ROS Notebook

Wu Yutian

2021.11.13

前言

本书的主要内容包括：

- 学习古月居的相关入门课程视频的内容记录
- 阅读胡春旭的《ROS 机器人开发实践》的笔记整理
- 参考高翔的《视觉 SLAM 十四讲》补充了关于三维刚体运动学的内容
- 参考一些博客阅读 ros-navigation 导航包源码的思路整理
-

Wu Yutian
2021.11.13

Contents

1 Navigation 详细学习 **1**

1.1 base_local_planner 源码学习 1

1.1.1 源码相关文件 1

1.1.2 整体结构图 2

1.1.3 参数配置 2

1.1.4 trajectory_planner_ros.cpp 2

1.1.5 trajectory_planner.cpp 10

1.1.6 MapGrid 类和 MapCell 类 22

1.1.7 CostmapModel 类 26

Chapter 1

Navigation 详细学习

1.1 base_local_planner 源码学习

Movebase 使用的局部规划器默认为 TrajectoryPlannerROS，它循环检查是否到达目标点位置（给定位置误差范围内），若未到达，则调用 TrajectoryPlanner 类函数来进行局部路径规划，得到下一步速度，反馈给 Movebase；若已到达，则检查是否到达目标姿态，若未到达，先给机器人降速至阈值内，再使它原地旋转，直至达到目标姿态（给定姿态误差范围内），至此局部规划器完成任务。

1.1.1 源码相关文件

- 源码链接：

  `https://github.com/ros-planning/navigation/tree/melodic-devel`

- 源码注释链接：

  `https://github.com/W-yt/ROS_Notes/tree/master/navigation-melodic-devel/base_local_planner`

对应源码中的相关文件：

- base_local_planner/src/trajectory_planner_ros.cpp
- base_local_planner/src/trajectory_planner.cpp
- base_local_planner/src/costmap_model.cpp
- base_local_planner/src/map_grid.cpp
- base_local_planner/src/map_cell.cpp
- base_local_planner/include/line_iterator.h

1.1.2 整体结构图

1.1.3 参数配置

参数配置文件为:
参数列表:

-

1.1.4 trajectory_planner_ros.cpp

trajectory_planner_ros.cpp 中定义了 *TrajectoryPlannerROS* 类，是 *base_local_planner* 与 *movebase* 交互的主要接口文件，包含了封装好的局部规划器的主要功能。

TrajectoryPlannerROS::initialize

Movebase 在初始化了局部规划器 *TrajectoryPlannerROS* 类实例后即调用了 *initialize* 函数，这个函数的主要工作是从参数服务器下载参数值给局部规划器赋参。

首先设置了全局和本地规划结果的发布器 *g_plan_pub_* 和 *l_plan_pub_*，并用传入的参数 *costmap_ros*（格式为 *Costmap2DROS ROS* 的地图封装类，它整合了静态层、障碍层、膨胀层地图）来初始化本地规划器用到的代价地图。

代码如下:

```
ros::NodeHandle private_nh("~/" + name);
//发布全局规划在~/本地规划器名称/global_plan话题上
g_plan_pub_ = private_nh.advertise<nav_msgs::Path>("global_plan", 1);
//发布本地规划在~/本地规划器名称/local_plan话题上
l_plan_pub_ = private_nh.advertise<nav_msgs::Path>("local_plan", 1);

//初始化tf、局部代价地图
tf_ = tf;
costmap_ros_ = costmap_ros;

......

//复制一个代价地图供本地规划器使用
costmap_ = costmap_ros_->getCostmap();
//地图坐标系
global_frame_ = costmap_ros_->getGlobalFrameID();
//机器人底盘坐标系
robot_base_frame_ = costmap_ros_->getBaseFrameID();
```

接下来从参数服务器下载参数，并用它们来创建 $TrajectoryPlanner$ 类实例（它完成实际的速度计算工作），并开启动态参数配置服务。

TrajectoryPlannerROS::setPlan

该函数的作用为传入全局规划 (与全局路径的贴合程度将作为局部规划路线的一个打分项)。$Movebase$ 通过调用这个函数传入当前位置和目标点间规划好的全局路径 (全局规划的结果传递给局部规划器)。

代码如下：

```
bool TrajectoryPlannerROS::setPlan(const std::vector<geometry_msgs::PoseStamped>& orig_global_plan){
    if (!isInitialized()) {
      ROS_ERROR("This planner has not been initialized, ......");
      return false;
    }

    global_plan_.clear();
    global_plan_ = orig_global_plan;

    //when we get a new plan, we also want to clear any latch we may have on goal tolerances
    xy_tolerance_latch_ = false;

    reached_goal_ = false;
    return true;
  }
```

TrajectoryPlannerROS::computeVelocityCommands

该函数是该文件的核心函数，它在 $Movebase$ 的 $executeCycle$ 函数中被调用，$executeCycle$ 函数本身是被循环执行的，所以能够不断进行局部速度规划，从而获得连续的速度指令，控制机器人行动。

首先，获取 $global$ 系的当前位姿（使用从底盘到 $global$ 的转换），它可以用来判断是否行进到目标点。并将全局规划结果 $global\_plan\_$ 从地图系转换到 $global$ 系，得到 $transformed\_plan$，这里调用的 $transformGlobalPlan$ 函数来自 $goal\_functions.cpp$，这个文件中定义了一些辅助函数。

代码如下：

```
geometry_msgs::PoseStamped global_pose;
//获取global系的当前位姿(使用从底盘到global的转换)
if (!costmap_ros_->getRobotPose(global_pose)) {
    return false;
}

//将全局规划结果global_plan_从map系转换到global系，得到transformed_plan
```

```
std::vector<geometry_msgs::PoseStamped> transformed_plan;
if (!transformGlobalPlan(*tf_, global_plan_, global_pose, *costmap_, global_frame_, tran
sformed_plan)){
    ROS_WARN("Could not transform the global plan to the frame of the controller");
    return false;
}
```

接下来，判断是否要修剪全局规划。修剪是指在机器人前进的过程中，将一定阈值外的走过的路径点从 $global\_plan\_$ 和 $transformed\_plan$ 中去掉

代码如下：

```
if(prune_plan_){
  prunePlan(global_pose, transformed_plan, global_plan_);
}
```

接下来获取全局规划的目标点 (认为全局规划的最后一个路径点即为目标点)，获取它，得到目标 x、y 坐标及朝向。代码如下：

```
//认为全局规划的最后一个路径点即为目标点 获取目标点
const geometry_msgs::PoseStamped& goal_point = transformed_plan.back();
const double goal_x = goal_point.pose.position.x;
const double goal_y = goal_point.pose.position.y;
const double yaw = tf2::getYaw(goal_point.pose.orientation);
double goal_th = yaw;
```

接下来判断机器人是否已经到达了目标周围，如果机器人已经到达了目标位置附近，则判断机器人的朝向 (姿态) 是否到达了目标朝向附近：

如果当前朝向在目标朝向附近，则认为完成了任务，设置速度为 0，置停机器人。

如果未达到朝向 (姿态) 要求，调用 $TrajectoryPlanner$ 类的 $findBestPath$ 函数（它完成局部规划的实际工作）。接下来进行两步，降速、旋转：

- 如果机器人还未停止，调用类内 $stopWithAccLimits$ 函数，给机器人降速，直到降至降至一个极小值范围内，表示机器人停止，跳出该层判断，执行下一步；

- 如果机器人停止了，调用类内 $rotateToGoal$ 函数，让机器人旋转至目标姿态。

当机器人位置、姿态均符合要求，则认为完成了任务，设置速度为 0，置停机器人。

有个疑问，这里调用它的作用是什么？既然位置到了，只有姿态未达到，那么下面两步-降速、旋转就足够了，这里何必再调用 $findBestPath$ 做局部规划？

这部分的代码如下：

```
//如果机器人已经到达了目标周围
if (xy_tolerance_latch_ || (getGoalPositionDistance(global_pose, goal_x, goal_y) <= xy_goal_tolera
```

```
nce_)) {
        if (latch_xy_goal_tolerance_) {
            xy_tolerance_latch_ = true;
        }

        //检查是否达到了目标朝向
        double angle = getGoalOrientationAngleDifference(global_pose, goal_th);
        //达到目标位置 并且达到目标朝向
        if (fabs(angle) <= yaw_goal_tolerance_) {
            //设置速度为0 制停机器人
            cmd_vel.linear.x = 0.0;
            cmd_vel.linear.y = 0.0;
            cmd_vel.angular.z = 0.0;
            rotating_to_goal_ = false;
            xy_tolerance_latch_ = false;
            reached_goal_ = true;
        }
        //达到目标位置 但是未达到目标朝向
        else {
            //这里还需要重新再做局部路径规划?
            //we need to call the next two lines to make sure that the trajectory
            //planner updates its path distance and goal distance grids
            tc_->updatePlan(transformed_plan);
            Trajectory path = tc_->findBestPath(global_pose, robot_vel, drive_cmds);
            map_viz_.publishCostCloud(costmap_);

            //获取里程计的数据
            nav_msgs::Odometry base_odom;
            odom_helper_.getOdom(base_odom);

            //如果没有停下来(线速度没有下降到阈值之下) 则让机器人减速
            if (!rotating_to_goal_ && !base_local_planner::stopped(base_odom, rot_stopped_velocity_, t
        rans_stopped_velocity_)) {
                //考虑机器人加速度的限制
                if (!stopWithAccLimits(global_pose, robot_vel, cmd_vel)) {
                    return false;
                }
            }
            //如果已经停下来了(线速度下降到阈值以下) 则开始旋转到目标姿态
            else{
                //设置这个标志位表示允许机器人开始旋转到目标姿态
                rotating_to_goal_ = true;
                if(!rotateToGoal(global_pose, robot_vel, goal_th, cmd_vel)) {
                    return false;
                }
            }
        }
```

```
        //发布一个空的plan 因为已经到了目标位置
        publishPlan(transformed_plan, g_plan_pub_);
        publishPlan(local_plan, l_plan_pub_);

        return true;
    }
```

若未到达目标点误差范围内，调用 $TrajectoryPlanner$ 类的 $updatePlan$ 函数，将 $global$ 系下的全局规划传入，再调用 $findBestPath$ 函数，进行局部规划，速度结果填充在 $drive\_cmds$ 中，并得到局部路线 $plan$。再将 $drive\_cmds$ 的结果存储进 cmd_vel，返还给 $Movebase$ 发布，完成对机器人的运动控制。代码如下：

```
    //如果没有到达目标位置 则更新全局规划
    tc_->updatePlan(transformed_plan);

    //调用findBestPath函数进行局部规划
    //速度结果填充在drive_cmds中，并得到局部路线plan
    Trajectory path = tc_->findBestPath(global_pose, robot_vel, drive_cmds);

    //发布代价地图点云
    map_viz_.publishCostCloud(costmap_);

    //将drive_cmds的结果存储进cmd_vel
    cmd_vel.linear.x = drive_cmds.pose.position.x;
    cmd_vel.linear.y = drive_cmds.pose.position.y;
    cmd_vel.angular.z = tf2::getYaw(drive_cmds.pose.orientation);
```

接下来对生成路径 $path$ 的代价进行判断，若为负，说明是无效路径，返回 $false$；若为正，说明找到有效路径，将其进行格式转换后通过话题发布，便于对局部规划结果可视化。代码如下：

```
    //若生成路径path的代价值为负 则说明是无效路径(对于所有模拟路径 机器人的足迹都在振荡)
    if (path.cost_ < 0) {
        ROS_DEBUG_NAMED("trajectory_planner_ros",
            "The rollout planner failed to find a valid plan. ......");
        local_plan.clear();
        publishPlan(transformed_plan, g_plan_pub_);
        publishPlan(local_plan, l_plan_pub_);
        return false;
    }

    //如果路径代价正常，代表找到了有效路径
    ROS_DEBUG_NAMED("trajectory_planner_ros", "A valid velocity command of (%.2f, %.2f, %.2f) was foun
d for this cycle.", cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z);

    //用path填充本地路径local_plan
```

```
for (unsigned int i = 0; i < path.getPointsSize(); ++i) {
    double p_x, p_y, p_th;
    path.getPoint(i, p_x, p_y, p_th);
    geometry_msgs::PoseStamped pose;
    pose.header.frame_id = global_frame_;
    pose.header.stamp = ros::Time::now();
    pose.pose.position.x = p_x;
    pose.pose.position.y = p_y;
    pose.pose.position.z = 0.0;
    tf2::Quaternion q;
    q.setRPY(0, 0, p_th);
    tf2::convert(q, pose.pose.orientation);
    local_plan.push_back(pose);
}

//发布全局规划和已填充好的本地规划(用于可视化)
publishPlan(transformed_plan, g_plan_pub_);
publishPlan(local_plan, l_plan_pub_);
return true;
```

TrajectoryPlannerROS::stopWithAccLimits

该函数的作用是，机器人已达目标附近范围而姿态未达姿态要求时，在调整姿态前，将机器人速度降至阈值以下。

首先计算当前速度以最大反向加速度在一个仿真周期 $sim\_period\_$ 内可以降至的速度，角速度同理，得到下一步的速度。然后对其调用 $TrajectoryPlanner$ 类的 $checkTrajectory$ 函数，检查该采样速度能否生成有效路径，若可以，则将下一步速度储存在 cmd_vel，否则，速度置 0。该函数代码如下：

```
bool TrajectoryPlannerROS::stopWithAccLimits(const geometry_msgs::PoseStamped& global_pose,
                                             const geometry_msgs::PoseStamped& robot_vel,
                                             geometry_msgs::Twist& cmd_vel){
    //x方向速度 = (当前x向速度符号)× max(0,当前x向速度绝对值-最大加速度×仿真周期)
    double vx = sign(robot_vel.pose.position.x) * std::max(0.0, (fabs(robot_vel.pose.position.x) -
acc_lim_x_ * sim_period_));
    double vy = sign(robot_vel.pose.position.y) * std::max(0.0, (fabs(robot_vel.pose.position.y) -
acc_lim_y_ * sim_period_));
    double vel_yaw = tf2::getYaw(robot_vel.pose.orientation);
    double vth = sign(vel_yaw) * std::max(0.0, (fabs(vel_yaw) - acc_lim_theta_ * sim_period_));

    //检查速度命令是否合法
    double yaw = tf2::getYaw(global_pose.pose.orientation);
    bool valid_cmd = tc_->checkTrajectory(global_pose.pose.position.x, global_pose.pose.position.y,
yaw, robot_vel.pose.position.x, robot_vel.pose.position.y, vel_yaw, vx, vy, vth);

    //上述计算的如果合法 把速度控制指令存放到cmd_vel
```

```
    if(valid_cmd){
        ROS_DEBUG("Slowing down... using vx, vy, vth: %.2f, %.2f, %.2f", vx, vy, vth);
        cmd_vel.linear.x = vx;
        cmd_vel.linear.y = vy;
        cmd_vel.angular.z = vth;
        return true;
    }
    //如果不合法 全部置0
    cmd_vel.linear.x = 0.0;
    cmd_vel.linear.y = 0.0;
    cmd_vel.angular.z = 0.0;
    return false;
}
```

TrajectoryPlannerROS::rotateToGoal

在达到目标点误差范围内，且速度降至极小后，最后一步的工作是原地旋转至目标姿态。

首先计算机器人当前位姿角度和目标角度的差值，差值计算完成后，需要用几个条件对它进行限制：

- 最直接的限制，下一步的角速度要在预先设置的角速度允许范围内;
- 由于有角加速度的限制，需要保证下一步的角速度能够由当前角加速度在规定角加速度范围内达到;
- 还需要确保当机器人旋转到目标姿态时可以直接停下来，这里依据了速度平方公式 (设结束速度为 0):$v^2 = 2ax$，若超过这个速度，当机器人旋转到目标姿态时角速度无法降至 0，会“转过头”；
- 再次用预设角速度范围来限制下一步的角速度。

然后再检查计算出来的下一步速度生成的路径是否有效，如果有效则用其填充 cmd_vel，该函数的代码如下：

```
bool TrajectoryPlannerROS::rotateToGoal(const geometry_msgs::PoseStamped& global_pose,
                                        const geometry_msgs::PoseStamped& robot_vel,
                                        double goal_th, geometry_msgs::Twist& cmd_vel){
    //机器人姿态的偏角yaw
    double yaw = tf2::getYaw(global_pose.pose.orientation);
    //机器人速度的航偏角vel_yaw?
    double vel_yaw = tf2::getYaw(robot_vel.pose.orientation);

    //线速度设置为0
    cmd_vel.linear.x = 0;
    cmd_vel.linear.y = 0;
```

```
    //通过计算当前姿态与目标姿态的差值，通过这个差值来控制下一步的角速度
    double ang_diff = angles::shortest_angular_distance(yaw, goal_th);

    //下一步的角速度要在预先设置的角速度允许范围内
    double v_theta_samp =
        ang_diff>0.0 ? std::min(max_vel_th_, std::max(min_in_place_vel_th_, ang_diff))
                     : std::max(min_vel_th_, std::min(-1.0 * min_in_place_vel_th_, ang_diff));

    //由于角加速度的限制，需要保证下一步的角速度能够由当前角加速度在规定角加速度范围内达到
    //实际最大角速度=当前角速度+最大角加速度×1个仿真周期
    double max_acc_vel = fabs(vel_yaw) + acc_lim_theta_ * sim_period_;
    //实际最小角速度=当前角速度-最大角加速度×1个仿真周期
    double min_acc_vel = fabs(vel_yaw) - acc_lim_theta_ * sim_period_;
    //考虑角加速度 对角速度进行限制
    v_theta_samp = sign(v_theta_samp) * std::min(std::max(fabs(v_theta_samp), min_acc_vel), max_ac
c_vel);

    //还需要确保当机器人旋转到目标姿态时可以直接停下来 依据速度平方公式(设结束速度为0):v^2 = 2ax
    double max_speed_to_stop = sqrt(2 * acc_lim_theta_ * fabs(ang_diff));

    v_theta_samp = sign(v_theta_samp) * std::min(max_speed_to_stop, fabs(v_theta_samp));

    //重复第一个角速度限制:再次用预设角速度范围来限制下一步的角速度(因为这比角加速度的限制更重要)
    v_theta_samp =
        v_theta_samp>0.0 ? std::min(max_vel_th_, std::max(min_in_place_vel_th_, v_theta_samp))
                         : std::max(min_vel_th_, std::min(-1.0 * min_in_place_vel_th_, v_theta_samp));

    //检查计算出来的下一步速度生成的路径是否合法
    bool valid_cmd = tc_->checkTrajectory(global_pose.pose.position.x, global_pose.pose.position.y,
yaw, robot_vel.pose.position.x, robot_vel.pose.position.y, vel_yaw, 0.0, 0.0, v_theta_samp);

    ROS_DEBUG("Moving to desired goal orientation, th cmd: %.2f, valid_cmd: %d", v_theta_samp, vali
d_cmd);

    //若有效 则用它填充cmd_vel
    if(valid_cmd){
        cmd_vel.angular.z = v_theta_samp;
        return true;
    }

    cmd_vel.angular.z = 0.0;
    return false;
}
```

TrajectoryPlannerROS::checkTrajectory 和 TrajectoryPlannerROS::scoreTrajectory

checkTrajectory 和 *scoreTrajectory* 是在足够接近目标时，局部规划器产生降速和自转时生成的对应速度的路径。它们各自调用 *TrajectoryPlanner* 类的同名函数。

1.1.5 trajectory_planner.cpp

该文件是 *trajectory_planner_ros.cpp* 中主要功能的底层实现代码。

TrajectoryPlanner::updatePlan

Movebase 调用全局规划器生成全局路径后，传入 *TrajectoryPlannerROS* 封装类，再通过这个函数传入真正的局部规划器 *TrajectoryPlanner* 类中，并且将全局路径的最终点最为目标点 *final_goal*。函数代码如下：

```
void TrajectoryPlanner::updatePlan(const vector<geometry_msgs::PoseStamped>& new_plan,
                                   bool compute_dists){
    global_plan_.resize(new_plan.size());
    for(unsigned int i = 0; i < new_plan.size(); ++i){
        global_plan_[i] = new_plan[i];
    }

    //判断全局路径是否有效
    if(global_plan_.size() > 0){
        geometry_msgs::PoseStamped& final_goal_pose = global_plan_[ global_plan_.size() - 1 ];
        final_goal_x_ = final_goal_pose.pose.position.x;
        final_goal_y_ = final_goal_pose.pose.position.y;
        final_goal_position_valid_ = true;
    } else {
        final_goal_position_valid_ = false;
    }

    //compute_dists默认为false，即本地规划器在更新全局plan时，不重新计算path_map_和goal_map_
    if (compute_dists) {
        //reset the map for new operations
        path_map_.resetPathDist();
        goal_map_.resetPathDist();

        //make sure that we update our path based on the global plan and compute costs
        path_map_.setTargetCells(costmap_, global_plan_);
        goal_map_.setLocalGoal(costmap_, global_plan_);
        ROS_DEBUG("Path/Goal distance computed");
    }
}
```

TrajectoryPlanner::findBestPath

局部规划的整个流程体现在 $findBestPath$ 函数中，它能够在范围内生成下一步的可能路线，选择出最优路径，并返回该路径对应的下一步的速度。

首先记录当前位姿（$global$ 系下）、速度，机器人当前位置的 $footprint$。并且，对 $path\_map\_$ 和 $goal\_map\_$ 的值重置后调用 $setTargetCells$ 函数 $setLocalGoal$ 函数进行更新。

这两个地图是局部规划器中专用的“地图”，即 $MapGrid$ 类，和 $costmap$ 的组织形式一样，都以 $cell$ 为单位，$path\_map\_$ 记录各 $cell$ 与全局规划路径上的 $cell$ 之间的距离，$goal\_map\_$ 记录各 $cell$ 与目标 $cell$ 之间的距离，再最终计算代价时，将这两个因素纳入考虑，以保证局部规划的结果既贴合全局规划路径、又不致偏离目标。这部分代码如下：

```
//将当前机器人位置和方向转变成float形式的vector
Eigen::Vector3f pos(global_pose.pose.position.x, global_pose.pose.position.y,
                    tf2::getYaw(global_pose.pose.orientation));
Eigen::Vector3f vel(global_vel.pose.position.x, global_vel.pose.position.y,
                    tf2::getYaw(global_vel.pose.orientation));

//重置地图 清除所有障碍物信息以及地图内容
path_map_.resetPathDist();
goal_map_.resetPathDist();

//利用机器人当前位姿，获得机器人footpoint(足迹/覆盖位置)
std::vector<base_local_planner::Position2DInt> footprint_list = footprint_helper_.getFootprintCells(
pos, footprint_spec_, costmap_, true);

//标记机器人初始footprint内的所有cell为within_robot
for (unsigned int i = 0; i < footprint_list.size(); ++i) {
    path_map_(footprint_list[i].x, footprint_list[i].y).within_robot = true;
}

//更新路径地图和目标地图
path_map_.setTargetCells(costmap_, global_plan_);
goal_map_.setLocalGoal(costmap_, global_plan_);
ROS_DEBUG("Path/Goal distance computed");
```

接下来，调用 $createTrajectories$ 函数，传入当前位姿、速度、加速度限制，生成合理速度范围内的轨迹，并进行打分，结果返回至 $best$。然后对返回的结果进行判断，若其代价为负，表示说明所有的路径都不可用；若代价非负，表示找到有效路径，为 $drive\_velocities$ 填充速度后返回。代码如下：

```
Trajectory best = createTrajectories(pos[0], pos[1], pos[2], vel[0], vel[1], vel[2],
                                     acc_lim_x_, acc_lim_y_, acc_lim_theta_);
ROS_DEBUG("Trajectories created");

//如果找到的best轨迹的代价为负，表示说明所有的路径都不可用
if(best.cost_ < 0){
    drive_velocities.pose.position.x = 0;
    drive_velocities.pose.position.y = 0;
    drive_velocities.pose.position.z = 0;
```

```
        drive_velocities.pose.orientation.w = 1;
        drive_velocities.pose.orientation.x = 0;
        drive_velocities.pose.orientation.y = 0;
        drive_velocities.pose.orientation.z = 0;
    }
    //若代价非负，表示找到有效路径，为drive_velocities填充速度后返回
    else{
        drive_velocities.pose.position.x = best.xv_;
        drive_velocities.pose.position.y = best.yv_;
        drive_velocities.pose.position.z = 0;
        tf2::Quaternion q;
        q.setRPY(0, 0, best.thetav_);
        tf2::convert(q, drive_velocities.pose.orientation);
    }
    //返回最优轨迹
    return best;
```

TrajectoryPlanner::createTrajectories

该函数传入当前位姿、速度、加速度限制, 生成合理速度范围内的轨迹, 并进行打分, 找到代价最低的轨迹返回。

首先，计算可行的线速度和角速度范围，接下来根据预设的线速度与角速度的采样数，和上面计算得到的范围，分别计算出采样间隔，并把范围内最小的线速度和角速度作为初始采样速度。代码如下:

```
    double max_vel_x = max_vel_x_, max_vel_theta;
    double min_vel_x, min_vel_theta;

    //如果最终的目标是有效的(全局规划非空)
    if(final_goal_position_valid_){
        //计算当前位置和目标位置之间的距离: final_goal_dist
        double final_goal_dist = hypot(final_goal_x_ - x, final_goal_y_ - y);
        //最大速度:考虑预设的最大速度和"起点与目标直线距离/总仿真时间"
        max_vel_x = min(max_vel_x, final_goal_dist / sim_time_);
    }

    //继续计算线速度与角速度的上下限，使用的限制是:在一段时间内，由最大加减速度所能达到的速度范围

    //dwa: dynamic window approach
    //如果使用dwa法，则用的是轨迹前向模拟的周期sim_period_ (专用于dwa法计算速度的一个时间间隔)
    if (dwa_) {
        max_vel_x = max(min(max_vel_x, vx + acc_x * sim_period_), min_vel_x_);
        min_vel_x = max(min_vel_x_, vx - acc_x * sim_period_);

        max_vel_theta = min(max_vel_th_, vtheta + acc_theta * sim_period_);
        min_vel_theta = max(min_vel_th_, vtheta - acc_theta * sim_period_);
```

```
//如果不使用dwa法，则用的是整段仿真时间sim_time_
} else {
    max_vel_x = max(min(max_vel_x, vx + acc_x * sim_time_), min_vel_x_);
    min_vel_x = max(min_vel_x_, vx - acc_x * sim_time_);

    max_vel_theta = min(max_vel_th_, vtheta + acc_theta * sim_time_);
    min_vel_theta = max(min_vel_th_, vtheta - acc_theta * sim_time_);
}

//根据预设的线速度与角速度的采样数，和上面计算得到的范围，分别计算出速度的采样间隔(也就是速度的分辨率)
double dvx = (max_vel_x - min_vel_x) / (vx_samples_ - 1);
double dvtheta = (max_vel_theta - min_vel_theta) / (vtheta_samples_ - 1);

//把范围内最小的线速度和角速度作为初始采样速度
double vx_samp = min_vel_x;
double vtheta_samp = min_vel_theta;
//y向速度不进行采样遍历
double vy_samp = 0.0;
```

声明 *best_traj* 和 *comp_traj*（将代价都初始化为-1），分别用来存储最优的轨迹和当前获得的轨迹：

```
Trajectory* best_traj = &traj_one;
best_traj->cost_ = -1.0;
Trajectory* comp_traj = &traj_two;
comp_traj->cost_ = -1.0;
```

在机器人没有处于逃逸状态时，开始遍历所有线速度和角速度，调用类内 *generateTrajectory* 函数用它们生成轨迹。在遍历时，单独拎出角速度 $= 0$，即直线前进的情况，避免由于采样间隔的设置而跃过了这种特殊情况。迭代过程中将代价最小的路径存放在 *best_traj*：

```
if (!escaping_) {
  //循环所有x速度
  for(int i = 0; i < vx_samples_; ++i) {
    //在遍历时，单独拎出角速度=0，即直线前进的情况，避免由于采样间隔的设置而跃过了这种特殊情况
    vtheta_samp = 0;
    //调用generateTrajectory函数生成轨迹,并将轨迹保存在comp_traj中
    generateTrajectory(x, y, theta, vx, vy, vtheta, vx_samp, vy_samp, vtheta_samp,
                       acc_x, acc_y, acc_theta, impossible_cost, *comp_traj);

    //如果新生成的轨迹的代价更小,则将其放到best_traj
    if(comp_traj->cost_ >= 0 && (comp_traj->cost_ < best_traj->cost_ || best_traj->cost_ < 0)){
      swap = best_traj;
      best_traj = comp_traj;
      comp_traj = swap;
    }
```

```
    //角速度=最小值
    vtheta_samp = min_vel_theta;
    //迭代循环生成所有角速度的路径,并打分
    for(int j = 0; j < vtheta_samples_ - 1; ++j){
      generateTrajectory(x, y, theta, vx, vy, vtheta, vx_samp, vy_samp, vtheta_samp,
                         acc_x, acc_y, acc_theta, impossible_cost, *comp_traj);

      if(comp_traj->cost_ >= 0 && (comp_traj->cost_ < best_traj->cost_ || best_traj->cost_ < 0)){
        swap = best_traj;
        best_traj = comp_traj;
        comp_traj = swap;
      }
      vtheta_samp += dvtheta;
    }
    vx_samp += dvx;
  }

  //只对holonomic robots迭代循环y速度，一般的机器人没有y速度
  if (holonomic_robot_) {...
}
```

接下来，继续考虑线速度 $= 0$（原地旋转）的情况，但是需要注意，这部分代码并不是所有线速度为 0 时都会执行，下面分析一下什么情况会使用这一部分线速度为 0 的代码：

- 当 $TrajectoryPlannerROS$ 中，位置已经到达目标（误差范围内），姿态已达，则直接发送 0 速；姿态未达，则调用降速函数和原地旋转函数，并调用 $checkTrajectory$ 函数检查合法性，直到旋转至目标姿态。而 $checkTrajectory$ 函数调用的是 $scoreTrajectory$ 和 $generateTrajectory$，不会调用 $createTrajectory$ 函数，所以，快要到达目标附近时的原地旋转，根本不会进入到这个函数的这部分来处理。

- 并且，由于局部规划器的路径打分机制（后述）是：“与目标点的距离”和“与全局路径的偏离”这两项打分都只考虑路径终点的 $cell$，而不是考虑路径上所有 $cell$ 的综合效果，机器人运动到一个 $cell$ 上，哪怕有任何一条能向目标再前进的无障碍路径，它的最终得分一定是要比原地旋转的路径得分来得高的。

- 所以，这里的原地自转，是**行进过程中的、未达目标附近时的原地自转**，并且，是**机器人行进过程中遇到障碍、前方无路可走只好原地自转**，或是**连原地自转都不能满足，要由逃逸状态后退一段距离，再原地自转调整方向**，准备接下来的行动。一种可能情况是机器人行进前方遇到了突然出现而不在地图上的障碍。

这一部分的代码如下：

```
vtheta_samp = min_vel_theta;
vx_samp = 0.0;
```

```
vy_samp = 0.0;

//let's try to rotate toward open space
double heading_dist = DBL_MAX;

//循环所有角速度
for(int i = 0; i < vtheta_samples_; ++i) {
    //强制最小原地旋转速度,因为底盘无法处理过小的原地转速(相对前进过程中转向来说)
    double vtheta_samp_limited = vtheta_samp > 0 ? max(vtheta_samp, min_in_place_vel_th_)
                                                 : min(vtheta_samp, -1.0 * min_in_place_vel_th_);
    //产生遍历角速度的路径
    generateTrajectory(x, y, theta, vx, vy, vtheta, vx_samp, vy_samp, vtheta_samp_limited,
                       acc_x, acc_y, acc_theta, impossible_cost, *comp_traj);

    //如果新生成的轨迹的代价更小,则将其放到best_traj
    //注意如果能找到合法的原地旋转,相比之下,我们就不希望选择以y向速度进行移动,而是选择进行原地旋转
    if(comp_traj->cost_ >= 0
        && (comp_traj->cost_ <= best_traj->cost_ || best_traj->cost_ < 0 || best_traj->yv_ != 0.0)
        && (vtheta_samp > dvtheta || vtheta_samp < -1 * dvtheta)){
        double x_r, y_r, th_r;
        //获取新路径的终点(原地)(因为是原地旋转)
        comp_traj->getEndpoint(x_r, y_r, th_r);
        //计算沿旋转后朝向的cell(也就是朝向前进heading_lookahead_距离后的位置)
        x_r += heading_lookahead_ * cos(th_r);
        y_r += heading_lookahead_ * sin(th_r);

        unsigned int cell_x, cell_y;
        //转换到地图坐标系
        if (costmap_.worldToMap(x_r, y_r, cell_x, cell_y)) {
            //计算到目标点的距离
            double ahead_gdist = goal_map_(cell_x, cell_y).target_dist;
            //取距离最小的放进best_traj(heading_dist初始为无穷大)
            if (ahead_gdist < heading_dist) {
                //结合后面的抑制震荡部分代码(防止机器人在一个小范围内左右来回乱转)
                //只有stuck_left为假(机器人上一时刻没有向左旋转)的时候
                //才允许使用向右的角速度更新best_traj
                if (vtheta_samp < 0 && !stuck_left) {
                    swap = best_traj;
                    best_traj = comp_traj;
                    comp_traj = swap;
                    heading_dist = ahead_gdist;
                }
                else if(vtheta_samp > 0 && !stuck_right) {
                    swap = best_traj;
                    best_traj = comp_traj;
                    comp_traj = swap;
                    heading_dist = ahead_gdist;
```

```
                }
            }
        }
    }
    vtheta_samp += dvtheta;
}
```

关于上面代码中 “计算沿旋转后朝向的 $cell$”，并以此为依据，计算代价更新 $best\_traj$ 的理解：

这个计算得到的相当于机器人原地旋转后“面向”的 $cell$,它和机器人的距离是 $headin\_lookahead\_$，由于 $goal\_map\_$ 上的障碍物 $cell$ 值为 $obstacleCosts$（大于正常 $cell$），所以经过迭代，必然会筛选掉计算结果是障碍物的情形，也就是机器人旋转后不会面向障碍物；同时筛选后，也能得到和目标点距离最短的计算结果，保证机器人在旋转后向前行进，距离目标点的路程更短。

至此轨迹生成已完成，如果轨迹 $cost$ 非负，即找到有效轨迹，则对生成的轨迹进行震荡抑制(震荡抑制能够避免机器人在一个小范围内左右来回乱转)，然后返回有效的轨迹，代码如下：

```
//如果最优轨迹的代价大于0(有效)
if (best_traj->cost_ >= 0) {
    //当找到的最优轨迹的线速度为负时(逃逸模式)
    //正常情况线速度的遍历范围都是正的 因此如果发现最优轨迹的线速度为<=0
    //说明上一次的轨迹无效 发布了后退的指令(逃逸模式)
    if (!(best_traj->xv_ > 0)) {
        //角速度为负 标记正在向右旋转(rotating_right)
        //这里要想印证确实是这样理解，需要检查角速度为负是否真的对应机器人向右旋转
        if (best_traj->thetav_ < 0) {
            //再一次发现向右旋转 标记stuck_right
            if (rotating_right){
                stuck_right = true;
            }
            rotating_right = true;
        }
        //角速度为正 标记正在向左旋转(rotating_left)
        else if (best_traj->thetav_ > 0) {
            //再一次发现向左旋转 标记stuck_left(禁止下一时刻直接向右旋转,导致左右乱转出现振荡)
            if (rotating_left){
                stuck_left = true;
            }
            rotating_left = true;
        }
        //y向速度为正 标记正在向右平移
        else if(best_traj->yv_ > 0) {
            if (strafe_right) {
                stuck_right_strafe = true;
            }
            strafe_right = true;
        }
        //y向速度为负 标记正在向左平移
```

```
        else if(best_traj->yv_ < 0){
            if (strafe_left) {
                stuck_left_strafe = true;
            }
            strafe_left = true;
        }
        //记录当前的位置
        prev_x_ = x;
        prev_y_ = y;
    }

    //必须远离上面记录的位置一段距离后才恢复标志位
    double dist = hypot(x - prev_x_, y - prev_y_);
    if (dist > oscillation_reset_dist_) {
        rotating_left = false;
        rotating_right = false;
        strafe_left = false;
        strafe_right = false;
        stuck_left = false;
        stuck_right = false;
        stuck_left_strafe = false;
        stuck_right_strafe = false;
    }

    //判断是否退出逃逸状态
    dist = hypot(x - escape_x_, y - escape_y_);
    if(dist > escape_reset_dist_ ||
        fabs(angles::shortest_angular_distance(escape_theta_, theta)) > escape_reset_theta_){
        escaping_ = false;
    }
    //注意这里直接返回了 不再对后面产生影响
    return *best_traj;
}
```

轨迹有效的部分结束，当轨迹 *cost* 为负即无效时，执行接下来的部分，设置一个负向速度，产生让机器人缓慢退后的轨迹。若后退速度生成的轨迹的终点有效（> -2.0），进入逃逸状态，循环后退、自转，并且记录下的逃逸位置和姿态，只有当离开逃逸位置一定距离或转过一定角度，才能退出逃逸状态，再次规划前向速度。代码如下：(注意在这过程中不断进行着震荡判断和逃逸判断)

```
//当轨迹cost为负即无效时，执行接下来的部分，设置一个负向速度，产生让机器人缓慢退后的轨迹
vtheta_samp = 0.0;
vx_samp = backup_vel_;
vy_samp = 0.0;
generateTrajectory(x, y, theta, vx, vy, vtheta, vx_samp, vy_samp, vtheta_samp,
                   acc_x, acc_y, acc_theta, impossible_cost, *comp_traj);

//即使静态地图显示后面为阻塞,仍允许机器人缓慢向后移动
```

```
//也就是不需要判断后退时的轨迹是否是best,只要给了向后的速度,那么默认就是best(因为都是不得已才给的)
swap = best_traj;
best_traj = comp_traj;
comp_traj = swap;

double dist = hypot(x - prev_x_, y - prev_y_);
if (dist > oscillation_reset_dist_) {
    rotating_left = false;
    rotating_right = false;
    strafe_left = false;
    strafe_right = false;
    stuck_left = false;
    stuck_right = false;
    stuck_left_strafe = false;
    stuck_right_strafe = false;
}

//若后退速度生成的轨迹的终点有效(>-2.0)(为什么这个条件就是有效), 进入逃逸状态(后退一段距离)
//逃逸状态起始就是不再前进, 不进入if(!escaping_)的分支
if (!escaping_ && best_traj->cost_ > -2.0) {
    escape_x_ = x;
    escape_y_ = y;
    escape_theta_ = theta;
    escaping_ = true;
}

//判断是否退出逃逸状态
dist = hypot(x - escape_x_, y - escape_y_);
if (dist > escape_reset_dist_ ||
        fabs(angles::shortest_angular_distance(escape_theta_, theta)) > escape_reset_theta_) {
    escaping_ = false;
}

//若后退轨迹遇障, 还是继续后退, 因为后退一点后立刻就会进入原地自转模式
if(best_traj->cost_ == -1.0)
    best_traj->cost_ = 1.0;
```

如果最终无法找到一个有效的路径，则返回一个负的 *cost*，本次局部规划失败。

TrajectoryPlanner::generateTrajectorie

该函数根据给定的速度和角速度采样生成单条路径和其代价 (该函数被 $scoreTrajectory$ 和 $createTrajectories$ 调用)。

首先保存当前的位置和速度值，进而计算仿真步数和每一步对应的时间：

```
//计算线速度的大小(速度合成后)
double vmag = hypot(vx_samp, vy_samp);
```

```
//计算仿真步数(这里的计算原理是什么 下面的公式量纲都不同)
int num_steps;
if(!heading_scoring_) {
    //在这里sim_granularity_表示仿真点之间的距离间隔
    //仿真步数 = max(速度模×总仿真时间/距离间隔, 角速度/角速度间隔), 四舍五入(fabs函数求绝对值)
    num_steps = int(max((vmag * sim_time_) / sim_granularity_,
                        fabs(vtheta_samp) / angular_sim_granularity_) + 0.5);
} else {
    //在这里sim_granularity_代表仿真的时间间隔
    num_steps = int(sim_time_ / sim_granularity_ + 0.5);
}

//至少选取一步,即使不会移动我们也会对当前位置进行评分
if(num_steps == 0) {
    num_steps = 1;
}

//每一步的时间
double dt = sim_time_ / num_steps;
double time = 0.0;
```

接下来循环生成轨迹，并计算轨迹对应的代价值。若该点足迹不遇障，且该点的 $goal_dist$ 与 $path_dist$ 存在，则将其加入轨迹。然后调用 $computeNewVelocity$ 函数计算该点的速度，再调用函数 $computeNewXPosition$ 通过航迹推演公式计算下一个路径点的坐标，用于下一次循环。如此往复填充路径坐标，并更新路径 $cost$。整体循环的代码如下：

```
//循环生成轨迹，并计算轨迹对应的代价值
for(int i = 0; i < num_steps; ++i){
    unsigned int cell_x, cell_y;
    //防止路径跑出已知地图
    //当前位置转换到地图上，如果无法转换，说明该路径点不在地图上，将其代价设置为-1.0，并返回
    if(!costmap_.worldToMap(x_i, y_i, cell_x, cell_y)){
        traj.cost_ = -1.0;
        return;
    }

    //考虑机器人的大小，把当前点扩张到机器人在该点的足迹范围
    //获得机器人在该点时它的足迹所对应的代价,如果足迹遇障，直接返回-1
    double footprint_cost = footprintCost(x_i, y_i, theta_i);

    //机器人在路径上遇障
    if(footprint_cost < 0){
        traj.cost_ = -1.0;
        return;
    }
```

```
//更新occ_cost: 把所有路径点的最大障碍物代价设置为路径的occ_cost
occ_cost = std::max(std::max(occ_cost, footprint_cost), double(costmap_.getCost(cell_x, cell_y)));

//这里感觉不管是简单的追踪策略还是复杂一些考虑goal_dist和path_dist甚至考虑heading_diff的策略
//实际上都是只考虑了当前仿真轨迹的最后一点的cost

//比较简单的追踪策略，只考虑与目标点之间的直线距离(只更新goal_dist)
if (simple_attractor_) {goal_dist = (x_i - global_plan_[global_plan_.size() -1].pose.position.x)
                                  * (x_i - global_plan_[global_plan_.size() -1].pose.position.x)
                                  + (y_i - global_plan_[global_plan_.size() -1].pose.position.y)
                                  * (y_i - global_plan_[global_plan_.size() -1].pose.position.y);
//借助goal_map_和path_map_获取该点与目标点及全局规划路径之间的距离(更新goal_dist和path_dist)
} else {
    bool update_path_and_goal_distances = true;

    //如果为朝向打分
    if (heading_scoring_) {
    //heading_scoring_timestep_是给朝向打分时在时间上要看多远
    //也就是在路径上走过一个特定时刻(heading_scoring_timestep_)后，才为朝向打分一次
    if (time >= heading_scoring_timestep_ && time < heading_scoring_timestep_ + dt) {
        //headingDiff函数的具体过程是:
        //  从全局路径终点(目标点)开始迭代，当前点与全局路径上的各点依次连线获得cost
        //  cost为正(无障碍)则计算:当前点与迭代到的点间的连线方向与当前点的姿态之差，并返回;
        //  若所有连线cost都为负，返回极大值
        heading_diff = headingDiff(cell_x, cell_y, x_i, y_i, theta_i);
    } else {
        update_path_and_goal_distances = false;
    }
    }

    //如果需要为朝向打分，则同样也等到为朝向打分的特定时刻才更新path_dist和goal_dist
    if (update_path_and_goal_distances) {
    //更新路径距离与目标距离(只考虑当前轨迹的终点cell)
    path_dist = path_map_(cell_x, cell_y).target_dist;
    goal_dist = goal_map_(cell_x, cell_y).target_dist;

    //如果目标距离或路径距离  impossible_cost(地图尺寸)，代价设置为-2.0
    if(impossible_cost <= goal_dist || impossible_cost <= path_dist){
        traj.cost_ = -2.0;
        return;
    }
    }
}

//若该点足迹不遇障，且该点的goal_dist与path_dist存在，加入轨迹
traj.addPoint(x_i, y_i, theta_i);
```

```
    //计算该点的速度
    //速度计算函数使当前速度在dt时间内以加速度acc_x向采样速度靠近，到达采样速度后将不再改变
    //所以实际上每条轨迹都是一个由当前速度趋向并稳定在采样速度的过程
    //所以说采样速度在这个函数中，是该函数的仿真过程的目标速度
    //而这里调用computeNewVelocity函数的返回值是仿真过程的每个step的速度
    //(因为无法在一个step就达到采样速度)
    //注意:这里对速度的计算与我们发布给机器人的速度无关，这个速度只为了推算下一个点，获得路径
    //而我们真正发布给机器人的速度是采样速度
    //真实世界里机器人由当前速度->采样速度的过程对应我们地图上本次仿真的轨迹
    vx_i = computeNewVelocity(vx_samp, vx_i, acc_x, dt);
    vy_i = computeNewVelocity(vy_samp, vy_i, acc_y, dt);
    vtheta_i = computeNewVelocity(vtheta_samp, vtheta_i, acc_theta, dt);

    //通过计算出的速度计算下一个位置、姿态(使用匀速运动公式)
    x_i = computeNewXPosition(x_i, vx_i, vy_i, theta_i, dt);
    y_i = computeNewYPosition(y_i, vx_i, vy_i, theta_i, dt);
    theta_i = computeNewThetaPosition(theta_i, vtheta_i, dt);

    //增加时间
    time += dt;
}
```

最后，我们整合路径距离、目标距离、障碍代价以及航向，得到一个综合的代价值:

```
double cost = -1.0;
//如果打分不考虑航向
if (!heading_scoring_) {
  cost = path_distance_bias_ * path_dist +
         goal_distance_bias_ * goal_dist +
         occdist_scale_ * occ_cost;
} else {
  cost = path_distance_bias_ * path_dist +
         goal_distance_bias_ * goal_dist +
         occdist_scale_ * occ_cost +
         0.3 * heading_diff;
}
traj.cost_ = cost;
```

TrajectoryPlanner::checkTrajectorie 和 TrajectoryPlanner::scoreTrajectorie

这两个函数并未进行实际工作，$checkTrajectory$ 调用 $scoreTrajectory$，$scoreTrajectory$ 调用 $generateTrajectory$，生成单条路径并返回代价。它们是在足够接近目标时，局部规划器产生降速和自转时生成的对应速度的路径。(它们被 $TrajectoryPlannerROS$ 类中的同名函数调用)

1.1.6 MapGrid 类和 MapCell 类

这两个类是局部规划器专门用来确定各 cell 与目标点和全局规划路径的距离的，它们是 *TrajectoryPlanner* 的成员，在为路径打分时被使用。

MapGrid 是 “地图”，它包含一个由 *MapCell* 类对象数组的成员，即 *cell* 数组，地图大小为 $size\_x\_ \times size\_y\_$。(*goal_map* 和 *path_map* 都是 *MapGrid* 类实例，分别称这两张地图为目标地图和路径地图)

MapGrid 类数据成员:

```
public:
    double goal_x_, goal_y_;
    unsigned int size_x_, size_y_;
private:
    std::vector<MapCell> map_;
```

MapCell 就代表地图上的一个 *cell*，它记录 x、y 坐标 (索引)。*target_dist* 表示目标距离，它被初始化为无穷大，经过 *path_map* 和 *goal_map* 的计算后，它可以表示该 *cell* 距全局路径和目标点的距离。*target_mark* 是该点已更新过的 *flag*。*within_robot* 表示该点在机器人足迹范围内。

MapCell 类数据成员:

```
public:
    unsigned int cx, cy;
    double target_dist;
    bool target_mark;
    bool within_robot;
```

MapGrid::setTargetCells

该函数处理路径地图，计算整个地图上的所有 *cell* 的路径距离 (*path_dist*)。

首先检查路径地图的尺寸以及索引是否正确，调整全局路径的分辨率使其能够达到 *cost_map* 的分辨率。

接下来将全局路径的点转换到路径地图上，并且在路径地图上将同时也位于全局路径上的点的 *target_dist*(路径距离) 设置为 0, 即表示该点在全局路径上。

接下来调用 $MapGrid :: computeTargetDistance$ 函数，让路径地图从 $target\_dist = 0$ 的 *cell* 队列开始向四周开始传播，直至路径地图上所有的 *cell* 都计算得到 *target_dist*。

该函数的代码如下:

```
void MapGrid::setTargetCells(const costmap_2d::Costmap2D& costmap,
                             const std::vector<geometry_msgs::PoseStamped>& global_plan) {
    //检查路径地图尺寸以及索引是否正确
    sizeCheck(costmap.getSizeInCellsX(), costmap.getSizeInCellsY());

    bool started_path = false;
```

```
    //用于储存全局路径上的MapCell
    queue<MapCell*> path_dist_queue;

    std::vector<geometry_msgs::PoseStamped> adjusted_global_plan;
    //传入的全局规划是global系下的
    //调用adjustPlanResolution函数对其分辨率进行调整，使其达到costmap的分辨率
    adjustPlanResolution(global_plan, adjusted_global_plan, costmap.getResolution());
    if (adjusted_global_plan.size() != global_plan.size()) {
        ROS_DEBUG("Adjusted global plan resolution, added %zu points", adjusted_global_plan.size()
    - global_plan.size());
    }

    unsigned int i;
    //将全局路径的点转换到路径地图上
    for (i = 0; i < adjusted_global_plan.size(); ++i) {
    double g_x = adjusted_global_plan[i].pose.position.x;
    double g_y = adjusted_global_plan[i].pose.position.y;
    unsigned int map_x, map_y;
    //如果成功把一个全局规划上的点的坐标转换到地图坐标(map_x,map_y)上,且在代价地图上这一点不是未知的
    if (costmap.worldToMap(g_x, g_y, map_x, map_y) && costmap.getCost(map_x, map_y) != costmap_2d::N
O_INFORMATION) {
        MapCell& current = getCell(map_x, map_y);
        //将这个点的target_dist(到path的距离)设置为0,即在全局路径上
        //setTargetCells和setLocalGoal这两个函数:(此函数是setTargetCells)
        //    前者在“路径地图”上将全局路径点target_dist标记为0
        //    后者在“目标地图”上将目标点target_dist标记为0
        current.target_dist = 0.0;
        //标记已经计算了距离
        current.target_mark = true;
        //把该点放进path_dist_queue队列中
        path_dist_queue.push(&current);
        //标记已经开始把点转换到地图坐标
        started_path = true;
    }
    //当代价地图上这一点的代价不存在了(规划路径已经到达了代价地图的边界)
    //并且标记了已经开始转换，退出循环
    //(这个else if写的有一点迷惑性,注意首先需要不满足if条件,才会判断elseif的条件)
    else if (started_path) {
        break;
    }
    }
    //如果循环结束后,开始转换标志(started_path)还没有置位 则报错
    if (!started_path) {
        ROS_ERROR("None of the %d first of %zu (%zu) points of the global plan were in the local cos
    tmap and free", i, adjusted_global_plan.size(), global_plan.size());
        return;
```

```
    }
    //计算路径地图上的每一个cell与规划路径之间的距离
    computeTargetDistance(path_dist_queue, costmap);
}
```

MapGrid::setLocalGoal

该函数处理目标地图，计算整个地图上的所有 *cell* 的目标距离 (*goal_dist*)。

通过迭代找到全局路径的终点，即目标点，但如果迭代过程当中到达了局部规划 *costmap* 的边际或经过障碍物，立即退出迭代，将上一个有效点作为终点。该函数的逻辑与 $MapGrid :: setTargetCells$ 基本相同，不再赘述，代码如下：

```
void MapGrid::setLocalGoal(const costmap_2d::Costmap2D& costmap,
                           const std::vector<geometry_msgs::PoseStamped>& global_plan) {
    //检查地图尺寸和索引
    sizeCheck(costmap.getSizeInCellsX(), costmap.getSizeInCellsY());

    int local_goal_x = -1;
    int local_goal_y = -1;
    bool started_path = false;

    std::vector<geometry_msgs::PoseStamped> adjusted_global_plan;
    //调整分辨率
    adjustPlanResolution(global_plan, adjusted_global_plan, costmap.getResolution());

    //逐个跳过全局路径中的点,一直到碰到了局部规划costmap的边界
    //则立即退出迭代，将上一个有效点作为终点
    //否则一直迭代到全局路径的终点(目标点)
    for (unsigned int i = 0; i < adjusted_global_plan.size(); ++i) {
    double g_x = adjusted_global_plan[i].pose.position.x;
    double g_y = adjusted_global_plan[i].pose.position.y;
    unsigned int map_x, map_y;
    if (costmap.worldToMap(g_x, g_y, map_x, map_y) && costmap.getCost(map_x, map_y) != costmap_2d::N
O_INFORMATION) {
        local_goal_x = map_x;
        local_goal_y = map_y;
        started_path = true;
    } else {
        if (started_path) {
        break;
        }// else we might have a non pruned path, so we just continue
    }
    }
    if (!started_path) {
        ROS_ERROR("None of the points of the global plan were in the local costmap, global plan poin
    ts too far from robot");
        return;
```

```
    }

    queue<MapCell*> path_dist_queue;
    if (local_goal_x >= 0 && local_goal_y >= 0) {
    MapCell& current = getCell(local_goal_x, local_goal_y);
    costmap.mapToWorld(local_goal_x, local_goal_y, goal_x_, goal_y_);
    //将迭代得到的目标点对应在“目标地图”上的cell的target_dist标记为0
    //setTargetCells和setLocalGoal这两个函数:(此函数是setLocalGoal)
    //    前者在“路径地图”上将全局路径点target_dist标记为0
    //    后者在“目标地图”上将目标点target_dist标记为0
    current.target_dist = 0.0;
    current.target_mark = true;
    path_dist_queue.push(&current);
    }
    //计算目标地图上的每一个cell与规划路径之间的距离
    computeTargetDistance(path_dist_queue, costmap);
}
```

MapGrid::computeTargetDistance

该函数在 $MapGrid :: setTargetCells$ 以及 $MapGrid :: setLocalGoal$ 中被调用，用来根据传入的 $target\_dist = 0$ 的 $cell$ 队列，向四周传播，直到获得整个地图 (路径地图/目标地图) 的 $target\_dist$ 值。

函数中首先从传入的 $target\_dist = 0$ 的 $cell$ 队列开始，循环每一个 $cell$ 作为“父 $cell$”，访问它四周的 $cell$，依据“父 $cell$” 的 $target\_dist$ 值更新“子 $cell$” 的 $target\_dist$ 值，然后将“父 $cell$” 踢出队列，“子 $cell$” 加入队列，如此直至所有 $cell$ 的 $target\_dist$ 被更新完成。

该函数的代码如下：

```
void MapGrid::computeTargetDistance(queue<MapCell*>& dist_queue,
                                    const costmap_2d::Costmap2D& costmap){
    MapCell* current_cell;
    MapCell* check_cell;
    unsigned int last_col = size_x_ - 1;
    unsigned int last_row = size_y_ - 1;

    //依次将队列中的cell作为"父cell",检查"父cell"四周的cell
    //对其调用updatePathCell函数，得到该cell的值
    //如果该cell的值有效，则加入循环队列里，弹出“父cell”，四周的cell成为新的“父cell”

    //C++中的queue(FIFO):
    //  只能访问queue<T>容器适配器的第一个和最后一个元素,只能在末尾添加新元素,只能从头部移除元素

    while(!dist_queue.empty()){
        current_cell = dist_queue.front();
        //将current_cell弹出
        dist_queue.pop();
```

```
        if(current_cell->cx > 0){
            check_cell = current_cell - 1;
            if(!check_cell->target_mark){
                //标记该cell被访问过了
                check_cell->target_mark = true;
                if(updatePathCell(current_cell, check_cell, costmap)) {
                    dist_queue.push(check_cell);
                }
            }
        }
        if(current_cell->cx < last_col){
            check_cell = current_cell + 1;
            if(!check_cell->target_mark){
                check_cell->target_mark = true;
                if(updatePathCell(current_cell, check_cell, costmap)) {
                    dist_queue.push(check_cell);
                }
            }
        }
        if(current_cell->cy > 0){
            check_cell = current_cell - size_x_;
            if(!check_cell->target_mark){
                check_cell->target_mark = true;
                if(updatePathCell(current_cell, check_cell, costmap)) {
                    dist_queue.push(check_cell);
                }
            }
        }
        if(current_cell->cy < last_row){
            check_cell = current_cell + size_x_;
            if(!check_cell->target_mark){
                check_cell->target_mark = true;
                if(updatePathCell(current_cell, check_cell, costmap)) {
                    dist_queue.push(check_cell);
                }
            }
        }
        //直到地图上所有cell的dist值都被计算出来
    }
}
```

1.1.7 CostmapModel 类

CostmapModel 类派生自 WorldModel 类，在 TrajectoryPlanner 中被使用，承担局部规划器与局部规划 Costmap 之间的桥梁工作。

这个类帮助局部规划器在 Costmap 上进行计算，footprintCost、lineCost、pointCost 三个类方法分别能通过 Costmap 计算出机器人足迹范围、两个 cell 连线、单个 cell 的代价，并将值返回给局部规划器。

CostmapModel::lineCost

该函数计算两点连线的代价，两点连线的代价就是线上点的最大代价，注意函数中使用 LineIterator 迭代器对线上的每一点进行迭代。代码如下：

```
double CostmapModel::lineCost(int x0, int x1, int y0, int y1) const {
    double line_cost = 0.0;
    double point_cost = -1.0;

    for(LineIterator line( x0, y0, x1, y1 ); line.isValid(); line.advance()){
        //LineIterator类的advance函数就是取线上的下一个点,然后getX和getY函数获取点的坐标
        point_cost = pointCost(line.getX(), line.getY());

        if(point_cost < 0)
            return point_cost;

        //两点连线的代价就是线上点的最大代价?
        if(line_cost < point_cost)
            line_cost = point_cost;
    }
    return line_cost;
  }
```

CostmapModel::footprintCost

该函数计算机器人足迹的代价（考虑机器人的外围形状）。该函数根据机器人的足迹包含的 cell 个数，分为两种情况：

- 如果脚印点数小于三，默认机器人形状为圆形，不考虑脚印，只考虑中心，返回中心 cell 的代价;
- 如果脚印点数大于三，需要考虑机器人的形状，把足迹视为多边形，循环调用 lineCost 计算多边形各边的 cell，注意首尾闭合，最后返回代价。

该函数的返回值有四种情况：

- -1.0 ： 覆盖至少一个障碍 cell
- -2.0 ： 覆盖至少一个未知 cell
- -3.0 ： 不在地图上
- 其他正 cost

该函数的代码如下:

```
double CostmapModel::footprintCost(const geometry_msgs::Point& position,
                                   const std::vector<geometry_msgs::Point>& footprint,
                                   double inscribed_radius, double circumscribed_radius){
    unsigned int cell_x, cell_y;

    //获取机器人中心点的cell坐标，存放在cell_x cell_y中
    //如果得不到坐标，说明不在地图上，直接返回-3
    if(!costmap_.worldToMap(position.x, position.y, cell_x, cell_y))
        return -3.0;

    //如果脚印点数小于三，默认机器人形状为圆形，不考虑脚印，只考虑中心
    if(footprint.size() < 3){
        unsigned char cost = costmap_.getCost(cell_x, cell_y);
        //如果中心位于未知代价的cell上，返回-2
        if(cost == NO_INFORMATION)
            return -2.0;
        //如果中心位于致命障碍cell上，返回-1（这几个宏是在costvalue中定义的，是灰度值）
        if(cost == LETHAL_OBSTACLE || cost == INSCRIBED_INFLATED_OBSTACLE)
            return -1.0;
        //如果机器人位置既不是未知也不是致命，返回它的代价
        return cost;
    }

    //如果脚印点数大于三，需要考虑机器人的形状，把足迹视为多边形
    unsigned int x0, x1, y0, y1;
    double line_cost = 0.0;
    double footprint_cost = 0.0;

    //we need to rasterize each line in the footprint
    for(unsigned int i = 0; i < footprint.size() - 1; ++i){
        //获取第一个点的cell坐标
        if(!costmap_.worldToMap(footprint[i].x, footprint[i].y, x0, y0))
            return -3.0;
        //获取第二个点的cell坐标
        if(!costmap_.worldToMap(footprint[i + 1].x, footprint[i + 1].y, x1, y1))
            return -3.0;

        //得到两点连线的代价
        line_cost = lineCost(x0, x1, y0, y1);
        footprint_cost = std::max(line_cost, footprint_cost);

        //如果某条边缘线段代价<0(碰到了障碍)，直接停止生成代价，返回这个负代价
        if(line_cost < 0)
            return line_cost;
    }
```

```
    //再把footprint的最后一个点和第一个点连起来，形成封闭图形
    if(!costmap_.worldToMap(footprint.back().x, footprint.back().y, x0, y0))
        return -3.0;
    if(!costmap_.worldToMap(footprint.front().x, footprint.front().y, x1, y1))
        return -3.0;
    line_cost = lineCost(x0, x1, y0, y1);
    footprint_cost = std::max(line_cost, footprint_cost);
    if(line_cost < 0)
        return line_cost;

    //如果所有边缘线的代价都是合法的，那么返回足迹的代价
    return footprint_cost;
}
```