轻舟智慧物流项目

技术文档

学　　校	中国矿业大学
组　　名	不显山车神
撰写日期	2022 年 7 月 26 日

目 录

一、阿克曼运动模型

1.1 阿克曼运动模型结构

阿克曼转向是一种现代汽车的转向方式，在汽车转弯的时候，内外轮转过的角度不一样，内侧轮胎转弯半径小于外侧轮胎。这种转向方式最初是由德国马车工程师 George Lankensperger 1817 年提出，他的代理商 Rudolph Ackerman 于 1818 年申请专利，所以从今往后这个转向原理就叫阿克曼转向几何了。阿克曼转向示意图如图 1.1-1 所示。

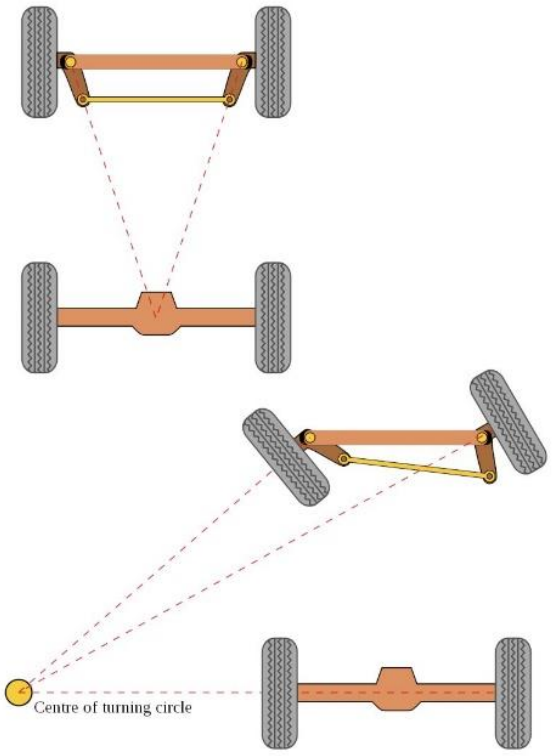

图 1.1-1 阿克曼转向示意图

普通的阿克曼智能小车考虑结构与控制成本的问题，一般不会采用复杂的差速器结构。一般的阿克曼智能小车底盘标配是后轮两个直流电机控制速度，前面一个舵机控制转向。由于没有差速器，为了能够很好地完成曲线行驶，后面左右两轮的转速一般不是相等的，而是根据转向角度有一个主动差速。所以，左右两个电机在负责小车速度的同时，还需要配合舵机控制小车的转向。

由于，左右电机是主动通过 PID 控制转速来达到期望速度和期望转角的，小车实际的航向由舵机和两个电机一起决定，相互影响。当舵机控制到的期望角度与电机要达到的期望角度不一致时，由于侧滑的存在，得到的轮式里程计

是不准确的。只有当舵机要达到的转向角和电机达到的转向角一致时，得到的轮式里程计才是准确的。

1.2 阿克曼运动模型原理

由于前面左右两轮的转角不同，小车的四个轮子绕圆心相同、半径不同的圆做圆周运动，如图 1.2-1 所示。

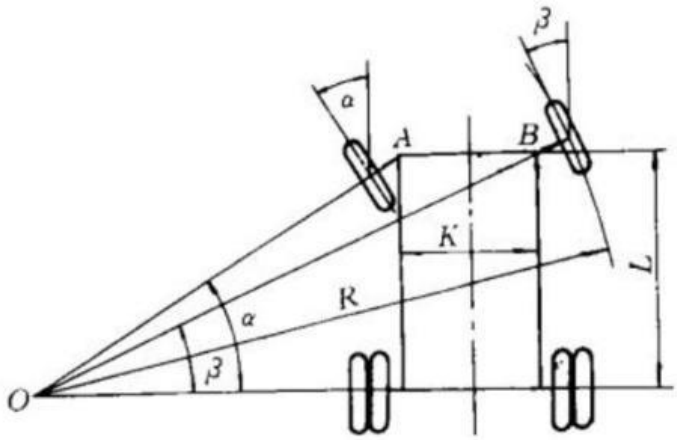

图 1.2-1 阿克曼运动模型示意图

单独分析前面左右两轮的转角对于分析汽车模型益处不大，于是将阿克曼四轮小车模型进一步简化为如图 1.2-2 所示的两轮单车模型（虚拟的前后轮假定在小车的中轴线上）：

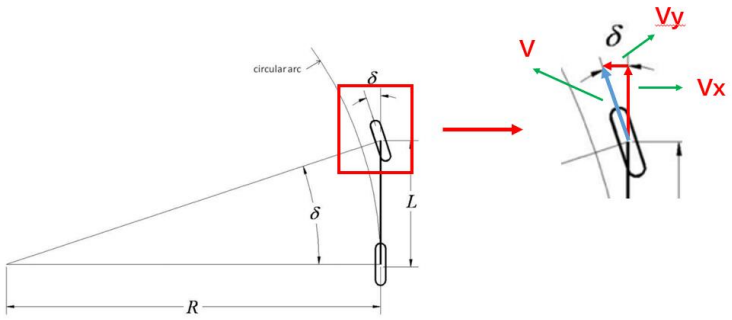

图 1.2-2 阿克曼运动简化模型

几何运动学模型如下：

$$\begin{cases} v_x = v\cos(\delta) \\ v_y = v\sin(\delta) \\ \omega = v\dfrac{\tan(\delta)}{L} \end{cases}.$$

其中，v_x表示 x 方向的速度，v_y表示 y 方向的速度，ω表示角速度。这里，

做了一步近似，由于前后轮绕圆心运动的半径相差的长度与它们半径的长度相比很小，因此可以忽略，即可以近似认为前轮速度等于后轮速度。

对于小车来说，X 轴对应其正前方，Y 轴对应其左方，Z 轴为机器人正上方。小车坐标系如图 1.2-3 所示。

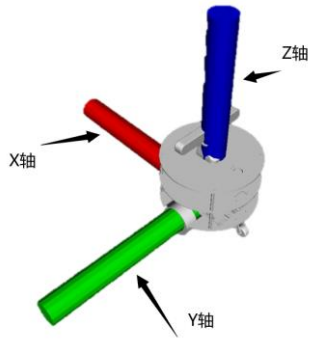

图 1.2-3 阿克曼小车坐标系示意图

因此，给出舵机的转角δ和小车的速度v，即可计算出小车在小车坐标系下的 X 和 Y 方向的线速度、绕 Z 轴旋转的角速度；相反，给出小车在 X 方向、Y 方向的线速度和绕 Z 轴旋转的角速度，也能得到舵机的转角δ和小车的速度v。这里小车的速度v指中轴线处的速度。这两种相反的推导分别对应阿克曼运动的正解和逆解。下面介绍如何在此基础上进行差速控制。

1.2.1.1 差速运动正解

机器人运动学正解，就是已知每个轮子的速度，解算出底盘的运动状态。即通过编码器获取到每个轮子的速度，可以解算出底盘在 X、Y、Z 三个坐标轴的速度和运行角度和距离，这个步骤，也就是常说的机器人里程计的计算，也就是航迹推演。航迹推演除了对机器人位姿进行估计，另一个很重要的关系是移动机器人前进速度v转向角速度ω与左轮速度v_l、右轮速度v_r之间的转换。

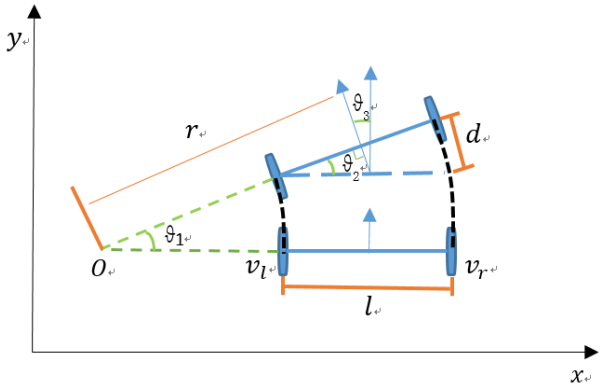

图 1.2-4 智能车后轮在两个相邻时刻位姿图

如图 1.2-4 是智能车（简化为两个后轮）在两个相邻时刻的位姿，其中θ_1是

两相邻时刻小车绕圆弧运动的角度，θ_3是两相邻时刻小车航向角的变化。l是左右轮之间的间距，d是右轮比左轮多走的距离。r是运动半径。

单纯考虑小车的 X 轴前进线速度，可以直观的看到，移动机器人前进的速度就等于左右轮速度的平均：

$$v = \frac{v_l + v_r}{2}.$$

对于小车的航向角，如图 5，把两个时刻的小车位置叠加起来，可以很清楚的看到航向角变化量为θ_3。从图中的几何关系可以得到：

$$\theta_1 = \theta_2 = \theta_3.$$

也就是说小车航向角变化了多少度，它就围绕运动轨迹的圆心旋转了多少度。由于相邻时刻时间很短，角度变化量很小，有下面的近似公式：

$$\theta_2 \approx sin\theta = \frac{d}{l} = \frac{(v_r - v_l)\Delta t}{l}.$$

所以可以得到机器人绕圆心运动的角速度：

$$\omega = \frac{\theta}{\Delta t} = \frac{v_r - v_l}{l}.$$

有了线速度、角速度，可以得到小车的圆弧运动的半径：

$$r = \frac{v}{w} = \frac{l \cdot (v_r + v_l)}{2(v_r - v_l)}.$$

从上式可以看出，当左轮速度等于右轮速度时，半径无穷大，即小车作直线运动。将上面的公式综合起来可以得到左右轮速度和线速度角速度之间的关系如下：

$$\begin{cases} v = \dfrac{v_l + v_r}{2} \\ \omega = \dfrac{v_r - v_l}{l} \\ r = \dfrac{v}{\omega} = \dfrac{l \cdot (v_r + v_l)}{2(v_r - v_l)} \end{cases}.$$

由此，可以利用编码器获取的两轮速度推导出小车的角速度和线速度，进一步对此角速度和线速度在时间上进行积分，即可得到机器人移动的距离，从而可以得到机器人里程计的定位的坐标。

1.2.1.2 差速运动逆解

机器人运动学逆解，就是已知期望的底盘运动速度，解算出两个轮子应该达到的速度。由上文正解得到的机器人线速度和角速度公式，即可解出v_r和v_l

$$\begin{cases} v_r = v + \dfrac{\omega \cdot l}{2} \\ v_l = v - \dfrac{\omega \cdot l}{2} \end{cases}.$$

将$\omega = v\dfrac{tan\,(\delta)}{L}$代入，得

$$\begin{cases} v_r = v\,(1 + \dfrac{tan\,(\delta) \cdot l}{2L}) \\ v_l = v\,(1 - \dfrac{tan\,(\delta) \cdot l}{2L}) \end{cases}.$$

二、中央调度管理

2.1 概述

上位机调度软件包含与导航功能包的 socket 通信实现、轻舟机器人坐标的显示、初始等待区、装货区、卸货区位置发送命令按钮，实现通过上位机下发目标点，调度轻舟机器人去往指定位置。

2.2 技术背景

2.2.1 开发软件概述

Qt Creator 是跨平台的集成开发环境（IDE），旨在为开发者带来最好的体验。Qt Creator 可在 Windows、Linux 和 macOS 桌面操作系统上运行，并允许开发人员在桌面、移动和嵌入式平台创建应用程序。

2.2.2 信号（Signal）与槽（Slot）机制

信号和槽机制是 Qt 的核心机制，要精通 Qt 编程，就必须对信号和槽有所了解。信号和槽是一种高级接口，应用于对象之间的通信，它是 Qt 的核心特性，也是 Qt 区别于其它工具包的重要地方。信号和槽是 Qt 自行定义的一种通信机制，它独立于标准的 C/C++语言，因此要正确的处理信号和槽，必须借助一个称为 MOC（Meta Object Compiler）的 Qt 工具，该工具是一个 C++预处理程序，它为高层次的事件处理自动生成所需要的附加代码。

信号就是在特定情况下被发射的事件，例如 PushButton 最常见的信号就是鼠标单击时发射的 clicked()信号，一个 ComboBox 最常见的信号是选择的列表项变化时发射的 CurrentIndexChanged()信号。

槽就是对信号响应的函数。槽就是一个函数，与一般的 C++函数是一样的，可以定义在类的任何部分（public、private 或 protected），可以具有任何参数，也可以被直接调用。槽函数与一般的函数不同的是：槽函数可以与一个信号关联，当信号被发射时，关联的槽函数被自动执行。

一种编码设置信号与槽的方法是：使用 connect()函数关联，其典型代码如下，按钮的 clicked()信号和槽函数 print()被连接在一起。

此外，还可以使用 Qt Designer 的 GUI 设计界面选择信号与槽的类型即可进行连接，无需手工写代码，而是交给程序完成。

2.2.3 套接字（Socket）

2.2.3.1 基本概念

Socket 是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。应用层通过传输层进行数据通信时，TCP 会遇到同时为多个应用程序进程提供并发服务的问题。多个 TCP 连接或多个应用程序进程可能需要通过同一个 TCP 协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP / IP 协议交互提供了 Socket 接口。应用层可以和传输层通过 Socket 接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。建立 Socket 连接至少需要一对套接字，其中一个运行于客户端（Client Socket），另一个运行于服务器端（Server Socket）。

2.2.3.2 连接步骤

Socket 连接的主要步骤如图 2.2-1 所示。具体来说，为了客户端和服务器端顺利连接成功，其中最重要的三个步骤是：服务器监听，客户端请求，连接确认。

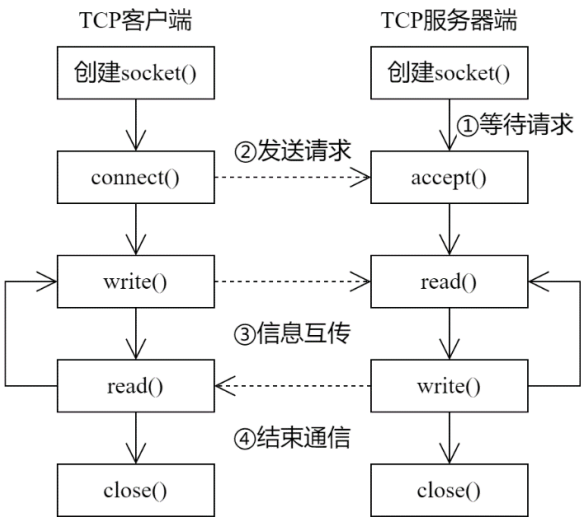

图 2.2-1 Socket 连接主要步骤

服务器监听：Server Socket 并不定位具体的 Client Socket，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

客户端请求：指 Client Socket 提出连接请求，要连接的目标是 Server Socket。为此，Client Socket 必须首先描述它要连接的 Server Socket，指出

Server Socket 的地址和端口号，然后就向其提出连接请求。

连接确认：当 Server Socket 监听到或者说接收到 Client Socket 的连接请求时，就响应 Client Socket 的请求，建立一个新的线程，把 Server Socket 的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而 Server Socket 继续处于监听状态，继续接收其他 Client Socket 的连接请求。

2.3 需求实现与界面设计

2.3.1 需求实现

(1) 实现题目要求的基本功能：Socket 通信，不同系统之间互传信息；

(2) 设置了程序的初始化步骤，打开程序时，自动填充上一次退出程序的记录，设置了默认端口 10000，减少用户操作步骤；

(3) 使用表格对 Socket 信息进行记录，相比命令行输出可视化效果更佳，同时也提供了筛选信息类型、检索信息内容的功能；

(4) 可以将表格内的数据以 XML 文件形式保存到本地，作为程序日志记录；

(5) 对于各类无效或者出错的情况（如无法连接到对方服务器、关闭了程序窗口、发送空字符串等）进行了不同处理，程序具有较高鲁棒性；

(6) 使用 Qt DarkStyle 美化程序界面，简洁大方，具有现代感。

2.3.2 界面设计

程序界面主要由如下模块构成，包括连接配置、交互状态记录、信息发送框，小车状态反馈模块、坐标发送模块。

在连接配置模块中，最上方的单选按钮，选择本机需要设置的类型（客户端或者服务器），接着需要使用单行文本框填写服务器 IP 和端口（对于服务器是填写自己监听的地址和端口，对于客户端是需要连接的服务器的地址和端口），便于后续连接操作。右侧有三个按钮，包括一键获取本机 IP 地址、监听/连接到服务器、停止监听/断开连接。

在交互状态记录模块中，左上方是一个下拉菜单，可以选择信息输入输出的方向，对信息类型进行筛选，上方是单行文本框，可以输入需要搜索的信息，对信息内容进行筛选。中间的表格控件则是把输入和接收各种信息按照时间顺序显示在表格中，其中一条信息主要显示的内容包括发送/接收时间、发送/接收的地址和端口，信息方向（输入/输出）、服务器/客户端当前状态信息或者发送/

接收的信息。下方是两个按钮，左下方的清空可以清空当前表格信息，右下方的保存可以将当前表格内的数据以 XML 文件的形式保存到本地。

在信息发送框模块中，中间是发送文本框，支持中英文、emoji 表情等，支持多行输入一次性输出，左下方是清空文本框按钮，按下后当前文本框的文本被清空，右下角是发送按钮，按下后当前文本框中的内容将被发送。

右侧支持接收小车的实时参数，可以发送小车当前需要到达的坐标如等待区、装货区、卸货区的坐标。

本程序界面如图 2.3-1 所示。

图 2.3-1 程序界面设计

2.4 主要功能设计

2.4.1 服务器监听

点击开始监听按钮后，首先判断当前类型是否为服务器类型，然后创建新的 TcpServer，构建监听信号与槽函数的连接，并使 TcpServer 保持 linsten 状态，当获得连接后进入 isListening 状态，输出“服务器已监听”的提示信息，其代码如代码 2.4-1 所示。

代码 2.4-1 服务器监听主要代码

widget.cpp

```
void Widget::on_btn_connect_clicked()
{
    … //ui 操作
    if (type == tr("服务器"))
    {
```

widget.cpp

```
        tcpServer = new QTcpServer(this);
        tcpSocket = NULL;

connect(tcpServer,SIGNAL(newConnection()),this,SLOT(onNewConnected())
);

        insertRow(ip + ":" + port, " ", tr("服务器尝试监听"));

        QHostAddress addr(ip);
        tcpServer->listen(addr, port.toInt());

        if (tcpServer->isListening())
        {
            insertRow(ip + ":" + port, " ", tr("服务器已监听"));
            ui->btn_disconnect->setEnabled(true);
        }
        else
        {
            insertRow(ip + ":" + port, " ", tr("服务器监听失败，请重试
"));
            … //ui 操作
        }
    }

    if (type == tr("客户端")){
        …
    }
}
```

2.4.2 服务器接收消息

服务器在 onSocketReadyRead()槽函数中进行接收消息的处理，当 TcpServer 接收到 readyRead()信号便触发该函数，读取相应客户端发送的消息。

首先判断消息是否可以多行读取，使用 readLine()方法将接收的信息加入接收消息列表。然后判断消息是否是以 position 开头的小车位置消息，是则需要显示在小车实时参数界面，否则当作普通消息处理，只需在表格中显示消息信息即可，其代码如代码 2.4-2 所示。

代码 2.4-2 服务器接收消息主要代码

widget.cpp

```
void Widget::onSocketReadyRead()
{
    QString recv_msg = "";
    while (tcpSocket->canReadLine())
    {
```

widget.cpp

```
        recv_msg.append(tcpSocket->readLine());
    }
    if (recv_msg.startsWith("position:")) {
        qDebug()<<recv_msg;
        QString position_str = recv_msg.split("position:")[1];
        position_str.chop(1);
        qDebug()<<"posi"<<position_str;
        insertRow(tcpSocket->peerAddress().toString() + ":" +
QString::number(tcpSocket->peerPort()), "in", position_str);
        QList position = position_str.split(',');
        ui->car_1->setText(position[0]);
        … //其他需要在 ui 显示参数的操作
    }
    else {
        recv_msg.chop(1);
        insertRow(tcpSocket->peerAddress().toString() + ":" +
QString::number(tcpSocket->peerPort()), "in", recv_msg);
    }
}
```

2.4.3 服务器发送消息

由于发送等待坐标、装货坐标、卸货坐标代码类似，以发送等待区坐标为例，首先判断 tcpServer 时候进行连接，信息是否为空，然后 tcpServer 使用 write 函数将坐标消息 str（格式为“x 坐标,y 坐标”）传送给相应的服务器。其代码如代码 2.4-3 所示。

代码 2.4-3 服务器发送坐标消息主要代码

widget.cpp

```
void Widget::on_btn_wait_clicked()
{
    if (!tcpSocket)
    {
        QMessageBox::warning(NULL, tr("警告"), tr("没有进行过连接，请先连
接！"), QMessageBox::Yes);
        return;
    }
    if (tcpSocket->state() != QAbstractSocket::ConnectedState)
    {
        QMessageBox::warning(NULL, tr("警告"), tr("当前非连接状态，请先连
接！"), QMessageBox::Yes);
        return;
    }


    if (ui->le_wait_x->text().isEmpty() || ui->le_wait_y-
```

widget.cpp

```
>text().isEmpty())
    {
        QMessageBox::warning(NULL, tr("警告"), tr("将要发送的信息为空，请
重新输入！"), QMessageBox::Yes);
        return;
    }

    QString send_msg = ui->le_wait_x->text() + ',' + ui->le_wait_y-
>text();

    insertRow(tcpSocket->localAddress().toString() + ":" +
QString::number(tcpSocket->localPort()), "out" ,send_msg);
    QByteArray str = send_msg.toUtf8();
    qDebug()<<str;
    str.append('\n');
    tcpSocket->write(str);
}
```

三、计算机视觉

3.1 概述

分析题目要求，需要设计计算机视觉模块，该模块提供的功能有：小车距离估计、红绿灯识别、车道线分割、循迹运动。计算机视觉模块实现红绿灯检测、车道线识别，为导航过程中的任务提供必要的信息输入，在导航过程中提供必要的红绿灯数据，判断是否可以通过；在执行 S 弯运动时，提供 S 弯道的识别结果。

3.2 红绿灯识别技术

红绿灯实现的前提要求是，能实现红绿灯定位，满足该前提的方法有很多种，比如训练神经网络完成红绿灯识别；借助 Aruco Marker 实现红绿灯定位。考虑到 Jetson Nano 处理器的性能，我们选择使用后者完成红绿灯定位，算法流程图如下图 3.2-1 所示。

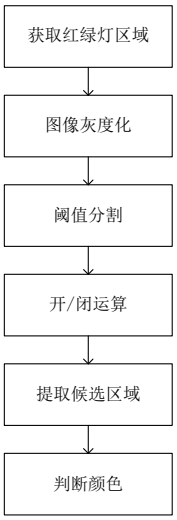

图 3.2-1 红绿灯检测算法流程图

3.2.1 各模块详细设计

3.2.1.1 获取红绿灯区域

可以借助 Opencv.aruco 模块检测 Marker 的候选区域，确定分析二维码确定 marker，这一检测过程可以使用 Opencv.aruco 模块中 detectMarkers()函数实现，该函数返回值是 Marker 四个角点的坐标以及 MarkerId，根据 Marker 的四个角点，分析 Marker 区域与红绿灯尺寸的比例，我们可以获取红绿灯大致的图像范围，取而代之的是我们不在整张图像中搜寻红绿灯，而是直接对获取到的大致

范围进行处理，红绿灯范围确定关键代码代码 3.2-1 所示。

代码 3.2-1 红绿灯范围确定关键代码

```
LightImg = Img[ MarkerROI[1]-10-2*H : MarkerROI[3]-3*H//2,
                MarkerROI[0] + W//4 : MarkerROI[2] - W//4, : ]
```

提取红绿灯结果图如图 3.2-2 所示。

图 3.2-2 红绿灯提取（左：原图，右：红绿灯区域）

3.2.1.2 图像灰度化

对提取到的图像执行灰度化，灰度化操作可以使图像高亮区域保持高亮，剔除掉颜色的影响，利于后续对红绿灯高亮区域的提取工作。关键代码如代码 3.2-2 所示：

代码 3.2-2 灰度化关键代码

```
LightImgGray = cv2.cvtColor(LightImg, cv2.COLOR_BGR2GRAY)
```

3.2.1.3 阈值分割

阈值分割的作用是根据设定的值处理图像的灰度值，比如灰度大于某个数值像素点保留。通过阈值以及有关算法可以实现从图像中抓取特定的图形，比如去除背景等。在灰度图的情况下，考虑到红绿灯的颜色相较于背景具有高亮度的特点，故此处使用阈值分割处理，进一步划分出红绿灯高亮部分。关键代码如代码 3.2-3 所示：

代码 3.2-3 阈值分割关键代码

```
th, MaskImg = cv2.threshold(LightImgGray, 150, 220,
cv2.THRESH_TOZERO)
```

3.2.1.4 开/闭运算

膨胀(dilated)是图像中的高亮部分进行膨胀，领域扩张，效果图拥有比原图更大的高亮区域。腐蚀(eroded)是图像中的高亮部分被腐蚀掉，领域缩减，效果图拥有比原图更小的高亮区域。开运算则是指先腐蚀再膨胀，运用开运算，可以能够除去孤立的小点、噪声点，而总体的位置和形状不变。对阈值分割结果进行开运算，达到去除噪声点，降低误检测的可能。闭运算操作则是先膨胀再腐蚀，达到补充小裂缝、缺少的点，而总体的位置和形状不变，最后处理结果则是包含红绿灯高亮区域的掩码图。关键代码如代码 3.2-4 所示：

代码 3.2-4 开/闭运算关键代码

```
MaskImg = cv2.morphologyEx(MaskImg, cv2.MORPH_OPEN, np.ones((2, 2),
np.uint8))
MaskImg = cv2.morphologyEx(MaskImg, cv2.MORPH_CLOSE, np.ones((3, 3),
np.uint8))
```

通过阈值分割，我们获取到了包含噪声、非红绿灯的源图像，通过采用开运算/闭运算操作对高亮度区域进行进一步去噪，去除掉非红绿灯区域（如边缘区域），处理结果如图 3.2-3 所示：

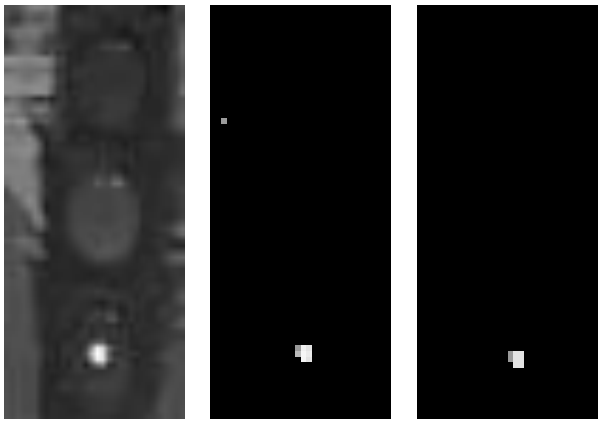

图 3.2-3 灰度化（左）、阈值分割（中）、开/闭运算（右）处理结果

3.2.1.5 提取候选区域

接下来对上述的结果图进行区域提取，选择需判断颜色的候选区域，并对区域的大小进行判断、筛选，关键代码如代码 3.2-5 所示：

代码 3.2-5 提取候选区域关键代码

```
contours,hierarchy=cv2.findContours(MaskImg,cv2.RETR_LIST,cv2.CHAIN_A
PPROX_SIMPLE)
for index, contour in enumerate(contours):
    Area = cv2.contourArea(contour)
    Hull = cv2.convexHull(contour, False)
    HullArea = cv2.contourArea(Hull)
        if Area >= 2 and Area < 1000 and Area / HullArea > 0.9:
```

3.2.1.6 判断颜色

此时对所有后候选区域范围内的图像求均值，均值结果作为该区域的颜色代表值，分别计算与红、黄、绿三种颜色的距离，距离最小者即为此时灯的颜色。

3.2.2 车道线识别技术

Unet 包含一个用于特征提取的 Encoder 结构和维度还原的 Decoder 结构。因其模型结构对称，形似英文字母 U 而被称为 Unet。

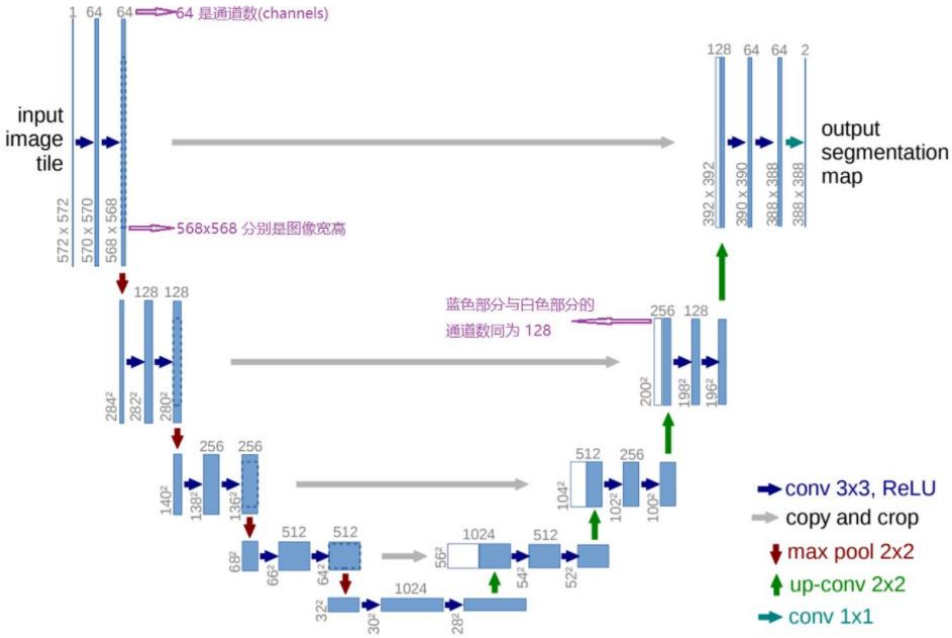

图 3.2-4 Unet 结构图

图 3.2-4 中，蓝/白色框表示 feature map；蓝色箭头表示 3x3 卷积，用于特征提取；灰色箭头表示 skip-connection，用于特征融合；红色箭头表示池化 pooling，用于降低维度；绿色箭头表示上采样 upsample，用于恢复维度；青色箭头表示 1x1 卷积，用于输出结果。

Encoder 由卷积操作和下采样操作组成，原文中所用的卷积结构统一为 3x3 的卷积核，padding 为 0，striding 为 1。由于 padding 为 0，所以每次卷积之后 feature map 的 H 和 W 变小了，在 Decoder 部分要将其还原。pytorch 代码如代码 3.2-6 所示。

代码 3.2-6 Unet 关键代码

```
nn.Sequential(nn.Conv2d(in_channels, out_channels, 3),
nn.BatchNorm2d(out_channels),
nn.ReLU(inplace=True))
```

feature map 经过 Decoder 恢复原始分辨率，该过程除了卷积之外比较关键的步骤就是 upsampling 与 skip-connection。上采样可以逐步恢复特征维度，使其与 Encoder 的各层输出跳层连接。而跳层连接则可以充分利用深浅层信息，防止过拟合。

我们在训练 Unet 车道线分割模型时，训练图像自行采集，自行标注。训练过程中设置 Epoch 为 300，batchsize 选取 15，学习率为 0.001。使用最终生成的模型进行检测，语义分割模型得到效果图如图 3.2-5 所示。

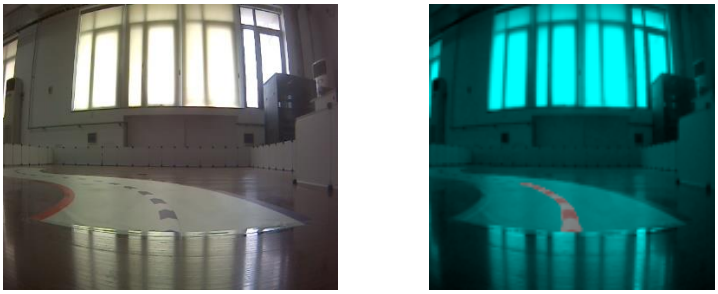

图 3.2-5 语义分割效果（左：原图，右：预测结果）

在整个方案中，为提高运行效率，我们将网络模型加载放入一个线程中，将加载的模型作为全局变量供主线程调用。处理过程为：首先通过调用相机获

取当前相机的图像，将图像传输到神经网络中，获取到预测结果图，提取结果图中预测的车道线位置，通过判断车道线中心偏移图像中心的角度，实时计算车辆转角。关键具体代码如下。

打开相机，读取图片，格式转换代码如代码 3.2-7 所示。

代码 3.2-7 预备工作关键代码

```
cap=cv2.VideoCapture(gstreamer_pipeline(flip_method=0),cv2.CAP_GSTREA
MER)
rate = rospy.Rate(5)
while 1:
        ret, Img = cap.read()
        WarpedImg = cv2.cvtColor(Img, cv2.COLOR_BGR2RGB)
        WarpedImg = ValImgTransform(Image.fromarray(WarpedImg))
        Input = WarpedImg.unsqueeze(dim=0)
        InputImg = Input.float().to(Device)
```

模型处理相机图像代码如代码 3.2-8 所示。

代码 3.2-8 语义分割关键代码

```
OutputImg = Unet_model(InputImg)
```

利用 cv2.moments()计算车道线的中心，通过将车道线中心与图片中心的对比来映射出角速度代码如代码 3.2-9 所示。

代码 3.2-9 车道线计算关键代码

```
h, w = cropped1.shape
M1 = cv2.moments(OutputImg, False)
try:
    cx1, cy1 = int(M1['m10'] / M1['m00']), int(M1['m01'] / M1['m00'])
    angle = -((int(cx1 - w / 2) / int(h - cy1))*0.55)
except:
    angle = 0
```

通过计算偏向，得到偏转的角度，包装成控制信号，向/cmd_vel 话题发布控制信号，完成巡线的代码如代码 3.2-10 所示。

代码 3.2-10 巡线话题发布关键代码

```
vel_msg = Twist()
vel_msg.angular.z = 0.001 * (dif ** 3)
if vel_msg.angular.z > 0.2:
    vel_msg.angular.z = 0.2
if vel_msg.angular.z < -0.2:
    vel_msg.angular.z = -0.2
vel_msg.linear.x = 0.2
qingzhou_vel_pub.publish(vel_msg)
rate.sleep()
```

四、路径规划模块

4.1 概述

智能机器人的基础性任务是机器人导航，任务内容是在环境中规划出到达目标点的一条前进轨线并引导机器人抵达目标点，实现地图导航关键在路径规划，根据对规划信息的考虑范围，路径规划算法可分为两类：全局路径规划和局部路径规划。全局路径规划是指根据全局环境信息进行路径规划；局部路径规划则是仅考虑以自身为中心的局部范围内的环境信息，结合传感器信息对路径进行实时规划。全局路径规划算法主要有 Dijkstra 算法、A*(A Star)算法、人工势场法、蚁群算法等。局部避障算法中最常使用的是动态窗口 DWA(Dynamic Window Approach)算法、TEB(Timed-Elastic-Band)。

上一个模块中，借助 SLAM 算法，配合激光雷达传感器，实现了对环境的精确建图，在路径规划、导航模块中，我们方案是结合 ROS 技术，合理配置 Amcl、move_base 等功能包，结合小车具体参数，设计匹配小车、适合小车的导航模块。

4.2 路径规划算法

本部分首先对实现机器人导航的路径规划算法进行介绍。

4.2.1 路径规划算法进行介绍

全局路径规划算法 Dijkstra 算法和 A*算法，以及局部路径规划算法动态窗口 DWA 算法和 TEB 算法。

4.2.1.1 Dijkstra 算法

Dijkstra 算法常用于解决有权图的最短路径问题，利用贪心算法的思想，每次选取当前最短路径作为下一步搜索的起始条件，通过不断迭代，直到扩展到目标点算法终止，此时搜索到的路径满足全局最优性。

详细算法步骤如下：

起点 s；集合 U 存有还未求得最短路径的顶点并维护每次扩展出的距离信息；集合 Q 维护当前抵达已扩展求得的顶点的最短距离。

a)算法初始化：集合 Q 只存有起点 s；集合 U 存有除起点外的所有顶点，并且存有每个节点到达起点 s 直接距离，如果顶点与起点直接相连，则属于一

步可达，两者间的距离为边的权重；反之，若该顶点不能一步到达起点，则距离为无穷大，可以用大过所有边权重的数值表示。

b)将当前集合 U 中距离最短的顶点 k 从集合 U 转移到集合 Q。

c)更新 U 中所有顶点到集合 Q 中所有顶点的距离。

重复步骤 b)和 c)，直到集合 U 为空，算法结束，集合 Q 中存有最优路径。

Dijkstra 算法搜索出来的路径满足全局最优，但随着搜索空间的增大，由于算法的广度优先策略，它遍历的节点过多，会导致算法效率低下。

4.2.1.2 A*算法

A*算法是 Dijkstra 算法的扩展，通过引入启发式函数，有效的提高了搜索效率，因而被广泛应用于寻路及图的遍历任务当中。

算法步骤：

起点 s；n 指向当前扩展到的节点；评价函数$f(n) = g(n) + h(n)$；评价函数中的$g(n)$表示从起始点到 n 节点的路径代价，或称路径距离；启发式函数$h(n)$表示从节点 n 到目标点的代价估计，或称估计距离。两个集合 OPEN、CLOSE，OPEN 集合维护算法扩展到的点和每个点的评价值，CLOSE 集合存有已扩展过的点和每个点的评价值，以及搜索路径；搜索图 G，以树的结构表示搜索路径过程。

a)算法初始化：初始化搜索图 G，G 根节点为起点 s；OPEN:={s}；CLOSE:={}。

b)取出 OPEN 表首的节点 n，扩展 n 的子节点并计算子节点的评价值 f(n)，将节点 n 插入到 CLOSE 集合中，将子节点插入搜索图 G 和 OPEN 表。

c)适当修改搜索图的指针，如在搜索过程中，再次搜索到位于 CLOSE 集合中的节点，此时需要根据情况决定是否将该节点取出放回到 OPEN 集合中。

d)按照评价函数值对 OPEN 表内节点排序

循环执行步骤 b)、c)、d)，直到搜索到目标节点，算法结束。最短路径存在在搜索图中。

A*算法中的关键是启发式函数的定义，A*算法启发式函数$h(n)$默认是曼哈顿距离法，如下方公式所示：

$$h(n)=D\times(|n_x-goal_x|+|n_y+goal_y|)$$
$$f(n)=g(n)+h(n)$$

D为启发式函数的权重值；(n_x,n_y)为当前节点的坐标；$(goal_x,goal_y)$为目标点的坐标；$f(n)$为代价估计函数，$g(n)$为当前路径代价函数。

改进版的 A*算法，考虑到二维平面上任意一个点有 8 个邻接点，故扩展方向应有 8 个方向，此时需要对启发式函数进行修改，常用的有对角线估计法和几何估计法，定义dx和dy如下方公式：

$$dx=|n_x-goal_x|,dy=|n_y-goal_y|$$

改进的几何估价法启发式函数$h(n)$为公式：

$$h(n)=D\times\sqrt{dx^2+dy^2}$$

改进的对角线估价法启发式函数$h(n)$为公式：

$$h(n)=D\times(dx+dy)+(D_2-2D)\times min(dx,dy)$$

D为启发式函数的权重，D_2为沿对角线方向的估计值(一般为$\sqrt{2}$)，$\min(dx,dy)$为当前点与目标点之间坐标差值的最小值。

4.2.1.3 动态窗口 DWA 算法

动态窗口 DWA 算法原理是在机器人的速度空间(v,w)进行多组采样，估计机器人运动模型，计算机器人在一段时间内的多组运动轨迹，并根据评价标准对轨迹进行评价，如轨迹是否会碰撞障碍物、距离目标点的距离等，选择评估表现最优的轨迹作为最优路径作为结果输出，示意如图 4.2-1。

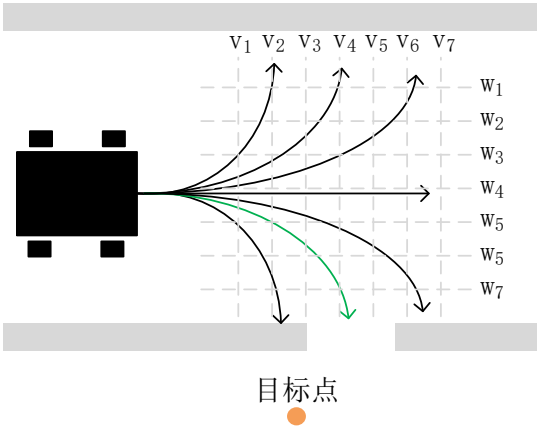

图 4.2-1 动态窗口法模拟预测轨迹示意图

4.2.1.4 时间弹性带法(TEB)

时间弹性带法 TEB 通过在起始点和目标点之间插入多个控制点，根据约束关系对控制点进行位置调整，约束关系如：全局路径规划结果、障碍物位置等，最终使得控制点序列远离障碍物并靠近全局路径规划结果。算法效果示意如图 4.2-2 所示。所规划路径中每个控制点都是在机器人坐标系下的位姿信息$x_i = (x_i, y_i, \beta_i)^T$，各个控制点间运动时间序列$T = \{\Delta T_i\}_{i=0,1,\dots,n-1}$，所有控制点组成序列$Q = \{x_i\}_{i=0,1,\dots,n}$表示了路径规划结果。

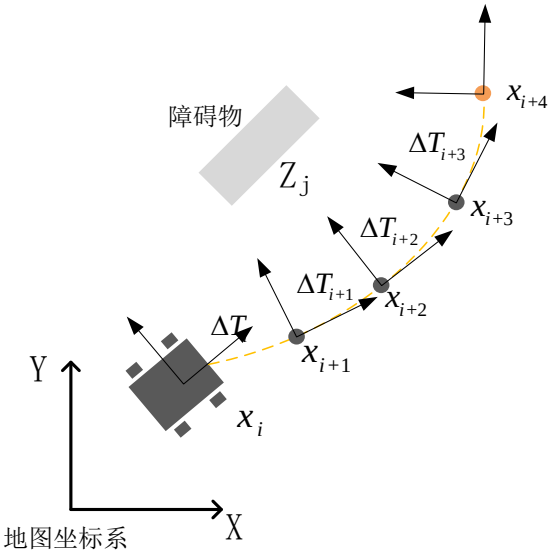

图 4.2-2 TEB 时间弹性带法模拟轨迹示意图

将控制点序列和运动时间序列合并成$B = (Q, T)$，通过多个目标约束函数进行非线性优化，最后得到的最优路径为$f(B)$，如下方公式：

$$f(B) = \sum \gamma_k f_k$$
$$B^* = \arg\min_B f(B)$$

上式中，$f(B)$为目标约束函数；$f_k(B)$为目标约束函数权值；B^*为路径规划最优结果。TEB 算法采用的目标函数有全局路径与障碍物的势场约束函数、速度与加速度范围动力学约束函数、最快路径约束函数等。在路径规划初始阶段，TEB 会根据配置的机器人参数和运动学动力学规划初始轨迹，同时在机器人前进过程中不断添加或删除控制点，根据约束条件持续调整规划路径。

4.2.2 算法对比

Dijkstra 算法采用层层扩展式直至目标节点，可以搜索到空间中的最优路径，

同时不可避免的存在一些多余的搜索区域。当搜索地图越来越大时，计算代价迅速增大，时间消耗明显，效率较低。A*算法始终向目标节点方向搜索，相比 Dijkstra 算法而言，搜索效率更高，但在某些特殊情况下也存在路径较长问题，此外，当启发式函数$h(s)$所估计的 s 点到终点的距离要小于等于实际 s 点到终点的距离时，A*具有最优性。

DWA 和 TEB 算法，两者相比：DWA 算法生成的局部路径具有高稳定、最优性的特点，适用于一些低速运动场景。TEB 算法生成的局部路径的特点是路径平滑，符合运动学，适合于高速运动场景。此外，当起始点的机器人朝向与目标点朝向不同时，DWA 算法会先抵达目标点，随后在原地调整自身朝向，TEB 算法则会在导航过程中调整自身朝向，两者在调整朝向上的时间花销不同。

4.3 实现代码分析

4.3.1 ROS Python 开发环境与工具

ROS 机器人开发平台，由斯坦福机器人研究实验室 Willow Garage 开发，它提供用于编写机器人软件程序软件架构、内部通信机制，能够实现机器人的底层抽象、硬件驱动以及常用功能。ROS 系统设计的初衷是降低开发机器人的门槛，提高开发的效率。此外，ROS 可以与仿真软件 Gazebo 配合使用，研发人员能够在 Gazebo 中搭建仿真环境，对算法进行仿真测试，进一步降低开发机器人的成本。Rviz 是一款可视化的调试软件，可配合 ROS 使用，达到可视化数据、实时显示、动态调试的目的。

ROS 内置 move_base 导航功能模块，其框架结构如图 4.3-1 所示。

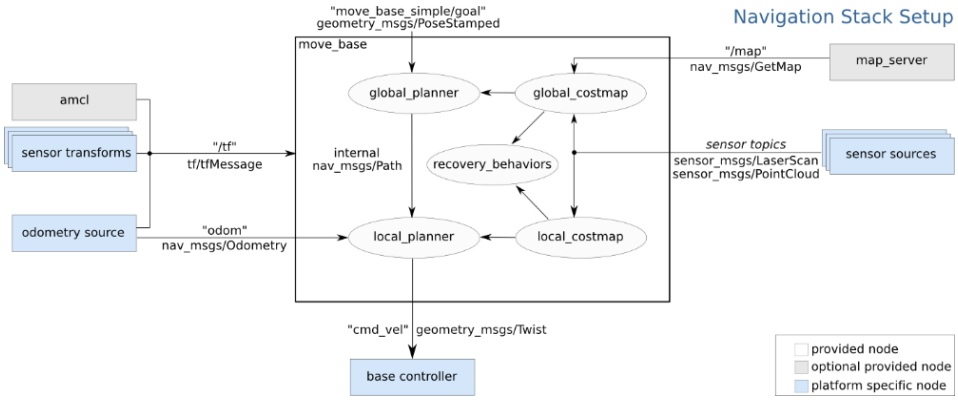

图 4.3-1 ROS 导航模块框图

(1)传感器转化(sensor transforms)指把传感器数据从传感器坐标系转化到机器人本体坐标系上。

(2)传感器源(sensor source)是指机器人导航的传感器数据输入，常见的有激光传感器数据、点云数据。

(3)里程计源(odometry source)是指里程计数据输入。

(4)控制者(base controller)，它是在导航运动过程中，接收路径规划结果 Twist，将 Twist 数据转换成下位机控制信号，如线速度、转向角度等计算过程。

(5)地图服务(map server)发布导航任务的地图。

(6)全局路径规划(global planner)依据 map_server 发布的地图计算得到全局代价地图，并计算全局路径规划。

(7)局部路径规划(local planner)根据 map_server 发布的地图计算得到局部代价地图，计算局部路径。

(8)恢复行为(recovery behavior)它是指当算法出现故障，无法正常定位导航时，机器人采取的一种自我恢复的行为，通常是原地旋转，实现重新定位。

总体来看，move_base 功能包通过获取 PoseStamped、tf、odom、map、Laserscan 等消息，内部使用规划算法计算最 8 优路径，并向下位机输出 Twist，实现机器人导航功能。

4.3.2 代码细节

导航模块通过 *qingzhou_move_base.launch* 文件进行参数配置，设定 map_server、amcl、move_base 节点，并加载配置文件，map_server 节点加载建图模块创建的地图文件，并将其发布在/map 话题下，amcl 节点设定 amcl 定位算法的具体参数，如 scan 话题、tf 变换树订阅、最大粒子数 max_particles、最小更新数 update_min_d 等，move_base 节点设定导航模块的具体参数，如在加载的代价地图 costmap_common_params.yaml 文件中声明的 observation_sources 参数，该参数定义了观察源设备的来源、数据类型、话题名等。如在 dwa_planner_params 文件中声明的 max_vel_x 设定小车最大前进速度、sim_time 设定局部路径规划算法仿真的时间，如代码 4.3-1 所示。

代码 4.3-1 导航模块关键代码

qingzhou_move_base.launch

```
<?xml version="1.0" ?>
<launch>
    <master auto="start"/>
    <include file="$(find qingzhou_nav)/launch/ydlidar.launch"/>
```

qingzhou_move_base.launch

```
    <node name="map_server" pkg="map_server" type="map_server"
args="$(find qingzhou_nav)/maps/bisai.yaml"
          output="screen"/>
    <include file="$(find qingzhou_nav)/launch/amcl.launch"/>
    <node pkg="move_base" type="move_base" respawn="false"
name="move_base" output="screen">
        <param name="base_global_planner"
value="global_planner/GlobalPlanner"/>
        <param name="planner_frequency" value="1.0"/>
        <param name="planner_patience" value="5.0"/>
        <param name="base_local_planner"
value="dwa_local_planner/DWAPlannerROS"/>
        <param name="controller_frequency" value="5.0"/>
        <param name="controller_patience" value="5.0"/>
        <rosparam file="$(find
qingzhou_nav)/config/costmap_common_params.yaml" command="load"
ns="global_costmap"/>
        <rosparam file="$(find
qingzhou_nav)/config/costmap_common_params.yaml" command="load"
ns="local_costmap"/>
        <rosparam file="$(find
qingzhou_nav)/config/local_costmap_params.yaml" command="load"/>
        <rosparam file="$(find
qingzhou_nav)/config/move_base_params.yaml" command="load"/>
        <rosparam file="$(find
qingzhou_nav)/config/global_costmap_params.yaml" command="load"/>
        <rosparam file="$(find
qingzhou_nav)/config/dwa_local_planner_params.yaml" command="load"/>
        <rosparam file="$(find
qingzhou_nav)/config/global_planner_params.yaml" command="load"/>
    </node>
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
qingzhou_nav)/rviz/move_base.rviz" output="screen"/>
</launch>
```

下方是 *costmap_common_params.yaml* 文件内容，如代码 4.3-2 所示：

代码 4.3-2 代价地图基本参数配置关键代码

costmap_common_params.yaml

```
robot_radius: 0.25
footprint:  [ [-0.16,-0.25], [0.16,-0.25], [0.16,0.25], [-
0.16,0.25] ] #[[0.23,0.19],[0.30,0],[0.23,-0.19],[-0.23, -0.19],[-
0.23,0.19]]
recovery_behavior_enabled: false
controller_frequency: 5
update_frequency: 1
observation_sources: scan
scan: {sensor_frame: base_link,  observation_persistence: 0.0,
max_obstacle_height: 0.3, min_obstacle_height: 0.05, data_type:
```

costmap_common_params.yaml

```
LaserScan, topic: /scan, marking: true,clearing: true}
```

通过上述过程，在配置好 move_base 等节点后，就可以在 Rviz 进行可视化，发布导航目标点进行导航，接下来需要获取地图中三个导航地点的位置，获取办法是记录在 Rviz 中发布装货区、卸货区、等待区处的导航点坐标，再使用 rospy 创建话题发布者，通过代码发布导航目标点，实现非可视化方式的导航，具体代码如代码 4.3-3 所示：

代码 4.3-3 导航实现关键代码

```
data = cli.recv(BUFSIZE)
if data == b"":
    eturn
data = data.decode()
    print('loading area is:', data)
    x, y = data.split(',')
    x = float(x)
    y = float(y)
    print(x,y)
    point=[(x, y, 0.000), (0.000, 0.000, -0.697, 0.717)]
    goal_zh=goal_pose(point)
    client.send_goal(goal_zh)
    client.wait_for_result()
```

大致流程为：

（1）socket 监听接收到来自客户端的导航目标点，并解析。

（2）使用 sent_to_data()发布导航目标点，进行导航

（3）执行导航任务（如红绿灯检测、车道线识别）

（4）wait_for_result()接收导航模块导航结果。

至此，完成一次导航任务。

五、二维栅格化地图的建立

5.1 概述

在建立二维栅格化地图的部分，采用激光 SLAM 技术。激光 SLAM 脱胎于早期的基于测距的定位方法（如超声和红外单点测距）。激光雷达（Light Detection And Ranging）的出现和普及使得测量更快更准，信息更丰富。激光雷达采集到的物体信息呈现出一系类分散的、具有准确角度和距离信息的点、被称为点云。通常，激光 SLAM 系统通过对不同时刻两片点云的匹配与比对，计算激光雷达相对运动的距离和姿态的改变，也就完成了对机器人自身的定位。激光雷达距离测量比较准确，误差模型简单，在强光直射以外的环境中运行稳定，点云的处理也比较容易。同时，点云信息本身包含直接的几何关系，使得机器人的路径规划和导航变得直观。激光 SLAM 技术的研究相对较早，在理论和实践方面较为成熟。基于视觉的 SLAM 目前主要有两种实现途径，一种是基于 RGB-D 深度相机的实现方式，另一种是基于单目、双目或者鱼眼相机的实现方式。视觉 SLAM 虽然在理论方面较为成熟，但在实际应用中还存在很多问题，应用很不成熟，因此目前主要是应用激光 SLAM 进行建图和导航。

SLAM（同时定位与建图）问题是一个鸡生蛋、蛋生鸡的问题。定位需要建图、建图需要先定位，SLAM 问题的难点也在于此。

本文选用 Gmapping 算法进行地图的构建。Gmapping 可以构建室内地图，适用于小场景，在构建小场景地图所需的计算量较小且精度较高。相比 Hector SLAM，Gmapping 对激光雷达频率要求低、鲁棒性高，Hector SLAM 在机器人快速转向的时候容易发生错误匹配，建出的地图发生错位。相比 Cartographer 算法，Gmapping 在构建小场景地图时，不需要太多的粒子并且没有回环检测，因此计算量大大减小，在资源有限的 Jetson Nano 板上实时性更好。Gmapping 算法有效利用了车轮里程计信息，提供了机器人的位姿先验，因此对激光雷达的频率要求低。而 Hector 和 Cartographer 的设计初衷不是为了解决平面移动机器人定位和建图，Hector 主要用于救灾等地面不平坦的情况，Cartographer 用于手持激光雷达完成定位与建图过程，因此都无法使用里程计。

Gmapping 算法基于 Rao-Blackwellized 粒子滤波算法（RBpf）。RBpf 是一种有效解决同时定位与建图的方法，将定位与建图分离，每一个粒子都携带一幅

地图，因此不适用于大场景。RBpf 存在的缺点有：所用粒子数多和频繁执行重采样。粒子数多会造成计算量和内存消耗增大，频繁执行重采样会造成粒子退化。Gmapping 算法在 RBpf 的基础上改进提议分布和选择性重采样，从而减少粒子个数和防止粒子退化。改进的提议分布不但考虑运动（里程计）信息还考虑最近的一次观测（激光）信息这样就可以使提议分布的更加精确从而更加接近目标分布。选择性重采样通过设定阈值，只有在粒子权重变化超过阈值时才执行重采样从而大大减少重采样的次数。

5.2 技术原理

Gmapping 功能包集成了 Rao-Blackwellized 粒子滤波算法，订阅机器人的深度信息、IMU 信息和里程计信息，同时完成一些必要的参数配置，即可创建完成基于概率的二维栅格化地图。

话题订阅、发布以及提供的服务的情况如表 5.2-1 所示：

表 5.2-1 话题订阅、发布以及提供的服务

	名称	类型	描述
话题订阅	tf	tf/tfMessage	需要进行激光，用于坐标系、基准和测距的相关框架转换
	scan	sensor_msgs/LaserScan	激光雷达扫描数据
话题发布	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
服务	Dynamic_map	nav_msgs/GetMap	调用此服务以获取地图数据

5.3 参数设置

本方案配置的参数及相应的描述如表 5.3-1 所示。

表 5.3-1 参数设置

参数	类型	值	描述
~throttle	int	1	每接收到该数量帧的激光雷达数据后只处理其中的一帧数据，默认每接收到一帧数据就处理一次
~base_frame	string	base_link	机器人基坐标系
~map_frame	string	map	地图坐标系
~odom_frame	string	odom	里程计坐标系
~map_update_interval	float	5	地图更新周期，该值越低，计算负载越大
~maxUrange	float	10	激光可探测的最大范围
~sigma	float	0.05	端点匹配的标准差
~kernelSize	int	1	在对应的内核中进行查找
~lstep	float	0.05	平移过程中的优化步长
~astep	float	0.05	旋转过程中的优化步长
~iterations	int	5	扫描匹配的迭代次数
~lsigma	float	0.075	似然计算的激光标准差
~ogain	float	3	似然计算时用于平滑重采样效果
~lskip	int	0	每次扫描跳过的光束数
~minimumScore	float	0	扫描匹配结果的最低值
~srr	float	0.1	平移函数，平移时的里程误差
~srt	float	0.2	旋转函数，平移时的里程误差
~str	float	0.1	平移函数，旋转时的里程误差
~stt	float	0.2	旋转函数，旋转时的里程误差
~linearUpdate	float	1	机器人每平移该距离后处理一次激光扫描数据
~angularUpdate	float	0.5	机器人每旋转该弧度后处理一次激光扫描数据
~temporalUpdate	float	3	如果最新扫描处理的速度比更新的速度慢，则处理一次扫描。该

			值为复数时关闭基于时间的更新
~resampleThreshold	float	0.5	基于 Neff 的重采样阈值
~particles	int	100	滤波器中的粒子数目
~xmin	float	-10	地图 x 向初始最小尺寸
~ymin	float	-10	地图 y 向初始最小尺寸
~xmax	float	10	地图 x 向初始最大尺寸
~ymax	float	10	地图 y 向初始最大尺寸
~delta	float	0.05	地图分辨率
~llsamplerange	float	0.01	似然计算的平移采样距离
~llsamplestep	float	0.01	似然计算的平移采样步长
lasamplerange	float	0.005	似然计算的角度采样距离
lasamplestep	float	0.005	似然计算的角度采样步长
~transform_publish_period	float	0.05	TF 变换发布的时间间隔
~occ_thresh	float	0.25	栅格地图占用率的阈值
~maxRange	float	-	传感器的最大范围

在实验中，发现 minimumScore 和 particles 参数对建图效果的影响较大。经过测试，将粒子数 particles 和最小匹配得分 minimumScore 分别设置为 100 和 50 时建图效果最好。

涉及的坐标变换情况如表 5.3-2 所示：

表 5.3-2 涉及的坐标变换

	TF 变换	描述
必须的 TF 变换	<scan frame> -> base_link	激光雷达坐标系之间与基坐标系之间的变换，一般由 start_transform_publisher 发布
	base_link -> odom	基坐标系与里程坐标系之间的变换，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