

# 运动规划课程文档

---

## 运动规划课程文档

### 第一章 运动规划和ROS导航模块简介

1.0 机器人智能导航和自动驾驶技术栈

#### 1.1 运动规划的简介

1.1.1 运动规划的分类：

全局规划和局部规划

离线路径规划和在线路径规划

1.1.2 运动规划的经典算法

#### 1.2 【公开直播】运动规划相关论文详细解读及答疑

相关论文介绍

论文一：Robust Navigation in Indoor Office Environment

论文二：Dynamic Window Approach to Collision Avoidance

论文三：Layered Costmap for Context-Sensitive Navigation

论文四：Trajectory Modification Considering Dynamic Constraints of Autonomous Robots

#### 1.3 ROS的简介

1.3.1 什么是ROS

1.3.2 ROS的特点

1.3.3 如果不用ROS:

1.3.4 在运动规划方面 ROS提供的便利

1.3.5 ROS 的发行版本

#### 1.4 Navigation的介绍

1.4.1 Navigation仓库的软件包

1.4.2 Navigation Stack 框架

#### 1.5 运动规划的Demo示例

视频中的操作步骤：

## 第二章 ROS的核心内容和仿真

### 2.1 ROS的核心内容和常用操作

通信机制

通信方式

### 2.2 ROS中常用的工具

2.3 Turtlesim的控制仿真

2.3.2 Stage仿真：使用TEB的差速模型导航

2.4 Gazebo仿真：turtlebot3的导航

## 第三章 导航框架Navigation (Part 1)

### 3.1 机器人的运动学

URDF <http://wiki.ros.org/urdf>

Navigation2 框架 <https://navigation.ros.org/>

### 3.3 nav\_core 源码解析

### 3.3 move\_base 源码解析

move\_base的作用

movebase状态之间（PLANNING, CONTROLLING, CLEARING）的切换条件

rviz下发任务后的数据流

恢复行为

恢复行为的使用场景

## 第四章 导航框架Navigation Stack学习 (Part 2)

4.1 costmap\_2d代价地图源码

Costmap\_2d框架：

Layer类

LayeredCostmap基类

costmap2D基类

CostmapLayer类  
StaticLayer类  
ObstacleLayer类  
Inflationlayer不维护专门的costmap

重要函数

    重要结构体和类:

## 第五章 全局规划算法

全局规划器 理论知识  
常用的 ROS Global Planner功能包  
架构-文件系统  
    planner\_core  
    参数调整  
    调用流程  
程序解读  
    sbpl\_lattice\_planner 介绍  
    Tutorials  
推荐相关文献:

## 第六章 局部规划器算法模块和代码讲解 (Part 1)

文件说明:

打分相关的:

## 第七章 局部规划器算法模块和代码讲解 (Part 2)

## 第八章 TEB局部规划器介

8.1 TEB规划效果Demo  
8.2 论文带读  
8.3 TEB 与 DWA 的比较  
8.4 TEB和相关tutorials的安装和使用  
    8.4.1  
    8.4.2 Teb\_local\_planner\_toturial 讲解  
8.6 costmap\_converter

## 第九章 TEB 源码分析 (Part 1)

9.1 在Move\_base中配置TEB  
    预习资料  
    diff\_drive  
    carlike  
9.2 TEB 源码架构分析  
9.3 TEB与ROS的接口  
9.4 优化轨迹的TebOptimalPlanner类  
9.5 时变弹性带TimedElasticBand

## 第十章 TEB 源码分析 (Part 2)

预备准备  
teb中主要9个目标函数  
其他材料

## 第十一章 ROS2 和 Navigation2

11.1 ROS1和ROS2  
    ROS2官网  
    ROS1 和 ROS2区别:  
11.2 行为树和状态机  
    movebase状态之间 (PLANNING, CONTROLLING, CLEARING ) 的切换条件  
11.3 lifecycle manager  
nav2.lifecycle\_manager  
11.4 planner server和controller server  
    action\_server

## 第十二章 总结和展望

12.1 移动机器人应用场景  
12.2 自动驾驶相关平台介绍  
    ROS在自动驾驶上的问题?  
    Apollo架构  
    Autoware架构

### 12.3 其他的规划算法

12.4 Regulated\_pure\_pursuit\_controller

12.5 关于navigation工程的个人看法

movebase状态之间 (PLANNING, CONTROLLING, CLEARING) 的切换条件

### 附录：安装环境

1. 安装Ubuntu系统

2. ROS系统安装

3. Navigation安装

4. Teb安装

### Docker

Docker是什么?

Docker 的优点 参考: <https://www.runoob.com/docker/docker-tutorial.html>

Docker的三个概念 (参考<https://zhuanlan.zhihu.com/p/23599229>)

### 安装docker

在ubuntu中 安装docker

去Docker Hub拉取ros的docker镜像

容器脚本

用镜像建立容器

在docker容器中，安装其他需要的ros包

安装navigation 和rviz

安装gazebo的接口相关模块

安装turtlebot的相关模块

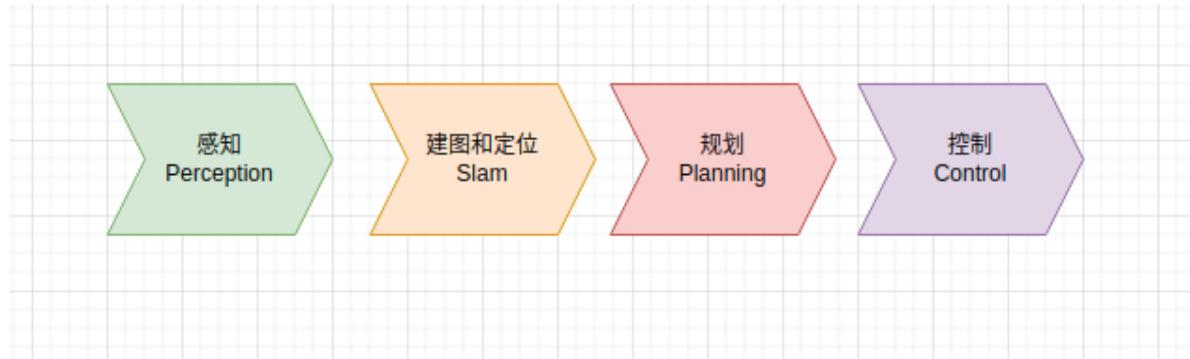


扫描学习SLAM、三维重建

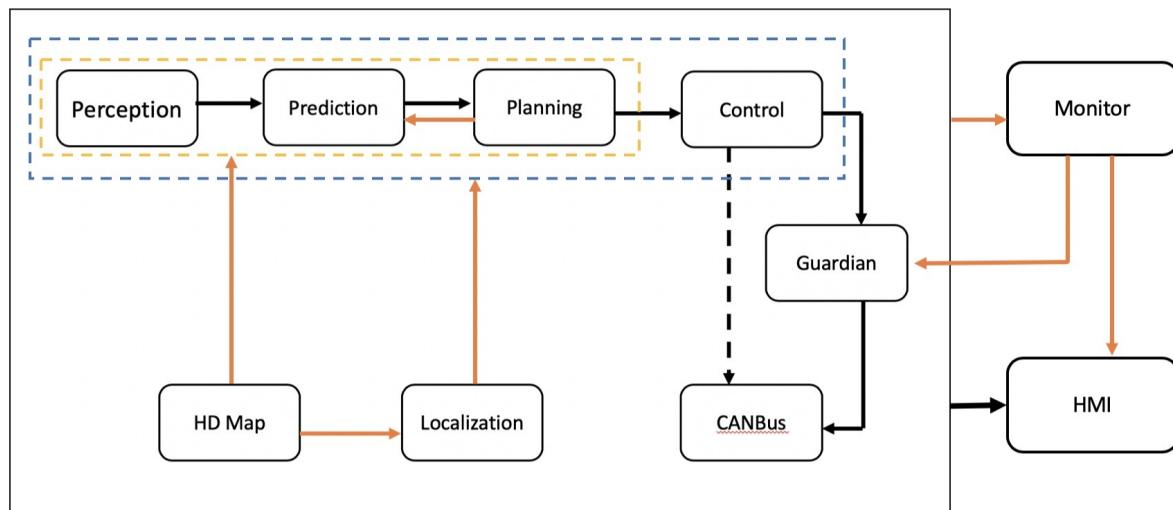
# 第一章 运动规划和ROS导航模块简介

## 1.0 机器人智能导航和自动驾驶技术栈

在软件层面机器人和自动驾驶技术基本是相通的。其中软件算法模块可以分成了感知、定位、建图、规划和控制，软件算法与感知单元采集的数据配合，完成机器人的各种功能。



自动驾驶的平台 Apollo , Autoware...



Apollo Software Overview - Navigation Mode

### 1.1 运动规划的简介

**运动规划** (Motion Planning) 是一个过程，用来寻找从起始状态Start到目标状态Goal的移动步骤。常需要在运动受到约束的条件下找到最优解。例如，一个清洁机器人在楼层里打扫，它不能撞墙，也不能从楼梯口掉下去。给定一个任务，运动规划算法计算出一个动作序列，告诉机器人前进多少米，然后左转右转多少度。如果机器人要操作物品或者探测未知地形等等，运动规划就变得很复杂了。

### 1.1.1 运动规划的分类：

**路径规划** (path planning)：在规定范围内的区域，连接起点状态和终点状态的序列点或曲线称之为路径，构成路径的策略称之为路径规划。一般路径规划的目标是规划出无碰撞的路径同时路径的长度尽量短。

**轨迹规划** (trajectory planning)：在路径规划的基础上加入时间序列信息，会考虑速度和运动学约束尽量按照规划路径进行。

#### 路径规划和轨迹规划的区别：

路径规划只有几何属性，与时间无关。

轨迹规划时间会考虑时间，每个时刻对应有位置、朝向甚至速度、加速度等变量。

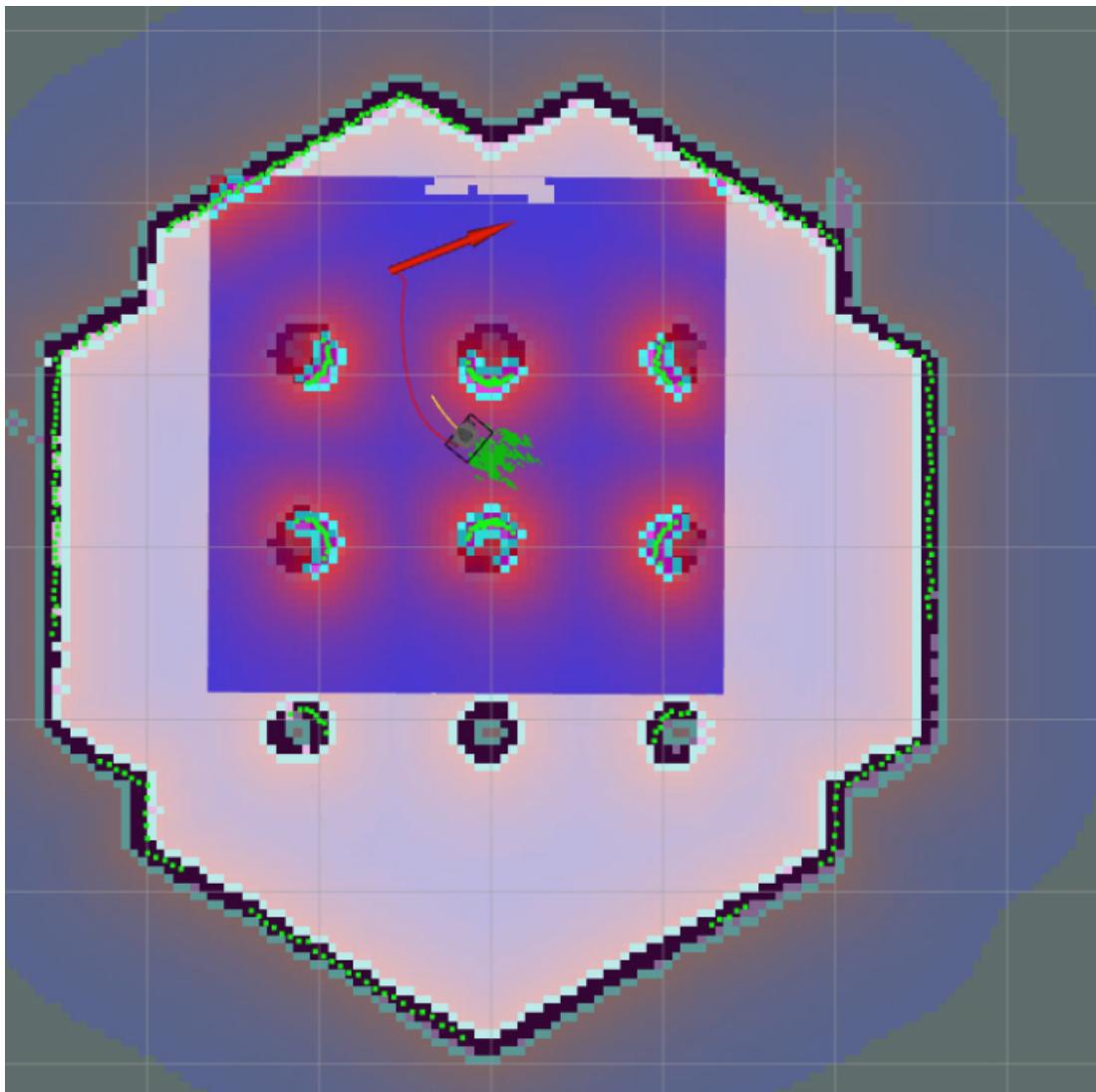
## 全局规划和局部规划

- 全局路径规划

已知地图信息，根据给定的目标状态进行总体路径的规划。规划出的路径有一系列的pose ( $x, y, \theta$ ) 组成。

- 局部轨迹规划

地图信息未知或者部分可知，根据附近的障碍物信息进行避障轨迹的规划。



## 离线路径规划和在线路径规划

离线路径规划是在环境先验信息上进行的。完整的先验信息只能适用于静态环境。

在线路径规划是基于传感器信息的不确定环境的路径规划，轨迹需要在线规划。

### 1.1.2 运动规划的经典算法

运动规划算法分类方式	算法
基于采样	PRM (概率路线图) , RRT(快速随机搜索树) DWA (动态窗口)
基于优化	TEB
基于图搜索	Dijkstra, A star, DFS, BFS
基于人工势场	人工势场法
基于曲线拟合	直线, 圆弧, 多项式曲线, 贝塞尔曲线, 羊角曲线

## 1.2 【公开直播】运动规划相关论文详细解读及答疑

---

### 相关论文介绍

本次直播将带读四篇论文，介绍运动规划的内容和相关的算法论文。

#### 论文一：Robust Navigation in Indoor Office Environment

- 运动规划中具体的应用场景和框架
- 导航的任务和挑战。

#### 论文二：Dynamic Window Approach to Collision Avoidance

- 局部路径规划算法DWA的算法流程介绍

#### 论文三：Layered Costmap for Context-Sensitive Navigation

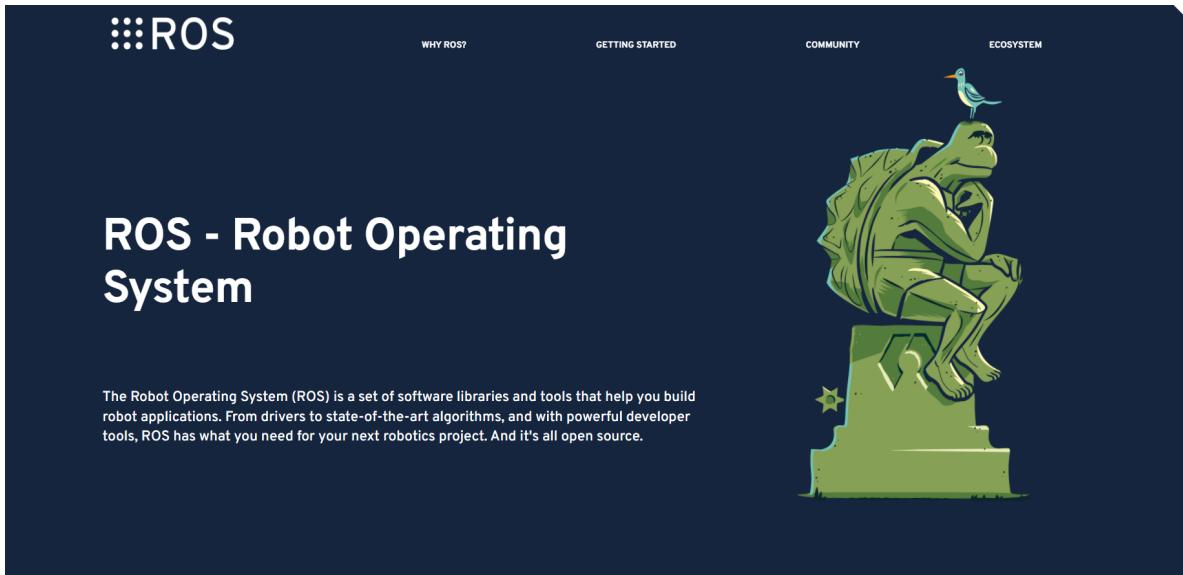
- 为什么用layered Costmap

#### 论文四：Trajectory Modification Considering Dynamic Constraints of Autonomous Robots

- 局部轨迹规划算法Timed elastic band 的介绍

## 1.3 ROS的简介

ROS官网：<https://www.ros.org/>



### 1.3.1 什么是ROS

ROS是Willow Garage公司发布的开源机器人操作系统，目前是主流的机器人操作系统，它提供了一系列的软件库（有许多开源的算法实现）和工具帮助开发者构建机器人的应用。它提供了操作系统应有的服务，包括

- 硬件抽象
- 底层设备控制
- 常用函数的实现
- 进程间消息传递
- 以及包管理。

它也提供用于获取、编译、编写、和跨计算机运行代码所需的工具和库函数。

### 1.3.2 ROS的特点

ROS的主要目标是为机器人研究和开发提供代码**复用**的支持。

- ROS是一个分布式的进程（也就是节点）框架，点对点设计，不同的功能放在不同的节点开发。
- 可以使一个工程的开发和实现从文件系统到用户接口完全独立决策
- 多语言支持，支持c++, python等等
- 海量工具包和第三方库支持
- 免费和算法开源

总的来说，ROS提供了一系列解决方案，从通信机制，开发工具，应用功能和生态系统。

### 1.3.3 如果不用ROS:

- 传感器数据的交互
- 不同的模块怎么开发
- 可视化怎么办
- 接口不一样，怎么避免重新开发...

### 1.3.4 在运动规划方面 ROS提供的便利

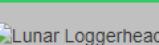
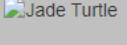
- 不用自己搭建机器人模型，有许多种机器人可供选择 Turtlebot
- Stage
- Gazebo
- Navigation stack
- ...

### 1.3.5 ROS 的发行版本

ROS1: Kinetic, Melodic, Noetic ...

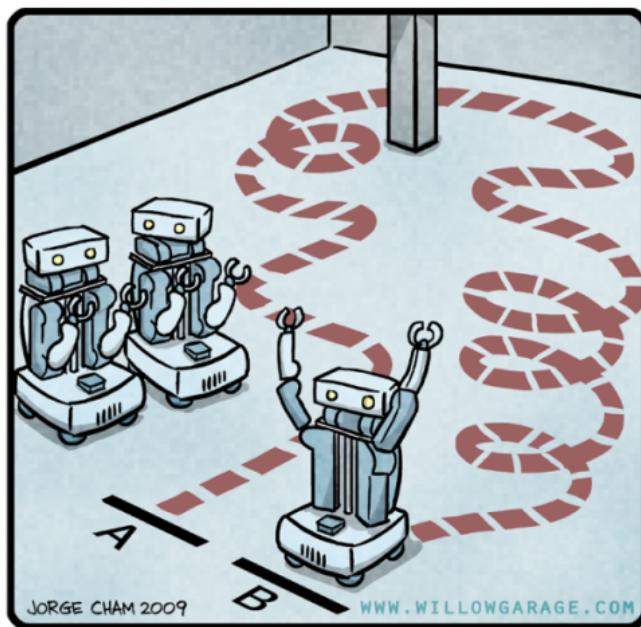
ROS2: Dashing, Galactic ...

## 3. List of Distributions

Distro	Release date	Poster	Tuturtle, turtle in tutorial	EOL date
ROS Noetic Ninjems <b>(Recommended)</b>	May 23rd, 2020	 Noetic Ninjems	 <a href="https://raw.githubusercontent.com/ros/ros_tutorials/noetic-devel/turtlesim/images/noetic.png">https://raw.githubusercontent.com/ros/ros_tutorials/noetic-devel/turtlesim/images/noetic.png</a>	May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018	 Melodic Morenia	 Melodic Morenia	May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017	 Lunar Loggerhead	 Lunar Loggerhead	May, 2019
ROS Kinetic Kame	May 23rd, 2016	 Kinetic Kame	 Kinetic Kame	April, 2021 (Xenial EOL)
ROS Jade Turtle	May 23rd, 2015	 JADE TURTLE ROS	 Jade Turtle	May, 2017
ROS Indigo Igloo	July 22nd, 2014	 INDIGO IGLOO	 i-turtle	April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013	 HYDRO MEDUSA	 H-turtle	May, 2015

## 1.4 Navigation的介绍

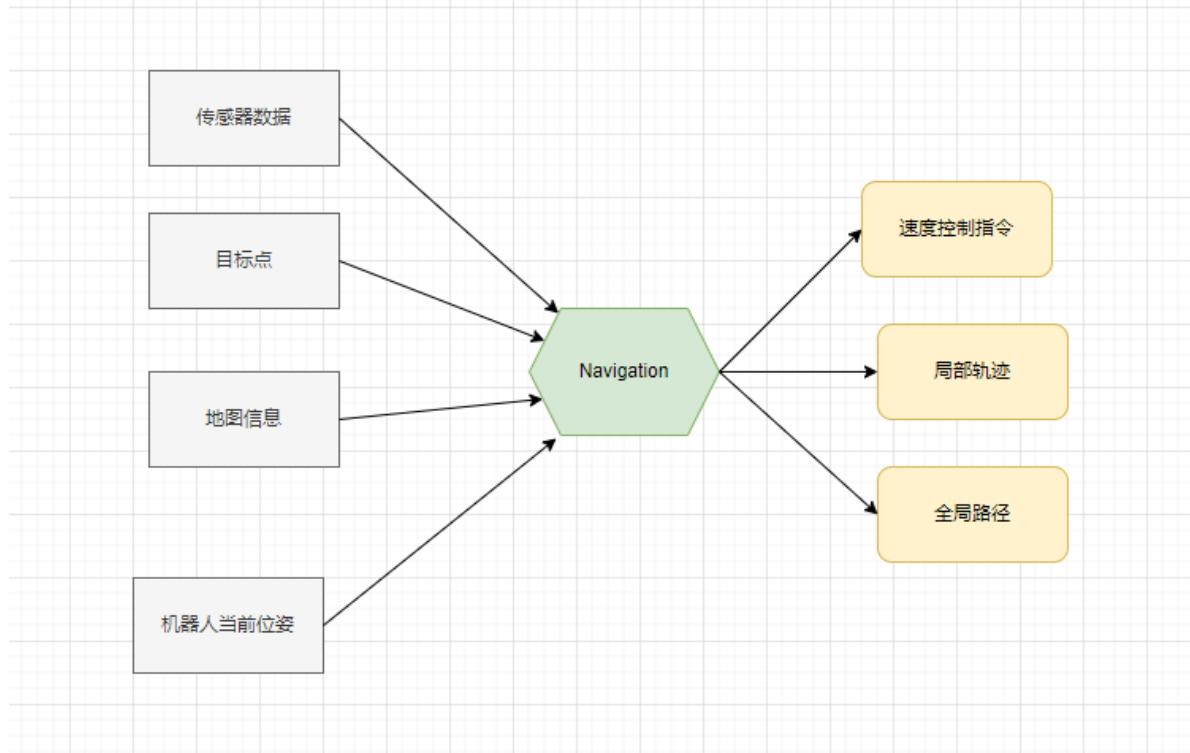
## R.O.B.O.T. Comics



"HIS PATH-PLANNING MAY BE  
SUB-OPTIMAL, BUT IT'S GOT FLAIR."

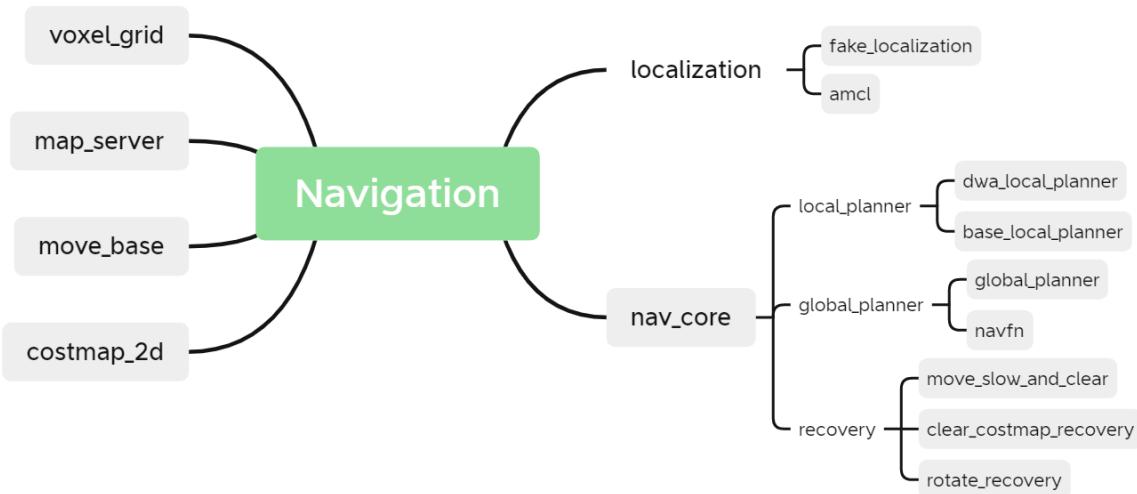
Navigation是ROS提供的二维导航包。

导航模块的作用：输入传感器信息、地图信息、机器人位姿以及目标点，计算并且输出安全可靠的速度控制指令（是对机器人的底座下方控制命令），全局路径和局部轨迹。



### 1.4.1 Navigation仓库的软件包

#### Navigation软件包的架构



#### Navigation各个子模块的功能

nav\_core: 存放全局规划器、局部路径规划和恢复行为的通用接口

move\_base: 整合导航的各个模块，实现导航功能。

map\_server: 地图服务器，保存地图和导入地图。

costmap\_2d: 代价地图，为全局和局部规划提供障碍物信息。

定位:

- fake\_localization : 一般是仿真用的定位
- amcl: 蒙特卡洛定位

全局规划:

- global\_planner : 提供dijkstra和A\*算法
- navfn : 提供dijkstra和A\*算法

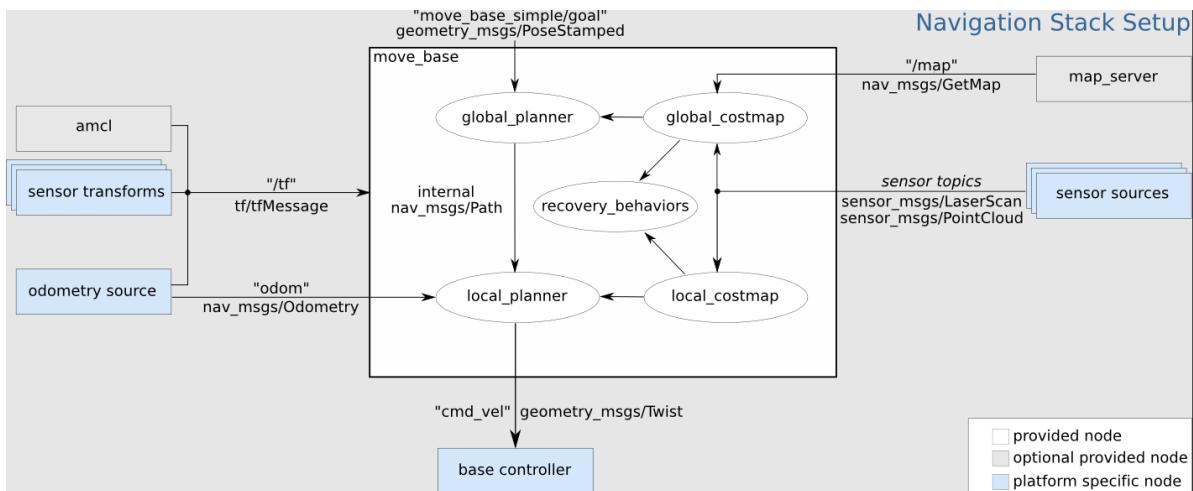
局部规划:

- dwa\_local\_planner: 动态窗口法
- base\_local\_planner: trajectory\_planner 轨迹规划器

恢复行为:

- move\_slow\_and\_clear : 清除代价地图中信息并且限制机器人移动速度
- clear\_costmap\_recovery: 清理代价地图
- rotate\_recovery: 旋转, 清理代价地图

## 1.4.2 Navigation Stack 框架



## 1.5 运动规划的Demo示例

### 视频中的操作步骤：

视频演示时，环境和代码已经安装好了。如果没有安装好，就按附录的步骤安装。

#### 1.5.1 开启仿真和建图：

来到工作地址，编译：

```
cd ~/catkin_ws/  
catkin_make
```

打开第一个窗口启动仿真：

```
source devel/setup.bash  
export TURTLEBOT3_MODEL=burger  
roslaunch turtlebot3_gazebo turtlebot3_world.launch # 启动仿真
```

打开第二个窗口，开始建图：

```
source devel/setup.bash  
export TURTLEBOT3_MODEL=burger  
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

打开第三个terminal，启动键盘

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

走了一圈后，关闭键盘。

保存地图（视频中的命令敲错了，不是rosrun map\_server map\_server.....）

```
mkdir -p ~/Desktop/map  
rosrun map_server map_saver -f ~/Desktop/map
```

map\_saver将地图数据写入map.pgm和map.yaml。使用-f选项为输出文件提供不同的基本名称

### 1.5.2 建好地图后开始导航：

在第一个窗口启动仿真：

```
source devel/setup.bash  
export TURTLEBOT3_MODEL=burger  
roslaunch turtlebot3_gazebo turtlebot3_world.launch # 启动仿真
```

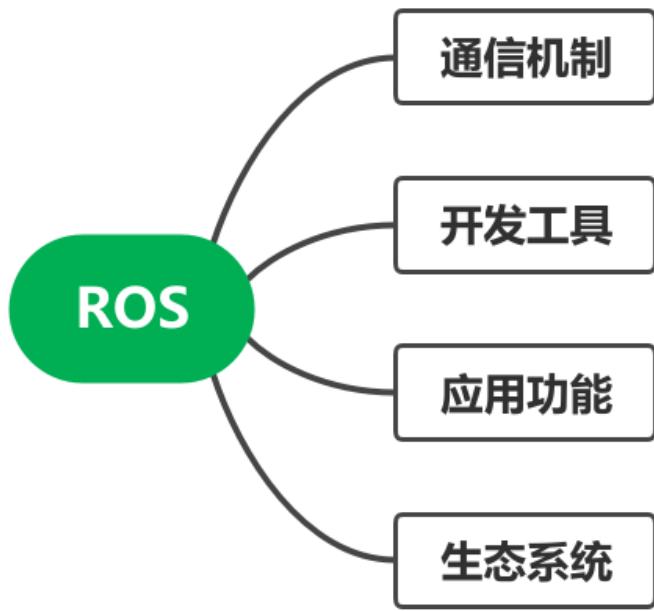
在第二个窗口开启导航，并加载刚刚建好的地图：

```
roslaunch turtlebot3_navigation turtlebot3_navigation.launch  
map_file:=~/Desktop/map/map.yaml
```

## 第二章 ROS的核心内容和仿真

---

### 2.1 ROS的核心内容和常用操作



## 通信机制

### ROS节点管理器

一个机器人控制系统由许多节点组成,各执行不同的功能。比如有的激光数据,有的处理定位信息,有的运动规划。

其中最重要的是节点管理器 (Master) 能够帮助节点找到彼此。节点通过与节点管理器通信来报告他们的注册信息。

<code>roscore</code>	开启Master(ROS名称服务)
<code>rosrun</code>	运行单个节点
<code>roslaunch</code>	运行多个节点及设置运行选项

## 通信方式

Topic (话题) : 发送/接收模式

话题消息通信中发布者或订阅者会一直工作直到被终止。

Service (服务) : 请求/响应模式

与topic不同, service是一次性的消息通信。需要提出request请求, 服务端才会响应, response返回对应消息。

Action (动作) :

类似于service的请求/响应通讯机制, 但是还有反馈机制, 用来不断向客户端反馈任务的进度, 还可以中途止运行。

## 2.2 ROS中常用的工具

- rqt
- rqt\_graph
- rviz

## 2.3 Turtlesim的控制仿真

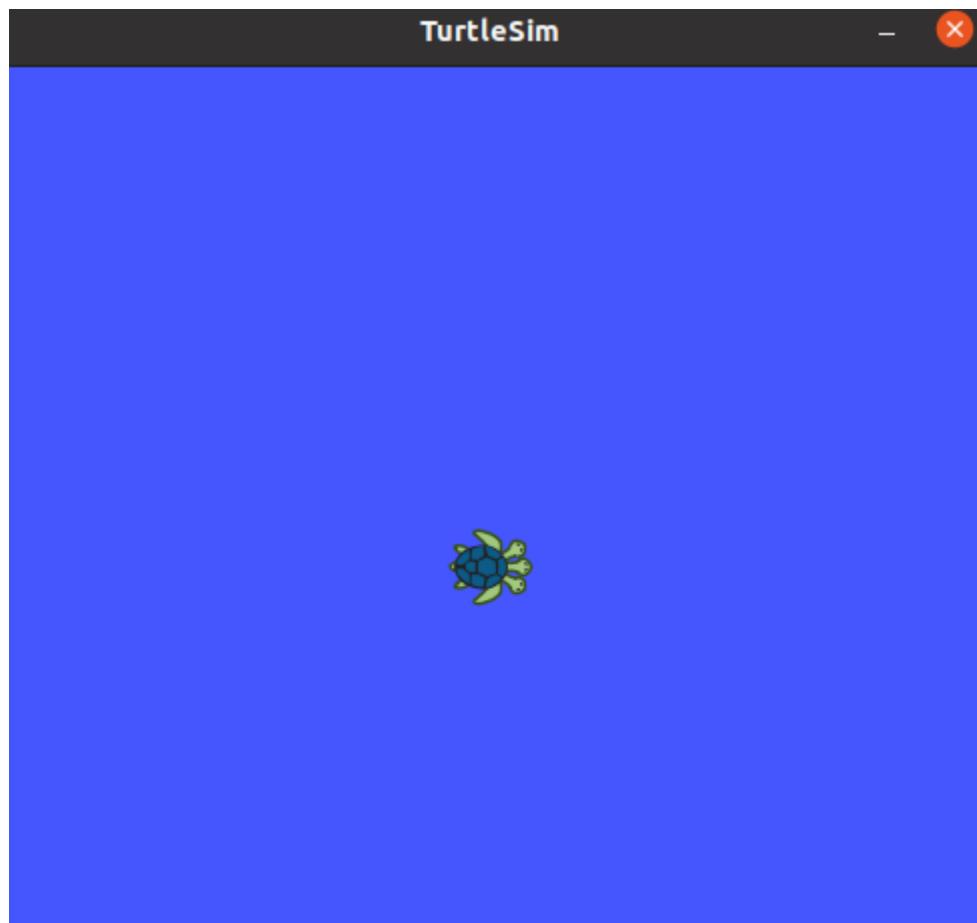
```
sudo apt-get install ros-noetic-turtlesim
```

第一个terminal, 输入

```
roscore
```

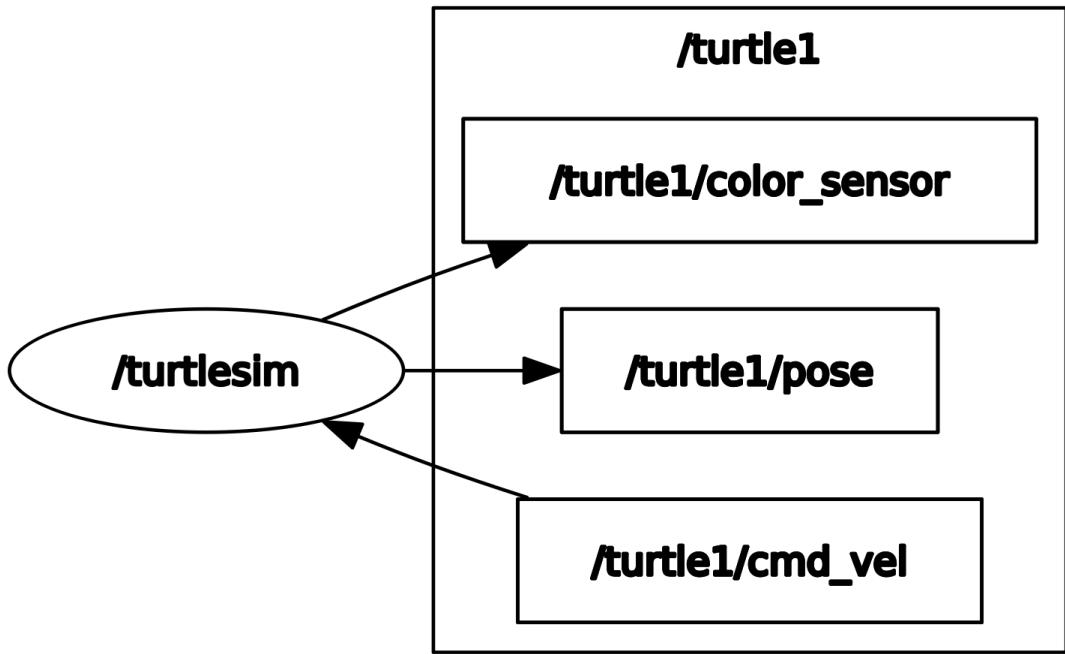
第二个terminal

```
rosrun turtlesim turtlesim_node
```



第三个terminal

rqt\_graph



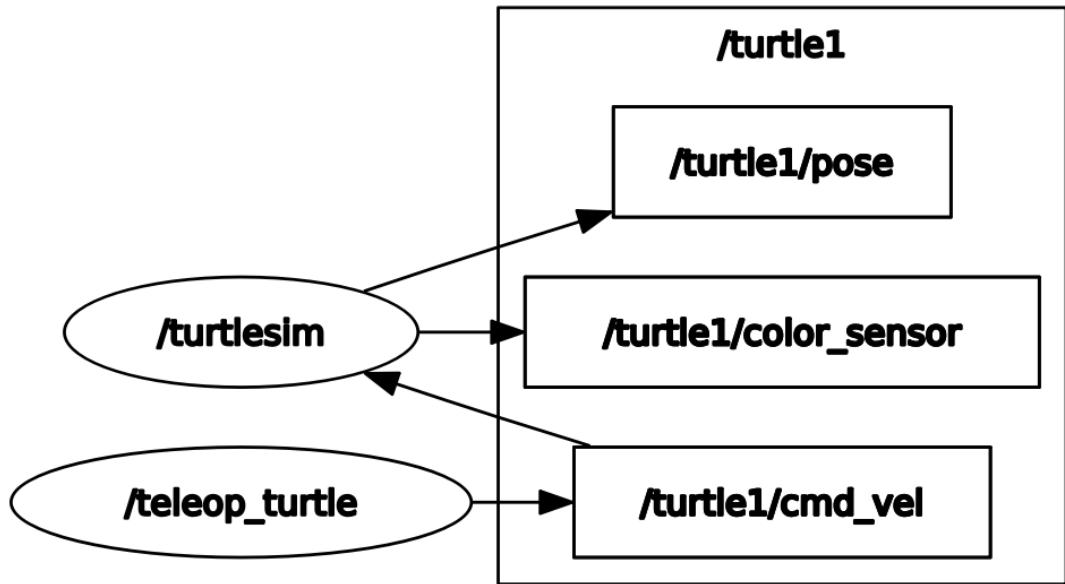
rostopic list 命令查看话题

```
user:/home/user/catkin_ws# rostopic list
/roslaunch
/roslaunch_agg
/statistics
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

键盘控制，输入下面命令后 按上下左右键进行控制。

```
rosrun turtlesim turtle_teleop_key
```

再次rosgraph就是



按键盘控制，查看速度消息

```

user:/home/user/catkin_ws# rostopic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
  ...

```

### 2.3.2 Stage仿真：使用TEB的差速模型导航

### 2.4 Gazebo仿真：turtlebot3的导航

三维仿真器，仿真中可以添加惯性，碰撞，重力，阻力等属性。

## 第三章 导航框架Navigation (Part 1)

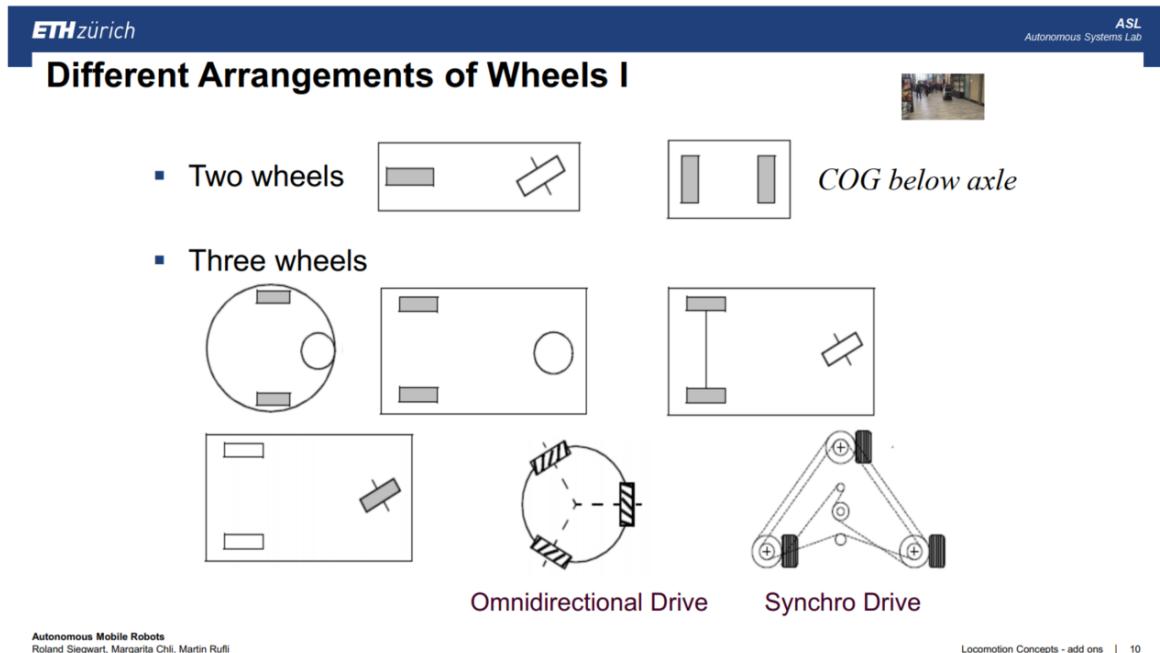
### 3.1 机器人的运动学

运动学：主要关注运动本身，即速度、加速度和空间位置这几个量之间的大小和方向关系。

动力学：要产生运动，应该对物体怎么产生力

二者关系：运动学描述物体怎么运动，动力学解释这样运动的原因

[Microsoft PowerPoint - 3 ETH Lecture Mobile Robots Kinematics add ons 2017 RS.pptx](#)

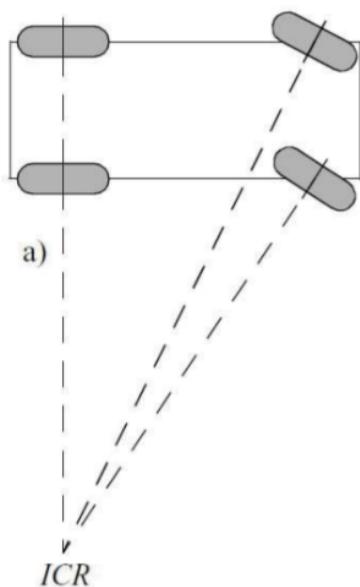


差速模型：

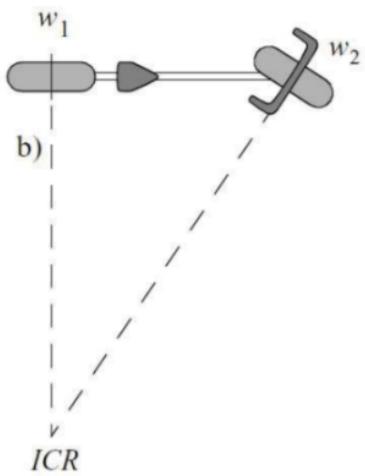
Ackerman模型

## Instantaneous Center of Rotation

Car-like Ackerman steering:

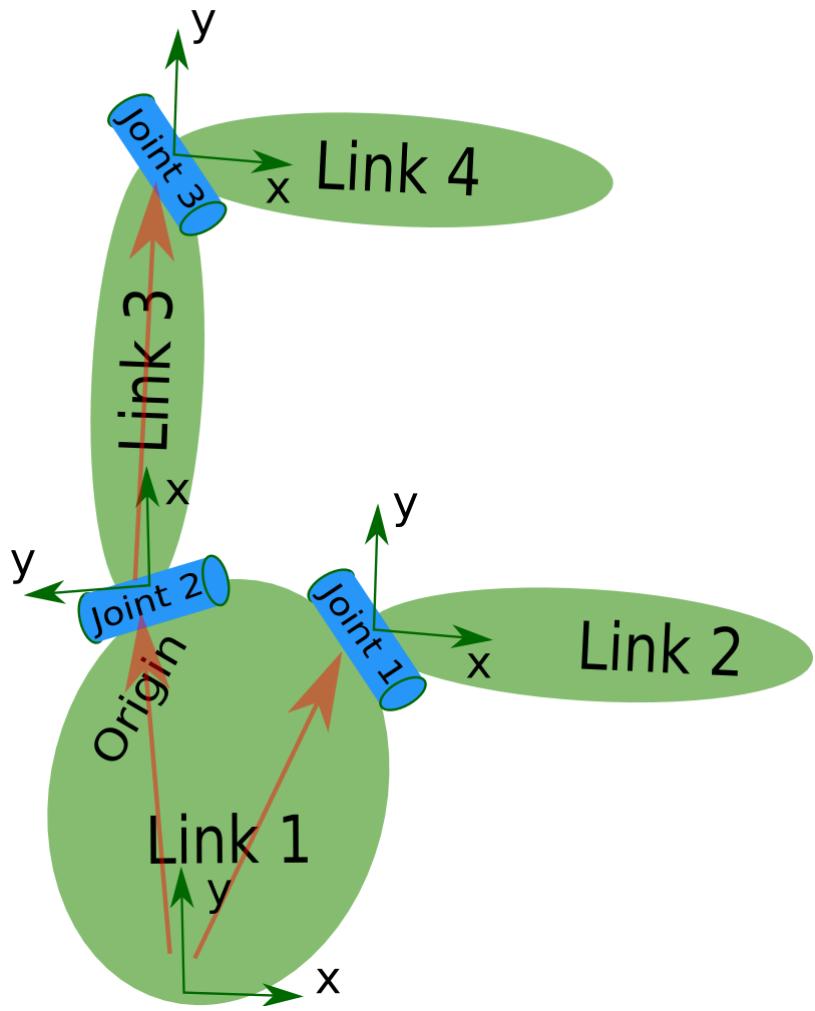


Bicycle:



URDF <http://wiki.ros.org/urdf>

URDF全称 (United Robotics Description Format) 统一机器人描述格式，是一个XML语法框架下用来描述机器人的语言格式. 将真实世界的机器人映射到计算机上。



```

<robot name="test_robot">
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
  </joint>

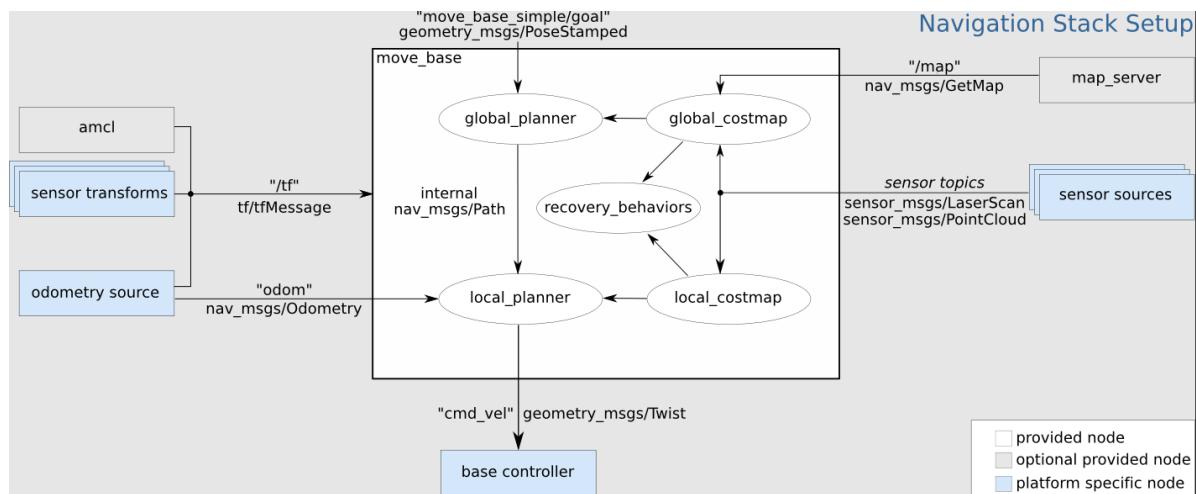
  <joint name="joint2" type="continuous">
    <parent link="link1"/>
    <child link="link3"/>
  </joint>

  <joint name="joint3" type="continuous">
    <parent link="link3"/>
    <child link="link4"/>
  </joint>
</robot>
```

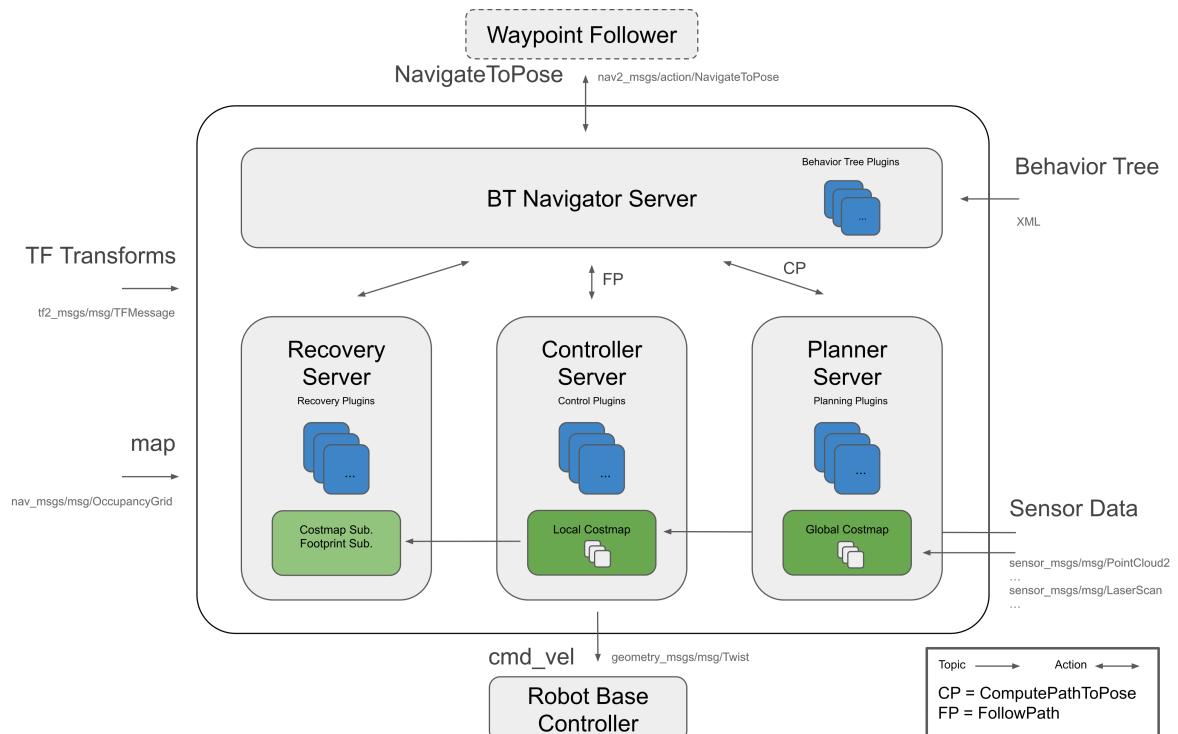
XACRO文件和URDF实质上是等价的。XACRO格式提供了一些更高级的方式来组织编辑机器人描述。

### 3.2 Navigation Stack和Navigation2框架

Navigation的框架 <http://wiki.ros.org/navigation>



Navigation2 框架 <https://navigation.ros.org/>



### 3.3 nav\_core 源码解析

BaseGlobalPlanner类，在实现全局规划器时需继承自该类。路径规划的实现在makePlan中，子类会继承再重写。

```
class BaseGlobalPlanner{
public:
    virtual bool makePlan(const geometry_msgs::PoseStamped& start,
                          const geometry_msgs::PoseStamped& goal,
                          std::vector<geometry_msgs::PoseStamped>& plan) = 0;
    virtual bool makePlan(const geometry_msgs::PoseStamped& start,
                          const geometry_msgs::PoseStamped& goal,
                          std::vector<geometry_msgs::PoseStamped>& plan,
                          double& cost)
    {
        cost = 0;
        return makePlan(start, goal, plan);
    }
    virtual void initialize(std::string name, costmap_2d::Costmap2DROS*
costmap_ros) = 0;
};
```

BaseLocalPlanner类，在实现局部规划器时需继承自该类。局部轨迹规划在，子类会继承再重写

```
class BaseLocalPlanner{
public:
    virtual bool computeVelocityCommands(geometry_msgs::Twist& cmd_vel) = 0;
    virtual bool isGoalReached() = 0;
    virtual bool setPlan(const std::vector<geometry_msgs::PoseStamped>& plan)
= 0;
    virtual void initialize(std::string name, tf2_ros::Buffer* tf,
costmap_2d::Costmap2DROS* costmap_ros) = 0;
};
```

RecoveryBehavior类，恢复行为继承的类。

```
class RecoveryBehavior{
public:
    virtual void initialize(std::string name, tf2_ros::Buffer* tf,
costmap_2d::Costmap2DROS* global_costmap, costmap_2d::Costmap2DROS*
local_costmap) = 0;
    virtual void runBehavior() = 0;
};
```

### 3.3 move\_base 源码解析

首先了解一下ROS 中的action

#### [ROS中关于Actionlib的介绍](#)

ROS Service会有阻塞的效果，程序无法进行其它的工作，机器人有时需要同时进行多个任务。

Action的通信方式则可以让程序的非阻塞执行。

#### Action server

向ROS系统广播指定action的Node，其它Node可以向该Node发出action目标请求

#### Action client

发出action目标请求的Node

#### Action通信的特点为：

1. Action是类似于Service的通信机制，也是一种请求响应机制的通信方式，ROS的action通信通过Actionlib库实现
2. Action主要弥补了service通信的一个不足，就是当机器人执行一个长时间的任务时，假如利用service通信方式，那么publisher会很长时间收不到反馈的reply，致使通信受阻。
3. Action适合实现长时间的通信过程，且可以随时查看过程进度，也可以终止请求

#### Action通信的原理为：

1. Action的工作原理是client-server模式，也是一个双向的通信模式
2. 通信双方在ROS Action Protocol下通过消息进行数据的交流通信
3. client和server为用户提供一个简单的API在客户端请求目标或在服务器端通过函数调用和回调来执行目标

下图是ROS Node Action通信的原理示意图，我们可以看到通信双方通过ROS Action Protocol实现通信。

### move\_base的作用

movebase是navigation中非常基础的一个包，实现了从机器人起始位置到目标位置的路径规划和运动控制的功能。它融合了许多的接口。

- global\_planner： 实现路径规划
- local\_planner： 实现轨迹规划和速度控制
- local\_costmap： 用于避障
- global\_costmap： 用于路径规划
- recovery： 导航过程中的异常处理

这几个成员作为插件被move\_base调用。

### movebase中重要的成员函数

bgp\_loader\_ 加载 BaseGlobalPlanner插件

bip\_loader\_ 加载 BaseLocalPlanner插件

recovery\_loader\_ 加载 RecoveryBehavior插件

planner\_costmap\_ros\_ : 全局代价地图指针

controller\_costmap\_ros\_ : 局部代价地图指针

## movebase中重要的函数

```
// 实现局部轨迹规划以及运动控制，被executeCb调用
bool executeCycle(geometry_msgs::PoseStamped& goal);
// 实现运动规划，进行任务调度策略
void executeCb(const move_base_msgs::MoveBaseGoalConstPtr& move_base_goal);
// 产生全局路径
bool makePlan(const geometry_msgs::PoseStamped& goal,
std::vector<geometry_msgs::PoseStamped>& plan);
// 专门开个线程用于路径规划
void planThread();
```

## MoveBase构造函数中：

```
as_ = new MoveBaseActionServer(ros::NodeHandle(), "move_base",
boost::bind(&MoveBase::executeCb, this, _1), false);
```

订阅的话题：

```
ros::NodeHandle simple_nh("move_base_simple");
goal_sub_ = simple_nh.subscribe<geometry_msgs::PoseStamped>("goal", 1,
boost::bind(&MoveBase::goalCB, this, _1));
```

发布的话题：

```
vel_pub_ = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);
current_goal_pub_ = private_nh.advertise<geometry_msgs::PoseStamped>
("current_goal", 0);

action_goal_pub_ = action_nh.advertise<move_base_msgs::MoveBaseActionGoal>
("goal", 1);
recovery_status_pub_= action_nh.advertise<move_base_msgs::RecoveryStatus>
("recovery_status", 1);
```

提供的service:

```
make_plan_srv_ = private_nh.advertiseService("make_plan",
&MoveBase::planService, this);
clear_costmaps_srv_ = private_nh.advertiseService("clear_costmaps",
&MoveBase::clearCostmapsService, this);
```

## movebase状态之间 (PLANNING, CONTROLLING, CLEARING ) 的切换条件

- 切换到CLEARING:

1. 得到路径但计算不出下一步控制时重新进行路径规划
2. 没有超时，但是没有找到有效全局路径

- 切换到CONTROLLING:

获得了全局路径，并且没有到达目标点

- 切换到PLANNING:

1. 构造函数初始化
2. 获得新的目标点
3. 目标点的坐标系和全局坐标系不一致
4. 执行recovery之后
5. resetState

## rviz下发任务后的数据流

1. 收到goal 信息

```
goal_sub_ = simple_nh.subscribe<geometry_msgs::PoseStamped>("goal", 1,
boost::bind(&MoveBase::goalCB, this, _1));
```

2. planner\_thread\_

```
planner_thread_ = new boost::thread(boost::bind(&MoveBase::planThread,
this)); // planner专门起一个线程
```

3. planner\_plan\_ ----> lastest\_plan\_ ----> controller\_plan\_

4. 局部规划器获得全局路径

```
tc_->setPlan(*controller_plan_)
```

5. 计算出速度。

```
tc_->computeVelocityCommands(cmd_vel)
```

## 恢复行为

在导航模块出现异常的时候，恢复行为会被触发。其中全局路径规划，局部规划或者来回震荡都会导致 recovery behavior 被触发。代码逻辑中是move\_base.cpp中的state会被更新为CLEARING状态，recovery\_trigger也会被更新为相应的行为。

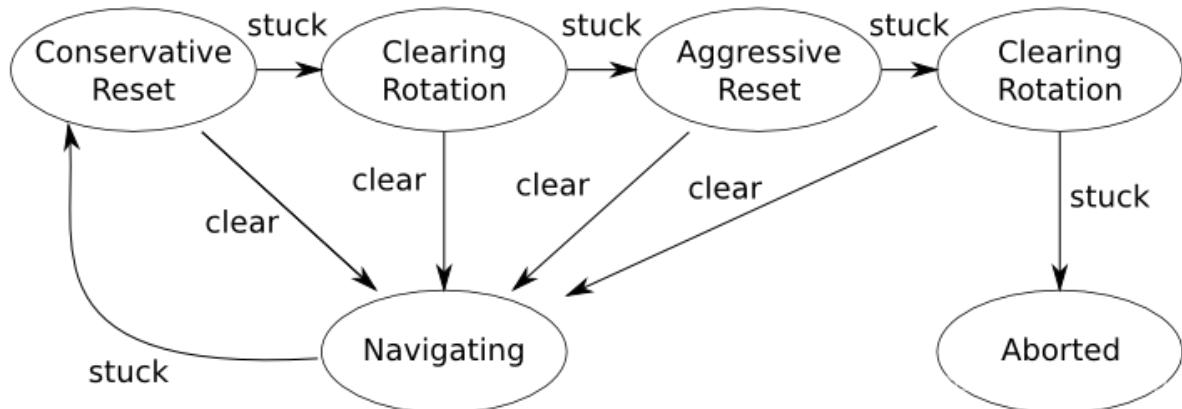
```
enum RecoveryTrigger
{
    PLANNING_R, // 全局规划失败
    CONTROLLING_R, // 局部轨迹规划失败
    OSCILLATION_R // 长时间在小区域运动
};
```

navigation中提供了三个恢复的行为

- clear\_costmap\_recovery
- rotate\_recovery
- move\_slow\_and\_clear

loadDefaultRecoveryBehaviors中依次加载了 conservative\_reset rotate\_recovery aggressive\_reset rotate\_recovery

## move\_base Default Recovery Behaviors



## 恢复行为的使用场景

Movebase有三个状态state\_：

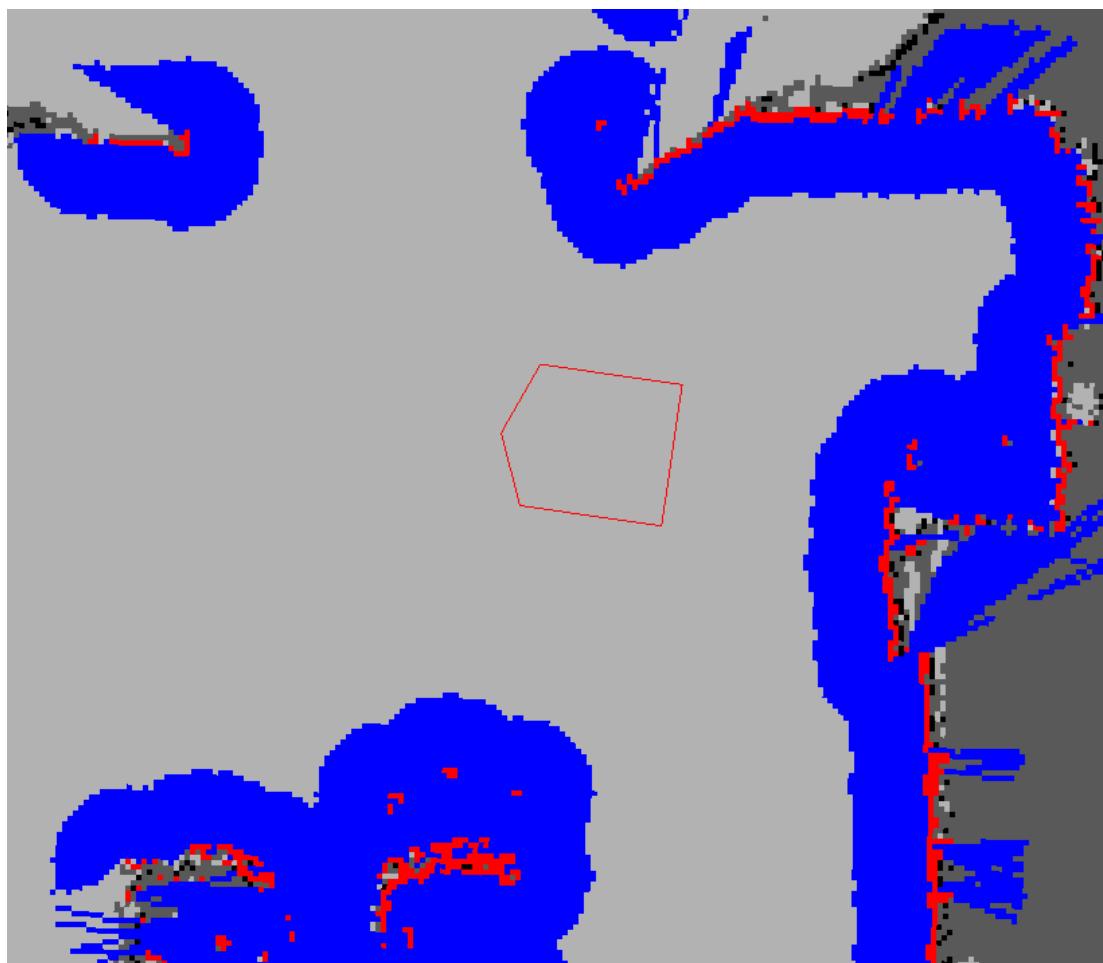
- PLANNING：全局规划中，尚未得到全局路径
- CONTROLLING：全局规划完成，进入局部规划
- CLEARING：恢复行为

# 第四章 导航框架Navigation Stack学习 (Part 2)

## 4.1 costmap\_2d代价地图源码

### 1、概述

红色点代表障碍物，蓝色点代表膨胀后的障碍物。红色的多边形表示机器人的footprint(足迹)，用来做碰撞检查的。机器人的footprint与红色的障碍物点不应该相交，footprint的中心的也不应该与蓝色点重合。



costmap\_2d包提供了一个机器人在其中导航的占据栅格地图。costmap接收传感器数据和用静态地图的信息来更新障碍物信息（通过costmap\_2d::Costmap2DROS的对象）。costmap\_2d类会加载不同的层

- static layer 静态层
- obstacle layer 障碍物层
- voxel layer 体素层

- inflation layer 膨胀层

这三层组合成了master map (最终的costmap) , 供给路线规划模块使用。

obstacle layer 维护的是2D 的地图, voxel layer 是3D的。

costmap\_2d::Costmap2DROS 维护许多ROS相关功能。costmap\_2d::LayeredCostmap 维护每个layer的信息。其中每个layer会在Costmap2DROS中用pluginlib 实例化，然后加到LayeredCostmap 中。每个layer可能会被单独的编译，也允许被修改。接下来介绍代价地图是怎么被更新的。

## 2、障碍物标记和清除

代价地图自动的订阅传感器话题并进行相应的更新。每个传感器数据可以被用来标记 (mark) 或者清除 (clear) 代价地图中的障碍物。Marking就是把改变对应单元格的代价。Clearing通过raytracing的方法清除障碍物，raytracing之后课程会具体介绍。

## 3、占据, 无障碍物和未知区域

每个单元格对应的代价范围为0~255。其实可以分成了三个类别。占据, 无障碍物和未知区域。代码中对应的代价表示为costmap\_2d::LETHAL\_OBSTACLE, costmap\_2d::FREE\_SPACE和costmap\_2d::NO\_INFORMATION。

## 4、地图更新

代价地图会按照一定的频率进行地图更新。代码中也有修改接口update\_frequency. 每个循环中，传感器的数据先进来，再在占据栅格地图上标记和清除障碍物。

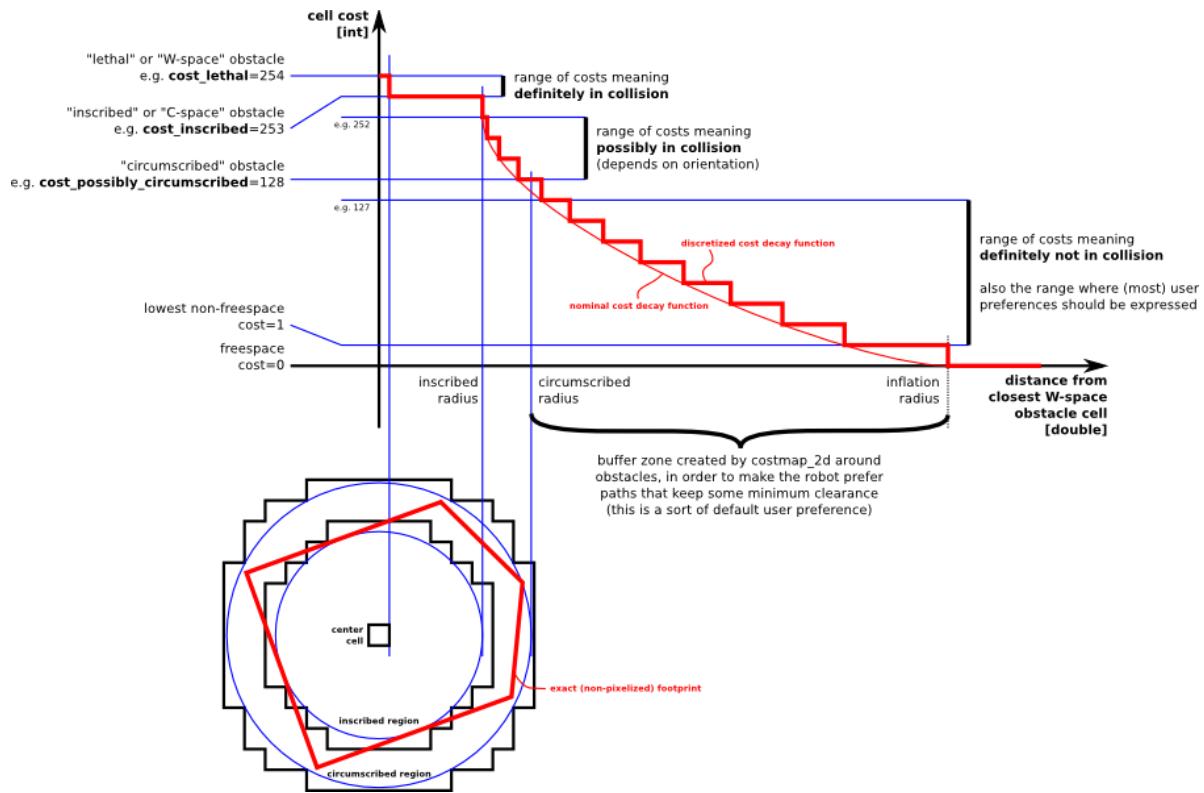
## 5、tf转换

tf转换在代价地图中非常重要。costmap\_2d::Costmap2DROS的实例广泛的使用了tf。一般传感器的数据来源是基于他们自己的坐标系比如laser\_frame或者camera\_frame。 使用tf将所以传感器数据和地图信息转换到同一个坐标系下比如global\_frame。tf转换中transform\_tolerance是转换的最大等待时间。如果tf 树没有及时的更新，导航模块会停止机器人的运动。

## 6、障碍物的膨胀

膨胀就是把被占据的单元格的代价向四周传播。代价地图的值可以分成5类:

- Lethal (致命): 单元格有障碍物，如果机器人的中心在这个地方，一定会碰撞。
- Inscribed: 机器人的内切圆与障碍物有相交，如果机器人的中心在这个地方，机器人肯定碰撞。
- Possibly circumscribed: 机器人的外接圆与障碍物有相交，
- Freespace: 没有碰撞风险
- Unknown: 所给单元格的代价是未知的。



## 7、地图类型

有两个初始化costmap\_2d::Costmap2DROS对象的方法。

一个是用户建的静态地图。代价地图在初始化的时候会匹配静态地图的宽，高和障碍物信息。这个配置一般和定位系统一起使用比如amcl，它允许机器人在map坐标系下用传感器数据注册障碍物和更新它的代价地图。

另外一个是给定宽度，高度和rolling\_window设置为true, 此时不管怎么运动，机器人一直位于代价地图的中心。如果障碍物信息离开了window的范围，就会被舍去。该配置一般用在odom坐标系下，因为机器人只关心局部区域的障碍物。

订阅话题：

`~<name>/footprint` ([geometry\\_msgs/Polygon](#))

发布的话题：

`~<name>/costmap` ([nav\\_msgs/OccupancyGrid](#))

- 代价地图的值

`~<name>/costmap_updates` ([map\\_msgs/OccupancyGridUpdate](#))

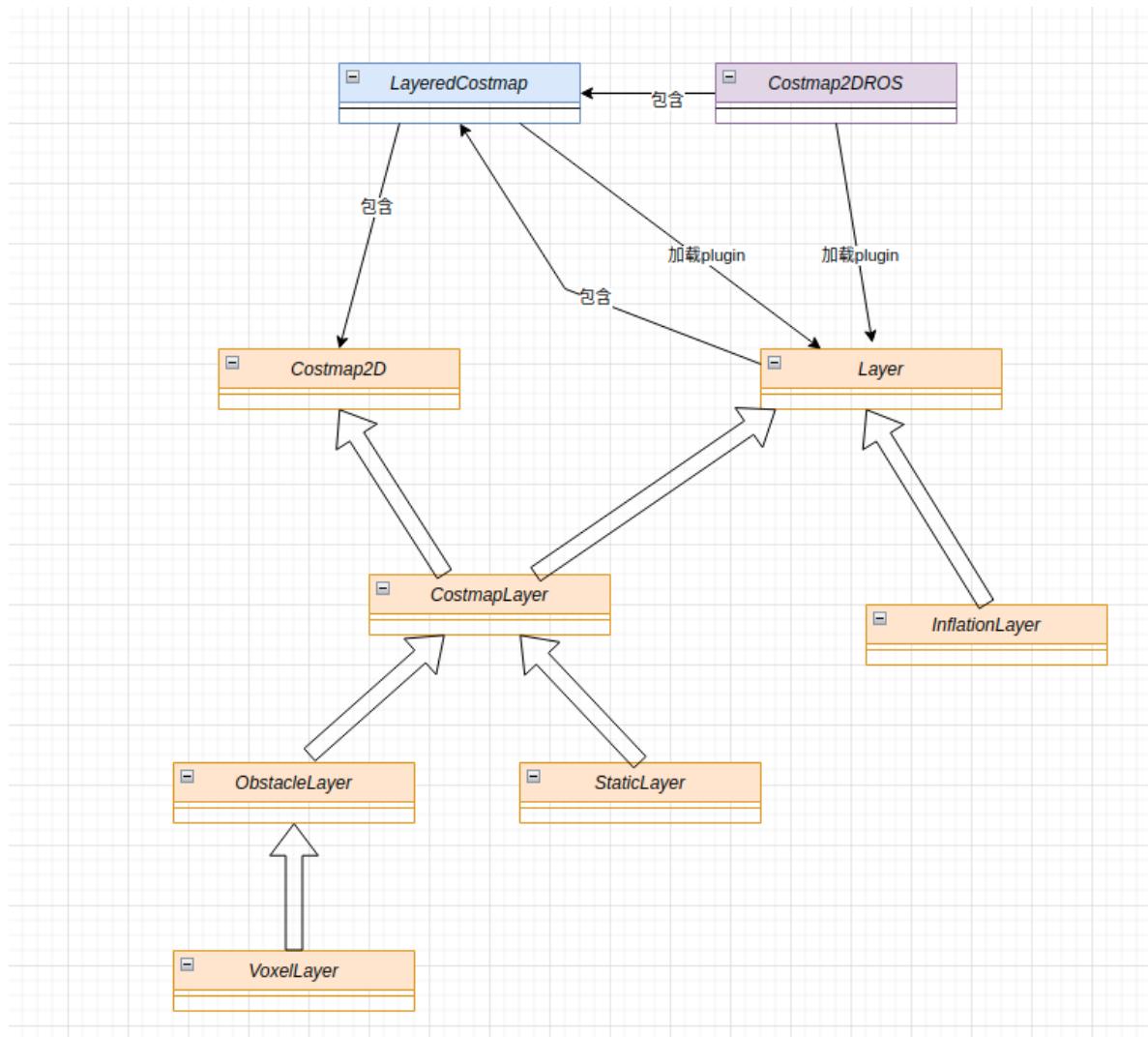
- 代价地图中更新区域的值

`~<name>/voxel_grid` ([costmap\\_2d/VoxelGrid](#))

- 可选的，如果栅格地图用的体素，就需要发布

## Costmap\_2d框架:

下图可以简要说明Costmap的各种接口的关系:



Costmap主接口是costmap\_2d::Costmap2DROS，维持Costmap在ROS中大多数的相关功能。它用所包含的costmap\_2d::LayeredCostmap类来跟踪每一个层。每层使用pluginlib (ROS插件机制) 来实例化并添加到LayeredCostmap类的对象中。各个层可以被独立的编译，且允许使用C++接口对costmap做出任意的改变。

### Costmap的ROS接口:

costmap\_2d::Costmap2DROS是ROS对costmap的封装类。具体的地图信息是储存在各个Layer中。该类负责与其他ROS模块交互，发布和订阅一些内容。

Costmap初始化流程:

- step 1 初始化旧的位姿
- step 2 获取全局和机器人基座的坐标系名字
- step 3 确认robot base frame 和 the global frame 能否转换很有必要
- step 4 检查costmap是否需要滑窗
- step 5 做了这么多准备工作，终于开始做重要的事情了。加载各layer

step 6 订阅footprint 话题，并设置footprint

step 7 这些设置在之后创建新的更新地图线程时会被用到

step 8 创建检测机器人运动的计时器

step 9 开启动态参数配置

发布和订阅的话题

```
private_nh.param(topic_param, topic, std::string("footprint"));
footprint_sub_ = private_nh.subscribe(topic, 1,
&Costmap2DROS::setUnpaddedRobotFootprintPolygon, this);

private_nh.param(topic_param, topic, std::string("footprint"));
footprint_pub_ = private_nh.advertise<geometry_msgs::PolygonStamped>(topic,
1);
```

当然也有专门发布costmap的Costmap2DPublisher类

```
costmap_pub_ = ros_node->advertise<nav_msgs::OccupancyGrid>(topic_name, 1,
boost::bind(&Costmap2DPublisher::onNewSubscription, this, _1));
costmap_update_pub_ = ros_node->advertise<map_msgs::OccupancyGridUpdate>
(topic_name + "_updates", 1);
```

## Layer类

基类layer有两个重要接口,论文中也有提到:

- updateBounds 用于更新范围
- updateCosts 更新cost

```
virtual void updateBounds(double robot_x, double robot_y, double robot_yaw,
double* min_x, double* min_y,
                           double* max_x, double* max_y) {}
virtual void updateCosts(Costmap2D& master_grid, int min_i, int min_j, int
max_i, int max_j) {}
```

- costmap2D基类主要换算坐标等等，负责打杂。里面定义了costmap的存储: unsigned char\* costmap

```
map_update_thread_ = new
boost::thread(boost::bind(&Costmap2DROS::mapUpdateLoop, this,
map_update_frequency));
```

Costmap的更新在mapUpdateLoop线程中实现，地图的更新是在Costmap2DROS::updateMap调用的LayeredCostmap实例的成员函数updateMap实现的，该实例的updateMap可以分为两步：

**Step 1** UpdateBounds 更新每个Layer的范围，StaticLayer只在第一次做更新，范围是加载的整张Map的大小，在UpdateBounds过程中没有对Static层的数据做过任何的修改。ObstacleLayer在这个阶段主要的操作是根据传感器信息来更新该层的数据，然后更新Bounds。InflationLayer则保持上一次的Bounds。

**Step 2** UpdateCosts：将各层数据复制到Master Map

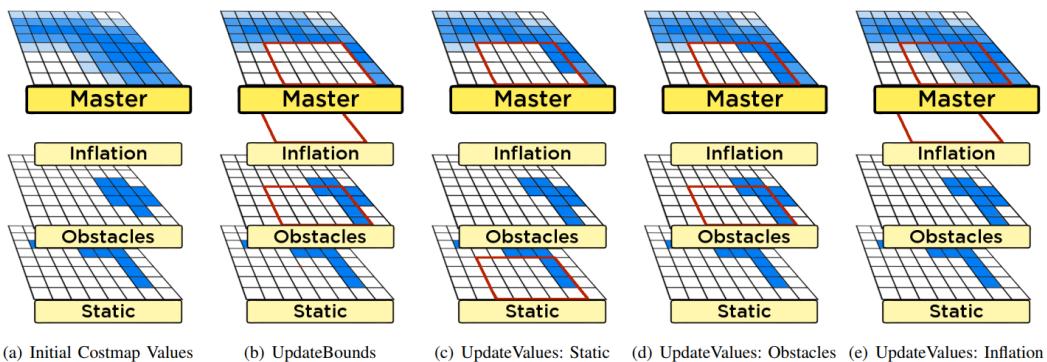


Fig. 2. Update Algorithm - In (a), the layered costmap has three layers and the master costmap. The obstacles and static layers maintain their own copies of the grid, while the inflation layer does not. To update the costmap, the algorithm first calls the `updateBounds` method (b) on each layer, starting with the first layer in the ordered list, shown on the bottom. To determine the new bounds, the obstacles layer updates its own costmap with new sensor data. The result is a bounding box that contains all the areas that each layer needs to update. Next, each layer in turn updates the master costmap in the bounding box using the `updateValues` method, starting with the static layer (c), followed by the obstacles layer (d) and the inflation layer (e).

在 (a) 中，有三个Layer和Master costmap,StaticLayer和ObstaclesLayer维护自己的栅格地图，但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)。

layer和costmap区别

layer会被LayeredCostmap调用，用于明确多大的地图范围用来更新。

## LayeredCostmap基类

实例化不同的layer插件并将他们叠加。

updateMap函数

## costmap2D基类

保存costmap的信息，也有坐标转换。成员函数mapToWorld和worldToMap经常被用到。

```
void Costmap2D::mapToWorld(unsigned int mx, unsigned int my, double& wx, double& wy) const
{
```

```

wx = origin_x_ + (mx + 0.5) * resolution_;
wy = origin_y_ + (my + 0.5) * resolution_;
}

bool Costmap2D::worldToMap(double wx, double wy, unsigned int& mx, unsigned int&
my) const
{
    if (wx < origin_x_ || wy < origin_y_)
        return false;
    mx = (int)((wx - origin_x_) / resolution_);
    my = (int)((wy - origin_y_) / resolution_);
    if (mx < size_x_ && my < size_y_)
        return true;
    return false;
}

```

## CostmapLayer类

同时继承了layer和costmap2D类。新增加了几种更新代价的函数。

```

void updateWithTrueOverwrite(costmap_2d::Costmap2D& master_grid, int min_i,
int min_j, int max_i, int max_j);
void updateWithOverwrite(costmap_2d::Costmap2D& master_grid, int min_i, int
min_j, int max_i, int max_j);
void updateWithMax(costmap_2d::Costmap2D& master_grid, int min_i, int min_j,
int max_i, int max_j);

```

## StaticLayer类

用来处理建好的静态地图。继承了CostmapLayer类并且对updateBounds和updateCosts成员函数重写。

```

class StaticLayer : public CostmapLayer{
    // ...
}

```

## ObstacleLayer类

同样继承了CostmapLayer类并且对updateBounds和updateCosts成员函数重写。加入了几个重要的回调函数，由于实时更新传感器传来的数据。

## Inflationlayer不维护专门的costmap

```

costmap_2d::Costmap2D* costmap = layered_costmap_->getCostmap();

```

map\_server (main.cpp 加载地图) , image\_loader(主要是loadMapFromFile函数) , map\_saver (保存地图)

map\_server 节点主要就是为了加载 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。

### map\_saver

主要函数是类 MapGenerator 的构造函数，订阅来自建图发布的 /map topic，从而写地图 pgm 和 yaml 文件。

```
map_sub_ = n.subscribe("map", 1, &MapGenerator::mapCallback, this);

std::string mapdatafile = mapname_ + ".pgm";
ROS_INFO("Writing map occupancy data to %s", mapdatafile.c_str());
FILE* out = fopen(mapdatafile.c_str(), "w");

std::string mapmetadatafile = mapname_ + ".yaml";
ROS_INFO("Writing map occupancy data to %s", mapmetadatafile.c_str());
FILE* yaml = fopen(mapmetadatafile.c_str(), "w");
```

## 4.8 导航框架里的定位模块

AMCL是提供在2D环境下的概率定位功能，是在已知地图（前期需要建好地图，在本课程中使用了ros提供的gmapping slam建立地图）中使用粒子滤波方法得到机器人位姿。AMCL根据Lidar数据、Odom数据，计算出机器人在地图的位姿。

AMCL (adaptive Monte Carlo localization) 是**自适应蒙特卡洛定位算法**，**基于粒子滤波器**对机器人在已知的**地图环境中进行定位**。AMCL是自主移动机器人在二维环境中下的基于概率的定位系统。

订阅的话题：

- **scan** : Laser scan messages from the LIDAR ([sensor\\_msgs/LaserScan](#)).
- **/tf** : Coordinate frame transformations ([tf/tfMessage](#)).
- **/initialpose** : The initial position and orientation of the robot using quaternions. ([geometry\\_msgs/PoseWithCovarianceStamped](#)) — RViz initial pose button publishes to this topic.
- **/map** : The occupancy grid map created using gmapping, Hector SLAM, or manually using an image ([nav\\_msgs/OccupancyGrid](#)).

发布的话题：

- **/amcl\_pose** : Robot's estimated pose in the map ([geometry\\_msgs/PoseWithCovarianceStamped](#)).
- **/particlecloud** : The set of pose estimates being maintained by the filter ([geometry\\_msgs/PoseArray](#)).
- **/tf** ([tf/tfMessage](#)): Publishes the transform from odom (which can be remapped via the ~odom\_frame\_id parameter) to map.

amcl节点：

amcl\_node.cpp

重要的算法实现：

- amcl/sensors/
  - amcl\_sensor.cpp
  - amcl\_odom.cpp
  - amcl\_laser.cpp

MCL：蒙特卡洛定位介绍

MCL算法流程：

1. 随机生成粒子，粒子权重相等。更常用是位置初始化可以设为(0,0,0)。
2. 根据运动模型估计新时刻位姿
3. 根据观测模型及观测值，更新该时候粒子携带位姿的权重
4. 重采样，调整粒子分布，收紧粒子(去掉可信度低的粒子，增加新的)
5. 计算粒子权重的均值和方差，得到最佳位置估计。
6. 回到第2步，进行下一帧的扫描匹配。

```
1:      Algorithm MCL( $\mathcal{X}_{t-1}, u_t, z_t, m$ ):
2:           $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$ 
3:          for  $m = 1$  to  $M$  do
4:               $x_t^{[m]} = \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$ 
5:               $w_t^{[m]} = \text{measurement\_model}(z_t, x_t^{[m]}, m)$ 
6:               $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
7:          endfor
8:          for  $m = 1$  to  $M$  do
9:              draw  $i$  with probability  $\propto w_t^{[i]}$ 
10:             add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
11:         endfor
12:         return  $\mathcal{X}_t$ 
```

```

1: Algorithm Augmented_MCL( $\mathcal{X}_{t-1}, u_t, z_t, m$ ):
2:   static  $w_{\text{slow}}, w_{\text{fast}}$ 
3:    $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$ 
4:   for  $m = 1$  to  $M$  do
5:      $x_t^{[m]} = \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$ 
6:      $w_t^{[m]} = \text{measurement\_model}(z_t, x_t^{[m]}, m)$ 
7:      $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
8:      $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{M} w_t^{[m]}$ 
9:   endfor
10:   $w_{\text{slow}} = w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{avg}} - w_{\text{slow}})$ 
11:   $w_{\text{fast}} = w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{avg}} - w_{\text{fast}})$ 
12:  for  $m = 1$  to  $M$  do
13:    with probability  $\max\{0.0, 1.0 - w_{\text{fast}}/w_{\text{slow}}\}$  do
14:      add random pose to  $\mathcal{X}_t$ 
15:    else
16:      draw  $i \in \{1, \dots, N\}$  with probability  $\propto w_t^{[i]}$ 
17:      add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
18:    endwith
19:  endfor
20:  return  $\mathcal{X}_t$ 

```

## 重要函数

地图获取的函数，可订阅topic也可request service

- rpc方式获取地图 : **requestMap**
- 订阅话题获取地图: **mapReceived**
  - handleMapMessage
  - freeMapDependentMemory // 释放 map, pf, odom, laser的空间
  - map\_ = convertMap(msg);
  - 创建粒子滤波器
  - 实例化odom对象
  - 实例化laser对象
    - LASER\_MODEL\_BEAM
    - LASER\_MODEL\_LIKELIHOOD\_FIELD\_PROB
  - applyInitialPose

### • **laserReceived**

scan处理函数完成任务:

计算雷达相对于机器人车体base坐标系的安装laser pose

获取里程计的估计信息,

判断是否需要更新

更新完成后,

## 完成重采样

- `odom->UpdateAction(pf, (AMCLSensorData*)&odata);`
  - ODOM\_MODEL\_OMNI
  - ODOM\_MODEL\_DIFF
  - ODOM\_MODEL\_OMNI\_CORRECTED
  - ODOM\_MODEL\_DIFF\_CORRECTED
    - `pf_ran_gaussian`
- `lasers[laser_index]->UpdateSensor(pf, (AMCLSensorData*)&ldata);`
  - LASER\_MODEL\_BEAM
  - LASER\_MODEL\_LIKELIHOOD\_FIELD
  - LASER\_MODEL\_LIKELIHOOD\_FIELD\_PROB
    - `pf_update_sensor`

## 重要结构体和类：

```
pf_vector_t : pose  
pf_sample_t : pose + weight  
pf_cluster_t mean + cov, m[4], c[2][2]  
pf_sample_set_t: pf_kdtree_t     pf_cluster_t  
pf_t : pf_sample_set_t    pf_init_model_fn_t  
  
// 关于传感器数据的类  
AMCLOdomData  
AMCLSensorData  
AMCLLaserData
```

# 第五章 全局规划算法

## 全局规划器 理论知识

### 全局路径规划定义（这里我推荐直接一段英文解释）：

Path planning seeks a sequence of collision-free motions of a mobile robot from a start to a goal pose among a configuration of static obstacles.

Path planning assumes a complete geometric description of the robots footprint and the environment. In the case of mobile robots the workspace is a 2D environment populated with planar obstacles represented by a map. The objective is to find a collision-free path through the environment that connects the start and goal.

路径规划在静态障碍物的配置中寻找移动机器人从起点到目标姿势的一系列无碰撞运动。

路径规划假设机器人足迹和环境的完整几何描述。在移动机器人的情况下，工作空间是一个 2D 环境，其中填充了由地图表示的平面障碍物。目标是在连接起点和目标的环境中找到一条无碰撞路径。

# 常用的 ROS Global Planner功能包

- navfn
- global\_planner
- Dijkstra
- A\*
- asr\_navfn
- MoveIt!
- sbpl\_lattice\_planner
- Voronoi\_planner
- hybrid\_a\_star

## 架构-文件系统

- plan\_node: 全局规划器的入口
- planner\_core
  - initialize: 初始化对象和参数
  - makePlan: 负责干活
    - planner->calculatePotentials: 计算出potential\_array
    - getPlanFromPotential: 提取全局路径
    - path\_maker\_->getPath
- astar
  - calculatePotentials: 计算路径代价的函数
  - add: 添加点并更新代价函数
    - p\_calc\_->calculatePotential
- potential\_calculator
  - calculatePotential: 返回当前点四周最小的代价值及累加前面累加的代价值
- quadratic\_calculator
  - calculatePotential: 使用二次逼近方式
- grid\_path
  - getPath: 棚格路径, 从终点开始找上下或左右4个中最小的棚格直到起点
- gradient\_path
  - getPath: 梯度路径, 从周围八个棚格中找到下降梯度最大的点

## planner\_core

- 声明插件
- 初始化
  - 参数加载
  - 对象功能初始化
- 干活函数 `makePlan`
  - 起始点, 目标点 坐标转换
  - 起始点, 目标点 范围界限判断
  - 棚格初始化处理
  - 计算势场 (核心步骤 1)
  - 提取路径 (核心步骤 2)

- 添加方向选项
- 发布，可视化

## 参数调整

```
GlobalPlanner:
allow_unknown: false
start_tolerance: 0.2
goal_tolerance: 0.1
#visualize_potential: false
use_dijkstra: false
use_quadratic: true
use_grid_path: true
lethal_cost: 253
neutral_cost: 60
cost_factor: 3
publish_potential: false
time_out: 0.2
#orientation_mode: 0
#orientation_window_size: 1
#outline_map: true
```

## 调用流程

## 程序解读

在Move Base中有全局规划器的如下调用，完成了全局规划器的实例化：

```
try {
    planner_ = bgp_loader_.createInstance(global_planner);
    planner_->initialize(bgp_loader_.getName(global_planner),
planner_costmap_ros_);
} catch (const pluginlib::PluginlibException& ex) {
    ROS_FATAL("Failed to create the %s planner, are you sure it is properly
registered and that the containing library is built? Exception: %s",
global_planner.c_str(), ex.what());
    exit(1);
}
```

planner初始化

planner由类NavFn实现，plan\_pub用于发布规划路径，make\_plan\_srv\_提供路径规划服务。

```
void GlobalPlanner::initialize(std::string name, costmap_2d::Costmap2D* costmap,
std::string frame_id) {
    if (!initialized_) {
        ros::NodeHandle private_nh("~/" + name);
        costmap_ = costmap;
        frame_id_ = frame_id;

        unsigned int cx = costmap->getSizeInCellsX(), cy = costmap-
>getSizeInCellsY();

        private_nh.param("old_navfn_behavior", old_navfn_behavior_, false);
```

```

if(!old_navfn_behavior_)
    convert_offset_ = 0.5;
else
    convert_offset_ = 0.0;

bool use_quadratic;
private_nh.param("use_quadratic", use_quadratic, true);
// 根据选择new出对应的p_calc_实例. 计算“一个点”的可行性
if (use_quadratic)
    p_calc_ = new QuadraticCalculator(cx, cy); // 使用二次差值近似的
potential
else
    p_calc_ = new PotentialCalculator(cx, cy); //

bool use_dijkstra;
private_nh.param("use_dijkstra", use_dijkstra, true);
// 根据选择new出对应的planner实例. 计算“所有”的可行点
if (use_dijkstra)
{
    // dijkstra算法
    DijkstraExpansion* de = new DijkstraExpansion(p_calc_, cx, cy);
    if(!old_navfn_behavior_)
        de->setPreciseStart(true);
    planner_ = de;
}
else
    planner_ = new AStarExpansion(p_calc_, cx, cy); // A*算法

bool use_grid_path;
private_nh.param("use_grid_path", use_grid_path, false);
// 路径方法, new出path_maker_实例。从可行点中提取路径
if (use_grid_path)
    // 棚格路径, 从终点开始找上下或左右4个中最小的棚格直到起点
    path_maker_ = new GridPath(p_calc_);
else
    // 梯度路径, 从周围八个棚格中找到下降梯度最大的点
    path_maker_ = new GradientPath(p_calc_);

orientation_filter_ = new OrientationFilter();

plan_pub_ = private_nh.advertise<nav_msgs::Path>("plan", 1);
potential_pub_ = private_nh.advertise<nav_msgs::OccupancyGrid>
("potential", 1);

private_nh.param("allow_unknown", allow_unknown_, true);
planner_->setHasUnknown(allow_unknown_);
private_nh.param("planner_window_x", planner_window_x_, 0.0);
private_nh.param("planner_window_y", planner_window_y_, 0.0);
private_nh.param("default_tolerance", default_tolerance_, 0.0);
private_nh.param("publish_scale", publish_scale_, 100);
private_nh.param("outline_map", outline_map_, true);

make_plan_srv_ = private_nh.advertiseService("make_plan",
&GlobalPlanner::makePlansService, this);

dsrv_ = new
dynamic_reconfigure::Server<global_planner::GlobalPlannerConfig>
(ros::NodeHandle("~/" + name));

```

```

dynamic_reconfigure::Server<global_planner::GlobalPlannerConfig>::CallbackType
cb = boost::bind(
    &GlobalPlanner::reconfigureCB, this, _1, _2);
dsrv_->setCallback(cb);

initialized_ = true;
} else
ROS_WARN("This planner has already been initialized, you can't call it
twice, doing nothing");

}

```

## 规划路径

NavfnROS::makePlan方法生成规划路径:

```

bool GlobalPlanner::makePlan(const geometry_msgs::PoseStamped& start, const
geometry_msgs::PoseStamped& goal,
                               std::vector<geometry_msgs::PoseStamped>& plan) {
    return makePlan(start, goal, default_tolerance_, plan);
}

// 两个步骤完成路径的生成:
// 1. 计算可行点矩阵potential_array (planner_->calculatePotentials)
// 2. 从可行点矩阵中提取路径plan (path_maker_->getPath())
bool GlobalPlanner::makePlan(const geometry_msgs::PoseStamped& start, const
geometry_msgs::PoseStamped& goal,
                             double tolerance,
std::vector<geometry_msgs::PoseStamped>& plan) {
    boost::mutex::scoped_lock lock(mutex_);
    // step 1: 是否已经初始化
    if (!initialized_) {
        ROS_ERROR(
            "This planner has not been initialized yet, but it is being used,
please call initialize() before use");
        return false;
    }

    // 清空路径
    plan.clear();

    ros::NodeHandle n;
    std::string global_frame = frame_id_;

    // step 2.1 目标点的坐标系应该和全局坐标系一致
    if (goal.header.frame_id != global_frame) {
        ROS_ERROR(
            "The goal pose passed to this planner must be in the %s frame.
It is instead in the %s frame.", global_frame.c_str(),
goal.header.frame_id.c_str());
        return false;
    }
    // step 2.2 起始点的坐标系应该和全局坐标系一致
    if (start.header.frame_id != global_frame) {
        ROS_ERROR(

```

```

        "The start pose passed to this planner must be in the %s frame.
It is instead in the %s frame.", global_frame.c_str(),
start.header.frame_id.c_str());
    return false;
}

double wx = start.pose.position.x;
double wy = start.pose.position.y;

unsigned int start_x_i, start_y_i, goal_x_i, goal_y_i;
double start_x, start_y, goal_x, goal_y;
// step 3 判断起始点和目标点是否超出了全局代价地图的范围
if (!costmap_->worldToMap(wx, wy, start_x_i, start_y_i)) {
    ROS_WARN(
        "The robot's start position is off the global costmap. Planning
will always fail, are you sure the robot has been properly localized?");
    return false;
}
if(old_navfn_behavior_){
    start_x = start_x_i;
    start_y = start_y_i;
}else{
    worldToMap(wx, wy, start_x, start_y);
}

wx = goal.pose.position.x;
wy = goal.pose.position.y;

if (!costmap_->worldToMap(wx, wy, goal_x_i, goal_y_i)) {
    ROS_WARN_THROTTLE(1.0,
        "The goal sent to the global planner is off the global costmap.
Planning will always fail to this goal.");
    return false;
}
if(old_navfn_behavior_){
    goal_x = goal_x_i;
    goal_y = goal_y_i;
}else{
    worldToMap(wx, wy, goal_x, goal_y);
}

// step 4 清除起始单元格, 它不可能是障碍物
clearRobotCell(start, start_x_i, start_y_i);

int nx = costmap_->getSizeInCellsX(), ny = costmap_->getSizeInCellsY();

// step 5 确保global_planner用的数组大小和地图一致
p_calc_->setSize(nx, ny);
planner_->setSize(nx, ny);
path_maker_->setSize(nx, ny);
potential_array_ = new float[nx * ny];

if(outline_map_)
    outlineMap(costmap_->getCharMap(), nx, ny, costmap_2d::LETHAL_OBSTACLE);
// step 6 核心步骤, 计算出potential_array_
bool found_legal = planner_->calculatePotentials(costmap_->getCharMap(),
start_x, start_y, goal_x, goal_y,

```

```

nx * ny * 2,
potential_array_);

if(!old_navfn_behavior_)
    planner_->clearEndpoint(costmap_->getCharMap(), potential_array_,
goal_x_i, goal_y_i, 2);
if(publish_potential_)
    publishPotential(potential_array_);

if (found_legal) {
    // step 7 提取全局路径, 用path_maker_->getPath得到路径
    if (getPlanFromPotential(start_x, start_y, goal_x, goal_y, goal, plan))
{
    // 更新目标点的时间戳, 和路径的其他点时间戳一致
    geometry_msgs::PoseStamped goal_copy = goal;
    goal_copy.header.stamp = ros::Time::now();
    plan.push_back(goal_copy);
} else {
    ROS_ERROR("Failed to get a plan from potential when a legal
potential was found. This shouldn't happen.");
}
}else{
    ROS_ERROR("Failed to get a plan.");
}

// step 8 给路径添加方向
orientation_filter_->processPath(start, plan);

// 发布路径和可视化
publishPlan(plan);
delete[] potential_array_;
return !plan.empty();
}

void GlobalPlanner::publishPlan(const std::vector<geometry_msgs::PoseStamped>&
path) {
if (!initialized_) {
    ROS_ERROR(
        "This planner has not been initialized yet, but it is being used,
please call initialize() before use");
    return;
}

//create a message for the plan
nav_msgs::Path gui_path;
gui_path.poses.resize(path.size());

gui_path.header.frame_id = frame_id_;
gui_path.header.stamp = ros::Time::now();

// Extract the plan in world co-ordinates, we assume the path is all in the
same frame
for (unsigned int i = 0; i < path.size(); i++) {
    gui_path.poses[i] = path[i];
}

plan_pub_.publish(gui_path);
}

```

## A star 核心算法

```
AStarExpansion::AStarExpansion(Potentialcalculator* p_calc, int xs, int ys) :  
    Expander(p_calc, xs, ys) {  
}  
// 计算路径代价的函数:  
// costs为代价地图的指针, potential为代价数组, cycles为循环次数, 代码里值为2*nx*ny为地图栅格数的两倍  
bool AStarExpansion::calculatePotentials(unsigned char* costs, double start_x,  
double start_y, double end_x, double end_y,  
int cycles, float* potential) {  
    // queue_为启发式搜索到的向量队列: <i , cost>  
    queue_.clear();  
    // 起点的索引  
    int start_i = toIndex(start_x, start_y);  
    // step 1 将起点放入队列  
    queue_.push_back(Index(start_i, 0));  
    // step 2 potential数组值全设为极大值  
    // std::fill(a,b,x) 将a到b的元素都赋予x值  
    std::fill(potential, potential + ns_, POT_HIGH); // ns_ : 单元格总数  
    // step 3 将起点的potential设为0  
    potential[start_i] = 0;  
    // 目标索引  
    int goal_i = toIndex(end_x, end_y);  
    int cycle = 0;  
    // 进入循环, 继续循环的判断条件为只要队列大小大于0且循环次数小于所有格子数的2倍  
    while (queue_.size() > 0 && cycle < cycles) {  
        // step 4 得到最小cost的索引, 并删除它, 如果索引指向goal(目的地)则退出算法, 返回  
        true  
        Index top = queue_[0];  
        // step 4.1 将向量第一个元素(最小的代价的Index)和向量最后一个位置元素对调, 再用  
        pop_back删除这个元素  
        // pop_heap(iterator, iterator, Compare) _Compare有两种参数, 一种是greater (小顶堆),  
        // 一种是less (大顶堆), 先对调, 再排序  
        std::pop_heap(queue_.begin(), queue_.end(), greater1());  
        // 删除最小代价的点  
        queue_.pop_back();  
  
        int i = top.i;  
        // step 4.2 若是目标点则终止搜索, 搜索成功  
        if (i == goal_i)  
            return true;  
        // step 4.3 将代价最小点i周围点加入搜索队里并更新代价值, 即对前后左右四个点执行add函  
        数  
        add(costs, potential, potential[i], i + 1, end_x, end_y);  
        add(costs, potential, potential[i], i - 1, end_x, end_y);  
        add(costs, potential, potential[i], i + nx_, end_x, end_y);  
        add(costs, potential, potential[i], i - nx_, end_x, end_y);  
  
        cycle++;  
    }  
  
    return false;  
}  
  
// 添加点并更新代价函数
```

```

void AstarExpansion::add(unsigned char* costs, float* potential, float
prev_potential, int next_i, int end_x,
int end_y) {
    // 超出范围, ns_为栅格总数
    if (next_i < 0 || next_i >= ns_)
        return;
    // 忽略已经搜索过的点
    if (potential[next_i] < POT_HIGH)
        return;
    // 忽略障碍物点
    if(costs[next_i]>=lethal_cost_ && !(unknown_ &&
costs[next_i]==costmap_2d::NO_INFORMATION))
        return;
    // p_calc_->calculatePotential() 采用简单方法计算值为costs[next_i] +
neutral_cost_+ prev_potentia 地图代价+单格距离代价(初始化为50)+之前路径代价 为G
    potential[next_i] = p_calc_->calculatePotential(potential, costs[next_i] +
neutral_cost_, next_i, prev_potential);
    // 算出该点的x,y坐标
    int x = next_i % nx_, y = next_i / nx_;
    // 注意这里计算的是曼哈顿距离不是欧几里得距离
    float distance = abs(end_x - x) + abs(end_y - y);

    // potential[next_i]: 起始点到当前点的cost即g(n)
    // distance * neutral_cost_: 当前点到目的点的cost即h(n)。
    // f(n)=g(n)+h(n): 计算完这两个cost后, 加起来即为f(n), 将其存入队列中
    // 加入搜索向量
    queue_.push_back(Index(next_i, potential[next_i] + distance *
neutral_cost_));
    // 对加入的再进行堆排序, 把最小代价点放到front堆头queue_[0]
    std::push_heap(queue_.begin(), queue_.end(), greater1());
}

```

## sbpl\_lattice\_planner 介绍

[http://wiki.ros.org/sbpl\\_lattice\\_planner](http://wiki.ros.org/sbpl_lattice_planner)

[https://github.com/ros-planning/navigation\\_experimental](https://github.com/ros-planning/navigation_experimental)

安装 `navigation_experimental` 功能包。

The most useful package in this repo is [sbpl\\_lattice\\_planner](#), a global planner plugin for `move_base` that wraps the SBPL library to produce kinematically feasible paths.

launch文件: `move_base_sbpl_fake_localization_2.5cm.launch`

原理介绍, 网站地址: <http://sbpl.net/>

## Tutorials

[Primer to the Search-Based Planning Library \(SBPL\)](#)

Learn the basic theory behind graph search and SBPL.

[SBPL Overview](#)

This has a brief overview of how SBPL is organized.

[Planning with SBPL: Example in X, Y, Theta State Space](#)

This has example code for planning with a 2d footprint of a robot in X, Y, and theta dimensions.

### [Navigation with SBPL](#)

This builds on the previous tutorial to handle exploring an unknown environment.

### [Kinematic Constraints and Motion Primitives](#)

An introduction to how kinematic constraints are dealt with in SBPL.

### [Generating Motion Primitives](#)

How to use the Matlab scripts in SBPL to generate custom motion primitives.

### [Forward Arc Motion Primitives](#)

### [Building, Installing, and Using SBPL](#)

个人分析：

SBPL lattice planner采用运动基元的方式，通过图搜的方法进行近似动力学路径采样，生成一条kinodynamic的path。

优点是在free space中生成易于Ackermann小车的带转弯的路径，但潜在的缺点是要平衡计算时间和路径质量的权衡关系，容易出现计算超时和内存过大的潜在风险。

## **推荐相关文献：**

[1] Lydia E. Kavraki, Steven M. La Valle, Motion Planning, *Springer Handbook of Robotics*

[2] Lydia E. Kavraki et al, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, *IEEE Trans. Robotics and Automation*, vol. 12, no. 4, pp. 566-580, (1996)

[3] Steven M. La Valle, James J. Kuffner, Randomized Kinodynamic Planning, *International Journal of Robotics Research*, vol. 20, no. 5, pp. 378-400, (2001)

[4] Peter I. Corke, *Robotics, Vision & Control: Fundamental Algorithms in Matlab*. Springer, (2011)

# **第六章 局部规划器算法模块和代码讲解 (Part 1)**

---

## **文件说明：**

world\_model.h

- class WorldModel:
  - 轨迹规划器用于和环境交互的一个接口

costmap\_model.h

- class CostmapModel : public WorldModel
  - 实现WorldModel接口，为用到Costmap的轨迹规划器提供基于网格的碰撞检测。

footprint\_helper.h

- class FootprintHelper
  - 提供footprint相关的处理函数
    - getFootprintCells
    - getLineCells
    - getFillCells

goal\_functions.h

- 没有定义类，定义了一下关于目标点的处理函数，以及全局路径的处理函数

latched\_stop\_rotate\_controller.h

- class LatchedStopRotateController
  - 主要处理机器人到了目标点位置后朝向不一致时的情况

local\_planner\_limits.h

- class LocalPlannerLimits
  - 存储机器人的速度，加速度限制，停止判断条件

local\_planner\_util.h

- class LocalPlannerUtil
  - 提供一些处理plan的工具类函数

map\_cell.h

- class MapCell
  - 计算路径或者目标点距离

map\_grid.h

- class MapGrid
  - 用于传播路径和目标点距离的MapCell的网格

map\_grid\_cost\_function.h

- Class MapGridCostFunction
  - 提供了基于小范围的map\_grid的代价

obstacle\_cost\_function.h

- ObstacleCostFunction

- 如果机器人的footprint映射到对应的路径点上后，与障碍物发生碰撞，则被赋值为负值。

odometry\_helper\_ros.h

- OdometryHelperRos
  - 处理关于odom信息的工具类

oscillation\_cost\_function.h

- OscillationCostFunction
  - 机器人振荡打分类

point\_grid.h

- PointGrid
  - 实现WorldModel接口的类，提供自由空间的碰撞检测，该类存储网格的点并且检查多边形中的点是否碰撞(通过把机器人的footprint映射到给定的位姿，看有没有和障碍物重合)

prefer\_forward\_cost\_function.h

- class PreferForwardCostFunction: public base\_local\_planner::TrajectoryCostFunction
  - 前进打分类

simple\_scored\_sampling\_planner.h

- class SimpleScoredSamplingPlanner : public base\_local\_planner::TrajectorySearch
  - 根据给定的产生器和代价函数计算一个局部轨迹。这里假设代价越小越好同时负值的代价意味着无穷大

simple\_trajectory\_generator.h

- class SimpleTrajectoryGenerator: public base\_local\_planner::TrajectorySampleGenerator
  - 路径产生的类

trajectory\_cost\_function.h

- TrajectoryCostFunction类
  - 轨迹打分的接口

trajectory\_planner\_ros.h

- TrajectoryPlannerROS类
  - trajectory\_planner的ROS封装类

trajectory\_planner.h

- TrajectoryPlanner
  - trajectory\_planner的功能实现类

trajectory\_sample\_generator.h

- TrajectorySampleGenerator类
  - 轨迹产生器的接口

trajectory\_search.h

- TrajectorySearch
  - 作为一个纯虚类，提供寻找局部轨迹的模块接口

trajectory.h

- Trajectory
  - 存储轨迹的类

twirling\_cost\_function

- class TwirlingCostFunction: public base\_local\_planner::TrajectoryCostFunction
  - 角速度打分的类

velocity\_iterator.h

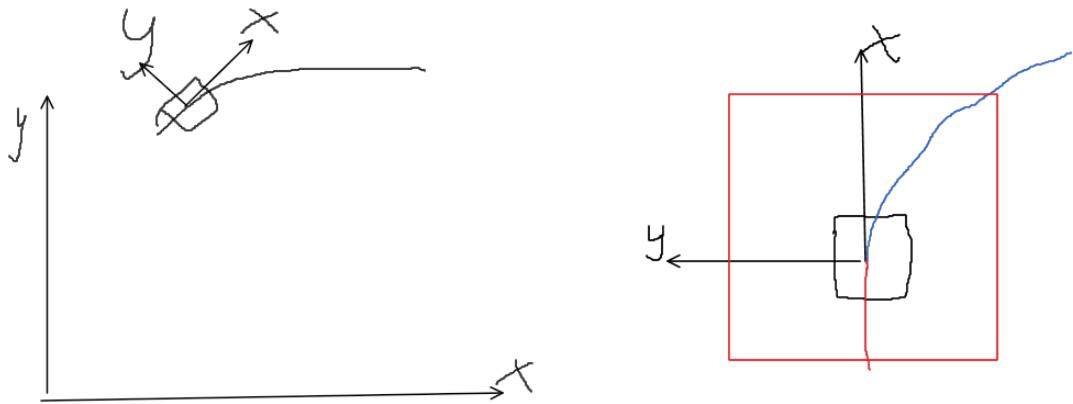
- VelocityIterator
  - VelocityIterator用于获得相同大小的最大和最小值

## 打分相关的：

```
virtual bool prepare() = 0;
virtual double scoreTrajectory(Trajectory &traj) = 0;
```

```
base_local_planner::SimpleTrajectoryGenerator generator_;

// 所有的costfunction都继承自TrajectoryCostFunction
base_local_planner::OscillationCostFunction oscillation_costs_;
base_local_planner::ObstacleCostFunction obstacle_costs_;
base_local_planner::MapGridCostFunction path_costs_;
base_local_planner::MapGridCostFunction goal_costs_;
base_local_planner::MapGridCostFunction goal_front_costs_;
base_local_planner::MapGridCostFunction alignment_costs_;
base_local_planner::TwirlingCostFunction twirling_costs_;
```



## 第七章 局部规划器算法模块和代码讲解 (Part 2)

DWAPlannerROS:

```

max_vel_x: 0.22 # x方向最大线速度绝对值
min_vel_x: 0.1 #最小线速度

max_vel_y: 0.0      # y方向最大线速度的绝对值
min_vel_y: 0.0

max_trans_vel: 0.22 #最大平移速度的绝对值, trans_vel = sqrt(vel_x^2 + vel_y^2)
min_trans_vel: 0.1 #最小平移速度的绝对值,这里不能设为零
trans_stopped_vel: 0.1 #速度低于该值认为机器人已停止

max_rot_vel: 3.2    #最大旋转角速度的绝对值
min_rot_vel: 0.1    #最小旋转角速度的绝对值
rot_stopped_vel: 0.1 #“停止”旋转速度阈值

acc_lim_x: 1.0 # 在x方向的极限加速度, m/s^2
acc_lim_theta: 2.0 # 极限旋转加速度, rad/s^2
acc_lim_y: 0.0

yaw_goal_tolerance: 0.05
xy_goal_tolerance: 0.10
latch_xy_goal_tolerance: true

# Forward Simulation Parameters 前向模拟参数
sim_time: 1.7
vx_samples: 3
vy_samples: 1      # diff drive robot, there is only one sample
vtheta_samples: 20

# Trajectory Scoring Parameters 轨迹评分参数

```

```

path_distance_bias: 32.0      #轨迹与给定路径接近程度的权重

goal_distance_bias: 24.0      #轨迹与局部目标点的接近程度的权重，也用于速度控制

occdist_scale: 0.01          # 轨迹躲避障碍物的程度

forward_point_distance: 0.325 #以机器人为中心，额外放置一个计分点的距离

stop_time_buffer: 0.2        # 为防止碰撞，机器人必须提前停止的时间长度

scaling_speed: 0.25          # - absolute velocity at which to start scaling
the robot's footprint

#开始缩放机器人足迹时的速度的绝对值，单位为m/s。(vmag - scaling_speed) /
(max_trans_vel - scaling_speed) * max_scaling_factor + 1.0

max_scaling_factor: 0.2       #最大缩放因子。max_scaling_factor为上式的值的大小。

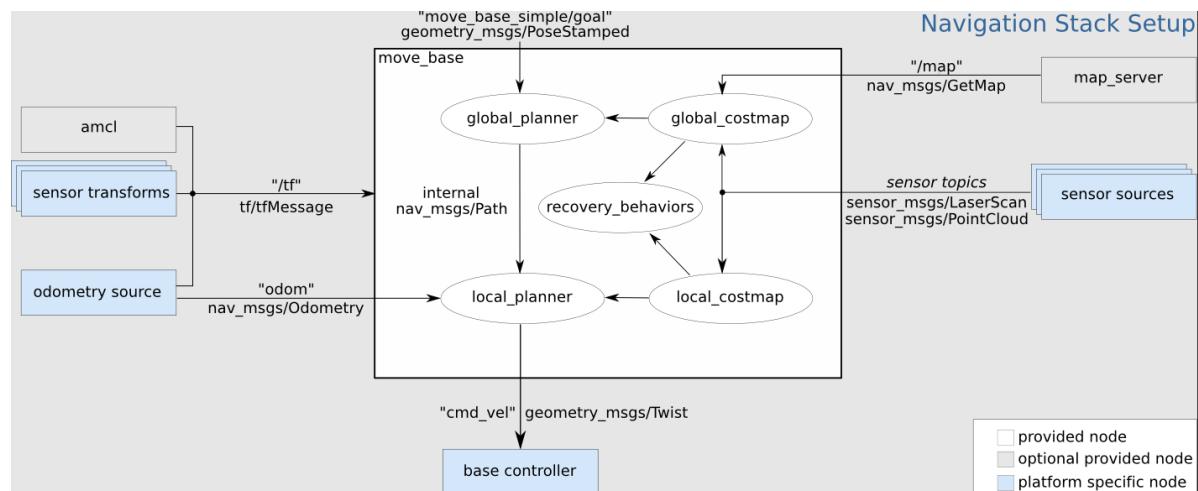
# Oscillation Prevention Parameters 振荡预防参数
oscillation_reset_dist: 0.05 #机器人运动多远距离才会重置振荡标记

# holonomic_robot: false

```

## 第八章 TEB局部规划器介绍

### 8.1 TEB规划效果Demo



TEB官方链接：

[http://wiki.ros.org/teb\\_local\\_planner](http://wiki.ros.org/teb_local_planner)

## teb\_local\_planner

Kinetic melodic

noetic

Show EOL distros:

[Documentation](#) [Status](#)

## Package Summary

✓ Released ✓ Continuous Integration: 3 / 3 ✓ Documented

The teb\_local\_planner package implements a plugin to the base\_local\_planner of the 2D navigation stack. The underlying method called Timed Elastic Band locally optimizes the robot's trajectory with respect to trajectory execution time, separation from obstacles and compliance with kinodynamic constraints at runtime.

- Maintainer status: maintained
- Maintainer: Christoph Rösmann <christoph.roesmann AT tu-dortmund DOT de>
- Author: Christoph Rösmann <christoph.roesmann AT tu-dortmund DOT de>
- License: BSD
- Source: git [https://github.com/rst-tu-dortmund/teb\\_local\\_planner.git](https://github.com/rst-tu-dortmund/teb_local_planner.git) (branch: noetic-devel)

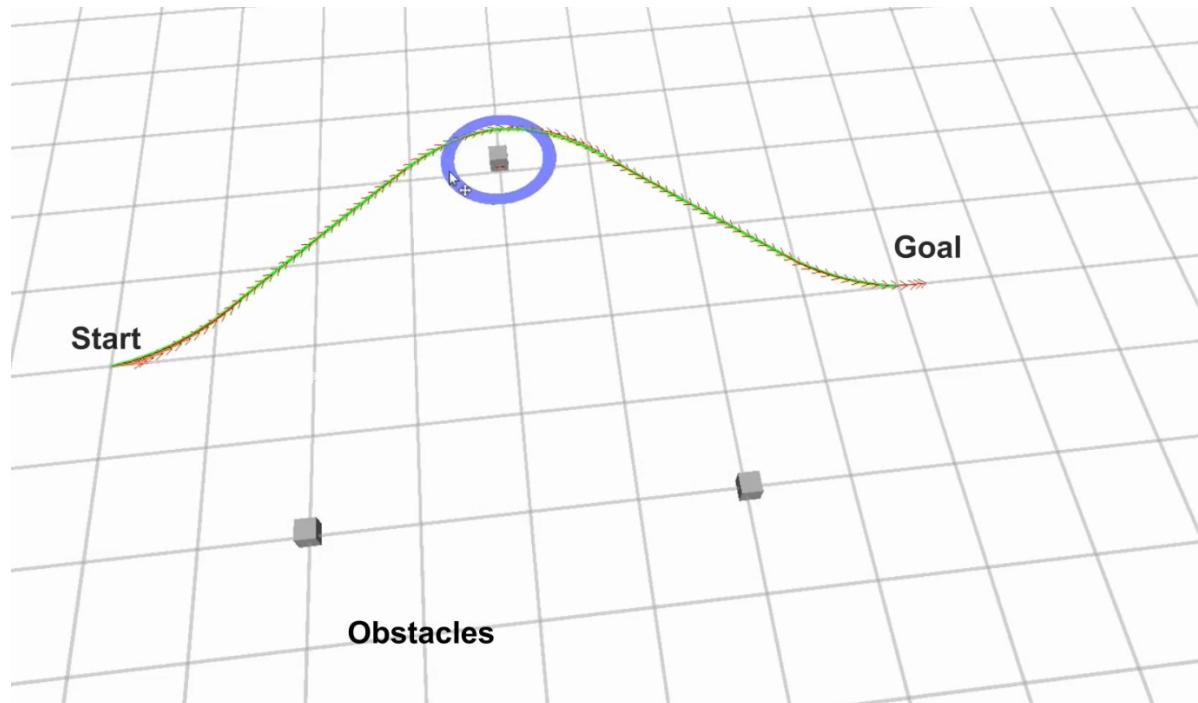
### Package Links

[Code API](#)  
[Msg API](#)  
[Tutorials](#)  
[FAQ](#)  
[Changelog](#)  
[Change List](#)  
[Reviews](#)

### Dependencies (23)

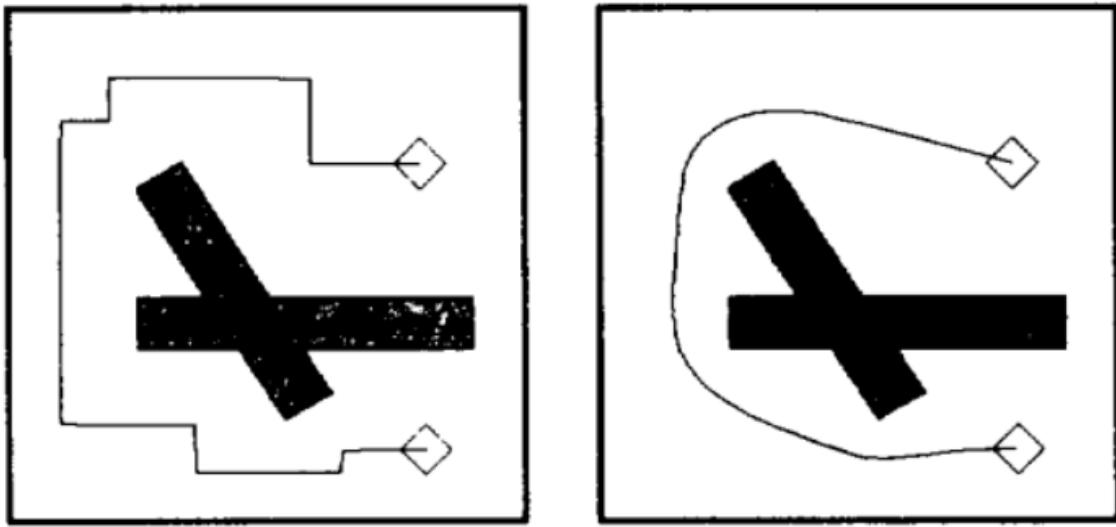
[Used by \(3\)](#)  
[Jenkins jobs \(10\)](#)

TEB测试：



## 8.2 论文带读

主要是根据障碍物(obstacles)来施加一个力(contraction force)让原本看起来松弛的路径变的紧绷  
(removes any slack in the path)



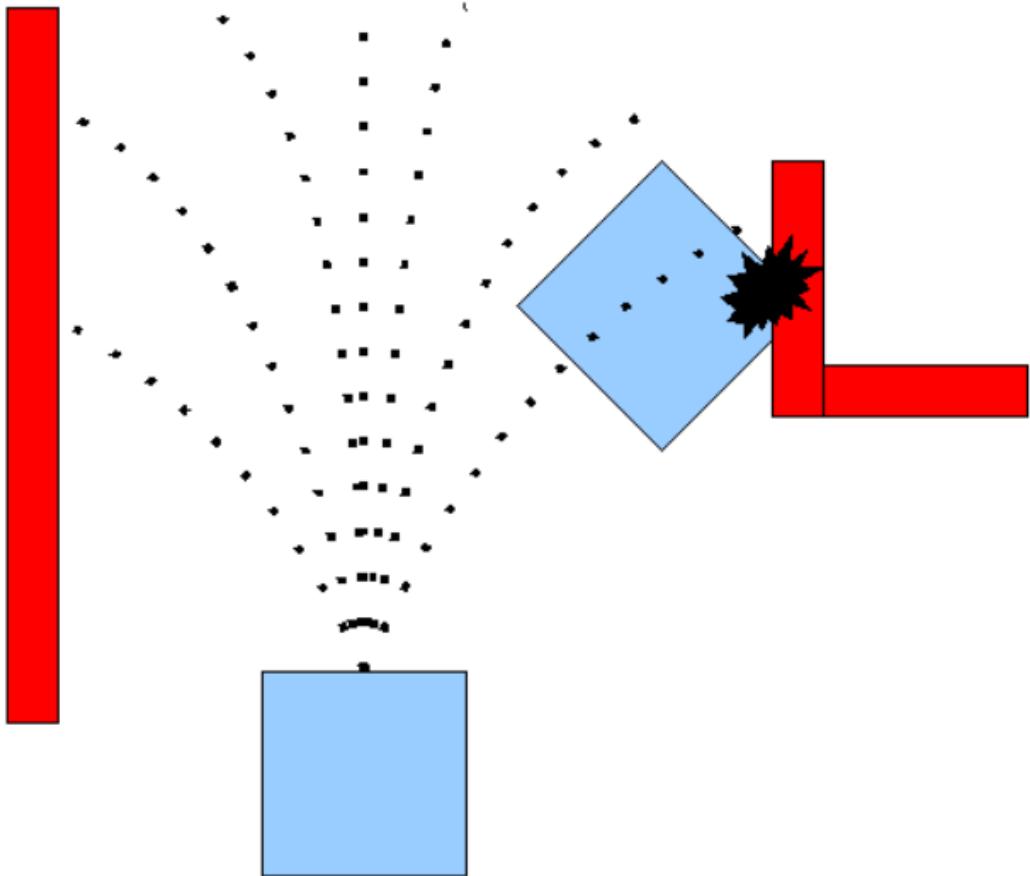
## 8.3 TEB 与 DWA 的比较

DWA 和TEB 共同点：

满足机器人的运动学，最小化代价函数，计算并且预测机器人运动轨迹，下发的控制速度也是用的预测轨迹中第一个速度指令

真正的最优解时很耗计算能力的，一般是用近似的方式简化模型和计算过程

	DWA	TEB
规划原理	速度采样	基于优化
输出轨迹性能	综合打分最优	时间最优（也可以通过调节来考虑其他因素）
轨迹形状	圆弧	elastic band (弹性带)
算力要求	较低	较高
适合的运动学模型	万向轮，差速	万向轮，差速，类车等
避障	静态，动态障碍物避障效果较差	静态，动态障碍物避障效果好



## 8.4 TEB和相关tutorials的安装和使用

---

### 8.4.1

- teb\_local\_planner
- teb\_local\_planner\_tutorials
- costmap\_converter

我们建议源码安装teb\_local\_planner和teb\_local\_planner\_tutorials包，以及costmap\_converter

```
cd ~/catkin_ws/src

git clone https://github.com/rst-tu-dortmund/teb_local_planner.git

git checkout <your_branch> # teb melodic 版本也可以在noetic上运行

git clone https://github.com/rst-tu-dortmund/tutorials.git

sudo apt-get install ros-melodic-stage-ros

git clone https://github.com/rst-tu-dortmund/costmap_converter.git
```

### 8.4.2 Teb\_local\_planner\_tutorial 讲解

teb\_local\_planner\_tutorials提供了一系列的仿真场景和案例

## 8.6 costmap\_converter

costmap-converter总共实现如下五种插件：

总共会生成五种插件，所有的class都是继承于BaseCostmapToPolygons：

CostmapToPolygonsDBSMCCH

将costmap转换为多边形，使用DBScan算法实现聚类，使用Monotone Chain Convex Hull检测凸多边形。

CostmapToLinesDBSMCCH

将costmap转换为line，使用DBScan算法实现聚类，使用Monotone Chain Convex Hull检测。

CostmapToLinesDBSRANSAC

使用RANSAC算法生成line

CostmapToPolygonsDBSConcaveHull

转换为非凸多边形

CostmapToDynamicObstacles

检测与追踪动态障碍物

DBSCAN: Density-based spatial clustering of applications with noise

costmap\_converter\_interfaces.h

- BaseCostmapToPolygons
- 纯虚函数：

```

virtual void initialize(ros::NodeHandle nh) = 0;
virtual void setCostmap2D(costmap_2d::Costmap2D* costmap) = 0; // 给插件传入costmap
virtual void updateCostmap2D() = 0; // 使用刚刚更新的costmap更新数据
virtual void compute() = 0; // 实际算法工作的函数，从costmap到polygon的转换

```

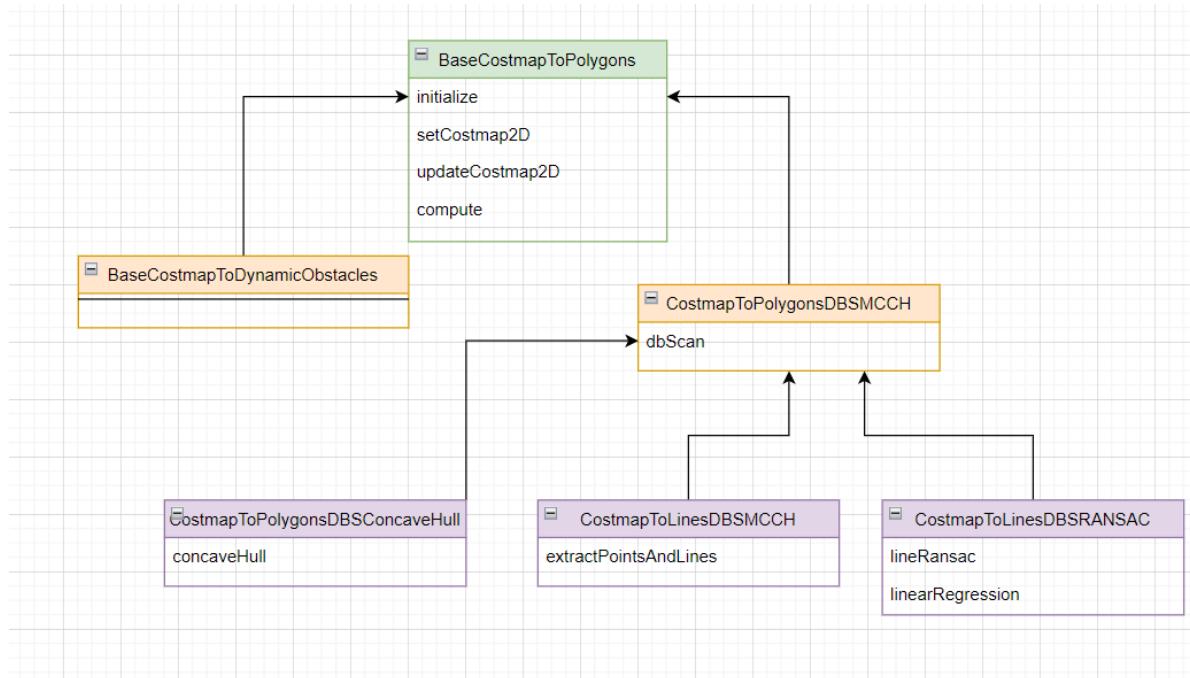
BaseCostmapToPolygons: costmap转换为多边形类型的插件接口类

BaseCostmapToDynamicObstacles: 动态costmap转换插件的类

CostmapToPolygonsDBSMCCH: costmap\_2d转换成凸多边形和点的集合

CostmapToLinesDBSMCCH: costmap\_2d转换成线段和点的集合

CostmapToLinesDBSRANSAC: costmap\_2d转换成线段和点的集合



## 第九章 TEB 源码分析 (Part 1)

**前言：**我认为你在这节课前已经掌握了哪些知识：

1. 如何安装，编译和使用 teb
2. 根据teb论文讲解，已经初步了解了teb是什么样的规划器，和最基本的流程

**本章知识点：**

- 9.1 在Move\_base中配置TEB
- 9.2 TEB 源码架构分析
- 9.3 TEB与ROS的接口
- 9.4 优化轨迹的TebOptimalPlanner类
- 9.5 时变弹性带TimedElasticBand

## 9.1 在Move\_base中配置TEB

### 预习资料

[http://wiki.ros.org/teb\\_local\\_planner/Tutorials](http://wiki.ros.org/teb_local_planner/Tutorials)

1. Set up and test Optimization (重要)
2. Inspect optimization feedback (重要)
3. Configure and run Robot Navigation (重要)
4. Obstacle Avoidance and Robot Footprint Model (重要)
5. Following the Global Plan (Via-Points)
6. Costmap conversion
7. Planning for car-like robots (重要)
8. Planning for holonomic robots
9. Incorporate customized Obstacles
10. Incorporate dynamic obstacles
11. Track and include dynamic obstacles via costmap\_converter
12. Frequently Asked Questions (重要)

### diff\_drive

```
TebLocalPlannerROS:  
  
    odom_topic: odom  
  
    # Trajectory  
  
    teb_autosize: True  
    dt_ref: 0.3  
    dt_hysteresis: 0.1  
    max_samples: 500  
    global_plan_overwrite_orientation: True  
    allow_init_with_backwards_motion: False  
    max_global_plan_lookahead_dist: 3.0  
    global_plan_viapoint_sep: -1  
    global_plan_prune_distance: 1  
    exact_arc_length: False  
    feasibility_check_no_poses: 5  
    publish_feedback: False  
  
    # Robot  
  
    max_vel_x: 0.4  
    max_vel_x_backwards: 0.2  
    max_vel_y: 0.0  
    max_vel_theta: 0.3  
    acc_lim_x: 0.5  
    acc_lim_theta: 0.5
```

```
min_turning_radius: 0.0 # diff-drive robot (can turn on place!)
```

```
footprint_model:
  type: "point"
```

```
# GoalTolerance
```

```
xy_goal_tolerance: 0.2
yaw_goal_tolerance: 0.1
free_goal_vel: False
complete_global_plan: True
```

```
# Obstacles
```

```
min_obstacle_dist: 0.25 # This value must also include our robot radius, since
footprint_model is set to "point".
inflation_dist: 0.6
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.5
obstacle_poses_affected: 15
```

```
dynamic_obstacle_inflation_dist: 0.6
include_dynamic_obstacles: True
```

```
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5
```

```
# Optimization
```

```
no_inner_iterations: 5
no_outer_iterations: 4
optimization_activate: True
optimization_verbose: False
penalty_epsilon: 0.1
obstacle_cost_exponent: 4
weight_max_vel_x: 2
weight_max_vel_theta: 1
weight_acc_lim_x: 1
weight_acc_lim_theta: 1
weight_kinematics_nh: 1000
weight_kinematics_forward_drive: 1
weight_kinematics_turning_radius: 1
weight_optimaltime: 1 # must be > 0
weight_shortest_path: 0
weight_obstacle: 100
weight_inflation: 0.2
weight_dynamic_obstacle: 10
weight_dynamic_obstacle_inflation: 0.2
weight_viapoint: 1
weight_adapt_factor: 2
```

```
# Homotopy Class Planner
```

```
enable_homotopy_class_planning: True
enable_multithreading: True
max_number_classes: 4
selection_cost_hysteresis: 1.0
```

```

selection_prefer_initial_plan: 0.9
selection_obst_cost_scale: 100.0
selection_alternative_time_cost: False

roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
roadmap_graph_area_length_scale: 1.0
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_heading_threshold: 0.45
switching_blocking_period: 0.0
viapoints_all_candidates: True
delete_detours_backwards: True
max_ratio_detours_duration_best_duration: 3.0
visualize_hc_graph: False
visualize_with_time_as_z_axis_scale: False

# Recovery

shrink_horizon_backup: True
shrink_horizon_min_duration: 10
oscillation_recovery: True
oscillation_v_eps: 0.1
oscillation_omega_eps: 0.1
oscillation_recovery_min_duration: 10
oscillation_filter_duration: 10

```

## carlike

```

TebLocalPlannerROS:

odom_topic: odom

# Trajectory

teb_autosize: True #优化期间允许改变轨迹的时域长度;
dt_ref: 0.3 #局部路径规划的解析度
dt_hysteresis: 0.1 #允许改变的时域解析度的浮动范围, 一般为 dt_ref 的 10% 左右;
max_samples: 500
global_plan_overwrite_orientation: True #覆盖全局路径中局部路径点的朝向
allow_init_with_backwards_motion: True #允许在开始时想后退来执行轨迹
max_global_plan_lookahead_dist: 3.0 #考虑优化的全局计划子集的最大长度（累积欧几里得距离）（如果为0或负数：禁用；长度也受本地Costmap大小的限制）
global_plan_viapoint_sep: -1 #发布包含完整轨迹和活动障碍物列表的规划器反馈
global_plan_prune_distance: 1
exact_arc_length: False #如果为真, 规划器在速度、加速度和转弯率计算中使用精确的弧长[>增加的CPU时间], 否则使用欧几里德近似。
feasibility_check_no_poses: 2 #检测位姿可到达的时间间隔
publish_feedback: False #发布包含完整轨迹和活动障碍物列表的规划器反馈

# Robot

max_vel_x: 0.4 #最大x前向速度
max_vel_x_backwards: 0.2 #最大x后退速度
max_vel_y: 0.0 #最大y方向速度
max_vel_theta: 0.3 # the angular velocity is also bounded by min_turning_radius
in case of a carlike robot (r = v / omega) 最大转向角速度

```

```

acc_lim_x: 0.5 #最大x加速度
acc_lim_theta: 0.5 #最大角速度

# **** Carlike robot parameters ****
min_turning_radius: 0.5      # Min turning radius of the carlike robot
(compute value using a model or adjust with rqt_reconfigure manually) 车类机器人的
最小转弯半径
wheelbase: 0.4                # wheelbase of our robot 驱动轴和转向轴之间的距离（仅
适用于启用了“Cmd_angle_而不是_rotvel”的Carlike机器人）；对于后轮式机器人，该值可能为负值
cmd_angle_instead_of_rotvel: True # stage simulator takes the angle instead of the
rotvel as input (twist message)将收到的角速度消息转换为 操作上的角度变化
# *****

footprint_model: # types: "point", "circular", "two_circles", "line", "polygon"
  type: "line"
  radius: 0.2 # for type "circular"
  line_start: [0.0, 0.0] # for type "line"
  line_end: [0.4, 0.0] # for type "line"
  front_offset: 0.2 # for type "two_circles"
  front_radius: 0.2 # for type "two_circles"
  rear_offset: 0.2 # for type "two_circles"
  rear_radius: 0.2 # for type "two_circles"
  vertices: [ [0.25, -0.05], [0.18, -0.05], [0.18, -0.18], [-0.19, -0.18],
[-0.25, 0], [-0.19, 0.18], [0.18, 0.18], [0.18, 0.05], [0.25, 0.05] ] # for type
"polygon"

# GoalTolerance

xy_goal_tolerance: 0.2 #目标 xy 偏移容忍度
yaw_goal_tolerance: 0.1 #目标 角度 偏移容忍度
free_goal vel: False #允许机器人以最大速度驶向目的地
complete_global_plan: True

# Obstacles

min_obstacle_dist: 0.27 # This value must also include our robot's expansion,
since footprint_model is set to "line". 和障碍物最小距离
inflation_dist: 0.6 #障碍物膨胀距离
include_costmap_obstacles: True #costmap 中的障碍物是否被直接考虑
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 15

dynamic_obstacle_inflation_dist: 0.6 #动态障碍物的膨胀范围
include_dynamic_obstacles: True #是否将动态障碍物预测为速度模型

costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5

# Optimization

no_inner_iterations: 5 #被外循环调用后内循环执行优化次数
no_outer_iterations: 4 #执行的外循环的优化次数执行的外循环的优化次数
optimization_activate: True #激活优化
optimization_verbose: False #打印优化过程详情
penalty_epsilon: 0.1 #对于硬约束近似，在惩罚函数中添加安全范围
obstacle_cost_exponent: 4
weight_max_vel_x: 2 #最大x速度权重

```

```

weight_max_vel_theta: 1 #最大角速度权重
weight_acc_lim_x: 1 #最大x 加速度权重
weight_acc_lim_theta: 1 #最大角速度权重
weight_kinematics_nh: 1000 #Optimization weight for satisfying the non-
holonomic kinematics
weight_kinematics_forward_drive: 1 #优化过程中，迫使机器人只选择前进方向，差速轮适用
weight_kinematics_turning_radius: 1 #优化过程中，车型机器人的最小转弯半径的权重
weight_optimaltime: 1 # must be > 0 #优化过程中，基于轨迹的时间上的权重
weight_shortest_path: 0
weight_obstacle: 100 #优化过程中，和障碍物最小距离的权重
weight_inflation: 0.2 #优化过程中，膨胀区的权重
weight_dynamic_obstacle: 10 #优化过程中，和动态障碍物最小距离的权重
weight_dynamic_obstacle_inflation: 0.2 #优化过程中，和动态障碍物膨胀区的权重
weight_viapoint: 1 #优化过程中，和全局路径采样点距离的权重
weight_adapt_factor: 2

# Homotopy Class Planner

enable_homotopy_class_planning: True #允许的线程数
enable_multithreading: True #允许多线程并行处理
max_number_classes: 4
selection_cost_hysteresis: 1.0
selection_prefer_initial_plan: 0.95
selection_obst_cost_scale: 1.0
selection_alternative_time_cost: False

roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
roadmap_graph_area_length_scale: 1.0
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_heading_threshold: 0.45
switching_blocking_period: 0.0
viapoints_all_candidates: True
delete_detours_backwards: True
max_ratio_detours_duration_best_duration: 3.0
visualize_hc_graph: False
visualize_with_time_as_z_axis_scale: False

# Recovery

shrink_horizon_backup: True #当规划器检测到系统异常，允许缩小时域规划范围00
shrink_horizon_min_duration: 10
oscillation_recovery: True #尝试检测和解决振荡
oscillation_v_eps: 0.1
oscillation_omega_eps: 0.1
oscillation_recovery_min_duration: 10
oscillation_filter_duration: 10

```

## 9.2 TEB 源码架构分析

学习工具：授人以鱼不如授人以渔，通过用类图的方法更好的理解大工程系统。

[http://docs.ros.org/en/noetic/api/teb\\_local\\_planner/html/](http://docs.ros.org/en/noetic/api/teb_local_planner/html/)

## 9.3 TEB与ROS的接口

类视图

[http://docs.ros.org/en/noetic/api/teb\\_local\\_planner/html/classteb\\_local\\_planner\\_1\\_1TebLocalPlannerROS.html](http://docs.ros.org/en/noetic/api/teb_local_planner/html/classteb_local_planner_1_1TebLocalPlannerROS.html)

## 9.4 优化轨迹的TebOptimalPlanner类

类视图

[http://docs.ros.org/en/noetic/api/teb\\_local\\_planner/html/classteb\\_local\\_planner\\_1\\_1TebOptimalPlanner.html#details](http://docs.ros.org/en/noetic/api/teb_local_planner/html/classteb_local_planner_1_1TebOptimalPlanner.html#details)

## 9.5 时变弹性带TimedElasticBand

类视图

[http://docs.ros.org/en/noetic/api/teb\\_local\\_planner/html/classteb\\_local\\_planner\\_1\\_1TimedElasticBand.html](http://docs.ros.org/en/noetic/api/teb_local_planner/html/classteb_local_planner_1_1TimedElasticBand.html)

# 第十章 TEB 源码分析 (Part 2)

## 预备准备

下载编译g2o

```
sudo apt-get install libsuiteparse-dev
git clone https://github.com/RainerKuemmerle/g2o.git
cd g2o
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=RELEASE
make -j8
sudo make install
```

## teb中主要9个目标函数

- 1.运动学
- 2.时间最短
- 3.速度最快
- 4.观测误差
- 5.路径最短
- 6.转向偏好
- 7.障碍距离
- 8.加速度最大
- 9.动态障碍

## 其他材料

推荐的参考博客

[https://epsavlc.github.io/2019/08/06/tel\\_g2o.html](https://epsavlc.github.io/2019/08/06/tel_g2o.html)

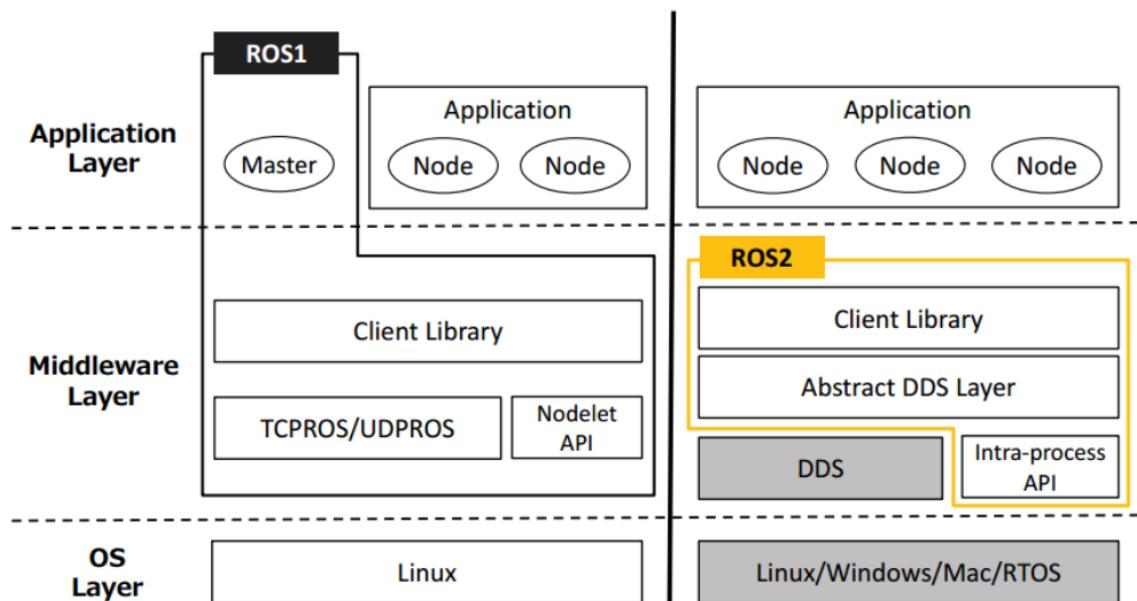
同时建议结合导读课的tel论文，和视频代码，一起解读的看。

# 第十一章 ROS2 和 Navigation2

## 11.1 ROS1和ROS2

ROS2官网

<https://docs.ros.org/en/foxy/Docs-Guide.html>



ROS1 和 ROS2区别：

1. ROS1主要构建在Linux系统之上，但是ROS2支持Linux、windows、Mac等等。
2. ROS1的通讯系统基于TCPROS/UDPROS，依赖于master节点的处理，master节点挂了，系统没法运行。ROS2的架构中没有master节点，ROS2的通讯系统是基于DDS的。
3. ROS中最重要的概念就是node，基于发布/订阅模型的节点使用，可以让开发者并行开发低耦合的功能模块，并且便于进行二次复用。得益于DDS的加入，ROS2的发布/订阅模型也会发生改变。

**DDS定义：**Data Distribution Service 数据分发服务，是新一代分布式实时通信中间件协议，采用发布/订阅体系架构，强调以数据为中心，提供丰富的QoS服务质量策略，可满足各种分布式实时通信应用需求。

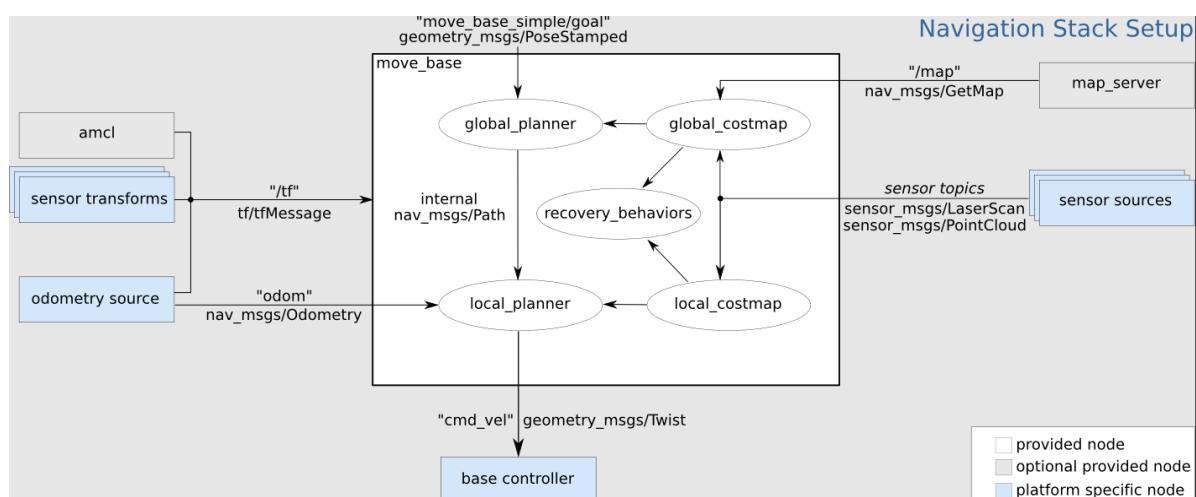
1. 参与者 (Domain Participant) : 一个参与者Participant就是一个容器，对应于一个使用DDS的用户，任何DDS的用户都必须通过Participant来访问全局数据空间。
2. 发布者 (Publisher) : 数据发布的执行者，支持多种数据类型的发布，可以与多个数据写入器 (DataWriter) 相联，发布一种或多种主题 (Topic) 的消息。
3. 订阅者 (Subscriber) : 数据订阅的执行者，支持多种数据类型的订阅，可以与多个数据读取器 (DataReader) 相联，订阅一种或多种主题 (Topic) 的消息。
4. 数据写入器 (DataWriter) : 应用向发布者更新数据的对象，每个数据写入器对应一个特定的 Topic，类似于ROS1中的一个消息发布者。
5. 数据读取器 (DataReader) : 应用从订阅者读取数据的对象，每个数据读取器对应一个特定的 Topic，类似于ROS1中的一个消息订阅者。
6. 主题 (Topic) : 这个和ROS1中的Topic概念一致，一个Topic包含一个名称和一种数据结构。
7. QoS Policy: Quality of Service，质量服务原则。QoS是DDS中非常重要的一环，控制了各方面与底层的通讯机制，主要从时间限制、可靠性、持续性、历史记录几个方面，满足用户针对不同场景的数据应用需求。

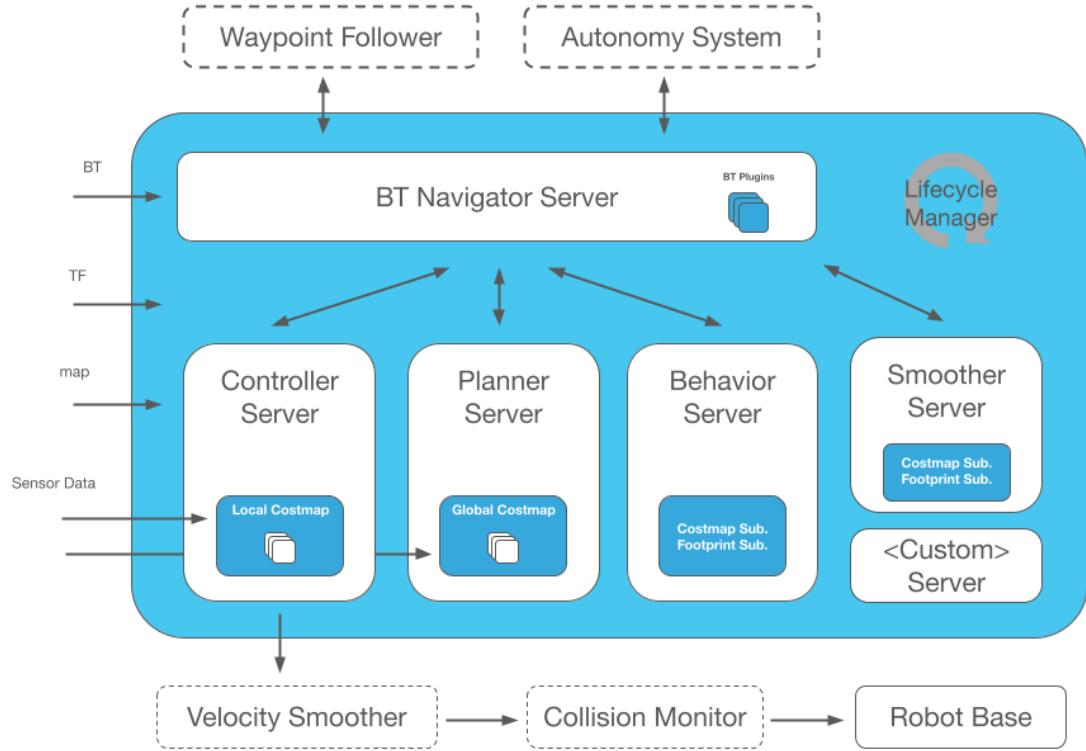
ROS2相对于ROS1的改善：

- 实时性增强：数据必须在deadline之前完成更新。
- 持续性增强：ROS1尽管存在数据队列的概念，但是还有很大的局限，订阅者无法接收到加入网络之前的数据；DDS可以为ROS提供数据历史的服务，就算新加入的节点，也可以获取发布的所有历史数据。
- 可靠性增强：通过DDS配置可靠性原则，用户可以根据需求选择性能模式（BEST EFFORT）或者稳定模式（RELIABLE）

## 11.2 行为树和状态机

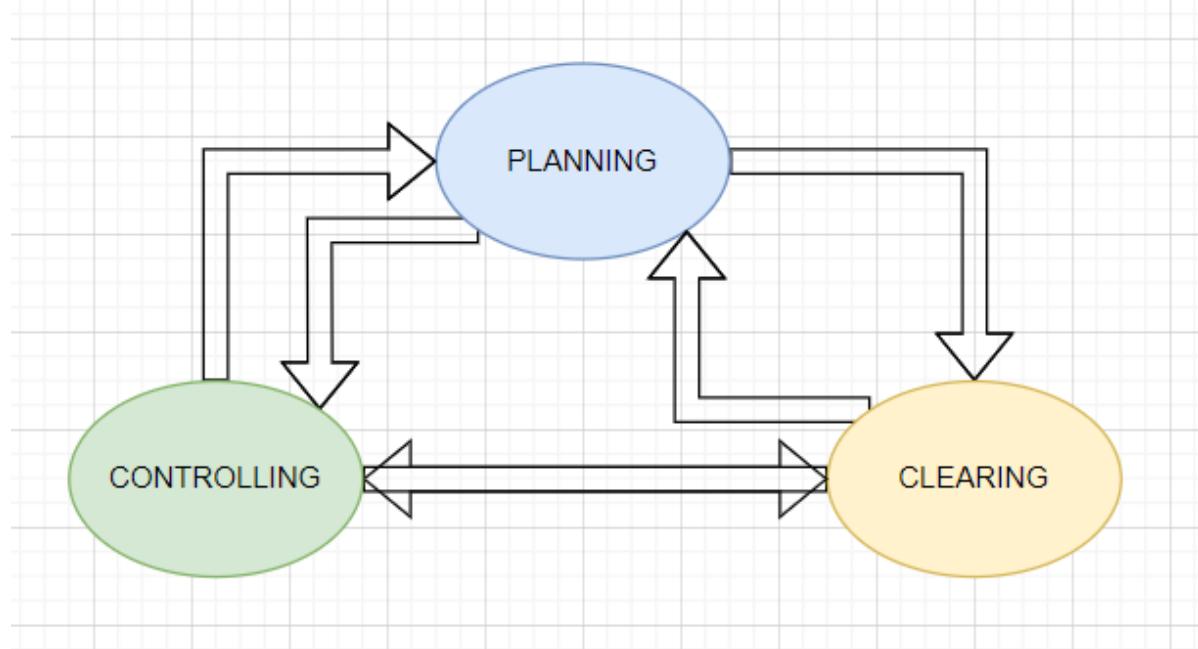
**navigation stack 和 navigation2区别**





move\_base中

- PLANNING
- CONTROLLING
- CLEARING



## **movebase状态之间 (PLANNING, CONTROLLING, CLEARING ) 的切换条件**

- 切换到CLEARING:

1. 得到路径但计算不出下一步控制时重新进行路径规划
2. 没有超时，但是没有找到有效全局路径

- 切换到CONTROLLING:

获得了全局路径，并且没有到达目标点

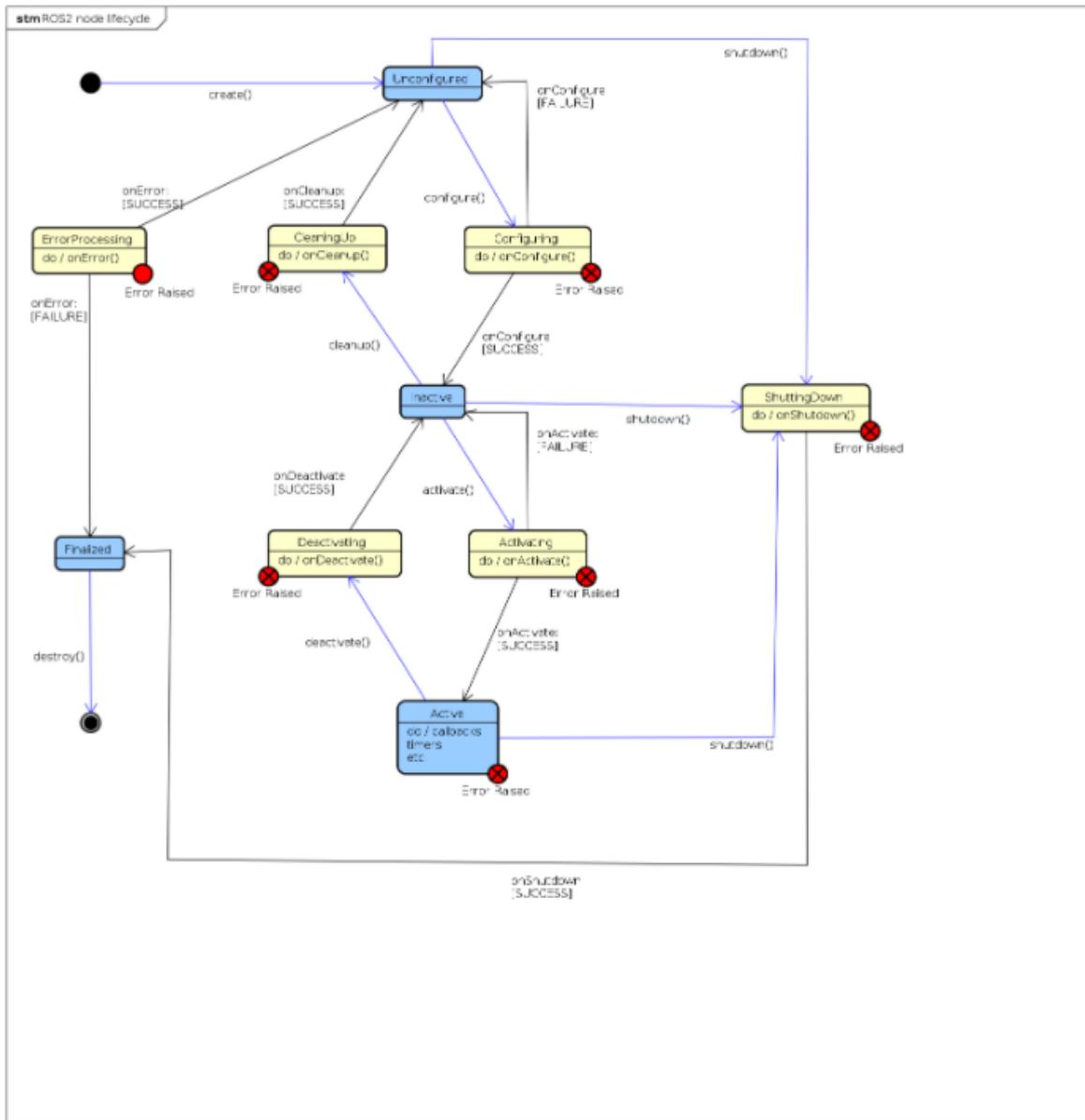
- 切换到PLANNING:

1. 构造函数初始化
2. 获得新的目标点
3. 目标点的坐标系和全局坐标系不一致
4. 执行recovery之后
5. resetState

[https://navigation.ros.org/behavior\\_trees/overview/nav2\\_specific\\_nodes.html](https://navigation.ros.org/behavior_trees/overview/nav2_specific_nodes.html)

## **11.3 lifecycle manager**

[https://design.ros2.org/articles/node\\_lifecycle.html](https://design.ros2.org/articles/node_lifecycle.html)



There are 4 primary states:

- `Unconfigured`
- `Inactive`
- `Active`
- `Finalized`

使用 ROS2 的 managed/lifecycle nodes 功能允许系统启动以确保所有需要的节点在开始执行之前都已正确实例化。使用 lifecycle nodes 还可以在线重启或更换节点。Nav2 中的几个节点，例如 map\_server、planner\_server 和 controller\_server，提供了 lifecycle node 的管理。这些节点提供了生命周期函数所需的重写：

`on_configure()`, `on_activate()`, `on_deactivate()`, `on_cleanup()`, `on_shutdown()`, and `on_error()`.

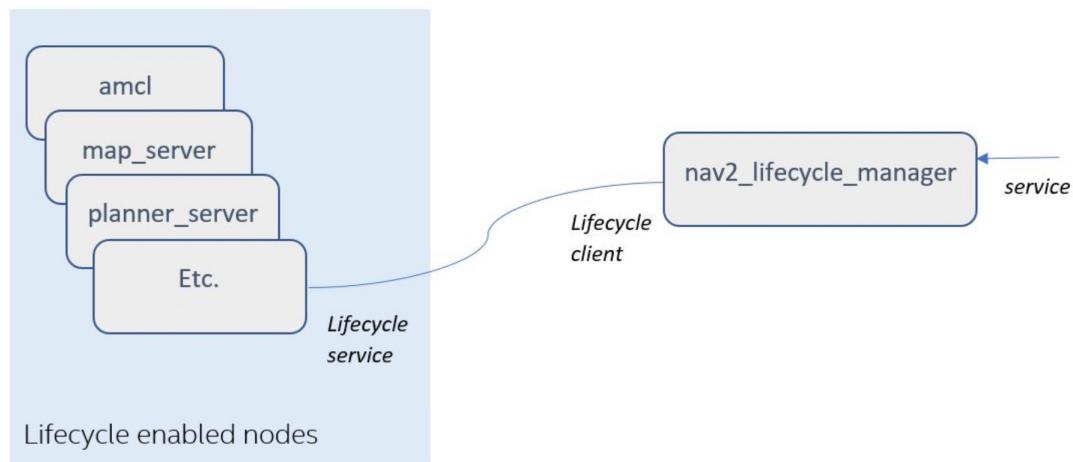
## nav2\_lifecycle\_manager

Nav2 的 lifecycle manager 用于改变 lifecycle 节点的状态，以实现 navigation stack 的可控启动、关闭、重置、暂停或恢复。lifecycle manager 提供了一个 lifecycle\_manager/manage\_nodes service，根据这个服务请求，客户端可以从中调用生命周期管理器，在生命周期被管节点中调用必要的生命周期服务。在按下 RVIZ 界面上的按钮（例如，startup, reset, shutdown 等）时使用这个 lifecycle\_manager/manage\_nodes。

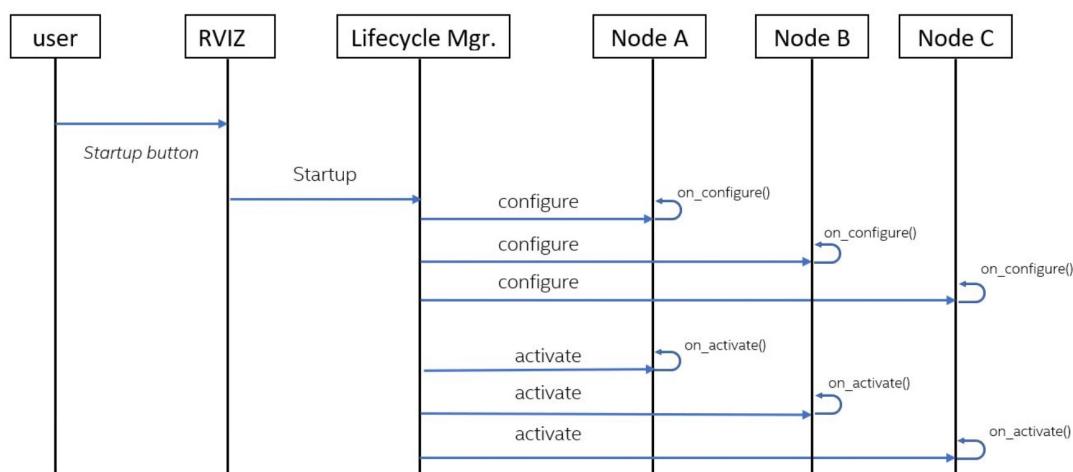
为了启动 navigation stack 并能够导航，必须要设置节点状态为 configured 和 activated。因此，例如当生命周期管理器的 manage\_nodes 服务请求 start\_up 时，lifecycle manager 会调用节点列表中 lifecycle enabled nodes 的 configure() 和 activate()。

生命周期管理器对其管理的所有节点都有一个默认节点列表。可以使用生命周期管理器的 “node\_names” 参数更改此列表。

下图显示了托管节点列表的示例，以及它如何与生命周期管理器交互。



下面的 UML 图显示了从生命周期管理器请求启动后的服务调用顺序。



## 11.4 planner server和controller server

全局规划器和局部规划器从navigation到navigation2的变化：

	<b>navigation</b>	<b>navigation2</b>
全局规划	专门建立了一个全局规划的线程进行全局路径规划	planner server(lifecycle node)
局部规划	MoveBaseState切换到CONTROLLING状态时，开始局部轨迹规划	controller server(lifecycle node)
全局路径数据传递到局部规划器的方式	通过在MoveBase类中，拷贝传递	通过行为树

### action\_server

<https://navigation.ros.org/concepts/index.html#navigation-servers>

## 第十二章 总结和展望

### 12.1 移动机器人应用场景

- 服务领域：消毒，清洁机器人
- 家居领域：扫地机器人
- 仓储物流
- 自动驾驶
- 航空领域
- 水下机器人
- 农业领域

## 12.2 自动驾驶相关平台介绍

### ROS在自动驾驶上的问题？

1. 单点失败问题
2. 带宽拥塞
3. 安全问题

### Apollo架构

[http://www.apollo.auto/developer/index\\_cn.html#/](http://www.apollo.auto/developer/index_cn.html#/)

Apollo自动驾驶开放平台是一个开放的、完整的、安全的自动驾驶开源平台。代码已经跑通了园区物流、自动泊车、园区接驳、智慧农业、高速物流、健康养老等场景，并稳步面向量产和运营。Apollo推出面向量产的人工智能车联网系统解决方案小度车载OS，具备开放语音、语义、多模交互、车载信息安全、驾驶员检测五大核心能力。

Apollo 的 Cyber RT 框架是基于组件概念来构建的。每个组件都是 Cyber RT 框架的一个 特定的算法模块，处理一组输入并产生其输出数据。

简单点说，apollo把每个模块分成一个一个的component，大体分成dreamview、drivers、guardian、localization、map、monitor、perception、planning、prediction、routing等模块，以上模块均独立成为一个组件，驱动模块又分成camera、lidar、radar、canbus、gnss等组件。

开源软件平台是Apollo自动驾驶系统的核心部分，包括功能模块、运行框架和实时操作系统三部分。功能模块可细分为：

- 地图引擎：运行高精度地图；
- 定位模块：通过GPS、V-SLAM、L-SLAM、里程计等多种定位源融合，结合高精度地图，实现精准定位；
- 感知：通过激光雷达、毫米波雷达、摄像头，精确感知车辆周围的环境路况，包括车辆、行人、交通标志等等；
- 规划：主要包括路径规划、运动障碍物的预测等；
- 控制：实现控制车辆的转向、油门、刹车等操作；End-to-End：基于深度学习的横向和纵向驾驶模型；
- HMI：人机交互模块。

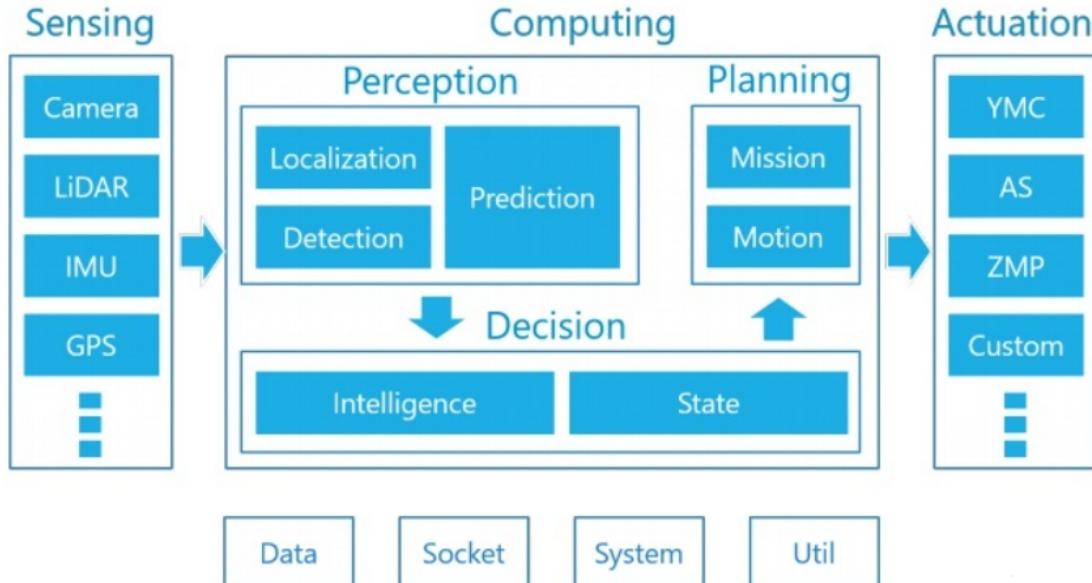
# Apollo技术框架

Cloud Service Platform	HD Map	Simulation	Data Platform	Security	OTA	DuerOS
Open Software Platform	Map Engine	Localization	Perception	Planning	Control	End-to-End
	Runtime Framework					
	RTOS					
Reference Hardware Platform	Computing Unit	GPS/IMU	Camera	LiDAR	Radar	HMI Device
Reference Vehicle Platform	Drive-by-wire Vehicle					

## Autoware架构

<https://www.autoware.org/>

Autoware是世界上第一款用于自动驾驶汽车的“一体化”开源软件。Autoware的功能主要适用于城市，但高速公路和非市政道路同样可以使用。同时在依赖ROS建立的Autoware自动驾驶开源软件上可以提供丰富的开发和使用资源



## 12.3 其他的规划算法

ROS2中新增的算法：

- smac\_planner
  - Hybrid A star
- nav2\_theta\_star\_planner
- Regulated Pure pursuit controller
- nav2\_dwb\_controller
- nav2\_waypoint\_follower

Apollo中的规划算法：

EM planner

<https://arxiv.org/abs/1807.08048>

Lattice planner

1. 采样候选的轨迹
2. 计算对应的代价
3. 选择代价最低的轨迹
4. 检测是否满足约束，碰撞检测
5. 输出

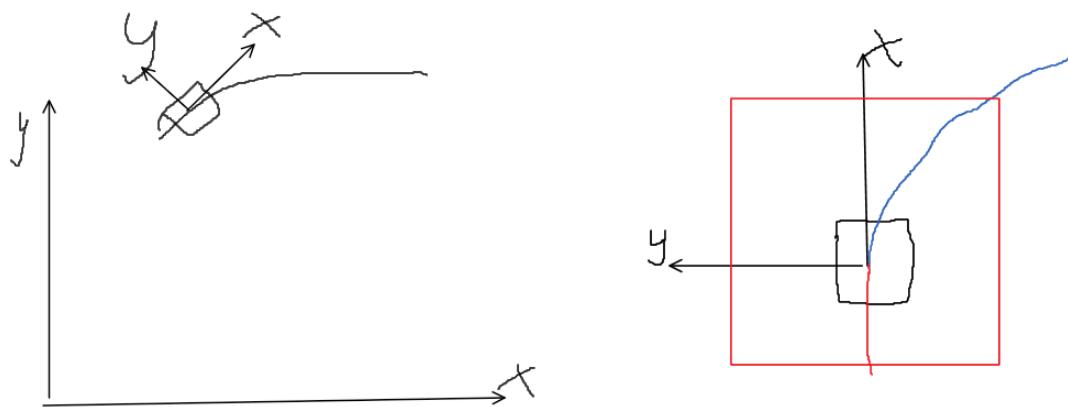
RRT

[https://github.com/hasauino/rrt\\_exploration](https://github.com/hasauino/rrt_exploration)

## 12.4 Regulated\_pure\_pursuit\_controller

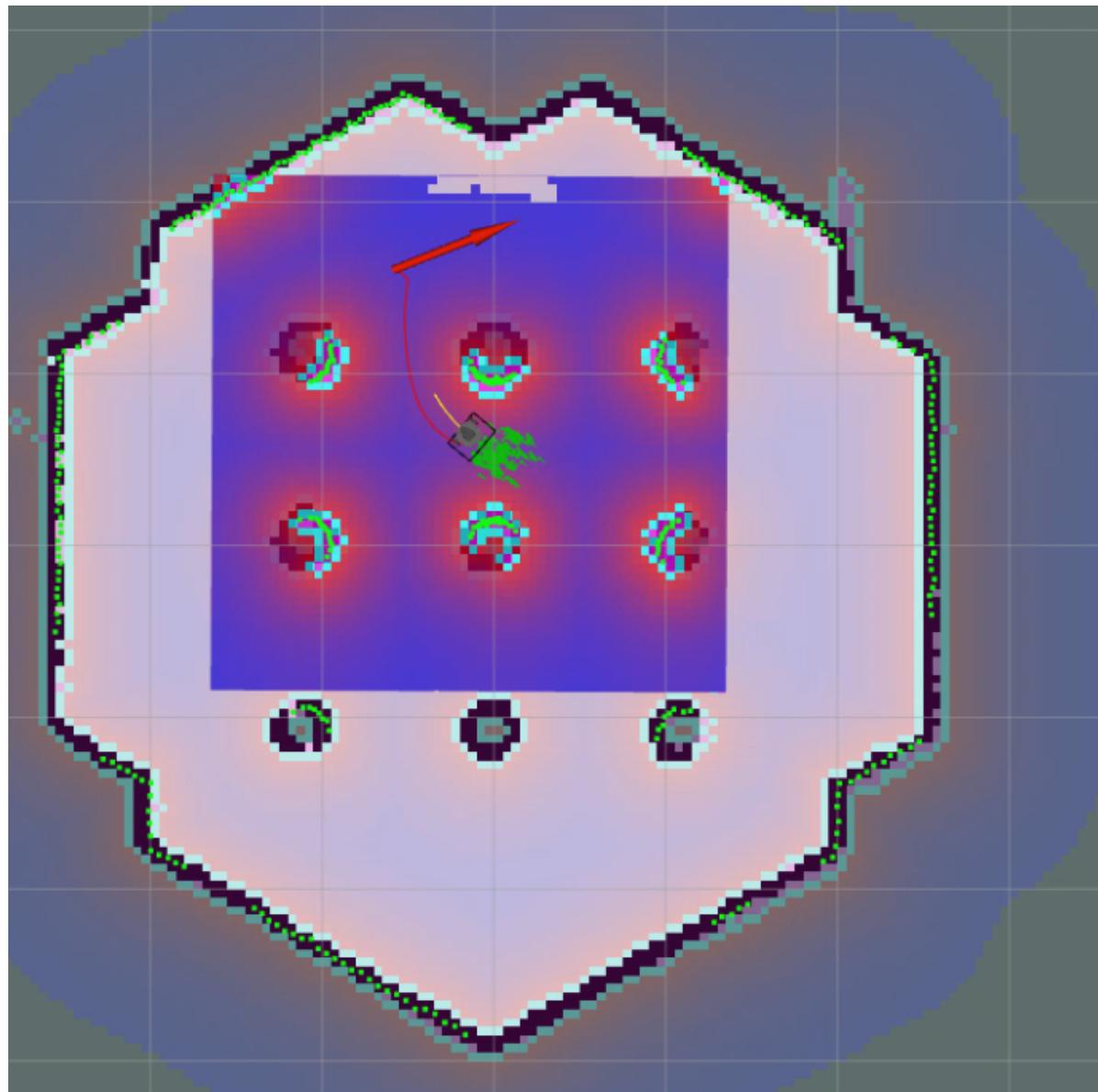
- 获得全局路径
- 转换全局路径到base坐标系下
- 获得前瞻距离 (lookahead\_dist)
- 检查是否后退

- 获得前瞻点
- 计算前瞻点到机器人距离的平方，再算出curvature
- 约束检测（是否满足之前设定的约束）
  - 是否需要旋转与目标点朝向对齐
  - 是否需要旋转和路径对齐
  - 使用applyConstraints函数进行约束检测
    - 通过曲率限制线速度，获得curvature\_vel
    - 靠近了障碍物，调节线速度，获得cost\_vel
    - 取curvature\_vel和cost\_vel中较小值，但不能小于最小移动速度regulated\_linear\_scaling\_min\_speed\_
    - 如果实际的前瞻距离dist\_error小于期望的，根据dist\_error求出approach\_vel ----> linear\_vel



## 12.5 关于navigation工程的个人看法

为什么运动规划分成了全局和局部规划



nav\_core接口

算法用插件化的方式加载

move\_base的架构

跳转逻辑太多

## movebase状态之间（PLANNING, CONTROLLING, CLEARING）的切换条件

- 切换到 CLEARING:

1. 得到路径但计算不出下一步控制时重新进行路径规划
2. 没有超时，但是没有找到有效全局路径

- 切换到CONTROLLING:

获得了全局路径，并且没有到达目标点

- 切换到PLANNING:
  1. 构造函数初始化
  2. 获得新的目标点
  3. 目标点的坐标系和全局坐标系不一致
  4. 执行recovery之后
  5. resetState

recovery\_behavior插件可以增减，顺序可以调整

使用单例模式和工厂模式

planner\_frequency和controller\_frequency要设置成多少

costmap 的layers要用到哪些

是否可以加上控制

global\_planner和navfn

## 附录： 安装环境

### 1. 安装Ubuntu系统

推荐双系统，最好不在虚拟机里跑Ubuntu系统。

推荐安装ubuntu的LTS版本，ubuntu18.04, ubuntu 20.04。

### 2. ROS系统安装

有如下有不同的ros版本对应不同的ubuntu版本

- kinetic (ubuntu16.04) ,
- melodic (ubuntu18.04)
- noetic (ubuntu20.04)

## 官方安装指导

<http://wiki.ros.org/cn/ROS/Installation>

**完整桌面版安装 (Desktop-Full, 推荐)**：除了**桌面版**的全部组件外，还包括2D/3D模拟器(simulator) 和2D/3D 感知包 (perception package)。

```
sudo apt install ros-noetic-desktop-full
```

**桌面版 (Desktop)**：包括了**ROS-Base**的全部组件，还有一些工具，比如[rqt](#)和[rviz](#)。

```
sudo apt install ros-noetic-desktop
```

**ROS-Base (仅含骨架)**：ROS packaging, build, 和communication库。没有图形界面 (GUI) 工具。

```
sudo apt install ros-noetic-ros-base
```

**注意：**记住换源 <https://developer.aliyun.com/mirror/ubuntu>

编辑器打开：

```
/etc/apt/sources.list
```

替换默认的

```
http://archive.ubuntu.com/
```

为

```
mirrors.aliyun.com
```

ubuntu 20.04(focal) 配置

```
deb http://mirrors.aliyun.com/ubuntu/ focal main restricted universe multiverse
deb-src http://mirrors.aliyun.com/ubuntu/ focal main restricted universe
multiverse

deb http://mirrors.aliyun.com/ubuntu/ focal-security main restricted universe
multiverse
deb-src http://mirrors.aliyun.com/ubuntu/ focal-security main restricted universe
multiverse

deb http://mirrors.aliyun.com/ubuntu/ focal-updates main restricted universe
multiverse
deb-src http://mirrors.aliyun.com/ubuntu/ focal-updates main restricted universe
multiverse
```

```
deb http://mirrors.aliyun.com/ubuntu/ focal-proposed main restricted universe  
multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ focal-proposed main restricted universe  
multiverse  
  
deb http://mirrors.aliyun.com/ubuntu/ focal-backports main restricted universe  
multiverse  
deb-src http://mirrors.aliyun.com/ubuntu/ focal-backports main restricted  
universe multiverse
```

安装好后，环境查看

```
printenv | grep ROS
```

环境变量设置文件

```
source /opt/ros/<distro>/setup.bash # 如果是apt 安装的  
  
source catkin_ws/devel/setup.bash # 如果是使用自己源码编译的包
```

创建和构建一个catkin工作空间

```
mkdir -p ~/catkin_ws/src  
cd ~/catkin_ws/  
catkin_make  
source devel/setup.bash
```

### 3. Navigation安装

**方式一：**通过apt-get安装编译好的包；

```
sudo apt-get install ros-noetic-navigation*
```

ROS系统搭好后，如果缺少什么包就

```
sudo apt-get install ros-noetic-xxx      # Ubuntu 20.04, noetic ROS版本  
sudo apt-get install ros-melodic-xxx     # Ubuntu 18.04, melodic ROS版本  
sudo apt-get install ros-kinetic-xxx      # Ubuntu 16.04, kinetic ROS版本
```

**方式二：**通过源码编译安装-推荐。会安装如下包：

或者我们已经注释好了的navigation仓库

<https://github.com/felderstehhost/navigation-noetic>

```
# git clone 到~/catkin_ws/src下面  
# 编译  
cd ~/catkin_ws/  
catkin_make
```

如果编译的过程报错，缺少包 aaaa\_bbb\_ccc

可以直接命令

```
sudo apt-get install ros-noetic-aaa-bbb-ccc
```

缺少什么就安装什么，很方便哈

直到编译通过。

## 4. Teb安装

我们建议源码安装teb\_local\_planner和teb\_local\_planner\_tutorials包，以及costmap\_converter

```
cd ~/catkin_ws/src  
git clone https://github.com/rst-tu-dortmund/teb_local_planner.git  
git checkout <your_branch> # teb melodic 版本也可以在noetic上运行  
git clone https://github.com/rst-tu-dortmund/teb_local_planner_tutorials.git  
sudo apt-get install ros-melodic-stage-ros  
git clone https://github.com/rst-tu-dortmund/costmap_converter.git
```

检查和安装依赖项

```
rosdep check teb_local_planner  
rosdep install teb_local_planner
```

编译

```
cd ~/catkin_ws  
catkin_make
```

如果编译报错，缺少什么包就apt install。直到编译通过。

# Docker

---

## Docker是什么？

Docker是一个虚拟环境容器，可以将开发环境、代码、配置文件等一并打包到这个容器中，并发布和应用到任意平台中。

**Docker 的优点 参考：**<https://www.runoob.com/docker/docker-tutorial.html>

Docker 是一个用于开发，交付和运行应用程序的开放平台。Docker 使您能够将应用程序与基础架构分开，从而可以快速交付软件。借助 Docker，您可以与管理应用程序相同的方式来管理基础架构。通过利用 Docker 的方法来快速交付，测试和部署代码，您可以大大减少编写代码和在生产环境中运行代码之间的延迟。

**Docker的三个概念（参考<https://zhuanlan.zhihu.com/p/23599229>）**

1. 镜像 (Image)：类似于虚拟机中的镜像，是一个包含有文件系统的面向Docker引擎的只读模板。任何应用程序运行都需要环境，而镜像就是用来提供这种运行环境的。例如一个ROS镜像就是一个包含ROS系统环境的模板。
2. 容器 (Container)：类似于一个轻量级的沙盒，可以将其看作一个极简的Linux系统环境（包括root权限、进程空间、用户空间和网络空间等），以及运行在其中的应用程序。Docker引擎利用容器来运行、隔离各个应用。容器是镜像创建的应用实例，可以创建、启动、停止、删除容器，各个容器之间是相互隔离的，互不影响。注意：镜像本身是只读的，容器从镜像启动时，Docker在镜像的上层创建一个可写层，镜像本身不变。
3. 仓库 (Repository)：类似于代码仓库，这里是镜像仓库，是Docker用来集中存放镜像文件的地方。注意与注册服务器 (Registry) 的区别：注册服务器是存放仓库的地方，一般会有多个仓库；而仓库是存放镜像的地方，一般每个仓库存放一类镜像，每个镜像利用tag进行区分，比如Ubuntu仓库存放有多个版本（12.04、14.04等）的Ubuntu镜像。

## 安装docker

### 在ubuntu中 安装docker

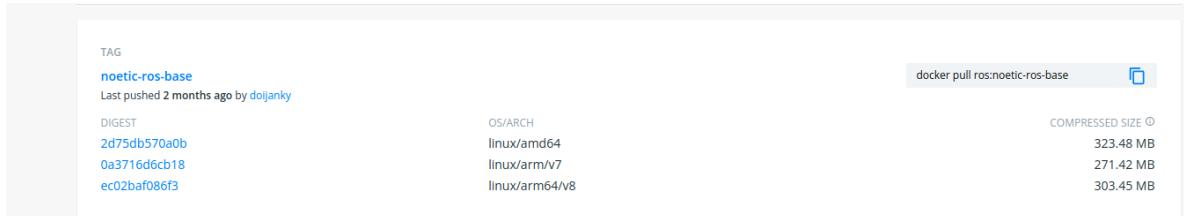
参考链接 <https://www.runoob.com/docker/ubuntu-docker-install.html>

docker安装好了之后可以用命令 docker pull 拉取相应的镜像。

## 去Docker Hub拉取ros的docker镜像

ros的docker镜像地址: [https://hub.docker.com/\\_/ros/](https://hub.docker.com/_/ros/)

```
# 下面三个镜像拉取命令选其中一个
sudo docker pull ros:noetic-ros-base # 推荐
sudo docker pull ros:noetic-robot-focal
sudo docker pull ros:noetic-robot
```



等拉取的进度条结束后可以查看一下镜像:

```
sudo docker images
```

有了镜像，接下来用镜像创建容器，我们先写一个启动脚本，方便创建。

## 容器脚本

建立自己的容器启动脚本 new\_container.sh:

```
vim new_container.sh
```

将下面的脚本命令复制进去，:wq保存退出。

```
C_NAME=$1
docker run -d --name ${C_NAME} --privileged --network host -v $HOME:$HOME -it \
ros:noetic-ros-base /bin/bash
docker attach ${C_NAME}
```

## 用镜像建立容器

```
sudo sh ./new_container.sh noetic # 创建一个名字为noetic的容器
sudo docker ps -a # 查看所有容器
```

每次容器相关的操作都要输入sudo,这样太麻烦，可以设置一下，免去每次都加sudo。

[设置非root账号不用sudo直接执行docker命令](#)

## 在docker容器中，安装其他需要的ros包

### 安装navigation 和rviz

```
sudo apt update # apt-get update --fix-missing  
sudo apt-get install ros-noetic-navigation*  
  
sudo apt-get install ros-noetic-rviz
```

### 安装gazebo的接口相关模块

```
sudo apt-get install ros-noetic-gazebo-ros-pkgs ros-noetic-gazebo-ros-control
```

### 安装turtlebot的相关模块

```
sudo apt-get install ros-noetic-turtlebot-*
```

#### 设置模型

```
export TURTLEBOT3_MODEL=burger
```

如果打不开仿真或者rviz，输入如下命令

```
xhost +  
# apt-get install ros-noetic-gazebo*
```