

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 ROS 的基本架构	1
1.1 整体架构	1
1.2 计算图的视角	2
1.2.1 节点 Node	2
1.2.2 话题 Topic	2
1.2.3 服务 Service	2
1.2.4 动作 Action	2
1.2.5 节点管理器 Master	2
1.3 文件系统	2
1.3.1 功能包	2
1.4 通信机制	3
1.4.1 话题通信机制——Topic	3
1.4.2 服务通信机制——Service	4
1.4.3 动作通信机制——Action	5
1.4.4 参数管理机制——Parameter	6
2 ROS 基础	7
2.1 turtlesim 功能包	7
2.2 创建工作空间和功能包	7
2.2.1 创建工作空间	7
2.2.2 创建功能包	8
2.3 工作空间的覆盖	8
2.4 Topic 中的 Publisher 和 Subscriber	8
2.4.1 Publisher 的创建	8
2.4.2 Subscriber 的创建	9
2.4.3 自定义话题消息	10
2.4.4 CMakeLists 的编写	11
2.5 Service 中的 Client 和 Server	12

2.5.1	创建 Client	12
2.5.2	创建 Server	13
2.5.3	自定义服务数据	14
2.5.4	CMakeLists 的编写	14
2.6	Action 中的 Client 和 Server	15
2.6.1	Client 的创建	15
2.6.2	Server 的创建	15
2.6.3	自定义动作数据	15
2.6.4	CMakeLists 的编写	15
2.7	ROS 中的命名空间	15
2.7.1	有效的命名	15
2.7.2	命名解析	15
2.7.3	命名重映射	16
2.8	多机通信	17
3	ROS 中的常用组件	18
3.1	launch 文件	18
3.1.1	启动节点	18
3.1.2	系统参数设置	19
3.1.3	设置内部变量	19
3.1.4	重映射机制	19
3.1.5	嵌套复用	19
3.2	TF 坐标变换	20
3.2.1	TF 辅助工具	20
3.2.2	TF 中的 Boardcaster 和 Listener	21
3.3	Qt 工具箱	23
3.3.1	日志输出工具 rqt_console	23
3.3.2	计算图可视化工具 rqt_graph	24
3.3.3	数据绘制工具 rqt_plot	24
3.3.4	参数动态配置工具 rqt_reconfigure	24
3.4	rviz 三维可视化平台	24
3.5	Gazebo 仿真环境	24
3.6	rosbag 数据记录与回放	24
3.6.1	记录数据	24
3.6.2	回放数据	25

4 机器人的建模与仿真	26
4.1 URDF 文件	26
4.1.1 <link> 标签	26
4.1.2 <joint> 标签	27
4.1.3 <robot> 标签	28
4.1.4 <gazebo> 标签	28
4.2 创建 URDF 模型	29
4.2.1 创建功能包	29
4.2.2 URDF 模型代码	29
4.3 使用 xacro 优化 URDF 模型	32
4.3.1 xacro 的三个机制	32
4.3.2 引用 xacro 文件	34
4.3.3 显示 xacro 优化后的模型	34
4.4 添加传感器模型	34
4.5 基于 ArbotiX 和 rviz 的仿真器	35
4.5.1 在 ROS-melodic 中安装 ArbotiX	35
4.5.2 配置 ArbotiX 控制器	36
4.5.3 运行仿真	37
4.6 ros_control	37
4.6.1 ros_control 的框架	37
4.6.2 控制器	37
4.6.3 硬件接口	38
4.6.4 传动系统	38
4.6.5 关节约束	38
4.6.6 控制器管理器	38
4.7 Gazebo 仿真	38
4.7.1 配置机器人模型	38
4.7.2 显示机器人模型	42
4.7.3 摄像头仿真	43
4.7.4 Kinect 仿真	44
4.7.5 激光雷达仿真	46
5 机器人 SLAM 与自主导航	48
5.1 准备工作	48
5.1.1 机器人要求	48
5.1.2 传感器信息	48
5.2 gmapping	50
5.2.1 gmapping 功能包介绍	50

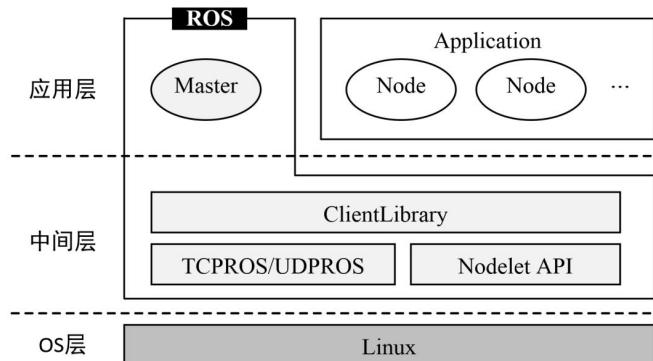
5.2.2 gmapping 节点配置与运行	51
5.2.3 在 Gazebo 中仿真 SLAM(gmapping)	53
5.3 hector-slam	53
5.4 cartographer	53
5.5 rgbdslam	53
5.6 ORB_SLAM	54
5.7 导航功能包	54
5.7.1 导航框架	54
5.7.2 move_base 功能包	55
5.7.3 amcl 功能包	56
5.7.4 代价地图的配置	57
5.7.5 本地规划器配置	60
6 Navigation 详细学习	62
6.1 三维空间中的刚体运动 (基础知识补充)	62
6.1.1 向量	63
6.1.2 欧式变换	63
6.1.3 旋转矩阵	64
6.1.4 变换矩阵与齐次坐标	64
6.1.5 旋转向量	65
6.1.6 欧拉角	66
6.1.7 四元数	66
6.1.8 四元数的运算	68
6.2 move_base 源码学习	68
6.2.1 源码相关文件	68
6.2.2 整体结构图	69
6.2.3 move_base_node.cpp	70
6.2.4 move_base.cpp	70
6.3 navfn 源码学习	83
6.3.1 Dijkstra 算法原理	83
6.3.2 源码相关文件	83
6.3.3 整体结构图	84
6.3.4 navfn_ros.cpp	84
6.3.5 navfn.cpp	89
6.4 global_planner 源码学习	105
6.4.1 A* 算法原理	105
6.4.2 源码相关文件	105
6.4.3 整体结构图	106

6.4.4	参数配置	106
6.4.5	plan_node.cpp	107
6.4.6	planner_core.cpp	107
6.4.7	aster.cpp	108
6.4.8	grid_path.cpp	111
6.4.9	gradient_path.cpp	112
6.4.10	potential_calculator.h	112
6.4.11	quadratic_calculator.cpp	113
6.4.12	expander.h	113
6.4.13	traceback.h	113
6.5	base_local_planner 源码学习	113
6.5.1	源码相关文件	114
6.5.2	整体结构图	115
6.5.3	参数配置	115
6.5.4	trajectory_planner_ros.cpp	116
6.5.5	trajectory_planner.cpp	123
6.5.6	MapGrid 类和 MapCell 类	135
6.5.7	CostmapModel 类	140
6.6	amcl 源码学习	143

Chapter 1

ROS 的基本架构

1.1 整体架构



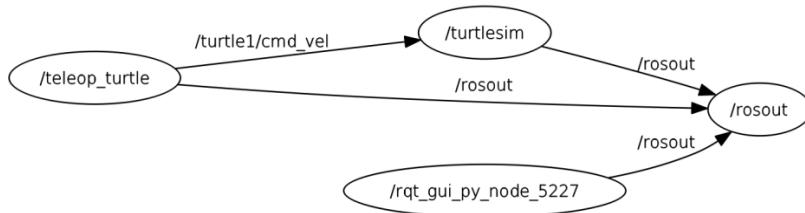
OS 层：是 ROS 依托的底层操作系统，一般是 Ubuntu。

中间层：最重要的就是基于 TCP/UDP 网络，进行封装形成的 TCPROS/UDPROS 通信系统，这其中包括了 Topic 的发布、订阅的通信方式，Service 的客户端、服务器的通信方式等。另外 ROS 还提供了一种进程内通信的方式——Nodelet，可以为多进程通信提供一种更优化的数据传输方式，适合对实时性要求较高的应用。

在通信机制的基础上，ROS 还在中间层提供了大量的机器人开发相关的实用功能，如：数据类型定义、坐标变换、运动控制等。

应用层：ROS 需要运行一个管理者——Matser，负责整个系统的正常运行。其他的一些相关的 ROS 功能包都是以节点（Node）的方式运行，一般来说，简单的开发工作只需要关注节点的标准输入输出接口，而不需要关注模块的内部实现。

1.2 计算图的视角



从计算图的视角来看 ROS 的功能模块，它们都是以节点为单位独立运行的，甚至可以分布于不同的主机中。

1.2.1 节点 Node

节点就是一些执行运算任务的进程，它们之间可以相互通信。

1.2.2 话题 Topic

消息以一种发布/订阅 (publish/subscribe) 的方式传递，发布者和订阅者并不了解彼此的存在，系统中可能有多个节点发布或者订阅同一个话题的消息。

1.2.3 服务 Service

对于双向的同步传输模式，采用基于客户端/服务器 (Client/Server) 的模型，包含请求和应答，类似于 Web 服务器，ROS 中只允许有一个节点提供指定命名的服务。

1.2.4 动作 Action

action 是一种类似于 Service 的问答通信机制，也采用服务器/客户端 (Client/Server) 的工作模式，不同之处在于 action 带有连续反馈，可以不断反馈任务进度，也可以在任务过程中中止运行。

1.2.5 节点管理器 Master

节点管理器帮助 ROS 节点之间相互查找、建立连接，同时还为系统提供参数服务器，管理全局参数。

1.3 文件系统

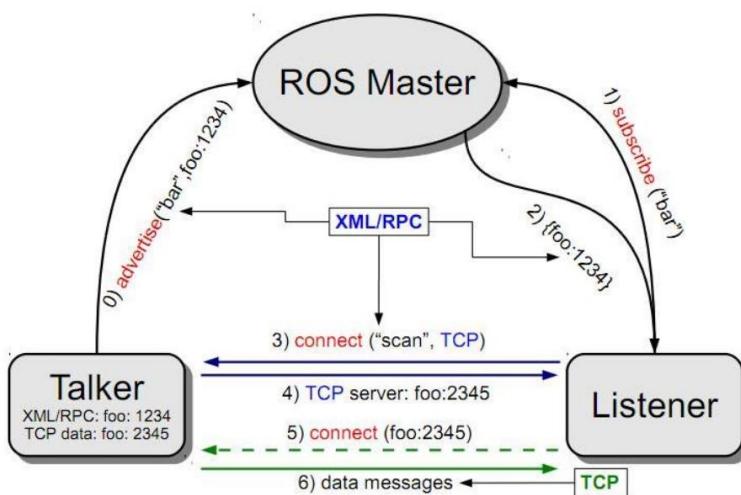
1.3.1 功能包

功能包相关的常用 ROS 命令：

命令	作用
catkin_create_pkg	创建功能包
rospack	获取功能包的信息
catkin_make	编译功能包的信息
rosdep	自动安装功能包依赖的其他包
roscd	功能包目录跳转
roscp	拷贝功能包中的文件
rosed	编辑功能包中的文件
rosrun	运行功能包中的可执行文件
roslaunch	运行启动文件

1.4 通信机制

1.4.1 话题通信机制——Topic



假设 Talker 首先启动，建立通信的详细过程：

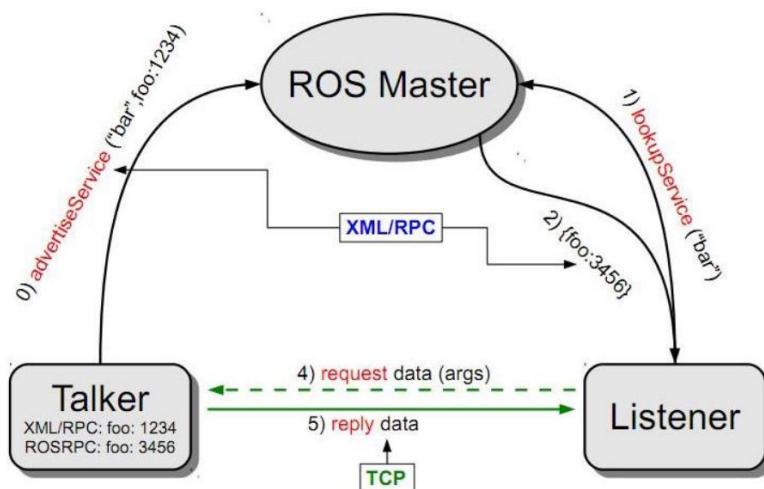
- 1、发布者（Talker）启动，通过 RPC 向 ROS Master 注册发布者的信息，包括：发布者节点信息，话题名，话题缓存大小等；Master 会将这些信息加入注册列表中；
- 2、订阅者（Listener）启动，通过 RPC 向 ROS Master 注册订阅者信息，包括：订阅者节点信息，话题名等；Master 会将这些信息加入注册列表；
- 3、Master 进行节点匹配：Master 会根据订阅者提供的信息，在注册列表中查找匹配的发布者；如果没有发布者（Talker），则等待发布者（Talker）的加入；如果找到匹配的发布者

(Talker)，则会主动把发布者（Talker）（有可能是很多个 Talker）的地址通过 RPC 传送给订阅者（Listener）节点；

- 4、Listener 接收到 Master 的发出的 Talker 的地址信息，尝试通过 RPC 向 Talker 发出连接请求（信息包括：话题名，消息类型以及通讯协议（TCP/UDP））；
- 5、Talker 收到 Listener 发出的连接请求后，通过 RPC 向 Listener 确认连接请求（包含的信息为自身 TCP 地址信息）；
- 6、Listener 接收到 Talker 的确认消息后，使用 TCP 尝试与 Talker 建立网络连接；
- 7、成功连接之后，Talker 开始向 Listener 发布话题消息数据；

需要注意的是：有可能多个 Talker 连接一个 Listener，也有可能是一个 Talker 连接上多个 Listener（多对多）。

1.4.2 服务通信机制——Service



与话题的通信相比，其减少了 Listener 与 Talker 之间的 RPC 通信，建立通信的详细过程：

- 1、发布者（Talker）启动，通过 RPC 向 ROS Master 注册发布者的信息，包括：发布者节点信息，话题名，话题缓存大小等；Master 会将这些信息加入注册列表中；
- 2、订阅者（Listener）启动，通过 RPC 向 ROS Master 注册订阅者信息，包括：订阅者节点信息，话题名等；Master 会将这些信息加入注册列表；
- 3、Master 进行节点匹配：Master 会根据订阅者提供的信息，在注册列表中查找匹配的发布者；如果没有发布者（Talker），则等待发布者（Talker）的加入；如果找到匹配的发布者（Talker），则会主动把发布者（Talker）（有可能是很多个 Talker）的地址通过 RPC 传送给订阅者（Listener）节点；

- 4、Listener 接收到 Talker 的确认消息后，使用 TCP 尝试与 Talker 建立网络连接；
- 5、成功连接之后，Talker 开始向 Listener 发布话题消息数据；

需要注意的是：有可能是一个 Talker 连接上多个 Listener（一对多）。

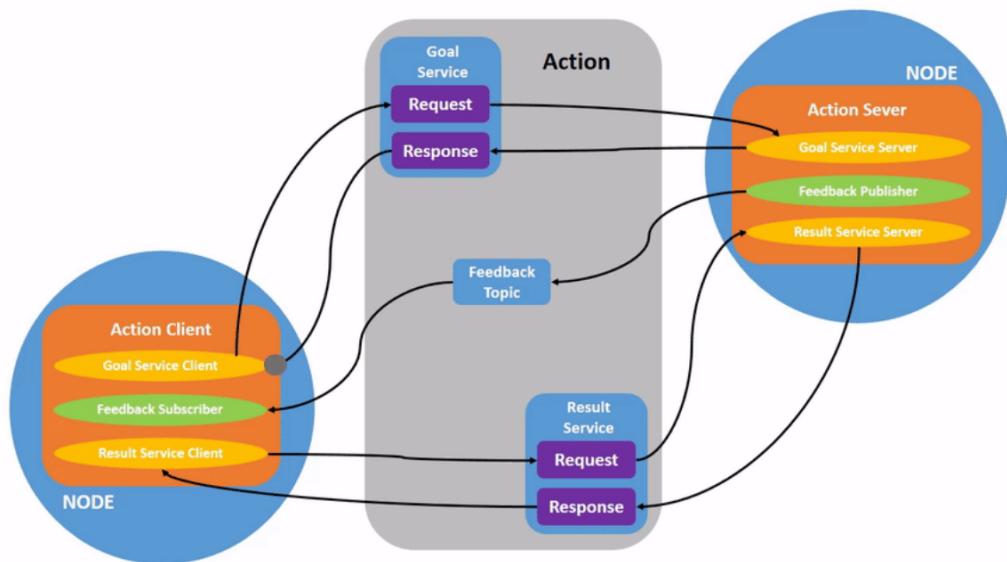
1.4.3 动作通信机制——Action

Action 并不是一个全新的机制，而是由底层的三个话题和服务组成：一个任务目标（Goal，服务），一个执行结果（Result，服务），周期数据反馈（Feedback，话题）。

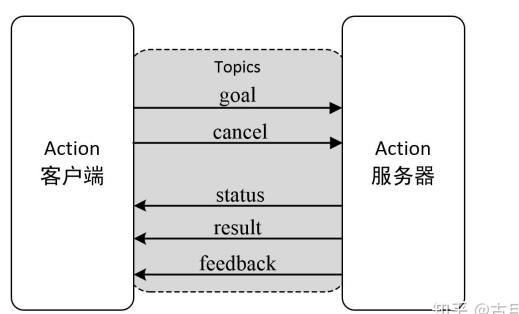
Action 是可抢占式的，由于需要执行一段时间，比如执行过程中你不想跑了，那可以随时发送取消指令，动作终止，如果执行过程中发送一个新的 action 目标，则会直接中断上一个目标开始执行最新的任务目标。

总体上来讲，Action 是一个客户端/服务器的通信模型，客户端发送一个任务目标，服务器端根据收到的目标执行并周期反馈状态，执行完成后反馈一个执行结果。

Action 通信机制的工作原理图如下：



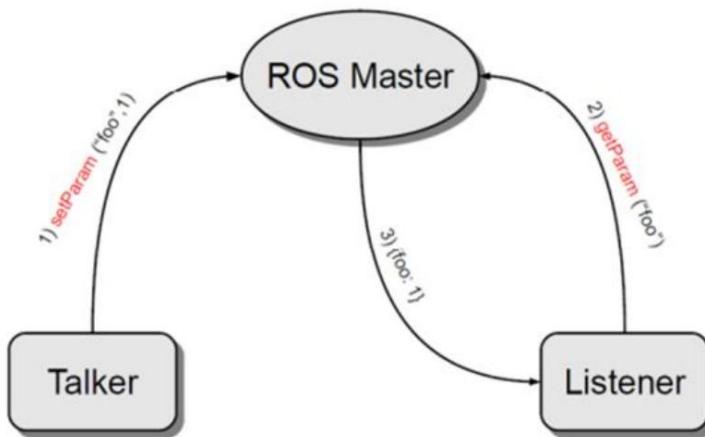
其中 Action 机制下，具体包含的一些信息种类有这五种：



- goal: 发布任务目标
- cancel: 请求取消任务
- status: 通知 Client 当前的状态
- feedback: 周期反馈任务运行的监控数据
- result: 向 Client 发送任务的执行结果, 只发布一次

Client 向 Server 端发布任务目标以及在必要的时候取消任务, Server 会向 Client 发布当前状态、实时反馈和任务执行的最终结果。

1.4.4 参数管理机制——Parameter



参数共享机制类似于程序中的全局变量, Talker 去更新全局变量 (共享的参数), Listener 去获取更新后的全局变量 (共享的参数); 这个通信过程不涉及 TCP/UDP 的通信;

- 1、Talker 更新全局变量; Talker 通过 RPC 更新 ROS Master 中的共享参数 (包含参数名和参数值);
- 2、Listener 通过 RPC 向 ROS Master 发送参数查询请求 (包含要查询的参数名);
- 3、ROS Master 通过 RPC 回复 Listener 的请求 (包括参数值);

需要注意的是: 如果 Listener 向实时知道共享参数的变化, 需要自己不停的去询问 ROS Master;

Chapter 2

ROS 基础

2.1 turtlesim 功能包

接触的第一个 ROS 功能包：turtlesim，其核心是 turtlesim_node 节点。

其中包含的话题和服务如下：

名称		类型	描述
话题订阅	turtleX/cmd_vel	geometry_msgs/ Twist	控制乌龟角速度与线速度的 输入指令
话题发布	turtleX/pose	turtlesim/Pose	乌龟的姿态信息：包括 x 与 y 坐标、角度、线速度和角速度
服务	clear	std_srvs/Empty	清楚仿真器中的背景颜色
	reset	std_srvs/Empty	复位仿真器到初始状态
	kill	turtlesim/Kill	删除一只乌龟
	spawn	turtlesim/Spawn	新生一只乌龟
	turtleX/set_pen	turtlesim/Setpen	设置画笔的颜色和线宽
	turtleX/teleport_absolute	turtlesim/ TeleportAbsolute	移动乌龟到指定的姿态
	turtleX/teleport_realative	turtlesim/ TeleportRealative	移动乌龟到指定的角度和距离

2.2 创建工作空间和功能包

2.2.1 创建工作空间

工作空间初始化：

```
mkdir ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
```

初始化后，可以编译整个工作空间：

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

编译后，在工作空间的根目录下会产生 build 和 devel 两个文件夹，在 devel 文件夹中有 setup.bash 形式的环境变量设置脚本，则可以使用 source 命令运行这些脚本配置环境变量，如：

```
source devel/setup.bash
```

但是 source 命令设置的环境变量只在当前终端中有效，所以为了方便，可以将终端的配置文件（.bashrc）中加入上面的环境变量的配置语句（要注意写全绝对路径）。

2.2.2 创建功能包

创建功能包的命令如下：

```
cd ~/catkin_ws/src
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

创建完成后，工作空间的 src 目录中会生成一个 <package_name> 的功能包，并且已经包含了 package.xml 和 CMakefile.txt 文件。其中 package.xml 文件提供描述功能包属性的信息，CMakefile.txt 文件记录功能包的编译规则。

进而可以回到工作空间的根目录下进行编译，并设置环境变量。

2.3 工作空间的覆盖

所有工作空间的路径会依次在 ROS_PACKAGE_PATH 环境变量中记录，当设置多个工作空间的环境变量后，新设置的路径在 ROS_PACKAGE_PATH 中会自动放在最前端。在运行时，ROS 会优先查找最前端的工作空间中是否存在指定的功能包，如果不存在，就顺序向后查找其他工作空间，知道最后一个工作空间为止。

2.4 Topic 中的 Publisher 和 Subscriber

2.4.1 Publisher 的创建

```
#include <iostream>
#include "ros/ros.h"
```

```

#include "std_msgs/String.h"
int main(int argc, char **argv){
    // ROS节点初始化
    ros::init(argc, argv, "talker");
    // 创建节点句柄
    ros::NodeHandle n;
    // 创建一个Publisher，发布名为chatter的topic，消息类型为std_msgs::String
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    // 设置循环的频率
    ros::Rate loop_rate(10);
    int count = 0;
    // 一旦发生异常，ros::ok()就会返回false，跳出循环
    while (ros::ok()){
        // 初始化std_msgs::String类型的消息
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        // 发布消息
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        // 循环等待回调函数
        // ros::spinOnce()函数用来处理节点订阅话题的所有回调函数
        // 虽然目前的发布节点并没有任何订阅信息，ros::spinOnce()不是必须的
        // 但是为了保证功能无误，建议所有节点都默认加入该函数
        ros::spinOnce();
        // 按照循环频率延时
        loop_rate.sleep();
        ++count;
    }
    return 0;
}

```

2.4.2 Subscriber 的创建

```

#include "ros/ros.h"
#include "std_msgs/String.h"
// 接收到订阅的消息后，会进入消息回调函数

```

```

// 当有消息到达时，会自动以消息指针作为参数
void chatterCallback(const std_msgs::String::ConstPtr& msg){
    // 将接收到的消息打印出来
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv){
    // 初始化ROS节点
    ros::init(argc, argv, "listener");
    // 创建节点句柄
    ros::NodeHandle n;
    // 创建一个Subscriber，订阅名为chatter的topic，注册回调函数chatterCallback
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    // 循环等待回调函数
    ros::spin();
    return 0;
}

```

2.4.3 自定义话题消息

编写 msg 文件

使用 msg 文件定义自己的消息类型，一般放置在功能包根目录下的 msg 文件夹中。msg 文件中既可以定义消息类型的变量，也可以定义常量：

```

string name
uint8 sex
uint8 age

uint8 unknown = 0
uint8 male    = 1
uint8 female  = 2

```

对于稍复杂一些的 ROS 自定义消息，还会包含一个标准格式的头信息 std_msgs/Header:

```

uint32 seq
time stamp
string frame_id

```

其中：seq 是消息的顺序标识，不需要手动设置，Publisher 在发布消息时会自动累加；stamp 是消息中与数据相关联的时间戳，可以用于时间同步；frame_id 是消息中与数据相关联的参考坐标系 id。

编译 msg 文件

- (1) 在 package.xml 中添加功能包依赖

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

- (2) 在 CMakeLists.txt 文件中添加编译选项

在 find_package 中添加消息生成依赖的功能包 message_generation:

```
find_package(catkin REQUIRED COMPONENTS
    geometry_msgs
    roscpp
    rospy
    std_msgs
    message_generation
)
```

设置 catkin 依赖:

```
catkin_package(
    # INCLUDE_DIRS include
    # LIBRARIES learning_communication
    CATKIN_DEPENDS geometry_msgs roscpp rospy std_msgs message_runtime
    # DEPENDS system_lib
)
```

设置需要编译的 msg 文件:

```
add_message_files(FILES Person.msg)
generate_messages(DEPENDENCIES std_msgs)
```

然后对功能包进行编译，自定义的消息类型就生效了。

2.4.4 CMakeLists 的编写

几个常用的编译选项:

- (1) include_directories

用于设置头文件的相对路径。功能包的一些头文件会放在功能包根目录下的 include 文件夹中，所以需要添加该文件夹。

- (2) add_executable

用于设置需要编译的代码和生成的可执行文件。第一个参数为期望生成的可执行文件的名称，后面的参数为参与的源码文件 (cpp)，如果需要多个代码文件，可以在后面依次列出，中间用空格分隔。

(3) target_link_libraries

用于设置链接库。第一个参数为期望生成的可执行文件的名称，后面依次列出需要链接的库，如果没有使用其他库，添加默认链接库 (`catkin_LIBRARIES`) 即可。

(4) add_dependencies

用于设置依赖。在很多应用中，我们需要定义语言无关的消息类型，消息类型会在编译过程中产生相应语言的代码，如果编译的可执行文件依赖这些动态生成的代码，则需要使用 `add_dependencies` 添加 `PROJECT_NAME_generate_messages_cpp` 配置，即该功能包动态产生的消息代码。

对于我们的这个例子，`CMakeLists.txt` 文件如下：

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker ${PROJECT_NAME}_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener ${PROJECT_NAME}_generate_messages_cpp)
```

2.5 Service 中的 Client 和 Server

2.5.1 创建 Client

```
#include <cstdlib>
#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"

int main(int argc, char **argv){
    ros::init(argc, argv, "add_two_ints_client");
    // 从终端命令行获取两个加数，argv[0]是路径，argv[1]和[2]是两个输入参数
    if (argc != 3){
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }
    ros::NodeHandle n;
    // 创建一个client，请求add_two_int service
    // service消息类型是learning_communication::AddTwoInts
```

```

ros::ServiceClient client = n.serviceClient\
    <learning_communication::AddTwoInts>("add_two_ints");
// 创建learning_communication::AddTwoInts类型的service消息
// 该变量包含两个成员：request和response
learning_communication::AddTwoInts srv;
// atoll()函数将字符串转化为整数
srv.request.a = atoll(argv[1]);
srv.request.b = atoll(argv[2]);
// 发布service请求，等待加法运算的应答结果
// 调用过程会发生阻塞，调用成功后返回true
if (client.call(srv)){
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);
}
else{
    ROS_ERROR("Failed to call service add_two_ints");
    return 1;
}
return 0;
}

```

2.5.2 创建 Server

```

#include "ros/ros.h"
#include "learning_communication/AddTwoInts.h"
// service回调函数，输入参数req，输出参数res
bool add(learning_communication::AddTwoInts::Request &req,
          learning_communication::AddTwoInts::Response &res){
    // 将输入参数中的请求数据相加，结果放到应答变量中
    res.sum = req.a + req.b;
    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
    ROS_INFO("sending back response: [%ld]", (long int)res.sum);
    return true;
}
int main(int argc, char **argv){
    ros::init(argc, argv, "add_two_ints_server");
    ros::NodeHandle n;
    // 创建一个名为add_two_ints的server，注册回调函数add()
    ros::ServiceServer service = n.advertiseService("add_two_ints", add);
}

```

```
// 循环等待回调函数
ROS_INFO("Ready to add two ints.");
ros::spin();
return 0;
}
```

2.5.3 自定义服务数据

编写 srv 文件

使用 srv 文件定义自己的消息类型，一般放置在功能包根目录下的 srv 文件夹中。该文件包含 request 和 response 两个数据域，两个数据域之间用”—”（三个减号）分隔，如：

```
int64 a
int64 b
---
int64 sum
```

编译 srv 文件

- (1) 在 package.xml 中添加功能包依赖（与自定义话题消息相同）

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

- (2) 在 CMakeLists.txt 文件中添加编译选项

与自定义话题消息相同也是添加 message_generation 包，

```
find_package(catkin REQUIRED COMPONENTS
    geometry_msgs
    roscpp
    rospy
    std_msgs
    message_generation
)
add_service_files(FILES AddTwoInts.srv)
```

2.5.4 CMakeLists 的编写

与 Topic 类似：

```
include_directories(include ${catkin_INCLUDE_DIRS})  
  
add_executable(server src/server.cpp)  
target_link_libraries(server ${catkin_LIBRARIES})  
add_dependencies(server ${PROJECT_NAME}_gencpp)  
  
add_executable(client src/client.cpp)  
target_link_libraries(client ${catkin_LIBRARIES})  
add_dependencies(client ${PROJECT_NAME}_gencpp)
```

2.6 Action 中的 Client 和 Server

2.6.1 Client 的创建

2.6.2 Server 的创建

2.6.3 自定义动作数据

2.6.4 CMakeLists 的编写

2.7 ROS 中的命名空间

2.7.1 有效的命名

- 1、首字符必须是 ([a-z|A-Z])、波浪线 (~) 或者左斜杠 (/)
- 2、后续字母可以是字母或数字 ([0-9|a-z|A-Z])、下划线 (_) 或者左斜杠 (/)

2.7.2 命名解析

全局名称: /global/name

全局名称的首字符是左斜杠，它之所以称为全局，是因为它的解析度最高，可以在全局范围内直接访问。

但是在系统中，全局名称越少越好，因为过多的全局名称会影响功能包的可移植性。

相对名称: relative/name

相对名称由 ROS 提供默认的命名空间，不需要带有开头的左斜杠，ROS 会对一个相对名称进行解析，进而得到一个全局名称来使用，就类似与我们平时使用的相对路径。相对名称的使用会提高可移植性。

例如：在默认命名空间/relative 内使用相对名称 name，则系统会将其解析为全局名称：/relative/name。

ROS 提供的三种指定默认命名空间的方式：

- 1、通过命令参数设置

调用 ros::init() 的程序会接受一个名为 __ns 的命令行参数，用来设置默认命名空间：

```
--ns:=default-namespace
```

- 2、在 launch 文件中设置

在 launch 文件中可以通过参数 ns 来设置默认命名空间：

```
<node pkg="turtlesim" type="turtlesim_node" name="turtlesim\_node" ns="sim1"/>
```

- 3、使用环境变量设置

在执行 ROS 程序的终端中设置默认命名空间的环境变量：

```
export ROS_NAMESPACE = default-namespace
```

私有名称： private/name

私有名称是一个节点内部私有的资源名称，只会在节点内部使用。私有名称以波浪线 “~” 开始。类似相对名称，也需要 ROS 为其解析，成为一个有意义的全局名称，不同的是，私有名称并不使用当前的默认命名空间，而是使用节点的全局名称作为命名空间。

例如有一个节点的全局名称是/sim1/pubvel，其中的一个私有名称为 ~/max_vel，则其会被解析成全局名称：/sim1/pubvel/max_vel。

ROS 命名解析总结

节点	全局名称	相对名称 (默认)	私有名称
/node1	/bar ->/bar	Bar ->/bar	~bar ->/node1/bar
/wg/node2	/bar ->/bar	Bar ->/wg/bar	~bar ->wg/node2/bar
/wg/node3	/foo/bar ->/foo/bar	foo/bar ->wg/foo/bar	~foo/bar ->wg/node3/foo/bar

2.7.3 命名重映射

所有的 ROS 节点内的资源名称都可以在节点启动的时候进行重映射，这一特性支持我们同事打开多个相同的节点，而不会发生命名冲突。

命名重映射语法：

```
old\_name:=new\_name
```

例如，要将 chatter 重映射为/wg/chatter，在节点启动时候可以输入如下命令：

```
$ rosrun rospy_tutorials talker chatter:=/wg/chatter
```

需要注意：ROS 的命名解析是在命名重映射之前发生的。所以当我们使用“foo:=bar”时，会将节点内所有 foo 命名映射为 bar，而如果我们重映射“/foo:=bar”时，ROS 只会讲全局解析为/foo 的名称重映射为 bar。

命名重映射和命名解析之间的关系：

节点命名空间	重映射参数	匹配名称	解析名称
/	foo:=bar	foo,/foo	/bar
/baz	foo:=bar	foo,/baz/foo	/baz/bar
/	/foo:=bar	foo,/foo	/bar
/baz	/foo:=bar	/foo	/baz/bar
/baz	/foo:=/a/b/c/bar	/foo	/a/b/c/bar

2.8 多机通信

设置 IP 地址

- 1、确保所有计算机处于同一网络中，使用 ifconfig 命令查看本机的局域网 ip 地址。
- 2、分别在每台计算机的/etc/hosts 文件中添加其他计算机的 ip 地址和对应的计算机名称。
- 3、测试是否能够 ping 通其他计算机。

设置 ROS_MASTER_URI

因为系统中只能存在一个 Master，所以从机需要知道 Master 的位置，可以在从机中使用如下命令，将 Master 的地址写入环境变量中：

```
$ echo "export ROS_MASTER_URI = http://<主机名>::11311" >> ~/.bashrc
```

Chapter 3

ROS 中的常用组件

3.1 launch 文件

launch 文件是 ROS 中同时启动多个节点的途径，它还可以自动启动 ROS Master 节点管理器，并且实现每个节点的各种配置。

launch 文件采用 XML 的形式进行描述，XML 文件必须包含一个根元素，launch 文件的根元素采用 <launch> 标签定义，文件中的其他内容都必须包含在这个标签中。

3.1.1 启动节点

采用 <node> 标签启动 ROS 节点，语法如下：

```
<node pkg = "package-name" type = "executable-name" name = "node-name"/>
```

- pkg 定义节点所在的功能包名称
- type 定义节点的可执行文件名称
- name 定义节点运行时的名称，讲覆盖节点中 init() 赋予节点的名称

另外还有如下可选的属性参数：

- output = "screen": 讲节点的标准输出打印到终端（默认输出为日志文档）
- respawn = "true": 复位属性，该节点停止时，会自动重启，默認為 flase
- required = "true": 必要节点，当该节点终止时，launch 文件中的其他节点也被终止
- ns = "namespace": 命名空间，为节点内的相对名称添加命名空间前缀
- args = "arguments": 节点需要输入的参数

3.1.2 系统参数设置

使用 `<param>` 标签来设置 ROS 系统运行中的参数（即 parameter），存储在参数服务器中。launch 文件执行后，parameter 就加载到 ROS 的参数服务器上。

每个活跃的节点都可以通过 `ros::param::get()` 接口来获取 parameter 的值，用户也可以在终端中通过 `rosparam` 命令获得 parameter 的值。

`<param>` 标签的语法如下：

```
<param name = "output_frame" value = "odom"/>
```

另外，ROS 也提供了一种从文件中批量加载参数的方法，使用标签 `<rosparam>`，其语法如下：

```
<rosparam file = "$(find 2dnav_pr2)/config/costmap_common_params.yaml" command = "load" ns = "local_costmap"/>
```

`<rosparam>` 标签可以帮我们将一个 YAML 格式的文件中的全部参数加载到 ROS 中，需要将 `command` 属性设置为“load”。

3.1.3 设置内部变量

使用 `<arg>` 标签可以设置 launch 文件内部的局部变量（argument），仅限于 launch 文件内部使用，语法如下：

```
<arg name = "arg-name" default = "arg-value"/>
```

在 launch 文件中使用 argument 时，可以使用如下语法进行调用：

```
<node pkg = "package" type = "type" name = "name" args = "$(arg arg-name)"/>
```

3.1.4 重映射机制

使用 `<remap>` 标签可以实现重映射的功能，可以给功能包的接口名称重映射一下，取一个别名，可以用来实现不同功能包之间的接口匹配，语法如下：

```
remap from = "turtlebot/cmd_vel" to = "/cmd_vel"/>
```

3.1.5 嵌套复用

使用 `<include>` 标签可以实现在一个 launch 文件中包含其他的 launch 文件。即可直接复用其他已有的 launch 文件中的内容，语法如下：

```
<include file = "$(dirname)/other.launch"/>
```

3.2 TF 坐标变换

TF 是一个让用户随时间跟踪多个坐标系的功能包，它使用树形数据结构，根据时间缓冲并维护多个坐标系之间的坐标变换关系。

3.2.1 TF 辅助工具

1.tf_monitor

功能是打印 TF 树中所有坐标系的发布状态，使用方法如下：

```
$ tf_monitor  
$ tf_monitor <source_frame> <target_frame>
```

2.tf_echo

功能是查看指定坐标系之间的变换关系，使用方法如下：

```
$ tf_echo <source_frame> <target_frame>
```

3.static_transform_publisher?

功能是发布两个坐标系之间的静态坐标变换，这两个坐标系不发生相对的位置变化，使用方法如下：

```
$ static_transform_publisher x y z yaw pitch roll frame_id child_frame_id  
period_in_ms  
$ static_transform_publisher x y z qx qy qz qw frame_id child_frame_id  
period_in_ms
```

以上两种命令格式，需要设置坐标的偏移参数和旋转参数：偏移参数使用相对于 xyz 轴的坐标位移；旋转参数分别采用了欧拉角和四元数的表达方式，并设置发送频率以 ms 为单位。

另外，该命令还可以在 launch 文件中使用，语法如下：

```
<launch>  
<node pkg = "tf" type = "static_transform_publisher" name = "link1_broadcaster"  
args = "1 0 0 0 0 0 1 link1_parent link1 100"/>  
<\launch>
```

4.view_frame

这是一个可视化的调试工具，可以生成 PDF 文件，显示整棵 TF 树的信息，使用方法如下：

```
$ rosrun tf view_frames
```

3.2.2 TF 中的 Boardcaster 和 Listener

以基于 TF 的乌龟自动跟踪例程为例。

创建 Broadcaster

创建一个发布乌龟坐标系与世界坐标系之间的 TF 变换的节点。

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>
std::string turtle_name;
//回调函数
void poseCallback(const turtlesim::PoseConstPtr& msg){
    // tf广播器
    static tf::TransformBroadcaster br;
    // 根据乌龟当前的位姿，设置相对于世界坐标系的坐标变换
    // setOrigin设置平移变换 setRotation设置旋转变换
    tf::Transform transform;
    transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
    tf::Quaternion q;
    q.setRPY(0, 0, msg->theta);
    transform.setRotation(q);
    // 发布坐标变换 TF消息的数据类型为tf::StampedTransform
    // 包含坐标变换、时间戳，并指定坐标变换的源坐标系(parent)和目标坐标系(child)
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world",
                                         turtle_name));
}
int main(int argc, char** argv){
    // 初始化节点
    ros::init(argc, argv, "my_tf_broadcaster");
    if (argc != 2){
        ROS_ERROR("need turtle name as argument");
        return -1;
    };
    turtle_name = argv[1];
    ros::NodeHandle node;
    // 订阅乌龟的pose信息 订阅到之后，就会进入回调函数进行TF广播
    ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10, &poseCallback);
```

```
    ros::spin();
    return 0;
};
```

创建 Listener

监听 TF 消息，并且从中获取 turtle2 相对于 turtle1 坐标系的变换，从而控制 turtle2 移动。

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/Twist.h>
#include <turtlesim/Spawn.h>
int main(int argc, char** argv){
    ros::init(argc, argv, "my_tf_listener");
    ros::NodeHandle node;
    // 通过Service，产生第二只乌龟turtle2
    ros::service::waitForService("spawn");
    ros::ServiceClient add_turtle =
        node.serviceClient<turtlesim::Spawn>("spawn");
    turtlesim::Spawn srv;
    add_turtle.call(srv);
    // 定义turtle2的速度控制发布器
    ros::Publisher turtle_vel =
        node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);
    // tf监听器
    tf::TransformListener listener;
    ros::Rate rate(10.0);
    while (node.ok()){
        // Broadcaster发布的就是这种类型的消息
        tf::StampedTransform transform;
        try{// 查找turtle2与turtle1的坐标变换
            // 其中/turtle2为当前坐标系，turtle1为目标坐标系
            listener.waitForTransform("/turtle2", "/turtle1", ros::Time(0),
            ros::Duration(3.0));
            listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0),
            transform);
        }
        catch (tf::TransformException &ex) {
            ROS_ERROR("%s", ex.what());
        }
```

```

        ros::Duration(1.0).sleep();
        continue;
    }

    // 根据turtle1和turtle2之间的坐标变换，计算turtle2需要的线速度和角速度
    // 并发布速度控制指令，使turtle2向turtle1移动
    geometry_msgs::Twist vel_msg;
    vel_msg.angular.z = 4.0 * atan2(transform.getOrigin().y(),
                                    transform.getOrigin().x());
    vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                  pow(transform.getOrigin().y(), 2));
    turtle_vel.publish(vel_msg);
    rate.sleep();
}

return 0;
};

```

其中两个重要函数：

- `waitForTransform()`

给定源坐标系和目标坐标系，等待两个坐标系之间指定时间的变换关系，该函数会阻塞程序运行，所以要设置超时时间（timeout）

- `lookupTransform()`

给定源坐标系和目标坐标系，得到两个坐标系之间指定时间的坐标变换，`ros::Time(0)` 表示获取最新一次的坐标变换。

3.3 Qt 工具箱

这是一个基于 Qt 架构的后台图形工具套件——`rqt_common_plugins`。

安装命令：

```
$ sudo apt-get install ros-kinetic-rqt
$ sudo apt-get install ros-kinetic-rqt-common-plugins
```

3.3.1 日志输出工具 `rqt_console`

`rqt_console` 用来图像化显示和过滤 ROS 系统运行状态中的所有日志消息，包括 info、warn、error 等，使用如下命令启动：

```
$ rqt_console
```

3.3.2 计算图可视化工具 rqt_graph

rqt_graph 可以图形化显示当前 ROS 系统中的计算图，使用如下命令启动：

```
$ rqt_graph
```

3.3.3 数据绘制工具 rqt_plot

rqt_plot 是一个二维数值曲线绘制工具，可以将需要显示的数据在 xy 坐标系中使用曲线绘制出来，使用如下命令启动：

```
$ rqt_plot
```

3.3.4 参数动态配置工具 rqt_reconfigure

rqt_reconfigure 可以在不重启系统的情况下，动态配置 ROS 系统中的参数，但是该功能需要在代码中设置参数的相关属性。从而支持动态配置，使用如下命令启动：

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

3.4 rviz 三维可视化平台

在 rviz 中，可以使用 XML 对机器人、周围物体等任何实物进行尺寸、质量、位置、材质、关节等属性的描述，并且在界面中呈现出来。

3.5 Gazebo 仿真环境

虽然 Gazebo 中的机器人模型与 rviz 使用的模型相同，但是需要在模型中加入机器人和周围环境的物理属性，例如质量、摩擦系数、弹性系数等。机器人的传感器信息也可以通过插件的形式加入仿真环境，以可视化的方式进行显示。

3.6 rosbag 数据记录与回放

rosbag 功能包提供了数据记录与回放的功能。

3.6.1 记录数据

开始数据记录的命令：

```
rosbag record -a
```

其中-a(all) 参数表示记录所有发布的消息。数据文件会以.bag 格式保存在当前目录下。

3.6.2 回放数据

查看数据记录文件的命令：

```
$ rosbag info <your bagfile>
```

从该命令的输出信息可以看到数据记录包中包含的所有话题、消息类型、消息数量等信息。

回放所记录的话题数据的命令：

```
$ rosbag play <your bagfile>
```

Chapter 4

机器人的建模与仿真

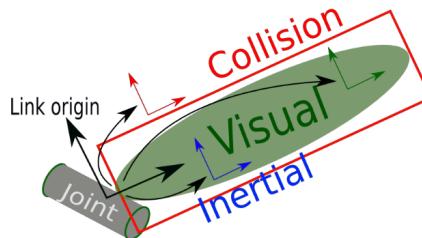
4.1 URDF 文件

URDF (Unified Robot Description Format, 统一机器人描述格式) 是 ROS 中一个非常重要的机器人模型描述格式, ROS 同时也提供了 URDF 文件的 C++ 解析器, 可以解析 URDF 文件中使用 XML 格式描述的机器人模型。

下面说明一下 URDF 文件中常用的几个 XML 标签:

4.1.1 <link> 标签

<link> 标签用于描述机器人某个刚体部分的外形和物理属性, 包括尺寸 (size)、颜色 (color)、形状 (shape)、惯性矩阵 (inertial matrix)、碰撞参数 (collision properties) 等。



从图中可以看出, 检测碰撞的 link 区域大于外观可视的区域, 这就意味着只要有其他物体与 collision 区域相交, 就认为 link 发生碰撞。

<link> 标签的一般结构如下:

```
<link name = "<name of the link>">
  <inertial> .....
  <visual> .....
```

```
<collision> ..... </collision>
</link>
```

其中：

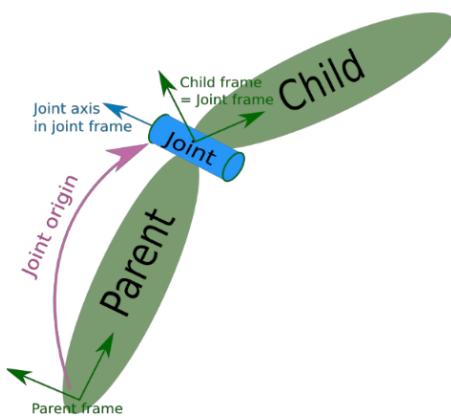
- <visual>：用于描述机器人 link 部分的外观参数
- <inertial>：用于描述 link 的惯性参数
- <collision>：用于描述 link 的碰撞部分

4.1.2 <joint> 标签

<joint> 标签用于描述机器人关节的运动学和动力学属性，包括关节运动的位置和速度限制。根据机器人的关节运动形式，可以将其分为六种类型：

关节类型	描述
continuous	旋转关节，可以围绕单轴无限旋转
revolute	旋转关节，有旋转的角度限制
prismatic	滑动关节，沿某一轴线移动的关节，带有位置极限
planar	平面关节，允许在平面正交方向上平移或者旋转
floating	浮动关节，允许进行平移、旋转运动
fixed	固定关节，不允许运动的特殊关节

机器人关节的主要作用是连接两个刚体 link，这两个 link 分别称为 parent link 和 child link，如下图所示：



<joint> 标签的一般结构如下：

```
<joint name = "<name of the joint>">
```

```

<parent link = "parent_link"/>
<child link = "child_link"/>
<calibration .... />
<dynamics damping .... />
<limit effort .... />
....
</joint>

```

其中必须指定 joint 的 parent link 和 child link，还可以设置关节的其他属性：

- <calibration>：关节的参考位置，用来校准关节的绝对位置。
- <dynamics>：用于描述关节的物理属性，例如阻尼、静摩擦力，经常在动力学仿真中出现。
- <limit>：用于描述运动的一些极限值，包括关节运动的上下限位置、速度限制、力矩限制等。
- <mimic>：用于描述该关节与已有关节的关系。
- <safety_controller>：用于描述安全控制器参数。

4.1.3 <robot> 标签

<robot> 是完整机器人模型的最顶层标签，<link> 和 <joint> 标签都必须包含在 <robot> 标签内。robot 标签内可以设置机器人的名称，其基本语法如下：

```

<robot name = "name of the robot">
  <link> ..... </link>
  <link> ..... </link>
  <joint> ..... </joint>
  <joint> ..... </joint>
</robot>

```

4.1.4 <gazebo> 标签

<gazebo> 标签用于描述机器人模型在 Gazebo 中仿真所需要的参数，包括机器人材料的属性、Gazebo 插件。该标签不是机器人模型的必需部分，只有在 Gazebo 中仿真时才需要加入，其基本语法如下：

```

<gazebo reference = "link_1">
  <material>Gazebo/Black</material>
</gazebo>

```

4.2 创建 URDF 模型

以 MRobot 机器人为例。

4.2.1 创建功能包

使用如下命令创建一个 urdf 模型的功能包：

```
$ catkin_create_pkg mrobot_description urdf xacro
```

创建好的功能包中包含如下四个文件夹：

- urdf：用于存放机器人模型的 URDF 文件或 xacro 文件
- meshes：用于放置 URDF 中引用的模型渲染文件
- launch：用于保存相关启动文件
- config：用于保存 rviz 的配置文件

4.2.2 URDF 模型代码

part 1

```
<?xml version="1.0" ?>
<robot name="mrobot_chassis">
```

首先在文件开头，需要声明该文件使用 XML 描述，然后使用 `<robot>` 根标签定义一个机器人模型，并定义机器人的名称。

part 2

```
<link name="base_link">
  <visual>
    <origin xyz=" 0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder length="0.005" radius="0.13"/>
    </geometry>
    <material name="yellow">
      <color rgba="1 0.4 0 1"/>
    </material>
  </visual>
</link>
```

这一段代码描述机器人的底盘 link，`<visual>` 标签定义底盘的外观属性；

在 `<geometry>` 标签下定义几何外观，我们将底盘抽象成一个圆柱，使用 `<cylinder>` 标签定义这个圆柱的半径和高；

然后声明这个底盘圆柱在三维坐标位置和旋转姿态，使用 `<origin>` 标签设置底盘中心位置，底盘中心位于界面的中心点，所以将坐标设置为“0 0 0”，旋转设置也设置为“0 0 0”即可（圆柱体默认是垂直地面放置的）；

另外，使用 `<material>` 标签设置底盘的颜色——“黄色”，其中 `<color>` 标签定义颜色的 RGBA 值（这里采用百分数描述，A 为透明度参数）。

part 3

```
<joint name="base_left_motor_joint" type="fixed">
  <origin xyz="-0.055 0.075 0" rpy="0 0 0" />
  <parent link="base_link"/>
  <child link="left_motor" />
</joint>
```

这一段代码定义一个关节 joint，用来连接机器人底盘和左边驱动电机，joint 类型为 fixed 类型，这种类型的 joint 是固定的（见 subsection 4.1.2）。

`<origin>` 标签设置了 joint 的起点，将起点设置在需要安装电机的底盘位置。

part 4

```
<link name="left_motor">
  <visual>
    <origin xyz="0 0 0" rpy="1.5707 0 0" />
    <geometry>
      <cylinder radius="0.02" length = "0.08"/>
    </geometry>
    <material name="gray">
      <color rgba="0.75 0.75 0.75 1"/>
    </material>
  </visual>
</link>
```

这一段代码描述了左侧电机的模型，外形也是圆柱体，采用 `<cylinder>` 标签。

关于 `<origin>` 标签的设置：由于我们上面定义了一个 joint 用来将电机连接到底盘上，电机的位置是相对于 joint 计算的。在 joint 的位置设置中，已经将其放置到了安装电机的位置，所以电机模型的位置设置到“0 0 0”坐标就可以了。

另外由于圆柱体默认垂直地面，因此我们需要将电机模型绕 x 轴旋转 90° 放置。

part 5

```
<joint name="left_wheel_joint" type="continuous">
    <origin xyz="0 0.0485 0" rpy="0 0 0"/>
    <parent link="left_motor"/>
    <child link="left_wheel_link"/>
    <axis xyz="0 1 0"/>
</joint>
```

这一段代码定义一个关节 joint，用来连接电机和轮子，joint 类型为 continuous 类型，这种类型的 joint 可以绕一个轴旋转（见subsection 4.1.2）。

<origin> 标签将轮子的起点设置到电机的一端，<axis> 标签定义该 joint 的旋转轴是 y 轴。

添加物理和碰撞属性

前面的代码仅创建了模型的可视化属性，还需要添加物理属性和碰撞属性，这里以机器人底盘 base_link 为例：

```
<link name="base_link">
    <intertial>
        <mass value="2"/>
        <origin xyz="0 0 0.0"/>
        <inertia ixx="0.01" ixy="0.0" ixz="0.0"
                  iyy="0.01" iyz="0.0"
                  izz="0.5"/>
    </intertial>

    <visual>
        <origin xyz=" 0 0 0" rpy="0 0 0" />
        <geometry>
            <cylinder length="${base_link_length}" radius="${base_link_radius}"/>
        </geometry>
        <material name="yellow">
        </material>
    </visual>

    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <cylinder length="${base_link_length}" radius="${base_link_radius}"/>
```

```

</geometry>
</collision>
</link>
```

其中 `<intertial>` 标签设置惯性参数，主要包括质量和惯性矩阵，如果是规则物体，可以通过尺寸、质量等公式计算得到惯性矩阵（这里有待学习补充）。

4.3 使用 xacro 优化 URDF 模型

URDF 文件不支持代码复用的特性，因此针对 URDF 模型提出了一种精简化、可复用、模块化的描述形式——xacro。

xacro 有两点优点：精简的模型代码、提供可编程接口。模型的后缀名由.urdf 变为.xacro，并且需要在模型的 `<robot>` 标签中加入 xacro 的声明，代码如下：

```

<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">
```

4.3.1 xacro 的三个机制

使用常量定义

定义常量的语法如下：

```
<xacro:property name="M_PI" value="3.14159"/>
```

使用常量的语法如下：

```
<origin xyz="0 0 0" rpy="${M_PI} 0 0"/>
```

调用数学公式

在“\$”语句中，不仅可以调用常量，还可以使用一些常用的数学运算，包括加减乘除、负号、括号等（所有运算都会被转换成浮点数进行），语法如下：

```
<origin xyz="0 ${{(motor_length+wheel_length)/2}} 0" rpy="0 0 0"/>
```

使用宏定义

xacro 文件可以使用宏定义来声明重复使用的代码模块，而且可以包含输入参数，以 MRobot 机器人的八根支撑柱为例，宏定义的语法示例如下：

```

<xacro:macro name="mrobot_standoff_2in" params="parent number x_loc y_loc z_loc">
  <joint name="standoff_2in_${number}_joint" type="fixed">
```

```

<origin xyz="{{$x_loc} {{$y_loc} {{$z_loc}}" rpy="0 0 0" />
<parent link="{{$parent}}"/>
<child link="standoff_2in_${number}_link" />
</joint>

<link name="standoff_2in_${number}_link">
  <inertial>
    <mass value="0.001" />
    <origin xyz="0 0 0" />
    <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
              iyy="0.0001" iyz="0.0"
              izz="0.0001" />
  </inertial>

  <visual>
    <origin xyz=" 0 0 0 " rpy="0 0 0" />
    <geometry>
      <box size="0.01 0.01 0.07" />
    </geometry>
    <material name="black">
      <color rgba="0.16 0.17 0.15 0.9"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    <geometry>
      <box size="0.01 0.01 0.07" />
    </geometry>
  </collision>
</link>
</xacro:macro>

```

以上的宏定义包含五个输入参数：joint 的 parent link，支撑住的序号，支撑柱在 xyz 三个方向上的偏移。这个宏定义在定义一个支撑柱的时候，分别对其 joint 和 link 两个标签进行了定义。

当需要使用该宏模块的时候，按照如下语法进行调用：

```
<mrobot_standoff_2in parent="base_link" number="1" x_loc="-${standoff_x}/2 + 0.03}"
y_loc="-${standoff_y} - 0.03}" z_loc="${plate_height}/2" />
```

4.3.2 引用 xacro 文件

引用示例如下：

```
<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:include filename="$(find mrobot_description)/urdf/mrobot_body.urdf
.xacro" />
    <!-- MRobot 机器人平台 -->
    <mrobot_body/>
</robot>
```

可以看到，在 robot 标签之间，首先使用了 xacro:include 标签，包含了另一个 xacro 模型文件，然后我们就可以在下面使用被包含文件中的模块了。接下来调用被包含文件中的机器人模型宏定义（机器人模型文件全部是在被包含文件中用一个宏来描述的）。

这样将整个机器人模型作为一个宏有什么好处呢？把机器人整体看做一个模块，方便与其他模型进行集成，比如在后续安装传感器等其他模块时。

4.3.3 显示 xacro 优化后的模型

将 xacro 文件转化成 URDF 文件

使用如下命令可以将 xacro 文件转换成 URDF 文件：

```
$ rosrun xacro xacro.py mrobot.urdf.xacro > mrobot.urdf
```

直接调用 xacro 文件解析器

也可以省略手动转换的过程，直接在启动文件中调用 xacro 解析器，自动将 xacro 转换成 URDF 文件，在 launch 文件中使用如下语句进行配置：

```
<arg name="model" default="$(find xacro)/xacro --inorder '$(find mrobot_description)/
<arg name="gui" default="true" />
```

进而可以直接使用这个修改后的启动文件，看到 xacro 格式的机器人模型。

4.4 添加传感器模型

首先我们需要自己创建一个传感器模型（xacro 文件），或者去网上下载一个传感器的模型，这里以一个摄像头为例，其模型文件为 camera.xacro。

进而我们可以创建一个顶层 xacro 文件，将机器人主体与摄像头连接起来：

```
<?xml version="1.0"?>
<robot name="mrobot" xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:include filename="$(find mrobot_description)/urdf/mrobot_body.urdf
.xacro" />
    <xacro:include filename="$(find mrobot_description)/urdf/camera.xacro" />

    <xacro:property name="camera_offset_x" value="0.1" />
    <xacro:property name="camera_offset_y" value="0" />
    <xacro:property name="camera_offset_z" value="0.02" />

    <!-- MRobot机器人平台-->
    <mrobot_body/>

    <!-- Camera -->
    <joint name="camera_joint" type="fixed">
        <origin xyz="${camera_offset_x} ${camera_offset_y} ${camera_offset_z}" rpy="0
0 0">
            <parent link="plate_2_link"/>
            <child link="camera_link"/>
        </joint>

        <xacro:usb_camera prefix="camera"/>
    </robot>
```

在这个顶层文件中，包含了描述摄像头的模型文件以及描述机器人的模型文件，然后使用了一个 fixed 类型的 joint 把摄像头固定到机器人的指定位置。

4.5 基于 ArbotiX 和 rviz 的仿真器

ArbotiX 提供一个差速控制器，通过接收速度控制指令更新机器人的 joint 状态，从而实现机器人在 rviz 中的运动。

4.5.1 在 ROS-melodic 中安装 ArbotiX

Arbotix 本质上就是一个功能包，我们需要像其他我们自己的功能包一样，将其放置在工作空间下的 src 目录下，直接从 git 上下载其源码：

```
$ git clone -b indigo-devel https://github.com/vanadiumlabs/arbotix_ros.git
```

然后重新编译工程即可（注意如果没有将设置环境变量的指令放到.bashrc 中，在这里要记得使用 source 命令设置环境变量）。

4.5.2 配置 ArbotiX 控制器

我们只需要适当修改原本的 launch 文件，然后再创建一个控制器相关的配置文件就可以了。

修改 launch 文件

只是在显示机器人模型的 launch 文件的基础上加上如下内容：

```
<node name="arbotix" pkg="arbotix_python" type="arbotix_driver" output="screen">
    <rosparam file="$(find mrobot_description)/config/fake_mrobot_arbotix.yaml"
command="load" />
    <param name="sim" value="true"/>
</node>
```

从以上代码可以看出，实际上就是添加了一个控制器节点，这里在仿真环境中使用，需要配置“sim”参数为 true。另外，从这里可以看到，启动时还需要加载一个叫“fake_mrobot_arbotix.yaml”的配置文件。

创建配置文件

配置文件的目录为：功能包目录/config/下，文件内容如下：

```
controllers: {
    base_controller: {
        type: diff_controller,
        base_frame_id: base_footprint,
        base_width: 0.26,
        ticks_meter: 4100,
        Kp: 12,
        Kd: 12,
        Ki: 0,
        Ko: 50,
        accel_limit: 1.0
    }
}
```

控制器的名称为“base_controller”，类型为“diff_controller”（差速控制器），另外还给出了参考坐标系、底盘尺寸、PID 参数等。

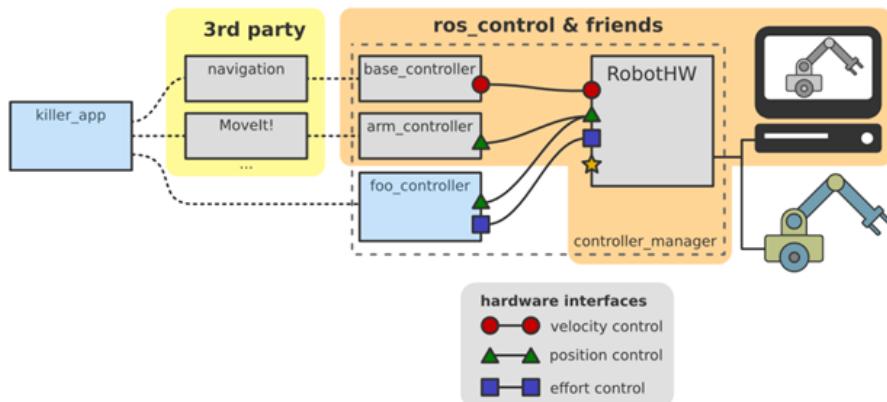
4.5.3 运行仿真

需要注意的是，我们要设置参考坐标系（fixed frame）为“odom”，才可以看到小车的移动。

4.6 ros_control

ros_control 是一套机器人控制中间件，包含一系列控制器接口、传动装置接口、硬件接口、控制器工具箱等。

4.6.1 ros_control 的框架



上图是 ros_control 的总体框架，可以看到正对不同类型的控制器（底盘、机械臂等），ros_control 可以提供多种类型的控制器，但是这些控制器的接口各不相同，为了提高代码的复用率，ros_control 还提供一个硬件的抽象层。硬件抽象层负责机器人硬件资源的管理，而 controller 从抽象层请求资源即可，并不直接接触硬件。

4.6.2 控制器

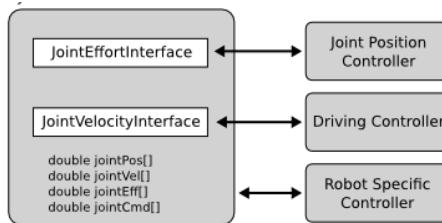
ROS 的 ros_controllers 功能包提供了一些常用的控制器：

- effort_controllers
 - joint_effort_controller
 - joint_position_controller
 - joint_velocity_controller
- joint_state_controller
 - joint_state_controller

- position_controllers
 - joint_position_controller
- velocity_controllers
 - joint_velocity_controller

另外，也可以根据自己的需求创建需要的控制器，并通过控制器管理器进行管理（具体方法有需要再补充）。

4.6.3 硬件接口



硬件接口是控制器与 RobotHW（硬件抽象层）沟通的接口，基本与控制器种类相对应。另外也可以根据自己的需求创建需要的接口（具体方法有需要再补充）。

4.6.4 传动系统

4.6.5 关节约束

4.6.6 控制器管理器

4.7 Gazebo 仿真

4.7.1 配置机器人模型

首先我们需要确定每个 link 的 `<inertia>` 元素已经进行了合理的设置，然后还要为每个必要的 `<link>`、`<joint>`、`<robot>` 设置 `<gazebo>` 标签，进而我们需要为模型添加传动装置以及控制器插件。

为 link 添加惯性参数和碰撞属性

这个在前面 URDF 模型文件中已经提到过了，但是在 rviz 中这一项并不是必须的，其中的模型可以只有显示部分，并没有物理属性，但是 gazebo 中进行的是物理仿真，所以相关的惯性参数以及碰撞属性等物理参数就是必须的了。

另外，这里由于我们一般都使用 xacro 文件作为模型文件，相比 URDF 文件多了宏定义的机制；而且，我们知道，对于规则均匀刚体，其惯性参数矩阵是有一个固定的计算公式的。因此我们可以不再像之前那样每一个 link 的惯性参数都手动输入了，可以按照下面的方式采用宏定义进行自动计算，例如球体，我们可以定义这样一个宏用于计算其惯性矩阵：

```
<xacro:macro name="sphere_inertial_matrix" params="m r">
  <inertial>
    <mass value="${m}" />
    <inertia ixx="${2*m*r*r/5}" ixy="0" ixz="0"
              iyy="${2*m*r*r/5}" iyz="0"
              izz="${2*m*r*r/5}" />
  </inertial>
</xacro:macro>
```

类似地，长方体：

```
<xacro:macro name="box_inertial_matrix" params="m w h d">
  <inertial>
    <mass value="${m}" />
    <inertia ixx="${m*(h*h+d*d)/12}" ixy = "0" ixz = "0"
              iyy="${m*(w*w+d*d)/12}" iyz = "0"
              izz="${m*(w*w+h*h)/12}" />
  </inertial>
</xacro:macro>
```

为 link 添加 `<gazebo>` 标签

需要为每一个 link 添加 `<gazebo>` 标签，包含的属性仅有 material。注意：这里的 material 属性和 `<visual>` 中的 material 属性作用相同，但是 Gazebo 无法通过 `<visual>` 中的 material 属性设置外观颜色，因此需要再添加 `<gazebo>` 标签进行设置，另外，Gazebo 中提供了一些可以直接使用的颜色供我们使用。

设置 `<gazebo>` 标签语法如下：

```
<gazebo reference="wheel_${lr}_link">
  <material>Gazebo/Black</material>
</gazebo>
```

为 joint 添加传动装置

需要在模型中加入 `<transmission>` 元素，将传动装置与 joint 进行绑定。语法如下：

```

<transmission name="wheel_${lr}_joint_trans">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="base_to_wheel_${lr}_joint" />
    <actuator name="wheel_${lr}_joint_motor">
        <hardwareInterface>VelocityJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>

```

其中 `<type>` 标签声明了所使用的传动装置类型；`<joint name>` 标签定义了将要绑定驱动器的 joint；`<actuator name>` 标签定义了传动装置的名称；并在其中使用 `<hardwareInterface>` 标签定义硬件接口类型，这里是速度控制接口；使用 `<mechanicalReduction>` 标签设置了传动比为 1。

添加 Gazebo 控制器插件

Gazebo 插件可以根据插件的运动范围应用到 URDF 模型的 `<robot>`、`<link>`、`<joint>` 上，需要使用 `<gazebo>` 标签作为封装，这里不同于上文中提到的“为 link 添加 `<gazebo>` 标签”，这里需要在 `<gazebo>` 标签下使用 `<plugin>` 标签来添加插件。

(1) 为 `<link>`、`<joint>` 标签添加插件

设置 `reference` 为对应 `<link>` 或 `<joint>` 的名字，其中 `<plugin>` 标签下的插件名字 `name` 可以自拟，`filename` 是 gazebo 提供的现成文件，可以查看 ROS 安装路径 (`/opt/ros/melodic/lib`) 下，所有插件都是以.so 命名的。

```

<gazebo reference="your_link_name">
    <plugin name="unique_name" filename="plugin_name.so"
        ...plugin parameters...
    </plugin>
</gazebo>

```

(2) 为 `<robot>` 标签添加插件：

不设置 `reference` 属性即可。

```

<gazebo>
    <plugin name="unique_name" filename="plugin_name.so"
        ...plugin parameters...
    </plugin>
</gazebo>

```

我们将一个差速控制的插件 (`libgazebo_ros_diff_drive.so`) 应用到我们的示例机器人模型上，语法如下：

```

<gazebo>
    <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
        <rosDebugLevel>Debug</rosDebugLevel>
        <publishWheelTF>true</publishWheelTF>
        <robotNamespace>/</robotNamespace>
        <publishTf>1</publishTf>
        <publishWheelJointState>true</publishWheelJointState>
        <alwaysOn>true</alwaysOn>
        <updateRate>100.0</updateRate>
        <legacyMode>true</legacyMode>
        <leftJoint>base_to_wheel_left_joint</leftJoint>
        <rightJoint>base_to_wheel_right_joint</rightJoint>
        <wheelSeparation>${base_link_radius*2}</wheelSeparation>
        <wheelDiameter>${2*wheel_radius}</wheelDiameter>
        <broadcastTF>1</broadcastTF>
        <wheelTorque>30</wheelTorque>
        <wheelAcceleration>1.8</wheelAcceleration>
        <commandTopic>cmd_vel</commandTopic>
        <odometryFrame>odom</odometryFrame>
        <odometryTopic>odom</odometryTopic>
        <robotBaseFrame>base_footprint</robotBaseFrame>
    </plugin>
</gazebo>

```

其中关键参数：

- <robotNamespace>：机器人的命名空间，插件所有数据的发布和订阅都在该命名空间下。
- <leftJoint> 和 <rightJoint>：左右轮转动关节的 joint，控制器插件最终需要控制这两个 joint 转动。
- <wheelSeparation>：轮子间距。
- <wheelDiameter>：轮子半径。
- **wheelTorque>：这个怎么确定的？不重要吗？**
- <wheelAcceleration>：车轮转动加速度。
- <commandTopic>：控制器订阅的速度控制指令，ROS 中一般都命名为 vel_cmd。
- <odometryFrame>：里程计数据的参考坐标系，ROS 中一般都命名为 odom。

4.7.2 显示机器人模型

使用类似于以下的 launch 文件：

```
<launch>

    <!-- 设置launch文件的参数 -->
    <arg name="world_name" value="$(find mrobot_gazebo)/worlds/playground.world"/>
    <arg name="paused" default="false"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>

    <!-- 运行gazebo仿真环境 -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(arg world_name)" />
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)" />
        <arg name="use_sim_time" value="$(arg use_sim_time)" />
        <arg name="headless" value="$(arg headless)" />
    </include>

    <!-- 加载机器人模型描述参数（模型的路径在这里） -->
    <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find mrobot_gazebo)/urdf/mrobot.urdf.xacro'" />

    <!-- 运行joint_state_publisher节点，发布机器人的关节状态 -->
    <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" ></node>

    <!-- 运行robot_state_publisher节点，发布tf -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" output="screen" >
        <param name="publish_frequency" type="double" value="50.0" />
    </node>

    <!-- 在gazebo中加载机器人模型-->
```

```
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen"
args="--urdf -model mrobot -param robot_description"/>
</launch>
```

当我们想要加载一个机器人模型到 Gazebo 中，都可以使用上面的这种 launch 文件的代码形式，主要需要修改的就是我们要加载的机器人模型的路径。

接下来，我们运行这个 launch 文件就可以启动 Gazebo，并且在其中看到我们的机器人模型了。另外，由于这个模型已经订阅了 vel_cmd 话题，我们也可以发布 vel_cmd 话题消息来对 Gazebo 中的机器人进行控制了。

4.7.3 摄像头仿真

类似于机器人模型中的差速控制器插件，传感器的 Gazebo 插件也需要在 URDF 模型中进行配置，在原有的摄像头模型中添加 <gazebo> 标签，代码如下：

```
<gazebo reference="${prefix}_link">
  <material>Gazebo/Black</material>
</gazebo>

<gazebo reference="${prefix}_link">
  <sensor type="camera" name="camera_node">
    <update_rate>30.0</update_rate>
    <camera name="head">
      <horizontal_fov>1.3962634</horizontal_fov>
      <image>
        <width>1280</width>
        <height>720</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
  </sensor>
</gazebo>
```

```

</camera>

<plugin name="gazebo_camera" filename="libgazebo_ros_camera.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>0.0</updateRate>
    <cameraName>/camera</cameraName>
    <imageTopicName>image_raw</imageTopicName>
    <cameraInfoTopicName>camera_info</cameraInfoTopicName>
    <frameName>camera_link</frameName>
    <hackBaseline>0.07</hackBaseline>
    <distortionK1>0.0</distortionK1>
    <distortionK2>0.0</distortionK2>
    <distortionK3>0.0</distortionK3>
    <distortionT1>0.0</distortionT1>
    <distortionT2>0.0</distortionT2>
</plugin>
</sensor>
</gazebo>

```

这里添加了两个 `<gazebo>` 标签：

第一个 `<gazebo>` 标签用来设置摄像头模型在 Gazebo 中的 material，与之前提到的为每一个 link 添加 `<gazebo>` 标签作用相同。

第二个 `<gazebo>` 标签设置摄像头插件。在加载摄像头插件的时候，需要使用 `<sensor>` 标签来包含传感器的各种属性。这里设置摄像头传感器，需要设置 type 为 camera，传感器名字 name 可以自由设置；然后使用 `<camera>` 标签具体描述摄像头的参数，包括分辨率、编码格式、图像范围、噪声参数等；最后使用 `<plugin>` 标签加载摄像头的插件文件 libgazebo_ros_camera.so，并设置插件的一些参数，包括命名空间、发布图像的话题、参考坐标系等。

启动 Gazebo 下的机器人仿真之后，输入如下命令，使用 rqt 工具来看到摄像头的图像：

```
$ rqt_image_view
```

注意需要选择合适的话题才可以正确查看。

4.7.4 Kinect 仿真

添加如下 `<gazebo>` 标签：

```

<gazebo reference="${prefix}_link">
    <sensor type="depth" name="${prefix}">
        <always_on>true</always_on>
        <update_rate>20.0</update_rate>

```

```

<camera>
    <horizontal_fov>${60.0*M_PI/180.0}</horizontal_fov>
    <image>
        <format>R8G8B8</format>
        <width>640</width>
        <height>480</height>
    </image>
    <clip>
        <near>0.05</near>
        <far>8.0</far>
    </clip>
</camera>
<plugin name="kinect_${prefix}_controller" filename="libgazebo_ros_openni_kinect.so">
    <cameraName>${prefix}</cameraName>
    <alwaysOn>true</alwaysOn>
    <updateRate>10</updateRate>
    <imageTopicName>rgb/image_raw</imageTopicName>
    <depthImageTopicName>depth/image_raw</depthImageTopicName>
    <pointCloudTopicName>depth/points</pointCloudTopicName>
    <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
    <depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>
    <frameName>${prefix}_frame_optical</frameName>
    <baseline>0.1</baseline>
    <distortion_k1>0.0</distortion_k1>
    <distortion_k2>0.0</distortion_k2>
    <distortion_k3>0.0</distortion_k3>
    <distortion_tinaji
</gazebo>

```

这里为什么不需要一个 `<gazebo>` 标签来设置摄像头模型在 Gazebo 中的 material 了？

这里需要设置传感器类型为 `depth`, `<camera>` 中的参数与摄像头的类似, 分辨率和检测距离都可以在 Kinect 的手册中找到, 最后使用 `<plugin>` 标签加载 Kinect 的插件文件 `libgazebo_ros_openni_kinect.so`, 并设置插件相关参数。

启动 Gazebo 下的机器人仿真之后, 输入如下命令, 使用 `rviz` 来查看 Kinect 的点云数据:

```
$ rosrun rivz rviz
```

注意需要设置 fixed frame 为 camera_frame_optical，并且添加一个 PointCloud2 插件，并设置插件的订阅话题为“/camera/depth/points”才可以正确查看。

4.7.5 激光雷达仿真

添加如下 <gazebo> 标签：

```
<gazebo reference="${prefix}_link">
    <material>Gazebo/Black</material>
</gazebo>

<gazebo reference="${prefix}_link">
    <sensor type="ray" name="rplidar">
        <pose>0 0 0 0 0 0</pose>
        <visualize>false</visualize>
        <update_rate>5.5</update_rate>
        <ray>
            <scan>
                <horizontal>
                    <samples>360</samples>
                    <resolution>1</resolution>
                    <min_angle>-3</min_angle>
                    <max_angle>3</max_angle>
                </horizontal>
            </scan>
            <range>
                <min>0.10</min>
                <max>6.0</max>
                <resolution>0.01</resolution>
            </range>
            <noise>
                <type>gaussian</type>
                <mean>0.0</mean>
                <stddev>0.01</stddev>
            </noise>
        </ray>
    <plugin name="gazebo_rplidar" filename="libgazebo_ros_laser.so">
        <topicName>/scan</topicName>
        <frameName>laser_link</frameName>
```

```
</plugin>
</sensor>
</gazebo>
```

激光雷达的传感器类型为 ray，rplidar 的相关参数可以在产品手册中找到，`<ray>` 标签中设置了如下的雷达参数：360° 检测范围、单圈 360 个采样点、5.5Hz 采样频率、最远 6m 检测范围等。最后使用 `<plugin>` 标签加载激光雷达的插件文件 libgazebo_ros_laser.so，并设置插件相关参数。

启动 Gazebo 下的机器人仿真之后，输入如下命令，使用 rviz 来查看激光雷达的点云数据：

```
$ rosrun rivz rviz
```

注意需要设置 fixed frame 为 base_footprint，并且添加一个 LaserScan 插件，并设置插件的订阅话题为“/scan”才可以正确查看。

Chapter 5

机器人 SLAM 与自主导航

三个重点问题：地图的精确建模、机器人准确定位、路径实时规划。

5.1 准备工作

5.1.1 机器人要求

ROS 中 SLAM 和自主导航的相关功能包可以用于各种移动机器人平台，但是为了达到最好的效果，对机器人硬件有如下的三个要求：

- 导航功能包对差分、轮式机器人的效果比较好，并且假设机器人可以直接使用速度指令进行控制，速度指令包括：linear（机器人在 xyz 三轴方向上的线速度，单位 m/s）;angular(机器人在 xyz 三轴方向上的角速度，单位是 rad/s)。
- 导航功能包要求机器人必须安装激光雷达等测距设备，可以获得环境的深度信息。
- 导航功能包以正方形和圆形机器人为模板进行开发，对于其他外形的机器人，虽然可以使用，但是效果可能不佳。

5.1.2 传感器信息

环境深度信息

针对激光雷达，ROS 在 sensor_msgs 包中定义了专用的数据结构——LaserScan，用于存储激光消息，其消息的核心内容如下：

- angle_min：可检测范围的起始角度。
- angle_max：可检测范围的终止角度，与 angle_min 组成激光雷达的可检测范围。
- angle_increment：采集到相邻数据帧之间的角度步长。

- time_increment: 采集到相邻数据帧之间的时间步长, 当传感器处于相对运动状态时进行补偿使用。
- scan_time: 采集一帧数据所需要的时间。
- range_min: 最近可检测深度的阈值。
- range_max: 最远可检测深度的阈值。
- ranges: 一帧深度数据的存储数组。

如果使用的机器人没有激光雷达, 但是配备有 Kinect 等 RGB-D 摄像头, 也可以通过红外摄像头获取周围环境中的深度信息。但是 RGB-D 摄像头获取的原始深度信息是三维点云数据, 我们需要将三维数据转换成二维数据, 即只抽取其中的一行, 重新封装为 LaserScan 消息, 就可以获得需要的二维激光雷达信息。

ROS 中也提供了相应功能包——depthimage_to_laserscan 来实现三维点云数据到二维的转换功能, 从而将点云深度数据转换成激光数据。可以在 launch 文件中使用如下方法调用:

```
<node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan" name="depth
image_to_laserscan" output="screen">
  <remap from="image" to="/camera/depth_registered/image_raw"/>
  <remap from="camera_info" to="/camera/depth_registered/camera_info"/>
  <remap from="scan" to="/kinect_scan"/>
  <param name="output_frame_id" value="/camera_link"/>
</node>
```

里程计信息

里程计根据传感器获取的数据来估计机器人随时间发生的位置变化, 在机器人平台中, 较为常见的里程计是编码器。里程计一般根据速度对时间的积分求得位置, 这种方法对误差十分敏感, 所以采取如精确的数据采集、设备标定、数据滤波等措施是十分必要的。

导航功能包要求机器人能够发布里程计 nav_msgs/Odometry 消息, 其中包含机器人在自由空间中的位置和速度的估算值:

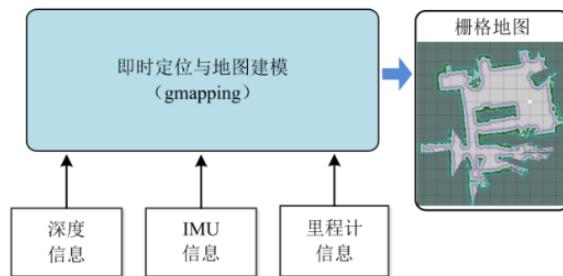
- pose: 机器人当前位置坐标, 包括机器人的 xyz 三轴位置与方向参数, 以及用于校正误差的协方差矩阵。
- twist: 机器人当前的运动状态, 包括 xyz 三轴的线速度与角速度, 以及用于校正误差的协方差矩阵。

上述数据结构中, 用于滤波算法的协方差矩阵, 在精度要求不高的机器人系统中可以使用默认的协方差矩阵; 而在精度要求较高的系统中, 需要先对机器人精确建模后, 再通过仿真、实验等方法确定该矩阵的具体数值。

5.2 gmapping

gmapping 已经集成在 ROS 中，是移动机器人中使用最多的 SLAM 算法。这个算法是一种基于 Rao-Blackwellized 的粒子滤波的 SLAM 方法。基于粒子滤波的算法用许多加权粒子表示路径的后验概率，每个粒子都给出一个重要性因子。但是，它们通常需要大量的粒子才能获得比较好的结果，从而增加该算法的计算复杂性。

5.2.1 gmapping 功能包介绍



gmapping 功能包订阅机器人的深度信息、IMU 信息和里程计信息，同时完成一些必要参数的配置，即可创建并输出基于概率的二维栅格地图。

gmapping 功能包向用户开放的接口如下：

(1) 话题和服务：

	名称	类型	描述
Topic 订阅	tf	tf/tfMessage	用于激光雷达坐标系、基坐标系、 里程计坐标系之间的变换
	scan	sensor_msgs/LaserScan	激光雷达扫描数据
Topic 发布	map_metadata	nav_msgs/MapMetaData	发布地图 Meta 数据
	map	nav_msgs/OccupancyGrid	发布地图栅格数据
	~entropy	std_msgs/Float64	发布机器人姿态分布熵的估计
Service	dynamic_map	nav_msgs/GetMap	获取地图数据

(2) 坐标变换：

	TF 变换	描述
必需的 TF 变换	<scan frame> → base_link	激光雷达坐标系与基坐标系之间的变换，一般由 robot_states_publisher 或者 static_transform_publisher 发布
	base_link → odom	基坐标系与里程计坐标系之间的变换，一般由里程计节点发布
发布的 TF 变换	map → odom	地图坐标系与机器人里程计坐标系之间的变换，估计机器人在地图中的位姿

(3) 参数

gmapping 功能包中可供配置的参数有很多，使用后将重点参数记录在下面：

5.2.2 gmapping 节点配置与运行

首先创建一个运行 gmapping 节点的 launch 文件 (gmapping.launch)，主要用于节点参数的配置，文件代码如下：

```

<launch>

  <arg name="scan_topic" default="scan" />

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen" clear_params="true">
    <param name="odom_frame" value="odom"/>
    <param name="map_update_interval" value="5.0"/>
    <!-- Set maxUrange < actual maximum range of the Laser -->
    <param name="maxRange" value="5.0"/>
    <param name="maxUrange" value="4.5"/>
    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.075"/>
    <param name="ogain" value="3.0"/>
    <param name="lskip" value="0"/>
    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
  </node>
</launch>

```

```

<param name="stt" value="0.02"/>
<param name="linearUpdate" value="0.5"/>
<param name="angularUpdate" value="0.436"/>
<param name="temporalUpdate" value="-1.0"/>
<param name="resampleThreshold" value="0.5"/>
<param name="particles" value="80"/>
<param name="xmin" value="-1.0"/>
<param name="ymin" value="-1.0"/>
<param name="xmax" value="1.0"/>
<param name="ymax" value="1.0"/>
<param name="delta" value="0.05"/>
<param name="llsamplerange" value="0.01"/>
<param name="llsamplestep" value="0.01"/>
<param name="lasamplerange" value="0.005"/>
<param name="lasamplestep" value="0.005"/>
<remap from="scan" to="$(arg scan_topic)"/>
</node>
</launch>

```

这些参数都有默认值，可以等 SLAM 可以正常运行之后，再考虑调整优化参数。但是需要重点检查这两个参数：

- 里程计坐标系的设置，odom_frame 参数需要和机器人本身的里程计坐标系一致。
- 激光雷达的话题名，gmapping 节点订阅的激光雷达话题名是 “/scan”，如果与机器人发布的激光雷达话题名不一致，需要使用 <remap> 进行重映射。

接下来创建一个启动 gmapping 例程的文件 (gmapping_demo.launch)。主要代码如下：

```

<launch>
  <include file="$(find mrobot_navigation)/launch/gmapping.launch"/>
  <!-- 启动 rviz -->
  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find mrobot_navigation)/rviz/>
</launch>

```

这个 launch 文件主要包含：

- 启动之前创建的 gmapping 节点。
- 启动 rviz 界面，查看传感器和地图构建的实时信息。

5.2.3 在 Gazebo 中仿真 SLAM(gmapping)

启动 Gazebo 仿真环境和 gmapping 节点：

```
$ roslaunch mrobot_gazebo mrobot_laser_nav_gazebo.launch  
$ roslaunch mrobot_navigation gmapping_demo.launch
```

启动键盘控制节点：

```
$ roslaunch mrobot_teleop mrobot_teleop.launch
```

操作机器人在地图中运动一圈，就可以基本完成地图的构建了，构建完成后，可以使用如下命令保存构建好的地图：

```
$ rosrun map_server map_saver
```

保存好的文件默认文件名为 map，包含：

- 一个 map.pgm 地图数据文件
- 一个 map.yaml 地图配置文件：其中包含关联的地图数据文件、地图分辨率、起始位置、地图数据和阈值等配置参数。

5.3 hector-slam

hector_slam 功能包使用高斯牛顿法，不需要里程计数据，只根据激光信息便可构建地图。因此该功能包可以很好的在空中机器人、手持构图设备以及特种机器人中使用。但是该算法需要高更新频率小测量噪声的激光雷达，当只有低更新率的激光传感器时，即便测距估计很精确，对该系统都会出现一定的问题。

5.4 cartographer

cartographer 是 Google 的实时室内建图项目，考虑到基于模拟策略的粒子滤波方法在较大环境中对内存和计算资源的需求较高，cartographer 采用基于图网络的优化方法。

5.5 rgbdslam

不同于上述方法，rgbdslam 是一个 3D SLAM 功能包。可以实现三维信息的地图构建，可以把周围环境的三维模型全部构建出来，机器人不仅知道地图中的什么位置有一个障碍物，而且知道该障碍物是什么。

5.6 ORB_SLAM

ORB_SLAM 是一个基于特征点的实时单目 SLAM 系统，能够实时解算摄像机的移动轨迹，同时构建简单的三维点云地图，在大范围中做闭环检测，并实时进行全局重定位，不仅适用于手持设备获取的一组连续图像，同时适用于汽车行驶过程中获取的连续图像。

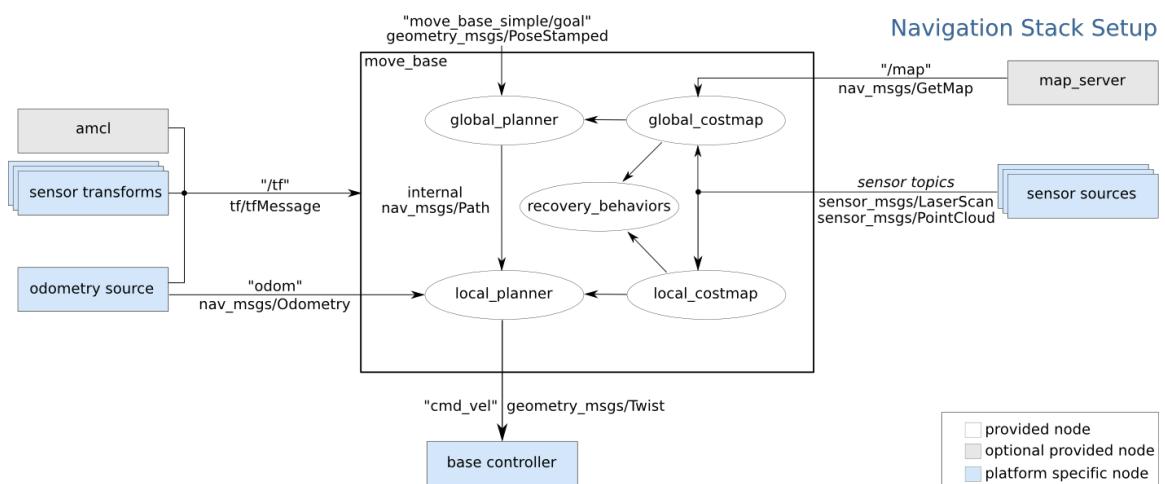
5.7 导航功能包

5.7.1 导航框架

导航的关键是机器人的定位和路径规划两大部分，ROS 也对应地提供了两个功能包：

- move_base：实现机器人导航中的最优路径规划。
- amcl：实现二维地图中的机器人定位。

在上述两个功能包的基础上，ROS 提供了一整套导航功能框架，如下图所示：



机器人需要发布必要的传感器信息和导航的目标位置；`move_base` 功能包提供导航的主要运行、交互接口；`amsl` 功能包对机器人进行精确定位，保障导航路径的准确性。

机器人通过 ROS 发布 `sensor_msgs/LaserScan` 或者 `sensor_msgs/PointCloud` 格式的消息，也就是二维激光信息或者三维点云信息，从而为导航功能包提供机器人的传感器信息，以达到实时避障的效果。

其次导航包要求机器人发布 `nav_msgs/Odometry` 格式的里程计信息，同时也要发布相应的 TF 变换。

最后导航功能包的输出是 `geometry_msgs/Twist` 格式的控制指令，这就要求机器人控制节点具备解析控制指令中线速度、角速度的能力，并且最终通过这些指令控制机器人完成相应的运动。

5.7.2 move_base 功能包

move_base 功能包中主要由两个规划器组成：

- 全局路径规划 (global_planner)：根据给定的目标位置和全局地图进行总体的路径规划（使用 Dijkstra 或者 A* 算法）。
- 本地路径规划 (local_planner)：针对地图信息和机器人附近随时可能出现的障碍物规划机器人每个周期内应该行使的路径。使之尽可能符合全局最优路径（使用 Dynamic Window Approaches 算法搜索躲避和行进的多条路径，综合各评价标准（是否会撞击障碍物、所需要的时间）选取最优路径，并且计算行使周期内的线速度和角速度，避免与动态出现的障碍物发生碰撞）。

move_base 功能包的一些接口：

(1) 话题和服务

	名称	类型	描述
动作 订阅	move_base/ goal	move_base_msgs/ MoveBaseActionGoal	move_base 的运动规划目标
	move_base/ cancel	actionlib_msgs/GoalID	取消特定目标的请求
动作 发布	move_base/ feedback	move_base_msgs/ MoveBaseActionFeedback	反馈信息 含有机器人底盘的坐标
	move_base/ status	actionlib_msgs/ GoalStatusArray	发送到 move_base 的目标状态信息
	move_base/ result	move_base_msgs/ MoveBaseActionResult	此处 move_base 操作结果为空
话题 订阅	move_base_ simple/goal	geometry_msgs/ PoseStamped	为无需追踪目标执行状态的用户 提供一个非 action 接口
话题 发布	cmd_vel	geometry_msgs/Twist	输出到机器人底盘的速度指令
服务	~make_plan	nav_msgs/GetPlan	允许用户从 move_base 获取给定目标 的路径规划，但不会执行该路径规划
	~clean_unknow _space	std_srvs/Empty	允许用户直接清除机器人周围的未知 空间，适合 costmap 停止很长时间之后 在一个全新环境中重新启动时使用
	~clear_costmaps	std_srvs/Empty	允许用户命令 move_base 节点清除 costmap 中的障碍，这可能会导致 机器人撞上障碍物，谨慎使用

(2) 参数

待补充

5.7.3 amcl 功能包

自主定位即机器人在任意状态下都可以推算出自己在地图中所处的位置。ROS 为开发者提供了一种自适应（或 kld 采样）的蒙特卡洛定位方法（amcl），这是一种概率统计方法，针对已有地图使用粒子滤波器跟踪一个机器人的姿态。

amcl 功能包的一些接口：

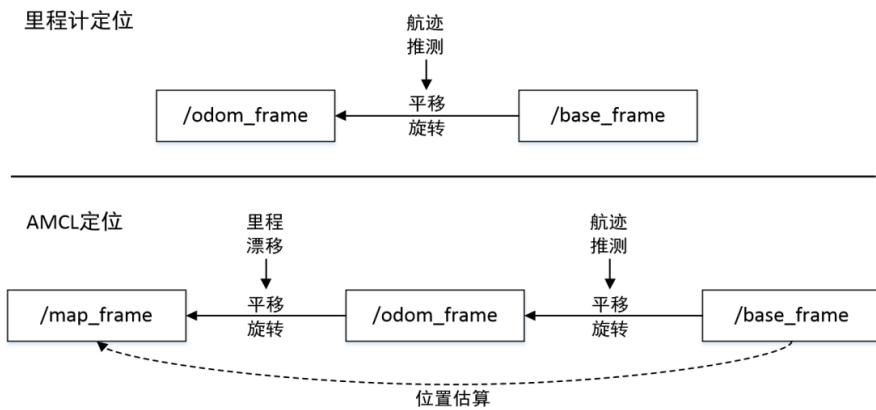
(1) 话题和服务

	名称	类型	描述
话题 订阅	Scan	sensor_msgs/LaserScan	激光雷达数据
	Tf	tf/tfMessage	坐标变换信息
	initialpose	geometry_msgs/ PoseWithCovarianceStamped	用来初始化粒子滤波器的均 值和协方差
	map	nav_msgs/OccupancyGrid	设置 use_map_topic 参数时 , amcl 订阅 map 话题以获取 地图数据, 用于激光定位
话题 发布	amcl_pose	geometry_msgs/ PoseWithCovarianceStamped	机器人在地图中的位姿估计, 带有协方差信息
	particlecloud	geometry_msgs/PoseArray	粒子滤波器维护的位姿估计 集合
	Tf	tf/tfMessage	发布从 odom 到 map 的转换
服务	global_localization	std_srvs/Empty	初始化全局定位, 所有粒子 被随机撒在地图的空闲区域
	request_nomotion_update	std_srvs/Empty	手动执行更新并发布更新的 粒子
服务 调用	static_map	nav_msgs/GetMap	amcl 调用该服务来获取地图 数据

(2) 参数

待补充

(3) 坐标变换



里程计定位和 amcl 定位的区别：

- 里程计定位：只是通过里程计的数据来处理/base 和/odom 之间的 TF 变换
- amcl 定位：可以估算机器人在地图坐标系/map 下的位姿信息，提供/base、/odom、/map 之间的 TF 变换。

5.7.4 代价地图的配置

代价地图的名称的意义是什么？

导航功能包使用两种代价地图存储周围环境中的障碍信息：一种用于全局路径规划 (global_costmap)，一种用于本地路径规划和实时避障 (local_costmap)。

两种代价地图需要使用共用的配置文件：通用配置文件；以及分别独立的配置文件：全局规划配置文件和本地规划配置文件。

通用配置文件

代价地图用来存储周围环境的障碍信息，其中需要声明地图专注的机器人传感器信息，以便于地图信息的更新。对于两种代价地图通用的配置选项，创建一个名为 costmap_common_params.yaml 的配置文件，代码如下：

```

obstacle_range: 2.5
raytrace_range: 3.0
# footprint: [[0.175, 0.175], [0.175, -0.175], [-0.175, -0.175], [-0.175, 0.175]]
# footprint_inflation: 0.01
robot_radius: 0.175
inflation_radius: 0.15
max_obstacle_height: 0.6
min_obstacle_height: 0.0

```

```
observation_sources: scan
scan: {data_type: LaserScan, topic: /scan, marking: true, clearing: true, expected_update_rate: 0}
```

其中代码段：

```
obstacle_range: 2.5
raytrace_range: 3.0
```

用来设置代价地图中障碍物的相关阈值：obstacle_range 参数用来设置机器人检测障碍物的最大范围，若设置为 2.5，则表示在 2.5m 范围内检测到的障碍物信息才会在地图中进行更新；raytrace_range 参数用来设置机器人检测自由空间的最大范围，若设置为 3.0，则表示在 3m 范围内，机器人将根据传感器信息清除范围内的自由空间（这里的自由空间是不是指没有障碍物的空间）。

代码段：

```
# footprint: [[0.175, 0.175], [0.175, -0.175], [-0.175, -0.175], [-0.175, 0.175]]
# footprint_inflation: 0.01
robot_radius: 0.175
inflation_radius: 0.1
```

footprint 参数用来设置机器人在二维地图上占用的面积，参数以机器人的中心作为坐标原点。如果机器人的外形是圆形的，则需要设置机器人的外形半径 robot_radius。inflation_radius 参数用来设置障碍物的膨胀系数，也就是机器人应该与障碍物保持的最小安全距离，这里设置为 0.1，表示机器人规划的路径应该与障碍物保持 0.1m 以上的安全距离。

代码段：

```
max_obstacle_height: 0.6
min_obstacle_height: 0.0
```

这两个参数用来描述障碍物的最大高度和最小高度。（有什么用呢？）

代码段：

```
observation_sources: scan
scan: {data_type: LaserScan, topic: /scan, marking: true, clearing: true, expected_update_rate: 0}
```

observation_sources 参数列出了代价地图需要关注的所有传感器信息，每个传感器信息都会在后面列举出详细内容。

这里以激光雷达为例：

- sensor_frame 表示传感器的参考系名称；

- `data_type` 表示激光数据或者点云数据使用的消息类型；
- `topic_name` 表示传感器发布的话题名称；
- `marking` 和 `clearing` 表示是否需要使用传感器的实时信息来添加或者清除代价函数中的障碍物信息。

全局规划配置文件

全局规划配置文件用来存储全局代价地图的配置参数，命名为 `global_costmap_params.yaml`，代码如下：

```
global_costmap:  
  global_frame: map  
  robot_base_frame: base_footprint  
  update_frequency: 1.0  
  publish_frequency: 1.0  
  static_map: true  
  rolling_window: false  
  resolution: 0.01  
  transform_tolerance: 1.0  
  map_type: costmap
```

- `global_frame` 参数用来表示全局代价函数需要在哪个参考系下匀性，这里选择了 `map` 参考系；
- `robot_base_frame` 参数用来设置代价地图可以参考的机器人本体的坐标系；
- `update_frequency` 参数决定全局地图信息更新的频率；
- `static_map` 参数用来决定代价地图是否需要根据 `map_server` 提供的地图信息进行初始化。

本地规划配置文件

本地规划配置文件用来存储本地代价地图的配置参数，命名为 `local_costmap_params.yaml`，代码如下：

```
local_costmap:  
  global_frame: odom  
  robot_base_frame: base_footprint  
  update_frequency: 3.0  
  publish_frequency: 1.0  
  static_map: true
```

```
rolling_window: false
width: 6.0
height: 6.0
resolution: 0.01
transform_tolerance: 1.0
```

前面几个参数的意义与全局规划配置文件相同，另外：

- publish_frequency 参数用来设置代价地图发布可视化信息的频率；
- rolling_window 参数用来设置在机器人移动过程中是否需要滚动窗口，以保持机器人在中心；
- width、height、resolution 参数设置代价地图的长、高、分辨率（米/格）。

5.7.5 本地规划器配置

本地规划器的主要作用为：根据规划的全局路径计算发布给机器人的速度控制指令。该规划器要根据机器人的规格配置相关参数，创建名为 base_local_planner_params.yaml 的配置文件，代码如下：

```
controller_frequency: 3.0
recovery_behavior_enabled: false
clearing_rotation_allowed: false

TrajectoryPlannerROS:
    max_vel_x: 0.5
    min_vel_x: 0.1
    max_vel_y: 0.0 # zero for a differential drive robot
    min_vel_y: 0.0
    max_vel_theta: 1.0
    min_vel_theta: -1.0
    min_in_place_vel_theta: 0.5
    escape_vel: -0.1
    acc_lim_x: 1.5
    acc_lim_y: 0.0 # zero for a differential drive robot
    acc_lim_theta: 1.2

    holonomic_robot: false
    yaw_goal_tolerance: 0.1 # about 6 degrees
    xy_goal_tolerance: 0.1 # 10 cm
```

```
latch_xy_goal_tolerance: false
pdist_scale: 0.9
gdist_scale: 0.6
meter_scoring: true

heading_lookahead: 0.325
heading_scoring: false
heading_scoring_timestep: 0.8
occdist_scale: 0.1
oscillation_reset_dist: 0.05
publish_cost_grid_pc: false
prune_plan: true

sim_time: 1.0
sim_granularity: 0.025
angular_sim_granularity: 0.025
vx_samples: 8
vy_samples: 0 # zero for a differential drive robot
vtheta_samples: 20
dwa: true
simple_attractor: false
```

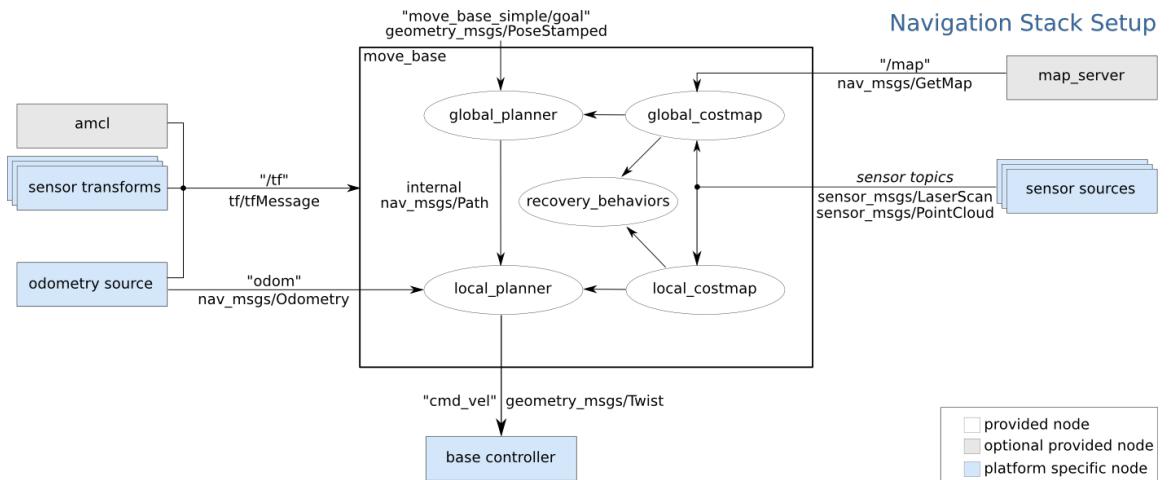
该配置文件中，声明机器人本地规划采用 Trajectory Rollout 算法，并设置算法中需要用到的机器人速度、加速度的阈值等参数。

Chapter 6

Navigation 详细学习

学习代码版本: melodic-devel-1.16.7

导航包功能框架图:



6.1 三维空间中的刚体运动 (基础知识补充)

为什么要研究这个问题? 因为当我们描述机器人的位姿时, 就是在描述一个刚体在三维空间中的运动。

三维空间中, 刚体的运动可以用两个概念来表示: 旋转和平移。平移比较简单一些, 一般用一个表示位移的向量来表示。而旋转则有多种表示方法, 例如旋转矩阵、旋转向量等等, 不同的表示方法各有优劣

6.1.1 向量

在描述旋转矩阵前我们先明确向量这个概念。向量是空间中的一个具体实物且不和任何实数相关联。为了描述向量，应该先确定一个具体的坐标系，明确该坐标系的线性基 $[e_1 \ e_2 \ e_3]$ 后才能够确定一个向量 a 在该坐标系下的坐标：

$$a = [e_1 \ e_2 \ e_3] \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = a_1 e_1 + a_2 e_2 + a_3 e_3 \quad (6.1)$$

也就是说：向量坐标的具体取值，和向量本身和选取的坐标系相关。

接下来介绍向量间的两种运算：内积（点乘）和外积（叉乘）

内积可以描述向量之间的投影关系：

$$a \cdot b = a^T b = \sum_{i=1}^3 a_i b_i = |a||b| \cos \langle a, b \rangle \quad (6.2)$$

外积可以表示向量的旋转：

$$a \times b = \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} b = a^\wedge b \quad (6.3)$$

外积的方向垂直于这两个向量，大小为 $|a||b|\sin \langle a, b \rangle$ ，是两个向量张成的四边形的有向面积。

其中我们引入了一个符号： \wedge ，把 a 写成一个反对称矩阵，可以将其记成一个反对称符号。因此我们就把外积 $a \times b$ 写成了矩阵和向量的乘法 $a^\wedge b$ ，从而将其变成了线性运算。

用外积表示旋转：

在右手法则下，用右手的四个手指（握拳）从 a 转向 b ，大拇指的朝向就是旋转向量的方向，也就是 $a \times b$ 的方向，它的大小由 a 和 b 的夹角决定。

6.1.2 欧式变换

假设存在两个坐标系：一个世界坐标系，是一个惯性系，认为它是固定不动的；另一个是一个机器人坐标系，是随机器人移动的坐标系。假设机器人观察到了某个向量 p ，它在这两个坐标系中分别有一套坐标。前面说了，向量是一个客观存在的实体，那么必然有一个关系能够将这两套坐标联系起来。

这个关系就是欧式变换。因为机器人的运动是一个刚体运动，所以同一个向量在不同坐标系下的模长和方向都不会发生变化。这样一个欧式变换就是由一个旋转和一个平移两部分组成。

6.1.3 旋转矩阵

我们先考虑欧式变换中的旋转变换：

我们设某个单位正交基 (e_1, e_2, e_3) ，经过一次旋转之后，变成了 (e'_1, e'_2, e'_3) ，那么对于同一个向量 a ，在两个坐标系下有不同的坐标：

$$\begin{bmatrix} e_1 & e_2 & e_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} e'_1 & e'_2 & e'_3 \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} \quad (6.4)$$

等式两边同时左乘 $\begin{bmatrix} e_1^T \\ e_2^T \\ e_3^T \end{bmatrix}$ ，则左边的系数变成了单位阵，因此得到了两个坐标系的旋转关系：

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} e_1^T e'_1 & e_1^T e'_2 & e_1^T e'_3 \\ e_2^T e'_1 & e_2^T e'_2 & e_2^T e'_3 \\ e_3^T e'_1 & e_3^T e'_2 & e_3^T e'_3 \end{bmatrix} \begin{bmatrix} a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = Ra' \quad (6.5)$$

我们把中间的矩阵拿出来，定义成一个矩阵 R ，它描述了旋转本身，又称为旋转矩阵，是由两组基之间的内积组成的。而且它是一个行列式为 1 的正交矩阵（矩阵的转置等于矩阵的逆）。它的逆（亦即转置）描述了一个相反的的旋转。

接下来，加入欧式变换中的平移部分：

平移可以简单地用一个平移向量 t 来表示，则欧式变换后的坐标为：

$$a' = Ra + t \quad (6.6)$$

因此我们可以用一个旋转矩阵 R 和一个平移向量 t 完整地描述一个欧氏空间地坐标变换。

6.1.4 变换矩阵与齐次坐标

上面我们已经可以完整地表达欧氏空间地旋转和平移了，但是这里的变换并不是线性的，如果进行两次变换，则会得到下面的结果：

$$c = R_2(R_1a + t_1) + t_2 \quad (6.7)$$

这样的形式在变换多次之后会变得非常复杂，因此我们需要引入齐次坐标，并将变换矩阵重写为：

$$\begin{bmatrix} a' \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} a \\ 1 \end{bmatrix} = T \begin{bmatrix} a \\ 1 \end{bmatrix} \quad (6.8)$$

这里的矩阵 T 我们称为变换矩阵。

这其实是一个数学的技巧：我们在一个三维向量的末尾添加一个 1，将其变为四维向量，称为齐次变换。对于四维向量，我们可以把旋转和平移写在一个变换矩阵中，从而使得整个关系变为线性关系。

变换矩阵的反向变换：

$$T^{-1} = \begin{bmatrix} R^T & -R^T t \\ 0^T & 1 \end{bmatrix} \quad (6.9)$$

6.1.5 旋转向量

我们容易想到，一个旋转只有三个自由度，一个三维刚体运动（包括旋转和平移）有六个自由度，但是对应的旋转矩阵有 9 个量，变换矩阵有 16 个量，很明显旋转矩阵和变换矩阵对于刚体运动的描述是冗余的，那么有没有紧凑的描述方式呢？

联系前面关于向量外积部分的说明，我们知道任意的旋转都可以用一个旋转轴和一个旋转角来刻画。因此我们想到，可以使用一个向量，其方向与旋转轴一致，长度等于旋转角，这样的向量一般称为旋转向量。这样的话，我们只需要一个三维的向量就可以描述旋转，再加上一个三维的平移向量，一次变换我们正好可以用一个六维的向量来描述。

这里不过多介绍旋转向量的问题，旋转向量在李代数的相关知识中会有讲解。我们介绍一下旋转向量和旋转矩阵之间的转换关系：

假设有一个旋转轴为 n ，旋转角度为 θ 的旋转，旋转向量到旋转矩阵的转换由罗德里格斯公式 (Rodrigues's Formula) 可以得到：

$$R = \cos\theta I + (1 - \cos\theta)nn^T + \sin\theta n^\wedge \quad (6.10)$$

符号 \wedge 是向量到反对称矩阵的转换符。

旋转矩阵到旋转向量的转换：

对于转角 θ ：

$$\begin{aligned} tr(R) &= \cos\theta tr(I) + (1 - \cos\theta)tr(nn^T) + \sin\theta tr(n^\wedge) \\ &= 3\cos\theta + (1 - \cos\theta) \\ &= 1 + 2\cos\theta \end{aligned} \quad (6.11)$$

因此有：

$$\theta = \arccos\left(\frac{tr(R) - 1}{2}\right) \quad (6.12)$$

对于转轴 n ：

由于旋转轴上的向量在旋转后不发生改变，说明：

$$Rn = n \quad (6.13)$$

因此，转轴 n 是矩阵 R 特征值 1 对应的特征向量，求解此方程再归一化，就得到了旋转轴。当然也可以直接从“旋转轴经过旋转后不变”的几何角度看待这个方程。

6.1.6 欧拉角

用一种非常直观的方式来描述旋转——欧拉角。把一个旋转分解成 3 次绕不同轴的旋转，由于旋转的顺序等可以有不同的定义，因此欧拉角也有很多种，以旋转顺序 ZYX 为例，可以得到 rpy 角：

- 绕物体的 Z 轴旋转，得到偏航角 yaw；
- 绕旋转之后的 Y 轴旋转，得到俯仰角 pitch；
- 绕旋转之后的 X 轴旋转，得到横滚角 roll。

欧拉角的一个重大缺点是万向锁问题，当俯仰角为 $\pm 90^\circ$ 时，第一次旋转与第三次旋转将使用同一个轴，使得系统丢失了一个自由度，这被称为奇异性问题。另外，可以证明，当我们想用 3 个实数来表示三维旋转时，都会不可避免地碰到奇异性问题。

6.1.7 四元数

欧拉角和旋转向量虽然是紧凑的，但是具有奇异性，我们找不到不带有奇异性的三维向量来描述旋转，因此我们需要用到四元数，它既是紧凑的，也没有奇异性。

一个四元数包含一个实部和三个虚部：

$$q = q_0 + q_1 i + q_2 j + q_3 k \quad (6.14)$$

这三个虚部满足如下关系式：

$$\begin{cases} i^2 = j^2 = k^2 = -1 \\ ij = k, \quad ji = -k \\ jk = i, \quad kj = -i \\ ki = j, \quad ik = -j \end{cases} \quad (6.15)$$

有时也将四元数用一个标量和一个向量来表示：

$$q = [s, \mathbf{v}], \quad s = q_0 \in \mathcal{R}, \quad \mathbf{v} = [q_1, q_2, q_3]^T \in \mathcal{R}^3 \quad (6.16)$$

类比复数，我们知道一个模长为 1 的复数可以表示复平面上的一个纯旋转（没有长度缩放），同样，也可以使用单位四元数表示三维空间中的一个旋转。

四元数的物理意义这里先不进行深入讨论，先给出四元数与其他旋转表示方式的转换关系：

四元数与旋转向量的转换:

假设某个旋转是绕单位向量 $\mathbf{n} = [n_x, n_y, n_z]^T$ 进行了角度为 θ 的旋转, 那么这个旋转的四元数形式为:

$$\mathbf{q} = [\cos \frac{\theta}{2}, n_x \sin \frac{\theta}{2}, n_y \sin \frac{\theta}{2}, n_z \sin \frac{\theta}{2}] \quad (6.17)$$

反之有:

$$\theta = 2\arccos q_0 [n_x, n_y, n_z]^T = [q_1, q_2, q_3]^T / \sin \frac{\theta}{2} \quad (6.18)$$

从上式我们可以看出, 如果对 θ 加 2π , 则理论上我们得到一个相同的旋转, 但是此时的四元数却变成了 $-q$, 因此: 任意的旋转都可以由两个互为相反数的四元数表示。

取 θ 为 0, 则得到一个没有任何旋转的四元数:

$$\mathbf{q}_0 = [\pm 1, 0, 0, 0]^T \quad (6.19)$$

四元数与旋转矩阵的转换:

设四元数为: $q = q_0 + q_1 i + q_2 j + q_3 k$, 则有:

$$R = \begin{bmatrix} 1 - 2q_2^2 - 2q_3^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & 1 - 2q_1^2 - 2q_3^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & 1 - 2q_1^2 - 2q_2^2 \end{bmatrix} \quad (6.20)$$

反之, 假设旋转矩阵 $R = m_{ij}$, $i, j \in [1, 2, 3]$, 则有:

$$\begin{aligned} q_0 &= \frac{\sqrt{\text{tr}(R) + 1}}{2}, & q_1 &= \frac{m_{23} - m_{32}}{4q_0} \\ q_2 &= \frac{m_{31} - m_{13}}{4q_0}, & q_3 &= \frac{m_{12} - m_{21}}{4q_0} \end{aligned} \quad (6.21)$$

需要注意, 由于 \mathbf{q} 和 $-\mathbf{q}$ 表示同一个旋转, 所以实际上一个 \mathbf{R} 对应的四元数表示并不是唯一的。

用四元数表示旋转:

假设一个空间中的三维点 \mathbf{p} , 经过一个由轴角 \mathbf{n} 和 θ 指定的旋转, 使得 \mathbf{p} 旋转为 \mathbf{p}' , 我们把三维空间的点 \mathbf{p} 用一个虚四元数来表示:

$$\mathbf{p} = [0, x, y, z] = [0, \mathbf{v}] \quad (6.22)$$

则根据根据旋转向量与四元数的转换关系得到该旋转的四元数表示:

$$\mathbf{q} = [\cos \frac{\theta}{2}, \mathbf{n} \sin \frac{\theta}{2}] \quad (6.23)$$

那么旋转后的点 \mathbf{p}' 可以这样用四元数表示：

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \quad (6.24)$$

上式的计算结果实部为 0，是纯虚四元数，其虚部的三个分量就是旋转后的点的三维坐标。

6.1.8 四元数的运算

待补充。

6.2 move_base 源码学习

Movebase 的主干部分是一个 *Action* 服务器，接收用户发送的目标位置，并调用全局规划器和局部规划器，基于各层代价地图的信息进行路径规划，得到最优路径，向用户反馈机器人速度指令，驱动机器人按照指令运动，最终到达目标位置。这里不涉及到规划路径和更新地图的具体算法和实现，主要是完成了一个大的调用框架，具体的实现在各子过程的 ROS 封装类中。

6.2.1 源码相关文件

- 源码链接：

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

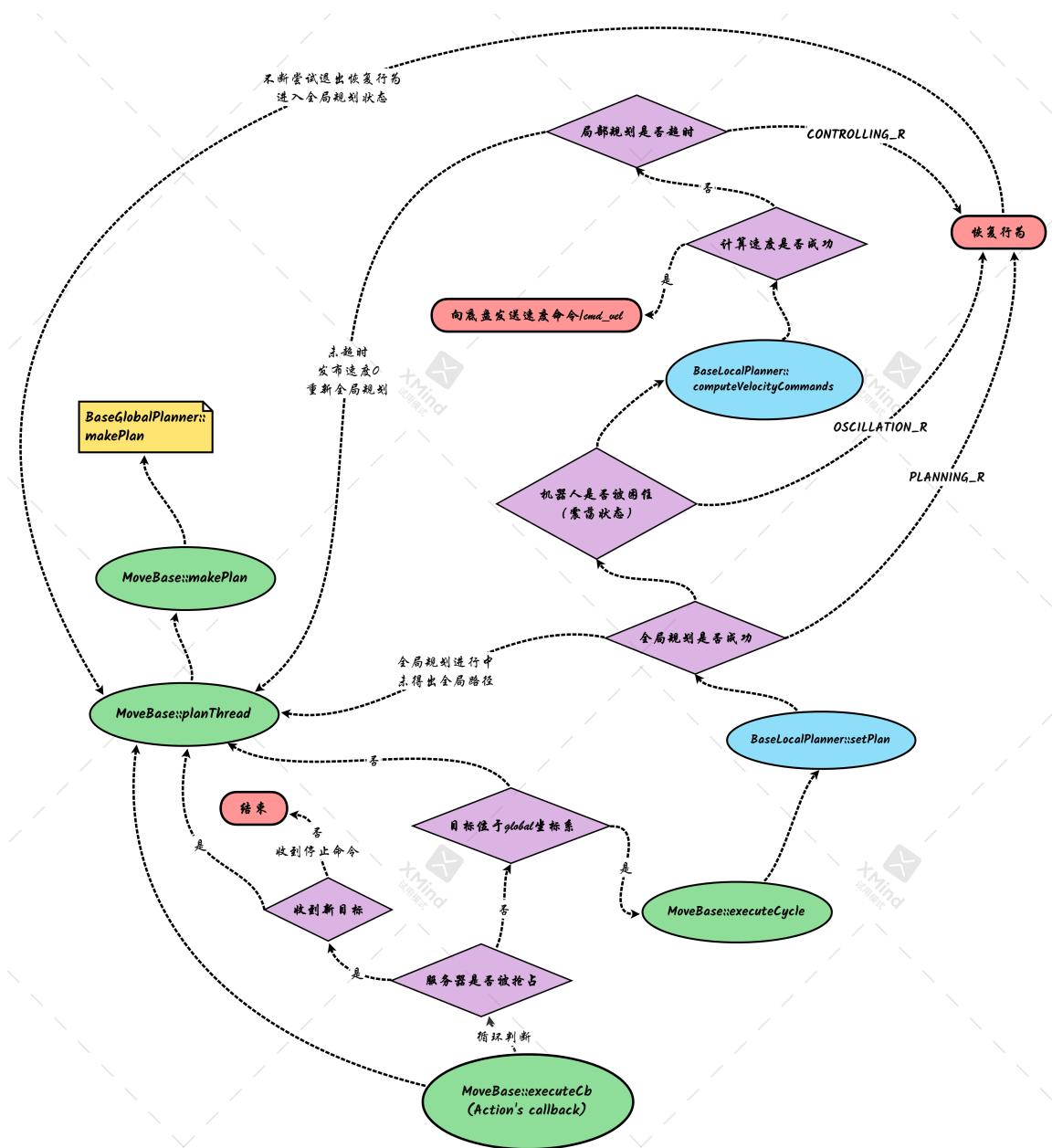
- 源码注释链接：

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

对应源码中的相关文件：

- move_base/src/move_base_node.cpp
- move_base/src/move_base.cpp

6.2.2 整体结构图



在 Movebase 主体中，各层地图的更新被启动，Action 的回调函数触发全局规划线程，若成功，则将全局规划结果传入局部规划器，循环进行局部规划，得到速度指令，控制机器人前进，直到到达目标。其间，需要判断机器人是否到达终点（若是则规划停止）、机器人是否状态异常如发生震荡行为（若是则进入恢复行为）、机器人是否超时（若是则停止规划发布零速，否则重新规划）等等。这个主体是一个大的调用框架，保证了运动规划的正常运行，具体算法在各子过程中分别实现。

6.2.3 move_base_node.cpp

move_base_node.cpp 文件中，完成了 *move_base_node* 节点的初始化、*MoveBase* 类的实例化。move_base 的主要工作：Action 服务的定义、全局规划器、局部规划器的调用都以类成员函数的形式定义在 *MoveBase* 类中。

之后，Action 开始进入监听状态，等待服务请求，然后进入回调函数 (*move_base.cpp* 中的 *MoveBase :: executeCb* 函数) 进行处理。

```
int main(int argc, char** argv){
    //初始化节点 move_base_node
    ros::init(argc, argv, "move_base_node");

    //tf监听器 (TransformListener) 会对监听到的tf数据进行10秒的缓存
    //buffer用来存储这10秒的tf数据
    tf2_ros::Buffer buffer(ros::Duration(10));
    tf2_ros::TransformListener tf(buffer);

    //实例化了MoveBase这个类 (move_base是namespace; MoveBase是类名)
    //Action服务的定义、全局规划器、局部规划器等都在这个类的成员函数中实现
    move_base::MoveBase move_base(buffer);

    //实例化之后，Action开始监听服务请求，并通过ros::spin()传递到Action的回调函数中进行处理
    ros::spin();

    return(0);
}
```

6.2.4 move_base.cpp

函数列表

核心函数：

- 构造函数——*MoveBase::MoveBase*
- 控制主体 (回调函数)——*MoveBase::executeCb*
- 全局规划线程——*MoveBase::planThread*
- 全局规划函数——*MoveBase::makePlan*
- 局部规划函数——*MoveBase::executeCycle*

- 析构函数——MoveBase:: MoveBase

其他函数：

- 动态加载配置参数——MoveBase::reconfigureCB
- 仿真时发布目标函数——MoveBase::goalCB
- 可以自动搜索终点附近可达到点的路径规划服务——MoveBase::planService
- 发布零速度——MoveBase::publishZeroVelocity
- 检查四元数合法性函数——MoveBase::isQuaternionValid
- 将目标点转换到全局坐标系下——MoveBase::goalToGlobalFrame
- 唤醒路径规划线程——MoveBase::wakePlanner
- 获取两点之间的水平路径——MoveBase::distance
- 加载恢复行为插件——MoveBase::loadRecoveryBehaviors
- 加载默认恢复行为插件——MoveBase::loadDefaultRecoveryBehaviors
- 路劲规划系统复位——MoveBase::resetState
- 获取在指定代价地图下的位姿——MoveBase::getRobotPose

MoveBase::MoveBase()

MoveBase 类的构造函数，获取服务器上的参数值，初始化规划的缓冲池，创建一些消息话题的收发端以及全局和本地规划器。

创建 Action 服务器：

```
//as_指向action服务器，当执行as_->start()时调用MoveBase::executeCb函数
as_ = new MoveBaseActionServer(ros::NodeHandle(), "move_base", boost::bind(&Move
Base::executeCb, this, _1), false);
```

初始化 plan 的缓冲池：

```
planner_plan_ = new std::vector<geometry_msgs::PoseStamped>();
latest_plan_ = new std::vector<geometry_msgs::PoseStamped>();
controller_plan_ = new std::vector<geometry_msgs::PoseStamped>();
```

从这里可以看出，plan 的结果其实就是一个 *geometry_msgs :: PoseStamped* 类型的一个队列。另外，*geometry_msgs :: PoseStamped* 是一个带有参考坐标帧和 timestamp 的 Pose，即用一个三维的点表示位置，一个四元数表示姿态。所以说 plan 的结果就是一个机器人的位姿队列。

初始化全局规划器和局部规划器的指针和各自的 costmap：

规划器用到的地图实质上是 Costmap2DROS 类的实例，这个类是 ROS 对 costmap 的封装。

```

//创建全局规划器的代价地图
planner_costmap_ros_ = new costmap_2d::Costmap2DROS("global_costmap", tf_);
planner_costmap_ros_->pause();

//初始化全局规划器， planner_指针
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);
}

//创建本地规划器的代价地图
controller_costmap_ros_ = new costmap_2d::Costmap2DROS("local_costmap", tf_);
controller_costmap_ros_->pause();

//创建本地规划器， tc_指针
try {
    tc_ = bsp_loader_.createInstance(local_planner);
    ROS_INFO("Created local_planner %s", local_planner.c_str());
    tc_->initialize(bsp_loader_.getName(local_planner), &tf_, controller_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", local_planner.c_str(), ex.what());
    exit(1);
}

```

开始动态更新代价地图：

```

planner_costmap_ros_->start();
controller_costmap_ros_->start();

```

启动 Action 服务器：

```
as_->start();
```

MoveBase::executeCb()

executeCb 是 *Action* 的回调函数，它是 *MoveBase* 控制流的主体，调用了 *MoveBase* 内另外几个作为子部分的重要成员函数，先后完成了全局规划和局部规划。

当有了一个 *goal* 之后，启动全局规划：

```

boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);
//用接收到的目标goal来更新全局规划目标（全局变量），它在planThread中会被用来做全

```

局规划的当前目标

```
planner_goal_ = goal;
runPlanner_ = true;

//开始全局规划并于此处阻塞
//在这里调用notify会直接启动全局规划器线程，进行全局路径规划
//全局规划器线程绑定的函数plannerThread()里有planner_cond_对象的wait函数
planner_cond_.notify_one();
lock.unlock();
```

全局规划完成后，循环调用 executeCycle 函数来控制机器人进行局部规划。

在循环中如果发现，Action 服务器被抢占，则分两种可能进行讨论：

- 如果是收到新目标，那么放弃当前目标，使用新目标，重复前面对目标进行的相关处理，并重新全局规划；
- 如果是收到取消行动命令，直接结束返回。

如果服务器未被抢占，或被抢占的 if 结构已执行完毕，接下来开始局部规划。代码如下：

```
if(as_->isPreemptRequested()){
    //如果是 局部规划进行过程中收到新的目标
    if(as_->isNewGoalAvailable()){
        //如果获得了新目标，接收并存储新目标，并将上述过程重新进行一遍
        move_base_msgs::MoveBaseGoal new_goal = *as_->acceptNewGoal();

        .....
    }
    //如果是 收到取消行动的命令
    else {
        //被取消了 则重置服务器状态
        resetState();

        //Action服务器清除相关内容，并调用setPreempted() 函数
        ROS_DEBUG_NAMED("move_base","Move base preempting the current goal");
        as_->setPreempted();

        return;
    }
}

//检查目标是否被转换到全局坐标系 (/map) 下，如果没有转换过来 则进行转换 然后再次执行上面的操作
if(goal.header.frame_id != planner_costmap_ros_->getGlobalFrameID()){
    goal = goalToGlobalFrame(goal);
    .....
}

ros::WallTime start = ros::WallTime::now();
```

```
//调用executeCycle函数进行局部规划，传入目标和全局规划路线
bool done = executeCycle(goal, global_plan);
```

MoveBase::planThread()

planThread() 的核心是调用 *makePlan* 函数，该函数中实际进行全局规划。当 *executeCB* 函数中唤醒 *planThread*（调用 *planner_cond_.notify_one()*），并将标志位 *runPlanner_* 设置为真，*planThread()* 跳出内部的循环，继续进行下面的全局规划部分。

循环等待 *executeCB* 函数中的唤醒操作：

```
while(wait_for_wake || !runPlanner_){
    //if we should not be running the planner then suspend this thread
    ROS_DEBUG_NAMED("move_base_plan_thread","Planner thread is suspending");
    //调用wait函数时，函数会自动调用lock.unlock()释放锁，使得其他被阻塞在锁竞争上的线程得以继续执行
    planner_cond_.wait(lock);
    wait_for_wake = false;
}
```

被唤醒后，调用 *makePlan* 函数，进行全局规划：

```
ros::Time start_time = ros::Time::now();

//把全局中被更新的全局目标planner_goal存储为临时目标
geometry_msgs::PoseStamped temp_goal = planner_goal_;
lock.unlock();
ROS_DEBUG_NAMED("move_base_plan_thread","Planning...");

//run planner
//全局规划初始化 清空
planner_plan_->clear();
//调用MoveBase类的makePlan函数，如果成功为临时目标制定全局规划planner_plan_，则返回true
bool gotPlan = n.ok() && makePlan(temp_goal, *planner_plan_);
```

若全局规划成功，则交换 *planner_plan_* 和 *latest_plan* 的值，即令 *latest_plan* 中存储的是本次全局规划的结果（最新），*planner_plan_* 中存储的是上次全局规划的结果（次新）。设置标志位 *new_global_plan_ = true*，表示得到了新的全局规划路线，并设置 *Movebase* 状态标志位 *state_* 为 *CONTROLLING*，即全局规划完成，开始进行局部控制。

如果全局规划失败，*MoveBase* 还在 *PLANNING* 状态，即机器人没有移动，则进入自转模式。

```
if(gotPlan){
    ROS_DEBUG_NAMED("move_base_plan_thread","Got Plan with %zu points!", planner_plan_->size());
    std::vector<geometry_msgs::PoseStamped>* temp_plan = planner_plan_;
    lock.lock();
    planner_plan_ = latest_plan_;
```

```

latest_plan_ = temp_plan;

last_valid_plan_ = ros::Time::now();
planning_retries_ = 0;
new_global_plan_ = true;

ROS_DEBUG_NAMED("move_base_plan_thread", "Generated a plan from the base_global_planner");

//确保只有在我们还没到达目标时才启动controller以局部规划
//如果runPlanner_在调用此函数时被置为真，将MoveBase状态设置为CONTROLLING（局部规划中）
if(runPlanner_)
    state_ = CONTROLLING;
//planner_frequency_ <= 0则全局规划就是触发机制的，不会自动循环
if(planner_frequency_ <= 0)
    runPlanner_ = false;
lock.unlock();

}

//如果全局规划失败并且MoveBase还在PLANNING状态，即机器人没有移动，则进入自转模式
else if(state_==PLANNING){
    ROS_DEBUG_NAMED("move_base_plan_thread", "No Plan...");
    ros::Time attempt_end = last_valid_plan_ + ros::Duration(planner_patience_);

    //检查时间和次数是否超过限制，若其中一项不满足限制，停止全局规划
    lock.lock();
    planning_retries_++;
    if(runPlanner_ && (ros::Time::now() > attempt_end || planning_retries_ > uint32_t(max_planning_retries_))){
        state_ = CLEARING;
        runPlanner_ = false;
        publishZeroVelocity();
        recovery_trigger_ = PLANNING_R;
    }
    lock.unlock();
}

```

MoveBase::makePlan()

在该函数中正式执行全局路径规划算法，结果保存在 `planner_` 中，但是算法实现的不在这里是同名函数 `BaseGlobalPlanner :: makePlan` 实现的。

先进行一些预备工作，如检查全局代价地图、起始位姿，然后将起始位姿的数据格式做转换。接下来是实际进行全局规划的函数，调用全局规划器的 `makePlan` 函数 `planner_->makePlan(start, goal, plan)`，传入机器人当前位姿和目标，得到 `plan`，若规划失败或得到的 `plan` 为空，返回 `false`，否则返回 `true`。

```

const geometry_msgs::PoseStamped& start = global_pose;
if(!planner_->makePlan(start, goal, plan) || plan.empty()){

```

```

ROS_DEBUG_NAMED("move_base", "Failed to find a plan to point (%.2f, %.2f)", goal.pose.position.x
, goal.pose.position.y);
return false;
}

```

MoveBase::executeCycle()

executeCycle 函数的作用是进行局部规划。

通过标志位 *new_global_plan_* 判断全局规划是否得到了新的路线。如果获得了新的路线，则通过指针交换，将 *latest_plan_*（最新的全局规划结果）的值传递给 *controller_plan_* 即局部规划使用，然后将上一次的局部规划路线传递给 *latest_plan*。进而在实例 *tc_* 上调用局部规划器 *BaseLocalPlanner* 的类函数 *setPlan()*，把全局规划的结果传递给局部规划器。代码如下：

```

if(new_global_plan_){
    new_global_plan_ = false;
    ROS_DEBUG_NAMED("move_base", "Got a new plan...swap pointers");

    std::vector<geometry_msgs::PoseStamped>* temp_plan = controller_plan_;
    boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);
    controller_plan_ = latest_plan_;
    latest_plan_ = temp_plan;
    lock.unlock();
    ROS_DEBUG_NAMED("move_base", "pointers swapped!");

    //在实例tc_上调用setPlan() 把全局规划的结果传递给局部规划器，如果传递失败，退出并返回。
    if(!tc_->setPlan(*controller_plan_)){
        //ABORT and SHUTDOWN COSTMAPS
        ROS_ERROR("Failed to pass global plan to the controller, aborting.");
        resetState();

        lock.lock();
        runPlanner_ = false;
        lock.unlock();

        as_->setAborted(move_base_msgs::MoveBaseResult(), "Failed to pass global plan to the controller.");
        return true;
    }
    .....
}

```

接下来对 *MoveBase* 状态进行判断，可能会有以下几种结果：

- *PLANNING*: 全局规划还没完成，还没得到一个全局路线，那么唤醒一个全局规划线程去制定全局路线；
- *CONTROLLING*: 全局规划成功，得到全局路线，这里进行真正的局部规划；

- *CLEARING*: 全局规划失败，进入恢复行为。

接下来分别解析这三种情况下的处理操作，若 *status_ = PLANNING*:

```
//PLANNING: 全局规划还没完成，还没得到一个全局路线，那么唤醒一个全局规划线程去制定全局路线
case PLANNING:
{
    boost::recursive_mutex::scoped_lock lock(planner_mutex_);
    runPlanner_ = true;
    planner_cond_.notify_one();
}
ROS_DEBUG_NAMED("move_base", "Waiting for plan, in the planning state.");
break;
```

若 *status_ = CONTROLLING*:

这里进行真正的局部规划，再次分为几种情况讨论：

- 这里进行真正的局部规划；
- 如果没到终点，检查机器人是否被困住，如果是，则进入恢复行为；
- 如果没到终点，且状态正常，调用局部规划器实例 *tc_->computeVelocityCommands(cmd_vel)* 函数，它根据结合传入的全局规划路线和其他因素计算得出局部规划结果，即速度指令，存放 在 *cmd_vel* 中，将其发布，控制机器人运行。

//CONTROLLING: 全局规划成功，得到全局路线，这里进行真正的局部规划：

```
case CONTROLLING:
ROS_DEBUG_NAMED("move_base", "In controlling state.");

//如果已经位于终点，结束局部规划；
if(tc_->isGoalReached()){
    ROS_DEBUG_NAMED("move_base", "Goal reached!");
    resetState();

    boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);
    runPlanner_ = false;
    lock.unlock();

    as_->setSucceeded(move_base_msgs::MoveBaseResult(), "Goal reached.");
    return true;
}

//如果机器人被困住，则进入恢复行为
if(oscillation_timeout_ > 0.0 && last_oscillation_reset_ + ros::Duration(oscillation_timeout_) <
ros::Time::now()){
    publishZeroVelocity();
    state_ = CLEARING;
```

```

        recovery_trigger_ = OSCILLATION_R;
    }

{ //如果没到终点 且状态正常
    boost::unique_lock<costmap_2d::Costmap2D::mutex_t> lock(*(*controller_costmap_ros_->getCostma
    p()->getMutex()));

    //局部规划器实例tc_被传入了全局规划后，调用computeVelocityCommands函数计算速度存储在cmd_vel中
    if(tc_->computeVelocityCommands(cmd_vel)){
        ROS_DEBUG_NAMED( "move_base", "Got a valid command from the local planner: %.3lf, %.3lf
        , %.3lf", cmd_vel.linear.x, cmd_vel.linear.y, cmd_vel.angular.z );
        //若成功计算速度，则说明本次局部规划成功， 将上一次有效局部控制的时间设为当前时间
        last_valid_control_ = ros::Time::now();
        //向底盘发送速度控制消息，一个循环只发一次速度命令
        vel_pub_.publish(cmd_vel);
        //如果恢复行为触发器值是局部规划失败，把索引置0
        if(recovery_trigger_ == CONTROLLING_R)
            recovery_index_ = 0;
    }
    //若速度计算失败
    else {
        ROS_DEBUG_NAMED("move_base", "The local planner could not find a valid plan.");
        //计算局部规划用时限制
        ros::Time attempt_end = last_valid_control_ + ros::Duration(controller_patience_);

        //若局部规划用时超过限制
        if(ros::Time::now() > attempt_end){
            //发布0速度，进入恢复行为，触发器置为局部规划失败
            publishZeroVelocity();
            state_ = CLEARING;
            recovery_trigger_ = CONTROLLING_R;
        }
        //若局部规划用时没超过限制 那就是找不到有效的控制路线（局部规划路线），则再次回到全局规划
        else{
            last_valid_plan_ = ros::Time::now();
            planning_retries_ = 0;
            state_ = PLANNING;
            publishZeroVelocity();

            //激活全局规划线程
            boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);
            runPlanner_ = true;
            planner_cond_.notify_one();
            lock.unlock();
        }
    }
}
}

```

```
break;
```

若 *status_* = CLEARING:

```
case CLEARING:
```

```
    ROS_DEBUG_NAMED("move_base", "In clearing/recovery state");
```

```
    //如果允许使用恢复行为，且恢复行为索引值小于恢复行为数组的大小
```

```
    if(recovery_behavior_enabled_ && recovery_index_ < recovery_behaviors_.size()) {
```

```
        ROS_DEBUG_NAMED("move_base_recovery", "Executing behavior %u of %zu", recovery_index_+1, recovery_behaviors_.size());
```

```
        move_base_msgs::RecoveryStatus msg;
```

```
        msg.pose_stamped = current_position;
```

```
        msg.current_recovery_number = recovery_index_;
```

```
        msg.total_number_of_recoveries = recovery_behaviors_.size();
```

```
        msg.recovery_behavior_name = recovery_behavior_names_[recovery_index_];
```

```
        //发布恢复行为的相关信息
```

```
        recovery_status_pub_.publish(msg);
```

```
        //开始恢复行为，在executeCycle的循环中一次次迭代恢复行为
```

```
        recovery_behaviors_[recovery_index_]->runBehavior();
```

```
        last_oscillation_reset_ = ros::Time::now();
```

```
        //在恢复行为时不断尝试切换进入PLANNING模式，检查恢复行为是否产生了效果（使机器人脱困）
```

```
        ROS_DEBUG_NAMED("move_base_recovery", "Going back to planning state");
```

```
        last_valid_plan_ = ros::Time::now();
```

```
        planning_retries_ = 0;
```

```
        state_ = PLANNING;
```

```
        recovery_index_++;
```

```
}
```

```
//若没有可用的恢复行为或所有恢复行为均无效
```

```
else{
```

```
    //关闭全局规划器并反馈信息
```

```
    if(recovery_trigger_ == CONTROLLING_R){
```

```
        //报错信息
```

```
}
```

```
    //找不到可行的全局规划策略
```

```
    else if(recovery_trigger_ == PLANNING_R){
```

```
        //报错信息
```

```
}
```

```
    //机器人无法摆脱振荡状态
```

```
    else if(recovery_trigger_ == OSCILLATION_R){
```

```
        //报错信息
```

```
}
```

```

    resetState();
    return true;
}
break;

```

MoveBase::reconfigureCB

该函数自动从参数服务器上读取修改的参数，并且进行一些参数变换的中间过渡处理（如 `base_global_planner` 规划插件变化）。

支持在参数服务器上直接设置恢复默认参数操作：

```

//如果在参数服务器上设置了恢复默认参数
if(config.restore_defaults) {
    config = default_config_;
    config.restore_defaults = false;
}

```

如果在中途切换全局规划器 `base_global_planner` 插件，则需要进行一些过渡过程（切换局部规划器插件同理）：

```

//全局规划器改变
if(config.base_global_planner != last_config_.base_global_planner) {
    boost::shared_ptr<nav_core::BaseGlobalPlanner> old_planner = planner_;
    ROS_INFO("Loading global planner %s", config.base_global_planner.c_str());
    try {
        //initialize the global planner
        planner_ = bgp_loader_.createInstance(config.base_global_planner);

        //等待当前的规划器完成本次的规划
        boost::unique_lock<boost::recursive_mutex> lock(planner_mutex_);

        //先清除规划器中的现有数据，并复位系统，再初始化新的规划器
        planner_plan_->clear();
        latest_plan_->clear();
        controller_plan_->clear();
        resetState();

        planner_->initialize(bgp_loader_.getName(config.base_global_planner), planner_costmap_ros_);

        lock.unlock();
    } 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", config.base_global_planner.c_str(), ex.what());
        //如果切换过程中出现问题了，就退回之前的规划器
        planner_ = old_planner;
        config.base_global_planner = last_config_.base_global_planner;
    }
}

```

```

    }
}
}
```

MoveBase::planService

提供给使用者一种指定 goal 的路径规划服务（在这种服务下如果无法达到 goal 会自动搜索终点容许误差范围内的可达到点作为终点）。

一般需要使用者没有指定一个机器人的起始位姿，如果没有指定（空的 frame-id 表示没有指定），则使用机器人的当前位姿：

```

if(req.start.header.frame_id.empty()){
    geometry_msgs::PoseStamped global_pose;
    if(!getRobotPose(global_pose, planner_costmap_ros_)){
        ROS_ERROR("move_base cannot make a plan for you because it could not get the start pose of the robot");
        return false;
    }
    start = global_pose;
}
//若指定了起始位姿
else{
    start = req.start;
}
```

首先会尝试直接以使用者指定的目标点为终点进行路径规划，如果规划失败（*global_plan* 为空），则在规定的误差范围内向外寻找可行的目标位置，算法会自动在允许的范围内逐渐扩大搜索的半径。并且当搜索到一个在目标附近的可达到的终点之后，可以选择（使用参数 *make_plan_add_unreachable_goal_* 进行配置）将原本不可达到的目标点也放到 *global_plan* 队列的末尾，因为有可能当我们达到其附近的可达到点之后，原本不可达到的目标点也可以达到了。

实现算法如下：

```

//first try to make a plan to the exact desired goal
std::vector<geometry_msgs::PoseStamped> global_plan;
if(!planner_->makePlan(start, req.goal, global_plan) || global_plan.empty()){
    ROS_DEBUG_NAMED("move_base", "Failed to find a plan to exact goal of (%.2f, %.2f), searching for a feasible goal within tolerance", req.goal.pose.position.x, req.goal.pose.position.y);

    //在规定的误差范围内向外寻找可行的目标位置
    geometry_msgs::PoseStamped p;
    p = req.goal;
    bool found_legal = false;
    float resolution = planner_costmap_ros_->getCostmap()->getResolution();
    float search_increment = resolution*3.0;
    if(req.tolerance > 0.0 && req.tolerance < search_increment)
        search_increment = req.tolerance;
    //在允许的范围内逐渐扩大搜索的半径
    for(float max_offset = search_increment; max_offset <= req.tolerance && !found_legal; max_offset += search_incr
ement){
```

MoveBase::loadRecoveryBehaviors

加载恢复行为插件可以自行指定恢复行为列表，如果指定的插件加载过程中没有问题，则该函数会将插件依次添加到恢复行为列表中，当机器人进入恢复行为后，就会依次轮询列表中的恢复行为，帮助机器人脱困。如果没有指定或者指定有误，则加载默认的恢复行为。

MoveBase::loadDefaultRecoveryBehaviors

加载默认的恢复行为插件，默认的恢复行为如下：

- #### - 保守的空间清理操作——*conservative reset*

- 自转——*rotate_recovery*
- 激进的空间清理操作——*aggressive_reset*
- 再次自转——*rotate_recovery*

6.3 navfn 源码学习

Movebase 使用的全局规划器默认为 *NavFn*, 默认使用 *Dijkstra* 算法, 在地图上的起始点和目标点间规划出一条最优路径, 供局部规划器具体导航使用。*NavFn* 的源码中实际上有基于 *Dijkstra* 的 *A** 规划算法的函数, 但早期 *NavFn* 包中的 *A** 有 bug, 没有处理, 后来发布了 *global_planner*, 修改好了 *A** 的部分。所以一般默认 *Dijkstra* 算法在 *NavFn* 中, *A** 算法在 *global_planner* 中。

6.3.1 Dijkstra 算法原理

设 $G = (V, E)$ 是一个带权有向图, 把图中顶点集合 V 分成两组, 第一组为已求出最短路径的顶点集合 (用 S 表示, 初始时 S 中只有一个源点, 以后每求得一条最短路径, 就将加入到集合 S 中, 直到全部顶点都加入到 S 中, 算法就结束了), 第二组为其余未确定最短路径的顶点集合 (用 U 表示), 按最短路径长度的递增次序依次把第二组的顶点加入 S 中。在加入的过程中, 总保持从源点 v 到 S 中各顶点的最短路径长度不大于从源点 v 到 U 中任何顶点的最短路径长度。此外, 每个顶点对应一个距离, S 中的顶点的距离就是从 v 到此顶点的最短路径长度, U 中的顶点的距离, 是从 v 到此顶点只包括 S 中的顶点为中间顶点的当前最短路径长度。

具体理解过程参照博客:

<https://www.cnblogs.com/yutian-blogs/p/15605767.html>

6.3.2 源码相关文件

- 源码链接:

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

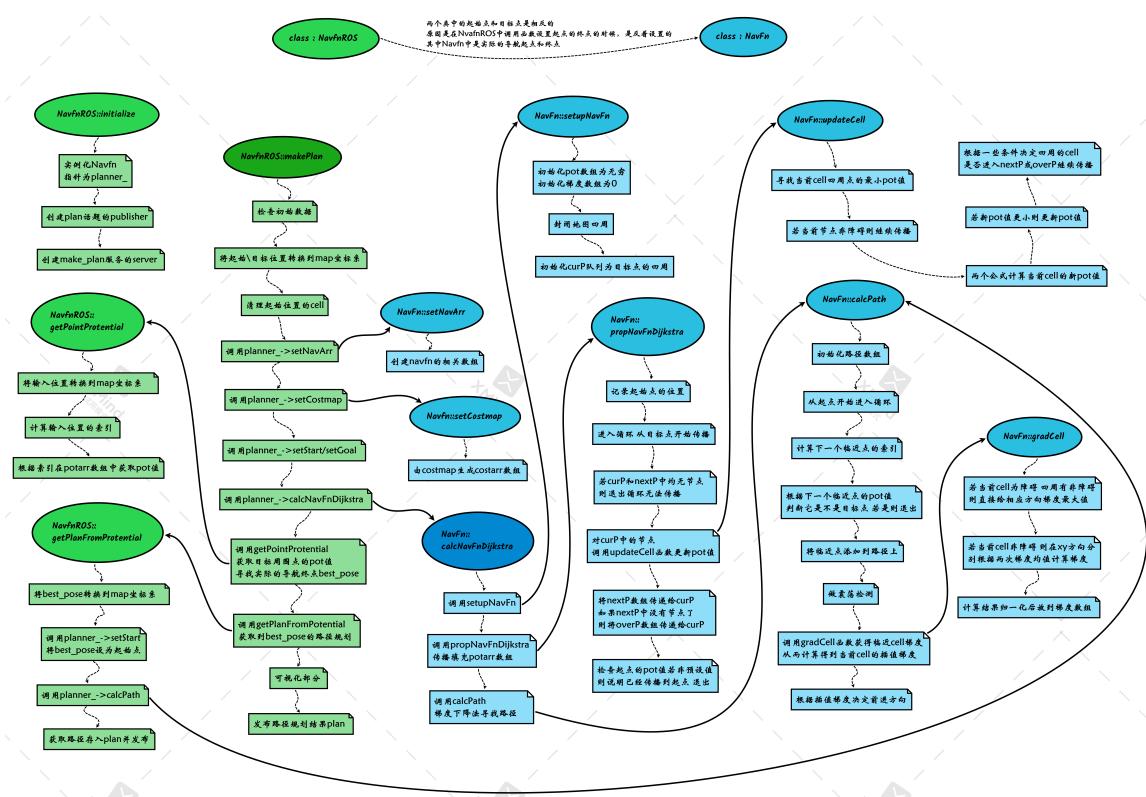
- 源码注释链接:

<https://github.com/ros-planning/navigation/tree/noetic-devel/navfn>

对应源码中的相关文件:

- navfn/src/navfn_ros.cpp
- navfn/src/navfn.cpp

6.3.3 整体结构图



Movebase 使用的全局规划器默认为 NavFn， 默认使用 Dijkstra 算法，在地图上的起始点和目标点间规划出一条最优路径，供局部规划器具体导航使用。NavFn 的源码中实际上是有 A* 规划算法的函数的，但早期 NavFn 包中的 A* 有 bug，没有处理，后来发布了 global_planner，在 global_planner 包中修改好了 A* 的部分。

navfn_ros.cpp 中定义了 NavfnROS 类，navfn.cpp 中定义了 NavFn 类，ROS Navigation 整个包的一个命名规则是，带有 ROS 后缀的类完成的是该子过程与整体和其他过程的衔接框架和数据流通，不带 ROS 后缀的类中完成该部分的实际工作，并作为带有 ROS 后缀的类的成员。

6.3.4 navfn_ros.cpp

函数列表

核心函数：

- **初始化函数**——NavfnROS::initialize
- **核心路径规划函数**——NavfnROS::makePlan
- **获取单点 Pot 值**——NavfnROS::getPointPotential
- **获取路径规划结果**——NavfnROS::getPlanFromPotential

NavfnROS::initialize

主要进行参数的初始化以及成员类 NavFn 的初始化。

其中对于成员类 NavFn 的初始化代码如下：

```
//对成员类NavFn初始化，这个类将完成全局规划实际计算
//planner_指向NavFn类实例，传入参数为 costmap_ 地图大小
planner_ = boost::shared_ptr<NavFn>(new NavFn(costmap_->getSizeInCellsX(), costmap_->getSizeInCellsY()));
```

NavfnROS::makePlan

makePlan 是在 *Movebase* 中对全局规划器调用的函数，它是 *NavfnROS* 类的重点函数，负责调用包括 *Navfn* 类成员在内的函数完成实际计算，控制着全局规划的整个流程。它的输入中最重要的是当前和目标的位置。

准备工作：规划前先清理 *plan*，等待 *tf*，存储当前起点位置并转换到地图坐标系，并将全局 *costmap* 上起点的 *cell* 设置为 *FREE_SPACE*。

接下来，调用 *NavFn* 类的 *setNavArr* 函数：给定地图的大小，创建 *costarr* 数组、*potarr* 数组以及 *x* 和 *y* 向的梯度数组），这三个数组构成 *NavFn* 类用 *Dijkstra* 计算的主干。

其各自的作用为：

- *costarr* 数组：记录全局 *costmap* 信息
- *potarr* 数组：储存各 *cell* 的 *Potential* 值
- *x* 和 *y* 向的梯度数组 (*gradx* 和 *grady*)：记录各个 *cell* 的梯度值用于生成路径

另外，调用 *setCostmap* 函数将 *costmap* 翻译成 *costarr*，根据代价值 *cost* 生成无向权重图，该部分代码如下：

```
planner_->setNavArr(costmap_->getSizeInCellsX(), costmap_->getSizeInCellsY());
planner_->setCostmap(costmap_->getCharMap(), true, allow_unknown_);
```

接下来将设置 *NavFn* 类的起点和目标位置，这里要注意的是设置 *NavFn* 类的终点和起点时，起点和终点是反着设置的，导致 *Navfn* 类和 *NavfnROS* 类的起点和终点是反着的。（这应该不是失误，而是 *Navfn* 类在进行具体传播规划过程中反着来更方便理解）

然后调用该类的 *calcNavFnDijkstra* 函数，这个函数可以完成全局路径的计算，这部分代码如下：

```
//设置NavFn类的终点和起点
planner_->setStart(map_goal);
planner_->setGoal(map_start);

planner_->calcNavFnDijkstra(true);
```

接下来，在目标位置附近 $2 * tolerance$ 的矩形范围内，寻找与目标位置最近的、且不是障碍物的 *cell*，作为全局路径实际的终点，这里调用了类内 *getPointPotential* 函数，目的是获取单点 *Potential* 值，与 *DBL_MAX* 比较，确定是否是障碍物。其中，*resolution* 是搜索的分辨率（也就是代价地图的分辨率）。代码如下：

```

double resolution = costmap_->getResolution();
geometry_msgs::PoseStamped p, best_pose;
p = goal;

bool found_legal = false;
double best_sdist = DBL_MAX;

p.pose.position.y = goal.pose.position.y - tolerance;

while(p.pose.position.y <= goal.pose.position.y + tolerance){
    p.pose.position.x = goal.pose.position.x - tolerance;
    while(p.pose.position.x <= goal.pose.position.x + tolerance){
        double potential = getPointPotential(p.pose.position);
        double sdist = sq_distance(p, goal);
        if(potential < POT_HIGH && sdist < best_sdist){
            best_sdist = sdist;
            best_pose = p;
            found_legal = true;
        }
        p.pose.position.x += resolution;
    }
    p.pose.position.y += resolution;
}

```

若成功找到实际终点 *best_pose*，调用类内 *getPlanFromPotential* 函数，将 *best_pose* 传递给 *NavFn*，获得最终 *Plan* 并发布，这是最终正式的路径规划，代码如下：

```

//若成功找到实际终点best_pose
if(found_legal){
    if(getPlanFromPotential(best_pose, plan)){
        //make sure the goal we push on has the same timestamp as the rest of the plan
        geometry_msgs::PoseStamped goal_copy = best_pose;
        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 should
n't happen.");
}
}
}

```

后面部分是 *potarr* 数组的发布，与主体关系不大，略过。

NavfnROS::getPointPotential

该函数在 *makePlan* 中被调用，主要工作是获取 *potarr* 数组记录的对应 *cell* 的 *Potential* 值。它首先将给定点转换到 *map* 坐标系下，然后计算其对应的 *potarr* 数组的索引，最后取出这个 *cell* 的 *pot* 值。代码如下：

```

unsigned int mx, my;
if(!costmap_->worldToMap(world_point.x, world_point.y, mx, my))
    return DBL_MAX;

unsigned int index = my * planner_->nx + mx;
return planner_->potarr[index];

```

NavfnROS::getPlanFromPotential

该函数在 *makePlan* 中被调用，主要工作是调用了 *NavFn* 类的一些函数，设置目标、获取规划结果。

该函数中，将 *makePlan* 末尾处找到的 *goal* 附近的 *best_pose* 坐标转换到地图坐标系，并通过调用的 *setStart* 函数传递（注意这里设置起点的时候也是反的哦），作为路径的实际终点，再调用 *calcPath* 函数，完成路径计算。代码如下：

```

//储存makePlan末尾处找到的goal附近的best_pose的坐标
double wx = goal.pose.position.x;
double wy = goal.pose.position.y;

//best_pose坐标转换到地图坐标系
unsigned int mx, my;
if(!costmap_->worldToMap(wx, wy, mx, my)){
    ROS_WARN_THROTTLE(.....);
    return false;
}

int map_goal[2];
map_goal[0] = mx;

```

```
map_goal[1] = my;

//将best_pose设置为路径的实际终点
planner_->setStart(map_goal);
//调用NavFn类calcPath函数，完成路径计算
planner_->calcPath(costmap_->getSizeInCellsX() * 4);
```

接下来，获取规划结果的坐标，填充 *plan* 之后将其发布：

```
//获取规划结果的坐标，填充plan之后将其发布
float *x = planner_->getPathX();
float *y = planner_->getPathY();
int len = planner_->getPathLen();
ros::Time plan_time = ros::Time::now();

for(int i = len - 1; i >= 0; --i){
    //convert the plan to world coordinates
    double world_x, world_y;
    mapToWorld(x[i], y[i], world_x, world_y);

    //只规划了位置，没有姿态
    geometry_msgs::PoseStamped pose;
    pose.header.stamp = plan_time;
    pose.header.frame_id = global_frame_;
    pose.pose.position.x = world_x;
    pose.pose.position.y = world_y;
    pose.pose.position.z = 0.0;
    pose.pose.orientation.x = 0.0;
    pose.pose.orientation.y = 0.0;
    pose.pose.orientation.z = 0.0;
    pose.pose.orientation.w = 1.0;
    plan.push_back(pose);
}

//publish the plan for visualization purposes
publishPlan(plan, 0.0, 1.0, 0.0, 0.0);
```

6.3.5 navfn.cpp

函数列表

核心函数:

- 由 costmap 生成 costarr 数组——NavFn::setCostmap
- 调用子函数完成 Dijkstra 路径计算总流程——NavFn::calcNavFnDijkstra
- 处理 costarr 数组并初始化 potarr、gradx、grady 数组——NavFn::setupNavFn
- 传播填充 potarr 数组——NavFn::propNavFnDijkstra
- 更新单个 cell 的 pot 值——NavFn::updateCell
- 梯度下降法生成路径——NavFn::calcPath
- 计算 cell 处的梯度值——NavFn::gradCell

其他函数:

NavFn::setCostmap

该函数将 *costmap* 翻译成 *costarr*, 根据代价值 *cost* 生成无向权重图。

在这个过程中, 将 *cell* 分为四种情况进行处理:

- 若当前 *cell* 在 *costmap* 上的值 $< COST_OB\$ROS(253)$, 即非致命障碍物 (障碍物附近), 重新将其赋值为 $COST_NEUTRAL(50) +$ 当前 *cell* 在 *costmap* 上的值 $\times 0.8$ 。作用为: 将原本 [0 253] 范围的数值变换到 [50 253] 范围;
- 若当前 *cell* 在 *costmap* 上的值 $= COST_OBS(254)$, 即致命障碍物 (障碍物本身), 值仍为 254;
- 若当前 *cell* 在 *costmap* 上的值 $= COST_UNKNOWN_ROS(255)$, 即未知区域, 赋值为 253。

该部分代码如下:

```
if (isROS){
    //在地图的长宽范围内进行迭代
    for (int i=0; i<ny; i++){
        //k值记录二重迭代的次数
        int k = i*nx;
        for (int j=0; j<nx; j++, k++, cmap++, cm++){
            if (cmap > ny*nx)
                break;
            if (cm > nx)
                break;
            if (costmap[i][j] < COST_OB$ROS)
                costarr[k][j] = COST_NEUTRAL + costmap[i][j] * 0.8;
            else if (costmap[i][j] == COST_OBS)
                costarr[k][j] = COST_OBS;
            else if (costmap[i][j] == COST_UNKNOWN_ROS)
                costarr[k][j] = COST_UNKNOWN;
        }
    }
}
```

```

//最小权重值为COST_NEUTRAL=50(无障碍物的free栅格)
//最大权重值为COST_OBS=254(致命障碍被禁止的栅格)
//次大权重值为COST_OBS_ROS=253(膨胀型障碍的栅格)
//未知权重值未COST_UNKNOWN_ROS=255(未知权重的栅格)

*cm = COST_OBS;

int v = *cmap;

//若当前cell在costmap上的值 < COST_OBS_ROS(253), 即非膨胀型障碍
if (v < COST_OBS_ROS){

    //重新将其赋值为COST_NEUTRAL(50)+当前cell在costmap上的值×比例0.8 (最高253)
    //将原本[0~253]范围的数值变换到[50~253]范围
    v = COST_NEUTRAL + COST_FACTOR*v;

    //数值限制 防止将非膨胀型障碍赋值大于253
    if (v >= COST_OBS)
        v = COST_OBS-1;

    //赋值给当前全局规划要使用的地图costarr
    *cm = v;
}

//若当前cell的值为COST_UNKNOWN_ROS(255), 未知区域
else if(v == COST_UNKNOWN_ROS && allow_unknown){

    //统一设置为253 (膨胀型障碍)
    v = COST_OBS-1;

    *cm = v;
}

}

}

}

}

```

如果地图不是 *ROSmap*, 可能是 *PGM* 地图, 同样进行类似的翻译。

NavFn::setupNavFn

该函数对翻译生成的 *costarr* 数组进行了边际设置等处理, 并初始化了 *potarr* 数组和梯度数组 *gradx*、*grady*。初始化 *potarr* 矩阵元素全部为最大值 *POT_HIGH*, 并初始化梯度表初始值全部为 0.0, 代码如下:

```

for (int i=0; i<ns; i++){

    //将pot数组初始化为最大值, 默认起点到所有点的行走代价都为最大
    potarr[i] = POT_HIGH;

    //这是什么情况使用的?
    if (!keepit)

        costarr[i] = COST_NEUTRAL;
}

```

```
//初始化x,y方向的梯度表
gradx[i] = grady[i] = 0.0;
}
```

接下来设置 *costarr* 的四条边的 *cell* 的值为 *COST_OBS*(致命层 254)，封闭地图四周，以防产生边界以外的轨迹，代码如下：

```
//设置costarr的四条边的cell的值为COST_OBS(致命层254)，封闭地图四周，以防产生边界以外的轨迹
COSTTYPE *pc;
//costarr第一行全部设置为COST_OBS
pc = costarr;
for (int i=0; i<nx; i++)
    *pc++ = COST_OBS;
//costarr最后一行全部设置为COST_OBS
pc = costarr + (ny-1)*nx;
for (int i=0; i<nx; i++)
    *pc++ = COST_OBS;
//costarr第一列全部设置为COST_OBS
pc = costarr;
for (int i=0; i<ny; i++, pc+=nx)
    *pc = COST_OBS;
//costarr最后一列全部设置为COST_OBS
pc = costarr + nx - 1;
for (int i=0; i<ny; i++, pc+=nx)
    *pc = COST_OBS;
```

接下来，初始化一些用于迭代更新 *potarr* 的数据，并初始化 *pending* 数组为全 0，设置所有的 *cell* 状态都为非代办状态，其中：

- *curP* 记录当前正要访问的栅格
- *nextP* 记录即将要访问的栅格中优先级较高的部分
- *overP* 记录即将要访问的栅格中优先级较低的部分

另外，*curT* 就是用来区分 *nextP* 和 *overP* 的传播阈值，后面具体的代码中（函数 *NavFn :: updateCell*）会有说明，这部分的代码如下：

```
curT = COST_OBS; //当前传播阈值
curP = pb1; //当前用于传播的cell索引数组
curPe = 0; //当前用于传播的cell的数量
nextP = pb2;//用于下个传播过程的cell索引数组
nextPe = 0; //用于下个传播过程的cell的数量
```

```

overP = pb3; //传播界限外的cell索引数组
overPe = 0; //传播界限外的cell的数量

//初始化pending数组为全0，即设置所有的cell状态都不是“待办状态”
memset(pending, 0, ns*sizeof(bool));

```

接下来设置目标 *goal* 在 *potarr* 中的值为 0（这就是后面 *NavFn :: calcPath* 函数中用来判断是否搜索到了目标点的依据，因为整个地图上只有目标点的 *pot* 值为 0[低于 *COST_NEUTRAL* = 50]），并把它四周非障碍物的 *cell* 加入 *curP* 数组（在函数 *initCost* 中完成）中，为下一步的 *Potential* 值在整张地图上的传播做准备。代码如下：

```

int k = goal[0] + goal[1]*nx;
//设置costarr的索引k（目标）的pot值为0
//并对它四周的cell在pending数组中标记为“待办状态”，并把索引存放入curP数组
initCost(k,0);

```

NavFn::propNavFnDijkstra

该函数以目标点（*Potential* 值已初始化为 0）为起点，向整张地图的 *cell* 传播，填充 *potarr* 数组，直到找到起始点为止。*potarr* 数组的数据能够反映“走了多远”和“附近的障碍情况”，为最后的路径计算提供了依据

使用 *dijkstra* 算法广度优先传播，更新 *potential* 数组，获得传播起点 (*goal*) 到传播过程中任意点的最优路径。

结束条件：

- 达到了最大循环次数 *cycles*
- 跑完了所有可以更新的 *cell*
- 找到了起始点（则 *atStart = true*）

函数中主体是循环迭代更新 *potarr*。如果当前正在传播和下一步传播的集都为空，那么说明已经无法继续传播，可能有无法越过的障碍或其他情况，则退出循环。

接下来传播 *curP*，即当前 *cell*，调用的函数 *updateCell*，它能更新当前 *cell* 在 *potarr* 数组中的值，并将其四周符合特定条件的点放入 *nextP* 或 *overP*，用于下一步的传播。调用完成后，将 *nextP* 数组中的 *cell* 传递给 *curP*，继续上述传播，若 *nextP* 没有 *cell* 可以用来传播，则引入 *overP* 中的 *cell*。

nextP 和 *overP* 都来自从目标点开始传播的四周的 *cell*，区别在于它们的“父 *cell*”的 *pot* 值是否达到阈值 *curT*，没达到则放入 *nextP*，达到则放入 *overP*。

在从目标点向全地图传播的过程中检查，当起点的 *Potential* 值不再是被初始化的无穷大，而是有一个实际的值时，说明到达了起点，传播停止。

```
//记录起始位置(也就是算法寻找的终点)的索引
int startCell = start[1]*nx + start[0];

//循环迭代最大为cycles
for (; cycle < cycles; cycle++){
    //如果当前正在传播和下一步传播的集都为空，则退出
    if (curPe == 0 && nextPe == 0)
        break;

    //curPe是当前用于传播的cell的数量(nwv没有用)
    nc += curPe;
    if (curPe > nwv)
        nwv = curPe;

    //对pending数组进行设置(curP的cell设置为非待办状态)
    int *pb = curP;
    int i = curPe;
    while (i-- > 0)
        pending[*pb] = false;

    pb = curP;
    i = curPe;
    while (i-- > 0)
        //传播当前节点，更新其pot值
        //并将其四周符合特定条件的点放入nextP或overP，用于下一步传播
        updateCell(*pb);

    if (displayInt > 0 && (cycle % displayInt) == 0)
        displayFn(this);

    //将nextP数组中的cell传递给curP，继续上述传播
    curPe = nextPe;
    nextPe = 0;
    //navigation源码中很多时候感觉直接赋值就可以了，这种交换并没有什么意义
    pb = curP;
    curP = nextP;
    nextP = pb;

    //若nextP没有cell可以用来传播，则引入overP中的cell
    if (curPe == 0){
```

```

//增大传播阈值 (这里怎么理解呢？)
//初始是 COST_OBS = 254)(每次增加priInc默认是 2*COST_NEUTRAL = 2*50 = 100
curT += priInc;
curPe = overPe;
overPe = 0;
pb = curP;
curP = overP;
overP = pb;
}

//检查我们是否到达了起始点
if (atStart)
    //当[起点]的Pot值(可以认为Pot值就是goal到该点的路径长度)不再是被初始化的无穷大
    //而是有一个有限值时, 说明到达了起点, 传播停止
    if (potarr[startCell] < POT_HIGH)
        break;
}

```

NavFn::updateCell

该函数用于更新单个 cell 的 Potential 值。

首先获取当前 cell 四周邻点的 potarr 值，并取最小的值存入 ta，另外，执行一个判断，只有当当前 cell 不是致命障碍物时，才由它向四周传播，否则到它后停止。代码如下：

```

//先获取当前cell四周邻点的potarr值
float u,d,l,r;
l = potarr[n-1];
r = potarr[n+1];
u = potarr[n-nx];
d = potarr[n+nx];

//寻找左右邻点最小pot值与上下邻点最小pot值
float ta, tc;
if (l<r) tc=l; else tc=r;
if (u<d) ta=u; else ta=d;

//只有当当前cell不是致命障碍物时, 才由它向四周传播, 否则到它后停止, 不传播
if (costarr[n] < COST_OBS){
    //获取当前点的cost值
    float hf = (float)costarr[n];
}

```

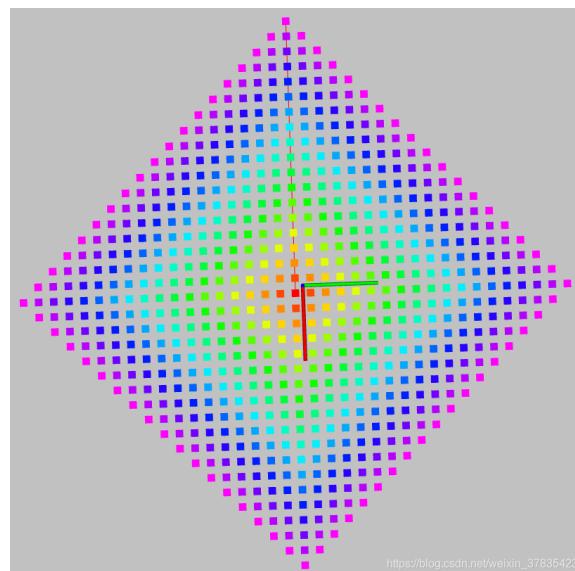
```
float dc = tc-ta;  
if (dc < 0){  
    //dc为左右邻点最小pot值与上下邻点最小pot值之差的绝对值  
    dc = -dc;  
    //将当前cell四周邻点中potarr的最小值赋给ta  
    ta = tc;  
}
```

接下来，计算当前 *cell* 的新的 *Potential* 值，计算 *Potential* 值时，有两种情况，需要对“左右邻点最小 pot 值与上下邻点最小 pot 值之差的绝对值”和“当前 *cell* 的 cost 值”比较（但是为什么是这个判断依据呢？），决定采用直接相加的公式还是二次逼近后再相加的公式。

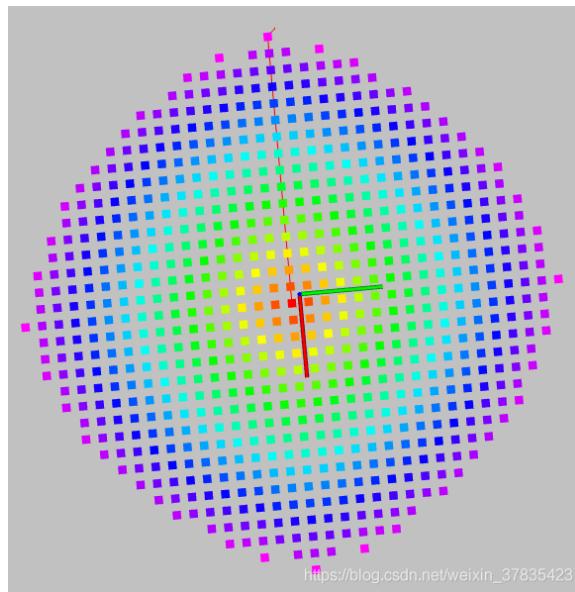
采用这两个公式的区别还是所谓的“菱形传播”和“圆形传播”，后者（二次逼近后再相加）能够产生效果更好的菱形传播。

参考博客：https://blog.csdn.net/weixin_37835423 采用两个不同的计算 Pot 值的公式导致圆形传播和菱形传播说明

直接利用 $pot = ta + hf$ 公式计算获得的 potential 分布图：



利用 $v = -0.2301d^2 + 0.5307d + 0.7040$, $pot = ta + hf * v$ 公式计算获得的 potential 分布图：



只有当前 *cell* 的 *Potential* 计算值 < 原本的 *Potential* 值，才更新，这意味着从目标点开始，它的 *Potential* 值被初始化为 0，不会被更新，接下来传播到它的四个邻点，才会开始更新他们的 *Potential* 值。最后，根据相应条件，将临近 *cell* 放入 *nextP* 或 *overP*，供下次迭代使用，代码如下（其中 $INVSQRT2 = \frac{1}{\sqrt{2}}$ ）：

```
// now add affected neighbors to priority blocks
if (pot < potarr[n]){
    float le = INVSQRT2*(float)costarr[n-1];
    float re = INVSQRT2*(float)costarr[n+1];
    float ue = INVSQRT2*(float)costarr[n-nx];
    float de = INVSQRT2*(float)costarr[n+nx];
    //只有当前cell的Potential计算值<原本的Potential值，更新当前cell的Potential值
    potarr[n] = pot;

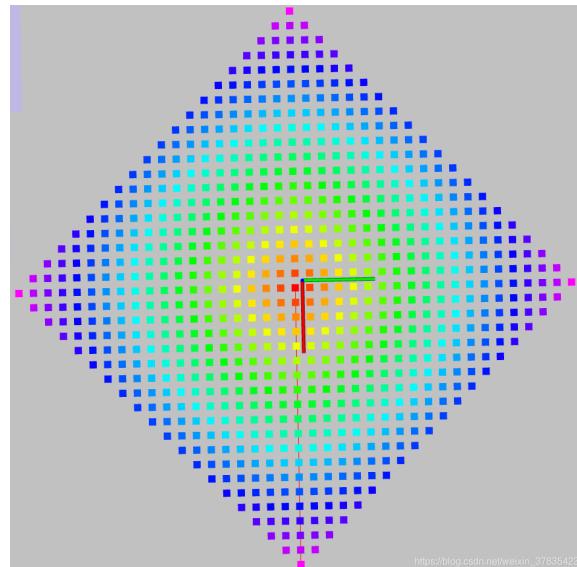
    //将临近cell放入nextP或overP，供下次迭代使用
    //如果当前cell的pot值小于传播阈值curT，则进入nextP，否则进入overP
    //curT是当前传播阈值 (curT=COST_OBS=254)
    //这里区分进入nextP和overP的目的只是为了获得更好的传播效果：
    //    若不加区分都进入nextP则是菱形传播
    //    若如此区分分别进入nextP和overP则是圆形传播(效果更好)
    if (pot < curT){
        //如果四周的cell本身的pot值比经过当前cell后再进入对应cell生成的pot值大的话
        //就进入nextP或overP，说明对应的cell的pot值可以进行优化
        if (l > pot+le) push_next(n-1);
        if (r > pot+re) push_next(n+1);
    }
}
```

```
    if (u > pot+ue) push_next(n-nx);
    if (d > pot+de) push_next(n+nx);
} else{
    if (l > pot+le) push_over(n-1);
    if (r > pot+re) push_over(n+1);
    if (u > pot+ue) push_over(n-nx);
    if (d > pot+de) push_over(n+nx);
}
}
```

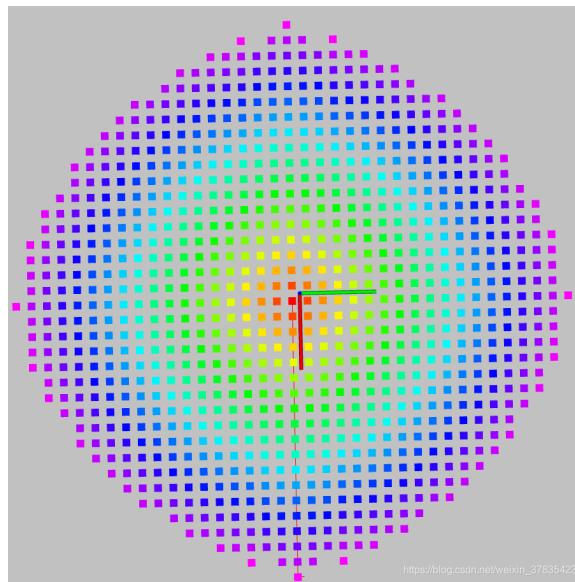
这里设置一个阈值 ($curT$) 来区分 $nextP$ 和 $overP$ 的传播先后顺序，结果是以目标点为圆心向外圆形传播，而不设阈值区分，则是以目标点为中心向外呈菱形传播，显然前者更合理。

参考博客：https://blog.csdn.net/weixin_37835423 设置传播阈值与否导致圆形传播和菱形传播说明

未设置传播阈值：



设置传播阈值：



从对比结果可以看出，当设置传播阈值时，传播方式是以 *potential* 值为边界一步步向外传播，而不设置传播阈值时，就是以固定的方形传播方式往外传播，会增加很多额外的不必要传播。所以设置传播阈值能提高传播速度。

NavFn::calcPath

该函数负责在 *potarr* 数组的基础上选取一些 *cell* 点来生成最终的全局规划路径，从起点开始沿着 *pot* 值梯度下降的方向寻找到目标点的最优轨迹。

首先初始化路径搜索的起始点，和偏移量（偏移量在后面的迭代过程中用于指示梯度下降的方向）：

```
//设置起始点，对于四点双线性插值 st总是在左上角
if (st == NULL) st = start; //st指向起点(是指针)
int stc = st[1]*nx + st[0]; //stc记录起点索引(是索引值)

//设置偏移量(用于指示梯度下降的传播方向)
float dx=0;
float dy=0;
//路径点索引
npath = 0;
```

这里所说的”四点双线性插值”不太理解，这个应该后后面 *NavFn :: gradCell* 的梯度计算有关，有待学习。

接下来进入路径搜索的主循环，其实就是通过计算当前 *cell* 的插值梯度，来决定梯度下降的前进方向，进而不断接近目标点，然后通过目标点的 *pot* 值 < *COST_NEUTRAL* (这是只有目标点

所具有的特性) 来判断是否找到了目标点。另外在循环中需要进行震荡检测, 防止陷入困境(如局部最低点等)。

主循环的代码如下:

```
//最多进行cycles次循环
for (int i=0; i<n; i++){
    //计算下一个临近点的索引(根据dx、dy给出的方向)
    int nearest_point = std::max(0,std::min(nx*ny-1,stc+(int)round(dx)+(int)(nx*round(dy))));
    //如果下一个临近点的pot值小于COST_NEUTRAL(只有被初始化为0的目标点才有可能)则表示找到了目标点
    //用梯度下降法搜索路径时, 是从起始点到目标点方向(不同于计算pot值时是从目标点到起始点)
    if (potarr[nearest_point] < COST_NEUTRAL){
        pathx[npath] = (float)goal[0];
        pathy[npath] = (float)goal[1];
        return ++npath;
    }

    //如果到了第一行或最后一行, 即超出边界
    if (stc < nx || stc > ns-nx){
        ROS_DEBUG("[PathCalc] Out of bounds");
        return 0;
    }

    //添加至路径点(dx、dy总是小于1的值, 相当于就是指示方向)
    pathx[npath] = stc%nx + dx;
    pathy[npath] = stc/nx + dy;
    npath++;

    //震荡检测(若某一步和上上步的位置相同则认为在振荡)
    bool oscillation_detected = false;
    if( npath > 2 && pathx[npath-1] == pathx[npath-3] && pathy[npath-1] == pathy[npath-3] ){
        ROS_DEBUG("[PathCalc] oscillation detected, attempting fix.");
        oscillation_detected = true;
    }

    int stcnx = stc+nx; //当前点下方的点的索引
    int stcpn = stc-nx; //当前点上方的点的索引

    //检查当前到达节点及周边的8个节点是否有障碍物(或者上一步发现了震荡现象)
    //如果有的话, 则直接将stc指向这8个节点中potential值最低的节点
    if (potarr[stc]      >= POT_HIGH || potarr[stc+1]      >= POT_HIGH ||
        potarr[stc-1]     >= POT_HIGH || potarr[stcnx]      >= POT_HIGH ||
        potarr[stcnx+1]   >= POT_HIGH || potarr[stcnx-1]   >= POT_HIGH ||
        potarr[stcpn]     >= POT_HIGH || potarr[stcpn+1]   >= POT_HIGH ||
        potarr[stcpn-1]   >= POT_HIGH || oscillation_detected){
        ROS_DEBUG("[Path] Pot fn boundary, following grid (%0.1f/%d)", potarr[stc], npath);
        int minc = stc;
        int minp = potarr[stc];
    }
}
```

```

int st = stcpix - 1;
//寻找周围八个邻点的pot中的最小值
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st++;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st++;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st = stc-1;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st = stc+1;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st = stcnx-1;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st++;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
st++;
if (potarr[st] < minp) {minp = potarr[st]; minc = st; }
stc = minc;
dx = 0;
dy = 0;

ROS_DEBUG("[Path] Pot: %0.1f pos: %0.1f,%0.1f", potarr[stc], pathx[npath-1], pathy[npath-1]);

if (potarr[stc] >= POT_HIGH){
ROS_DEBUG("[PathCalc] No path found, high potential");
//savemap("navfn_highpot");
return 0;
}
}

//当周围八个邻点没有障碍物时
//如果有好的梯度，则直接计算梯度，并沿着梯度方向查找下一个节点
else{
//计算以下四个点的梯度值(用于插值得到当前的点的插值梯度)
gradCell(stc); //当前点
gradCell(stc+1); //当前点右侧点
gradCell(stcnx); //当前点下方点
gradCell(stcnx+1); //当前点右下方点

//获取插值梯度(不太理解这个计算方法的意义)
float x1 = (1.0-dx)*gradx[stc] + dx*gradx[stc+1];
float x2 = (1.0-dx)*gradx[stcnx] + dx*gradx[stcnx+1];
float x = (1.0-dy)*x1 + dy*x2;
float y1 = (1.0-dx)*grady[stc] + dx*grady[stc+1];
float y2 = (1.0-dx)*grady[stcnx] + dx*grady[stcnx+1];
float y = (1.0-dy)*y1 + dy*y2;

//显示梯度
}

```

```

ROS_DEBUG("[Path] %0.2f,%0.2f  %0.2f,%0.2f  %0.2f,%0.2f  %0.2f,%0.2f; final x=%f, y=%f\n"
, gradx[stc], grady[stc], gradx[stc+1], grady[stc+1], gradx[stcnx], grady[stcnx], gradx[stcnx+1],
grady[stcnx+1], x, y);

//检查梯度是否为0
if (x == 0.0 && y == 0.0){
    ROS_DEBUG("[PathCalc] Zero gradient");
    return 0;
}

//向正确方向移动(pathStep设置移动的步长)
//hypot函数返回给定数字的斜边(即sqrt(x^2+y^2))
float ss = pathStep/hypot(x, y);
dx += x*ss;
dy += y*ss;

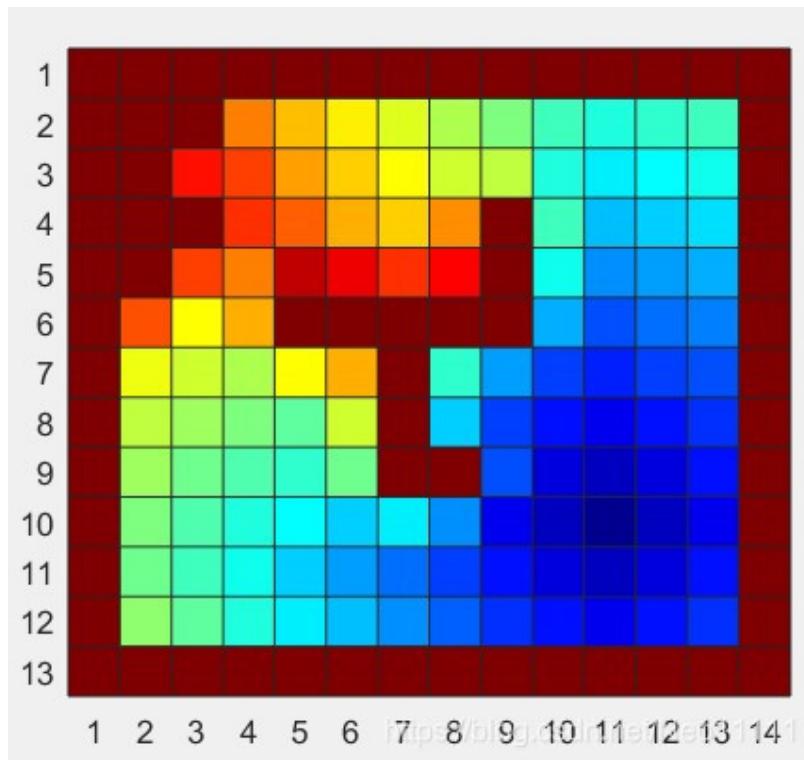
//检查溢出
if (dx > 1.0) { stc++; dx -= 1.0; }
if (dx < -1.0) { stc--; dx += 1.0; }
if (dy > 1.0) { stc+=nx; dy -= 1.0; }
if (dy < -1.0) { stc-=nx; dy += 1.0; }
}
}

```

参考博客：关于根据 costmap 生成 potarr 数组以及梯度数组 gradx 和 grady 的图示
原始的 *costmap* 如下图所示：

254	254	254	254	254	254	254	254	254	254	254	254	254	254	254	254
254	50	50	50	50	50	50	50	50	50	50	50	50	50	50	254
254	50	50	50	50	50	50	50	50	150	50	50	50	50	50	254
254	50	50	50	50	50	50	50	150	254	150	50	50	50	50	254
254	50	50	50	150	150	150	150	254	150	50	50	50	50	50	254
254	50	50	150	254	254	254	254	254	150	50	50	50	50	50	254
254	50	50	50	150	150	254	150	150	50	50	50	50	50	50	254
254	50	50	50	50	150	254	150	150	50	50	50	50	50	50	254
254	50	50	50	50	50	254	150	150	50	50	50	50	50	50	254
254	50	50	50	50	50	50	150	150	50	50	50	50	50	50	254
254	50	50	50	50	50	50	50	150	50	50	50	50	50	50	254
254	50	50	50	50	50	50	50	50	254	50	50	50	50	50	254
254	254	254	254	254	254	254	254	254	254	254	254	254	254	254	254

由它生成的 potarr 图示如下 (蓝色为小值, 红色为大值, 暗红色为无穷大):



由它生成的梯度数组 gradx (这个部分我还没搞懂) 如下图所示:

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.8	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.5	0.4	0.0	0.0	0.0	0.0	0.0	0.8	-0.3	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	-0.4	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.6	-0.4	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.5	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.6	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.7	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-1.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

由它生成的梯度数组 grady (这个部分我还没搞懂) 如下图所示:

0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	-0.5	-0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	-0.9	-0.9	0.0	0.0	0.0	0.0	0.6	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.7	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.8	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.7	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

NavFn::gradCell

该函数计算一个 cell 处的梯度值，定义向右和向下为梯度的正方向。

首先针对当前的 cell 是否是障碍区，对该函数中的 dx、dy（注意这里的 dx、dy 是该函数中的局部变量，和其他函数中的不同）（这里的 dx 和 dy 其实就是未进行归一化的梯度）有两种赋值方式：

```

float dx = 0.0;
float dy = 0.0;

//如果当前cell是障碍区
//则如果四周点不是障碍区就直接给dx、dy赋最大值(COST_OBS=+-254)(右下为正方向)
if (cv >= POT_HIGH){
    if (potarr[n-1] < POT_HIGH)
        dx = -COST_OBS;
    else if (potarr[n+1] < POT_HIGH)
        dx = COST_OBS;
    if (potarr[n-nx] < POT_HIGH)
        dy = -COST_OBS;
    else if (potarr[n+nx] < POT_HIGH)
        dy = COST_OBS;
}

//如果当前cell不是障碍区
//根据左右侧的梯度的平均值获得dx，根据上下方梯度平均值获得dy
else{
    if (potarr[n-1] < POT_HIGH)
        dx += potarr[n-1]- cv;
}

```

```

    if (potarr[n+1] < POT_HIGH)
        dx += cv - potarr[n+1];
    if (potarr[n-nx] < POT_HIGH)
        dy += potarr[n-nx]- cv;
    if (potarr[n+nx] < POT_HIGH)
        dy += cv - potarr[n+nx];
}

```

然后将 dx、dy 归一化后作为梯度值保存在 gradx 和 grady 梯度数组中：

```

float norm = hypot(dx, dy);
if (norm > 0){
    norm = 1.0/norm;
    gradx[n] = norm*dx;
    grady[n] = norm*dy;
}

```

NavFn::calcNavFnDijkstra

这个函数内完成了整个路径计算的流程，顺序调用了几个子部分的函数，代码如下：

```

//对翻译生成的costarr数组进行了边际设置等处理，并初始化了potarr数组和梯度数组gradx、grady
setupNavFn(true);

//从目标点开始传播计算pot值
propNavFnDijkstra(std::max(nx*ny/20,nx+ny),atStart);

//从起始点开始梯度下降搜索最优路径
int len = calcPath(nx*ny/2);

//如果找到了有效路径
if (len > 0){
    ROS_DEBUG("[NavFn] Path found, %d steps\n", len);
    return true;
}
else{
    ROS_DEBUG("[NavFn] No path found\n");
    return false;
}

```

6.4 global_planner 源码学习

全局路径规划 *BaseGlobalPlanner* 的 *plugin* 有三种: *navfn/NavfnROS*, *global_planner/GlobalPlanner*, *carrot_planner/CarrotPlanner*. 其中常用的为 *global_planner/GlobalPlanner*, 它是 *navfn/NavfnROS* 的改进版本, 包含 *Dijkstra* 和 *A** 算法进行全局路径规划。它与 *Navfn* 类似, 也是提供了一个 *makePlan* 函数作为它被 *move_base* 调用的总接口, 负责实现全局的路径规划。

6.4.1 A* 算法原理

6.4.2 源码相关文件

- 源码链接:

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

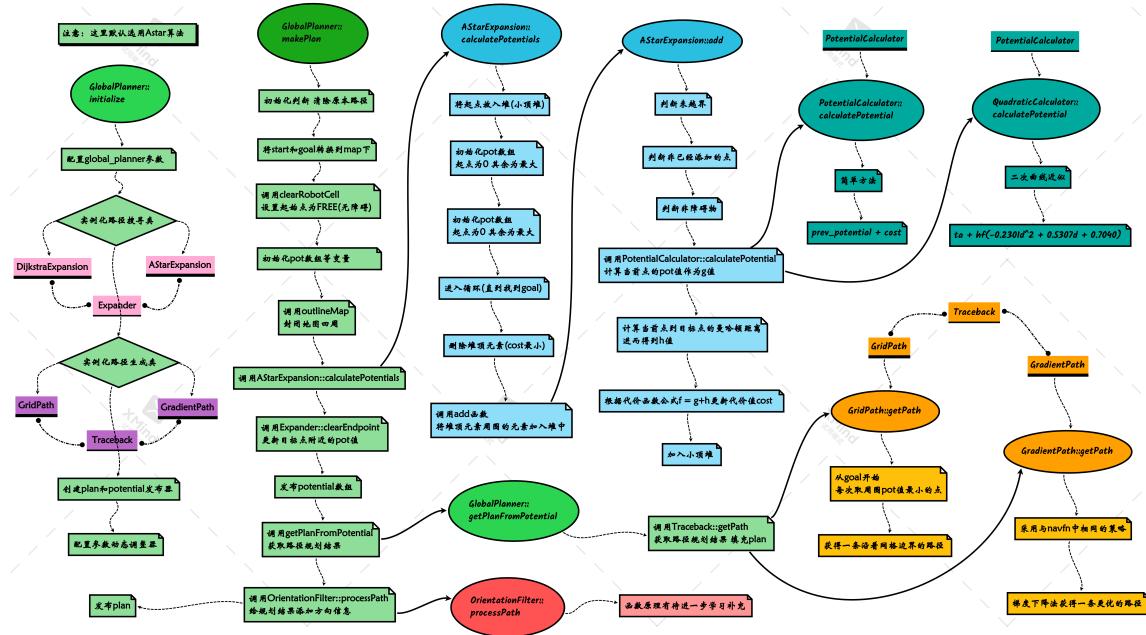
- 源码注释链接:

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

对应源码中的相关文件:

- *global_planner/src/plan_node.cpp*
- *global_planner/src/planner_core.cpp*
- *global_planner/src/astar.cpp*
- *global_planner/src/grid_path.cpp*
- *global_planner/src/gradient_path.cpp*
- *global_planner/include/potential_calculator.h*
- *global_planner/src/quadratic_calculator.cpp*
- *global_planner/include/expander.h*
- *global_planner/include/traceback.h*

6.4.3 整体结构图



6.4.4 参数配置

参数配置文件为: `global_planner_params.yaml`

参数列表:

- `allow_unknown`: 是否允许规划器规划穿过未知区域的路径 (只设计该参数为 `true` 还不行, 还要在 `costmap_commons_params.yaml` 中设置 `track_unknown_space` 参数也为 `true` 才行)
- `default_tolerance`: 当设置的目的地被障碍物占据时, 需要以该参数为半径寻找到最近的点作为新目的地
- `visualize_potential`: 是否显示从 `PointCloud2` 计算得到的势区域
- `use_dijkstra`: 设置为 `true`, 将使用 `dijkstra` 算法, 否则使用 `A*` 算法
- `use_quadratic`: 设置为 `true`, 将使用二次函数近似函数, 否则使用更加简单的计算方式, 这样节省硬件计算资源
- `use_grid_path`: 如果设置为 `true`, 则会规划一条沿着网格边界的路径, 倾向于直线穿越网格, 否则将使用梯度下降算法, 路径更为光滑点
- `old_navfn_behavior`: 若在某些情况下, 想让 `global_planner` 完全复制 `navfn` 的功能, 那就设置为 `true`, 但是需要注意 `navfn` 是非常旧的 ROS 系统中使用的, 现在已经都用 `global_planner` 代替 `navfn` 了, 所以不建议设置为 `true`

- *lethal_cost*: 致命代价值, 默认是设置为 253, 可以动态来配置该参数
- *neutral_cost*: 中等代价值, 默认设置是 50, 可以动态配置该参数
- *cost_factor*: 代价地图与每个代价值相乘的因子
- *publish_potential*: 是否发布 costmap 的势函数
- *orientation_mode*: 如何设置每个点的方向 (*None* = 0, *Forward* = 1, *Interpolate* = 2, *ForwardThenInterpolate* = 3, *Backward* = 4, *Leftward* = 5, *Rightward* = 6) (可动态重新配置)
- *orientation_window_size*: 根据 *orientation_mode* 指定的位置积分来得到使用窗口的方向. 默认值 1, 可以动态重新配置

6.4.5 plan_node.cpp

plan_node.cpp 文件中是一个 *PlannerWithCostmap* 类, 它继承自 *GlobalPlanner* 类 (在 *planner_core* 中定义)。在这个类中, 主要开启了两个线程, 提供了两种方式去开启 *global_planner* 的规划, 一种是服务请求, 一种是发布目标 goal 话题。这部分在该类的构造函数中实现, 代码如下:

```
PlannerWithCostmap::PlannerWithCostmap(string name, Costmap2DROS* cmap) :
    GlobalPlanner(name, cmap->getCostmap(), cmap->getGlobalFrameID()) {
    ros::NodeHandle private_nh("~");
    cmap_ = cmap;

    //第一个线程提供plan_service
    //一旦有请求, 就调用bool success = makePlan(req.start, req.goal, path);
    make_plan_service_ = private_nh.advertiseService("make_plan", &PlannerWithCostmap::makePlanService, this);

    //第二个线程是去订阅goal
    //拿到goal之后, 就调用makePlan(start, *goal, path);
    pose_sub_ = private_nh.subscribe<rm::PoseStamped>("goal", 1, &PlannerWithCostmap::poseCallback,
    this);
}
```

6.4.6 planner_core.cpp

planner_core.cpp 文件是 *global_planner* 的核心部分, 包括了其中的 *makePlan* 函数, 实现了全局路径规划的整体流程, 相对与 *navfn* 的实现, 代码更精简、封装更好, 十分优雅。

GlobalPlanner::initialize

GlobalPlanner 类的构造函数就是调用 *initialize* 函数, *initialize* 函数中做了参数读取、功能包中用到的类的实例化, 并且设置了参数动态配置。

GlobalPlanner::makePlan

该函数中包含了全局路径规划的整体流程逻辑，是该功能包的重点，代码逻辑与 Navfn 类似。

首先将目标点和起始点转换到 *map* 下，然后将起始点设置为 *FREE*，并初始化 *pot* 数组，封闭地图四周，这些部分和 *Navfn* 相同，只是做了一层封装，用函数调用的方式实现。

接下来调用函数计算 *p*otential 值，*planner_* 指针指向 *A** 或 *Dijkstrta* 算法类调用其各自的 *calculatePotentials* 函数，这是函数的重点，代码如下：

```
bool found_legal = planner_->calculatePotentials(costmap_->getCharMap(),
                                                    start_x, start_y, goal_x, goal_y,
                                                    nx * ny * 2, potential_array_);
```

接下来，如果在 *calculatePotentials* 中找到了合法的目标点，就可以获取规划的路径了，代码如下：

```
if (found_legal) {
    //根据pot数组获取路径规划结果plan(调用函数getPath 这个函数也有两种实现方式)
    if (getPlanFromPotential(start_x, start_y, goal_x, goal_y, goal, plan)) {
        //确保目标点和其余点有相同的时间戳
        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.....");
    }
} else{
    ROS_ERROR("Failed to get a plan.");
}
```

然后，给获得的路径数组添加方向信息，并发布规划结果，用于可视化：

```
//添加方向信息(给path “顺毛” 保证据弯的角度别变得太快)
orientation_filter_->processPath(start, plan);
//发布plan
publishPlan(plan);
```

6.4.7 aster.cpp

该文件中实现了 *aster* 路径搜索算法，实现方式简介优雅（与 *Dijkstrta* 算法的复杂实现形成对比），值得借鉴。

需要注意：*A** 算法是策略寻路，不保证一定是最短路径；*Dijkstrta* 算法是全局遍历，确保运算结果一定是最短路径，*Dijkstrta* 算法且算法需要载入全部数据，遍历搜索。

AStarExpansion::calculatePotentials

从起点开始逐渐扩散，将节点放入 *open* 堆，根据每个 *cell* 的代价值 (*cost* 值) 实现一个小顶堆。实现启发式搜索，不断计算更新经过的节点的 *pot* 值。

附带详细注释的代码如下：

```

bool AStarExpansion::calculatePotentials(unsigned char* costs,
                                         double start_x, double start_y,
                                         double end_x, double end_y,
                                         int cycles, float* potential) {
    //清空队列
    //queue_是启发式搜索到的向量队列<i,cost>
    queue_.clear();
    //toIndex函数获取索引值
    int start_i = toIndex(start_x, start_y);
    //将起点放入队列
    queue_.push_back(Index(start_i, 0));

    //std::fill: 将一个区间的元素都赋予指定的值，即在 (first, last)范围内填充指定值
    //将所有点的potential都设为一个极大值,potential就是估计值g,f=g+h
    //potential为g，也就是从起点到当前点的代价(Dijkstra算法中只有这个值)
    std::fill(potential, potential + ns_, POT_HIGH);
    //起点的potential值为0
    potential[start_i] = 0;

    //终点的索引
    int goal_i = toIndex(end_x, end_y);
    int cycle = 0;

    //循环目的：得到最小cost的索引，并删除它，如果索引指向goal则退出算法，返回true
    while (queue_.size() > 0 && cycle < cycles) {
        //将首元素放到最后，其他元素按照Cost值从小到大排列
        Index top = queue_[0];
        //pop_heap()是将堆顶元素与最后一个元素交换位置，之后用pop_back将最后一个元素删除
        //greate
        r1()是自己定义的一个针对Index的比较函数，这里表示依据Index的小顶堆
        std::pop_heap(queue_.begin(), queue_.end(), greater1());
        queue_.pop_back();

        //小顶堆 所以top就是cost最小的点的索引
        int i = top.i;
    }
}

```

```

//如果到了目标点就结束
if (i == goal_i)
    return true;

//对前后左右四个点执行add函数，将代价最小点i周围点加入搜索队里并更新代价值
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;
}

```

AStarExpansion::add

该函数向 *open* 小顶堆中添加节点，并更新代价函数。如果是已经添加的点则忽略，根据 *costmap* 的值如果是障碍物的点也忽略。

更新代价函数的公式：

$$f(n) = g(n) + h(n) \quad (6.25)$$

其中，*g(n)* 和 *h(n)* 的意义见注释，代码如下：

```

void AStarExpansion::add(unsigned char* costs,
                        float* potential, float prev_potential,
                        int next_i,
                        int end_x, int end_y) {
    //越界了
    if (next_i < 0 || next_i >= ns_)    return;

    //未搜索的点cost为POT_HIGH，如小于该值，则为已搜索点，跳过
    if (potential[next_i] < POT_HIGH)    return;

    //障碍物或者无信息的点
    if(costs[next_i]>=lethal_cost_ &&
       !(unknown_ && costs[next_i]==costmap_2d::NO_INFORMATION))
        return;

    //potential[next_i]是起点到当前点的cost即g(n)
    //prev_potential是父节点的pot值
    potential[next_i] = p_calc_->calculatePotential(potential, costs[next_i] + neutral_cost_, next_i,
    prev_potential);
}

```

```

int x = next_i % nx_, y = next_i / nx_;
//计算distance: 从节点n到目标点最佳路径的估计代价，这里选用了曼哈顿距离（不能斜着走 且无视障碍物）
float distance = abs(end_x - x) + abs(end_y - y);

//distance只是格子个数，还有乘上每个格子的真实距离或是分辨率，所以最后h = distance*neutral_cost_
//因此最后的f = h + g = potential[next_i] + distance*neutral_cost_
//neutral_cost_默认值为50
queue_.push_back(Index(next_i, potential[next_i] + distance * neutral_cost_));

//插入小顶堆
std::push_heap(queue_.begin(), queue_.end(), greater1());
}

}

```

6.4.8 grid_path.cpp

因为如 A^* 算法已经完成了路径搜索的话，其实要获取路径只是需要从 $goal$ 出发逆向走一遍就好了，所以这一部分要想实现很简单，但是这也只是一种简单的实现方式：创建一条沿着网格边界的路径；也可以采用梯度下降法，使用与 $navfn$ 中相同的梯度下降算法实现方式（在 $gradient_path.cpp$ 中实现）。

`getPath` 函数实现代码如下：

```

bool GridPath::getPath(float* potential,
                      double start_x, double start_y,
                      double end_x, double end_y,
                      std::vector<std::pair<float, float>>& path) {
    //将goal的坐标放入current中
    std::pair<float, float> current;
    current.first = end_x;
    current.second = end_y;

    int start_index = getIndex(start_x, start_y);

    path.push_back(current);
    int c = 0;
    int ns = xs_ * ys_;

    while (getIndex(current.first, current.second) != start_index) {
        float min_val = 1e10;
        int min_x = 0, min_y = 0;
        //从周围的8个点中寻找pot值最小的点
        for (int xd = -1; xd <= 1; xd++) {
            for (int yd = -1; yd <= 1; yd++) {

```

```

        if (xd == 0 && yd == 0)
            continue;
        int x = current.first + xd, y = current.second + yd;
        int index = getIndex(x, y);
        if (potential[index] < min_val) {
            min_val = potential[index];
            min_x = x;
            min_y = y;
        }
    }
    if(min_x == 0 && min_y == 0)
        return false;
    current.first = min_x;
    current.second = min_y;
    path.push_back(current);

    //设置了一个循环次数的上限
    if(c++ > ns*4){
        return false;
    }

}
return true;
}

```

6.4.9 gradient_path.cpp

梯度下降算法实现路径获取的实现方式，与 *navfn* 中的实现相同，不再赘述。

6.4.10 potential_calculator.h

calculatePotential() 计算根据 *use_quadratic* 的值有下面两个选择：

该函数计算 pot 值。

- 若为 True 则使用二次曲线计算
- 若为 False 则采用简单方法计算

简单方法即直接返回如下算式：

$$\begin{aligned}
 newpot &= prev_potential + cost \\
 &= prev_potential + costs[next_i] + neutral_cost_ \\
 &= \text{之前路径代价 } g + \text{地图代价} + \text{单格距离代价 (初始化为 50)}
 \end{aligned} \tag{6.26}$$

代码如下：

```

virtual float calculatePotential(float* potential, unsigned char cost, int n, float prev_potential=-1){
    //如果父节点的pot值小于0(调用时没有赋值 使用了缺省值-1)
    //在函数clearEndpoint中会出现这种缺省调用的情况)
    if(prev_potential < 0){
        //则将周围四个点的pot的最小值当做父节点的pot值
        float min_h = std::min(potential[n - 1], potential[n + 1]);
        float min_v = std::min(potential[n - nx_], potential[n + nx_]);
        prev_potential = std::min(min_h, min_v);
    }

    return prev_potential + cost;
}

```

其实实现的代码就是直接 `return` 就好了，但是为了适配 `clearEndpoint` 函数也可以直接调用，设置了缺省值，添加了上面的部分。

6.4.11 quadratic_calculator.cpp

该函数计算 pot 值，采用了与 navfn 中相同的二次曲线的计算方法，不再赘述。

6.4.12 expander.h

这个文件中定义了 `Expander` 类，这是两个路径搜索算法 (`astar` 和 `dijkstra`) 的父类，如果想要自己实现一个路径搜索算法也可以考虑继承该类。具体代码没有什么内容，不再介绍。

6.4.13 traceback.h

这个文件中定义了 `Traceback` 类，这是两个路径获取算法 (`grid_path` 和 `gradient_path`) 的父类，如果想要自己实现一个路径获取算法也可以考虑继承该类。具体代码没有什么内容，不再介绍。

6.5 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，先给机器人减速至阈值内，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。

6.5.1 源码相关文件

- 源码链接：

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

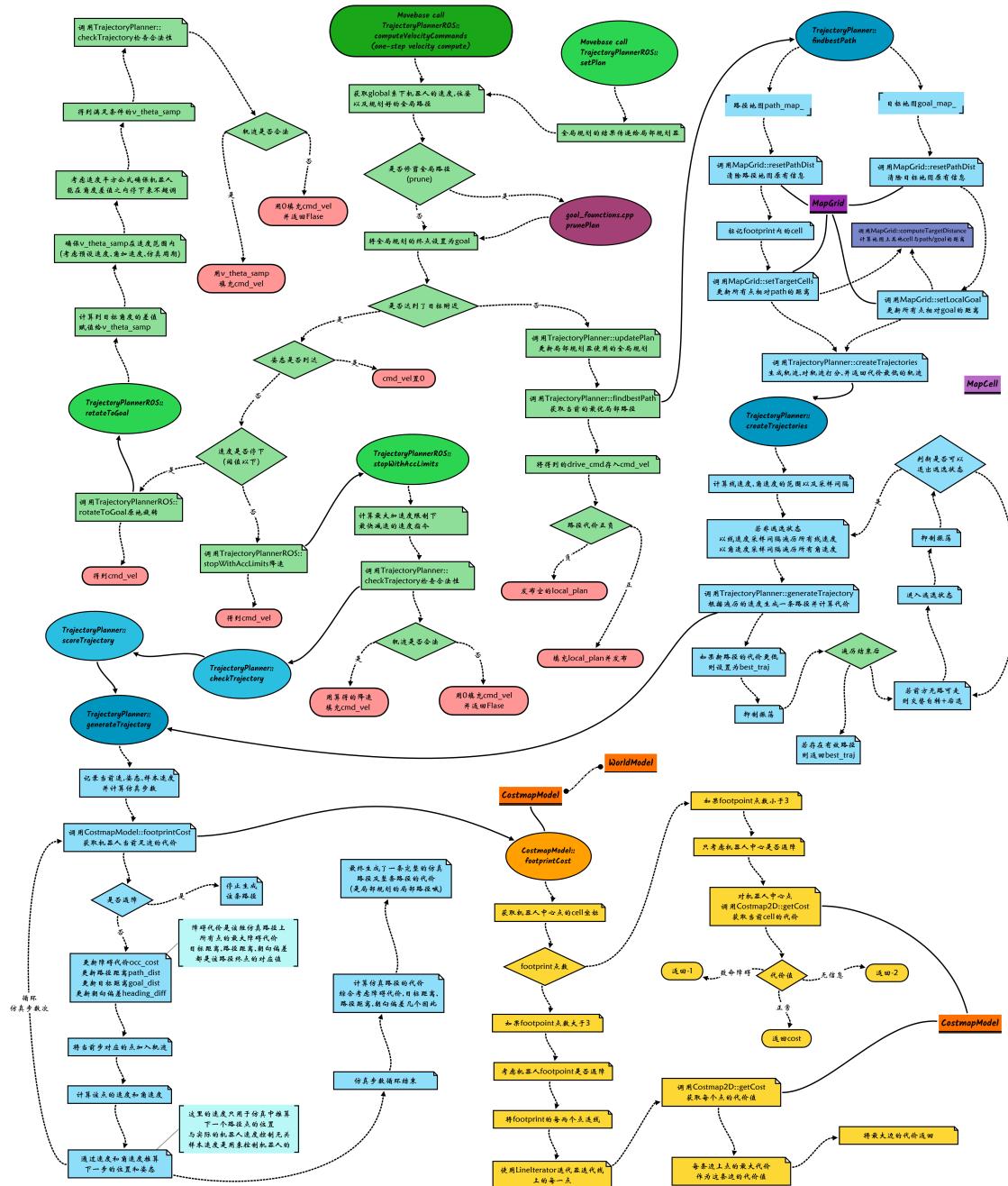
- 源码注释链接：

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

6.5.2 整体结构图



6.5.3 参数配置

参数配置文件为：

参数列表：

6.5.4 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），它可以用来自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_, transformed_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，减速、旋转：

- 如果机器人还未停止，调用类内 `stopWithAccLimits` 函数，给机器人减速，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-减速、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_tolerance_)) {
    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_, trans_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 found
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，局部规划器产生减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。

6.5.5 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，则调用减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 *heading_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::generateTrajectories

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::checkTrajectory 和 TrajectoryPlanner::scoreTrajectory

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）

6.5.6 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_ × 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
}
}
}

```

6.5.7 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;  
}
```

6.6 amcl 源码学习