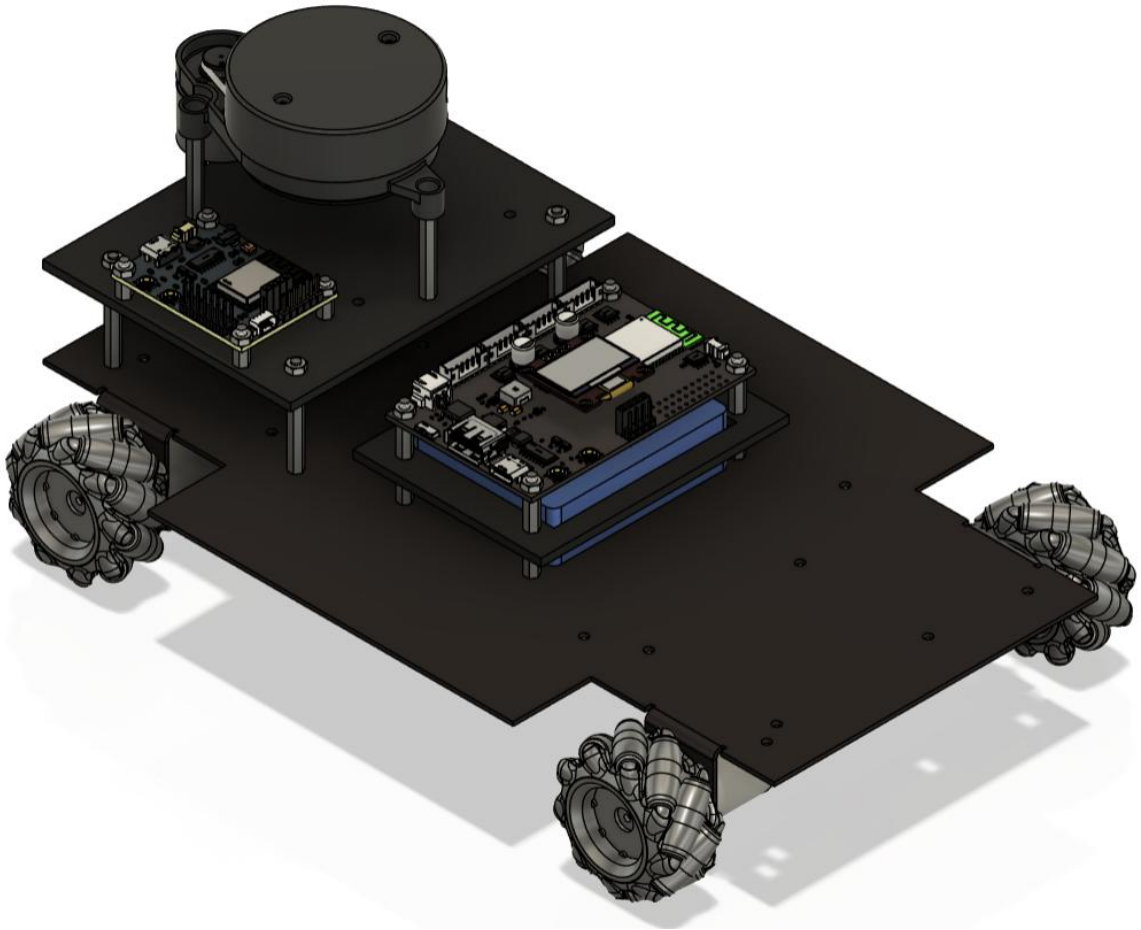


第九章搭建一个实体移动机器人【FishBot 四驱 V2 版本】

店铺：鱼香 ROS

目录

第9章 搭建一个实体移动机器人【四驱】

在前面的章节中，我们通过使用仿真的机器人设备学习了建图导航、运动控制和路径规划相关知识。但你肯定想拥有一辆属于自己的移动机器人，然后在真实机器上部署自己的运动控制和路径规划算法。本章我们就来学习如何一步步搭建一个实体的移动机器人开发平台。

9.1 移动机器人系统设计

在 6.1 节中，我们对机器人系统的组成有过简单的介绍，对于一个移动机器人来说，从系统功能角度来看，机器人由感知、决策和控制三部分组成。感知是通过各种传感器来实现的，比如激光雷达和编码器，决策部分软件上则是由各种算法组合实现，比如可以进行路径规划和运动控制的 Navigation 2，硬件上则依托性能较强的处理器实现，控制部分通常是由驱动系统和电机组成。

如果说要真的从零开始制作机器人，可能一整本书的篇幅都不够用，所以本章将依托于一款低成本的移动机器人平台 FishBot 的硬件，着重介绍实体机器人的软件部分的开发。以 FishBot 为例，我们先对移动机器人各个组成部分进行介绍。

9.1.1 机器人传感器

机器人的感知往往由各种传感器实现，FishBot 搭载了雷达、超声波、编码器和 IMU 四种传感器。要跑起来导航只需要重点关注激光雷达和编码器这两种传感器即可。

在之前仿真中我们已经介绍和使用过激光雷达传感器，它可以获取环境的深度信息，FishBot 使用的是单线旋转式激光雷达，它的外观如图 9-1 所示。

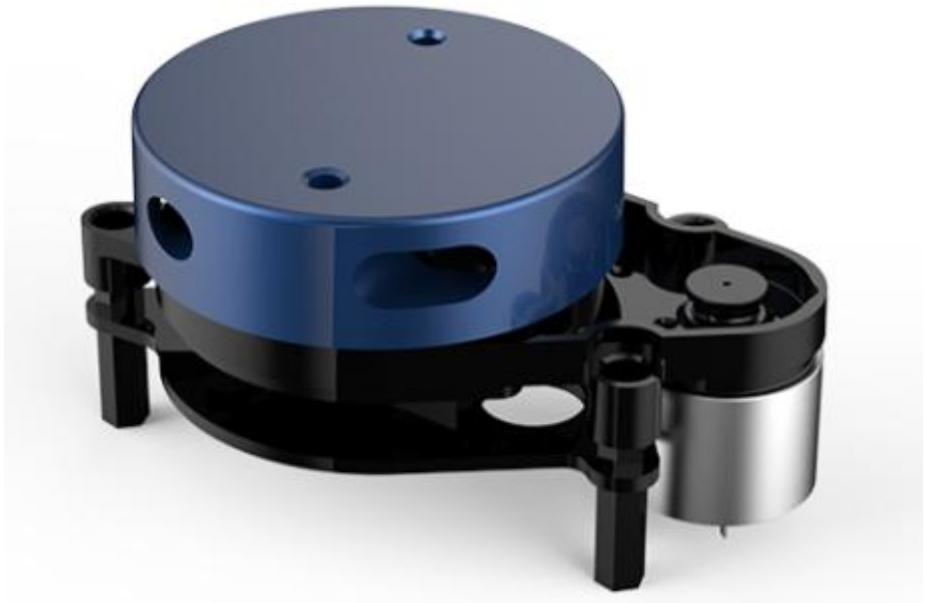


图 9-1 单线旋转式激光雷达外观

在激光雷达的头部由一个激光发射头和线性 CCD 接收头，发射头发射出的光属于波长大约在 1000nm 左右的红外光，肉眼是不可见的，当接收头收到反射的激光就可以计算出障碍物的距离。我们前面导航中使用的/scan 话题数据来源在真实机器中就是由激光雷达驱动和发布的。

除了激光雷达以外，另一个重要的传感器就是编码器。在移动机器人中我们需要实时的获取到机器人各个轮子的转速，通过转速根据机器人的运动学模型将每个轮子的速度转换成机器人的速度，通过对速度进行积分得到机器人行走的距离和角度，也就是里程计数据/odom。FishBot 采用的是 AB 电磁编码器，如图 9-2 所示，编码器位于电机的后部。

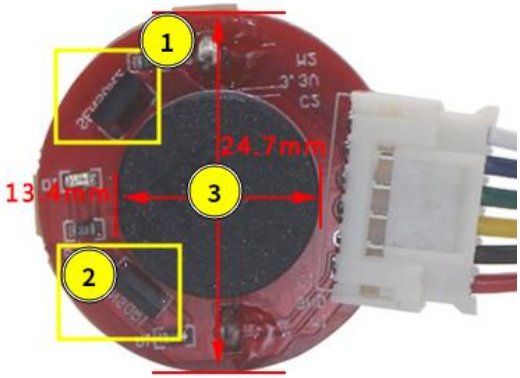


图 9-2 电磁编码器

如图 9-2 所示，电磁编码器是由 1 和 2 这两个霍尔传感器和圆形磁铁 3 共同组成的，该磁铁的磁性是间隔分布的，磁铁固定在电机的转子上，当电机转动时，带动磁铁转动，此时用于检测磁性的霍尔传感器就会检测到磁性的变化，从而就可以测量出电机在某段时间内转了多少圈即电机的转速。

了解完机器人感知部分的传感器，接着我们来了解控制部分的执行器。

9.1.2 机器人执行器

所谓执行器就是负责动的部件，在移动机器人上，最重要的一个执行器就是电机了。电机有很多分类，大类可以按照有刷无刷、直流交流来分类。FishBot 采用的是一个额定电压 12V 的 370 减速电机，额定转速为 130 转/分、额定电流 0.5A，转矩 600 克力厘米。电机尺寸和外观如图 9-3 所示。

图 9-3 带减速器的电机

需要注意的是 FishBot 采用的是减速电机，所谓减速电机指的是带减速器的电机。我们知道电机一般由定子和转子组成的，一般转速都比较快，但输出的力矩比较小，所以我们会给电机配备减速器，让转速降低，提高力矩，上图中长度为 31mm 的是电机，而长度为 L 的部分则是减速器。

了解完执行器我们来看一下机器人的决策部分硬件组成。

9.1.3 机器人决策系统

决策部分主要负责根据传感器数据以及任务要求来控制机器人的运动，决策部分的硬件一般使用处理能力较强的计算机，比如性能较强但体积更小的工控机，像树莓派和 Jetson Nano 一样的卡片式电脑等。

决策系统往往需要算力较大，所以硬件成本也是最贵的，为了节省成本，FishBot 采用了同时支持无线和有线连接的驱动控制板，让大家可以直接使用自己的电脑作为决策端，直接和感知以及控制系统进行通信。

除了感知、决策和控制部分，一个机器人要想动起来还需要一些硬件来打配合，比如电池、电源模块以及必要的支撑结构等。

9.2 单片机开发基础

做实体机器人无法避免和硬件系统打交道，机器人的传感器驱动和电机控制，都是在微型控制单元（Micro Control Unit，MCU）上编码完成的，MCU 又称单片微型计算机，简称就是单片机，而编写在单片机上的代码就叫单片机开发。FishBot 四驱机器人的驱动控制板采用的就是一款国产的单片机 ESP32S3，该单片机支持 WIFI、蓝牙等无线通信，通过该单片机以及其外围电路我们可以方便的读取传感器数据以及进行电机控制。

但在实现机器人控制系统前，我们需要先学习单片机的开发基础知识。

9.2.1 开发平台介绍与安装

计算机运行需要与之配套的操作系统，单片机也一样不仅需要硬件，还需要与之配套的软件才能运行，对于同一款单片机，支持的开发平台可以有很多种，比如 FishBot 驱动控制板采用的

ESP32S3 单片机，除了支持厂家提供的 ESP IDF 外还支持 Arduino（开源电子原型平台），因为 Arduino 相比之下更简单易用，本章将采用 Arduino 进行接下来的学习和移动机器人开发。

开发 Ardunio 我们可以采用 PlatformIO IDE 进行开发，该 IDE 支持多种类型的单片机，可以在 VSCode 中直接通过插件进行安装。PlatformIO IDE 主要使用 Python 编写，为了能够跨多个版本使用，PlatformIO IDE 在 Python 虚拟环境运行，所以我们需要先安装虚拟环境工具，命令如代码清单 9-1 所示。

代码清单9-1 安装虚拟环境工具

```
$ sudo apt install python3-venv
```

安装完成后打开 VS Code 的扩展商店，如图 9-4 所示，搜索安装 PlatformIO IDE。

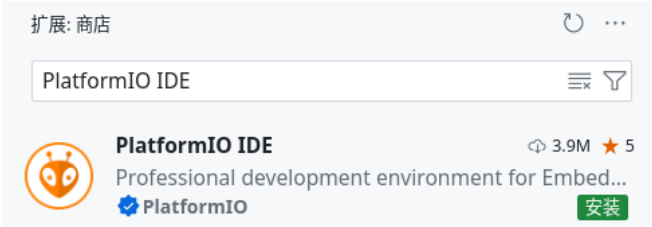


图 9-4 PlatformIO IDE 插件

安装完成后在 VS Code 的侧边就可以看到 PlatformIO IDE 的按钮，点击按钮就会执行 PlatformIO IDE 的首次初始化程序。如果初始化过慢可以手动进行初始化安装，命令如代码清单 9-2 所示。

代码清单9-2 在虚拟环境中安装 platformio

```
$ source ~/.platformio/penv/bin/activate  # 激活虚拟环境
$ pip install platformio -i https://pypi.tuna.tsinghua.edu.cn/simple # 安装
platformio 核心
```

安装完 PlatformIO 就可以来安装 ESP32S3 单片机的 Arduino 开发环境，第一步我们来下载一个示例工程，然后通过这个示例工程安装 ESP32S3 开发依赖。

代码清单9-3 下载示例工程并安装

```
$ git clone https://github.com/fishros/esp32s3_pio_template.git
example_helloword
$ cd esp32s3_pio_template/
$ pio run
```

9.2.2 第一个 HelloWorld 工程

Arduino 采用 C++作为编程语言进行单片机开发，一般单片机开发流程可以分为四步，分别是编写代码、编译工程、烧录二进制文件和运行测试。接下来我们来逐步完成第一个 HelloWorld

工程。

打开 src/main.cpp 接着编写如代码清单 9-4 所示的内容。

代码清单9-4 src/main.cpp

```
   #include <Arduino.h>

// setup 函数，启动时调用一次
void setup() {
  Serial.begin(115200);  // 设置串口波特率
}

// loop 函数，setup 后会被重复调用
void loop() {
  Serial.printf("Hello World!\n");  //打印 Hello World!
  delay(1000);  // 延时函数，单位 ms
}
```

为了简化开发流程，Arduino 提供了 setup 和 loop 两个声明好的函数，其中 setup 函数在启动时调用一次，一般初始化设置都在该函数中进行，之后是 loop 函数，该函数会在 setup 函数后循环调用。上面的代码中我们使用了串口 Serial 进行通信，串口是一种通信端口，通过专门的 USB 转串口芯片，我们可以在电脑上通过串口与单片机交换数据。所以代码清单 9-14 中首先在 setup 函数中初始化串口，并设置波特率为 115200，接着在 loop 函数中调用串口打印数据，并延时 1000 毫秒。

编写好代码，接着我们就可以编译工程。如图 9-9 所示，在 VS Code 的左下角，PlatformIO IDE 提供了几个快捷按钮用于编译和上传工程。点击编译按钮，看到如代码清单 9-5 所示的打印则表示编译成功。

图 9-9 PIO 提供的快捷按钮

代码清单9-5 工程构建结果

```
   ...
Found 33 compatible libraries
   Scanning dependencies...
   No dependencies
   Building in release mode
```

```
   Retrieving maximum program size .pio\build\esp32s3\firmware.elf
   Checking size .pio\build\esp32s3\firmware.elf
   Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
   RAM:   [=         ]   5.6% (used 18432 bytes from 327680 bytes)
   Flash: [=         ]  12.6% (used 263513 bytes from 2097152 bytes) *  终端将
被任务重用，按任意键关闭。
```

编译成功后打开.pio\build\esp32s3 文件夹，可以看到文件 firmware.bin 就是编译生成的二进制文件，下面我们就可以将这个二进制文件下载到单片机中。将开发板通过一个 USB 转 Type-C 线连接到电脑，首次使用串口设备有可能会出现占用和权限问题，在任意终端运行代码清单 9-6 中的命令后重启系统可以解决。

代码清单9-6 卸载占用及添加权限

```
$ sudo apt remove --purge brltty -y  # 卸载占用项目
$ sudo usermod -aG dialout `whoami` # 添加权限
```

重启系统后打开工程，接着点击上传按钮，就可以下载代码到开发板中了。下载完成后，可以使用串口查看工具来查看来自开发板的数据，在 VS Code 中搜索如图 9-10 所示的 Serial Monitor 插件并安装。

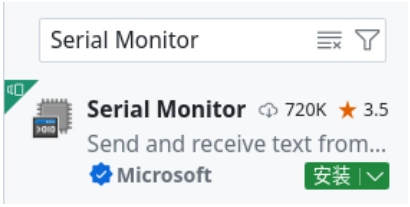


图 9-10 串口模拟器插件

安装完成后在 VS Code 的终端旁就会多出串行监视器一栏，在 Port 处选择/dev/ttyUSB 开头的设备，波特率 Baud rate 选择 115200，接着点击 Start Monitoring 就可以查看来自串口设备的数据了，完整配置如图 9-11 所示。

图 9-11 打开串口查看数据

可以看到，Hello World！已经在串行监视器中显示出来了，可以通过修改延时来改变打印速率，不过在下次下载代码前一定要记得关闭串行监视器，因为同一个串口设备同一时间只能由一个程序打开使用。

9.2.3 使用代码点亮 LED 灯

LED（Light Emitting Diode）是一种能够将电能转化为可见光的固态的半导体器件。在 FishBot 驱动控制板上也有一个可以使用代码控制的 LED 灯，它的原理图如图 9-12 所示。

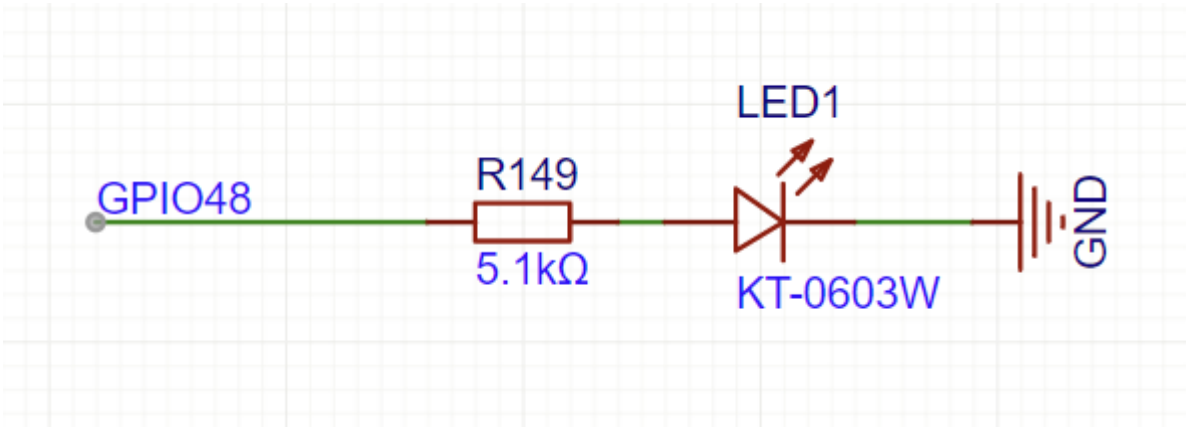


图 9-12 FishBot LED 原理图

R149 是一个 1K 欧姆的电阻，LED1 是一个白色的 LED 灯，右侧是 GND(0V)的电压源，左侧 GPIO48 是单片机的引脚，电流由电压高的地方流向电压低的地方，如果我们将 GPIO48 的电压设成 0V 时，此时电路两端电压相同，没有电流经过，此时 LED1 不工作。如果我们将 GPIO48 电压设置成 3.3V 时，此时左侧电压高，左侧电压低，电流从左侧 GPIO48 流过 LED1 流入 GND，此时 LED1 开始工作。所以我们可以通过 GPIO48 的电压高低来控制 LED1 灯的亮灭，这个就是点灯电路的原理。

复制 example01_helloword 工程 example02_led，接着在单独的 VS Code 窗口中打开该工程，然后在 src/main.cpp 中编写代码清单 9-7 中的内容。

代码清单9-7 src/main.cpp

```
   #include <Arduino.h>

void setup()
{
  pinMode(48, OUTPUT); // 设置 2 号引脚模式为 OUTPUT 模式
}

void loop()
{
  digitalWrite(48, LOW);  // 低电平，关闭 LED 灯
  delay(1000);            // 休眠 1000ms
  digitalWrite(48, HIGH); // 高电平，打开 LED 灯
```

```
  delay(1000);              // 休眠 1000ms
}
```

上面的代码有两个重要函数，第一个是 pinMode 函数，用于设置指定引脚的模式，OUTPUT 表示输出模式，对应的还有 INPUT 输入模型，第二个函数是 digitalWrite，用于设置指定引脚的电平状态，HIGH 表示高电平，LOW 表示低电平。

编译工程并下载到开发板，观察开发板上的 LED 灯，此时正在每间隔 1000 毫秒闪烁一次。

9.2.4 使用超声波测量距离

超声波测距传感器是一种使用超声波进行测量的传感器，可以可靠地检测部分或完全透明的物体，并进行精确的距离测量，一般的超声波传感器外观如图 9-13 所示。

图 9-13 超声波传感器

超声波上有一个发射头一个接收头，发射头负责发送超声波，超声波遇到障碍物就会反射回来，接收头就可以接收到信号，我们可以根据时间差和声速来计算障碍物的距离。

FishBot 采用的超声波有四个引脚，分别是 VCC、GND、TRIG 和 ECHO，其 VCC 和 GND 两个引脚负责供电，TRIG 引脚是发送引脚，我们给这个引脚输出高电平时就可以发射出超声波，当收到回波时 TRIG 引脚电平就会产生相应的电平变化，计算 TRIG 引脚高电平持续时间就是超声波在空中的飞行时间。我们按照这个逻辑来编写代码。

新建工程 example03_ultrasound，在 src/main.cpp 中编写如代码清单 9-8 所示的内容。

代码清单9-8 example03_ultrasound/src/main.cpp

```
#include <Arduino.h>
#define TRIG 21 // 设定发送引脚
#define ECHO 47 // 设置接收引脚

void setup() {
  Serial.begin(115200);
  pinMode(TRIG, OUTPUT); // 设置输出模式
  pinMode(ECHO, INPUT);  // 设置为输入状态
}

void loop() {
  // 产生一个 10us 的高脉冲去触发超声波
  digitalWrite(TRIG, HIGH);
  delayMicroseconds(10); // 延时 10 微秒
  digitalWrite(TRIG, LOW);

  double delta_time = pulseIn(ECHO, HIGH); // 检测高电平持续时间，注意返回值是微秒 us
  float detect_distance = delta_time * 0.0343 / 2; // 计算距离单位 cm，声速 0.0343 cm/us
  Serial.printf("distance=%f cm\n", detect_distance); // 打印距离
  delay(500);
}
```

因为 FishBot 超声波接口 TRIG 连接在单片机 21 引脚，ECHO 连接在 47 引脚，所以代码开头使用宏定义引脚编号，接着在 setup 函数中分别初始化了串口和引脚模式。然后在 loop 函数中，首先通过引脚电平设置函数 digitalWrite 和微秒延时函数 delayMicroseconds 在 TRIG 引脚上产生一个 10 微秒的脉冲触发超声波，然后利用 pulseIn 函数计算 ECHO 引脚上高电平持续的时间，接着根据声速和时间得到计算距离并打印出来。

编译工程并下载到开发板，打开串行监视器，如代码清单 9-9 所示，可以看到间隔 500 毫秒左右的距离数据输出。

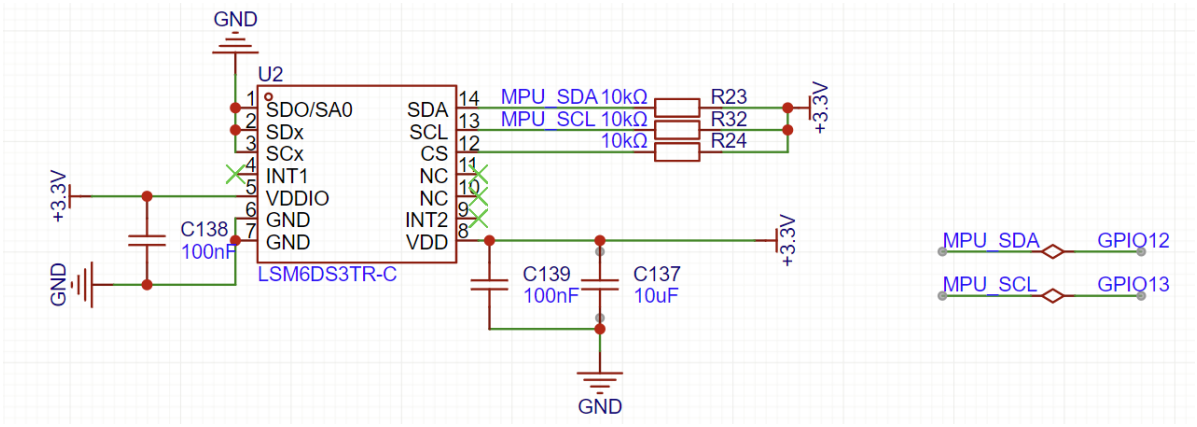
代码清单9-9 查看距离输出

```
distance=25.896500 cm
distance=25.879351 cm
distance=25.896500 cm
```

9.2.5 使用开源库驱动 IMU

IMU 即惯性测量单元，FishBot 采用一块 MPU6050 模块用于惯性测量，MPU6050 为全球首例

集成六轴传感器的运动处理组件，它通过 I2C 协议和单片机进行通信，相比上一节的超声波模块，驱动 MPU6050 更为复杂，不过我们不用从头编写，因为 Arduino 还支持通过第三方库来驱动硬件，本节我们就尝试用开源库驱动 FishBot 的 IMU 传感器。



新建工程 example04_imu , 接着编辑工程目录下的 platformio.ini 文件，添加依赖库 MPU6050_light，添加的命令如代码清单 9-10 所示。

代码清单9-10 添加 MPU6050 依赖库

```
   lib_deps =
    https://github.com/fishros/MPU6050_light.git
```

保存文件后，PlatformIO IDE 会自动下载开源库到目录.pio/libdeps/featheresp32，打开该目录就可以看到这个库的源代码，一般情况开源库都会为我们提供使用的示例代码，打开文件.pio/libdeps/esp32/MPU6050_light/examples/GetAngle/GetAngle.ino，把文件的内容复制到 src/main.cpp 中，接着修改串口波特率为 115200，修改 I2C 连接的引脚为 18 和 19，修改完成并添加注释的代码如代码清单 9-11 所示。

代码清单9-11 src/main.cpp

```
   #include "Wire.h"                // 引入 Wire 库，用于 I2C 通信
#include <MPU6050_light.h>     // 引入 MPU6050 库，用于与 MPU6050 传感器通信

MPU6050 mpu(Wire);              // 创建 MPU6050 对象，使用 Wire 对象进行通信
unsigned long timer = 0;        // 用于计时的变量

void setup() {
  Serial.begin(115200);         // 初始化串口通信，波特率为 115200
  Wire.begin(12,13);            // 初始化 I2C 总线，设置 SDA 引脚为 12，SCL 引脚为 13

  byte status = mpu.begin();    // 启动 MPU6050 传感器并获取状态
  Serial.print(F("MPU6050 status: "));
```

```
  Serial.println(status);
  while(status != 0) { }         // 如果无法连接到 MPU6050 传感器，停止一切

  Serial.println(F("Calculating offsets, do not move MPU6050"));
  delay(1000);
  mpu.calcOffsets();             // 计算陀螺仪和加速度计的偏移量
  Serial.println("Done!\n");
}

void loop() {
  mpu.update();                 // 更新 MPU6050 传感器的数据

  if ((millis() - timer) > 10) { // 每 10 毫秒打印一次数据
    Serial.print("X : ");
    Serial.print(mpu.getAngleX());  // 打印 X 轴的倾斜角度
    Serial.print("\tY : ");
    Serial.print(mpu.getAngleY());  // 打印 Y 轴的倾斜角度
    Serial.print("\tZ : ");
    Serial.println(mpu.getAngleZ());  // 打印 Z 轴的旋转角度
    timer = millis();
  }
}
```

代码清单 9-11 中首先使用 Wire 来初始化 MPU6050 类的对象 mpu，接着在 setup 函数中初始化串口和 I2C 总线 Wire，接着启动 MPU6050 传感器并校准，最后在 loop 函数中不断更新 MPU6050 数据并打印角度信息。

编译工程并下载到开发板，打开串行监视器，就可以看到如代码清单 9-12 所示角度信息的输出。

代码清单9-12 角度信息输出

```
X : -0.03       Y : 0.03        Z : -2.23
X : -0.04       Y : 0.03        Z : -2.23
X : -0.02       Y : 0.01        Z : -2.23
```

9.3 机器人控制系统实现

控制移动机器人运动，就是控制机器人上的电机转动，让电机转起来并不复杂，但要让电机按照要求的速度转起来就需要下更多功夫。FishBot 四驱的底盘上有四个连接电机的驱动轮，我们通过改变四个轮子的速度实现转弯和移动。

9.3.1 使用开源库驱动多路电机

让电机动起来只需要通电就行，电机的转速可以通过改变通电的时间来控制，因为电机需要

的电压和电流较大，无法直接接到单片机引脚上，所以需要额外的驱动电路放大来自单片机引脚的信号。FishBot 开发板采用 AT8236 芯片来实现电机的驱动，开发板上相关原理图如图 9-14 所示。

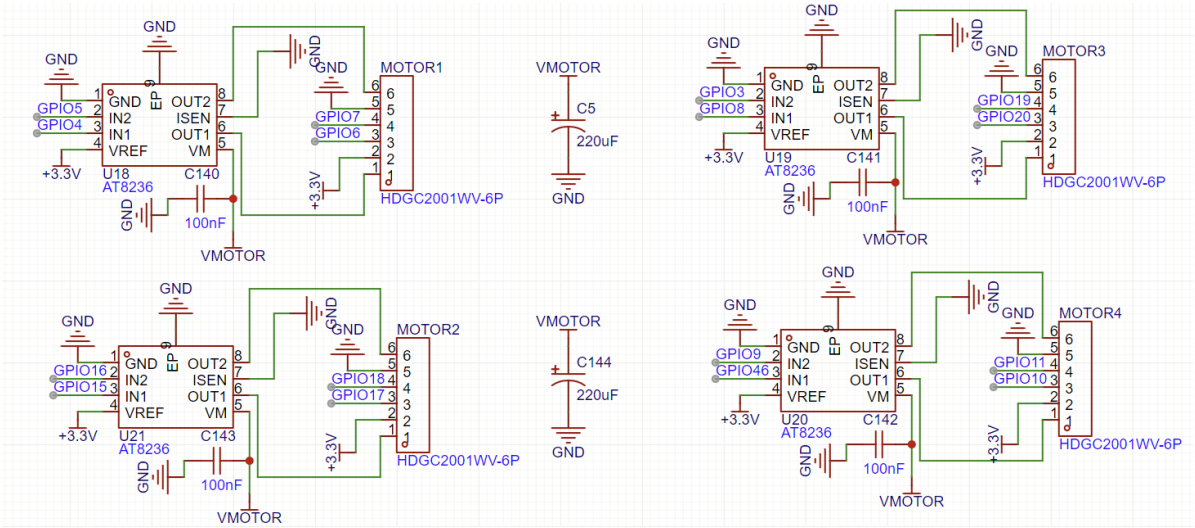


图 9-14 电机驱动原理图

以电路中 MOTOR1 为例，AT8236 可以将左侧 GPIO4/5 电路上的电信号放大到对应的输出引脚 OUT1/2 上，我们将电机接在输出引脚即可。

新建工程 fishbot_motion_control，接着修改 platformio.ini，添加依赖库 Esp32McpwmMotor，具体内容如代码清单 9-13 所示。

代码清单9-13 添加 Esp32McpwmMotor 依赖库

```
lib_deps =
 https://github.com/fishros/Esp32McpwmMotor.git
```

这里我们使用开源库 Esp32McpwmMotor 来驱动电机，该库可以同时控制 6 个直流电机，对于只有 2 个驱动轮的 FishBot 来说绰绰有余。 接着打开 src/main.cpp，编写如代码清单 9-14 所示的内容。

代码清单9-14 fishbot_motion_control/src/main.cpp

```
#include <Arduino.h>
#include <Esp32McpwmMotor.h>

Esp32McpwmMotor motor; // 创建一个名为 motor 的对象，用于控制电机

void setup()
{
    motor.attachMotor(0, 5, 4);
    motor.attachMotor(1, 15, 16);
```

```
    motor.attachMotor(2, 3, 8);
    motor.attachMotor(3, 46, 9);
}

void loop()
{
    motor.updateMotorSpeed(0, 70);  // 设置电机 0 的速度(占空比)为正 70%
    motor.updateMotorSpeed(1, 70);  // 设置电机 1 的速度(占空比)为正 70%
    motor.updateMotorSpeed(2, 70);  // 设置电机 2 的速度(占空比)为正 70%
    motor.updateMotorSpeed(3, 70);  // 设置电机 3 的速度(占空比)为正 70%
    delay(2000); // 延迟两秒

    motor.updateMotorSpeed(0, -70); // 设置电机 0 的速度(占空比)为负 70%
    motor.updateMotorSpeed(1, -70); // 设置电机 1 的速度(占空比)为负 70%
    motor.updateMotorSpeed(2, -70); // 设置电机 2 的速度(占空比)为负 70%
    motor.updateMotorSpeed(3, -70); // 设置电机 3 的速度(占空比)为负 70%
    delay(2000); // 延迟两秒
}
```

代码清单 9-14 是用于控制两个电机进行正反转的程序，上面的代码主要使用了两个方法，第一个方法 attachMotor()用于连接电机，该方法的第一个参数是电机编号，后面两个是电机的引脚。第二个方法是 updateMotorSpeed()用于更新电机速度，第一个参数是电机编号，第二个参数是电机速度百分比。

将代码编译并下载到开发板并打开电池开关，就可以看到电机已经转起来了。

9.3.2 电机速度测量与转换

通过 9.1.1 节的学习我们知道，FishBot 采用的电机都安装了两个霍尔传感器，当电机转动时霍尔传感器就会根据磁性的有无产生高低电平的变化，我们将这种电平从低到高再到低的过程称作一个脉冲（Pluse）。因为有减速器的存在，当减速器的输出轴，也就是连接轮子的轴转动了一圈，实际电机转动远不止一圈，产生的脉冲数则更多。

单片机可以检测到电平变化，从而得到脉冲数，要将脉冲数转换为轮子实际行走的距离，就需要测量轮子转一圈所产生的脉冲数。下面我们就尝试使用开源库来驱动编码器，计算轮子转一圈的脉冲数，然后计算轮子转速。

修改文件 fishbot_motion_control/platformio.ini，添加依赖库 Esp32PcntEncoder，添加的内容如代码清单 9-15 所示。

代码清单9-15 fishbot_motion_control/platformio.ini

```
lib_deps =
 https://github.com/fishros/Esp32McpwmMotor.git
 https://github.com/fishros/Esp32PcntEncoder.git
```

Esp32PcntEncoder 是基于 ESP32 的脉冲计算外设编写的脉冲计算开源库，使用非常简单。修改 fishbot_motion_control/src/main.cpp，内容如代码清单 9-16 所示。

代码清单9-16 fishbot_motion_control/src/main.cpp

```
#include <Arduino.h>
#include <Esp32PcntEncoder.h>
Esp32PcntEncoder encoders[4]; // 创建一个数组用于存储编码器

void setup()
{
    // 1.初始化串口
    Serial.begin(115200); // 初始化串口通信，设置通信速率为 115200
    // 2.设置编码器
    encoders[0].init(0, 6, 7);
    encoders[1].init(1, 18, 17);
    encoders[2].init(2, 20, 19);
    encoders[3].init(3, 11, 10);
}

void loop()
{
    delay(10); // 等待 10 毫秒
    // 读取并打印四个编码器的计数器数值
    Serial.printf("tick1=%d,tick2=%d,tick3=%d,tick4=%d\n",
encoders[0].getTicks(), encoders[1].getTicks(), encoders[2].getTicks(),
encoders[3].getTicks());
}
```

代码清单 9-16 中引入了 Esp32PcntEncoder 库的头文件，接着创建了两个对象数组，并在 setup 函数中使用 init 方法初始化编码器，init 方法的第一个参数是编码器编号，后面两个是编码器的引脚编号，引脚编号可以从 FishBot 驱动控制板原理图中查询。最后在 loop 函数中，打印了编码器对脉冲的计数值。

下载代码到开发板，接着打开串行监视器，打开对应串口后观察打印信息，尝试手动转动轮子，观察打印数据的变化。FishBot 采用的电机直径为 65 毫米，乘上圆周率后就可以得到转动一圈行走的距离，测量出电机转动一周的脉冲数后就可以计算出单个脉冲数对应的距离。

手动将轮子转动 10 圈，观察终端脉冲数变化，如代码清单 9-17 所示，这里测量出 10 圈的脉

冲数是 19419。

代码清单9-17 轮子转动 10 圈的脉冲值

```
tick1=0,tick2=19422,tick3=0,tick4=0
```

我们可以计算出一圈的脉冲数约等于 1942，一个脉冲对应的距离计算结果如代码清单 9-18 所示。

代码清单9-18 单个脉冲的轮子前进距离

```
0.10353≈65*3.1415926/1942
```

有了参数，再次编写代码将脉冲数转换为速度，完整代码如代码清单 10-19 所示。

代码清单9-19 fishbot_motion_control/src/main.cpp

```
#include <Arduino.h>
#include <Esp32McpwmMotor.h>
#include <Esp32PcntEncoder.h>
Esp32PcntEncoder encoders[4]; // 创建一个数组用于存储四个编码器
Esp32McpwmMotor motor; // 创建一个名为 motor 的对象，用于控制电机

int64_t last_ticks[4];    // 记录上一次读取的计数器数值
int32_t delta_ticks[4];   // 记录两次读取之间的计数器差值
int64_t last_update_time; // 记录上一次更新时间
float current_speeds[4];  // 记录四个电机的速度

void setup()
{
    // 1.初始化串口
    Serial.begin(115200); // 初始化串口通信，设置通信速率为 115200
    motor.attachMotor(0, 5, 4);
    motor.attachMotor(1, 15, 16);
    motor.attachMotor(2, 3, 8);
    motor.attachMotor(3, 46, 9);

    // 2.设置编码器
    encoders[0].init(0, 6, 7);
    encoders[1].init(1, 18, 17);
    encoders[2].init(2, 20, 19);
    encoders[3].init(3, 11, 10);

    // 设置电机速度
    for (int i = 0; i < 4; i++)
    {
        motor.updateMotorSpeed(i, 70);
```

```
        }

    }

    void loop() {
        delay(10); // 等待 10 毫秒
        // 计算时间差
        uint64_t dt = millis() - last_update_time;

        // 使用 for 循环处理所有电机
        for (int i = 0; i < 4; i++) {
            // 计算编码器差值
            delta_ticks[i] = encoders[i].getTicks() - last_ticks[i];
            // 距离比时间获取速度 单位 mm/ms 相当于 m/s
            current_speeds[i] = float(delta_ticks[i] * 0.1051566) / dt;
            // 更新上一次的计数器数值
            last_ticks[i] = encoders[i].getTicks();
        }

        // 更新数据
        last_update_time = millis(); // 更新上一次更新时间

        // 打印所有电机数据
        Serial.printf("speeds: 1=%fm/s 2=%fm/s 3=%fm/s 4=%fm/s\n",
                      current_speeds[0], current_speeds[1], current_speeds[2],
current_speeds[3]);
    }
```

代码清单 9-19 中，通过 getTicks()获取编码器数值，通过 millis()函数获取当前时间。接着计算编码器数值差，然后乘上参数 0.1051566 获取距离，最后除上时间差就得到了单位为 mm/ms 的速度数据，换算单位后刚好是 m/s，同时为了方便查看，我们利用 Esp32McpwmMotor 将两个电机的速度设置为 70 %。

下载代码，打开串口监视器，如代码清单 9-20 所示，可以看到电机的速度不断的打印出来了。

代码清单9-20 电机速度打印

```
    speeds: 1=0.231345m/s 2=0.262892m/s 3=0.231345m/s 4=0.262892m/s
    speeds: 1=0.252376m/s 2=0.262892m/s 3=0.241860m/s 4=0.252376m/s
```

9.3.3 使用 PID 控制轮子转速

上一节我们通过编码器获取到了两个电机的实时速度，但你会发现即使设置相同的速度，两

个电机实际转速并不一致，如果要保持相同的转速则要根据电机来动态调节设置的速度值，我们可以使用 PID 控制器来实现这一功能。

PID 控制器是一种广泛应用于工业控制、自动化控制等领域的控制算法，其名称来源于“比例-积分-微分”三个控制器参数，即 Proportional（比例）、Integral（积分）、Derivative（微分）。PID 控制器的基本原理是通过测量目标系统的反馈信号和期望输出信号之间的误差，根据一定的数学模型计算出控制信号，使目标系统能够稳定地达到期望输出。对于轮子速度控制来说，PID 控制器会根据目标速度和实际速度的差值，计算出让实际速度更接近目标速度的速度值。PID 控制器的计算公式如下所示。

$$\text{Output} = K_p \cdot \text{Error} + K_i \cdot \int \text{Error}\ \text{dt} + K_d \cdot \frac{\text{d(Error)}}{\text{dt}}$$

其中 Kp、Ki 和 Kd 是参数，Error 是误差，其值等于目标速值减当前测量值，与 Ki 相乘的部分是 Error 的积分，就是把所有的 Error 相加，为了防止加到非常离谱的数值，一般都会设置一个积分上限，与 Kd 相乘的部分是误差的微分，我们取当前误差和上次误差之差。公式很抽象，我们来把它变成具体的代码。

在 fishbot_motion_control/lib 目录下新建 PidController，接着在目录下新建 PidController.h 和 PidController.cpp，接着在 PidController.h 中编写如代码清单 9-21 所示的内容。

代码清单9-21 fishbot_motion_control/lib/PidController/PidController.h

```
   #ifndef __PIDCONTROLLER_H__ // 如果没有定义__PIDCONTROLLER_H__
#define __PIDCONTROLLER_H__ // 定义__PIDCONTROLLER_H__

class PidController
{ // 定义一个 PID 控制器类
public:
    PidController() = default;                    // 默认构造函数
    PidController(float kp, float ki, float kd); // 构造函数，传入 kp、ki、kd

private:
    float target_;       // 目标值
    float out_min_;      // 输出下限
    float out_max_;      // 输出上限
    float kp_;           // 比例系数
    float ki_;           // 积分系数
    float kd_;           // 微分系数
    // pid
    float error_sum_;            // 误差累积和
```

```
    float derror_;               // 误差变化率
    float error_last_;           // 上一次误差
    float error_pre_;            // 上上次误差
    float intergral_up_ = 2500; // 积分上限

public:
    float update(float current);                    // 提供当前值返回下次输出值
    void update_target(float target);               // 更新目标值
    void update_pid(float kp, float ki, float kd); // 更新 PID 系数
    void reset();                                   // 重置 PID 控制器
    void out_limit(float out_min, float out_max);  // 设置输出限制
};

#endif // __PIDCONTROLLER_H__ // 结束条件
```

代码清单 9-21 中定义了一个 PID 控制器类，其中成员函数 update 用于根据当前值计算下依次输出值，update_target 函数用于设置目标值。有了头文件，接着我们来实现各个函数，编写 PidController.cpp 如代码清单 9-22 所示。

代码清单9-22 fishbot_motion_control/lib/PidController/PidController.cpp

```
   #include "PidController.h"
#include "Arduino.h"

PidController::PidController(float kp, float ki, float kd)
{
    reset(); // 初始化控制器
    update_pid(kp, ki, kd); // 更新 PID 参数
}

float PidController::update(float current)
{
    // 计算误差及其变化率
    float error = target_ - current; // 计算误差
    derror_ = error_last_ - error; // 计算误差变化率
    error_last_ = error;  // 更新上一次误差为当前误差

    // 计算积分项并进行积分限制
    error_sum_ += error;
    if (error_sum_ > intergral_up_)
        error_sum_ = intergral_up_;
    if (error_sum_ < -1 * intergral_up_)
        error_sum_ = -1 * intergral_up_;

    // 计算控制输出值
```

```
    float output = kp_ * error + ki_ * error_sum_ + kd_ * derror_;

    // 控制输出限幅
    if (output > out_max_)
        output = out_max_;
    if (output < out_min_)
        output = out_min_;

    return output;
}

void PidController::update_target(float target)
{
    target_ = target; // 更新控制目标值
}

void PidController::update_pid(float kp, float ki, float kd)
{
    reset(); // 重置控制器状态
    kp_ = kp; // 更新比例项系数
    ki_ = ki; // 更新积分项系数
    kd_ = kd; // 更新微分项系数
}

void PidController::reset()
{
    // 重置控制器状态
    target_ = 0.0f; // 控制目标值
    out_min_ = 0.0f; // 控制输出最小值
    out_max_ = 0.0f; // 控制输出最大值
    kp_ = 0.0f; // 比例项系数
    ki_ = 0.0f; // 积分项系数
    kd_ = 0.0f; // 微分项系数
    error_sum_ = 0.0f; // 误差累计值
    derror_ = 0.0f; // 误差变化率
    error_last_ = 0.0f; // 上一次的误差值
}

void PidController::out_limit(float out_min, float out_max)
{
    out_min_ = out_min; // 控制输出最小值
    out_max_ = out_max; // 控制输出最大值
}
```

代码清单 9-22 中除了 update 方法，其他方法实现都较简单，update 方法中首先根据目标值和

当前值计算误差，接着通过上一次的误差来计算误差变化，接着对误差进行累积，如果误差总和大于积分上限则限制在积分上限内，接着根据 PID 公式计算输出，最后将输出限制在最大和最小输出之间并返回。

接着我们修改 src/main.cpp，引入 PID 控制器到代码中，完成后的代码如代码清单 9-23 所示。

代码清单9-23 fishbot_motion_control/src/main.cpp

```
 #include <Arduino.h>
#include <Esp32McpwmMotor.h>
#include <Esp32PcntEncoder.h>
#include <PidController.h> // 引入 PID 控制器头文件

Esp32PcntEncoder encoders[4];    // 创建一个数组用于存储四个编码器
Esp32McpwmMotor motor;           // 创建一个名为 motor 的对象，用于控制电机
PidController pid_controller[4]; // 创建 PID 控制器对象数组

int64_t last_ticks[4];    // 记录上一次读取的计数器数值
int32_t delta_ticks[4];   // 记录两次读取之间的计数器差值
int64_t last_update_time; // 记录上一次更新时间
float current_speeds[4];  // 记录四个电机的速度

void motorSpeedControl()
{
    // 计算时间差
    uint64_t dt = millis() - last_update_time;

    // 使用循环处理所有 4 个电机
    for (int i = 0; i < 4; i++)
    {
        // 计算编码器差值
        delta_ticks[i] = encoders[i].getTicks() - last_ticks[i];
        // 距离比时间获取速度单位 mm/ms 乘 1000 转换为 mm/s，方便 PID 计算
        current_speeds[i] = float(delta_ticks[i] * 0.1051566) / dt * 1000;
        // 更新上一次计数器数值
        last_ticks[i] = encoders[i].getTicks();
        // 根据当前速度，更新电机速度值
        motor.updateMotorSpeed(i,
pid_controller[i].update(current_speeds[i]));
    }

    // 更新上一次更新时间
    last_update_time = millis();
```

```
        // 打印所有电机的速度数据
        Serial.printf("speed1=%f mm/s, speed2=%f mm/s, speed3=%f mm/s,
speed4=%f mm/s\n",
                      current_speeds[0], current_speeds[1], current_speeds[2],
current_speeds[3]);
    }

    void setup()
    {
        // 1.初始化串口
        Serial.begin(115200); // 初始化串口通信，设置通信速率为 115200
        motor.attachMotor(0, 5, 4);
        motor.attachMotor(1, 15, 16);
        motor.attachMotor(2, 3, 8);
        motor.attachMotor(3, 46, 9);

        // 2.设置编码器
        encoders[0].init(0, 6, 7);
        encoders[1].init(1, 18, 17);
        encoders[2].init(2, 20, 19);
        encoders[3].init(3, 11, 10);
        // 初始化 pid 控制器
        for (int i = 0; i < 4; i++)
        {
            pid_controller[i].update_pid(0.625, 0.125, 0.0);
            pid_controller[i].out_limit(-100, 100);
        }
        // 设置电机速度
        for (int i = 0; i < 4; i++)
        {
            // 初始化目标速度，单位 mm/s，使用毫米防止浮点运算丢失精度
            pid_controller[i].update_target(100);
        }
    }

    void loop()
    {
        delay(10);            // 等待 10 毫秒
        motorSpeedControl(); // 调用速度控制函数
    }
```

代码清单 9-23 中，引入了 PidController 头文件，同时在 setup 函数中对 PID 控制器进行初始

化并设置目标速度，为了方便计算，目标速度采用毫米每秒为单位。在 motorSpeedControl 函数中，首先计算电机速度，这次我们乘上 1000 将单位转换成毫米每秒，最后调用 pid_controler 的 update 方法，该方法根据当前速度和目标值计算新的输出值，最后在 loop 函数中，间隔 10 毫秒调用一次 motorSpeedControl()函数进行速度计算和 PID 控制。

下载代码，打开串口监视器，此时观察串口数据如代码清单 9-24 所示，可以看到速度基本稳定在 100 mm/s ,和设定的速度非常接近。

代码清单9-24 闭环控制速度打印

```
speed1=105.156601 mm/s, speed2=105.156601 mm/s, speed3=105.156601 mm/s,
speed4=105.156601 mm/s
speed1=105.156601 mm/s, speed2=94.640938 mm/s, speed3=94.640938 mm/s,
speed4=94.640938 mm/s
speed1=94.640938 mm/s, speed2=94.640938 mm/s, speed3=94.640938 mm/s,
speed4=105.156601 mm/s
speed1=94.640938 mm/s, speed2=105.156601 mm/s, speed3=105.156601 mm/s,
speed4=94.640938 mm/s
```

现在我们可以控制机器人的两个轮子按照设定速度转动，但是轮子的速度要和机器人的速度进行转换还需要经过运动学正逆解才行。

9.3.4 运动学正逆解实现

1. 四轮差速运动学正逆解

在第 7 章机器人导航和键盘控制时，我们下发的是线速度和角速度信息，并不会直接对轮子转速进行直接控制，这就需要我们把角速度和线速度转换成机器人四个轮子的速度，我们把这一过程称为运动学逆解，反之通过轮子的实际转速计算机器人的线速度和角速度的过程就是运动学正解。接着我们对运动学正逆解进行推导。

现在闭上眼睛，把机器人左侧两个轮子当成一个来看，右边的两个轮子也当成一个，左边这个轮子的速度=（左上轮子速度+左下轮子速度）/2，右边同理，这样我们就把四轮差速模型简化成了两轮差速模型，所以下面来看两轮差速模型的运动学推导过程。

假设机器人在一小段时间t内，它的左右轮子线速度为v_l和v_r，两轮之间的安装间距l，求机器人的线速度v，角速度ω。

机器人的线速度方向和轮子转动方向始终保持一致，所以机器人的线速度为左右轮线速度的平均值，即$v = (v_l + v_r)/2$，我们知道线速度和角速度的关系是$v = \omega * r$，根据图 9-15 可知$l = r_r - r_l = v_r/\omega_r - v_l/\omega_l$，机器人移动时各轮角速度相同，所以$\omega_l = \omega_r$，结合前面的公式可以求出$\omega =$

$(v_r - v_l)/l$。至此我们得出了左右轮速度和机器人速度之间的关系如图 9-15 所示。

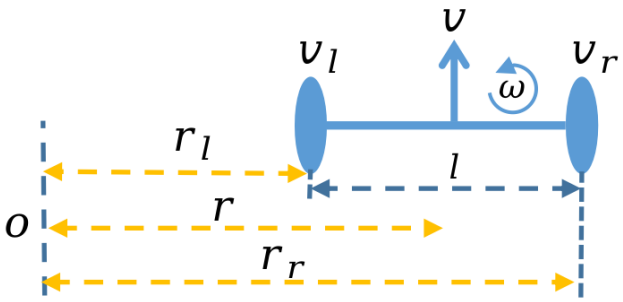


图 9-15 两轮差速运动示意图

左右轮速度和机器人速度的关系为：

$$v = (v_l + v_r)/2$$

$$\omega = (v_r - v_l)/l$$

也就是两轮差速运动学正解的公式。反之，当我们已知目标机器人速度，利用正解公式可以得出左右轮子的线速度。

$$v_l = v - \omega l/2$$

$$v_r = v + \omega l/2$$

对于四轮差速模型 vl = (v1+v3)/2 vr=(v2+v4)/2

有了运动学正逆解公式，接着我们就可以根据公式编写代码。在 fishbot_motion_control/lib 下新建目录 Kinematics，接着在目录下新建 Kinematics.h，然后在头文件中编写如代码清单 9-25 所示的内容。

代码清单9-25 fishbot_motion_control/lib/Kinematics/Kinematics.h

```
#ifndef __KINEMATICS_H__ /* 防止头文件被多次包含 */
#define __KINEMATICS_H__

#include <Arduino.h> /* 包含 Arduino 核心库*/

// 定义一个结构体用于存储电机参数
typedef struct
{
    float per_pulse_distance;  /* 单个脉冲对应轮子前进距离 */
    int16_t motor_speed;       /* 当前电机速度 mm/s，计算时使用*/
    int64_t last_encoder_tick; /* 上次电机的编码器读数*/
} motor_param_t;

/* 定义一个类用于处理机器人运动学 */
```

```
class Kinematics
{
public:
    /* 构造函数，默认实现 */
    Kinematics() = default;
    /* 析构函数，默认实现 */
    ~Kinematics() = default;

    /* 设置电机参数，包括编号和每个脉冲对应的轮子前进距离 */
    void set_motor_param(uint8_t id, float per_pulse_distance);
    /* 设置轮子间距*/
    void set_wheel_distance(float wheel_distance);
    /* 逆运动学计算，将线速度和角速度转换为左右轮的速度*/
    void kinematic_inverse(float linear_x_speed, float linear_y_speed,
float angular_speed,float &out_wheel1_speed, float &out_wheel2_speed, float
&out_wheel3_speed, float &out_wheel4_speed);
    /* 正运动学计算，将左右轮的速度转换为线速度和角速度*/
    void kinematic_forward(float wheel1_speed, float wheel2_speed, float
wheel3_speed, float wheel4_speed, float &linear_x_speed,float
&linear_y_speed, float &angular_speed);
    /* 更新电机速度和编码器数据*/
    void update_motor_speed(uint64_t current_time, int32_t motor_tick1,
int32_t motor_tick2, int32_t motor_tick3, int32_t motor_tick4);
    /* 获取电机速度*/
    int16_t get_motor_speed(uint8_t id);

private:
    motor_param_t motor_param_[4]; /* 存储四个电机的参数*/
    uint64_t last_update_time;     /* 上次更新数据的时间，单位 ms*/
    float wheel_distance_;         /* 轮子间距*/
};

#endif // __KINEMATICS_H__
```

代码清单 9-25 中我们创建了一个电机参数结构体 motor_param_t，将电机对应的脉冲前进距离、电机速度和上一次编码器读数放到该结构体中。接着定义了一个运动学类 Kinematics，在该类成员函数定义了 kinematic_forward 和 kinematic_inverse 方法，用于运动学正逆解，定义方法 update_motor_speed 和 get_motor_speed 用于计算和获取电机速度，定义方法 set_wheel_distance 和 set_motor_param 用于设置机器人轮子间距和电机单脉冲距离参数。成员变量部分，声明了电机参数数组、轮子间距和上次速度数据更新时间。

接着根据方法声明，我们逐一实现各个方法，在 fishbot_motion_control/lib/Kinematics/目录下

新建 Kinematics.cpp，接着编写如代码清单 9-26 所示的代码。

代码清单9-26 Kinematics.cpp 中设置和获取方法实现

```
   #include "Kinematics.h"
void Kinematics::set_motor_param(uint8_t id, float per_pulse_distance) {
  motor_param_[id].per_pulse_distance = per_pulse_distance;  /* 电机每个脉冲前进
距离*/
}
void Kinematics::set_wheel_distance(float wheel_distance) {
  wheel_distance_ = wheel_distance;
}
// 传入电机的编号 id 返回该编号电机的速度
int16_t Kinematics::get_motor_speed(uint8_t id) {
  return motor_param_[id].motor_speed;
}
```

代码清单 9-26 中实现了三个参数设置和获取方法，内容较为简单，只是对相对应的参数进行赋值即可。接着我们来编写电机速度更新方法 update_motor_speed，继续在 Kinematics.cpp 下添加如代码清单 9-27 所示的代码。

代码清单9-27 Kinematics.cpp 中电机速度计算方法实现

```
   /**
 * @brief 更新电机速度和编码器数据
 * @param current_time 当前时间（单位：毫秒）
 * @param left_tick 左轮编码器读数
 * @param right_tick 右轮编码器读数
 */
void Kinematics::update_motor_speed(uint64_t current_time, int32_t
motor_tick1, int32_t motor_tick2, int32_t motor_tick3, int32_t motor_tick4) {
  // 计算出自上次更新以来经过的时间 dt
  uint32_t dt = current_time - last_update_time;
  last_update_time = current_time;

  // 计算电机 1 和电机 2 的编码器读数变化量 dtick1 和 dtick2。
  int32_t dtick1 = motor_tick1 - motor_param_[0].last_encoder_tick;
  int32_t dtick2 = motor_tick2 - motor_param_[1].last_encoder_tick;
   int32_t dtick3 = motor_tick3 - motor_param_[2].last_encoder_tick;
   int32_t dtick4 = motor_tick4 - motor_param_[3].last_encoder_tick;
   motor_param_[0].last_encoder_tick = motor_tick1;
   motor_param_[1].last_encoder_tick = motor_tick2;
    motor_param_[2].last_encoder_tick = motor_tick3;
    motor_param_[3].last_encoder_tick = motor_tick4;
```

```
  // 轮子速度计算
  motor_param_[0].motor_speed =
      float(dtick1 * motor_param_[0].per_pulse_distance) / dt * 1000;
  motor_param_[1].motor_speed =
      float(dtick2 * motor_param_[1].per_pulse_distance) / dt * 1000;
  motor_param_[2].motor_speed =
      float(dtick2 * motor_param_[2].per_pulse_distance) / dt * 1000;
  motor_param_[3].motor_speed =
      float(dtick2 * motor_param_[3].per_pulse_distance) / dt * 1000;
}
```

电机速度更新方法接收当前时间和左右轮编码器计数值为参数，在函数内先计算时间差和编码器计数差，最后根据脉冲距离比计算出速度，单位使用 mm/s 。接着我们来完善运动学正逆解方法，继续在 Kinematics.cpp 添加如代码清单 10-28 所示的代码。

代码清单9-28 Kinematics.cpp 中运动学正逆解实现

```
   /**
 * @brief 正运动学计算，将左右轮的速度转换为线速度和角速度
 * @param left_speed 左轮速度（单位：毫米/秒）
 * @param right_speed 右轮速度（单位：毫米/秒）
 * @param[out] out_linear_speed 线速度（单位：毫米/秒）
 * @param[out] out_angle_speed 角速度（单位：弧度/秒）
 */
void Kinematics::kinematic_forward(float wheel1_speed, float wheel2_speed,
float wheel3_speed, float wheel4_speed,float &linear_x_speed, float
&linear_y_speed, float &angular_speed) {
     linear_x_speed = (wheel1_speed + wheel2_speed + wheel3_speed +
wheel4_speed) / 4.0f;
     linear_y_speed = 0.0f;
     angular_speed = (wheel2_speed + wheel4_speed - wheel1_speed -
wheel3_speed) / (2.0f * wheel_distance_);
}

/**
 * @brief 逆运动学计算，将线速度和角速度转换为左右轮的速度
 * @param linear_speed 线速度（单位：毫米/秒）
 * @param angle_speed 角速度（单位：弧度/秒）
 * @param[out] out_left_speed 左轮速度（单位：毫米/秒）
 * @param[out] out_right_speed 右轮速度（单位：毫米/秒）
 */
void Kinematics::kinematic_inverse(float linear_x_speed, float
linear_y_speed, float angular_speed, float &out_wheel_speed1, float
&out_wheel_speed2, float &out_wheel_speed3, float &out_wheel_speed4) {
   out_wheel_speed1 = linear_x_speed - (angular_speed * wheel_distance_ / 2);
```

```
out_wheel_speed3 = linear_x_speed - (angular_speed * wheel_distance_ / 2);
out_wheel_speed2 = linear_x_speed + (angular_speed * wheel_distance_ / 2);
out_wheel_speed4 = linear_x_speed + (angular_speed * wheel_distance_ / 2);
}
```

正逆解方法我们完全按照推导的公式编写而成，不做过多解释，接着我们来修改 main.cpp ,调用 Kinematics 类完成速度计算以及正逆解，完成后 main.cpp 代码如代码清单 9-29 所示。

代码清单9-29 fishbot_motion_control/src/main.cpp

```
#include <Arduino.h>
#include <Esp32McpwmMotor.h>
#include <Esp32PcntEncoder.h>
#include <PidController.h> // 引入 PID 控制器头文件
#include <Kinematics.h>    // 引入运动学头文件
Kinematics kinematics;
Esp32PcntEncoder encoders[4];    // 创建一个数组用于存储四个编码器
Esp32McpwmMotor motor;           // 创建一个名为 motor 的对象，用于控制电机
PidController pid_controller[4]; // 创建 PID 控制器对象数组

int64_t last_ticks[4];    // 记录上一次读取的计数器数值
int32_t delta_ticks[4];   // 记录两次读取之间的计数器差值
int64_t last_update_time; // 记录上一次更新时间
float current_speeds[4];  // 记录四个电机的速度

float target_linear_x_speed = 50.0; // 目标 X 方向线速度 毫米每秒
float target_linear_y_speed = 0.0;  // 目标 Y 方向线速度 毫米每秒
float target_angular_speed = 0.1f;  // 目标角速度 弧度每秒
float out_speed[4];                 // 用于存储运动学逆解后的速度

void motorSpeedControl()
{
    // 计算时间差
    uint64_t dt = millis() - last_update_time;

    // 使用循环处理所有 4 个电机
    for (int i = 0; i < 4; i++)
    {
        // 计算编码器差值
        delta_ticks[i] = encoders[i].getTicks() - last_ticks[i];
        // 距离比时间获取速度单位 mm/ms 乘 1000 转换为 mm/s，方便 PID 计算
        current_speeds[i] = float(delta_ticks[i] * 0.1051566) / dt * 1000;
        // 更新上一次计数器数值
        last_ticks[i] = encoders[i].getTicks();
        // 根据当前速度，更新电机速度值
```

```
            motor.updateMotorSpeed(i,
pid_controller[i].update(current_speeds[i]));
        }

        // 更新上一次更新时间
        last_update_time = millis();

        // 打印所有电机的速度数据
        Serial.printf("speed1=%f mm/s, speed2=%f mm/s, speed3=%f mm/s,
speed4=%f mm/s\n",
                      current_speeds[0], current_speeds[1], current_speeds[2],
current_speeds[3]);
    }

    void setup()
    {
        // 1.初始化串口
        Serial.begin(115200); // 初始化串口通信，设置通信速率为 115200
        motor.attachMotor(0, 5, 4);
        motor.attachMotor(1, 15, 16);
        motor.attachMotor(2, 3, 8);
        motor.attachMotor(3, 46, 9);

        // 2.设置编码器
        encoders[0].init(0, 6, 7);
        encoders[1].init(1, 18, 17);
        encoders[2].init(2, 20, 19);
        encoders[3].init(3, 11, 10);
        // 初始化 pid 控制器
        for (int i = 0; i < 4; i++)
        {
            pid_controller[i].update_pid(0.625, 0.125, 0.0);
            pid_controller[i].out_limit(-100, 100);
        }

        kinematics.set_wheel_distance(360);
        kinematics.set_motor_param(0, 0.1051566);
        kinematics.set_motor_param(1, 0.1051566);
        kinematics.set_motor_param(2, 0.1051566);
        kinematics.set_motor_param(3, 0.1051566);
        // 运动学逆解并设置速度
        kinematics.kinematic_inverse(target_linear_x_speed,
target_linear_y_speed, target_angular_speed, out_speed[0], out_speed[1],
out_speed[2], out_speed[3]);
```

```
    // 设置电机速度
    for (int i = 0; i < 4; i++)
    {
        // 初始化目标速度，单位 mm/s，使用毫米防止浮点运算丢失精度
        pid_controller[i].update_target(out_speed[i]);
    }
}

void loop()
{
    delay(10);            // 等待 10 毫秒
    motorSpeedControl(); // 调用速度控制函数
}
```

在代码清单 9-29 中，我们引入头文件并新建了 Kinematics 对象，接着在 setup 函数中初始化轮子间距和脉冲距离比参数，最后根据目标角速度和线速度进行逆运动学解得出需要的左右轮速度，然后将速度设置给 PID 控制器。

重新编译和下载代码到开发板，观察机器人运动轨迹，可以看到机器人将以半径 0.5 米转圈，这是因为我们将目标线速度设置为 50 毫米每秒，线速度设置为 0.1 弧度每秒，线速度比角速度就可以得到运动半径。

2. 麦克纳姆轮运动学正逆解

在第 7 章机器人导航和键盘控制时，我们下发的是线速度和角速度信息，并不会直接对轮子转速进行直接控制，这就需要我们把角速度和线速度转换成机器人四个轮子的速度，我们把这一过程称为运动学逆解，反之通过轮子的实际转速计算机器人的线速度和角速度的过程就是运动学正解。接着我们对运动学正逆解进行推导。

推导过程放在了本文档附页，这里直接放结论：

逆解

$$VM1 = Vtx - Vty - w(a+b)$$
$$VM2 = Vtx + Vty + w(a+b)$$
$$VM3 = Vtx + Vty - w(a+b)$$
$$VM4 = Vtx - Vty + w(a+b)$$

正解

$$Vtx=(VM1+VM2+VM3+VM4)/4$$

$$Vty=(-VM1+VM2+VM3-VM4)/4$$

$$w=(-VM1+VM2-VM3+VM4)/(4*(a+b))$$

有了运动学正逆解公式，接着我们就可以根据公式编写代码。在 fishbot_motion_control/lib 下新建目录 Kinematics，接着在目录下新建 Kinematics.h，然后在头文件中编写如代码清单 9-25 所示的内容。

代码清单9-30 fishbot_motion_control/lib/Kinematics/Kinematics.h

```
#ifndef __KINEMATICS_H__ /* 防止头文件被多次包含 */
#define __KINEMATICS_H__

#include <Arduino.h> /* 包含 Arduino 核心库*/

// 定义一个结构体用于存储电机参数
typedef struct
{
    float per_pulse_distance;  /* 单个脉冲对应轮子前进距离 */
    int16_t motor_speed;       /* 当前电机速度 mm/s，计算时使用*/
    int64_t last_encoder_tick; /* 上次电机的编码器读数*/
} motor_param_t;

/* 定义一个类用于处理机器人运动学 */
class Kinematics
{
public:
    /* 构造函数，默认实现 */
    Kinematics() = default;
    /* 析构函数，默认实现 */
    ~Kinematics() = default;

    /* 设置电机参数，包括编号和每个脉冲对应的轮子前进距离 */
    void set_motor_param(uint8_t id, float per_pulse_distance);
    /* 设置轮子间距*/
    void set_wheel_distance(float wheel_distance_a, float wheel_distance_b
);
    /* 逆运动学计算，将线速度和角速度转换为左右轮的速度*/
```

```
        void kinematic_inverse(float linear_x_speed, float linear_y_speed,
float angular_speed,float &out_wheel1_speed, float &out_wheel2_speed, float
&out_wheel3_speed, float &out_wheel4_speed);
        /* 正运动学计算，将左右轮的速度转换为线速度和角速度*/
        void kinematic_forward(float wheel1_speed, float wheel2_speed, float
wheel3_speed, float wheel4_speed, float &linear_x_speed,float
&linear_y_speed, float &angular_speed);
        /* 更新电机速度和编码器数据*/
        void update_motor_speed(uint64_t current_time, int32_t motor_tick1,
int32_t motor_tick2, int32_t motor_tick3, int32_t motor_tick4);
        /* 获取电机速度*/
        int16_t get_motor_speed(uint8_t id);

    private:
        float wheel_distance_a_and_b_; // 轮子间距 a 和 b 方向
        float wheel_distance_a_; // 轮子间距 a 方向
        float wheel_distance_b_; // 轮子间距 b 方向
        motor_param_t motor_param_[4]; /* 存储四个电机的参数*/
        uint64_t last_update_time;     /* 上次更新数据的时间，单位 ms*/
        float wheel_distance_;         /* 轮子间距*/
    };

    #endif // __KINEMATICS_H__
```

代码清单 9-25 中我们创建了一个电机参数结构体 motor_param_t，将电机对应的脉冲前进距离、电机速度和上一次编码器读数放到该结构体中。接着定义了一个运动学类 Kinematics，在该类成员函数定义了 kinematic_forward 和 kinematic_inverse 方法，用于运动学正逆解，定义方法 update_motor_speed 和 get_motor_speed 用于计算和获取电机速度，定义方法 set_wheel_distance 和 set_motor_param 用于设置机器人轮子间距和电机单脉冲距离参数。成员变量部分，声明了电机参数数组、轮子间距和上次速度数据更新时间。

接着根据方法声明，我们逐一实现各个方法，在 fishbot_motion_control/lib/Kinematics/目录下新建 Kinematics.cpp，接着编写如代码清单 9-26 所示的代码。

代码清单9-31 Kinematics.cpp 中设置和获取方法实现

```
    #include "Kinematics.h"
void Kinematics::set_motor_param(uint8_t id, float per_pulse_distance) {
  motor_param_[id].per_pulse_distance = per_pulse_distance;  /* 电机每个脉冲前进
距离*/
}
void Kinematics::set_wheel_distance(float wheel_distance_a, float
wheel_distance_b)
```

```
    {
        wheel_distance_a_ = wheel_distance_a;
        wheel_distance_b_ = wheel_distance_b;
        wheel_distance_a_and_b_ = (wheel_distance_a_ + wheel_distance_b_) / 2;
}
// 传入电机的编号 id 返回该编号电机的速度
int16_t Kinematics::get_motor_speed(uint8_t id) {
  return motor_param_[id].motor_speed;
}
```

代码清单 9-26 中实现了三个参数设置和获取方法，内容较为简单，只是对相对应的参数进行赋值即可。接着我们来编写电机速度更新方法 update_motor_speed，继续在 Kinematics.cpp 下添加如代码清单 9-27 所示的代码。

代码清单9-32 Kinematics.cpp 中电机速度计算方法实现

```
    /**
 * @brief 更新电机速度和编码器数据
 * @param current_time 当前时间（单位：毫秒）
 * @param left_tick 左轮编码器读数
 * @param right_tick 右轮编码器读数
 */
void Kinematics::update_motor_speed(uint64_t current_time, int32_t
motor_tick1, int32_t motor_tick2, int32_t motor_tick3, int32_t motor_tick4) {
  // 计算出自上次更新以来经过的时间 dt
  uint32_t dt = current_time - last_update_time;
  last_update_time = current_time;

  // 计算电机 1 和电机 2 的编码器读数变化量 dtick1 和 dtick2。
   int32_t dtick1 = motor_tick1 - motor_param_[0].last_encoder_tick;
   int32_t dtick2 = motor_tick2 - motor_param_[1].last_encoder_tick;
   int32_t dtick3 = motor_tick3 - motor_param_[2].last_encoder_tick;
   int32_t dtick4 = motor_tick4 - motor_param_[3].last_encoder_tick;
   motor_param_[0].last_encoder_tick = motor_tick1;
   motor_param_[1].last_encoder_tick = motor_tick2;
    motor_param_[2].last_encoder_tick = motor_tick3;
    motor_param_[3].last_encoder_tick = motor_tick4;

  // 轮子速度计算
  motor_param_[0].motor_speed =
      float(dtick1 * motor_param_[0].per_pulse_distance) / dt * 1000;
  motor_param_[1].motor_speed =
      float(dtick2 * motor_param_[1].per_pulse_distance) / dt * 1000;
 motor_param_[2].motor_speed =
      float(dtick2 * motor_param_[2].per_pulse_distance) / dt * 1000;
```

```
  motor_param_[3].motor_speed =
      float(dtick2 * motor_param_[3].per_pulse_distance) / dt * 1000;
}
```

电机速度更新方法接收当前时间和左右轮编码器计数值为参数，在函数内先计算时间差和编码器计数差，最后根据脉冲距离比计算出速度，单位使用 mm/s 。接着我们来完善运动学正逆解方法，继续在 Kinematics.cpp 添加如代码清单 10-28 所示的代码。

代码清单9-33 Kinematics.cpp 中运动学正逆解实现

```
   /**
 * @brief 正运动学计算，将左右轮的速度转换为线速度和角速度
 * @param left_speed 左轮速度（单位：毫米/秒）
 * @param right_speed 右轮速度（单位：毫米/秒）
 * @param[out] out_linear_speed 线速度（单位：毫米/秒）
 * @param[out] out_angle_speed 角速度（单位：弧度/秒）
 */
void Kinematics::kinematic_forward(float wheel1_speed, float wheel2_speed,
float wheel3_speed, float wheel4_speed,float &linear_x_speed, float
&linear_y_speed, float &angular_speed) {
   linear_x_speed = (wheel1_speed + wheel2_speed + wheel3_speed +
wheel4_speed) / 4.0f;
   linear_y_speed = (-wheel1_speed + wheel2_speed + wheel3_speed -
wheel4_speed) / 4.0f;
   angular_speed = float(-wheel1_speed + wheel2_speed - wheel3_speed +
wheel4_speed) / (4.0f * (wheel_distance_a_and_b_));
}

/**
 * @brief 逆运动学计算，将线速度和角速度转换为左右轮的速度
 * @param linear_speed 线速度（单位：毫米/秒）
 * @param angle_speed 角速度（单位：弧度/秒）
 * @param[out] out_left_speed 左轮速度（单位：毫米/秒）
 * @param[out] out_right_speed 右轮速度（单位：毫米/秒）
 */
void Kinematics::kinematic_inverse(float linear_x_speed, float
linear_y_speed, float angular_speed, float &out_wheel_speed1, float
&out_wheel_speed2, float &out_wheel_speed3, float &out_wheel_speed4) {
  out_wheel_speed1 = linear_x_speed - linear_y_speed - angular_speed *
(wheel_distance_a_and_b_);
  out_wheel_speed2 = linear_x_speed + linear_y_speed + angular_speed *
(wheel_distance_a_and_b_);
  out_wheel_speed3 = linear_x_speed + linear_y_speed - angular_speed *
(wheel_distance_a_and_b_);
```

```
  out_wheel_speed4 = linear_x_speed - linear_y_speed + angular_speed *
(wheel_distance_a_and_b_);
   }
```

正逆解方法我们完全按照推导的公式编写而成，不做过多解释，接着我们来修改 main.cpp ,调用 Kinematics 类完成速度计算以及正逆解，完成后 main.cpp 代码如代码清单 9-29 所示。

代码清单9-34 fishbot_motion_control/src/main.cpp

```
#include <Arduino.h>
#include <Esp32McpwmMotor.h>
#include <Esp32PcntEncoder.h>
#include <PidController.h> // 引入 PID 控制器头文件
#include <Kinematics.h>    // 引入运动学头文件
Kinematics kinematics;
Esp32PcntEncoder encoders[4];    // 创建一个数组用于存储四个编码器
Esp32McpwmMotor motor;           // 创建一个名为 motor 的对象，用于控制电机
PidController pid_controller[4]; // 创建 PID 控制器对象数组

int64_t last_ticks[4];    // 记录上一次读取的计数器数值
int32_t delta_ticks[4];   // 记录两次读取之间的计数器差值
int64_t last_update_time; // 记录上一次更新时间
float current_speeds[4];  // 记录四个电机的速度

float target_linear_x_speed = 50.0; // 目标 X 方向线速度 毫米每秒
float target_linear_y_speed = 0.0;  // 目标 Y 方向线速度 毫米每秒
float target_angular_speed = 0.1f;  // 目标角速度 弧度每秒
float out_speed[4];                 // 用于存储运动学逆解后的速度

void motorSpeedControl()
{
    // 计算时间差
    uint64_t dt = millis() - last_update_time;

    // 使用循环处理所有 4 个电机
    for (int i = 0; i < 4; i++)
    {
        // 计算编码器差值
        delta_ticks[i] = encoders[i].getTicks() - last_ticks[i];
        // 距离比时间获取速度单位 mm/ms 乘 1000 转换为 mm/s，方便 PID 计算
        current_speeds[i] = float(delta_ticks[i] * 0.1051566) / dt * 1000;
        // 更新上一次计数器数值
        last_ticks[i] = encoders[i].getTicks();
        // 根据当前速度，更新电机速度值
```

```
            motor.updateMotorSpeed(i,
pid_controller[i].update(current_speeds[i]));
        }

        // 更新上一次更新时间
        last_update_time = millis();

        // 打印所有电机的速度数据
        Serial.printf("speed1=%f mm/s, speed2=%f mm/s, speed3=%f mm/s,
speed4=%f mm/s\n",
                      current_speeds[0], current_speeds[1], current_speeds[2],
current_speeds[3]);
    }

    void setup()
    {
        // 1.初始化串口
        Serial.begin(115200); // 初始化串口通信，设置通信速率为 115200
        motor.attachMotor(0, 5, 4);
        motor.attachMotor(1, 15, 16);
        motor.attachMotor(2, 3, 8);
        motor.attachMotor(3, 46, 9);

        // 2.设置编码器
        encoders[0].init(0, 6, 7);
        encoders[1].init(1, 18, 17);
        encoders[2].init(2, 20, 19);
        encoders[3].init(3, 11, 10);
        // 初始化 pid 控制器
        for (int i = 0; i < 4; i++)
        {
            pid_controller[i].update_pid(0.625, 0.125, 0.0);
            pid_controller[i].out_limit(-100, 100);
        }

        kinematics.set_wheel_distance(216,177);
        kinematics.set_motor_param(0, 0.1051566);
        kinematics.set_motor_param(1, 0.1051566);
        kinematics.set_motor_param(2, 0.1051566);
        kinematics.set_motor_param(3, 0.1051566);
        // 运动学逆解并设置速度
        kinematics.kinematic_inverse(target_linear_x_speed,
target_linear_y_speed, target_angular_speed, out_speed[0], out_speed[1],
out_speed[2], out_speed[3]);
```

```
    // 设置电机速度
    for (int i = 0; i < 4; i++)
    {
        // 初始化目标速度，单位 mm/s，使用毫米防止浮点运算丢失精度
        pid_controller[i].update_target(out_speed[i]);
    }
}

void loop()
{
    delay(10);            // 等待 10 毫秒
    motorSpeedControl(); // 调用速度控制函数
}
```

在代码清单 9-29 中，我们引入头文件并新建了 Kinematics 对象，接着在 setup 函数中初始化轮子间距和脉冲距离比参数，最后根据目标角速度和线速度进行逆运动学解得出需要的左右轮速度，然后将速度设置给 PID 控制器。

重新编译和下载代码到开发板，观察机器人运动轨迹，可以看到机器人将以半径 0.5 米转圈，这是因为我们将目标线速度设置为 50 毫米每秒，线速度设置为 0.1 弧度每秒，线速度比角速度就可以得到运动半径。

9.3.5 机器人里程计计算

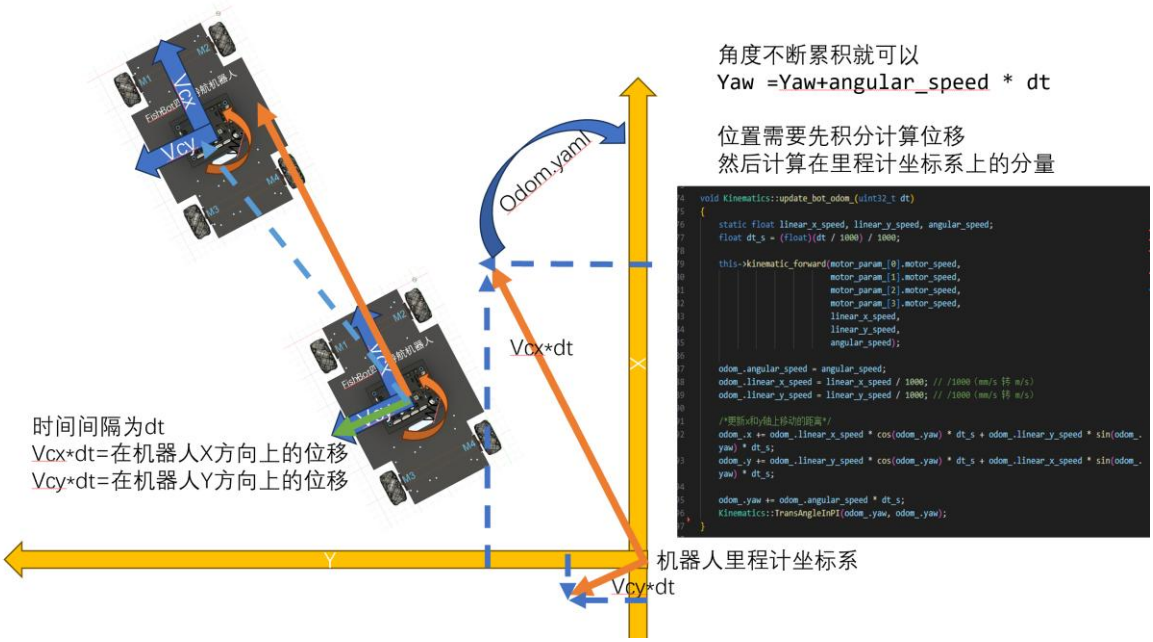
通过运动学正解，可以获取机器人底盘实时的速度信息，通过对线速度和角速度的积分，就可以获取机器人的实时位置信息，我们来推导下计算公式，假设在某一个时间段t中，机器人的线速度为,$v_{tx}v_{ty}$，角速度为w_t，机器人在初始时刻的位置为x_t，y_t朝向为θ_t,求经过t时刻后机器人新的位置(x_{t+1}, y_{t+1})和朝向θ_{t+1}。

这一段时间内机器人在自身 x 方向上前进的距离$dx = v_{tx} * t$，y 方向上前进的距离是$d = v_{ty} * t$转过的角度为$\theta = \omega_t * t$，则机器人新的角度$\theta_{t+1} = \theta_t + \theta$，我们将机器人前进的距离根据其朝向分解为在 x 和 y 轴上的位移量，则可得出以下公式。

$$x_{t+1} = x_t + dx * \cos(\theta_{t+1}) + dy * \sin(\theta_{t+1})$$

$$y_{t+1} = y_t + dx * \sin(\theta_{t+1}) + dy * \cos(\theta_{t+1})$$

再补充一张图



有了公式，我们就可以来写代码，速度积分可以在每次速度更新时计算，修改 Kinematics.h 代码如代码清单 10-30 所示。

代码清单9-35 fishbot_motion_control/lib/Kinematics.h

```
   #include <Arduino.h> // 包含 Arduino 核心库
...

typedef struct
{
       float x;                  // 坐标 x
       float y;                  // 坐标 y
       float yaw;                // yaw
       quaternion_t quaternion; // 姿态四元数
       float linear_x_speed;       // x 线速度
       float linear_y_speed;       // y 线速度
       float angular_speed;      // 角速度
} odom_t;



class Kinematics {
public:
  ...

  /* 更新里程计数据 */
  void update_odom(uint16_t dt);
   /* 获取里程计数据 */
```

```
  odom_t &get_odom();
  /* 用于将角度转换到 -π 到 π 的范围内 */
  static void TransAngleInPI(float angle, float &out_angle);

private:
  odom_t odom_;  /* 存储里程计信息 */
  ...
};
```

代码清单 9-30 中首先定义了一个 odom_t 结构体，来表示里程计信息，并声明了成员变量 odom_t 用于存储里程计。在成员方法中定义了更新和获取里程计方法，用于计算和对外提供里程计数据，为了让角度保持在-π 到 π 之间，我们定义了 TransAngleInPI 方法用于限制角度信息。接着我们来逐一实现各个新增的方法。

首先在 Kinematics.cpp 中添加里程计获取 get_odom 和角度限制方法 TransAngleInPI，具体内容如代码清单 9-31 所示。

代码清单9-36 Kinematics.cpp 中 TransAngleInPI 方法实现

```
   odom_t &Kinematics::get_odom() { return odom_; }

// 用于将角度转换到 -π 到 π 的范围内
void Kinematics::TransAngleInPI(float angle, float &out_angle) {
  // 如果 angle 大于 π，则将 out_angle 减去 2π
  if (angle > PI) {
    out_angle -= 2 * PI;
  }
  // 如果 angle 小于 -π，则将 out_angle 加上 2π
  else if (angle < -PI) {
    out_angle += 2 * PI;
  }
}
```

接着在 Kinematics.cpp 中添加里程计更新方法 update_odom，完整代码如代码清单 9-32 所示。

代码清单9-37 Kinematics.cpp 中 update_odom 方法实现

```
void Kinematics::update_odom(uint16_t dt) {
       static float linear_x_speed, linear_y_speed, angular_speed;
       float dt_s = (float)(dt / 1000) / 1000;
       this->kinematic_forward(motor_param_[0].motor_speed,
                               motor_param_[1].motor_speed,
                               motor_param_[2].motor_speed,
                               motor_param_[3].motor_speed,
```

```
                                linear_x_speed,
                                linear_y_speed,
                                angular_speed);
        odom_.angular_speed = angular_speed;
        odom_.linear_x_speed = linear_x_speed / 1000; // /1000 (mm/s 转 m/s)
        odom_.linear_y_speed = linear_y_speed / 1000; // /1000 (mm/s 转 m/s)

        /*更新 x 和 y 轴上移动的距离*/
        odom_.x += odom_.linear_x_speed * cos(odom_.yaw) * dt_s +
odom_.linear_y_speed * sin(odom_.yaw) * dt_s;
        odom_.y += odom_.linear_y_speed * cos(odom_.yaw) * dt_s +
odom_.linear_x_speed * sin(odom_.yaw) * dt_s;
        odom_.yaw += odom_.angular_speed * dt_s;
        Kinematics::TransAngleInPI(odom_.yaw, odom_.yaw);

}
```

该函数用于计算机器人的实时里程计数据，输入参数是时间间隔 dt 单位毫秒，所以在函数内首先将其转换为秒为单位，接着调用运动学正解，将轮速度转换为机器人的角速度和线速度，并转换线速度的单位为米每秒。然后让角速度乘上时间，得到 dt 时间内角度的变化量，接着将变化累积到当前角度上，并使用 TransAngleInPI 将角度值限制在 -π 到 π 的范围内。最后则根据前面的公式计算机器人自身在 x 和 y 方向上前进的距离，根据公式将其分解里程计为 x 和 y 轴上的分量，并累加到里程计的 x 和 y 上。

完成所有方法的编写后，还需要修改 update_motor_speed 方法，在其最后一行添加对 update_odom 的调用，添加的代码如代码清单 9-33 所示。

代码清单9-38 Kinematics.cpp 中 update_motor_speed 方法实现

```
void Kinematics::update_motor_speed(uint64_t current_time, int32_t left_tick,
                                    int32_t right_tick) {
  ...
  // 更新里程计信息
  update_odom(dt);
}
```

为了能够实时看到机器人的位置信息，我们可以添加打印到 main.cpp 的 loop 方法中，添加的代码和位置如代码清单 9-34 所示。

代码清单9-39 在 main.cpp 中添加打印

```
void loop() {
  ...
  Serial.printf("x=%f,y=%f,angle=%f\n", kinematics.get_odom().x,
```

```
                kinematics.get_odom().y, kinematics.get_odom().angle);
}
```

重新编译工程并下载到开发板，可以观察到终端的打印信息如代码清单 9-35 所示。

代码清单9-40 里程计信息打印结果

```
x=-0.059717,y=0.996842,angle=-3.031903
x=-0.060079,y=0.996802,angle=-3.031274
x=-0.060651,y=0.996739,angle=-3.030646
```

从打印信息中可以看出里程计信息已经可以正常获取了，需要注意的是，里程计是从左右轮实时速度计算而来的，并不是通过目标速度计算得出。

好了，到这里便完成了机器人底盘控制系统的开发，接下来我们就来将控制系统接入到 ROS 2 中。

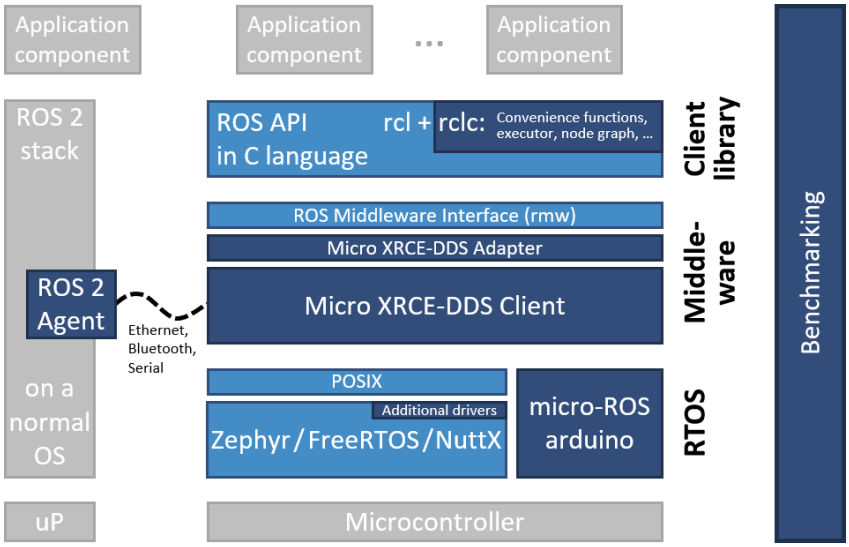
9.4 使用 micro-ROS 接入 ROS 2

上一节完成后我们实现了对机器人速度的控制和里程计计算，但要与导航配合使用，还需要把数据接入到 ROS 2 系统中，将控制指令和里程计变成 ROS 2 的话题。micro-ROS 是一组运行在微控制器的软件，通过它我们可以在微控制器上使用话题订阅发布等 API 进行通信，它的图标如图 9-16 所示。

图 9-16 micro-ROS 图标

9.4.1 介绍与第一个节点

micro-ROS 的目标是将 ROS 2 引入到微控制器上使用，针对微控制器资源受限的特点，micro-ROS 对相关 API 进行了深度优化，micro-ROS 整体框架如图 9-17 所示。



Application component ——应用组件

ROS 2 stack —— ROS 2 堆栈

On a normal OS —— 在一个正常的 OS 上

uP——上位机

Microcontroller——微控制器

Benchmarking——基准测试

RTOS——实时操作系统

Middleware——中间件

Client Library——客户端库

ROS Middleware Interface(rmw)——ROS 中间件接口（rmw）

Micro XRCE-DDS Adapter——Micro XRCE-DDS 适配器

Micro XRCE-DDS Client——Micro XRCE-DDS 客户端

ROS API in C language——ROS 在 C 语言中的应用程序接口

图 9-17 micro-ROS 整体框架

其中深色组件是专为 micro-ROS 开发的，浅色组件取自标准 ROS 2 软件。这里重点对深色部分进行介绍，最左边的 ROS 2 Agent 是 micro-ROS 在正常系统上的代理，它通过串口、蓝牙或者以太网等协议和微处理器平台的 micro-ROS 进行连接，并进行数据的转发。中间深色部分，最下面 RTOS 部分的 micro-ROS arduino 是基于 arduino 开发的代码库，中间的中间件部分是经过优化的微型 DDS 适配器和客户端，上面的客户端库则是提供了一套基于 rclc 的 API 接口。最右侧是用于嵌入式软件基准测试的工具。

要将微控制器连接到 ROS 2 中，需要做两部分工作，第一个是在正常系统中安装 Agent，第二个是在微控制器中编写 micro-ROS 程序。我们先在系统中安装 Agent，在主目录下新建 chapt9/fishbot_ws/src 目录，接着克隆 micro-ROS Agent 源码到 src 目录，指令如代码清单 10-36 所示。

代码清单9-41 下载 micro-ROS-Agent 源码

```
   $ cd fishbot_ws/src
$ git clone https://github.com/micro-ROS/micro-ROS-Agent.git -b $ROS_DISTRO
$ git clone https://github.com/micro-ROS/micro_ros_msgs.git -b $ROS_DISTRO
```

接着使用 colcon 进行功能包构建，构建完成，就可以直接运行 agent 了，构建及运行命令如代码清单 9-37 所示。

代码清单9-42 构建并运行 Agent

```
   $ colcon build
$ source install/setup.bash
$ ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888
```

这里我们启动 micro_ros_agent 节点，并使用 udp4 作为传输协议，指定端口号为 8888，在接下来编写微控制器代码时就需要指定通信协议、主机地址和端口号。主机地址就是系统的 IP 地址，如图 9-18 所示，可以在系统的设置的网络模块查看地址。

图 9-18 查看当前系统网络地址

Agent 准备好后，接着我们尝试在原来的工程里引入 micro-ROS，修改 fishbot_motion_control/platformio.ini 文件，添加依赖库和配置如代码清单 9-38 所示。

代码清单9-43 添加 micro-ROS 依赖库

```
   ...
board_microros_transport = wifi
lib_deps =
    ...
    https://gitee.com/ohhuo/micro_ros_platformio.git
```

这里添加了 micro-ROS 的 platformio 版本依赖库，并添加了传输协议配置项 board_microros_transport 指定使用 wifi 与 ROS 2 Agent 进行连接。接着修改 src/main.cpp 的代码，我们来创建第一个 micro-ROS 节点，修改后添加的代码及位置如代码清单 9-39 所示。

代码清单9-44 fishbot_motion_control/src/main.cpp

```
   ...
// 引入 micro-ROS 和 WiFi 相关头文件
   #include <WiFi.h>
   #include <micro_ros_platformio.h>
```

```
   #include <rcl/rcl.h>
   #include <rclc/rclc.h>
   #include <rclc/executor.h>
...
float out_right_speed;

// 声明相关的结构体对象
rcl_allocator_t allocator;      // 内存分配器，用于动态内存分配管理
rclc_support_t support;         // 用于存储时钟、内存分配器和上下文，提供支持
rclc_executor_t executor;       // 执行器，用于管理订阅和计时器回调的执行
rcl_node_t node;                  // 节点

// 单独创建一个任务运行 micro-ROS ，相当于一个线程
void micro_ros_task(void *parameter) {
  // 1.设置传输协议并延时等待设置完成
  IPAddress agent_ip;
  agent_ip.fromString("192.168.4.136"); // 替换为你自己主机的 IP 地址
  set_microros_wifi_transports("WIFI_NAME", "WIFI_PASSWORD", agent_ip, 8888);
  delay(2000);
  // 2.初始化内存分配器
  allocator = rcl_get_default_allocator();
  // 3.初始化 support
  rclc_support_init(&support, 0, NULL, &allocator);
  // 4.初始化节点 fishbot_motion_control
  rclc_node_init_default(&node, "fishbot_motion_control", "", &support);
  // 5.初始化执行器
  unsigned int num_handles = 0;
  rclc_executor_init(&executor, &support.context, num_handles, &allocator);
  // 循环执行器
  rclc_executor_spin(&executor);
}

void setup() {
  ...
  // 创建任务运行 micro_ros_task
  xTaskCreate(micro_ros_task,   // 任务函数
              "micro_ros",      // 任务名称
              10240,            // 任务堆栈大小（字节）
              NULL,             // 传递给任务函数的参数
              1,                // 任务优先级
              NULL              // 任务句柄
  );
}
...
```

在代码清单 9-39 中，我们首先引入了三个 micro-ROS 相关的头文件，接着声明了内存分配器 allocator，由于微处理器平台资源有限，所以使用 allocator 进行内存的分配和回收。声明 support，用于存储时钟、内存分配器和上下文，提供支持。声明 executor，用于管理订阅和计时器回调的执行，声明 node 用于存储节点。

创建了一个名为 micro_ros_task 的单独任务函数，负责初始化并运行 micro-ROS 节点，任务在嵌入式系统中可以简单理解为线程。在任务函数中，首先设置传输协议为无线网络 wifi，这里需要设置 Agent 所在系统的 IP 地址、端口和 WIFI 信息，当开发板连接到 WIFI 后就会尝试发送数据到 Agent 所在的 IP 地址和端口。接着第二步我们使用 rcl_get_default_allocator 函数使用默认设置初始化 allocator，第三步调用 rclc_support_init 初始化 support，该函数的四个参数分别是 rclc_support_t 结构体指针，参数数量，参数数组指针和分配器指针。第四步是调用 rclc_node_init_default 初始化节点，该函数有四个参数，分别是 rcl_node_t 指针，节点名称，命名空间和 rclc_support_t 指针。第五步是调用 rclc_executor_init 初始化执行器，一共有四个参数，分别是 rclc_executor_t 指针，support 中的上下文指针，可处理的句柄数量和分配器的指针。最后调用 rclc_executor_spin 对执行器的事件进行不断的循环处理。

最后在 setup 函数中，通过 xTaskCreate 函数来创建任务，将其添加到系统的任务队列中以运行。

代码清单 9-39 中的代码就是一个完整的 micro-ROS 节点的编写方法，要和 Agent 建立通信，需要确保 WIFI 账户信息、Agent 地址和端口号正确，另外需要注意的是 ESP32 仅支持 2.4 GHz 的 WIFI 信号。

将代码下载到开发板，连接成功后可以看到 micro-ROS Agent 终端的提示如代码清单 9-40 所示，表示连接成功。

代码清单9-45 运行 micro_ros_agent

```
   $ ros2 run micro_ros_agent micro_ros_agent udp4 --port 8888
---
[1695806525.612639] info     | UDPv4AgentLinux.cpp | init
| running...             | port: 8888
[1695806525.612840] info     | Root.cpp           | set_verbose_level
| logger setup           | verbose_level: 4
[1695806538.128993] info     | Root.cpp           | create_client
| create                 | client_key: 0x3E93E8A4, session_id: 0x81
[1695806538.129030] info     | SessionManager.hpp | establish_session
| session established    | client_key: 0x3E93E8A4, address:
192.168.4.157:47138
```

```
[1695806538.156269] info     | ProxyClient.cpp    | create_participant
| participant created    | client_key: 0x3E93E8A4, participant_id: 0x000(1)
[1695806538.171986] info     | ProxyClient.cpp    | create_topic
| topic created          | client_key: 0x3E93E8A4, topic_id: 0x000(2),
participant_id: 0x000(1)
[1695806538.185341] info     | ProxyClient.cpp    | create_subscriber
| subscriber created     | client_key: 0x3E93E8A4, subscriber_id: 0x000(4),
participant_id: 0x000(1)
[1695806538.204017] info     | ProxyClient.cpp    | create_datareader
| datareader created     | client_key: 0x3E93E8A4, datareader_id: 0x000(6),
subscriber_id: 0x000(4)
```

接着在新的终端中输入命令查看节点列表，命令及结果如代码清单 9-41 所示。

代码清单9-46 查看节点列表

```
   $ ros2 node list
---
/fishbot_motion_control
```

可以看到查询到了来自微控制器的 fishbot_motion_control 节点，有了节点，下面我们就来创建话题订阅者，来接收来自主机的控制指令。

9.4.2 订阅话题控制机器人

上一节我们将微处理器中的节点通过 Agent 接入了 ROS 2 系统中，本节我们尝试在微控制器平台上创建话题订阅者，订阅目标角速度和线速度指令。我们知道在 ROS 2 中使用的消息接口 geometry_msgs/msg/Twist 来表示速度指令，该消息接口定义如代码清单 9-42 所示。

代码清单9-47 查看 geometry_msgs/msg/Twist 接口定义

```
$ ros2 interface show geometry_msgs/msg/Twist
---
# This expresses velocity in free space broken into its linear and angular
parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
```

linear 表示线速度，angular 表示角速度，x、y 和 z 表示速度在三个轴上的分量，在 ROS 2

中，定义机器人的正前方为 x 轴，z 轴垂直地面向上，符合右手坐标系，所以线速度在 x 轴上的分量就是我们需要的机器人线速度，角速度在 z 轴的分量就是机器人的角速度。

接着在代码清单 9.39 的基础上编写代码，添加速度话题的订阅者和回调函数，在 src/main.cpp 中添加的代码如代码清单 9-43 所示。

代码清单9-48 fishbot_motion_control/src/main.cpp

```
   ...
#include <geometry_msgs/msg/twist.h>
...
rcl_subscription_t subscriber;     // 订阅者
geometry_msgs__msg__Twist sub_msg; // 存储订阅到的速度消息

void twist_callback(const void *msg_in) {
  // 将接收到的消息指针转化为 geometry_msgs__msg__Twist 类型
  const geometry_msgs__msg__Twist *twist_msg =
      (const geometry_msgs__msg__Twist *)msg_in;
  // 运动学逆解并设置速度
  kinematics.kinematic_inverse(twist_msg->linear.x * 1000,
twist_msg->angular.z,
                               out_left_speed, out_right_speed);
  pid_controller[0].update_target(out_left_speed);
  pid_controller[1].update_target(out_right_speed);
}

void microros_task(void *parameter) {
  ...
  // 4.初始化执行器
  unsigned int num_handles = 0+1;
  rclc_executor_init(&executor, &support.context, num_handles, &allocator);
  // 5. 初始化订阅者并添加到执行器中
  rclc_subscription_init_best_effort(
      &subscriber, &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(geometry_msgs, msg, Twist), "/cmd_vel");
  rclc_executor_add_subscription(&executor, &subscriber, &sub_msg,
                                 &twist_callback, ON_NEW_DATA);
  // 循环执行器
  rclc_executor_spin(&executor);
}
```

和使用 C++语言添加订阅者相同，第一步要引入订阅者头文件，接着声明话题订阅者 subscriber 和存储消息的 sub_msg，微处理器资源有限，所以提前创建了 geometry_msgs__msg__Twist 类型的消息结构体 sub_msg 用于存储订阅到的速度指令。

我们在 micro_ros_task 中首先修改了 rclc_executor_init 函数的句柄数量参数 num_handles 为 1，这是因为执行器需要一个句柄来处理订阅事件，接着调用函数 rclc_subscription_init_best_effort 初始化订阅者，该函数的四个参数分别是订阅者指针、节点指针、消息接口类型和话题名称，初始化订阅者后，调用 rclc_executor_add_subscription 函数将订阅者添加到执行器中，当执行器收到新的消息时就会调用 twist_callback 函数进行处理，该函数的五个参数分别是执行器指针、订阅者指针、订阅的消息接口指针、回调函数指针和调用情形。需要注意这里使用的是 best_effort 即最大努力订阅数据，发布者不用确保消息的到达，10.1 节有对 Qos 的详细介绍。

当收到速度指令时就会调用 twist_callback 函数，该函数参数是一个空指针，在 C 语言中，空指针可以表示任意类型的指针，要使用它需要把它强制类型转换到我们需要的类型指针，所以在回调函数中我们第一步将其转换成 geometry_msgs__msg__Twist 类型的指针，接着调用运动学逆解函数，将目标的线速度和角速度转换成左右轮的目标速度，然后调用 PID 控制器更新目标速度。需要注意的是 ROS 2 订阅来的线速度单位为米每秒，这里乘上 1000 将其转换成毫米每秒。

将代码下载到开发板，保持 Agent 运行和网路通畅，连接成功后使用代码清单 10-44 中的指令查看话题。

代码清单9-49 查看话题列表

```
$ ros2 topic list -v
---
Published topics:
 * /parameter_events [rcl_interfaces/msg/ParameterEvent] 2 publishers
 * /rosout [rcl_interfaces/msg/Log] 2 publishers

Subscribed topics:
 * /cmd_vel [geometry_msgs/msg/Twist] 1 subscriber
```

可以看到多出了一个/cmd_vel 话题的订阅者，消息接口类型是 geometry_msgs/msg/Twist，接着我们可以使用 ROS 2 的键盘控制节点来向这个话题发布速度指令，也可以直接使用代码清单 9-45 中的命令来发布，运行键盘控制节点。

代码清单9-50 使用键盘控制机器人

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
---

This node takes keypresses from the keyboard and publishes them
as Twist messages. It works best with a US keyboard layout.
---------------------------
Moving around:
```

```
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

按 x 键可以降低线速度，使用 c 键可以降低角速度，接着按 i 就可以控制机器人前进，可以将机器人放到地面，尝试使用键盘控制节点控制机器人移动。

9.4.3 发布机器人里程计话题

机器人导航除了要发布速度命令给机器人，还需要获取到里程计数据，在 ROS 2 中，里程计消息接口为 nav_msgs/msg/Odometry，该消息接口定义如代码清单 9-46 所示。

代码清单9-51 查看 nav_msgs/msg/Odometry 消息接口定义

```
$ ros2 interface show nav_msgs/msg/Odometry
 ---
# This represents an estimate of a position and velocity in free space.
# The pose in this message should be specified in the coordinate frame given
by header.frame_id
# The twist in this message should be specified in the coordinate frame given
by the child_frame_id

# Includes the frame id of the pose parent.
std_msgs/Header header
        builtin_interfaces/Time stamp
                int32 sec
```

```
                uint32 nanosec
        string frame_id

# Frame id the pose points to. The twist is in this coordinate frame.
string child_frame_id

# Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/PoseWithCovariance pose
        Pose pose
                Point position
                        float64 x
                        float64 y
                        float64 z
                Quaternion orientation
                        float64 x 0
                        float64 y 0
                        float64 z 0
                        float64 w 1
        float64[36] covariance

# Estimated linear and angular velocity relative to child_frame_id.
geometry_msgs/TwistWithCovariance twist
        Twist twist
                Vector3  linear
                        float64 x
                        float64 y
                        float64 z
                Vector3  angular
                        float64 x
                        float64 y
                        float64 z
        float64[36] covariance
```

从注释看可以将该消息接口分为四个部分，第一是 header，包含时间戳和当前的 frame_id，第二是子坐标系名称 child_frame_id，通过 frame_id 和 child_frame_id 来确定里程计是哪两个坐标系之间的关系，第三部分是姿态 pose，第四部分是速度 twist。

了解了接口，我们来编写代码，修改 src/main.cpp，添加发布者、进行时间同步和创建定时器，修改的内容如代码清单 9-47 所示。

代码清单9-52 fishbot_motion_control/src/main.cpp

```
...
#include <nav_msgs/msg/odometry.h>
#include <micro_ros_utilities/string_utilities.h>
```

```
rcl_publisher_t odom_publisher;   // 发布者
nav_msgs__msg__Odometry odom_msg; // 里程计消息
rcl_timer_t timer;  // 定时器，可以定时调用某个函数

// 在定时器回调函数中完成话题发布
void callback_publisher(rcl_timer_t *timer, int64_t last_call_time) {
  odom_t odom = kinematics.get_odom();     // 获取里程计
  int64_t stamp = rmw_uros_epoch_millis(); // 获取当前时间
  odom_msg.header.stamp.sec = static_cast<int32_t>(stamp / 1000); // 秒部分
  // 纳秒部分
  odom_msg.header.stamp.nanosec = static_cast<uint32_t>((stamp % 1000) *
1e6);
  odom_msg.pose.pose.position.x = odom.x;
  odom_msg.pose.pose.position.y = odom.y;
  odom_msg.pose.pose.orientation.w = cos(odom.angle * 0.5);
  odom_msg.pose.pose.orientation.x = 0;
  odom_msg.pose.pose.orientation.y = 0;
  odom_msg.pose.pose.orientation.z = sin(odom.angle * 0.5);
  odom_msg.twist.twist.angular.z = odom.angle_speed;
  odom_msg.twist.twist.linear.x = odom.linear_speed;
  // 发布里程计
  if(rcl_publish(&odom_publisher, &odom_msg, NULL)!=RCL_RET_OK){
    Serial.printf("error: odom publisher failed!\n");
  }
}


void micro_ros_task(void *parameter) {
  ...
     unsigned int num_handles = 0 + 2;
     rclc_executor_init(&executor, &support.context, num_handles,
&allocator);
  // 6.初始化发布者和定时器
  odom_msg.header.frame_id =
      micro_ros_string_utilities_set(odom_msg.header.frame_id, "odom");
  odom_msg.child_frame_id =
      micro_ros_string_utilities_set(odom_msg.child_frame_id,
"base_footprint");
  rclc_publisher_init_best_effort(
      &odom_publisher, &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(nav_msgs, msg, Odometry), "/odom");
  // 7. 时间同步
  while (!rmw_uros_epoch_synchronized()) { // 如果没有同步
```

```
    rmw_uros_sync_session(1000);            // 尝试进行时间同步
    delay(10);
  }
  // 8. 创建定时器，间隔 50 ms 发布调用一次 callback_publisher 发布里程计话题
  rclc_timer_init_default(&timer, &support, RCL_MS_TO_NS(50),
callback_publisher);
  rclc_executor_add_timer(&executor, &timer);
  // 循环执行器
  rclc_executor_spin(&executor);
}
```

在代码清单 9-47 中，首先添加了 nav_msgs/msg/odometry.h 消息接口头文件，又额外添加了 string_utilities.h 头文件用于消息中的字符串分配空间和赋值。接着定义了里程计发布者 odom_publisher、里程计消息 odom_msg 和定时器 timer。在 micro_ros_task 中，首先调用 micro_ros_string_utilities_set 对里程计消息的 frame_id 和 child_frame_id 进行初始化赋值。接着调用 rclc_publisher_init_best_effort 初始化里程计发布者，该函数的四个参数分别是里程计发布者指针、节点指针、消息接口和话题名称，需要注意这里使用的是 best_effort 即最大努力发布数据，不用确保消息的到达。

接下来第 7 步，进行微控制器和主机之间的时间同步，因为里程计消息中需要写明当前的时间，使用 rmw_uros_epoch_synchronized 函数用于检查时间同步状态，如果返回 false，表示时间尚未同步，则使用 rmw_uros_sync_session 尝试进行时间同步。最后，为了定时发布里程计话题，我们创建了一个定时器 timer，它将每 50 毫秒调用一次 callback_publisher 函数进行里程计话题的发布。

在 callback_publisher 函数中首先调用 kinematics.get_odom 获取里程计，然后调用 rmw_uros_epoch_millis 函数获取当前时间，单位 ms，接着分别对各个数据进行赋值，因为 odom_msg 对角度的表示使用的是四元数，所以我们根据欧拉角 Yaw 角转四元数的公式，调用正余弦将欧拉角转成四元数，最后调用 rcl_publish 函数发布里程计消息，同时根据返回值判断是否发送出错，出错则打印发布失败。

完成代码，重新构建工程并下载到开发板，接着在主机端运行 Agent，连接成功后使用命令行工具查看话题列表，可以看到，如代码清单 9-48 所示，里程计话题已经出现了。

代码清单9-53 查看话题列表

```
$ ros2 topic list -v
---
Published topics:
```

```
 * /odom [nav_msgs/msg/Odometry] 1 publisher
...
```

继续使用命令行查看里程计数据，命令如代码清单 9-49 所示。

代码清单9-54 输出里程计数据

```
$ ros2 topic echo /odom --once
---
header:
  stamp:
    sec: 1695890269
    nanosec: 351
  frame_id: odom
child_frame_id: base_footprint
pose:
  pose:
    position:
      x: 0.0025945839006453753
      y: 4.091512528248131e-05
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.014828025828128564
      w: 0.9998900587814844
 ...
---
```

除了使用命令行来查看里程计外，还可以使用 RViz 来可视化里程计数据，打开 RViz，依次点击 Add/By Topic/Odometry/OK，然后修改固定坐标系为 odom，修改 Odometry/Topic/Reliablity Policy 为 Best Effort，匹配微控制器发布者的服务质量。接着就可以在如图 9-19 所示的右侧窗口中看到红色箭头，就是机器人的当前里程计位置。

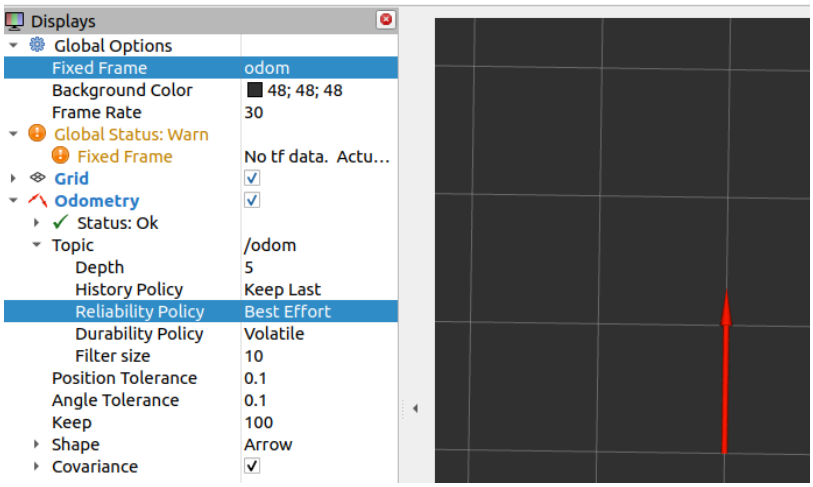


图 9-19 RViz 中机器人里程计显示

完成里程计话题发布和速度命令的控制就完成了机器人底盘控制系统了，接着我们可以来实现移动机器人建图和导航。

9.5 移动机器人建图与导航实现

前面章节使用仿真机器人实现过导航，真实机器人导航和仿真类似，都需要提供必要的话题和坐标系变换。前面几节我们成功完成了机器人底盘控制系统的开发，将里程计话题和速度控制话题都完成了，但要运行建图和导航，还需要准备雷达话题和坐标变换。本小节我们来完成剩下的步骤，并进行真机的建图和导航配置。

9.5.1 驱动并显示雷达点云

FishBot 采用的是性价比较高的 YdLidar X2 单线激光雷达，配合一块单独的串口转无线转接板，可以将雷达的数据通过无线网络转发到电脑上，完整连接驱动流程如图 9-20 所示。

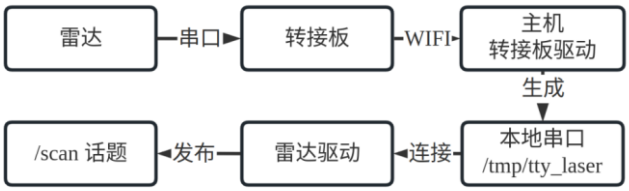


图 9-20 FishBot 雷达工作流程

转接板有配置（FLASH）、无线（WIFI）和有线（UART）三种工作模式，首先根据说明书调整转接板到配置模式，使用数据线连接转接板到电脑，打开串口监视器。如图 9-21 所示，选择/dev/ttyUSB*开头的端口，修改 Line ending 为 LF 表示发送数据时在行尾增加一个‘\n’回车，最后打开串口，在下方输入指令$command=read_config 并发送，等待返回。

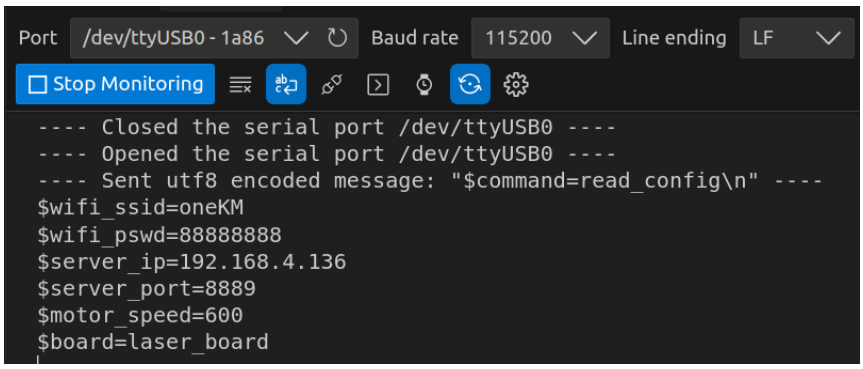


图 9-21 使用命令行读取转接板配置

指令$command=read_config 用于读取转接板的所有配置，和前面的主控板相同，这里我们需要修改 WIFI 名称、密码和主机的 IP 地址，比如修改 WIFI 账号为 fishbot，则发送指令$wifi_ssid=fishbot 即可，修改其他选项同样发送指令即可，发送完成后，切换转接板到无线模

式。

转接板配置完成，接着我们下载转接板驱动，创建工作空间并下载转接板驱动，命令如代码清单 9-50 所示。

代码清单9-55 下载雷达转接板驱动

```
   $ cd chapt9/fishbot_ws/src
$ git clone https://github.com/fishros/ros_serial2wifi.git
```

下载完代码，构建工作空间，接着运行串口转 WIFI 驱动，桥接 8889 端口到本地的/tmp/tty_laser，命令如代码清单 9-51 所示。

代码清单9-56 构建并运行转接板驱动

```
   $ colcon build
$ source install/setup.bash
$ ros2 run ros_serail2wifi tcp_server --ros-args -p
serial_port:=/tmp/tty_laser
---
[INFO] [1696085689.924353727] [tcp_socket_server_node]: TCP 端口:8889, 已映射到
串口设备:/tmp/tty_laser
[INFO] [1696085689.924522129] [tcp_socket_server_node]: 等待接受连接..
[INFO] [1696085690.444348935] [tcp_socket_server_node]: 来自('192.168.4.207',
57277)的连接已建立
```

在新的终端中输入 cat /tmp/tty_laser 命令可以查看来自雷达的数据，但需要雷达驱动才能解析出数据的内容，如代码清单 9-52 所示，下载雷达驱动到 fishbot_ws/src 目录下。

代码清单9-57 下载雷达驱动

```
   $ cd fishbot_ws/src
$ git clone https://github.com/fishros/ydlidar_ros2.git -b fishbot
```

克隆好代码，我们来修改几个配置，匹配转接板驱动和雷达，修改雷达驱动的配置文件 ydlidar_ros2/params/ydlidar.yaml，修改端口号为/tmp/tty_laser 来和我们转接板使用的端口保持一致，修改 frame_id 为 laser_link，修改完成后重新构建工作空间并运行雷达驱动，驱动运行指令如代码清单 9-53 所示。

代码清单9-58 运行雷达驱动

```
   $ ros2 launch  ydlidar ydlidar_launch.py
---
[INFO] [launch]: All log files can be found below
/home/fishros/.ros/log/2023-09-30-22-53-11-250106-fishros-linux-27436
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [ydlidar_node-1]: process started with pid [27447]
```

```
[ydlidar_node-1] [YDLIDAR INFO] Current ROS Driver Version: 1.4.5
[ydlidar_node-1] [YDLIDAR]:SDK Version: 1.4.5
[ydlidar_node-1] [YDLIDAR]:Lidar running correctly ! The health status: good
[ydlidar_node-1] [YDLIDAR] Connection established in
[/tmp/tty_laser][115200]:
[ydlidar_node-1] Firmware version: 1.5
[ydlidar_node-1] Hardware version: 1
[ydlidar_node-1] Model: S4
[ydlidar_node-1] Serial: 2021051900000032
[ydlidar_node-1] timout count: 1
[ydlidar_node-1] [YDLIDAR]:Fixed Size: 340
[ydlidar_node-1] [YDLIDAR]:Sample Rate: 3K
[ydlidar_node-1] [YDLIDAR INFO] Current Sampling Rate : 3K
[ydlidar_node-1] [YDLIDAR INFO] Now YDLIDAR is scanning ......
```

接着查看所有话题列表，就可以看到雷达话题/scan 了，通过命令行可以打印一帧话题数据，命令及结果如代码清单 9-54 所示。

代码清单9-59 打印点云信息

```
   $ ros2 topic echo /scan --once
---
header:
  stamp:
    sec: 1696087288
    nanosec: 399012000
  frame_id: laser_link
angle_min: -3.1415927410125732
angle_max: 3.1415927410125732
angle_increment: 0.018534470349550247
time_increment: 0.00033333300962112844
scan_time: 0.11466655135154724
range_min: 0.10000000149011612
range_max: 8.0
ranges:
- 0.10949999839067459
- 0.11349999904632568
- 0.11649999767541885
- '...'
intensities:
- 1016.0
- 1016.0
- 1016.0
- '...'
---
```

header 中放的是时间和固定的坐标系名称，angle_min 是起始角度，angle_max 是结束角度，angle_increment 角度增量，这个值表示相邻激光束之间的角度差，scan_time 扫描时间，表示激光扫描的持续时间，range_min 表示测量范围的最小值，range_max 表示测量范围的最大值，ranges 是数组，包含激光束的测量距离数据，intensities 同样是数组，包含激光束的强度数据。

使用 RViz 可以实现雷达数据的可视化，打开 RViz，依次点击 Add/By Topic/LaserScan/OK，然后修改 Fixed Frame 为 laser_link，修改 LaserScan/Topic/Reliablity Policy 为 Best Effort，匹配雷达传感器话题的服务质量，最后可以看到雷达话题如图 9-22 所示。

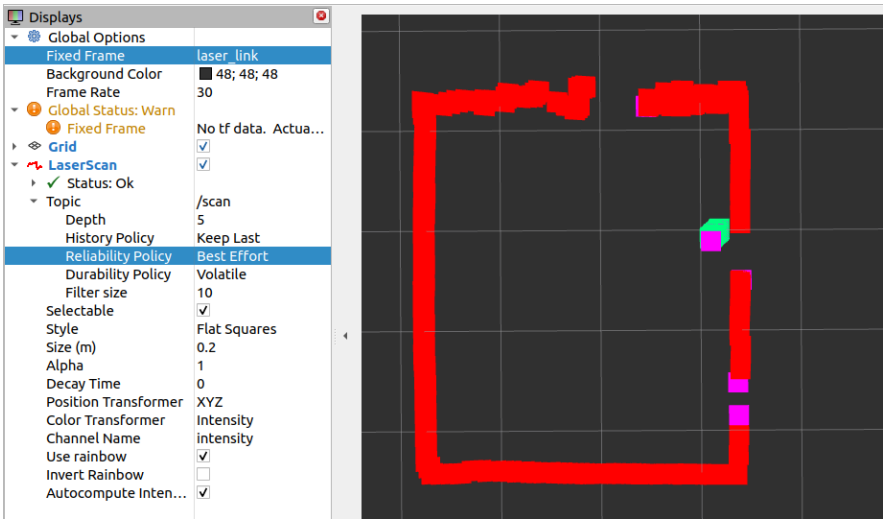


图 9-22 RViz 中显示点云

好了，到这里，雷达话题、里程计话题和速度控制话题都有了，但要建图和导航，还需要准备坐标变换。

9.5.2 移动机器人的坐标系框架介绍

基于 ROS 进行移动机器人开发时，我们需要约定好坐标系变换，作为补充的约定，ROS 提供了一系列的提升建议即 REP（ROS Enhancement Proposal）。REP-105 名为移动平台的坐标系框架（Coordinate Frames for Mobile Platforms），该提案由 Wim Meeussen 于 2010 年 10 月 27 日创建，主要规定了移动机器人坐标系的位置、连接规范和连接维护组件。

首先我们来介绍各个坐标系以及所在的位置，第一个要介绍的是基坐标系 base_link，该坐标系固定在移动机器人的基座上，为了让机器人的轮子贴合地面，还会使用 base_footprint 作为 base_link 的父坐标系。

第二个要介绍的是里程计坐标系 odom，里程计坐标系是一个固定在世界位置的坐标系，odom 坐标系会随着时间变化而漂移，所以它无法作为长期的全局坐标系使用。但机器人在 odom 坐标系中的位置都是连续的变化，并不会发生跳跃，这是其优点，所以在自定义控制器计算速度

时，采用的是里程计位置作为当前位置计算速度。

第三个要介绍的是地图坐标系 map，地图坐标系也是一个固定在世界位置的坐标系，Z 轴向上，机器人在 map 坐标系的姿态不会随着时间而漂移，所以 map 坐标系作为长期全局参考使用。但 map 坐标系不是连续变化的，即机器人在 map 坐标系中的姿态会随时发生跳跃性的变化。

第四个要介绍的是地球坐标系 earth，该坐标系固定在地心位置即 ECEF（Earth Centered Earth Fixed），当同时使用多个地图时，则可以通过 earth 坐标系进行连接。

在建图和导航时，我们需要维护图 9-23 所示的坐标系关系。

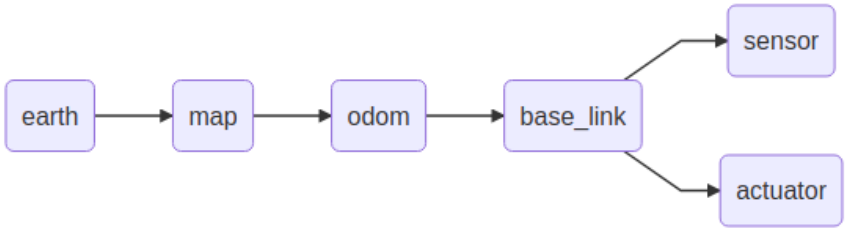


图 9-23 建图是需要维护的坐标变换

如果只使用单地图，则不需要维护 earth 到 map 之间的变换，而 base_link 到机器人传感器 sensor 和执行器 actuator 之间的变换一般使用 URDF 进行描述，然后使用 robot_state_publisher 节点进行广播。

odom 到 base_link 之间的坐标变换通过测距模块发布，这个测距模块可能是轮式里程计，也有可能是视觉里程计或其他模块。在 FishBot 中，我们需要根据里程计数据发布坐标变换。

map 到 odom 之间的坐标变换则是由定位组件基于传感器观测并不断重新计算机器人在 map 坐标系中的姿态，但定位组件并不会广播从 map 到 base_link 的变换。它首先接收从 odom 到 base_link 的变换，然后使用计算 map 到 odom 之间的坐标变换并发布，在 Navigation 2 中，由 AMCL 模块来完成这个工作，但使用 slam_toobox 建图时，则由 slam_toolbox 内部组件完成这部分工作。

好了，关于移动机器人坐标系框架的介绍就到这里，下面我将带你一起结合 FishBot 真机完善这个坐标变换框架。

9.5.3 准备机器人 URDF

URDF 用于描述机器人模型，通过 URDF 文件和 robot_state_publisher 节点就可以发布基坐标系和各个组件之间的变换，上一节我们了解了移动机器人所需的坐标变换，本节我们就来编写 URDF 并广播变换。

在 chapt9/fishbot_ws/src 下新建 fishbot_description 功能包，采用默认的构建类型即可，接着在

src/fishbot_description/下新建 urdf 目录，新建 fishbot.urdf 文件并编写如代码清单 9-55 所示的代码。

代码清单9-60 src/fishbot_description/urdf/fishbot.urdf

```
   <?xml version="1.0"?>
<robot name="fishbot">
  <link name="base_footprint" />

  <!-- base link -->
  <link name="base_link">
  <visual>
    <origin xyz="0 0 0.0" rpy="0 0 0" />
    <geometry>
      <cylinder length="0.12" radius="0.10" />
    </geometry>
    <material name="blue">
      <color rgba="0.1 0.1 1.0 0.5" />
    </material>
  </visual>
  </link>
  <joint name="base_joint" type="fixed">
    <parent link="base_footprint" />
    <child link="base_link" />
    <origin xyz="0.0 0.0 0.076" rpy="0 0 0" />
  </joint>

  <!-- laser link -->
  <link name="laser_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder length="0.02" radius="0.02" />
    </geometry>
    <material name="black">
      <color rgba="0.0 0.0 0.0 0.5" />
    </material>
  </visual>
  </link>
  <joint name="laser_joint" type="fixed">
    <parent link="base_link" />
    <child link="laser_link" />
    <origin xyz="-0.10 0 0.075" rpy="0 0 0" />
  </joint>
```

```
</robot>
```

代码清单 9-55 中的 URDF 较为简洁，只添加了 base_footprint、base_link 和 laser_link，并根据机器人实际参数修改 laser_joint 和 base_joint 的平移和旋转。在建图和导航时，会根据 base_link 和 laser_link 之间的坐标转换对激光点的坐标进行转换，如果有用到其他传感器，也需要在 URDF 中添加相应的部件和关节。保存好代码，在 CMakeLists.txt 中添加拷贝 urdf 目录到 install 下的指令，添加的内容如代码清单 9-56 所示。

代码清单9-61 CMakeLists.txt

```
   ...

install(DIRECTORY
  urdf
  DESTINATION share/${PROJECT_NAME}
)

ament_package()
```

接着新建 fishbot_bringup 功能包，同样采用默认构建类型，我们将加载 URDF 和启动相关的命令放到该功能包中，在 src/fishbot_bringup 下新建 launch 目录，接着新建 urdf2tf.launch.py，该 launch 文件和 6.2.2 节的代码类似，编写的文件内容如代码清单 9-57 所示。

代码清单9-62 src/fishbot_bringup/launch/urdf2tf.launch.py

```
   import launch
import launch_ros
from ament_index_python.packages import get_package_share_directory


def generate_launch_description():
    # 获取默认路径
    urdf_tutorial_path = get_package_share_directory('fishbot_description')
    fishbot_model_path = urdf_tutorial_path + '/urdf/fishbot.urdf'
    # 为 launch 声明参数
    action_declare_arg_mode_path = launch.actions.DeclarelaunchArgument(
        name='model', default_value=str(fishbot_model_path),
        description='URDF 的绝对路径')
    # 获取文件内容生成新的参数
    robot_description = launch_ros.parameter_descriptions.ParameterValue(
        launch.substitutions.Command(
            ['cat ', launch.substitutions.launchConfiguration('model')]),
        value_type=str)
```

```
    # 状态发布节点
    robot_state_publisher_node = launch_ros.actions.Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        parameters=[{'robot_description': robot_description}]
    )
    # 关节状态发布节点
    joint_state_publisher_node = launch_ros.actions.Node(
        package='joint_state_publisher',
        executable='joint_state_publisher',
    )
    return launch.LaunchDescription([
        action_declare_arg_mode_path, joint_state_publisher_node,
        robot_state_publisher_node,
    ])
```

接着在 CMakeLists.txt 添加拷贝 launch 目录到 install 下，添加的内容如代码清单 9-58 所示。

代码清单9-63 CMakeLists.txt

```
   ...

install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}
)
ament_package()
```

重新构建工程，接着启动 launch 文件，启动命令及结果如代码清单 9-59 所示。

代码清单9-64 启动 urdf2tf.launch.py

```
$ ros2 launch fishbot_bringup urdf2tf.launch.py
---
[INFO] [launch]: All log files can be found below
/home/fishros/.ros/log/2023-10-02-12-58-40-842701-fishros-linux-16477
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [joint_state_publisher-1]: process started with pid [16479]
[INFO] [robot_state_publisher-2]: process started with pid [16481]
[robot_state_publisher-2] [INFO] [1696222720.896457344]
[robot_state_publisher]: got segment base_footprint
[robot_state_publisher-2] [INFO] [1696222720.896500115]
[robot_state_publisher]: got segment base_link
[robot_state_publisher-2] [INFO] [1696222720.896504164]
[robot_state_publisher]: got segment laser_link
[joint_state_publisher-1] [INFO] [1696222721.044186491]
```

```
[joint_state_publisher]: Waiting for robot_description to be published on the
robot_description topic...
```

关于是否发布 TF 变换，我们可以通过 rqt-tf-tree 插件进行验证，打开插件，就可以看到如图 9-24 所示的 TF 结构。

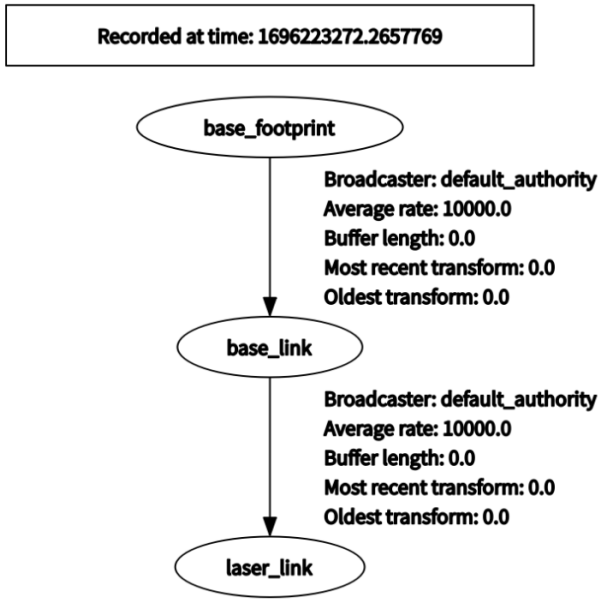


图 9-24 URDF 提供的 TF 变换

9.5.4 发布里程计 TF

里程计到机器人的坐标变换，表示的是机器人在里程计坐标系中的位置和姿态，这和里程计话题所表示的内容一致，我们只需要将里程计话题转换成里程计和机器人的坐标变换。

在 fishbot_ws/src/fishbot_bringup/src 下新建 odom2tf.cpp 文件，接着编写如代码清单 9-60 所示的内容。

代码清单9-65 fishbot_ws/src/fishbot_bringup/src/odom2tf.cpp

```
   #include <rclcpp/rclcpp.hpp>
#include <tf2/utils.h>
#include <tf2_ros/transform_broadcaster.h>
#include <geometry_msgs/msg/transform_stamped.hpp>
#include <nav_msgs/msg/odometry.hpp>

class OdomTopic2TF : public rclcpp::Node {
public:
  OdomTopic2TF(std::string name) : Node(name) {
    // 创建 odom 话题订阅者，使用传感器数据的 Qos
    odom_subscribe_ = this->create_subscription<nav_msgs::msg::Odometry>(
        "odom", rclcpp::SensorDataQoS(),
```

```
        std::bind(&OdomTopic2TF::odom_callback_, this,
std::placeholders::_1));
    // 创建一个 tf2_ros::TransformBroadcaster 用于广播坐标变换
    tf_broadcaster_ = std::make_unique<tf2_ros::TransformBroadcaster>(this);
  }

private:
  rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_subscribe_;
  std::unique_ptr<tf2_ros::TransformBroadcaster> tf_broadcaster_;
  // 回调函数，处理接收到的 odom 消息，并发布 tf
  void odom_callback_(const nav_msgs::msg::Odometry::SharedPtr msg) {
    geometry_msgs::msg::TransformStamped transform;
    transform.header = msg->header; // 使用消息的时间戳和框架 ID
    transform.child_frame_id = msg->child_frame_id;
    transform.transform.translation.x = msg->pose.pose.position.x;
    transform.transform.translation.y = msg->pose.pose.position.y;
    transform.transform.translation.z = msg->pose.pose.position.z;
    transform.transform.rotation.x = msg->pose.pose.orientation.x;
    transform.transform.rotation.y = msg->pose.pose.orientation.y;
    transform.transform.rotation.z = msg->pose.pose.orientation.z;
    transform.transform.rotation.w = msg->pose.pose.orientation.w;
    // 广播坐标变换信息
    tf_broadcaster_->sendTransform(transform);
  };
};

int main(int argc, char **argv) {
  rclcpp::init(argc, argv);
  auto node = std::make_shared<OdomTopic2TF>("odom2tf");
  rclcpp::spin(node);
  rclcpp::shutdown();
  return 0;
}
```

代码清单 9-60 中订阅了里程计话题，并在回调函数里发布坐标变换信息，需要注意的是，因为 odom 话题发布的服务质量为 best_effort，这里订阅时，使用 rclcpp::SensorDataQoS()进行匹配。

修改 CMakeLists.txt，注册 odom2tf 节点，添加的代码及位置如代码清单 9-61 所示。

代码清单9-66 CMakeLists.txt

```
   ...
find_package(rclcpp REQUIRED)
find_package(tf2 REQUIRED)
```

```
find_package(tf2_ros REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(nav_msgs REQUIRED)

add_executable(odom2tf src/odom2tf.cpp)
ament_target_dependencies(odom2tf
  rclcpp tf2  nav_msgs  geometry_msgs  tf2_ros
)
install(TARGETS odom2tf
  DESTINATION lib/${PROJECT_NAME})
...
ament_package()
```

重新构建功能包，首先运行 micro_ros_agent 让机器人接入，确保 odom 话题数据正常后，运行 odom2tf 节点，最后使用 rqt-tf-tree 查看 TF 结构，如图 9-25 所示。

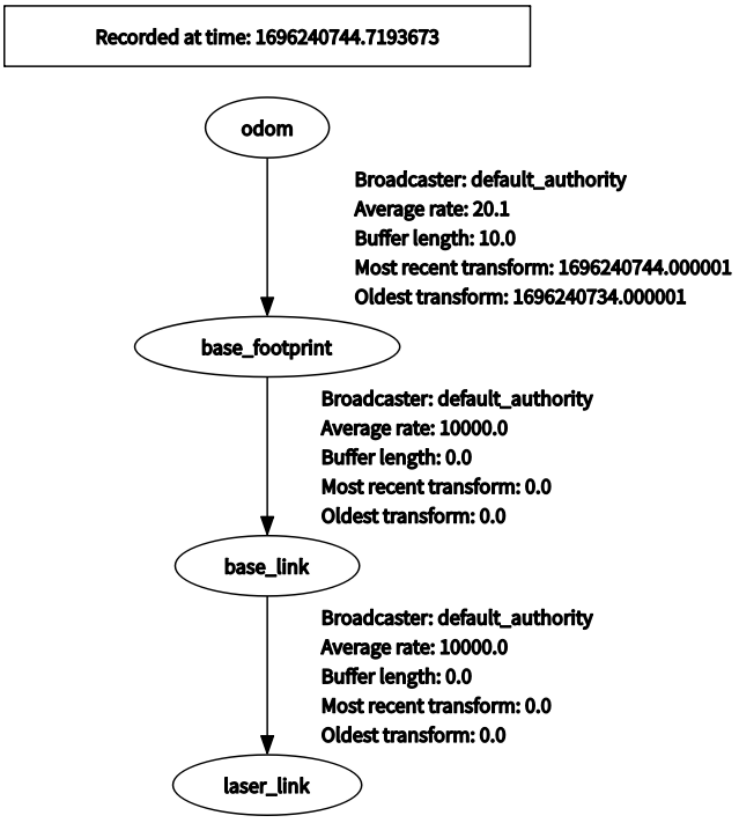


图 9-25 连接里程计后的 TF 树

9.5.5 完成机器人建图并保存地图

里程计、速度控制话题以及 odom 到 laser_link 之间的 TF 树都已经有了，我们就可以直接建图了。在建图前，可以将启动底盘和雷达的指令都放到一个 launch 里面，在 fishbot_ws/src/fishbot_bringup/launch 下新建 bringup.launch.py，接着编写如代码清单 9-62 所示的内

容。

代码清单9-67 fishbot_ws/src/fishbot_bringup/launch/bringup.launch.py

```
   import launch
import launch_ros
from ament_index_python.packages import get_package_share_directory
from launch.launch_description_sources import PythonLLaunchDescriptionSource

def generate_launch_description():
    fishbot_bringup_dir = get_package_share_directory(
        'fishbot_bringup')
    ydlidar_ros2_dir = get_package_share_directory(
        'ydlidar')

    urdf2tf = launch.actions.IncludeLaunchDescription(
        PythonLaunchDeLaunchDescriptionSource(
            [fishbot_bringup_dir, '/launch', '/urdf2tf.launch.py']),
    )

    odom2tf = launch_ros.actions.Node(
        package='fishbot_bringup',
        executable='odom2tf',
        output='screen'
    )

    microros_agent = launch_ros.actions.Node(
        package='micro_ros_agent',
        executable='micro_ros_agent',
        arguments=['udp4','--port','8888'],
        output='screen'
    )

    ros_serail2wifi =  launch_ros.actions.Node(
        package='ros_serail2wifi',
        executable='tcp_server',
        parameters=[{'serial_port': '/tmp/tty_laser'}],
        output='screen'
    )

    ydlidar = launch.actions.IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            [ydlidar_ros2_dir, '/launch', '/ydlidar_launch.py']),
    )
```

```
    # 使用 TimerAction 启动后 5 秒执行 ydlidar 节点
    ydlidar_delay = launch.actions.TimerAction(period=5.0, actions=[ydlidar])
    return launch.LLaunchDescription([
        urdf2tf,
        odom2tf,
        microros_agent,
        ros_serail2wifi,
        ydlidar_delay
    ])
```

代码清单 9-62 中，将多个节点启动放到同一个 launch 文件。因为雷达驱动依赖串口转 WIFI 驱动，所以使用 TimerAction，延时 5 秒后启动雷达驱动节点。保存并重新构建功能包，运行该节点，接着给机器人重新上电，在各个节点正常运行后，检查各话题和 TF 树是否正常，然后就可以进行建图了。

建图我们依然使用 slam_toolbox 进行，如果没有安装可以根据章节 7.2.1 的介绍进行安装，在新的终端运行 slam_toobox 并设置不使用仿真时间，命令及运行结果如代码清单 9-63 所示。

代码清单9-68 启动 slam_toolbox 进行在线建图

```
   $ ros2 launch slam_toolbox online_async_launch.py use_sim_time:=False
---
[INFO] [launch]: All log files can be found below
/home/fishros/.ros/log/2023-10-02-22-52-36-917373-fishros-linux-79801
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [async_slam_toolbox_node-1]: process started with pid [79802]
[async_slam_toolbox_node-1] [INFO] [1696258356.969203979] [slam_toolbox]:
Node using stack size 40000000
[async_slam_toolbox_node-1] [INFO] [1696258356.987488838] [slam_toolbox]:
Using solver plugin solver_plugins::CeresSolver
[async_slam_toolbox_node-1] [INFO] [1696258356.987606280] [slam_toolbox]:
CeresSolver: Using SCHUR_JACOBI preconditioner.
[async_slam_toolbox_node-1] Info: clipped range threshold to be within
minimum and maximum range!
[async_slam_toolbox_node-1] [WARN] [1696258357.129191771] [slam_toolbox]:
maximum laser range setting (20.0 m) exceeds the capabilities of the used
Lidar (8.0 m)
[async_slam_toolbox_node-1] Registering sensor: [Custom Described Lidar]
```

接着打开 RViz，修改 Fixed Frame 为 map，然后添加地图等插件，最后再打开终端运行在线建图节点就可以控制机器人进行建图了，RViz 插件及地图如图 9-26 所示。

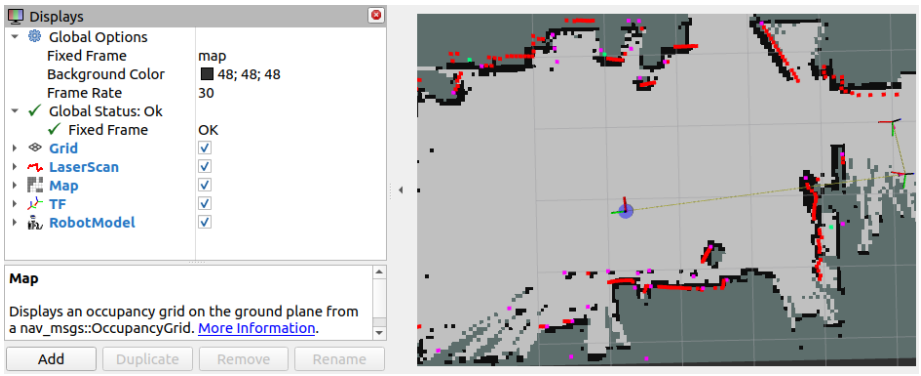


图 9-26 在 RViz 中查看地图

建好图就可以保存地图了，和仿真相同，我们使用 nav2_map_server 保存地图，如果没有安装该功能包，可以按照 7.2.2 节的介绍进行安装。

在 chapt9/fishbot_ws/src/下新建功能包 fishbot_navigation2，接着在功能包下新建 maps 目录，然后打开终端，进入 maps 目录，运行代码清单 9-64 中的命令来保存地图。

代码清单9-69 使用命令行保存地图

```
   $ ros2 run nav2_map_server map_saver_cli  -f room
---
[INFO] [1685008671.843881744] [map_saver]:
        map_saver lifecycle node launched.
        Waiting on external lifecycle transitions to activate
        See https://design.ros2.org/articles/node_lifecycle.html for more
information.
[INFO] [1685008671.847218662] [map_saver]: Creating
[INFO] [1685008671.847433320] [map_saver]: Configuring
[INFO] [1685008671.983234756] [map_saver]: Saving map from 'map' topic to
'room' file
[WARN] [1685008671.983298103] [map_saver]: Free threshold unspecified.
Setting it to default value: 0.250000
[WARN] [1685008671.983305316] [map_saver]: Occupied threshold unspecified.
Setting it to default value: 0.650000
[WARN] [map_io]: Image format unspecified. Setting it to: pgm
[INFO] [map_io]: Received a 376 X 222 map @ 0.05 m/pix
[INFO] [map_io]: Writing map occupancy data to room.pgm
[INFO] [map_io]: Writing map metadata to room.yaml
[INFO] [map_io]: Map saved
[INFO] [1685008672.264439985] [map_saver]: Map saved successfully
[INFO] [1685008672.265073478] [map_saver]: Destroying
```

保存好地图，打开 rqt-tf-tree，可以看到此时的 TF 树结构如图 9-27 所示。

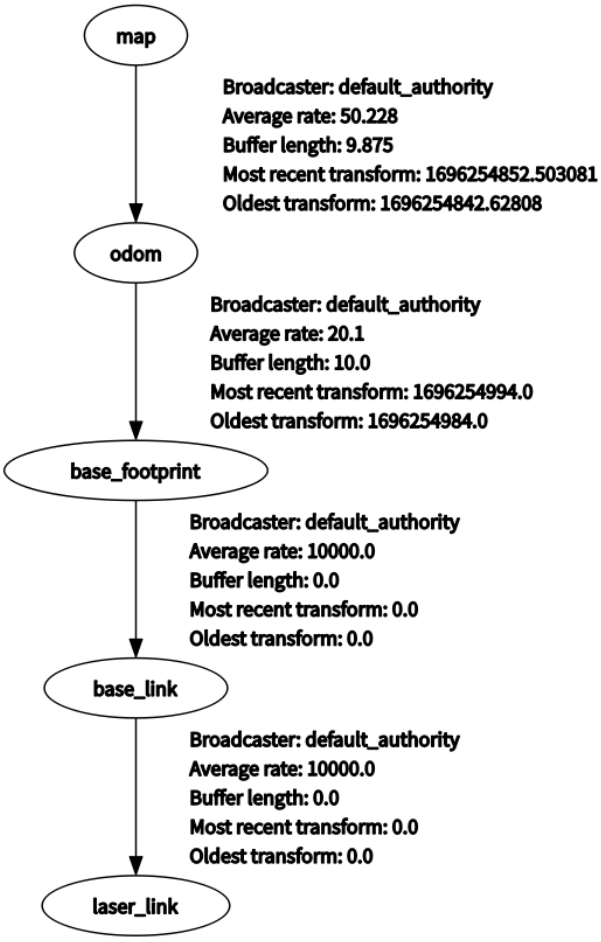


图 9-27 建图时的 TF 树

现在整个机器人的坐标变换结构和 9.5.2 节中介绍的相同，map 到 odom 之间的坐标变换是由 slam_toobox 维护的。从此可以看出 SLAM 除了建图还具备定位功能，在导航时若不运行 SLAM，定位功能则是由导航中的 AMCL 模块节点完成。

9.5.6 完成机器人导航

和使用仿真机器人导航相同，真实机器人使用导航只需要配置参数和 launch 文件即可，首先安装好 Navigation 2，接着在功能包 fishbot_navigation2 下创建 config 目录，然后将 nav2_bringup 提供的默认参数复制到 config 目录下，命令如代码清单 9-65 所示。

代码清单9-70 复制 nav2 参数到当前配置目录

```
   $ cp /opt/ros/$ROS_DISTRO/share/nav2_bringup/params/nav2_params.yaml
src/fishbot_navigation2/config
```

可以按照 7.3 节的介绍适当调节机器人半径等参数，接着就可以编写 launch 文件了，在 fishbot_navigation2 功能包下新建 launch 目录，然后在目录下新建 navigation2.launch.py，将 7.3.3 节中的 launch 配置直接复制到该文件中即可。最后修改 CMakeLists.txt，添加 launch、config 和

maps 三个目安装到 install 目录下的指令，然后重新构建功能包完成文件拷贝。

退出所有程序，再次启动 bringup.launch.py ，接着打开新的终端，运行 navigation2.launch.py 并设置不使用仿真时间，命令如代码清单 9-66 所示。

代码清单9-71 启动导航

```
   $ ros2 launch fishbot_navigation2 navigation2.launch.py
use_sim_time:=False
```

启动后可以看到 RViz 已经正确加载出我们建的地图了，但此时启动终端中会报 TF 相关的错误，这是因为我们还没有设定机器人初始位置，使用 2D Pose Estimmate 工具初始化位置，初始化完成位置之后的地图如图 9-28 所示。

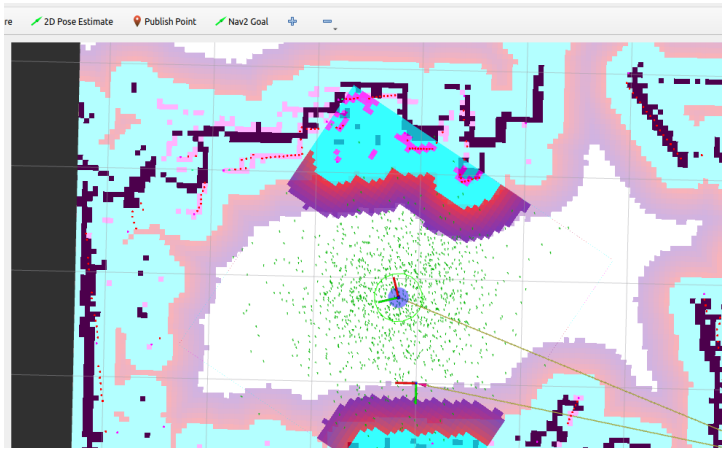


图 9-28 机器人和地图在 RViz 中的显示

接着使用 Nav2 Goal 就可以设置目标点进行导航了，此时会生成一条如图 9-29 所示的到达目标点的路径，并且机器人也动起来了。

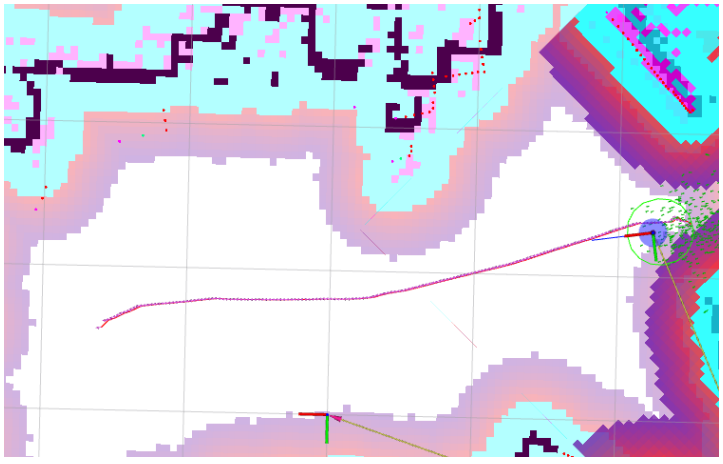


图 9-29 导航生成的从机器人到目标点之间的路径

好了，到这里我们就完成了真实机器人的建图和导航的开发，下面让我们对本章的内容做一个总结吧。

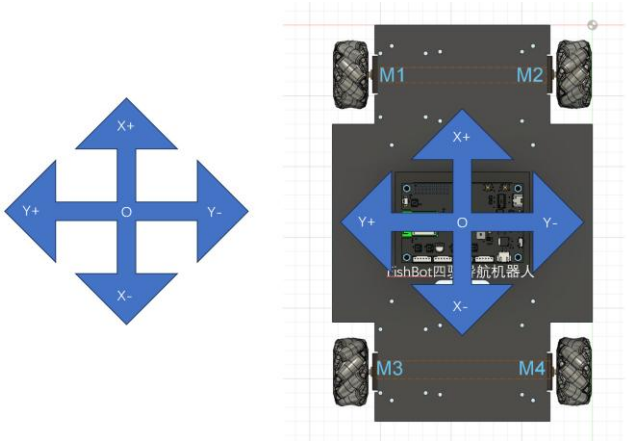
9.6 章节小结与点评

相比于仿真，实体机器人开发会让人觉得更加有趣，本章我们将重点放在了真实移动机器人的开发上，从学习单片机开发基础开始，学接着又习了如何一步步搭建移动机器人控制系统，然后学习了 micro-ROS 框架并将机器人接入到 ROS 2 中，最后学习了如何驱动雷达，并了解了移动机器人的坐标变换框架，最后实现了真实机器人的建图和导航。

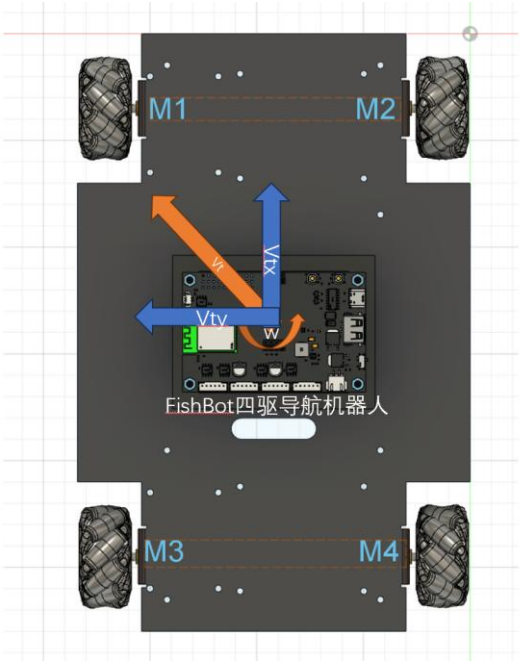
本章的内容很精彩，不过也困难重重，但都被你都一一解决了，让我们继续踏上后面的旅程吧。

9.7 附件

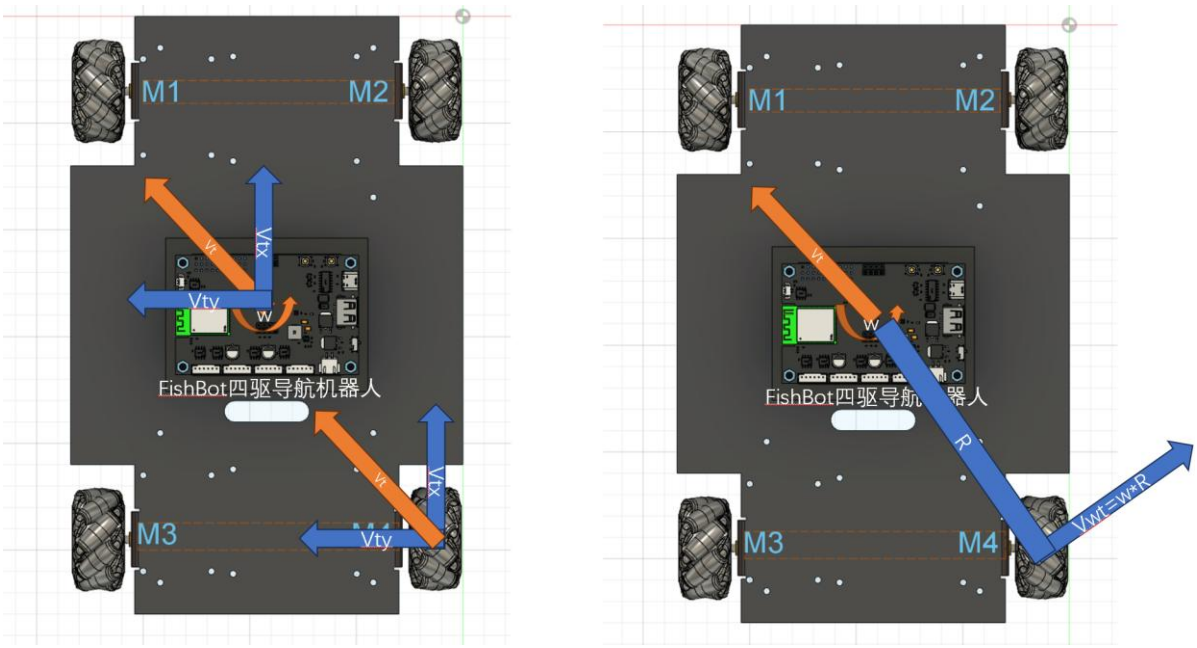
9.7.1 1：麦轮运动学正逆解分析



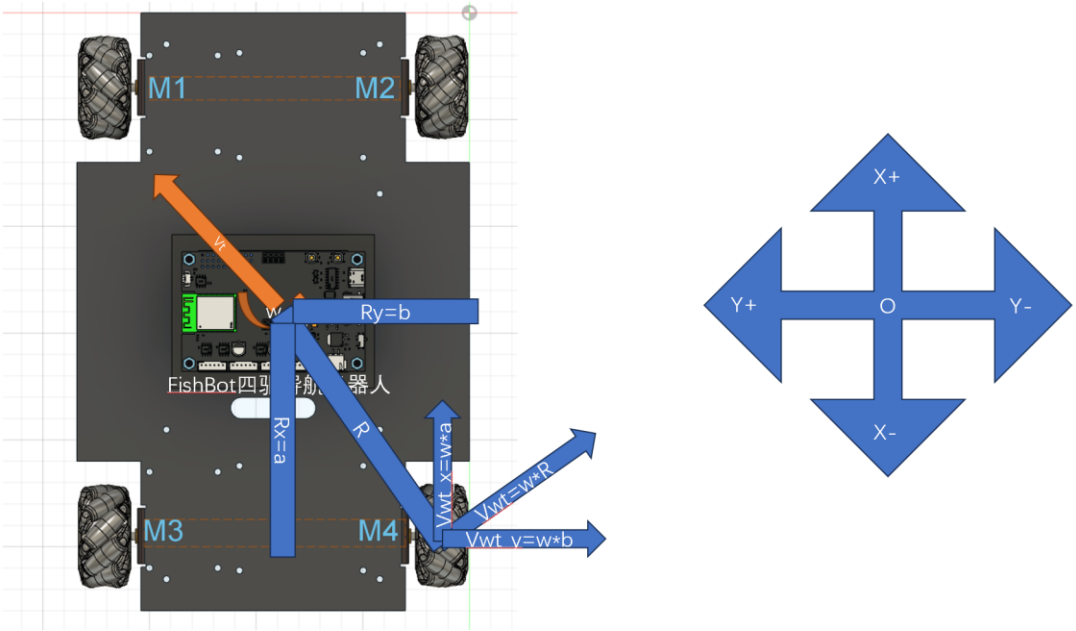
首先观察上图，我们先定一下坐标系和轮子编号，从左到右，从上到下依次是 M1、M2、M3、M4。机器人向前的方向是 X+(X 轴正方向，机器人的本体中心坐标系)。



现在机器人某一时刻的速度是 Vt，分解到 X 轴和 Y 轴是 Vtx,Vty，同时还有一个旋转速度 w。现在我们来看某一个轮子上的速度和中心速度的关系。以 M4 为例。



可以看到 M4 的速度是由两部分组成的，第一部分是速度 Vt 提供的 Vtx 和 Vty，第二部分是 w 提供的 Vwt=w*R，R 是半径。是机器人中心到 M4 中心的距离。

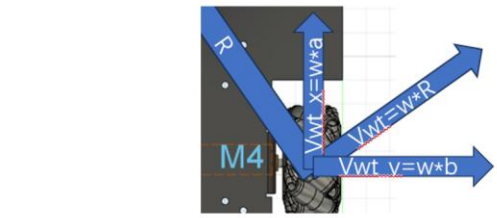


继续分解 Vwt，可以分解到 X 轴和 Y 轴，我们假设 X 方向距离 Rx=a，Y 方向距离 Ry=b

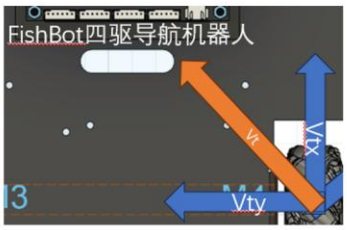
Vwt_x=w*a

Vwt_y=w*b

然后把速度加一下



VM4_x=Vtx+Vwt_x=Vtx+w*a
VM4_y=Vty-Vwt_y=Vty-w*b



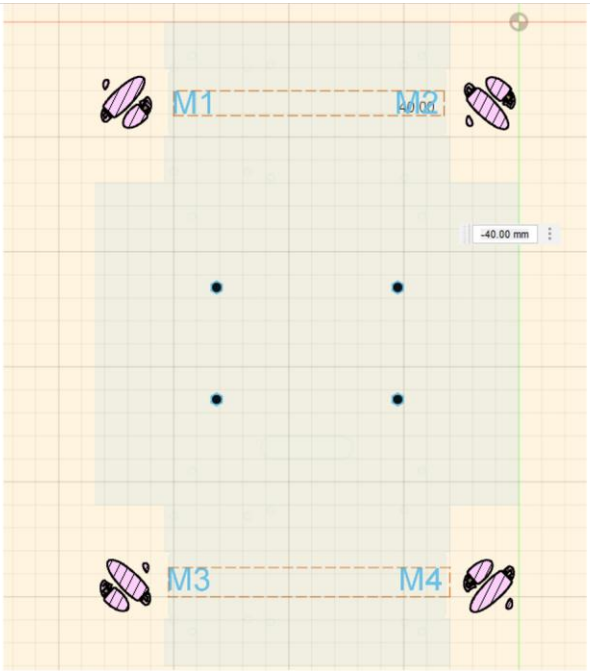
可以得到：

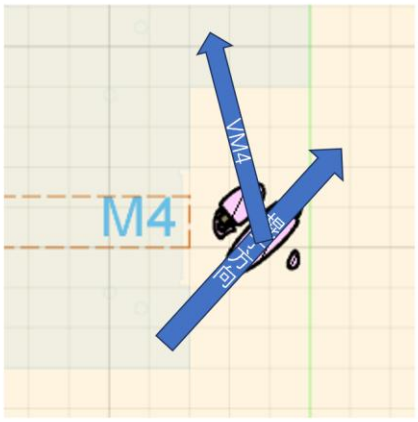
VM4_x=Vtx+Vwt_x=Vtx+w*a

VM4_y=Vty-Vwt_y=Vty-w*b

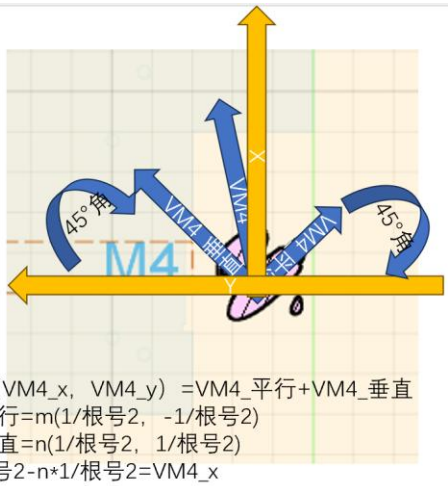
到这里就得到了单个轮子和机器人中心速度的关系了。

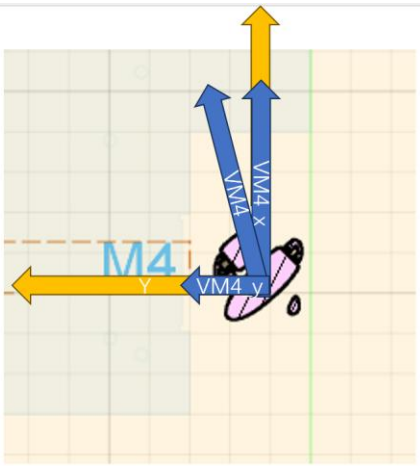
接着我们来看麦轮的模型是由上面的棍子 45 度角组成的，这是接触地面的样子。





最终 M4 的速度都是由棍子和地面摩擦力产生的，所以棍子的速度和 M4 的速度是有一定关系的。

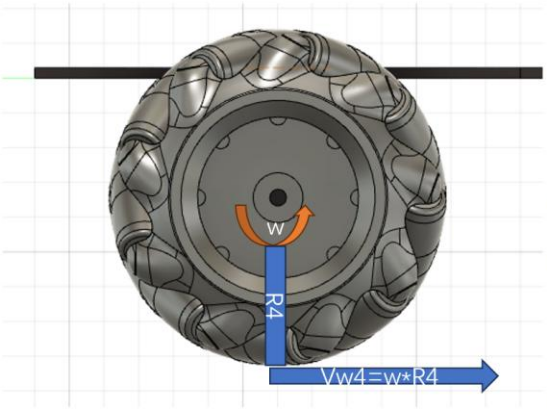


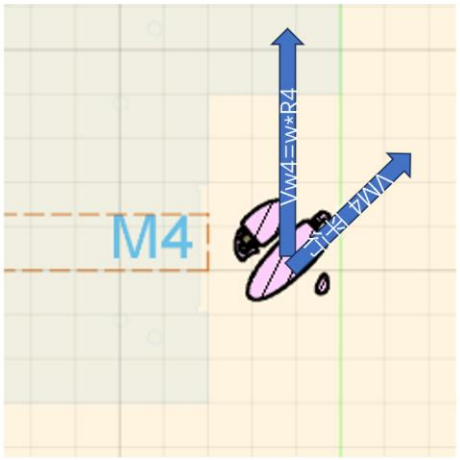


VM4=（VM4_x，VM4_y）=VM4_平行+VM4_垂直
VM4_平行=m(1/根号2，-1/根号2)
VM4_垂直=n(1/根号2，1/根号2)
m*1/根号2-n*1/根号2=VM4_x
m*1/根号2+1/n*根号2=VM4_y
m=（VM4_x-VM4_y）/根号2
VM4_平行=（（VM4_x-VM4_y）/2，（VM4_x-VM4_y）/2）
|VM4_平行|=根号（（VM4_x-VM4_y）/2）^2 +（VM4_x-VM4_y）/2）^2）
=1/根号2*VM4_x-1/根号2*VM4_y

我们把 VM4 分解一下成平行和垂直棍子两个方向，因为棍子和 X 轴和 Y 轴的夹角是 45 度（机械结构决定的），所以通过分解可以得出 VM4_ 平行=1/根号 2*VM4_x-1/根号 2*VM4_y。

垂直方向可以忽略，垂直方向上的力只能带动棍子自转而已，对整个底盘的速度没有贡献。





Vw4=w*R4

|Vw4|=|V4M_平行|/cos(45)

刚才我们分析的是轮子中心的速度，接着上面这张图我们来看轮子中心的速度和轮子表面速度的关系。

|Vw4|=|V4M_平行|/cos(45)

加上之前的结果

VM4_x=Vtx+Vwt_x=Vtx+w*a

VM4_y=Vty-Vwt_y=Vty-w*b

|V4M_平行|=1/根号2*VM4_x-1/根号2*VM4_y
VM4_x=Vtx+Vwt_x=Vtx+w*a
VM4_y=Vty-Vwt_y=Vty-w*b

|Vw4|=|V4M_平行|/cos(45)
=(1/根号2*VM4_x-1/根号2*VM4_y)/(1/根号2)
=VM4_x- VM4_y
=Vtx+w*a-(Vty-w*b)
= Vtx-Vty+w(a+b)

重复上面的步骤：
VM1 = Vtx-Vty-w(a+b)
VM2 = Vtx+Vty+w(a+b)
VM3 = Vtx+Vty-w(a+b)
VM4 = Vtx-Vty+w(a+b)

```
101  void Kinematics::kinematic_inverse(float linear_x_speed, float linear_y_speed,
     float angular_speed,
102                                     float &out_wheel_speed1, float &
                                        out_wheel_speed2, float &out_wheel_speed3,
                                        float &out_wheel_speed4)
103  {
104      const float a = 108.0f;
105      const float b = 88.5f;
106
107      out_wheel_speed1 = linear_x_speed - linear_y_speed - angular_speed * (a + b);
108      out_wheel_speed2 = linear_x_speed + linear_y_speed + angular_speed * (a + b);
109      out_wheel_speed3 = linear_x_speed + linear_y_speed - angular_speed * (a + b);
110      out_wheel_speed4 = linear_x_speed - linear_y_speed + angular_speed * (a + b);
111
112  }
```

就可以得到

$$|Vw4|== Vtx-Vty+w(a+b)$$

同理可以得到

$$VM1 = Vtx-Vty-w(a+b)$$

$$VM2 = Vtx+Vty+w(a+b)$$

$$VM3 = Vtx+Vty-w(a+b)$$

$$VM4 = Vtx-Vty+w(a+b)$$

对应代码

```
101  void Kinematics::kinematic_inverse(float linear_x_speed, float linear_y_speed,
     float angular_speed,
102                                     float &out_wheel_speed1, float &
                                        out_wheel_speed2, float &out_wheel_speed3,
                                        float &out_wheel_speed4)
103  {
104      const float a = 108.0f;
105      const float b = 88.5f;
106
107      out_wheel_speed1 = linear_x_speed - linear_y_speed - angular_speed * (a + b);
108      out_wheel_speed2 = linear_x_speed + linear_y_speed + angular_speed * (a + b);
109      out_wheel_speed3 = linear_x_speed + linear_y_speed - angular_speed * (a + b);
110      out_wheel_speed4 = linear_x_speed - linear_y_speed + angular_speed * (a + b);
111
112  }
```

把这个结果联合起来求 Vtx,Vty 和 w 之间的关系

$$VM1 = Vtx-Vty-w(a+b)$$

$$VM2 = Vtx+Vty+w(a+b)$$

$$VM3 = Vtx+Vty-w(a+b)$$

$$VM4 = Vtx-Vty+w(a+b)$$

解 Vtx：

$$VM1+VM2+VM3+VM4=4Vtx$$

$$Vtx=(VM1+VM2+VM3+VM4)/4$$

解 Vty：

$$VM2-VM1=2Vty+2w(a+b)$$

$$VM3-VM4=2Vty-2w(a+b)$$

$$VM2-VM1+VM3-VM4=4Vty$$

$$Vty=(-VM1+VM2+VM3-VM4)/4$$

解 w：

$$VM4-VM3=-2Vty+2w(a+b)$$

$$VM2-VM1=2Vty+2w(a+b)$$

$$VM4-VM3+VM2-VM1=4w(a+b)$$

$$w=(-VM1+VM2-VM3+VM4)/(4*(a+b))$$

写成代码:

```
114 void Kinematics::kinematic_forward(float wheel1_speed, float wheel2_speed, float wheel3_speed,
    float wheel4_speed,
115                                   float &linear_x_speed, float &linear_y_speed, float &
                                      angular_speed)
116 {
117     // TODO : 参数化
118     const float a = 108.0f;
119     const float b = 88.5f;
120 
121     // 计算机器人的 x 轴线速度，公式为四个轮子转速之和的平均值。
122     linear_x_speed = (wheel1_speed + wheel2_speed + wheel3_speed + wheel4_speed) / 4.0f;
123     linear_y_speed = (-wheel1_speed + wheel2_speed + wheel3_speed - wheel4_speed) / 4.0f;
124     angular_speed = float(-wheel1_speed + wheel2_speed - wheel3_speed + wheel4_speed) / (4.0f
    * (a + b));
125 }
```