# move\_base代码流程：

1.创建服务 action server

as\_=new MoveBaseActionServer(ros::NodeHandle(), "move\_base", boost::bind(&MoveBase::executeCb, this, \_1), false);

2.加载参数

private\_nh.param("base\_global\_planner", global\_planner, std::string("navfn/NavfnROS"));

3.建立3个plan buffer

//set up plan triple buffer

planner\_plan\_=new std::vector<geometry\_msgs::PoseStamped>();

latest\_plan\_=new std::vector<geometry\_msgs::PoseStamped>();

controller\_plan\_=new std::vector<geometry\_msgs::PoseStamped>();

4.创建线程

planner\_thread\_ = new boost::thread(boost::bind(&MoveBase::planThread, this));

5.发布和订阅topic

advertise cmd\_vel,current\_goal ,move\_base/goal

subcribe move\_base\_simple/goal("move\_base\_simple/goal" geometry\_msgs/PoseStamped)

6.创建并初始化global costmap 和global planner

7.创建并初始化local costmap 和local planner

1. 启动 costmap

9.发布server

make\_plan,clear\_costmap

10.启动 ACTION SERVER。

11. 动态配置服务

dynamic\_reconfigure 回调函数 ： reconfigureCB

Move base 初始化流程图：

Move\_base初始化

订阅move\_base\_simple/goal

创建线程planner\_thread\_

创建服务 action server

创建并初始化global costmap 和global planner

创建并初始化local costmap 和local planner

make\_plan,clear\_costmap

启动 ACTION SERVER

控制流程：

move\_base\_simple/goal topic

# Move\_base

goalCB

Action server

executeCb

Wake up

planner\_thread\_

executeCycle

makeplan

# Global planner

Make\_plan

# Local planner

computeVelocityCommands

vel\_pub\_**.**publish**(**cmd\_vel**);**

**Costmap\_2d\_ros**

1. Costmap\_2d\_ros 初始化：

找到 global\_frame 和 robot\_base\_frame 并等待有效。

1. 创建 LayeredCostmap

resetOldParameters ：初始化plugins 的参数.

Subcribe footprint\_topic / footprint topic.

发布footprint topic.

makeFootprintFromParams：解析文件，创建footprint 的points

1. 创建 Costmap2DPublisher

Costmap2DPublisher发布两个topic costmap 和 costmap\_updates

创建timer检查机器人位置：

private\_nh**.**createTimer**(**ros**::**Duration**(**.1**),** **&**Costmap2DROS**::**movementCB**,** **this);**

Costmap2DROS**::**start**() 中依次激活plugins：**

**(\***plugin**)->**activate**();**

StaticLayer**::**onInitialize**()：订阅 map 和map\_updates topic**

Layer: updateBounds 根据机器人的位置，输出一个世界坐标系的范围。代表costmap的区域？

Layer: updateCosts 更新 某个 区域的cost值，到传入的master\_grid中。

# Inflation layer的算法:

1. costmap的范围做了两次膨胀，第一次是覆盖到膨胀范围内的障碍物，第二次是覆盖外部障碍物的膨胀。

原始范围

初始化两个cache，减小运行时的计算消耗：

dx,dy的对应的距离数组(计算直角三角行斜边长度)：

cached\_distances\_**[**i**][**j**]** **=** hypot**(**i**,** j**);**

**距离对应的cost值(计算e的指数函数)：**

cached\_costs\_**[**i**][**j**]** **=** computeCost**(**cached\_distances\_**[**i**][**j**]);**

**2. 创建一个** inflation\_cells\_，该表格存放如下内容:index，当前cell的x,y，相邻障碍物所在cell的x,y。

**从障碍物所在的cell开始，依次将障碍物周边的cell加入到**inflation\_cells\_[distance]中.

在enqueue函数中会过滤distance > cell\_inflation\_radius\_ 的cell。

然后再遍历 inflation\_cells\_[]中每个distance中的每个cell，调用costLookup 后与原来的cost做比较，取较大值.

Abstacle layer的算法:

订阅配置文件中所有的topic（"PointCloud2"，"PointCloud"， "LaserScan"），分别注册这些topic的回调函数（laserScanCallback，laserScanValidInfCallback，pointCloud2Callback，pointCloudCallback）。

这些topic的msg最终都会转成sensor\_msgs::PointCloud2 类型，存放到buffer->bufferCloud(cloud); 中。

ObservationBuffer ：

将sensor\_msgs**::**PointCloud2转成 pcl**::**PointCloud**<**pcl**::**PointXYZ**>**，再将这些点tf到global的frame下,然后遍历每一个point，过滤高度异常的点：max\_obstacle\_height\_，min\_obstacle\_height\_。

过滤后的点存入observation\_list\_.front()中（front是函数开头新创建的Observation()，再push到observation\_list\_中）：observation\_list\_.push\_front(Observation());

updateBounds：

函数中遍历observation\_list\_ 中的cloudy数据，剔除超出 obstacle\_range\_和max\_obstacle\_height\_ 的point,更新costmap的值：costmap\_**[**index**]** **=** LETHAL\_OBSTACLE**;**

然后调整minx,miny,maxx,maxy的范围，使其包含该point。

除了pcd之外，还需要包含Footprint的范围：updateFootprint**(**robot\_x**,** robot\_y**,** robot\_yaw**,** min\_x**,** min\_y**,** max\_x**,** max\_y**);**

最终返回一个包含所有有效点和foo他print的范围.

updateCosts：

更新costmap中的值到 master\_grid中，有两种方式：

**case** 0**:**  *// Overwrite*  updateWithOverwrite**(**master\_grid**,** min\_i**,** min\_j**,** max\_i**,** max\_j**);**  
  **break;**  
**case** 1**:**  *// Maximum*

\_\_default\_\_**.**neutral\_cost **=** 50**;**

**//这个值代表path中每增加一个cell的路程代价值增加50。**

Globalplan流程

makePlanService ->makePlan->calculatePotentials->add

->QuadraticCalculator**::**calculatePotential

->getPlanFromPotential

→orientation\_filter\_→processPath //计算每个点的角度

→publishPlan(plan);

**astar**算法**:**

**1. calculatePotential**

从起始点开始向外扩展，计算每一个点的代价值，一直扩展到**goal**点结束。

2. getPlanFromPotential

从结束点开始向4个方向找代价值最小的点，一直找到start点结束。这些点的集合就构成了一个path。