**ROS Navigation**

#### ROS导航功能包集概述（navigation packge）

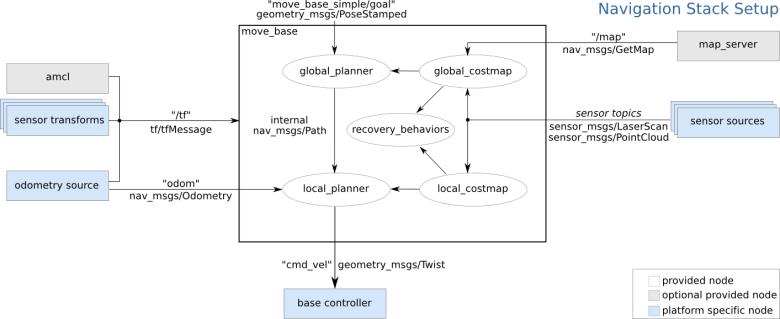
机器人在未知环境中需要使用激光传感器（或者是深度传感器）进行地图建模，然后根据构建的地图进行导航与定位，在ROS中可以利用以下三个功能包集实现自主导航。

Gmapping:根据激光数据（或者是深度数据模拟的激光数据）构建地图

Moves\_base:根据参照消息进行路径规划，使机器人到达指定位置

Amcl:根据已有地图进行定位

整体导航包框架如图所示：



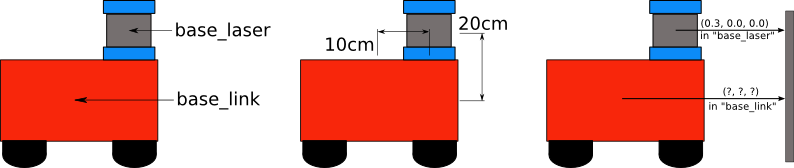
其中白色框内的是ROS已经为我们准备好的必须使用的组件，灰色框内的是ROS中可选的组件，蓝色的是用户需要提供的机器人平台上的组件。

另外，我们可以清楚得看见navigation的输入：里程计odometry， 激光雷达或者rgbd-camera的信息sensor\_topics，还有已知的先验地图（map\_server）（可选），坐标系变换信息，输出就是cmd\_vel速度。

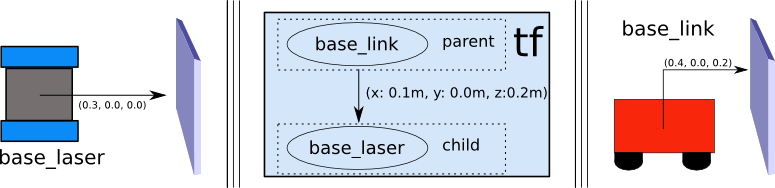
下面对这张图的每一部分进行逐一介绍：

1. 传感器坐标转换sensor transform（tf）

因为我们使用的机器人的控制中心不一定是在传感器上，所以要把传感器的数据转换成在控制中心上的坐标信息。如下图所示，传感器获取的数据是在base\_laser的坐标系统中的，但是我们控制的时候是以base\_link为中心，所以要根据两者的位置关系进行坐标的变换。



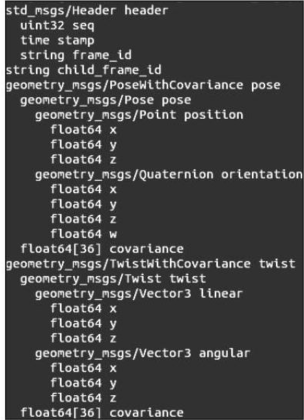
变换的过程不需要我们自己处理，只需要将base\_laser和base\_link两者之间的位置关系告诉tf，就可以自动转换了。

（2）导航包传感器数据输入

这里是机器人导航传感器数据输入，一般只有两种：  
1）激光传感器数据 Laserscan  
2）点云数据 Cloudpoint

（3）里程信息——odometry source

里程信息描述的是机器人相对某一点的距离，它的消息类型如下图：



如图所示，nav\_msgs/Odometry提供了机器人从frame\_id坐标系到child\_frame\_id坐标系的相对位置，它还通过geometry\_msgs/Pose消息提供机器人位姿信息，geometry\_msgs/Twists消息表示速度信息。位姿信息中包含着两个数据结构，一个表示欧拉坐标系中的位置，另一个则使用四元向量描述机器人的方向，也就是机器人的角位移。速度信息包含了线速度和角速度。

(4)基础控制器——base controller

对于导航功能包来说，基础控制器是非常重要的，因为这是唯一能有效控制机器人的方法。它能够直接和机器人的电子设备进行通信。

ROS并不提供任何标准的基础控制器，因此必须自己编写针对特定机器人平台的基础控制器。base controller必须订阅cmd\_vel主题，并将该部分负责将之前得出来的数据转封装成具体的线速度和转向角度信息（Twist），并且发布给硬件平台。

(5)地图服务——map\_server

在导航过程中，地图并不是必须的，但是此时相当于是在一个无限大的空地上进行导航，并没有任何障碍物。考虑到实际情况，在我们使用导航的过程中还是需要一个地图。

(6)move\_base

它实现了一个供上层节点调用的action 接口（通过actionlib 实现），即给定坐标信息与相应坐标位置，执行相应的安全导航动作。对于下层控制器，之前说过了输出为cmd\_vel 2d速度。 它规定了这个navigation的行为。

从软件架构角度来讲move\_base是navigation的逻辑核心存在。

从移动机器人体系结构来说，move\_base规定了整个规划层的行为流程。 而如果要配置ROS navigation，重点就是move\_base 与它组件的配置。

行为层： move\_base 综合机器人状态与上层指令，给出机器人当前行为：正常导航，执行恢复动作，给上层节点返回失败，终止导航。其中恢复动作可以自己定义。

全局规划层：global\_planner

局部规划层： local\_planner

控制器层（一般就是之前自己写的速度发送部分）

costmap\_2d ： 这一部分可看作为navigation的输入处理器。不同的传感器输入的数据差异很大（激光雷达 & RGBD-camera）通过costmap\_2d，不同的数据被处理成统一的格式：栅格地图，权值用经过概率方法处理过的，表示空间中障碍物，未知与安全区域。生成出来的costmap则是planner的输入。

global\_planner ： 为navigation的全局规划器，接受costmap生成的 global costmap 规划出从起始点到目标点的路径，为local\_planner 作出参考。

local\_planner ： 为navigation 的局部规划器，接受costmap 生成的local costmap 规划出速度。

recovery\_behavior : 规定move\_base 行为集合中处理异常情况的行为

这是主体部分。 要理解ROS navigation 最重要的部分是nav\_core: 这个包里面就包含了global\_planner ， local\_planner 与 recovery\_behavior的基类的头文件。但是极其重要。

在ROS navigation中， move\_base 提供的是框架，在move\_base 中是通过nav\_core 中规定的planner 与 recovery\_behavior 的基类的接口进行调用。与具体的实现方法隔离开来。而具体采用的方法由pluginlib 根据不同参数导入。这样的实现方法使得navigation的可定制性大大增加。像base\_local\_planner 中就实现了两种局部路径规划方法，global\_planner 实现了A\* 与Dijkstra 两种方法，在navigation\_experimental 中还有更多这样的实现。这赋予了这个框架很大的灵活性。通过不同的配置方法可以让navigation适应很多不同的任务。

#### 2.costmap讲解

在讲解costmap之前先介绍一下costmap的基本概念，只要大家把基本概念弄清楚之后，才能根据自己的实际应用环境对自己的代码做出修改。

##### 2.1 基本概念

Voxel：体素，即顾名思义是体积的像素。用来在[三维空间](https://baike.baidu.com/item/%E4%B8%89%E7%BB%B4%E7%A9%BA%E9%97%B4)中表示一个显示基本点的单位。类似于二维平面下的[pixel](https://baike.baidu.com/item/pixel)（[像素](https://baike.baidu.com/item/%E5%83%8F%E7%B4%A0)）。voxel是三维空间中定义一个点的图象信息的单位。在平面中定义一个点要两个坐标X和Y就够了，而在三维世界中还要有一个[坐标](https://baike.baidu.com/item/%E5%9D%90%E6%A0%87/85345)。光有[3维坐标位置](https://baike.baidu.com/item/%E4%BD%8D%E7%BD%AE/10820618)还不行，还要有[颜色](https://baike.baidu.com/item/%E9%A2%9C%E8%89%B2/3916197)等信息，这就是voxel的含义。

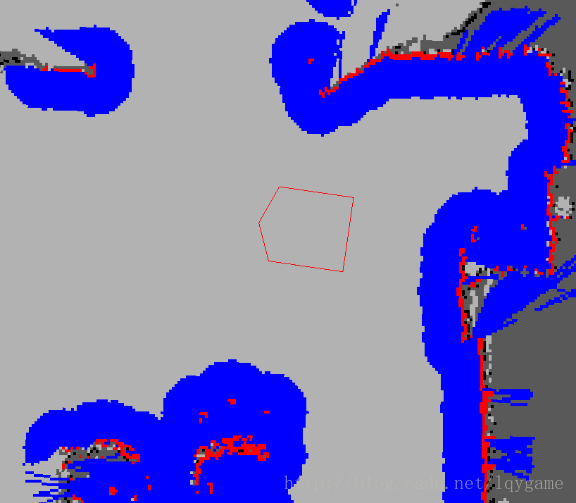
机器人在costmap\_2D中的模型：两个同心圆，在图1里可以看到左下角有两个淡蓝色同心圆，一个机器人的轮廓外切圆和一个机器人内切圆，机器人在costmap里就能够简化成为这两个圆。根据机器人中心至边界或者障碍物的距离和两个同心圆半径比较来判断是否碰撞。

footprint：足迹，即机器人的轮廓。在ROS中，它由二维数组表示[x0,y0] ; [x1,y1] ; [x2,y2]……不需要重复第一个坐标。该占位面积将用于计算内切圆和外接圆的半径，用于以适合此机器人的方式对障碍物进行膨胀。为了安全起见，我们通常将足迹稍大于机器人的实际轮廓。要确定机器人的占地面积，最直接的方法是参考机器人的图纸。 此外，您可以手动拍摄其基座顶视图。 然后使用CAD软件（如Solidworks）适当缩放图像，并将鼠标移动到基座轮廓上并读取其坐标。 坐标的起点应该是机器人的中心。 或者，您可以将机器人移动到一张大纸上，然后绘制基座的轮廓。 然后选择一些顶点并使用标尺来确定它们的坐标。

cost：代价或者占用，0-255的取值，表示该机器人位于该网格点(grid cell)的代价，或者机器人的frootprint中心cell走到该网格点的代价（中心到达某个位置的代价与非中心部分到达某个位置付出的代价不同，如撞击造成的损伤程度等）。

##### 2.2 costmap网络代价计算

无论是激光雷达还是如kinect 或xtion pro深度相机作为传感器跑出的2D或3D SLAM地图，都不能直接用于实际的导航，必须将地图转化为costmap(代价地图),ROS的costmap通常采用grid(网格)形式。Costmap是机器人收集传感器信息建立和更新的二维或三维地图，可以从下图简要了解。



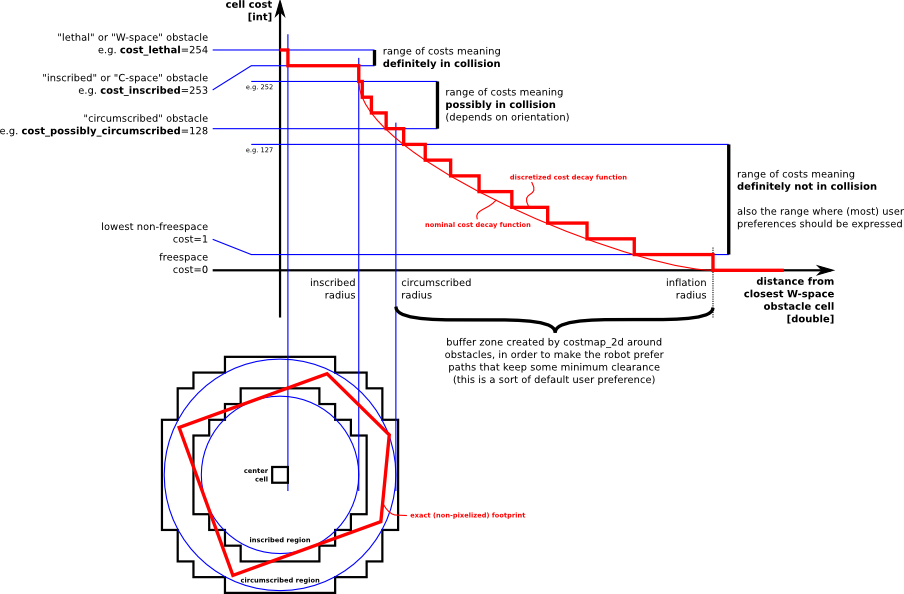
机器人和障碍物以及膨胀区域在2维地图上的footprint和投影表示

红色cell（图中红色蓝色区域都是一系列cell堆叠出来的）代表的是代价地图中的障碍，蓝色cell代表的是通过机器人内切圆半径计算的障碍物膨胀，红色多边形代表的是机器人footprint(机器人轮廓的垂直投影)。 为了使机器人不碰到障碍物，机器人的footprint绝对不允许与红色cell相交，机器人的中心绝对不允许与蓝色cell相交。

栅格地图一个栅格占1个字节，也就是八位，可以存0-255中数据，也就是每个cell cost（网格的值）从0-255我们只需要三种情况：Occupied被占用（有障碍）, Free自由区域（无障碍）,  Unknown Space未知区域。。

当投影到代价地图时候，每种状态被赋一个特定的代价值，也就是说每个cell的cost值是由这个cell对应的各层中对应的cell的状态进行加权得到的。 如果列有一定量的占用就被赋代价值costmap\_2d::LETHAL\_OBSTACLE， 如果列有一定量的unknown cells 就被赋代价值costmap\_2d::NO\_INFORMATION, 剩余其它列赋代价值为costmap\_2d::FREE\_SPACE。

在某时刻和机器人当前前进方向上的网格点的代价计算示意图如下（如果机器人的前进方向改变，则网格点的代价也会发生变化）：



lethal：值为254，传感器扫描到的实际障碍转化成map中的lethal网格。

inscribed ：内部半径，如果网格和lethal网格距离小于等于该半径，则为inscribed网格，值为253。取robot几何型所能容纳最大的圆的半径。

被标记为253～254的网格是障碍网格，也就是lethal和inscribed网格是障碍网格。当robot的边界覆盖到这两种网格时，一定会发生碰撞。

possibly circumscribed：膨胀半径，以robot的中心为园点旋转一周所能覆盖的最大圆半径，如果所在网格距离lethal小于等于膨胀半径，大于内部半径，则为possibly circumscribed网格。值为128～252，取决与robot的运动方向和轨迹，还有算法。当robot的边界覆盖到这种网格时，可能会发生碰撞。

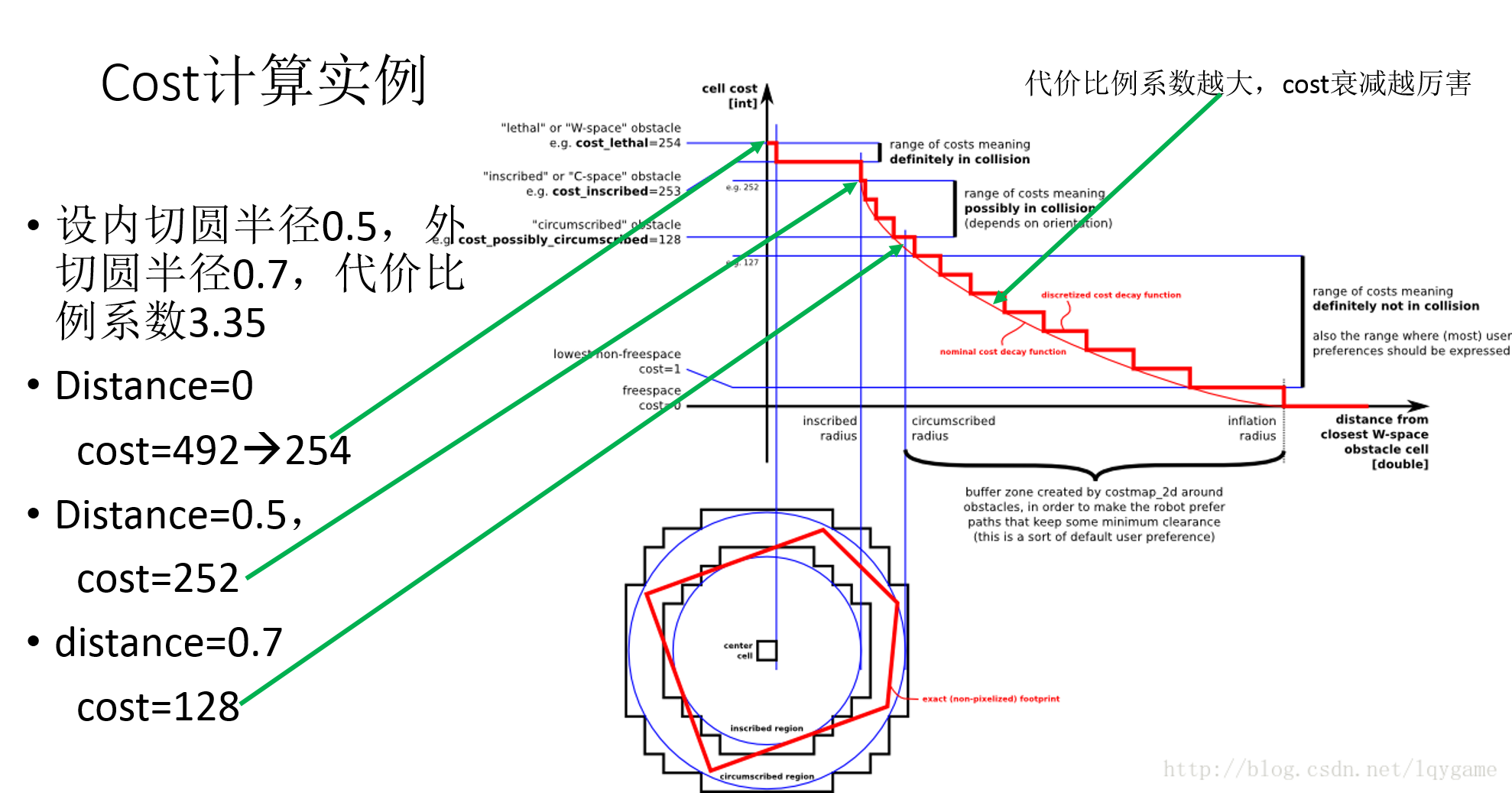
free：值为0,说明没有任何信息可以阻碍机器人运行到这里。

unkonw：没有任何信息，传感器还没有扫描到该区域，都是障碍网格

1～127：不会发生碰撞的区域，这里的值赋值取决于与lethal的距离，还有用户自定义的算法

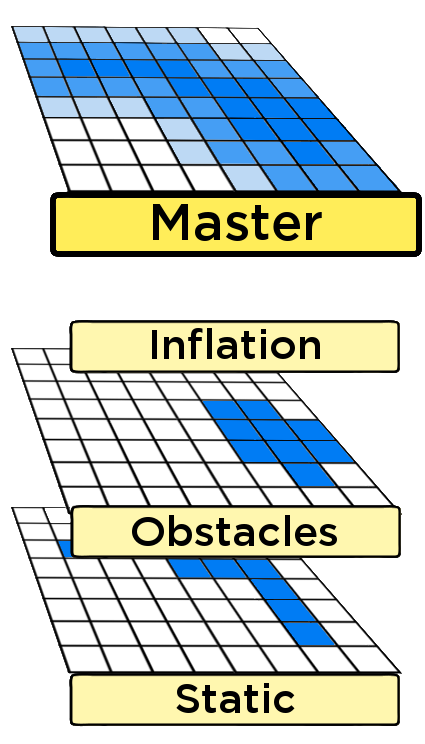
PS：这里介绍一下costmap\_2d中计算cost的方法。假设，机器人内切半径为0.5m，外切半径为0.7m，当激光返回障碍距离在机器人中心附近，叫致命障碍，机器人一定能碰到障碍物，比如说0m,直接贴着机器人，或会取小于栅格的边长，比如小于0.1m范围内,则这个栅格值就设为254；当返回来的数值在0.1-0.5m之间，就设253；当在0.5-0.7之间，则可以设128，或者在252-128找个比例值（程序中可以控制），属于受限区域，可能发生碰撞，是否碰撞，取决于机器人的姿态；当0.7-膨胀半径之间，设1-127之间的映射值，不会发生碰撞；当大于膨胀距离，则设为0，称为freespace。Unknown -- 意味着给定的单元没有相应的信息。我们看坐标系中较细的红色光滑曲线就是cost曲线，x是距离机器人footprint的圆心距离，而y是cost值，cost随着x的增大而减小距离，当x>=内切圆半径时开始有值；当x=0时，y=254；当x=resolution/2时，cost=253；（图中右上角的较粗的台阶状红线是单元格的边线，或者认为是障碍物（单元格化后）的边线）

20180308161918966



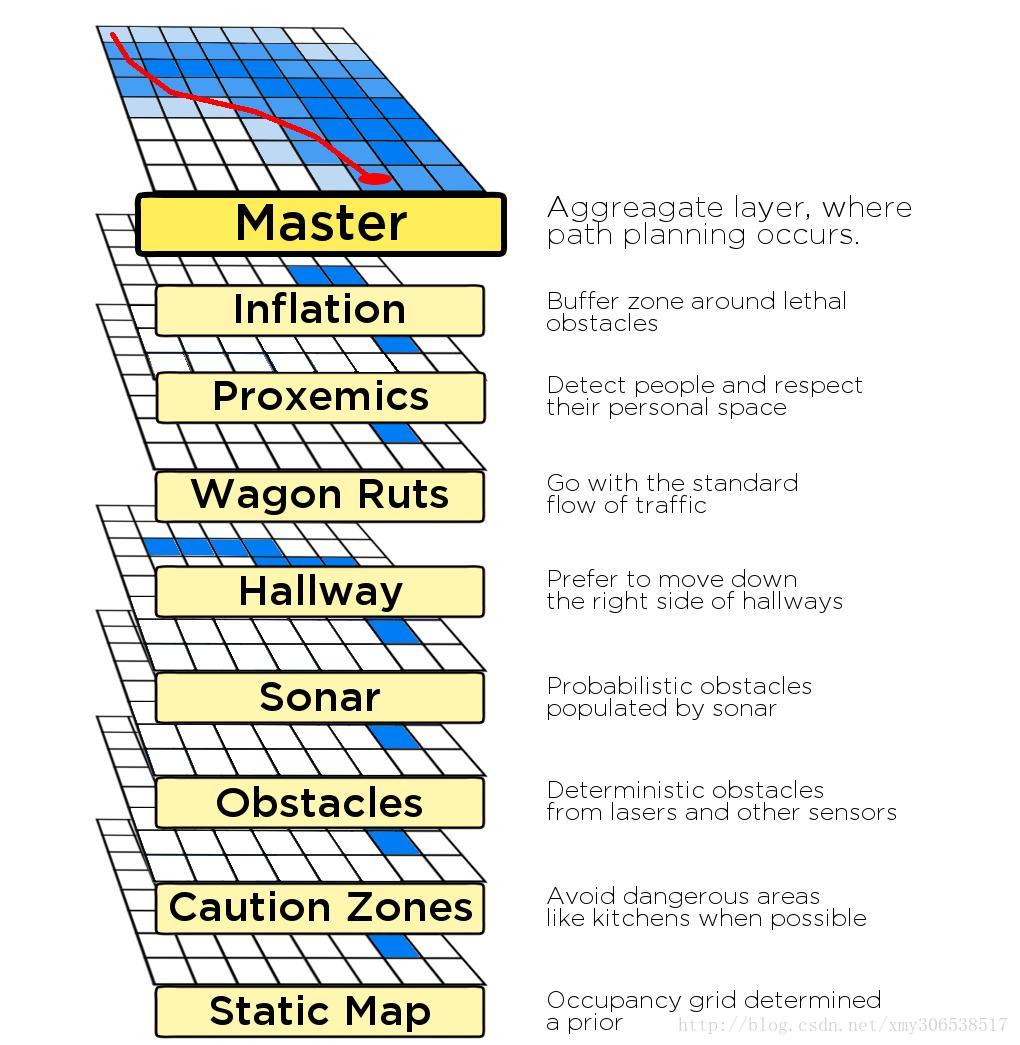
##### 2.3 costmap分层与更新

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。 Costmap由多层组成，每种功能放置一层中。 例如图3所示，静态地图是一层，障碍物是另一层。 缺省情况下，障碍物层维护的是3D信息，3D障碍物数据可以让层更加灵活的标记和清除障碍物。例如在costmap\_2d包中，StaticLayer（静态地图层）是第一层，ObstacleLayer（障碍物层）是第二层，InflationLayer（膨胀层）是第三层。这三层组合成了master map（最终的costmap），供给路线规划模块使用。



costmap\_2D中的4个分层（从Hydro版本之后采用这种分层结构）

自己定义的障碍物也可以是一层（假如我不想让机器人通过一个freespace就可以自己插入个障碍物，主要的接口是costmap\_2d::Costmap2DROS，在每一层中使用pluginlib实例化Costmap2DROS并将每一层添加到LayeredCostmap），各个层可以被独立的编译。如下图所示：



**costmap\_2d**包提供了一种可配置框架来维护机器人在代价地图上应该如何导航的信息。 代价地图使用来自传感器的数据和来自静态地图中的信息，通过**costmap\_2d::Costmap2DROS**来存储和更新现实世界中障碍物信息。**costmap\_2d::Costmap2DROS**给用户提供了纯2D的接口，这意味着查询障碍只能在列上进行。例如，在XY平面上位于同一位置的桌子和鞋，虽然在Z方向上有差异但是它们在**costmap\_2d::Costmap2DROS**对象代价地图中对应的cell上拥有相同的代价值。 这种设计对平面空间进行路径规划是有帮助的。

costmap\_2D提供的ROS化功能接口主要就是**costmap\_2d::Costmap2DROS**，它使用**costmap\_2d::LayeredCostmap** 来跟踪每一层。 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，即LayerdCostmap为Costmap2DROS（用户接口）提供了加载地图层的插件机制，每个插件（即地图层）都是Layer类型的。**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。

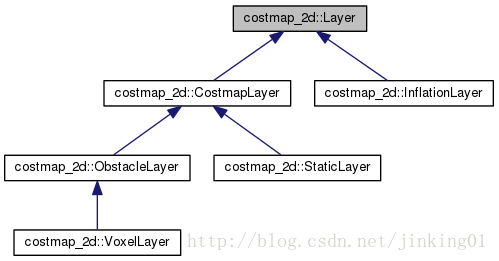


图4. costmap中的Layer的继承关系

costmap中各Layer之间的继承关系如图4所示，本文后面我们附的layeredcostmap的相关介绍还会用到这个图。

##### 2.4 costmap初始化流程

在navigation的主节点move\_base中(costmap隶属于navigation包，或者说是navigation的一个子模块)，建立了两个costmap。其中planner\_costmap\_ros\_是用于全局导航的地图，controller\_costmap\_ros\_是用于局部导航用的地图。图5为costmap的初始化流程。

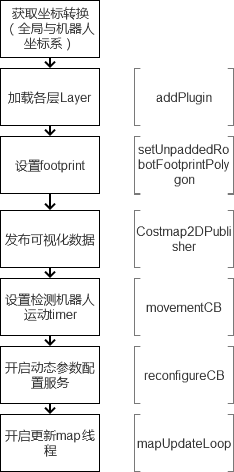
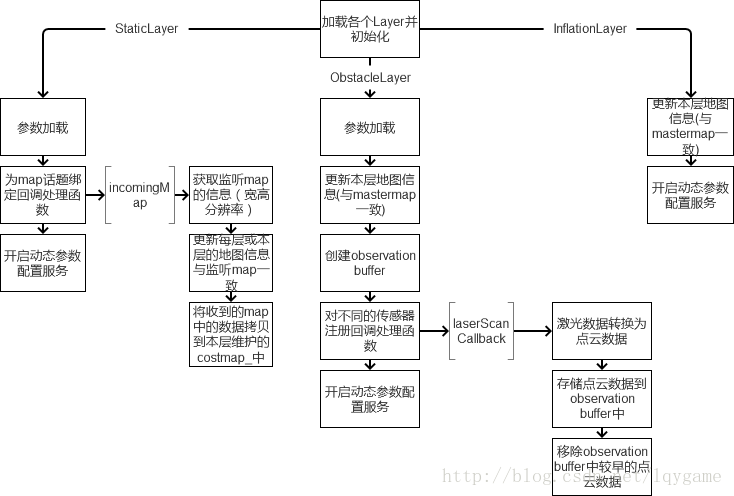


图5. costmap的初始化流程

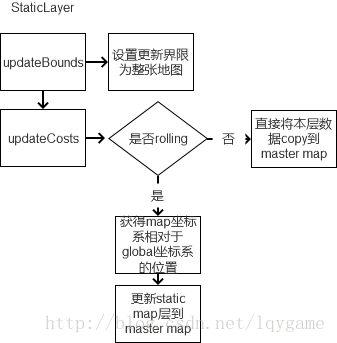
（1）Costmap初始化首先获得全局坐标系和机器人坐标系的转换   
（2）加载各个Layer，例如StaticLayer，ObstacleLayer，InflationLayer。   
（3）设置机器人的轮廓   
（4）实例化了一个Costmap2DPublisher来发布可视化数据。   
（5）通过一个movementCB函数不断检测机器人是否在运动   
（6）开启动态参数配置服务，服务启动了更新map的线程。

##### 2.5 costmap中各层的更新

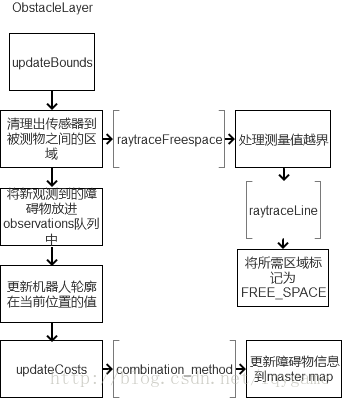
costmap初始化过程中的各层加载的调用过程：



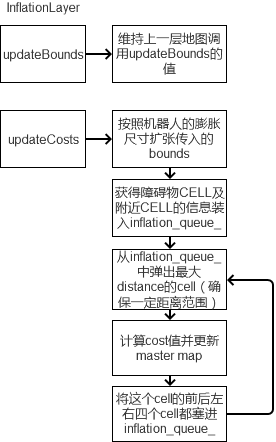
在move\_base刚启动时就建立了两个costmap，而这两个costmap都加载了三个Layer插件，它们的初始化过程如上图所示。

StaticLayer主要为处理gmapping或者amcl等产生的静态地图。ObstacLayer主要处理机器人移动过程中产生的障碍物信息。 InflationLayer主要处理机器人导航地图上的障碍物信息膨胀（让地图上的障碍物比实际障碍物的大小更大一些），尽可能使机器人更安全的移动。   
costmap在mapUpdateLoop线程中执行更新地图的操作，每个层的工作流程如下：   
(1)StaticLayer工作流程   


上图是StaticLayer的工作流程，updateBounds阶段将更新的界限设置为整张地图，updateCosts阶段根据rolling参数（是否采用滚动窗口）设置的值，如果是，那静态地图会随着机器人移动而移动，则首先要获取静态地图坐标系到全局坐标系的转换，再更新静态地图层到master map里。   
(2)ObstacleLayer工作流程



上图是ObstacleLayer的工作流程，updateBounds阶段将获取传感器传来的障碍物信息经过处理后放入一个观察队列中，updateCosts阶段则将障碍物的信息更新到master map。   
(3)inflationLayer工作流程



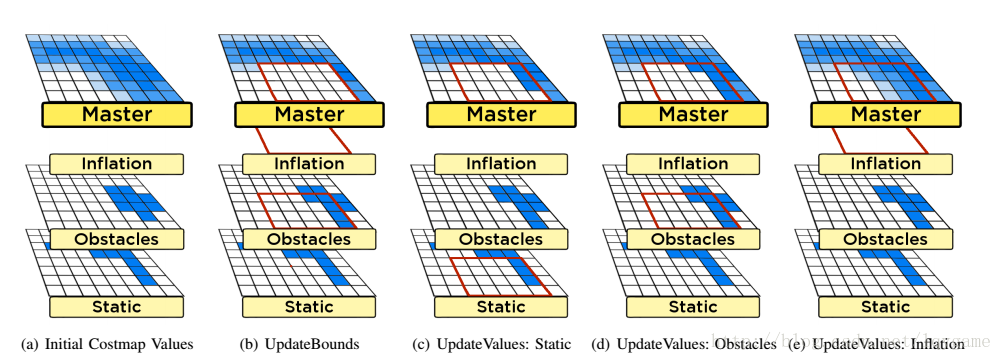
上图是inflationLayer的工作流程，updateBounds阶段由于本层没有维护的map，所以维持上一层地图调用的Bounds值（处理区域）。updateCosts阶段用了一个CellData结构存储master map中每个grid点的信息，其中包括这个点的二维索引和这个点附近最近的障碍物的二维索引。改变每个障碍物CELL附近前后左右四个CELL的cost值，更新到master map就完成了障碍物的膨胀。

##### 2.6 costmap更新

前面说过，“Costmap2DROS使用costmap\_2d::LayeredCostmap 来跟踪每一层”。这里我们要清楚每一层有什么操作。

代价地图自动订阅传感器发布的主题并基于数据进行相应自我更新。 对每个传感器来说，其可以用来执行mark（将障碍物信息插入到代价地图），也可用来执行clear（从代价地图移除障碍物）或者二者都执行。marking操作就是索引到数组内修改cell的代价。然而对于clearing操作，每次观测报告都需要传感器源向外发射线，由射线穿过的珊格组成（是不是clear要清除该cell对应的诸多层中对应cell的状态）。 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。

costmap以[参数update\_frequency](http://wiki.ros.org/costmap_2d#Rate_parameters) 指定的周期进行costmap更新。每个周期传感器数据进来后，都要在代价地图底层占用结构上执行标记和清除障碍操作，并且这种结构会被投影到代价地图附上相应代价值。 这完成之后，对代价赋值为costmap\_2d::LETHAL\_OBSTACLE的每个cell执行障碍物的膨胀操作，即从每个代价cell向外传播代价值，直到用户定义的膨胀半径为止。这里确实只需要对状态为LETHAL\_OBS TACLE的cell进行膨胀操作即可。

Costmap的更新在mapUpdateLoop线程中实现，此线程分为两个阶段：   
（阶段一）UpdateBounds：这个阶段会更新每个Layer的更新区域，这样在每个运行周期内减少了数据拷贝的操作时间。StaticLayer的Static map只在第一次做更新，Bounds 范围是整张Map的大小，而且在UpdateBounds过程中没有对Static Map层的数据做过任何的更新）。ObstacleLayer在这个阶段主要的操作是更新Obstacles Map层的数据，然后更新Bounds）。InflationLayer则保持上一次的Bounds。   
（阶段二）UpdateCosts：这个阶段将各层数据逐一拷贝到Master Map，可以通过下图观察Master Map的生成流程。（图来源于David Lu的《Layered Costmaps for Context-Sensitive Navigation》）   


在（a）中，初始有三个Layer和Master costmap,Static Layer和Obstacles Layer维护它们自己的地图，而inflation Layer并没有。为了更新costmap,算法首先在各层上调用自己的UpdateBounds方法（b）。为了决定新的bounds,Obstacles Layer利用新的传感器数据更新它的costmap。然后每个层轮流用UpdateCosts方法更新Master costmap的某个区域,从Static Layer开始（c），然后是Obstacles Layer(d)，最后是inflation Layer(e)。

（参考David Lu的《Layered Costmaps for Context-Sensitive Navigation》<http://download.csdn.net/download/jinking01/10272584>）

# 4. LayeredCostmap代码分析(实训讲解） 在数据成员中，有两个重要的变量：Costmap2D costmap\_; 和 std::vector<boost::shared\_ptr<Layer> > plugins\_; 。  这个类相对比较简单，首先来看构造函数：

LayeredCostmap::LayeredCostmap(std::string global\_frame, bool rolling\_window, bool track\_unknown) :

costmap\_(), global\_frame\_(global\_frame), rolling\_window\_(rolling\_window), initialized\_(false), size\_locked\_(false)

{

if (track\_unknown)

costmap\_.setDefaultValue(255);

else

costmap\_.setDefaultValue(0);

}

调用costmap\_ 的setDefaultValue 方法，实际上设定了类costmap\_2d 的一个成员变量default\_value\_ 这个值在class costmap\_2d 中是这样使用的：memset(costmap\_, default\_value\_, size\_x\_ \* size\_y\_ \* sizeof(unsigned char)); 实际存储地图的变量就是class costmap\_2d 的 costmap\_ 数据成员。

析构函数中，所有的操作就是弹出plugin： plugins\_.pop\_back(); 。

函数LayeredCostmap::resizeMap 就是给class costmap\_2d 的 costmap\_ 成员的大小重新做分配。然后根据plugin对每一层的地图调用其父类Costmap2D成员的initial 方法，实际效果就是将plugin所指向的每一层地图的大小都设置为和LayeredCostmap::costmap\_ 数据成员一样的空间大小。

{

size\_locked\_ = size\_locked;

costmap\_.resizeMap(size\_x, size\_y, resolution, origin\_x, origin\_y);

for (vector<boost::shared\_ptr<Layer> >::iterator plugin = plugins\_.begin(); plugin != plugins\_.end();

++plugin)

{

(\*plugin)->matchSize();

}

}

函数 LayeredCostmap::updateMap 完成对每一层地图的更新，更新过程分为两步updateBounds和updateCosts：

void LayeredCostmap::updateMap(double robot\_x, double robot\_y, double robot\_yaw)

{

if (rolling\_window\_)

{

double new\_origin\_x = robot\_x - costmap\_.getSizeInMetersX() / 2;

double new\_origin\_y = robot\_y - costmap\_.getSizeInMetersY() / 2;

costmap\_.updateOrigin(new\_origin\_x, new\_origin\_y);

}

if (plugins\_.size() == 0)

return;

minx\_ = miny\_ = 1e30;

maxx\_ = maxy\_ = -1e30;

for (vector<boost::shared\_ptr<Layer> >::iterator plugin = plugins\_.begin(); plugin != plugins\_.end();

++plugin)

{

(\*plugin)->updateBounds(robot\_x, robot\_y, robot\_yaw, &minx\_, &miny\_, &maxx\_, &maxy\_);

}

int x0, xn, y0, yn;

costmap\_.worldToMapEnforceBounds(minx\_, miny\_, x0, y0);

costmap\_.worldToMapEnforceBounds(maxx\_, maxy\_, xn, yn);

x0 = std::max(0, x0);

xn = std::min(int(costmap\_.getSizeInCellsX()), xn + 1);

y0 = std::max(0, y0);

yn = std::min(int(costmap\_.getSizeInCellsY()), yn + 1);

ROS\_DEBUG("Updating area x: [%d, %d] y: [%d, %d]", x0, xn, y0, yn);

if (xn < x0 || yn < y0)

return;

{

*// Clear and update costmap under a single lock*

boost::unique\_lock<Costmap2D::mutex\_t> lock(\*(costmap\_.getMutex()));

costmap\_.resetMap(x0, y0, xn, yn);

for (vector<boost::shared\_ptr<Layer> >::iterator plugin = plugins\_.begin(); plugin != plugins\_.end();

++plugin)

{

(\*plugin)->updateCosts(costmap\_, x0, y0, xn, yn);

}

}

}

这里我们来看这两个更新过程参数：

(\*plugin)->updateBounds(robot\_x, robot\_y, robot\_yaw, &minx\_, &miny\_, &maxx\_, &maxy\_);

(\*plugin)->updateCosts(costmap\_, x0, y0, xn, yn);

更新Bounds过程由于传入的参数是&minx\_, &miny\_, &maxx\_, &maxy\_ 构成了一个矩形范围。由于针对不同的类的实例，调用不同的类的方法。   
对于Static Map：

{

if (!map\_received\_ || !(has\_updated\_data\_ || has\_extra\_bounds\_))

return;

useExtraBounds(min\_x, min\_y, max\_x, max\_y);

double wx, wy;

mapToWorld(x\_, y\_, wx, wy);

\*min\_x = std::min(wx, \*min\_x);

\*min\_y = std::min(wy, \*min\_y);

mapToWorld(x\_ + width\_, y\_ + height\_, wx, wy);

\*max\_x = std::max(wx, \*max\_x);

\*max\_y = std::max(wy, \*max\_y);

has\_updated\_data\_ = false;

}

Static map 只在第一次做更新，Bounds 范围是整张Map的大小，而且在UpdateBounds过程中没有对Static Map层的数据做过任何的更新。

而对于 ObstacleLayer::updateBounds ：主要的操作是更新Obstacles Map层的数据，然后才是更新Bounds

std::vector<Observation> observations, clearing\_observations;

*// get the marking observations*

current = current && getMarkingObservations(observations);

*// get the clearing observations*

current = current && getClearingObservations(clearing\_observations);

*// update the global current status*

current\_ = current;

*// raytrace freespace*

for (unsigned int i = 0; i < clearing\_observations.size(); ++i)

{

raytraceFreespace(clearing\_observations[i], min\_x, min\_y, max\_x, max\_y);

}

for (std::vector<Observation>::const\_iterator it = observations.begin(); it != observations.end(); ++it)

{

const Observation& obs = \*it;

const pcl::PointCloud<pcl::PointXYZ>& cloud = \*(obs.cloud\_);

for (unsigned int i = 0; i < cloud.points.size(); ++i)

{

double px = cloud.points[i].x, py = cloud.points[i].y, pz = cloud.points[i].z;

*// now we need to compute the map coordinates for the observation*

unsigned int mx, my;

if (!worldToMap(px, py, mx, my))

{

continue;

}

unsigned int index = getIndex(mx, my);

costmap\_[index] = LETHAL\_OBSTACLE;

touch(px, py, min\_x, min\_y, max\_x, max\_y);

}

}

updateFootprint(robot\_x, robot\_y, robot\_yaw, min\_x, min\_y, max\_x, max\_y);

对于InflationLayer::updateBounds 则保持上一次的min\_x, min\_y, max\_x, max\_y 。   
对于VoxelLayer::updateBounds 更新过程和 ObstacleLayer::updateBounds 基本一致，只是增加了z 作为判断是否将2d地图的点设定为LETHAL\_OBSTACLE 。

updateCosts： 完成updateBounds 后，开始调用(\*plugin)->updateCosts(costmap\_, x0, y0, xn, yn); 。函数的第一个参数是指的master map，后面的bounds是对每个plugin自己维护的map的更新界限做设定。这里需要分析每一个单独的costmap和master map是哪些类在维护：   
Master map： 这是由类LayeredCostmap 的 Costmap2D costmap\_ 维护。   
StaticLayer StaticLayer VoxelLayer： 这些类是继承于Costmap2D ，因此可以直接操作Costmap2D 的数据成员 unsigned char\* costmap\_;。因此可以看成每一层地图都是类Costmap2D 的一个实例。   
InflationLayer 没有继承于Costmap2D 是因为这个类并不需要维护一张自己的地图，它仅仅是需要直接操作master map的数据就可以了。   
每个plugin调用自己代表的层的updateCosts方法：   
StaticLayer 和ObstacleLayer 基本上都是调用了CostmapLayer::updateWithOverwrite，CostmapLayer::updateWithTrueOverwrite， CostmapLayer::updateWithMax 等方法。因为CostmapLayer 是这两个的父类。   
但是InflationLayer::updateCosts 则不同，因为它既没有自己层的map实例，也不是从CostmapLayer 继承而来。它的updateCosts 是这个类的核心操作。关于他的updateCosts 操作，将在InflationLayer 篇具体分析这个算法的实现过程，这个算法实现了对障碍物膨胀操作。

函数bool LayeredCostmap::isCurrent() 主要的操作是对操作的实时性提供保证，提供是否发生超时的信息。

bool current = (ros::Time::now() - last\_updated\_).toSec() <= expected\_update\_rate\_.toSec();

函数void LayeredCostmap::setFootprint(conststd::vector<geometry\_msgs::Point>& footprint\_spec) ：

{

footprint\_ = footprint\_spec;

costmap\_2d::calculateMinAndMaxDistances(footprint\_spec, inscribed\_radius\_, circumscribed\_radius\_);

for (vector<boost::shared\_ptr<Layer> >::iterator plugin = plugins\_.begin(); plugin != plugins\_.end();

++plugin)

{

(\*plugin)->onFootprintChanged();

}

}

inscribed\_radius\_, circumscribed\_radius\_ 是计算得到的机器人尺寸的内切圆和外切圆半径。   
这里重点关注InflationLayer 类是如何调用onFootprintChanged() 的。对于其他类型的plugin实例来说，其本身并没有重载这个函数，所以都是调用的Layer类的空函数virtual void onFootprintChanged() {}：

cell\_inflation\_radius\_ = cellDistance(inflation\_radius\_);

computeCaches();

函数computeCaches()：

void InflationLayer::computeCaches()

{

if (cell\_inflation\_radius\_ == 0)

return;

*// based on the inflation radius... compute distance and cost caches*

if (cell\_inflation\_radius\_ != cached\_cell\_inflation\_radius\_)

{

deleteKernels();

cached\_costs\_ = new unsigned char\*[cell\_inflation\_radius\_ + 2];

cached\_distances\_ = new double\*[cell\_inflation\_radius\_ + 2];

for (unsigned int i = 0; i <= cell\_inflation\_radius\_ + 1; ++i)

{

cached\_costs\_[i] = new unsigned char[cell\_inflation\_radius\_ + 2];

cached\_distances\_[i] = new double[cell\_inflation\_radius\_ + 2];

for (unsigned int j = 0; j <= cell\_inflation\_radius\_ + 1; ++j)

{

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

}

}

cached\_cell\_inflation\_radius\_ = cell\_inflation\_radius\_;

}

for (unsigned int i = 0; i <= cell\_inflation\_radius\_ + 1; ++i)

{

for (unsigned int j = 0; j <= cell\_inflation\_radius\_ + 1; ++j)

{

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

}

}

}

在函数定义中，维护两个指针：

cached\_costs\_ = new unsigned char\*[cell\_inflation\_radius\_ + 2];

cached\_distances\_ = new double\*[cell\_inflation\_radius\_ + 2];

第一阶段，计算出cached\_distances\_: cached\_distances\_[i][j] = hypot(i, j); ,其中i j 的范围都是0：cell\_inflation\_radius\_ + 1。   
第二阶段，通过计算得到的cached\_distances\_ 计算cached\_costs\_：cached\_costs\_[i][j] = computeCost(cached\_distances\_[i][j]);。通过这个操作，现在可以任意给出在0-cell\_inflation\_radius\_ cell范围的两个cells的costs，以后对地图做膨胀时，只需要查看某个cell（i1,j1）和obstacle cell(i,j)的下标就可以通过查表知道这个cell的代价是多少。这个表的大小仅仅和机器人的几何尺寸相关，一旦机器人尺寸发生改变，这个函数就需要再次被调用。