：

* dynamic\_map (nav\_msgs/GetMap)，调用该服务可以获取地图数据

重要参数说明:

particles (int, default: 30) gmapping算法中的粒子数，因为gmapping使用的是粒子滤波算法，粒子在不断地迭代更新，所以选取一个合适的粒子数可以让算法在保证比较准确的同时有较高的速度。

minimumScore (float, default: 0.0) 最小匹配得分，这个参数很重要，它决定了对激光的一个置信度，越高说明对激光匹配算法的要求越高，激光的匹配也越容易失败而转去使用里程计数据，而设的太低又会使地图中出现大量噪声，所以需要权衡调整。

1、slam\_gmapping

slam\_gmapping包也是依赖开源openslam\_gmapping库。下载openslam\_gmapping后编译会生成几个动态库，libgridfastslam.so，libscanmatcher.so，libsensor\_base.so，libsensor\_odometry.so，libsensor\_range.so和libutils.so。在slma\_gmapping包中编译后会生成好几个节点，不过真正需要调用的建图节点的主要实现在slam\_gmapping.cpp中，通过roslaunch slam\_gmapping slam\_gmapping.launch或者rosrun gmapping slam\_gmapping scan:=scan启动，开始扫图。

首先是初始化，载入或者配置参数。Gmapping中调用startLiveSlam函数开启建图模式，在这个函数中会发布三个话题，/enropy(机器人姿态分布熵的估计)，/map(建图过程中的数据，用占有网格数据描述，其值在0~255之间)，/map\_metadta(地图的描述消息)；会发布一个服务，/dynamic\_map，也是为了获取地图数据；会订阅激光数据和tf坐标转换后的数据，还用到了message\_filters方法，也是为了加快数据传输效率。此外，还开启了一个线程，用于不断广播map和odom之间的坐标变换。

其次的关键就是对获取的点云scan数据进行处理，这个在回调函数laserCallback中实现。一旦第一次获取到scan数据，会调用initMapper函数形成一个初始的地图形状，这个函数里面也是进行了大量tf变换，然后通过getOdomPose函数获取机器人初始位置，正常情况下该函数只会执行一次，再往后获取到的scan数据则由addScan函数处理了。addScan函数中会先得到里程计数据，然后处理scan数据。最后更新地图数据，由updateMap函数完成，并调用computePoseEntropy函数计算机器人位置分布熵，同时发布map和map\_metadata消息。

slam\_gmapping包的主要工作就这些，就是将数据处理为了在ros通讯机制下的，其核心算法还是在openslam\_gmapping里面实现的。

2、map\_server

map\_server的功能就是地图服务器。编译map\_server后会生成map\_server\_image\_loader动态库，这个用于处理pgm格式的图片，其依赖了SDL库；还会生成map\_server和map\_saver节点。

map\_server节点的源代码很容易看懂，主要就是为了加载pgm格式的地图，用yaml文件描述的，同时发布map\_metadata和map话题，以及static\_map服务，其目的都是为了方便其他节点获取到地图数据。指令为rosrun map\_server map\_server mymap.yaml。

map\_saver的作用就是为了将slam\_gmapping建图数据保存下来，**主要就是订阅map消息。rosrun map\_server map\_saver -f mymap。**

执行 rosrun map\_server map\_saver 保存地图，执行rosrun map\_server map\_server map.yaml，发布地图话题：

|  |
| --- |
| wsc@wsc-pc:~$ rostopic list  /map  /map\_metadata  /rosout  /rosout\_agg |

### **二、librviz的使用**

如果想看到建图时的实时效果可通过rviz软件查看。rviz不仅仅是一个可视化工具，还是一个库，可被其他程序所调用。加载rviz显示的关键代码如下：

|  |
| --- |
| rviz::RenderPanel \* render\_panel\_;  render\_panel\_ = new rviz::RenderPanel;  rviz::VisualizationManager \*manager\_;  manager\_ = new rviz::VisualizationManager(render\_panel\_);  render\_panel\_->initialize(manager\_->getSceneManager(), manager\_);  manager\_->initialize();  manager\_->startUpdate();  rviz::Display \*map = manager\_->createDisplay("rviz/Map", "adjustable map", true);  map->subProp("Topic")->setValue("/map"); |

首先是声明一个rviz的RenderPanel，这个面板类最终也是继承了QWidget，然后声明一个可视化管理类，继承于QObject，再初始化，VisualizationManager就可以开始不断更新了。上面代码段显示了如何加载地图，也是通过创建一个显示(createDisplay)，然后添加topic值。

**AMCL**

AMCL是机器人在二维地图移动过程中的概率定位系统。它应用自适应的蒙特卡洛定位方式（或者KLD采样），**采用粒子滤波方法来跟踪已知地图中机器人的位姿**。使用时需要在 base\_scan 话题中指定数据类型：

 AMCL以激光扫描地图，激光扫描数据，位姿变换信息为输入，输出姿态信息。因此订阅节点：

**订阅主题**

Scan（sensor\_msgs / LaserScan）

激光扫描。

tf（tf / tfMessage）

变换。

initialpose（geometry\_msgs / PoseWithCovarianceStamped）

用于（重新）初始化粒子滤波器的均值和协方差。

Map（nav\_msgs / OccupancyGrid）

当use\_map\_topic参数设置，AMCL赞同这一主题，以获取用于基于激光的定位地图。新导航1.4.2。

**发布主题**

amcl\_pose（geometry\_msgs / PoseWithCovarianceStamped）

机器人在地图中的估计姿势，具有协方差。

particlecloud（geometry\_msgs / PoseArray）

姿势估计集由过滤器维护。

tf（tf / tfMessage）

发布从变换奥多姆（其可以经由〜odom\_frame\_id参数被重新映射），以映射。。

**服务**

global\_localization（std\_srvs / Empty）

启动全局定位，其中所有粒子通过地图中的自由空间随机分散。

request\_nomotion\_update（std\_srvs / Empty）

用于手动执行更新和发布更新粒子的服务。

set\_map（nav\_msgs / SetMap）

用于手动设置新地图和姿势的服务。

**被呼叫的服务**

static\_map（nav\_msgs / GetMap）

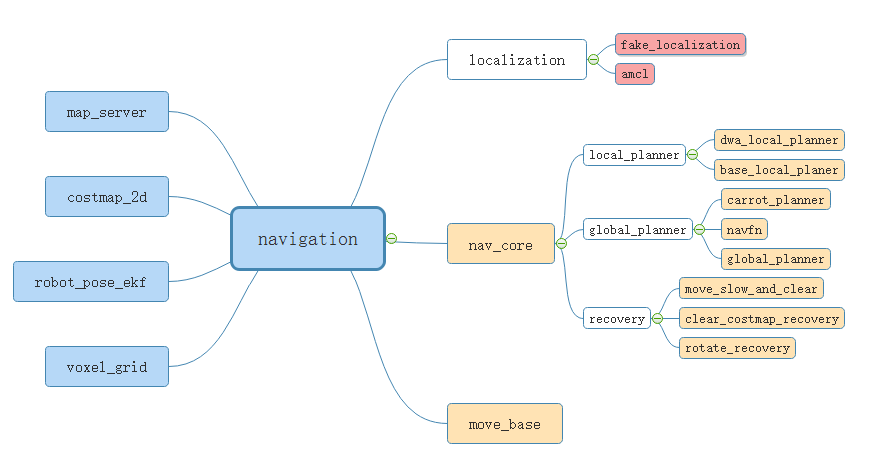
amcl调用此服务来检索用于基于激光的本地化的地图; 启动阻止从此服务获取地图。

# ROS navigation分析：navigation框架

 ROS navigation stack是ROS提供的一个非常重要且常用的模块。它的主要作用是实现机器人的定位、导航和避障功能。**navigation需要的输入是里程计信息（必选）、传感器信息（比如超声、红外、深度相机、激光等等），地图。输出是可以控制机器人运动的一系列速度指令。**

代码包括：





梳理完之后，可以简单把navigation stack分成7个部分。

map\_server：顾名思义，地图服务器，主要功能是保存地图和导入地图。

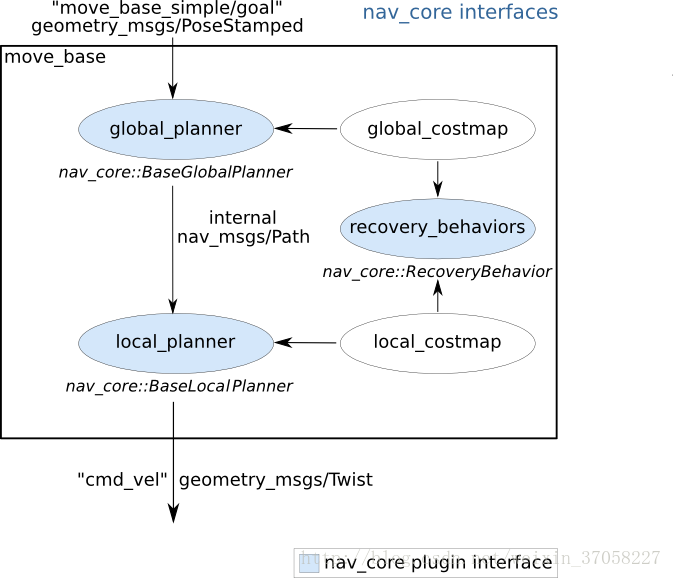
costmap\_2d：可以生产代价地图，以及提供各种相关的函数。

robot\_pose\_ekf：扩展卡尔曼滤波器，输入信息是里程计、IMU、VO中的任意两个或者三个都输入。输出是一个融合之后的pose，还是蛮好用的。

voxel\_grid：这个嘛，wiki是上这么写的，“The voxel\_grid has no API that we're supporting officially. You should still feel free to use the code, just know that we're making no guarantees about the stability of the current API”。这个package就随缘吧。

localization：这里是两个定位用的package。fake\_localization一般是仿真用的，amcl才是实际定位用的package。

nav\_core：是一个纯粹的接口package，里面规定了核心的几个类应该有基本功能。这里面只有三个文件，对应的是全局路径规划、局部路径规划、recovery\_action的通用接口定义，具体功能实现则是在各个对应的规划器package里。nav\_core包包含了导航功能包的关键接口。



BaseGlobalPlanner是全局导航的接口，规定一个功能函数makePlan，给定起始跟目标，输出路径(一系列pose)走过去；BaseLocalPlanner规定了一个核心函数computeVelocityCommands，就是计算局部地图内的下一步控制指令(线速度，角速度)；还有一个RecoveryBehavior，规定一个runBehavior，在小车卡住情况下执行运动恢复，回到正常的导航状态。

nav\_core::BaseGlobalPlanner支持了供导航中全局规划器使用的接口。 move\_base node中使用的所有全局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：

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")

navfn - 一个基于grid的全局规划器，使用navigation函数计算机器人路径global planner that uses a navigation function to compute a path for a robot. (pluginlib name: "navfn/NavfnROS")

carrot\_planner - 一个简单的全局规划器，移动机器人到用户定义的目标点，即使目标在障碍中也会靠近目标。 (pluginlib name: "carrot\_planner/CarrotPlanner")

nav\_core::BaseLocalPlanner给导航中使用的具体规划器提供接口。 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来做本地规划控制

eband\_local\_planner - 实现了Elastic Band method on the SE2 manifold

teb\_local\_planner - 实现了Timed-Elastic-Band method for online trajectory optimization

 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

2.globalplanner

这里要明确一点的是，全局规划器globalplanner计算起点终点路径，局部规划器localplanner根据之前生成的路径，计算移动机器人的运动速度。

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。

navigation包里面globalplanner用了dijkstra和A\*两种算法，任选一种都可以。这一点ros wiki上面可以看到他们的具体差别。

另外还有一些其他的算法可以使用的，包括D\*算法，在网上可以找到其他人的实现。

这里还有一个点直接注意的是，在navigation包里面move\_base的实现里面，是添加了动态规划的，也就是在小车的移动过程中，globalplanner也是有个线程一直在计算当前位置到终点的路径。D\*这样的算法可以实现动态规划，可是navigation并没有采用这种方法，而且启动了一个线程，在移动过程中对路径不断的进行重新规划。不过这种方法有个优点，在不妨碍原有实现上(也就是dijkstra和A\* 寻路上) 添加了动态规划，**global\_planner是navfn的升级版替代者。**

3.localplanner

提供了两种实现，一是航迹推算法（TrajectoryROS），一是动态窗口法(DWA)，

对于全向机器人来说，也就是存在x方向的速度，y方向的速度，和角速度。DWA确实效率高一点。但是如果是非全向机器人，比如说只存在角速度和线速度。trajectorylocalplanner会更适用一点。 base\_local\_planner这个包通过地图数据，通过算法搜索到达目标的多条路经，利用一些评价标准（是否会撞击障碍物，所需要的时间等等）选取最优的路径，并且计算所需要的实时速度和角度。

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。

move\_base：这里实现的是整个导航的流程。什么时候调用全局路径规划、什么时候调用局部路径规划、什么时候调用recovery\_action都是这个package管的。就是下图中间方框里做的事情，可以说是整个navigation stack的核心。它的功能是给定一个目标点，尝试产生一系列能让机器人到达这个目标点的控制指令。简单来说，就是输入是一个goal(geometry\_msgs/PoseStamped)，输出是一系列command(geometry\_msgs/Twist)。详细一点来说，就是在已知环境地图和机器人在地图中位置的情况下，利用地图和传感器数据处理得到的 global costmap 和 local costmap，调用全局路径规划器得到一条全局路径，调用局部路径规划器使机器人在移动过程中能在大致沿着路径前进并动态规避障碍物，整个行进过程中还要handle一部分异常情况即recovery\_behaviors。

