9.4.3 拓展部分

拓展卡尔曼轨迹卡尔曼

9.4.4 卡尔曼滤波实战

十、 导航技术初步 *

对于一个脱离了愚蠢人类操作手的智能哨兵机器人来说，想去哪就去哪的能力是至关重要的。在本章节中，我们将了解导航技术的工作原理，并学习一些基本的路径规划算法，并将其应用到我们的机器人中。

10.1 导航技术总览

现在，你坐在哨兵的脑袋里面，你的目标是悄悄摸到敌方基地后面偷家。但是没认真看规则的你忘记了地图长什么样，你打开手机，发现好心的 wzx 部长已经帮你存了一份**标准赛场地图 [1]**。于是你打开**缺德地图 [2]**，把赛场地图传上去。你又打开手机 GPS，把自己的**定位 [3]** 也告诉缺德地图，最后，你输入目的地，缺德地图就帮你规划好了一条用时最少的路。于是你就跟着这条路径走，同时你并没有关掉 GPS，这样才知道自己有没有走错。在路上，你在**视野 [4]** 里看到了许多其他机器人，你以你秋名山车神的**车技 [5]**，在他们之间穿梭，避开了所有车。突然！What can I say, man! 无人机坠机坠在了你前进的道路上。你对车身大小了如指掌。只看一眼，你计算出哨兵过不去，路被堵了。鉴于无人机死状惨烈，这无人机短时间是飞不起来，你走不了这条路了。不过还好，每隔一段时间，缺德地图会再算一次路径，你可能会听到“正在为您重新规划路径”，他帮你找到了另一条路。虽然“缺德地图持续为您服务”，但是有时候你可能会开到一些“信号不好”，无法获取缺德地图服务的地方。此时，你要能够在不依靠缺德地图的情况下**自己找到 [6]** 信号比较好能重新连接上缺德地图服务的地方。最终你成功到达了目的地。

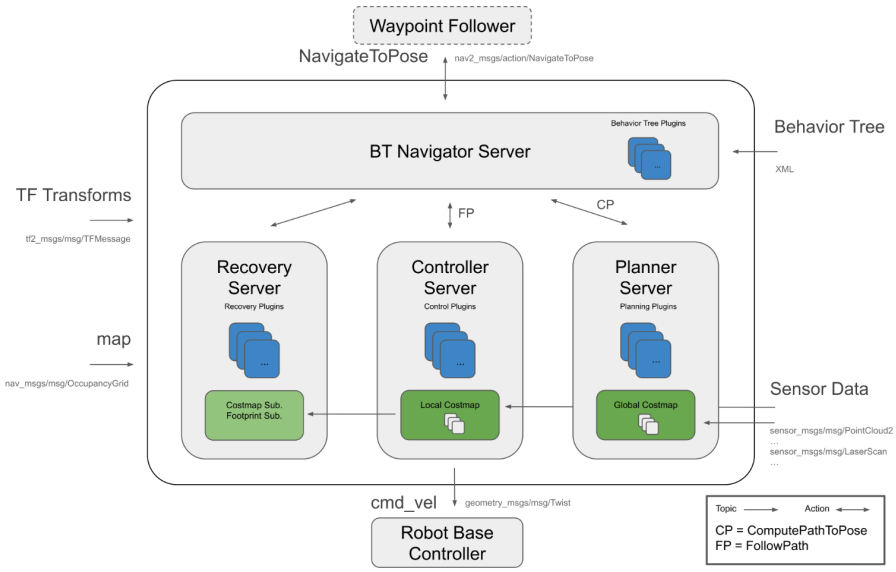

图 64　Nav2 导航框架

从上面可以看出，导航需要：

[1] 地图（全局代价地图 Global Costmap）

[2] 导航系统（全局规划器 Global Planner）

[3] 定位系统（多传感器融合定位）Sensor Transform

[4] 附近的视野（局部代价地图 Local Costmap）

[5] 车手（局部规划器 Local Planner）

[6] 应急方案（恢复行为树 Recovery Behavior Tree）

因此，导航的工作流程就是：

通过传感器数据与全局地图匹配，获得自身在地图的坐标；根据目标坐标，利用全局规划器算出路径；实时更新自身定位，并将扫描到的障碍物信息融合到局部代价地图中；根据局部代价地图，利用局部规划器算出车各个方向的速度应该是多少，发给下位机；如果遇到特殊情况则进入恢复行为树，暂停部分服务，直到脱离特殊情况；根据规定的时间间隔（或情况），重新获取定位和目标，规划全局路径，并一直循环。

10.2 定位

导航最重要的技术就是 **SLAM（simultaneous localization and mapping, 实时定位和建图技术）**简称 SM(不是)。先通过一个视频直观了解一下`https://www.bilibili.com/video/BV1tS4y1S7qE` 视频中随着机器人的移动，机器人的传感器获取到了环境信息，然后完成对博物馆地图的构建。有小伙伴可能会问，只看到建图没看到定位呀。细心观察可以发现建图的过程中，其实一直都在计算机器人的位置。

该技术的基础就是定位。我们先来了解一下在 ROS 中如何表示定位信息。在 ROS

中，我们使用 tf 树（transform tree）来表示物体（或者其他东西）之间的相对位置关系。每个需要用到的东西（如车体，雷达，相机）都有一个以自身为原点的三维坐标系（frame）。一个 tf 表示一个坐标系在另一个坐标系下的位姿信息（平移、旋转）。tf 树的原理就是通过表示坐标系之间的关系来确定各个坐标系之间的相对位置。常见的 frame 的约定俗成的名称有：

- /map：世界坐标系
- /odom：里程计坐标系
- /base_link：小哨兵的坐标系
- /camera_link：相机坐标系
- /laser_link：激光雷达坐标
- /imu_link：IMU 坐标系
- ...

tf 的主要信息包含父子节点（即两个坐标系，注意有父子关系，由父节点指向子节点）之间由 xyz 表示的平移关系和四元数表示的旋转关系。多个父子节点的 tf 可以组成一个 tf 树，树的根节点一般是 map，树的叶子节点一般是各个传感器的坐标系。通过 tf 树可以算出任意两个节点之间的坐标关系（相对位置）。

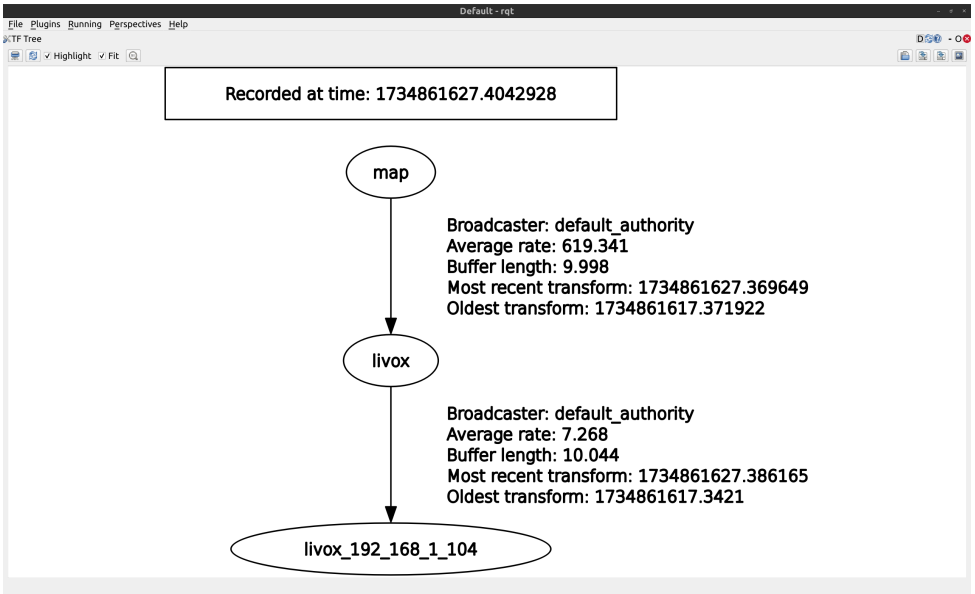

图 65　map 到 livox 到 livox_192_168_1_104 的 tf 树

根据ROS社区标准，在导航项目中，需要提供两个主要的坐标转换。 `map` 到 `odom` 的坐标变换由定位系统 (定位，建图，SLAM)提供， `odom` 到 `base_link` 的坐标转换由里程计系统提供。

注解：无需在机器人上使用LIDAR即可使用导航系统。不需要使用基于激光雷达的防撞、定位或SLAM系统。但是，Nav2确实可以提供说明和支持使用激光雷达对这些系统进行尝试和真实实现。使用基于视觉或深度传感器的定位系统和使用其他传感器来避免碰撞可以同样成功。唯一的要求就是在选择具体实现方式时遵循REP-105标准。

> REP-105标准
>
> REP 105 定义了导航和更大的ROS生态系统所需的框架和约定。应始终遵循这些约定，以利用社区中丰富的定位、里程计和SLAM项目。
>
> 简而言之，REP-105至少必须为机器人构造一个包含 `map` -> `odom` -> `base_link` -> `[sensorframes]` 的完整 的TF树。TF2是 ROS 2中的时变坐标变换库，Nav2使用TF2来表达和获取时间同步的坐标变换。全球定位系统 (GPS、SLAM、动作捕捉Motion Capture) 的工作是至少要提供 `map-> odom` 的坐标转换。然后，里程计系统的作用是提供 `odom` -> `base_link` 的坐标转化。关于 `base_link` 的其余坐标转换应该是静态的，并应在 URDF 中定义。

图 66　REP-105 标准

但是可怜的小哨兵并没有 GPS 直接定位。所以他只能用其他的定位技术，比如激光雷达、IMU、里程计、视觉相机等。

10.2.1 激光雷达定位

先简单介绍一下激光雷达的工作原理：激光雷达的工作原理是通过发射一束激光，然后接收到反射回来的信号，通过解码，得到反射点的坐标，根据反射光的强弱还能获得强度信息。

激光雷达定位主要分为四步：

1. 所有定位方法开始，都要先要有一张已有的地图，地图上记录了各个障碍物的位置、大小、置信度等信息。将地图上的某一点设为原点，以原点为中心展开的坐标系为 map 坐标系。
2. 激光雷达扫描四周环境，得到一张激光雷达“视野内”的地图，上面记载了周围的障碍物信息（更多细节将在下一节详细介绍）
3. 将这张地图与已有的地图进行对比匹配，得到雷达最可能存在的位置，将该位置与 map 坐标系之间的相对位置变化记为 map 与 ladar 之间的 tf。
4. 结合 laser 与 baselink 之间的 tf(一般是提前固定写好的，由雷达固定在车体的位置和朝向决定)，得到 baselink 与 map 之间的相对位置变化，即得到小哨兵在 map 坐标系下的位置。

其中从匹配方式大体可分为两种：二维和三维匹配。

- 三维匹配：直接将扫描到的点云与之前存储的点云匹配，定位更准确，但是匹配速度慢 (如 ICP,NDT 匹配算法)。常会用到 kd-tree,oct-tree 等数据结构加速匹配。

- 二维匹配：将三维的点云信息按一定规则压缩到二维（二向箔），获得附近的实时的二维地图，与已有的地图进行匹配（如 AMCL 算法）。地图为栅格地图。

按照匹配对象来分，也可以分为两种：全局匹配和局部匹配。

- 局部匹配：一般与先前一小段时间内的储存的点云或者栅格地图进行匹配，获得相对移动，是一种动态的匹配。因为要防止匹配目标过大，耗时太久，导致实时性不佳（局部路径规划对实时性需求较高），所以需要将较久之前扫描到的点云，或者距离较远的点云从 kdtree 之类的数据结构中清理掉。
- 全局匹配：将激光雷达的扫描信息与整个地图进行匹配，得到小哨兵的位置。一般用于重定位，即小哨兵迷失了方向的时候（如刚刚上电不知道自己在哪；被狠狠地创飞了，定位可能歪了；被神秘的力量干扰，导致进入恢复行为树；刚从恢复行为里出来，与定位系统失联许久……），将原先的 tf 清除，重新进行定位。

运动过程中多用动态的匹配，以便较快的更新定位，提高实时性。与整张全局地图的匹配，一般在特定情况下（如刚开机上电不知道自己在哪的时候；定位可能出现偏差的情况；或者执行完恢复行为之后。也可以时不时重定位一下，上赛季就是这么干的，效果好像不太行 QAQ）才进行，又叫重定位，即直接清零更新 baselink 和 map 之间的 tf。

10.2.2 里程计，IMU 定位

里程计定位较为简单，直接获取机器人各个方向运动的里程，加在 map 坐标系下的位置即为小哨兵的位置。IMU，即惯性测量单元，可以测量各方向的角速度和线加速度，通过积分，获得类似里程计的数据，加在 map 坐标系下的位置即为小哨兵的位置。这两个方法一般电控端就可以独立完成，纯电控的机器人常用。

10.2.3 视觉定位

视觉定位也是近年来较为流行的 SLAM 技术，它与激光雷达定位类似，但他获取的特征信息在相同情况下比激光雷达多。它通过提取图像中的特征点（一般是角点），与原先存储的地图里的特征点对比匹配，计算出相机的相对位移，累加后得到小哨兵在 map 坐标系的坐标变换。

10.2.4 多传感器融合

以上各个定位方法都有自己的局限性：

- imu, 里程计定位：由于 IMU 和里程计对运动（速度，加速度）测量的精度较高，定位较为准确。但由于其本身稳定性较差，易受干扰（如 IMU 受热噪声干扰的零漂，轮式里程计打滑等情况）；且他们都是“开环”的测量方式，一旦出现意外情况（比

如受干扰，或者某些情况下超过其测量范围 (比如被别的机器人肘了一下，瞬时加速度太大 IMU 测量不了))，测量的位置歪掉了，就再也回不来了，因为没法校正。

- 视觉定位：视觉定位的精度较低，且受相机的分辨率、光照、遮挡等因素影响较大。且对需要物体纹理较多效果才比较好。
- 雷达定位：激光雷达只能获取深度信息，这会导致在一些特殊情况中定位失效。如雷达在非常长的两堵一模一样的墙之间前进，虽然车在前进，但在雷达的视野里，左右离墙的距离不变，而前后因为超出探测距离所以数据也不变，他就以为自己没有运动了；又如雷达在一个一个正方形房间的正中心，你趁雷达不注意（即在两次雷达扫描的间隔之间），把他旋转 90 度。在雷达的视野里，前后左右的障碍物与之前的完全相同，所以他会以为自己没旋转。

为了解决这些问题，人们将多种传感器的数据融合在一起得到更准确定位方法。

基本原理是通过前后数据之间对比得出机器人位置的变化。然后回环检测，判断机器人是否到达之前到过的位置，可以解决位置估计误差问题，建图时可以纠正地图误差。

最常见的融合方法（也是我们现在使用的方法）是激光雷达和里程计的融合。我们新建一个节点 odom，他作为 map 的子节点，作为 base_link 的父节点。里程计提供 odom -> base_link 的 tf。里程计可以来自许多数据源，包括激光雷达、车轮编码器、VIO 和 IMU。里程计的目标是提供基于机器人运动的平滑和连续的局部坐标系。这样平滑输出就可用于精确运动的航行位置推算和在全局位置更新之间准确地更新机器人的位置。

而雷达重定位输出 map 到 odom 的 tf 作为校正数值记录。全局定位系统会相对全局坐标的坐标变换进行更新，以解决里程计的漂移问题。

这样，小哨兵的位置就通过激光雷达和里程计 fusion 的方式得到。

其他的融合方法还有：激光雷达与视觉融合，将点云与图像匹配，使点云除了位置信息外，还能拥有颜色等信息。

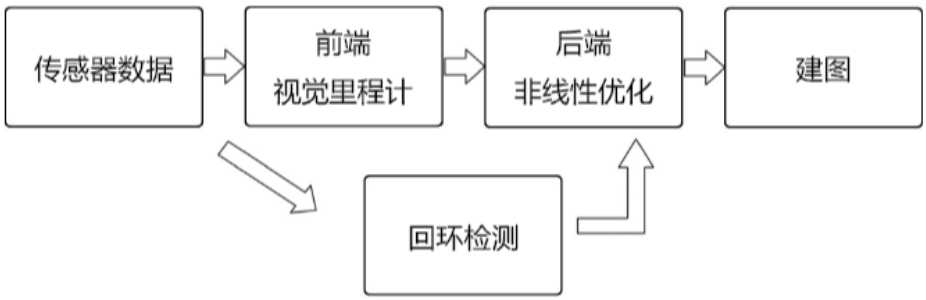

图 67　经典视觉 SLAM 结构

从算法的对数据的处理方式上看，目前常用的 SLAM 开源算法可以分为两类

1. 基于滤波: 比如扩展卡尔曼滤波（EKF: Extended Kalman Filter）、粒子滤波 (PF:

Particle Filter) 等。ROS 中的 gmapping、hector_slam 算法都是基于滤波实现的。我们使用的 point-lio 也使用了流形扩展卡尔曼滤波。

2. 基于图优化，先通过传感器进行构图，然后对图进行优化。

10.3 建图

导航需要地图。

我们使用的是二维地图（毕竟小哨兵还没学会飞行，在 z 轴上运动能力为 0（其实还有 roll 和 pitch），而且二维导航相关的导航，定位，建图方案都比较成熟）在机器人导航的时候，仅仅靠一张 SLAM 建立的原始地图是不够的，机器人在运动过程中可能会出现新的障碍物，也有可能发现原始地图中某一块的障碍物消失了，所以在机器人导航过程中维护的地图是一个动态的地图。那么怎么样才能建立一个动态的地图呢?

10.3.1 地图的获取

首先，我们接收激光雷达扫描到的点云图，然后对其进行处理，压缩到二维，得到地图。

处理的第一步是进行点云的滤波。下面是一些常用的滤波方法：*** 注意：因为点云是通过物体表面反射光信息获取的，所以一个实体只有表面会有点云，内部是没有的。**

1. **Voxel 滤波**：因为我们用的激光雷达比较牛逼，获得的近处点云比较稠密（保证远处的分辨率），这样直接计算会导致计算量过大，为了防止出现“三帧电竞”的情况出现，我们可以将点云按照空间分辨率进行分割，又叫体素网格下采样。

```
pcl::VoxelGrid<pcl::PointXYZ> voxfilter;
voxfilter.setInputCloud(cloud);
// 设置滤波器的体素大小
voxfilter.setLeafSize(0.1, 0.1, 0.1);
voxfilter.filter(*cloud);
```

这样，我们就将 cloud 里每个 0.1m*0.1m*0.1m 的体素（三维空间中的一个小立方体单元）内的点替换为该体素的中心点。将临近的点聚合到一个体素中来实现，可以减少点云数据的复杂度，同时保留整体的形状和结构。

2. **直通滤波**：直通滤波就是最简单的剔除不需要点云的滤波方法。比如小哨兵小小的，对它来说 50cm 以上的障碍物根本不影响他通过，那我们就可以把 0.5m 以上的点剔除掉。

```
pcl::PassThrough<pcl::PointXYZI> pass_through_filter_z_;
pass_through_filter_z_.setFilterFieldName("z");
```

```
    pass_through_filter_z_.setFilterLimits(min_z, max_z);
    pass_through_filter_z_.setFilterLimitsNegative(false);
    pass_through_filter_z_.setInputCloud(input_cloud);
    pass_through_filter_z_.filter(*input_cloud);
```

这样，我们就将 cloud 里的 z 轴坐标在 min_z 和 max_z 之外的点剔除掉了。同样的道理，还可以通过设置 xy 范围，把远处的，懒得考虑的点剔除掉。以及设置剔除一定半径内的点云，把哨兵本体的点云剔除掉。

3. **统计滤波**：在点云处理中，统计滤波是一种用于识别和移除离群点的方法，这些离群点可能是由于测量误差或噪声引起的。该方法基于假设，即点云数据中的大部分点是符合高斯分布的，而离群点则是那些与这种分布显著不同的点。

```
    pcl::StatisticalOutlierRemoval<pcl::PointXYZ> sor;
    sor.setInputCloud(cloud);
    // 设置近邻点的数量
    sor.setMeanK(50);
    // 设置标准差倍数阈值
    sor.setStddevMulThresh(1.0);
    // 设置为false表示移除离群点，设置为true表示保留离群点
    sor.setNegative(false);
    sor.filter(*cloud);
```

通过这种方式，我们可以计算每个点到其 K 个最近邻点的平均距离，并根据这个平均距离与全局平均距离的比较来确定是否为离群点。如果一个点的平均距离超过了全局平均距离加上标准差倍数的阈值，那么这个点就被认为是离群点并被移除。这样可以有效地减少点云中的噪声和异常值，提高后续处理的准确性和效率。

4. **特殊方法滤波**：有时候一些特殊的物体需要滤掉，这时候就要自己编写特定的代码剔除。比如地板，是的，地板也是实体，也会产生点云，但他不是障碍物，所以需要过滤掉。我们可以简单的通过直通滤波，把 z 轴坐标在轮子以下的点剔除掉。也可以通过聚类（即将点云数据中的点根据它们的空间位置或属性特征分组），将最大的聚类剔除掉，这样就剔除了地板。而我们现在用的是 tc 学长写的法向量剔除法，计算每个点的法向量，然后剔除法向量与 z 平面夹角小于一定角度的点（通俗来说：平的就是地板）。

现在，你就获取到了此时此刻哨兵周围的障碍物地图 (准确来说是一个地图层：障碍层，下面会介绍) 了。然后，让哨兵动起来，经过赛场的每一个角落，将每一时刻获取的点云叠加起来（通过我们上一节讲的定位方法就可以获取每一帧点云之间的相对位

置，就可以进行拼接叠加了），就得到了一个完整的地图——这，就是建图（比赛前会有哨兵建图环节，就是把赛场建成一张图。在陌生未知的环境下也可以一遍建图一遍在刚刚建出来的地图上进行导航）。这张地图可以为三维点云格式（.pcd 文件），也可以压缩成二维栅格地图格式（.pgm, .png, .jpg 文件），因为是按照真实世界按比例缩小，所以要有一个 yaml 文件来记录比例尺等参数来表示缩小比例。。

*** 栅格地图：即用二维图片记录地图信息。用一个个像素表示地图上的障碍物的位置、大小信息，灰度表示障碍物置信度，支持 pgm,png,jpg 格式，可人工修改。比如建完的图有一些噪声导致的奇怪黑点，我们就可以打开 PhotoShop，用白色笔刷手动擦除；又如，我们不想让哨兵到某个地方，就可以用黑笔画地为牢，桀桀桀**

10.3.2 地图的处理

现在的地图以及可以满足定位这一功能的需求了，但是为了满足地图的另一功能——路径规划的需求，我们还需要对地图进行处理。

代价地图 首先我们引入代价地图的概念, 想要计算出最佳的路径，我们就要给每一条路径一个分值。所以我们把不想去的地方设置一个大大的代价，比如我们不希望哨兵出现在障碍物的内部，那么就给把它的代价设为无穷大，在路径规划的时候路径就会避开它（具体的算法见 10.4 节）。

地图层 layer 为了让人们方便对地图进行自定义操作和拓展（“掺点小料”），nav2 提供了一种非常灵活的地图处理机制，即地图的 “层”，又叫 $plugins$ “插件”。将一层层的地图层叠加在一起，就可以形成一个完整的地图（根据不同的需求，我们可以有很多张完整的地图，用在不同的地方，最常见的就是，根据更新频率方式和用途不同，可以分为全局代价地图和局部代价地图）。比如：

global_costmap 全局代价地图是用来进行全局路径规划的，一般包含

1. ”$static_layer$” 静态层
2. ”$obstacle_layer$” 障碍层
3. ”$inflation_layer$” 膨胀层

静态层一般就是之前 SLAM 建好的地图 (.pgm 文件), 可以在上面进行路径规划。障碍层用于动态的记录传感器感知到的障碍物信息。但哨兵不是一个不计体积，不计质量，不计重力的理想质点。而路径规划又是按照质点来规划的，所以为了不让哨兵撞墙，膨胀层堂堂登场！

膨胀层通过将指定地图层（一般是所有下方地图层）里的障碍物膨胀（就是把障碍物周围的点的代价加大，一般离障碍物越近代价越大，但不会到无穷大，可以自定义加

大规则）一定的距离（一般至少是机器人的半径，但建议能覆盖全图，形成一个势场），让路径规划的时候避开这些膨胀的障碍物。

local_costmap 局部 (本地) 代价地图是用来进行局部路径规划的，一般包含

1. $obstacle\_layer$ 障碍层
2. $inflation\_layer$ 膨胀层

local_costmap 相比于 global_costmap，一般会限定一个较小的范围（如以机器人为中心的 5m*m 的正方形）以加快控制器计算速度。这是因为控制器的计算复杂度比较高，且由于是通过直接发布速度控制信息（x,y 线速度,yaw 角速度）直接控制机器人的运动，发布频率越快越好。

* 而且没必要。

对人类来说，永远不要担心 2 小时以后，和 8 公里以外的事情 (哲学小课堂)。

而对于哨兵来说，永远不要担心 dt_ref(轨迹时间分辨率) 以后，和 local_costmap 以外的事情。但是因此要确保 local_costmap 之外的障碍物不能影响到机器人的路径规划，换句话说，任何障碍物突然进入 local_costmap 的范围内，机器人都应该能从容优雅地避开，所以 local_costmap 的极限范围与机器人的性能有关（最大速度，最大加速度）

还有一些其他地图层：

1. $DenoiseLayer$：从最终成本图中去除椒盐噪声。
2. $KeepoutFilter$：在成本图中标记禁止区、较高权重或较低权重的区域
3. $SpeedFilter$：根据位置降低或提高机器人速度

除了这些地图层，还可以自定义的地图层。比如你想哨兵不走回头路，那么你就可以把哨兵走过的路径当成障碍物，变成一层地图层，添加在整张地图里，这样路径规划时就不会考虑这片区域了。

* 值得注意的是，地图层之间是有顺序的。比如一般膨胀层都在最后（最上面），这样它就会把下面的静态层，障碍层，自定义层……都进行膨胀。

10.4 路径规划

现在我们有了地图和定位，终于可以进入我们的正题导航了。我们使用的是 Nav2 导航框架，该项目力求以安全的方式让移动机器人从 A 点移动到 B 点。Nav2 也可以应用于其他应用，包括机器人导航，如下动态点跟踪，在这个过程中需要完成动态路径规划、计算电机的速度、避免障碍、恢复行为。

Nav2 使用行为树（详情见下一个章节）调用模块化服务器来完成一个动作。动作可以是计算路径、控制力、恢复或任何其他与导航相关的操作。服务器可以简单分为一大三小四个服务。一大：BT Navigator Server 导航行为树服务，通过这个大的服务来进行下面三个小服务组织和调用。

10.4.1 全局规划器

Planner Server，规划服务器，（全局规划器）是 Nav2 中的核心模块之一，它负责根据所选的命名法和算法规划机器人的全局路径，即从起始点到目标点的路径。说白了就是在地图上找路。他们通过全局代价地图 global_costmap，根据当前机器人的位置和目标位置，计算出从当前位置到目标位置的 (时间或路程) 最优路径。路径规划算法有很多种，还有他们的优化版本。这里，我们介绍两个最经典好用的算法：$Dijkstra$ 算法和 $A*$ 算法

$Dijkstra$ 算法 $Dijkstra$ 算法是一种最简单的广度优先的路径规划算法，它通过计算从起始点到其他所有点的最短路径，直到覆盖到终点。

`https://www.bilibili.com/video/BV19b4y1d7Hz/`

10.4.2 控制器

如果只有规划器会怎么样呢?

现在，登录你舍友的 LOL 账号，开一把排位晋级赛，在小地图上点击敌方基地。

图 68　全局路径（用 A* 算法根据障碍物（建筑）算出来的）

好的，一条全局导航路径生成出来了；好的，你的英雄沿着规划好的路径动起来了；好的，你被对面拿下一血并被队友问候全家。这是为什么？因为敌方是动态的，他的位置不是事先知道的，事先知道的只有地图，而全局规划器只考虑了地图，未考虑动态障碍物。局部规划器就是在你的视野里出现动态障碍物的时候，规划出一条道路绕开他，然后继续跟着全局路径走。

回到缺德地图，你这位车手就是局部规划器，缺德地图（全局规划器）只给大致的路线。如何避让冲出来的小学生，掉头，超车，等红灯等都是车手（局部规划器）决定的。

控制器，在 ROS 1 中也被称为局部规划器，是我们跟随全局计算路径或完成局部任务的方法。控制器有权访问局部环境表达，以尝试计算要跟随的基准路径的可行**控制工作** (即速度、加速度、角速度等)。许多控制器会将机器人向前投射到空间中，并在每次更新迭代时计算局部可行路径。控制器可以被编写为具有以下功能的工具：

- 跟随路径
- 使用里程计坐标系中的探测器，让机器人与充电站（桩）对接
- 登上电梯
- 与某个工具的接口交互

在 Nav2 中，控制器的一般任务是计算一个有效的**控制工作**以跟随全局规划路径。然而，有多个控制器类和局部规划器类。Nav2 项目的目标就是所有控制器算法都可以作为此服务器中的插件，以用于一般研究和产业任务中。

常用的局部规划器有 DWA,TEB 等，我们使用的是 TEB 算法。

10.5 恢复行为树

十一、 决策树算法 *

在机器学习领域，决策树算法是一种常用的算法，它可以使得机器人可以灵活地对外界环境做出正确的响应。在本章节中，我们将学习决策树算法的基本原理，并将其应用到我们的机器学习项目中。

A 第四章代码附录

1.1 my_node's cmake

Listing 1: my_node's cmake

```
cmake_minimum_required(VERSION 3.8)
project(my_package)
```