

智能遥控战车 设计说明书 ——全向轮

2025年4月19日



浙狮战队

**浙江师范大学新工科人才培养创新**

**工作室人员名单**

（按姓氏拼音排序）

**学小心生**：

陈心强 陈永杰 黄雨晴 金陈超 李 瑞 李兴龙 洛桑尼玛李予洋 林昱轩 林振宇 李子俊 马下众 牟心舒 潘震翊

汤健明 徐诗超 谢逸飞 姚茂涛 杨宫羽 周佳豪 赵一帆

赵祖安 张俊博 张下泓 朱 宁

**指导老师**：

樊下俊 兰下虎 刘中中 潘下睿 邵金均

目 录

[第一部分 机械结构设计 1](#_Toc15708)

[1.1 智能遥控战车机械分析 1](#_Toc23069)

[1.1.1 底盘设计方案 2](#_Toc2577)

[1.1.1.1 井字型框架 2](#_Toc20988)

[1.1.1.2 避震结构 3](#_Toc29587)

[1.1.1.3 全向轮轮组 4](#_Toc24762)

[1.1.2 云台设计方案 4](#_Toc10674)

[1.1.3 发射机构设计方案 5](#_Toc19186)

[1.1.3.1 供弹机构 6](#_Toc10852)

[1.1.4 导航机构 8](#_Toc23401)

[1.2 材料选择 9](#_Toc20344)

[1.3 结构装配 9](#_Toc15973)

[第二部分 嵌入式控制开发 11](#_Toc12037)

[2.1 全向轮哨兵总体控制 11](#_Toc2268)

[2.1.1 诉求分析 11](#_Toc3169)

[2.1.2 软件设计 12](#_Toc21028)

[2.1.3 系统架构 12](#_Toc17813)

[2.1.4 控制框架 14](#_Toc15363)

[2.1.5 功能概述 15](#_Toc18858)

[2.1.6 底盘任务 16](#_Toc25676)

[2.1.7 云台任务 18](#_Toc18001)

[2.1.8 平台任务 19](#_Toc21621)

[2.1.9 射击任务 20](#_Toc28577)

[2.2 底盘部分 20](#_Toc26310)

[2.2.1底盘模式判断 20](#_Toc6922)

[2.2.2全向轮速度解算 22](#_Toc5649)

[2.2.3底盘电机通讯控制 24](#_Toc28341)

[2.2.4底盘硬件离线检测初始化 26](#_Toc5137)

[2.2.5底盘通信线程初始化 26](#_Toc12503)

[2.3 云台部分 27](#_Toc9189)

[2.3.1 云台模式判断 27](#_Toc24271)

[2.3.2 云台控制算法 28](#_Toc24898)

[2.3.3 云台电机通讯控制 29](#_Toc12807)

[2.3.4 自瞄模式 30](#_Toc8783)

[2.3.4.1 自瞄数据获取 30](#_Toc10469)

[2.3.4.2 视觉数据结构体 31](#_Toc20809)

[2.3.4.3 视觉数据更新 32](#_Toc7257)

[2.3.5 云台硬件离线检测 33](#_Toc17505)

[2.3.6 云台通信线程初始化 33](#_Toc28364)

[2.3.7 视觉交互数据回调函数 34](#_Toc3445)

[2.4 平台部分 35](#_Toc1904)

[2.4.1 平台模式判断 35](#_Toc18570)

[2.4.2 平台控制算法 36](#_Toc9549)

[2.4.3 平台电机通讯控制 37](#_Toc26427)

[2.4.4 雷达巡航模式 38](#_Toc12161)

[2.4.5 平台硬件离线检测 39](#_Toc10355)

[2.4.6 平台通信线程初始化 39](#_Toc5244)

[2.4.7 决策信息回调函数 40](#_Toc10035)

[2.5.射击部分 42](#_Toc6223)

[2.5.1 射击任务 42](#_Toc40)

[2.5.2 射击判断 43](#_Toc12869)

[2.5.2.1 射击判断（遥控器控制段） 43](#_Toc19811)

[2.5.2.2 射击判断（键鼠控制段） 45](#_Toc23641)

[第三部分 视觉系统设计 47](#_Toc28792)

[3.1 背景与目标 47](#_Toc2888)

[3.2 视觉自动瞄准算法设计 47](#_Toc9920)

[3.3 设计方案 48](#_Toc31825)

[3.3.1目标检测 48](#_Toc158)

[3.3.2目标运动解算 53](#_Toc11751)

[3.3.3 火控策略 62](#_Toc23538)

[第四部分 导航算法 65](#_Toc17105)

[4.1 背景与目标 65](#_Toc17185)

[4.2 导航避障算法设计 65](#_Toc19356)

[4.3 设计方案 65](#_Toc23896)

[4.3.1 地图建立算法 66](#_Toc4170)

[4.3.2 重定位算法 73](#_Toc16888)

[4.3.3 目标点发布算法 77](#_Toc1168)

[4.3.4 路径规划算法（Teb算法） 81](#_Toc20556)

[第五部分 硬件设计 89](#_Toc9021)

[5.1 全向轮战车基础硬件设计 89](#_Toc30194)

[5.1.1 全向轮战车器件 89](#_Toc4458)

[5.2 全向轮战车特殊硬件设计 91](#_Toc25000)

[5.2.1 总体方案 91](#_Toc12461)

[5.2.2底盘接线板PCB设计 91](#_Toc19797)

[5.2.3滑环转接PCB设计 93](#_Toc16229)

[5.2.4超级电容系统 93](#_Toc8584)

[5.2.4.1总体方案 93](#_Toc24788)

[5.2.4.2控制部分 94](#_Toc1477)

[5.2.4.3功率部分 95](#_Toc29350)

[5.2.4.4 关键器件选型 96](#_Toc19538)

[5.2.4.5 相关参数计算 97](#_Toc1045)

**第一部分 机械结构设计**

1.1 智能遥控战车机械分析

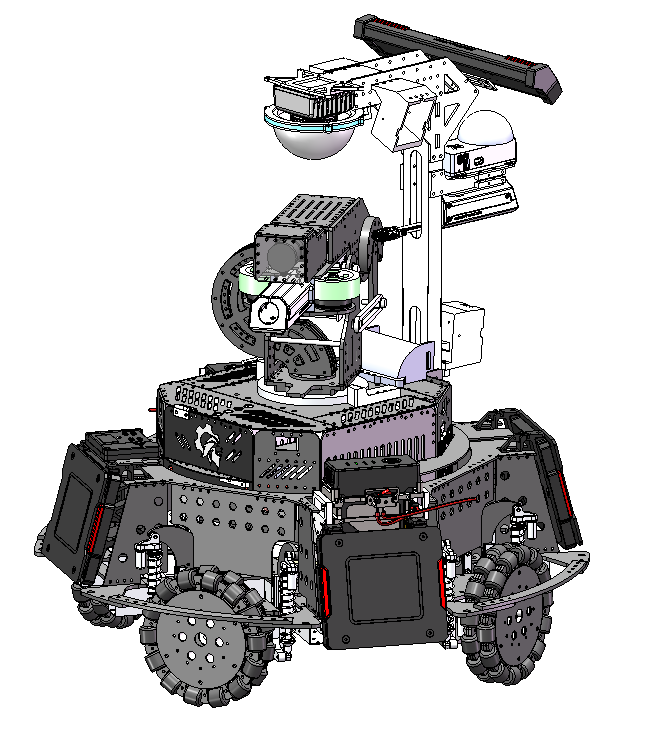
结合新工科人才培养创新工作室项目制课程教学的任务要求，以设计制造智能遥控战车为项目目标，此智能遥控战车对机械结构（如图 1-1）的整体设计要求如下：

（1）尺寸要求：初始尺寸要求不超过 700mm×700mm×700mm（长×宽×高），最大伸展尺寸不超过 800mm×800mm×800mm；

（2）重量要求：最大重量不超过 28.47kg；

（3）功能要求：能实现全自动导航和移动并稳定发射17mm 弹丸；

（4）设计制作周期：6个月（组队）。



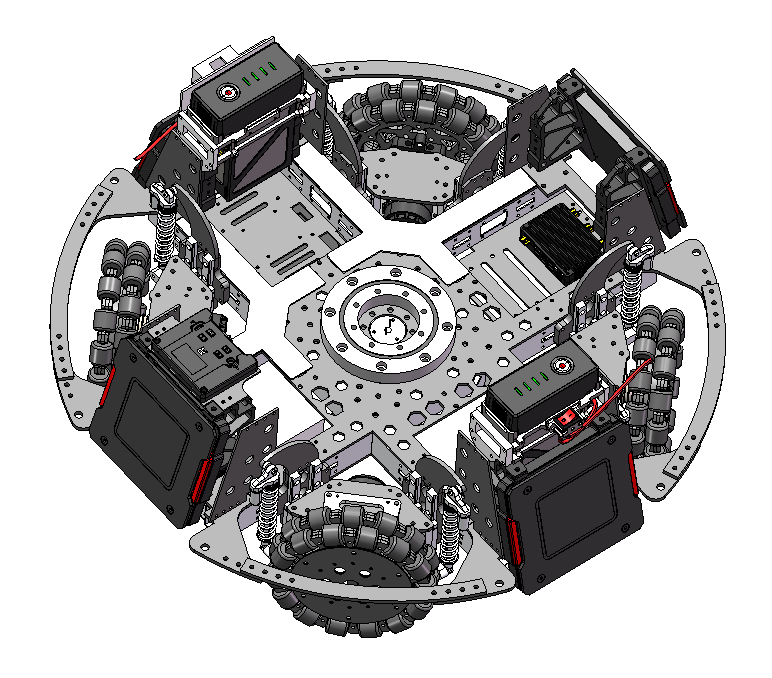
**图 1-1 全向轮智能遥控战车**

本项目为了研发出性能更加优异且稳定可靠的智能遥控战车，在对该项目任务及规则进行了深刻解读的基础上，对全向轮机器人的各项功能进行了需求分析。整车采用碳纤维板材、铝材和PLA打印材料，较大地减轻了重量，合理归置重心，为后续机器人稳定性做了充足准备。

**1.1.1 底盘设计方案**

底盘的整体结构设计采用井字型框架，该结构保证轮组、避震、电池等机械部件在战车整体上均为对称分布，并置于框架外围。这种结构既为云台设计预留了更大的装配空间，又起到减小重心偏移的作用，增强了战车的稳定性。

如图 1-2 所示：

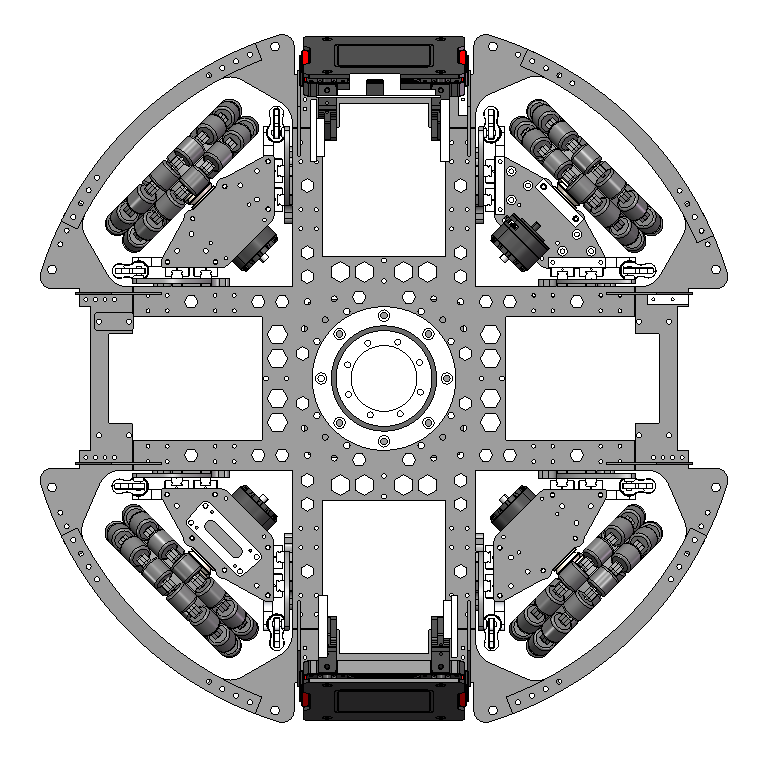


**图 1-2 全向轮智能遥控战车底盘**

**1.1.1.1 井字型框架**

底盘结构采用井字型框架，结构简洁清晰并中心对称，减小重心偏移，有利于增强结构稳定性。采用铝方管搭建并覆以碳纤维板材加固，提高底盘承载强度与稳定性的同时尽可能减轻重量，增强机器人机动性。

如图 1-3 所示：



**图 1-3 井字型框架**

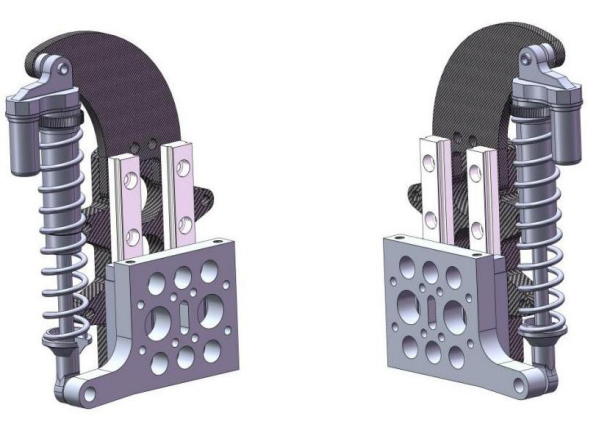
**1.1.1.2 避震结构**

为增强机器人在不同地形环境下的适应能力，需要添加避震结构设计。

该避震结构采用线性滑轨避震结构，该结构由弹簧、线性滑轨，铝合金加工件和碳纤维板材悬挂构成，可以起到较好的减震缓冲作用。该避震结构可以保证无论悬挂的弹簧如何受压变形，线性滑轨都可以校正受力方向，使弹簧只受到轴向的作用力，保证轮组相对于地面的直立，限制轮组的偏移，提高机器人稳定性。

采用线性滑轨避震结构的机器人能够在高速、高精度运动中保持平稳，提高机器人的稳定性和精度，减少故障率和维护成本，从而提高机器人的运行效率和使用寿命。

如图 1-4 所示：

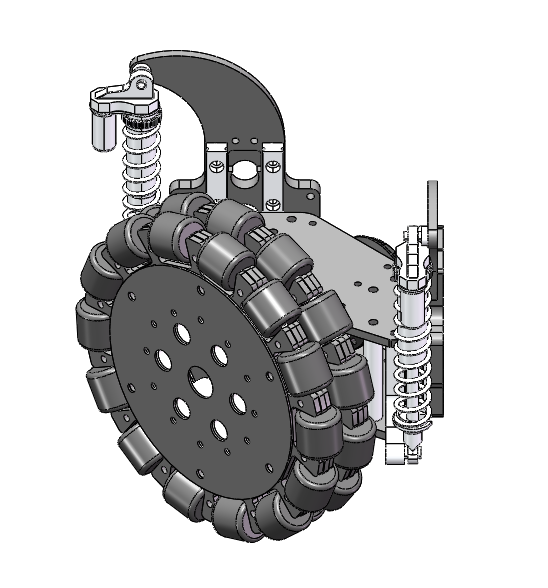


**图 1-4 避震悬挂**

**1.1.1.3 全向轮轮组**

全向轮赋予机器人卓越的机动性和操控能力，能够快速适应各种环境和任务需求。此外，全向轮设计还提供精准的定位能力，通过控制轮组的运动实现高精度的定位和导航。采用全向轮的机器人能够实现灵活多样的运动方式，包括前进、后退、转向、平移和旋转，使其在狭小空间内移动更加便捷高效。

该全向轮轮组使用碳纤维板材、PLA打印材料以及聚氨酯辊子组成，采用 M3508电机驱动整个全向轮轮组运动通过辊子之间和整体的转动实现全向移动，且整体采用碳纤维制作，强度高、质量轻，有利于机器人的轻量化设计。如图 1-5 所示：



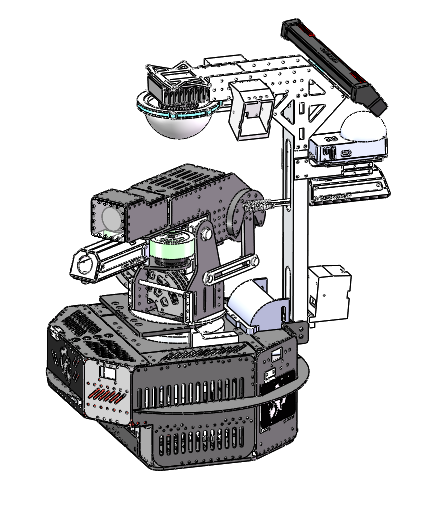
**图 1-5 全向轮轮组**

**1.1.2 云台设计方案**

为让机器人具有快速识别响应以及强大的跟随打击能力，机器人的云台采用双Yaw轴云台设计（如图 1-7）。

该云台的下方大Yaw轴由翎控9025电机驱动，电机中部安装滑环，实现底盘与云台的数据传递，上方小Yaw轴由GM6020电机驱动，通过精确的嵌入式控制实现大小Yaw轴分离转动。该设计可以将控制小电脑与弹舱等重量大的结构放在大Yaw上，大大减轻小Yaw重量，降低转动惯性，进一步提高转动速度与精确度。

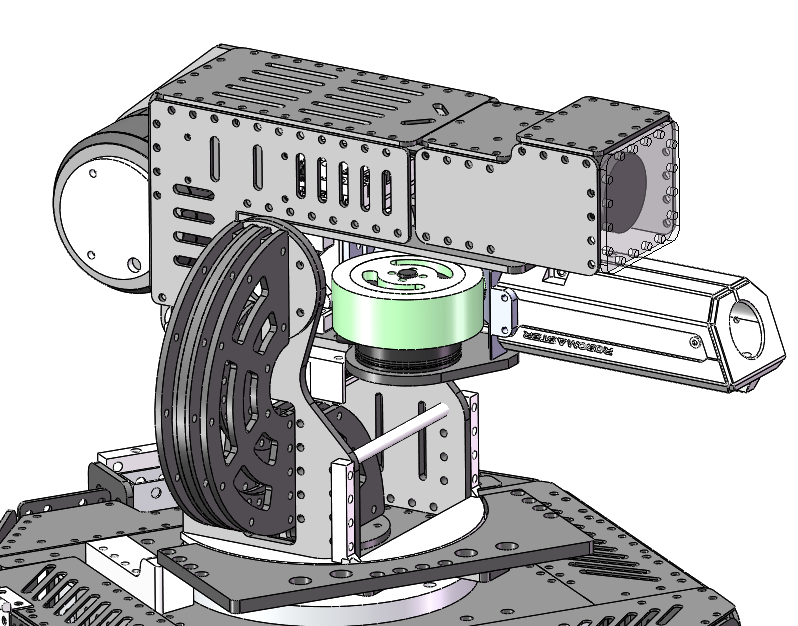
另外，小Yaw轴云台的Pitch轴由GM6020电机采用连杆形式驱动，安装简单，便于维护。该方法将电机后置用于配平小Yaw轴云台前后重量，降低Pitch轴转动惯性，提高电机响应速度与控制精度，增强机器人整体的射击精度。



**图 1-7 双Yaw轴云台**

**1.1.3 发射机构设计方案**

发射机构由橡胶摩擦轮和拆除减速箱的M3508电机组成，分别有两组对称分布，通过调整摩擦轮间距，使弹丸经过电机的驱动的摩擦轮通过挤压发射，并在摩擦轮后方设计单发限位，由SG10轴承限位，确保正常情况下低头不溜弹，单发时不多发，连发时补卡弹。如图 1-8。



**图 1-8 发射机构**

**1.1.3.1 供弹机构**

供弹机构由拨弹机构（图1-9），储弹机构（图1-10）和供弹链路（图1-11）组成。

拨弹机构采用中心供弹机构，工作原理为GM2006电机直接驱动小齿轮转动，小齿轮通过齿轮啮合带动大齿轮，12孔拨弹盘跟随大齿轮一同转动，通过拨弹盘的转动将孔内的弹丸依次通过弯道从正中心输出。拨弹机构位于大Yaw轴云台中心，减轻小Yaw轴云台的负担，减小转动时的惯性力。

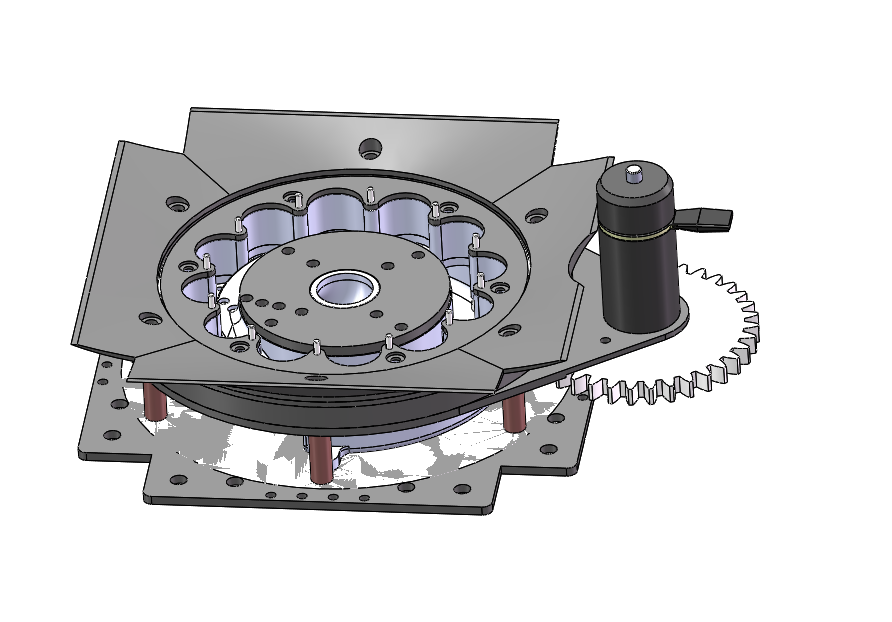
缺点在于该拨弹机构传动时存在齿轮减速比，导致拨盘的转速降低，从而导致发射的弹频降低。

储弹机构由碳纤维板材和PLA打印件围成，较大的储弹机构可以容纳更多的弹丸，这意味着更长的输出时间，更强的火力压制。同时将储弹机构放在大Yaw轴云台中心，可以减少小Yaw轴云台的负担，让云台更灵活，精确度更高。

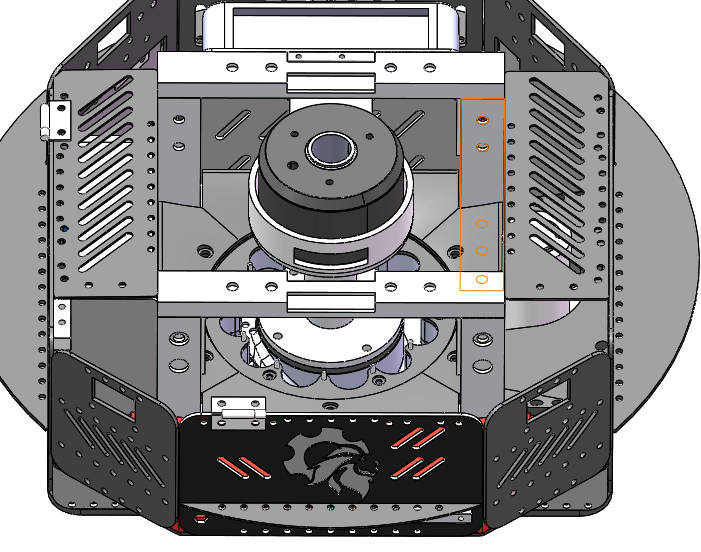
缺点在于处于云台中心，若有杂物落入其中容易导致卡弹，且无法直接从外部看到拨盘底部，不便维修。

供弹链路由双法兰铝合金加工件，GM6020电机通孔，碳纤维板材组合通道和PLA打印弯管组成。弹丸从储弹机构经过拨盘机构的中心供弹从云台中心输出，依次经过上述四结构的通道，最后到达发射机构挤压摩擦轮射出。该设计可以保证不论云台旋转时供弹链路长度始终保持不变，可以一直稳定持续的输出弹丸，结构稳定简洁，保证供弹与发射时的稳定性。

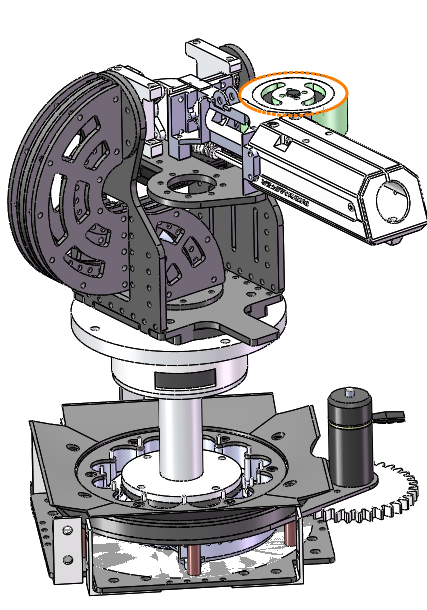
缺点在于供弹链路较长，供弹过程中链路储弹较多，弹丸之间、弹丸与链路之间均有摩擦，若发射机构的单发限位不合适会导致卡弹。



**图 1-9 拨弹机构**



**图 1-10 储弹机构**

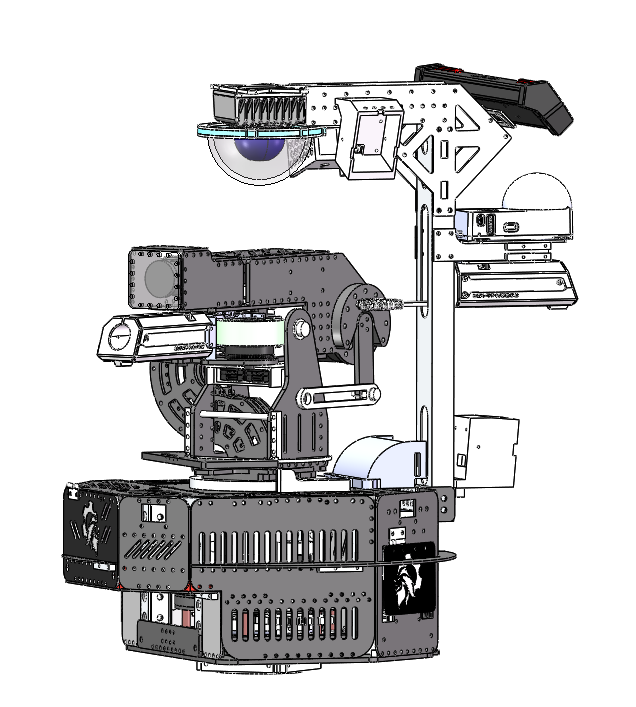


**图 1-11 供弹链路**

**1.1.4 导航机构**

该机器人要求可以全自动导航移动，这就需要雷达来检测障碍物与规划路线，从而提高运动的精确性。

该导航机构（如图1-12）除mid360导航雷达外还可以安装三个感知相机，分别位于左方、右方和后方，可以基本实现全视野观测。雷达安装在雷达架上，处于整个机器人的中心，可以极大限度地识别到周围的环境，进行实时演算，正确准确地导航机器人。



**图 1-12 导航机构**

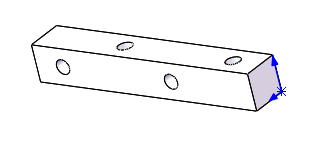
**1.2 材料选择**

机器人整体零件为碳纤维板材，铝合金加工件和3D 打印件，主要工艺为 2D 雕刻，车铣加工和3D打印技术。在机器人底盘处主要由铝方管覆以碳纤维板材搭建骨架，轻量化的同时强度也得到了保证；在机器人外壳部分主要使用碳板，适量选择碳板厚度及镂空，做到极致减重；而在需要高精度的地方，如枪管发射头，本项目选择使用3D打印技术，可以进行快速及时的调整修改，提升机器人的射击精准；以上几种加工工艺都非常常用，能保证机器人基本精度要求，而且加工周期较短，成本也不会太高。

**1.3 结构装配**

全向轮机器人主体大部分连接采用内六角杯头螺丝和防松螺母，全车无焊接和黏贴方式连接与装配，方便后续组装设计机器人时拆卸与优化，方便对损坏和磨损的零件进行更换。相比于传统的十字金属螺丝，采用内六角杯头的螺丝可以防止多次拆卸组装时产生的滑丝问题和摩擦力太小的问题，此外再配合防松螺母的装配组装，可以更加有效地连接固定各个零件，防止在机器人运作时产生的螺丝松动问题。

在一些不能使用导电板材的地方，如PCB电路板的安装，则采用3D打印件垫底隔离的连接方式。另外 本项目使用了大量标准件——四孔直角连接件（图 1-13），方便机器人壳体的拼接。



**图 1-13 四孔直角连接**

**第二部分 嵌入式控制开发**

2.1 全向轮哨兵总体控制

2.1.1 诉求分析

在嵌入式控制开发之前，需要进行诉求分析。将全向轮哨兵战车的功能拆分为底盘功能、云台/平台功能、武器系统、其他功能四个方向，并针对这四个方向进行全面分析，列出需要完成的内容。然后再进行功能量化，将任务细化，方便后续的展开。

**表2-1 控制参数**

|  |  |
| --- | --- |
| 参数 | 描述 |
| 电容容量 | 总储存能量1999J，工作电压范围12V |
| 最大工作电压 | 24V |
| 电机数目与用途 | GM6020×4：云台Pitch与Yaw轴驱动  M3508×10：底盘驱动、拨弹电机、摩擦轮电机 |

图示

AI 生成的内容可能不正确。

**图2-1 功能定义**

2.1.2 软件设计

软件项目使用CubeMX软件配置，STM32HAL库编写，参考Robomaster开发板C型开源程序，重新构建了较为简洁的源码。开发环境为STM32CubeMX+MDKARM，使用MDK的调试功能和串口等观察数据，进行调试。

2.1.3 系统架构

首先，总览框架的设计模式：

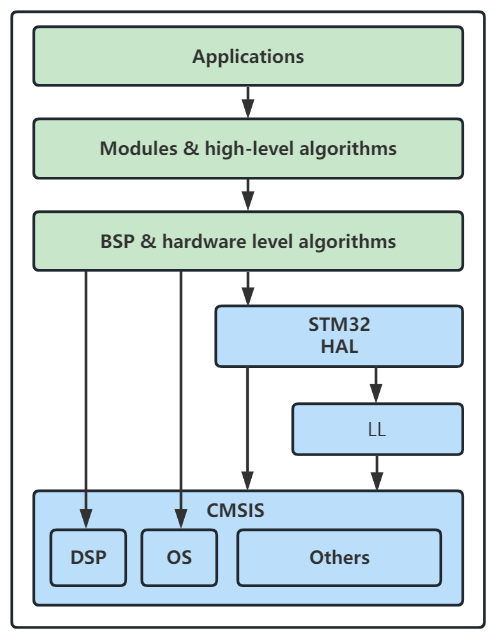
框架在结构上分为三层: BSP/Components/Application。整体使用的设计模式是结构层级模式，即每个“类”包含需要使用的底层“类”，通过组装不同的基础模块实现更强大的功能。而最顶层的Application之间则过pub-sub消息机制进行解耦，使得在编写代码时不会出现相互包含的情况。

我们希望通过BSP对硬件的抽象使得Components的编写更为轻松，不需要考虑底层的硬件具体是如何运作的;再通过Components的外接模块的抽象，使得Application的编写可以通过完全硬件无关的方式考虑，达到只阅读Components说明文档就能迅速开发应用的程度。BSP和Components的设计愿景，就是成为人们常说的中间件。

pub-sub机制的体现:以全向轮哨兵战车软件的Application层为例，含了chassis、gimbal、platform、shoot、cmd五个应用，每个应用都对应了机器人上的不同模组。cmd应用负责从机器人控制信号来源(遥控器/环境传感器）处获取信息并解析成各个执行单元的实际动作(电机/舵机/速度/位置/角度/开度等)并将此信息发布出去。chassis、gimbal、platform、shoot等包含了执行单元的应用则订阅这些消息，并通过自己包含的子模块，调用它们的接口实现动作。

结构层级模式的体现:以chassis应用为例，chassis中包含了4个底盘电机模块。当chassis收到cmd应用的信息，希望让底盘以1m/s的速度前进。chassis首先根据底盘的类型(舵轮/麦克纳姆轮/全向轮）以及对应的动力学/运动学解算函数，计算得到每个电机的目标输入值，此时chassis将输入通过电机模块(motor components)的接口将设定值告知电机。而每个电机模块又有各自的PID计算模和自身电流&速度&角度传感器的信息，可以计算出最终需要的电流设定值。假设该电机使用CAN协议与电调通信，则电机通过自身包含的CANInstance(bsp\_can提供)用于实际硬件交互，电机模块将设定值电流值或其他指令按照通信协议组织在CAN报文中，通过CANInstance提供的接口，把最终控制数据发送给电调，实现控制闭环。从调用来看，三个层级的包含关系为chassis∈motor∈bspcan。

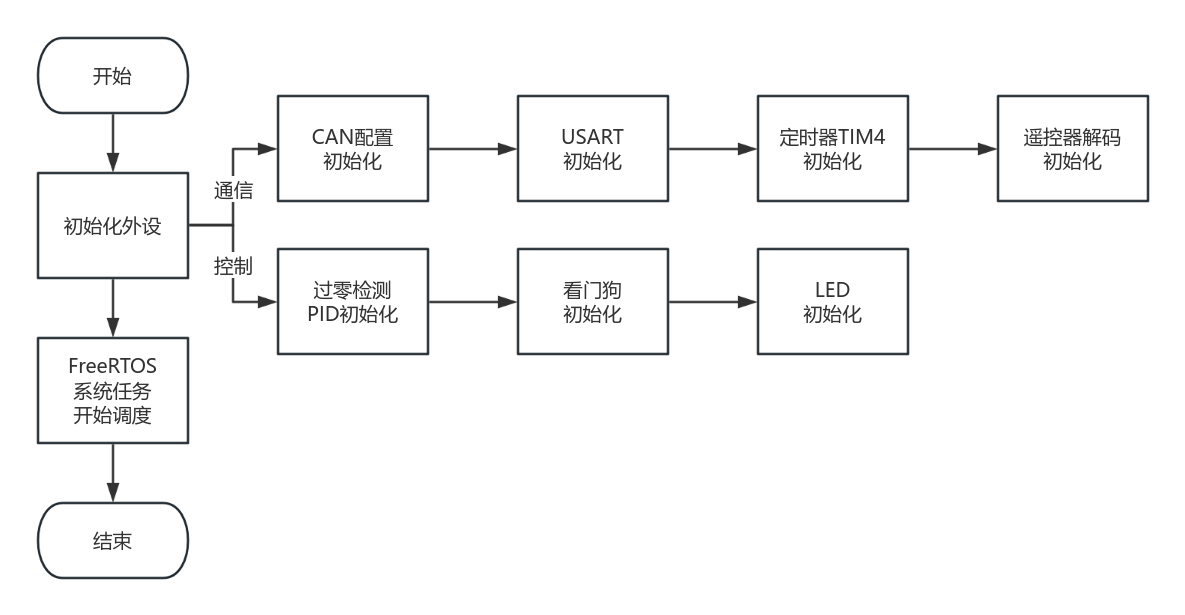
整体架构如图2-2所示。



**图2-2 整体系统架构**

2.1.4 控制框架

首先进行各个硬件模块的初始化，如CAN通信、USART通信、编码器采集定时器配置与看门狗初始化配置等，之后我们将所要实现的功能划分成不同的任务包括了对云台、底盘、摩擦轮射击和遥控器选择模式的控制任务和掉线检测任务，各个任务之间通过freertos优先级调度的机制循环运行。同时，借助CAN通信、串口通信等外部中断获取到实时有效的数据，各个任务将获取到的数据更新于控制算法中，并最后输出给到电机等执行机构，实现整个系统的闭环控制。另外加入滤波算法来保证系统读取信号的稳定性和正确性，减少噪声干扰。整体控制流程如图2-3。



**图2-3 整体控制流程**

2.1.5 功能概述

1. **云台多模式控制功能** 云台可执行初始化、跟随平台、正常、自瞄及哨兵模式，融合编码器与陀螺仪数据，通过 PID 控制实现精准角度调整。自瞄模式下，云台依据视觉数据自动锁定并跟踪目标，确保快速精准打击；哨兵模式下，云台根据视觉信息动态调整姿态，保障目标跟踪稳定性。
2. **平台巡航功能** 平台专注于 YAW 轴控制，涵盖雷达与巡航任务，具备初始化、跟随云台、雷达控制及手动模式。平台融合传感器数据，利用 PID 控制实现精确运动。雷达模式下，平台依据雷达与导航指令调整角度，达成雷达任务目标；巡航模式下，平台自动调节 YAW 轴角速度，确保稳定巡航。
3. **底盘控制模式切换和超级电容模块** 提供基于键盘按键的底盘模式切换功能，可以实 现机器人对角模式前进、基础的底盘跟随前进、小陀螺运动前进等功能，保证了操作手在机器 人遇到不同场景可以进行有效应对。另外，加入了超级电容模块，能够在短时间内提供非常大 的功率输出并且不会违反裁判系统功率限制的规则，极大提高了机器人的机动性

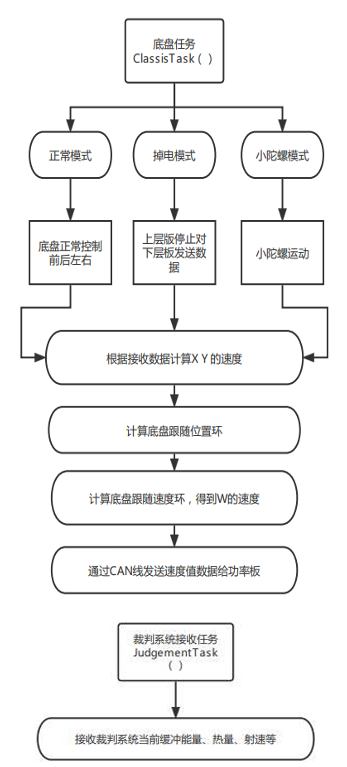
其余功能见表2-2。

**表2-2 其余功能**

|  |  |  |
| --- | --- | --- |
| 参数 | 描述 | 频率/HZ |
| **Start\_Task()** | 程序启动任务，用于区分云台，平台和底盘控制板并执行对应任务初始化函数 | 单次 |
| **Chassis\_Task()** | 战车底盘任务，包括全向轮基本运动和速度切换 | 500 |
| **Gimbal\_Task()** | 云台控制任务，包括 yaw，pitch和云台抬升的控制 | 1000 |
| **Platform\_Task()** | 平台控制任务，包括 yaw和雷达巡航的控制 | 1000 |
| **INS\_Task()** | 姿态解算任务，通过板载陀螺仪获取欧拉角，经过处理后反馈到各板任务中，以便实现底盘跟随等模式控制 | 1000 |
| **Shoot\_Task()** | 云台发射任务，包括拨弹轮及摩擦轮的控制及热量检测 | 1000 |
| **Ui\_Task()** | UI绘制任务，包括与服务器通信及UI图形打印 | 1000 |
| **Comm\_Task()** | 通讯任务，用于桥接云台，平台及底盘三板，同时承担与视觉及各模块如电机等的数据交互 | 1000 |
| **Console\_Task()** | 控制台任务，获取各应用层数据句柄如遥控器以对整车进行总控，协调云台与底盘控制 | 500 |

2.1.6 底盘任务

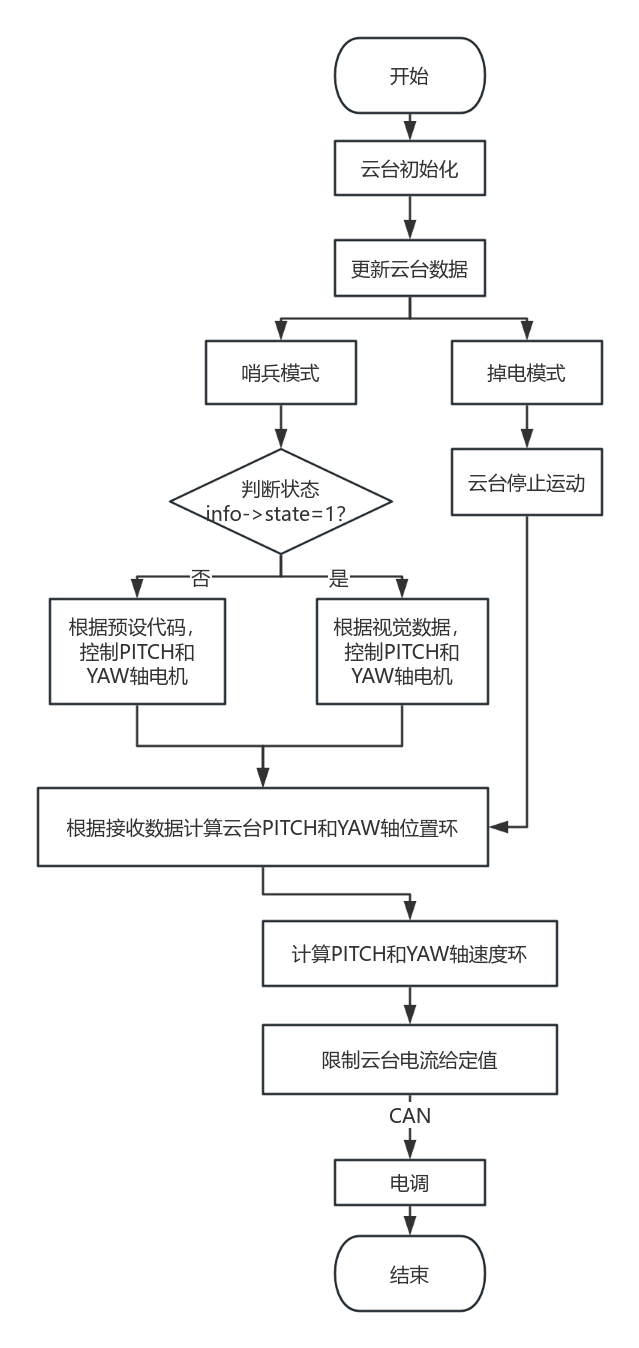
在 Chassis\_Task 中, 底盘控制，主要模式为底盘无力或跟随云台以及小陀螺模式，控制底 盘速度方向，以 及功率限制，主要流程如下。首先进行底盘初始化，主要为设置初值和 PID 参数初始化，然后进入主循环。循环中，首先更新底盘数据，包括底盘电机转速、YAW 轴云 台电机编码器角度(用于底盘跟随云台)。然后根据不同的控制模式进行处理。如果底盘为无力 状态，则直接设置底盘电机电流值为 0。否则，根据键盘或遥控拨杆的控制，计算机器人XY 方向期望速度。否则，根据 YAW 轴云台电机编码器角度 进行 PID 计算期望的底盘旋转速度。 之后根据 XY 方向速度和底盘旋转速度计算四个底盘电 机分别的期望转速。根据期望转速进 行速度环 PID 计算，获得期望的底盘电机电流值。之后， 根据超级电容状态和裁判系统返回功率，计算底盘实际的期望电流。



**图2-4 底盘控制流程**

2.1.7 云台任务

全向轮哨兵战车的云台控制任务包括使用遥控对云台角速度、角度控制，以及开启自瞄时的角度控制，云台控制流程如图2-5。首先进行云台初始化，主要为设置初值和PID参数初始化，然后进入主循环。循环中，首先更新云台数据，包括角度、角速度。然后根据不同的控制模式进行处理。如果鼠标或遥控拨杆有控制，则根据这个控制信号设置角速度，不进行角度环计算。如果没有控制数据，则保持上一刻的角度，根据这个角度进行角度环PID计算。角度环PID计算得出期望角速度。之后，根据以上几种控制模式下的期望角速度，进行速度环PID计算，获得期望的电机电流。



**图2-5 云台控制流程**

2.1.8 平台任务

全向轮哨兵战车的平台控制任务负责平台 YAW 轴的精确控制，涵盖雷达与巡航功能。任务启动时先初始化，设置初始值和 PID 参数。主循环中，定期更新编码器和陀螺仪等传感器数据，依据控制指令切换模式，执行对应控制逻辑：初始化模式归零 YAW 轴，雷达控制模式依雷达数据调整角度，巡航模式自动调节角速度，手动模式则按遥控或鼠标信号调整。控制过程融合传感器数据，利用 PID 控制器闭环控制，确保精准运动。任务结束时，平台进入放松模式，停止电机并清空 PID 积分项。平台任务具备多模式支持、传感器融合、PID 控制和任务解耦等特点，可灵活应对不同任务需求，与云台任务独立又协同。

图示

AI 生成的内容可能不正确。

**图2-6 射击控制流程**

2.1.9 射击任务

在Shoot\_Task 中，进行摩擦轮转速控制，以及拨弹盘的控制，主要流程如下。其中，根据 裁判系统信息，获得弹丸速度，设置摩擦轮转速。拨弹盘在Task开始时，进行PID的初始化， 根据操作手操作，如果为单发模式，则首先进行位置PID的计算；如果为连发模式，则直接设 置摩擦轮转速。之后进行拨弹盘速度环PID，获得期望的电机电流。

图示

AI 生成的内容可能不正确。

**图2-6 射击控制流程**

2.2 底盘部分

2.2.1底盘模式判断

void ChassisTask(void const\*argument)

{

    for(;;)

    {

        ChassisSensorUpdata();

        ChassisCtrlModeSwitch();

              ChassisSuperCapTest();   //超电检测

        switch (chassis\_handle.ctrl\_mode)//底盘模式切换判断

        {

            case CHASSIS\_STOP:

            {

                ChassisStopMode();//停止模式（有力）

            }break;

            case CHASSIS\_FOLLOW\_GIMBAL:

            {

                ChassisFollowGimbalMode();//底盘跟随模式

            }break;

            case CHASSIS\_SEPARATE\_GIMBAL:

            {

                ChassisSeparateGimbalMode();//云底分离模式

            }break;

            case CHASSIS\_SPIN:

            {

                ChassisSpinMode();//自旋模式（变速）

            }break;

            case CHASSIS\_SUPSPIN:

            {

                ChassisSupSpinMode();//超电自旋模式

            }break;

            case CHASSIS\_HALF:

            {

                ChassisHalfMode();//90 degrees turn mode

            }break;

            case CHASSIS\_HALF\_SPIN:

            {

                ChassisHalfSPINMode();

            }break;

            default:

                break;、

        }

2.2.2全向轮速度解算

void Chassis\_MoveTransform(ChassisHandle\_t\* chassis\_handle, fp32\* chassis\_vx, fp32\* chassis\_vy)

{

    static fp32 sin\_yaw = 0.0f, cos\_yaw = 0.0f;

sin\_yaw = arm\_sin\_f32(chassis\_handle->gimbal\_yaw\_ecd\_angle / RADIAN\_COEF);

cos\_yaw = arm\_cos\_f32(chassis\_handle->gimbal\_yaw\_ecd\_angle / RADIAN\_COEF);

\*chassis\_vx = cos\_yaw \* chassis\_handle->vx + sin\_yaw \* chassis\_handle->vy;

\*chassis\_vy =-sin\_yaw \* chassis\_handle->vx + cos\_yaw \* chassis\_handle->vy;

}

//全向轮速度解算

void Mecanum\_Calculate(ChassisHandle\_t\* chassis\_handle, fp32 chassis\_vx, fp32 chassis\_vy, fp32 chassis\_vw)

{

    static float rotate\_ratio\_fr;

    static float rotate\_ratio\_fl;

    static float rotate\_ratio\_bl;

    static float rotate\_ratio\_br;

    static float wheel\_rpm\_ratio;

    rotate\_ratio\_fl = ((chassis\_handle->structure.wheelbase + chassis\_handle->structure.wheeltrack)/2.0f \

            - chassis\_handle->structure.rotate\_x\_offset - chassis\_handle->structure.rotate\_y\_offset)/RADIAN\_COEF;

    rotate\_ratio\_fr = ((chassis\_handle->structure.wheelbase + chassis\_handle->structure.wheeltrack)/2.0f \

            - chassis\_handle->structure.rotate\_x\_offset + chassis\_handle->structure.rotate\_y\_offset)/RADIAN\_COEF;

    rotate\_ratio\_bl = ((chassis\_handle->structure.wheelbase + chassis\_handle->structure.wheeltrack)/2.0f \

            + chassis\_handle->structure.rotate\_x\_offset - chassis\_handle->structure.rotate\_y\_offset)/RADIAN\_COEF;

    rotate\_ratio\_br = ((chassis\_handle->structure.wheelbase + chassis\_handle->structure.wheeltrack)/2.0f \

            + chassis\_handle->structure.rotate\_x\_offset + chassis\_handle->structure.rotate\_y\_offset)/RADIAN\_COEF;

    wheel\_rpm\_ratio = 60.0f/(chassis\_handle->structure.wheel\_perimeter \* M3508S\_REDUCTION\_RATIO);

    VAL\_LIMIT(chassis\_vx, -MAX\_CHASSIS\_VX\_SPEED, MAX\_CHASSIS\_VX\_SPEED);  //mm/s

    VAL\_LIMIT(chassis\_vy, -MAX\_CHASSIS\_VY\_SPEED, MAX\_CHASSIS\_VY\_SPEED);  //mm/s

    VAL\_LIMIT(chassis\_vw, -MAX\_CHASSIS\_VW\_SPEED, MAX\_CHASSIS\_VW\_SPEED);  //deg/s

    fp32 wheel\_rpm[4];

    fp32 max = 0;

    wheel\_rpm[0] = ( chassis\_vx + chassis\_vy - chassis\_vw \* rotate\_ratio\_fl) \* wheel\_rpm\_ratio;//分解到各轮电机转速

    wheel\_rpm[1] = (-chassis\_vx + chassis\_vy - chassis\_vw \* rotate\_ratio\_fr) \* wheel\_rpm\_ratio;//分解到各轮电机转速

    wheel\_rpm[2] = ( chassis\_vx - chassis\_vy - chassis\_vw \* rotate\_ratio\_bl) \* wheel\_rpm\_ratio;//分解到各轮电机转速

    wheel\_rpm[3] = (-chassis\_vx - chassis\_vy - chassis\_vw \* rotate\_ratio\_br) \* wheel\_rpm\_ratio;//分解到各轮电机转速

    //find max item

    for (uint8\_t i = 0; i < 4; i++)

    {

        if (fabs(wheel\_rpm[i]) > max)

        {

            max = fabs(wheel\_rpm[i]);

        }

    }

    //equal proportion

    if (max > MAX\_WHEEL\_RPM)

    {

        float rate = MAX\_WHEEL\_RPM / max;

        for (uint8\_t i = 0; i < 4; i++)

        {

            wheel\_rpm[i] \*= rate;

        }

    }

    memcpy(chassis\_handle->wheel\_rpm, wheel\_rpm, 4 \* sizeof(fp32));

}

2.2.3底盘电机通讯控制

Chassis\_ControlCalc(&chassis\_handle);          //底盘解算

        for (uint8\_t i = 0; i < 4; i++)

        {

            chassis\_handle.chassis\_motor[i].given\_speed = chassis\_handle.wheel\_rpm[i];

chassis\_handle.chassis\_motor[i].current\_set = pid\_calc(&chassis\_handle.chassis\_motor[i].pid,

chassis\_handle.chassis\_motor[i].motor\_info->speed\_rpm,

chassis\_handle.chassis\_motor[i].given\_speed);//PID计算电机控制转速大小

        }

// Chassis\_LimitCap(&chassis\_handle);                 //20超电电压限制

Chassis\_LimitPower(&chassis\_handle);           //底盘功率限制

ChassisMotorSendCurrent(chassis\_handle.chassis\_motor[0].current\_set, chassis\_handle.chassis\_motor[1].current\_set,

chassis\_handle.chassis\_motor[2].current\_set, chassis\_handle.chassis\_motor[3].current\_set);//电机控制信息发送（CAN线）

        osDelay(CHASSIS\_TASK\_PERIOD);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Function: Motor\_SendMessage

 \* Description: 电机控制数据发送

 \* Input: obj CAN对象指针

 \*        std\_id CAN发送标识符

 \*        cur1 电机1电流值

 \*        cur2 电机2电流值

 \*        cur3 电机3电流值

 \*        cur4 电机4电流值

 \* Return: 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Motor\_SendMessage(CAN\_Object\_t\* obj, uint32\_t std\_id, int16\_t cur1, int16\_t cur2, int16\_t cur3, int16\_t cur4)

{

    uint8\_t TxData[8] = {0};

    TxData[0] = (uint8\_t)(cur1 >> 8);

    TxData[1] = (uint8\_t)cur1;

    TxData[2] = (uint8\_t)(cur2 >> 8);

    TxData[3] = (uint8\_t)cur2;

    TxData[4] = (uint8\_t)(cur3 >> 8);

    TxData[5] = (uint8\_t)cur3;

    TxData[6] = (uint8\_t)(cur4 >> 8);

    TxData[7] = (uint8\_t)cur4;

    BSP\_CAN\_TransmitData(obj, std\_id, TxData, 8);

}

2.2.4底盘硬件离线检测初始化

/\*----------------event-------------|-------enable-------|--offline time-|-beep\_times-\*/

**OfflineHandle\_Init(OFFLINE\_CHASSIS\_MOTOR1,  OFFLINE\_ERROR\_LEVEL,       100,         1);**

**OfflineHandle\_Init(OFFLINE\_CHASSIS\_MOTOR2,  OFFLINE\_ERROR\_LEVEL,       100,         2);**

**OfflineHandle\_Init(OFFLINE\_CHASSIS\_MOTOR3,  OFFLINE\_ERROR\_LEVEL,       100,         3);**

**OfflineHandle\_Init(OFFLINE\_CHASSIS\_MOTOR4,  OFFLINE\_ERROR\_LEVEL,       100,         4);**

**OfflineHandle\_Init(OFFLINE\_REFEREE\_SYSTEM,  OFFLINE\_WARNING\_LEVEL,     100,         0);//1**

**OfflineHandle\_Init(OFFLINE\_GIMBAL\_INFO,     OFFLINE\_WARNING\_LEVEL,     100,         3);**

**OfflineHandle\_Init(OFFLINE\_DBUS,            OFFLINE\_WARNING\_LEVEL,     100,         0);//2**

2.2.5底盘通信线程初始化

SoftwareTimerRegister(Transmit\_RefereeData, (void\*)NULL, 5);      //20=1s

Comm\_TransmitInit(&client\_ui\_tx\_handle, client\_ui\_tx\_fifo\_buffer, REFEREE\_SYSTEM\_FIFO\_SIZE, ClientUI\_UploadDataHook);

    ClientUI\_Init(&client\_ui\_tx\_handle);

   /\*UI界面设置\*/

    SoftwareTimerRegister(ClientUI\_Data, (void\*)NULL, 100);

    /\*通信回调设置\*/

    BSP\_UART\_SetRxCallback(&dbus\_obj, DBUS\_ReceiveCallback);

    BSP\_UART\_SetRxCallback(&com1\_obj, COM1\_ReceiveCallback);

    BSP\_UART\_SetRxCallback(&com2\_obj, COM2\_ReceiveCallback);

    BSP\_CAN\_SetRxCallback(&can1\_obj, CAN1\_ReceiveCallback);

    BSP\_CAN\_SetRxCallback(&can2\_obj, CAN2\_ReceiveCallback);

2.3 云台部分

2.3.1 云台模式判断

for(;;)

    {

        GimbalSensorUpdata(); // 更新云台传感器数据

        VisionDataUpdate();     // 处理视觉数据

        GimbalCtrlModeSwitch(); // 切换云台控制模式

        switch (gimbal\_handle.ctrl\_mode)

        {

            case GIMBAL\_INIT:

            {

                GimbalInitMode(); // 执行初始化模式逻辑

            }break;

            case GIMBAL\_FOLLOW\_PLATFORM:

            {

                GimbalFollowPlatformMode(); // 执行跟随平台模式逻辑

            }break;

            case GIMBAL\_NORMAL:

            {

                GimbalNormalMode(); // 执行正常模式逻辑(底盘跟随)

            }break;

            case GIMBAL\_VISION\_AIM:

            {

                GimbalVisionAimMode(); // 执行视觉瞄准模式逻辑

            }break;

            case GIMBAL\_SENTRY:

            {

                GimbalSentryMode(); // 执行哨兵模式逻辑

            }break;

            default:

                break;

        }

2.3.2 云台控制算法

void GimbalMotorControl(M6020\_Motor\_t\* motor)

{

    GimbalMotorChangeProtect(motor); // 保护电机模式切换

    if (motor->mode == RAW\_VALUE\_MODE)

    {

        motor->current\_set = motor->given\_value; // 原始模式下直接设置当前值

        dpid\_clear(&motor->dpid); // 清除双环PID

    }

else if(motor->mode == GYRO\_MODE &&

(console.gimbal\_cmd == GIMBAL\_VISION\_AIM\_CMD ||

console.gimbal\_cmd == GIMBAL\_SENTRY\_CMD))

    { // 陀螺仪模式下，特定命令使用视觉ESO双环PID控制

        motor->current\_set = Vision\_ESO\_DoublePID\_Calc(&motor->dpid,

                                             motor->given\_value,

                                             motor->sensor.gyro\_angle,

                                             motor->sensor.palstance,

                                             &ESO\_6020);

    }

    else if(motor->mode == GYRO\_MODE)

    {// 陀螺仪模式下使用标准双环PID控制

        motor->current\_set = DoublePID\_Calc(&motor->dpid,

                                         motor->given\_value,

                                         motor->sensor.gyro\_angle,

                                         motor->sensor.palstance);

    }

    else if(motor->mode == ENCONDE\_MODE)

    {// 编码器模式下使用标准双环PID控制

        motor->current\_set = DoublePID\_Calc(&motor->dpid,

                                         motor->given\_value,

                                         motor->sensor.relative\_angle,

                                         motor->sensor.palstance);

    }

}

2.3.3 云台电机通讯控制

GimbalMotorSendCurrent((int16\_t)YAW\_MOTO\_POSITIVE\_DIR \* gimbal\_handle.yaw\_motor.current\_set,

(int16\_t)PITCH\_MOTO\_POSITIVE\_DIR \* gimbal\_handle.pitch\_motor.current\_set);// 电流控制

osDelay(GIMBAL\_TASK\_PERIOD); /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Function: Motor\_SendMessage

 \* Description: 电机控制数据发送

 \* Input: obj CAN对象指针

 \*        std\_id CAN发送标识符

 \*        cur1 电机1电流值

 \*        cur2 电机2电流值

 \*        cur3 电机3电流值

 \*        cur4 电机4电流值

 \* Return: 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Motor\_SendMessage(CAN\_Object\_t\* obj, uint32\_t std\_id, int16\_t cur1, int16\_t cur2, int16\_t cur3, int16\_t cur4)

{

    uint8\_t TxData[8] = {0};

    TxData[0] = (uint8\_t)(cur1 >> 8);

    TxData[1] = (uint8\_t)cur1;

    TxData[2] = (uint8\_t)(cur2 >> 8);

    TxData[3] = (uint8\_t)cur2;

    TxData[4] = (uint8\_t)(cur3 >> 8);

    TxData[5] = (uint8\_t)cur3;

    TxData[6] = (uint8\_t)(cur4 >> 8);

    TxData[7] = (uint8\_t)cur4;

    BSP\_CAN\_TransmitData(obj, std\_id, TxData, 8);

}

2.3.4 自瞄模式

2.3.4.1 自瞄数据获取

static void GimbalVisionAimMode(void)

{ //使能云台yaw轴陀螺仪模式

gimbal\_handle.yaw\_motor.mode = GYRO\_MODE;

//使能云台pitch轴编码器模式

gimbal\_handle.pitch\_motor.mode = ENCONDE\_MODE;

Aim\_contorl(&yaw\_aim);

pid\_paramSA(&yaw\_aim);//自适应PID

VisionDatabase\_t\* info = VisionData\_Pointer();//更新视觉数据

info->yaw -= ESO\_6020.z1;

gimbal\_handle.pitch\_motor.given\_value +=

gimbal\_handle.console->gimbal.pitch\_v

+ vision\_comps\_pitch

\* pid\_calc(&pitch\_vision\_pid,

gimbal\_handle.pitch\_motor.given\_value,

gimbal\_handle.pitch\_motor.given\_value+info->pitch);

gimbal\_handle.yaw\_motor.given\_value +=

gimbal\_handle.console->gimbal.yaw\_v

+pid\_calc(&yaw\_vision\_pid,

gimbal\_handle.yaw\_motor.given\_value,

gimbal\_handle.yaw\_motor.given\_value+info->yaw);

VAL\_LIMIT(gimbal\_handle.pitch\_motor.given\_value, gimbal\_handle.pitch\_motor.min\_relative\_angle, gimbal\_handle.pitch\_motor.max\_relative\_angle);//pitch角度限位

gimbal\_handle.yaw\_motor.given\_value = AngleTransform(gimbal\_handle.yaw\_motor.given\_value, gimbal\_handle.yaw\_motor.sensor.gyro\_angle);

}

2.3.4.2 视觉数据结构体

typedef struct

{

    float    pitch;                  // 当前俯仰角度

    float    yaw;                    // 当前偏航角度

    float    palstance;              // 当前角速度

    float    accelerated\_speed;      // 当前加速度

    float    last\_pitch;             // 上一次的俯仰角度

    float    last\_yaw;               // 上一次的偏航角度

    float    last\_yaw\_form\_center;   // 上一次偏离中心的偏航角度

    float    last\_palstance;         // 上一次的角速度

    float    last\_accelerated\_speed; // 上一次的加速度

    float    rpm;                    // 转速

    int      can\_shoot;              // 是否可以射击标志

    int      move\_state;             // 当前移动状态

    int      last\_move\_state;        // 上一次的移动状态

    int      state;                // 当前状态 (VisionState\_e, 0x01->Comm\_Successed)

    int      last\_state;             // 上一次的状态

    VisionAim\_e yaw\_success;         // 偏航角是否瞄准成功

    VisionAim\_e pitch\_success;       // 俯仰角是否瞄准成功

} VisionDatabase\_t;

2.3.4.3 视觉数据更新

static void VisionDataUpdate(void)

{

    Comm\_VisionInfo\_t\* info = VisionInfo\_Pointer();

if(CheckDeviceIsOffline(OFFLINE\_VISION\_INFO))

{vision\_data.pitch = vision\_data.yaw  = 0;return;} // 检查视觉设备是否离线

    vision\_data.last\_state = vision\_data.state; // 保存上一次的状态

    vision\_data.state = TwoBytesToInt(info->state); // 更新当前状态

    vision\_enemy\_data.enemy\_kind = info->enemy\_kind; // 更新敌人种类

    vision\_enemy\_data.enemy\_x = TwoBytesToInt(info->enemy\_x); // 更新敌人X坐标

    vision\_enemy\_data.enemy\_y = TwoBytesToInt(info->enemy\_y); // 更新敌人Y坐标

    vision\_data.can\_shoot = TwoBytesToInt(info->can\_shoot); // 更新是否可以射击

    vision\_data.move\_state = TwoBytesToInt(info->move\_state); // 更新移动状态

    vision\_data.last\_move\_state = vision\_data.move\_state; // 保存上一次的移动状态

    vision\_data.last\_pitch = vision\_data.pitch; // 保存上一次的俯仰角

    ESO\_6020.y = vision\_data.yaw - vision\_data.last\_yaw; // 计算ESO的输入

    vision\_data.last\_yaw = vision\_data.yaw; // 保存上一次的偏航角

    vision\_data.last\_palstance = vision\_data.palstance; // 保存上一次的角速度

    vision\_data.accelerated\_speed = (vision\_data.palstance - vision\_data.last\_palstance) / diff\_sof\_time; // 计算加速度

    vision\_data.pitch = FourBytesToFloat(info->pitch); // 更新俯仰角

    vision\_data.yaw = FourBytesToFloat(info->yaw); // 更新偏航角

    vision\_data.palstance = FourBytesToFloat(info->palstance) \* 180 / PI; // 更新角速度并转换为角度

    yaw\_aim.systeam\_time = BSP\_GetTime\_ms(); // 获取系统运行时间

    ESO\_calc(&ESO\_6020,gimbal\_handle.yaw\_motor.current\_set); // 计算ESO

}

2.3.5 云台硬件离线检测

/\*--------------------event-----------------|-----------enable-----------|-offline time-|-beep\_times-\*/

OfflineHandle\_Init(OFFLINE\_VISION\_INFO, OFFLINE\_WARNING\_LEVEL, 200, 0);

OfflineHandle\_Init(OFFLINE\_GIMBAL\_PITCH, OFFLINE\_ERROR\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_GIMBAL\_YAW, OFFLINE\_ERROR\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR1, OFFLINE\_ERROR\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR2, OFFLINE\_ERROR\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_TRIGGER\_MOTOR, OFFLINE\_ERROR\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_REFEREE\_SYSTEM, OFFLINE\_WARNING\_LEVEL, 100, 0);

// OfflineHandle\_Init(OFFLINE\_PLATFORM\_INFO, OFFLINE\_ERROR\_LEVEL, 100, 2);

// OfflineHandle\_Init(OFFLINE\_WBUS, OFFLINE\_WARNING\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_DBUS, OFFLINE\_ERROR\_LEVEL, 100, 1);

2.3.6 云台通信线程初始化

Comm\_TransmitInit(&vision\_tx\_handle, vision\_tx\_fifo\_buffer, COMPUTER\_DATA\_FIFO\_SIZE, COM1\_UploadDataHook);

SoftwareTimerRegister(Vision\_RobotInfoUploadCallback, (void\*)NULL, 5);

Comm\_ReceiveInit(&vision\_rx\_handle, COMPUTER\_PROTOCOL\_HEADER\_SOF, vision\_rx\_fifo\_buffer, COMPUTER\_DATA\_FIFO\_SIZE, ComputerProtocol\_ParseHandler);

Comm\_ReceiveInit(&referee\_rx\_handle, REFEREE\_SYSTEM\_HEADER\_SOF, referee\_rx\_fifo\_buffer, REFEREE\_SYSTEM\_FIFO\_SIZE, RefereeSystem\_ParseHandler);

Comm\_TransmitInit(&gimbal\_tx\_handle, gimbal\_tx\_fifo\_buffer, GIMBAL\_CHASSIS\_DATA\_FIFO\_SIZE, CAN1\_UploadDataHook);

SoftwareTimerRegister(GimbalInfoUploadCallback, (void\*)NULL, 10);

Comm\_ReceiveInit(&gimbal\_rx\_handle, USER\_PROTOCOL\_HEADER\_SOF, gimbal\_rx\_fifo\_buffer, GIMBAL\_CHASSIS\_DATA\_FIFO\_SIZE, UserProtocol\_ParseHandler);

BSP\_UART\_SetRxCallback(&dbus\_obj, DBUS\_ReceiveCallback);

BSP\_UART\_SetRxCallback(&com1\_obj, COM1\_ReceiveCallback);

BSP\_UART\_SetRxCallback(&com2\_obj, COM2\_ReceiveCallback);

BSP\_CAN\_SetRxCallback(&can1\_obj, CAN1\_ReceiveCallback);

BSP\_CAN\_SetRxCallback(&can2\_obj, CAN2\_ReceiveCallback);

2.3.7 视觉交互数据回调函数

static int32\_t Vision\_RobotInfoUploadCallback(void \*argc)

{

Comm\_RobotInfo\_t\* info = RobotInfo\_Pointer();

uint16\_t robot\_id = ChassisInfo\_Pointer()->robot\_state\_t.robot\_id;

info->data\_head = 0xAA;

if (robot\_id > 100) //ID大于100是蓝方 应该打红方；

{

info->enemy\_color = Red;

}

else if (robot\_id > 1)

{

info->enemy\_color = Blue;

}

else

{

info->enemy\_color = AllColor;

}

info->yaw\_relative\_angle = (gimbal\_handle.imu->attitude.yaw - attitude\_yaw\_initial) \* ANGLE\_TO\_RAD;

info->pitch\_relative\_angle = (gimbal\_handle.imu->attitude.pitch - gimbal\_handle.imu->i\_attitude.i\_pitch) \* ANGLE\_TO\_RAD;

info->robot\_level = ChassisInfo\_Pointer()->robot\_state\_t.robot\_level;

Comm\_TransmitData\_Computer(&vision\_tx\_handle, (uint8\_t\*)info, sizeof(Comm\_RobotInfo\_t));

return 0;

}

2.4 平台部分

2.4.1 平台模式判断

    for(;;)

    {

        PlatformSensorUpdata(); // 更新平台传感器数据

        PlatformCtrlModeSwitch(); // 切换平台控制模式

        switch (platform\_handle.ctrl\_mode)

        {

            case PLATFORM\_INIT:

            {

                PlatformInitMode(); // 执行初始化模式逻辑

            }break;

            case PLATFORM\_FOLLOW\_GIMBAL:

            {

                PlatformFollowGimbalMode(); // 执行跟随云台模式逻辑

            }break;

            case COMPUTER\_RADAR\_CONTROL:

            {

                PlatformComputerRadarCtrlMode(); // 执行计算机雷达控制模式逻辑

            }break;

            case HUMAN\_CONTROL:

            {

                PlatformHumanCtrlMode(); // 执行人工控制模式逻辑

            }break;

            default:

                break;

        }

2.4.2 平台控制算法

void PlatformMotorControl(Motor\_9025\_t\* motor)

{

        if (motor->last\_mode != motor->mode) // 检查电机模式是否发生变化

    {

        if(motor->mode == RAW\_VALUE\_MODE) // 原始模式

        {

            motor->given\_value = motor->powerControl; // 设置给定值为功率控制值

        }

        else if (motor->mode == GYRO\_MODE) // 陀螺仪模式

        {

            motor->given\_value = motor->sensor.gyro\_angle; // 设置给定值为陀螺仪角度

        }

        else if (motor->mode == ENCONDE\_MODE) // 编码器模式

        {

            motor->given\_value = motor->sensor.relative\_angle; // 设置给定值为相对角度

        }

    }

    motor->last\_mode = motor->mode; // 更新最后的模式

        if(motor->mode == GYRO\_MODE) // 如果是陀螺仪模式

        {// 使用双环PID计算功率控制值

                motor->powerControl = DoublePID\_Calc(&motor->dpid,

                                             motor->given\_value,

                                             motor->sensor.gyro\_angle,

                                             motor->sensor.palstance);

        }

        else if(motor->mode == ENCONDE\_MODE) // 如果是编码器模式

        {// 使用双环PID计算功率控制值

                motor->powerControl = DoublePID\_Calc(&motor->dpid,

                                             motor->given\_value,

                                             motor->sensor.relative\_angle,

                                             motor->sensor.palstance);

        }

}

if (platform\_handle.ctrl\_mode == PLATFORM\_RELAX)

        {

            dpid\_clear(&platform\_handle.yaw\_motor.dpid);

            platform\_handle.yaw\_motor.powerControl = 0;

        }

2.4.3 平台电机通讯控制

        Torque\_Control(platform\_handle.platform\_can ,0x141, platform\_handle.yaw\_motor.powerControl);

        osDelay(PLATFORM\_TASK\_PERIOD);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 \* Function: Torque\_Control

 \* Description: 电机开环控制数据发送

 \* Input: obj CAN对象指针

 \*        std\_id CAN发送标识符

 \*        powerControl 电机电流值，数值范围-850~850

 \* Return: 无

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void Torque\_Control(CAN\_Object\_t\* obj, uint32\_t std\_id, int16\_t powerControl)

{

    uint8\_t TxData[8] = {0};

    TxData[0] = 0xA0;

    TxData[1] = 0x00;

    TxData[2] = 0x00;

    TxData[3] = 0x00;

    TxData[4] = (uint8\_t)powerControl;

    TxData[5] = (uint8\_t)(powerControl >> 8);

    TxData[6] = 0x00;

    TxData[7] = 0x00;

    BSP\_CAN\_TransmitData(obj, std\_id, TxData, 8);

}

2.4.4 雷达巡航模式

static void PlatformComputerRadarCtrlMode(void)

{

    ext\_robot\_hurt\_t\* robot\_hurt\_t = RefereeSystem\_HeatData\_Pointer();

    Comm\_NavCmd\_t\* NavCmd\_info = NavCmd\_Pointer();

    uint8\_t armor\_id = robot\_hurt\_t->armor\_id;

    int detect\_id = 0;

    platform\_handle.yaw\_motor.mode = GYRO\_MODE;

    Comm\_GimbalInfo\_t\* info = GimbalInfo\_Pointer();

    fp32 yaw\_target = 0;

    // 根据 under\_attack 和 vision\_state 计算 yaw\_target

    if(platform\_handle.console->vision\_delay == 1 && platform\_handle.console->radar\_state == 0)

    {

        target\_angle = platform\_handle.yaw\_motor.given\_value;

    }

    else if(NavCmd\_info->detect\_id == 4)

    {

        fp32 info\_yaw = NavCmd\_info->vw \* ANGLE\_TO\_RAD;

        platform\_handle.yaw\_motor.given\_value += pid\_calc(&platform\_handle.move\_pid, platform\_handle.yaw\_motor.given\_value, platform\_handle.yaw\_motor.given\_value+info\_yaw);

    }

    else if (platform\_handle.console->under\_attack == 0 || platform\_handle.console->radar\_state == 1)

    {

        // 未受攻击且无视觉输入

            platform\_handle.yaw\_motor.given\_value += NavCmd\_info->vw \* Platform\_cruise\_RATIO;

}

        if(platform\_handle.console->under\_attack == 0)

        {

            platform\_handle.yaw\_motor.given\_value = AngleTransform(platform\_handle.yaw\_motor.given\_value, platform\_handle.yaw\_motor.sensor.gyro\_angle);

        }

        else

        {

            platform\_handle.yaw\_motor.given\_value = AngleTransform(target\_angle, platform\_handle.yaw\_motor.sensor.gyro\_angle);

        }

}

2.4.5 平台硬件离线检测

/\*--------------------event-----------------|-----------enable-----------|-offline time-|-beep\_times-\*/

OfflineHandle\_Init(OFFLINE\_PLATFORM\_YAW, OFFLINE\_ERROR\_LEVEL, 100, 0);

// OfflineHandle\_Init(OFFLINE\_REFEREE\_SYSTEM, OFFLINE\_WARNING\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_GIMBAL\_INFO, OFFLINE\_ERROR\_LEVEL, 100, 1);

OfflineHandle\_Init(OFFLINE\_NAV\_INFO, OFFLINE\_ERROR\_LEVEL, 100, 1);

// OfflineHandle\_Init(OFFLINE\_WBUS, OFFLINE\_WARNING\_LEVEL, 100, 0);

OfflineHandle\_Init(OFFLINE\_DBUS, OFFLINE\_ERROR\_LEVEL, 100, 1);

2.4.6 平台通信线程初始化

Comm\_TransmitInit(&nav\_tx\_handle, nav\_tx\_fifo\_buffer, COMPUTER\_DATA\_FIFO\_SIZE, COM1\_UploadDataHook);

Comm\_ReceiveInit(&nav\_rx\_handle, COMPUTER\_PROTOCOL\_HEADER\_SOF, nav\_rx\_fifo\_buffer, COMPUTER\_DATA\_FIFO\_SIZE, ComputerProtocol\_ParseHandler);

SoftwareTimerRegister(Nav\_DecisionInfoUploadCallback, (void\*)NULL, 10);

Comm\_ReceiveInit(&referee\_rx\_handle, REFEREE\_SYSTEM\_HEADER\_SOF, referee\_rx\_fifo\_buffer, REFEREE\_SYSTEM\_FIFO\_SIZE, RefereeSystem\_ParseHandler);

Comm\_TransmitInit(&platform\_tx\_handle, platform\_tx\_fifo\_buffer, GIMBAL\_CHASSIS\_DATA\_FIFO\_SIZE, CAN1\_UploadDataHook);

Comm\_ReceiveInit(&platform\_rx\_handle, USER\_PROTOCOL\_HEADER\_SOF, platform\_rx\_fifo\_buffer, GIMBAL\_CHASSIS\_DATA\_FIFO\_SIZE, UserProtocol\_ParseHandler);

SoftwareTimerRegister(PlatformInfoUploadCallback, (void\*)NULL, 10);

BSP\_UART\_SetRxCallback(&dbus\_obj, DBUS\_ReceiveCallback);

BSP\_UART\_SetRxCallback(&com1\_obj, COM1\_ReceiveCallback);

BSP\_UART\_SetRxCallback(&com2\_obj, COM2\_ReceiveCallback);

BSP\_CAN\_SetRxCallback(&can1\_obj, CAN1\_ReceiveCallback);

BSP\_CAN\_SetRxCallback(&can2\_obj, CAN2\_ReceiveCallback);

2.4.7 决策信息回调函数

static int32\_t Nav\_DecisionInfoUploadCallback(void \*argc)

{// 决策信息上传回调函数

Comm\_DecisionInfo\_t\* info = DecisionInfo\_Pointer();

Comm\_ChassisInfo\_t\* chassis\_info = ChassisInfo\_Pointer();

uint16\_t robot\_id = chassis\_info->robot\_state\_t.robot\_id;

info->data\_head = 0xAA;

info->game\_state = RefereeSystem\_GameState\_Pointer()->game\_progress;

// info->enemy\_color = 1;

// info->bullet\_remaining\_num\_17mm = 2;

// info->ally\_sentry\_HP = 3;

// info->ally\_infantry\_HP = 4;

// info->ally\_hero\_HP = 5;

// info->enemy\_sentry\_HP = 6;

// info->enemy\_infantry\_HP = 7;

// info->enemy\_hero\_HP = 8;

if (robot\_id > 100) //ID大于100是蓝方 应该打红方；

{

info->enemy\_color = 1;

info->bullet\_remaining\_num\_17mm = (uint8\_t)(RefereeSystem\_GameRobot\_bullet\_remaining\_t\_Pointer()->bullet\_remaining\_num\_17mm / 10);

info->ally\_sentry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_7\_robot\_HP/10);

info->ally\_infantry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_3\_robot\_HP/10);

info->ally\_hero\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_1\_robot\_HP/10);

info->enemy\_sentry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_7\_robot\_HP/10);

info->enemy\_infantry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_3\_robot\_HP/10);

info->enemy\_hero\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_1\_robot\_HP/10);

}

else if (robot\_id > 1)

{

info->enemy\_color = 2;

info->bullet\_remaining\_num\_17mm = (uint8\_t)(RefereeSystem\_GameRobot\_bullet\_remaining\_t\_Pointer()->bullet\_remaining\_num\_17mm / 10);

info->ally\_sentry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_7\_robot\_HP/10);

info->ally\_infantry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_3\_robot\_HP/10);

info->ally\_hero\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.red\_1\_robot\_HP/10);

info->enemy\_sentry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_7\_robot\_HP/10);

info->enemy\_infantry\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_3\_robot\_HP/10);

info->enemy\_hero\_HP = (uint8\_t)(chassis\_info->robot\_HP\_t.blue\_1\_robot\_HP/10);

}

info->detect\_enemy\_kind = GimbalInfo\_Pointer()->enemy\_kind;

info->detect\_enemy\_y = GimbalInfo\_Pointer()->enemy\_y;

info->detect\_enemy\_x = GimbalInfo\_Pointer()->enemy\_x;

info->yaw\_gyro\_angle = GimbalInfo\_Pointer()->yaw\_ecd\_angle;

Comm\_TransmitData\_Computer(&nav\_tx\_handle, (uint8\_t\*)info, sizeof(Comm\_DecisionInfo\_t));

return 0;

}

2.5.射击部分

2.5.1 射击任务

void ShootTask(void const\*argument)

{

    HAL\_TIM\_PWM\_Start(&htim1, TIM\_CHANNEL\_1);

    for(;;)

    {

        ShootSensorUpdata();//数据更新

        ShootCtrlModeSwitch();//射击模式切换

        MagazineCtrlModeSwitch();//弹仓电机模式切换

        Shoot\_LaserCtrl(shoot\_handle.ctrl\_mode);//红外线控制

        Shoot\_MagazineMotorCtrl(&shoot\_handle);//弹仓电机控制

//          Shoot\_HeatingTime(&shoot\_handle);

//          Shoot\_HeatingLimit(&shoot\_handle);

        Shoot\_TriggerMotorCtrl(&shoot\_handle);//拨轮电机控制

        Shoot\_FrictionWheelMotorCtrl(shoot\_handle.ctrl\_mode, shoot\_handle.fric\_wheel\_motor);//摩擦轮电机控制

        if (shoot\_handle.ctrl\_mode == SHOOT\_RELAX)

        {

            shoot\_handle.fric\_wheel\_motor[0].current\_set = 0;

            shoot\_handle.fric\_wheel\_motor[1].current\_set = 0;

            shoot\_handle.trigger\_motor.current\_set = 0;

        }

        ShootMotorSendCurrent(shoot\_handle.fric\_wheel\_motor[0].current\_set,

                              shoot\_handle.fric\_wheel\_motor[1].current\_set,

                              TRIGGER\_MOTOR\_POSITIVE\_DIR \* shoot\_handle.trigger\_motor.current\_set,

                              0);//电机电流控制(单条CAN总线)

        \_\_HAL\_TIM\_SET\_COMPARE(&htim1, TIM\_CHANNEL\_1, shoot\_handle.magazine\_pwm);

        osDelay(SHOOT\_TASK\_PERIOD);

2.5.2 射击判断

2.5.2.1 射击判断（遥控器控制段）

if (console.shoot\_cmd == SHOOT\_STOP\_CMD)

    {

        if (wheel\_switch.switch\_state == REMOTE\_SWITCH\_CHANGE\_3TO1)//左拨轮上拨

        {

            console.shoot\_cmd = SHOOT\_START\_CMD;

        }

        if(wheel\_switch.switch\_value\_raw== REMOTE\_SWITCH\_VALUE\_DOWN&&magazine\_flag==1&&remotecontrol\_loose\_time>50)

        {

          console.magazine\_cmd = MAGAZINE\_OFF\_CMD;

          magazine\_flag=0;

          remotecontrol\_loose\_time=0;

        }

        if(wheel\_switch.switch\_value\_raw== REMOTE\_SWITCH\_VALUE\_DOWN&&magazine\_flag==0&&remotecontrol\_loose\_time>50)

        {

           console.magazine\_cmd = MAGAZINE\_ON\_CMD;

           magazine\_flag=1;

           remotecontrol\_loose\_time=0;

        }

        remotecontrol\_loose\_time++;

    }

    else if (console.shoot\_cmd == SHOOT\_START\_CMD)

    {

            if(console.magazine\_cmd== MAGAZINE\_ON\_CMD)

        {

          console.magazine\_cmd = MAGAZINE\_OFF\_CMD;

          magazine\_flag=0;

            }

            if(last\_rc.sw1 == REMOTE\_SWITCH\_VALUE\_DOWN)

            {

             console.shoot\_cmd = SHOOT\_STOP\_CMD;

            }

            if (wheel\_switch.switch\_state == REMOTE\_SWITCH\_CHANGE\_3TO1)

            {

             console.shoot\_cmd = SHOOT\_STOP\_CMD;

            }

            else if (wheel\_switch.switch\_state == REMOTE\_SWITCH\_CHANGE\_3TO2)//左拨轮下拨

            {

             console.shoot.fire\_cmd = ONE\_FIRE\_CMD;

            }

            else if (wheel\_switch.switch\_value\_raw == REMOTE\_SWITCH\_VALUE\_DOWN)

            {

            shoot\_time++;

            if(shoot\_time > 50)

                console.shoot.fire\_cmd = ONE\_FIRE\_CMD ;

            else

               console.shoot.fire\_cmd = STOP\_FIRE\_CMD;

           }

          else

           {

            console.shoot.fire\_cmd = STOP\_FIRE\_CMD;

            shoot\_time = 0;

           }

2.5.2.2 射击判断（键鼠控制段）

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*射击模式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(!last\_rc.kb.bit.V && console.rc->kb.bit.V ) //自动打弹开启

{

console.auto\_aim\_flag++;

if(console.auto\_aim\_flag==2)

{

console.auto\_aim\_flag=0;

}

}

if(console.rc->mouse.r) //开启自瞄

{

console.gimbal\_cmd = GIMBAL\_VISION\_AIM\_CMD;

console.shoot\_cmd = SHOOT\_START\_CMD;

// console.auto\_aim\_flag = 1;

}

else

{

console.gimbal\_cmd= GIMBAL\_NORMAL\_CMD;

if(vision\_flag!=1)

{

console.shoot\_cmd = SHOOT\_STOP\_CMD;

}

}

if(!last\_rc.kb.bit.Z && console.rc->kb.bit.Z)

{

if(console.heat\_limit!= Heat\_limit\_on)

{

console.heat\_limit = Heat\_limit\_on;

}

else

{

console.heat\_limit = Heat\_limit\_off;

}

}

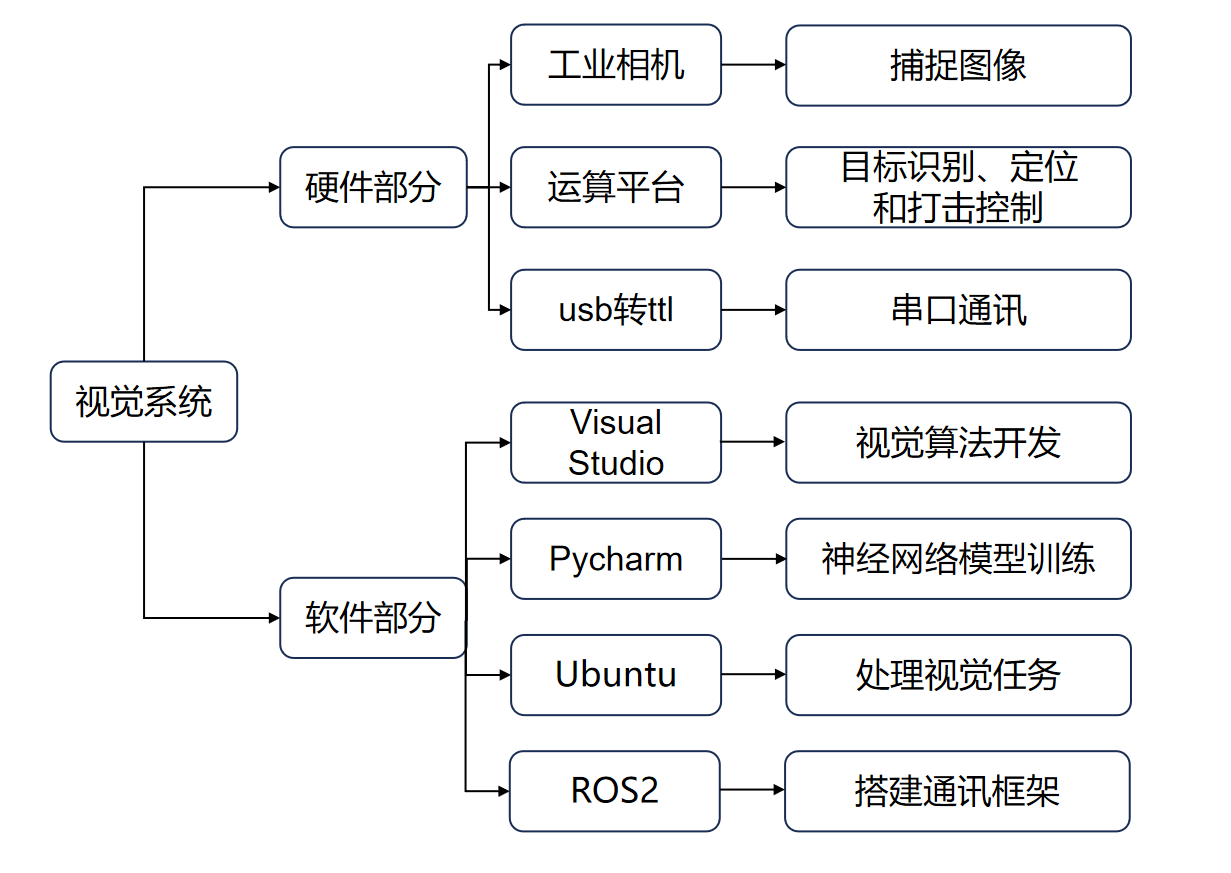
**第三部分 视觉系统设计**

3.1 背景与目标

在以智能制造为核心的工业4.0时代背景下，机器人正从“自动化”迈向“智能化”，而视觉系统作为其感知和理解环境的核心，已成为现代机器人不可或缺的“眼睛”。基于当今视觉系统在机器人中发挥的重要作用，我们通过融合机器视觉、目标预测与运动控制技术开发了这套机器人视觉系统，实现了机器人全自动目标锁定与打击功能。

3.2 视觉自动瞄准算法设计

在视觉系统设计中，机器人需精准识别并命中敌方装甲板以造成有效伤害。系统需处理两种规格装甲板：大型（230×127mm）和小型（135×125mm）。每块装甲板具有数字标签（1-5）和特殊图案用于机器人类型区分，同时边缘配备红蓝双色灯条用于敌我区分。数字标签同时用于区分不同敌方机器人，为攻击策略制定提供依据。基于上述要求，我们开发了机器人的视觉系统以实现机器人云台对目标的自动瞄准与打击。



**图3-1 机器人视觉系统设计图**

3.3 设计方案

机器人视觉系统部分主要由目标检测、目标运动解算以及火控策略三部分组成，基于此系统，机器人可以实现从目标识别、目标运动解算、云台锁敌到自动开火的全过程。

3.3.1目标检测

战车识别的目标是带有两侧发光灯条和中央数字的装甲板，根据装甲板特征，我们开发了目标检测融合算法。该算法基于计算机视觉、深度学习等技术，融合了YOLOv5检测算法与Opencv图像处理方法，实现了对目标的快速检测，表现出较强的鲁棒性与精准度。

首先，系统采用Hikvision工业相机实时捕获图像，通过YOLOv5深度学习模型快速定位候选装甲板区域，初步输出灯条的四角点坐标与类别标签。

std::vector<Armor> YoloModel::find\_armors(cv::Mat src)

{

// 神经网络推理

std::vector<bbox\_t> bbox\_ts = forward(src);

// 剔除边缘目标

bbox\_ts = screen\_out\_edge\_targets(bbox\_ts, src.cols, src.rows);

std::vector<Armor> out\_armors;

for (const auto& x : bbox\_ts) {

out\_armors.push\_back(Armor(x.pts, x.label));

};

return out\_armors;

}

std::vector<YoloModel::bbox\_t> YoloModel::forward(cv::Mat src)

{

// 设定

double image\_width = src.cols;

double image\_height = src.rows;

src = letterbox(src, image\_size, image\_size, padd\_w\_, padd\_h\_);

// double start = get\_now\_time();

auto input = iq.get\_input\_tensor(0);

input.set\_shape({ 1,3,static\_cast<unsigned long long>(src.cols),static\_cast<unsigned long long>(src.rows) });

// 转换颜色空间

cv::cvtColor(src, src, cv::COLOR\_BGR2RGB);

src.convertTo(src, CV\_32F, 1.0 / 255.0);

// 分离通道并复制数据到输出向量

std::vector<cv::Mat> channels(3);

cv::split(src, channels);

float\* input\_data\_host = input.data<float>();

int image\_area = src.rows \* src.cols;

std::copy(channels[0].begin<float>(), channels[0].end<float>(), input\_data\_host + image\_area \* 0);

std::copy(channels[1].begin<float>(), channels[1].end<float>(), input\_data\_host + image\_area \* 1);

std::copy(channels[2].begin<float>(), channels[2].end<float>(), input\_data\_host + image\_area \* 2);

iq.infer(); // 推理过程，这可能是最耗时的部分

auto output = iq.get\_output\_tensor(0);

float confidence\_threshold = 0.25;

int output\_numbox = output.get\_shape()[1]; // TOPK\_NUM = 25200

int output\_numprob = output.get\_shape()[2]; // 49

int modle\_last\_length = 13;

int num\_classes = output\_numprob - modle\_last\_length; // 36

float\* output\_buffer = output.data<float>();

int TOPK\_NUM = output\_numbox;

// 该四点模型采用 49: 四点分别是左上,左下,右下,右上

// x0 y0 x1 y1 confince ltx lty lbx lby rbx rby rtx rty ==> 0 - 12

// 13 - 48 分类,总共36种

std::vector<bbox\_t> rst;

rst.reserve(TOPK\_NUM);

std::vector<uint8\_t> removed(TOPK\_NUM);

for (int i = 0; i < TOPK\_NUM; i++) {

// 获取每一个i列数据的位信息

auto\* box\_buffer = output\_buffer + i \* output\_numprob;

// 其中4为confince置信度

if (box\_buffer[4] < confidence\_threshold) continue;

if (removed[i]) continue;

rst.emplace\_back();

auto& box = rst.back();

// box\_buffer + 5 位移到该地址

memcpy(&box.pts, box\_buffer + 5, 8 \* sizeof(float));

for (auto& pt : box.pts) {

pt.x = (pt.x - padd\_w\_) / (image\_size - 2 \* padd\_w\_) \* image\_width;

pt.y = (pt.y - padd\_h\_) / (image\_size - 2 \* padd\_h\_) \* image\_height;

};

box.confidence = sigmoid(box\_buffer[4]);//prob \* objness;

// 类型的指针为 13之后

float\* pclass = box\_buffer + modle\_last\_length;

box.label = argmax(pclass, num\_classes);

for (int j = i + 1; j < TOPK\_NUM; j++) {

auto\* box2\_buffer = output\_buffer + j \* output\_numprob;

if (box2\_buffer[4] < confidence\_threshold) continue;

if (removed[j]) continue;

if (is\_overlap(box\_buffer, box2\_buffer)) removed[j] = true;

};

};

std::vector<bbox\_t> out\_rst;

// label过滤

for (const auto& rst\_ : rst)

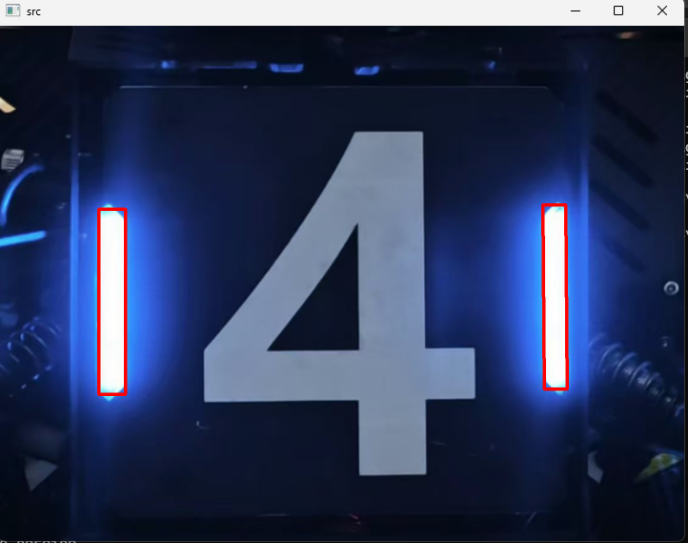
if (strip\_filter(rst\_.label, this->enemy\_blue))

out\_rst.push\_back(rst\_);

return out\_rst;

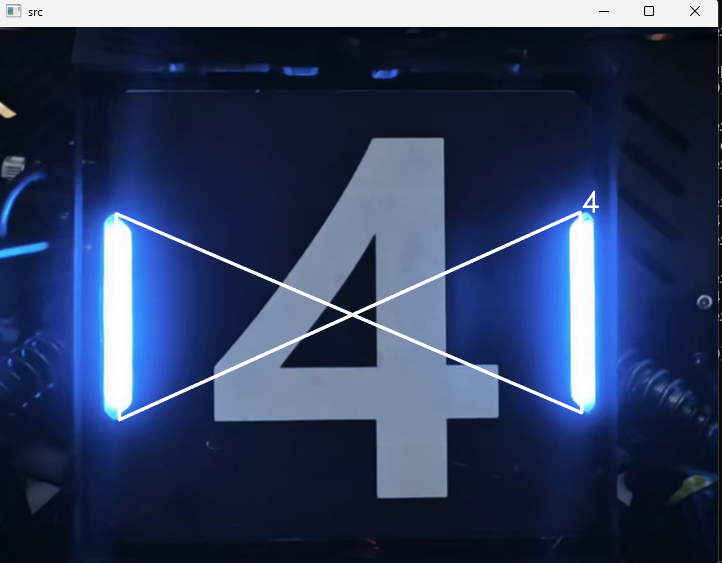
};

为进一步验证目标有效性并提高检测精度，系统提取候选装甲板的左右灯条ROI区域，结合Opencv图像处理方法进行精细化处理。通过灰度化获取单通道灰度图像，并利用二值化与轮廓提取技术筛选符合几何属性的灯条轮廓（图3-2）并拟合最小切矩形（图3-3）。

**图3-2 灯条轮廓提取图 图3-3 灯条切矩形拟合图**

在装甲板生成阶段，系统对左右灯条进行几何配对，基于长宽比、高度比及中心角度等多维度特征验证其有效性。针对多目标场景，引入非极大值抑制（NMS）与噪声去除算法，剔除重复或受干扰的检测结果，确保输出列表的纯净性。最终，通过敌方颜色过滤与尺寸分类，动态标记装大小装甲板，为后续跟踪与打击提供高置信度的目标信息，最终识别效果如3-4图所示。



**图3-4 识别效果展示图**

std::vector<SendArmor> FusionDetector::detect(cv::Mat src)

{

std::vector<SendArmor> sendarmors;

std::vector<Armor> armors = YoloModel\_\_->find\_armors(src);

for (int i = 0, I = armors.size(); i < I; i++) {

SendArmor send\_armor(armors[i])

cv::Rect L\_roi = capture\_light\_roi(true, armors[i].middle\_four\_points,this->\_width\_expand\_, this->\_height\_expand\_,

src.cols,src.rows);

cv::Rect R\_roi = capture\_light\_roi(false, armors[i].middle\_four\_points, this->\_width\_expand\_, this->\_height\_expand\_,

src.cols, src.rows);

cv::Mat L\_src = src(L\_roi); cv::Mat L\_drt = Machine\_\_->machine(L\_src);

cv::Mat R\_src = src(R\_roi); cv::Mat R\_drt = Machine\_\_->machine(R\_src);

// 提取灯条

std::vector<Light> left\_lights = MiddleLightFinder\_\_->find\_lights(L\_drt, L\_src);

std::vector<Light> right\_lights = MiddleLightFinder\_\_->find\_lights(R\_drt, R\_src);

// 跳过无灯条检测到的情况

bool ret\_light\_loss = true;

if (!left\_lights.empty() && !right\_lights.empty())

{

// 一旦出现灯条未在中心的情况

if (MiddleLightFinder\_\_->light\_center\_in\_rect\_middle(left\_lights[0], L\_roi)&&

MiddleLightFinder\_\_->light\_center\_in\_rect\_middle(right\_lights[0], R\_roi)) {

// 位置还原

Light left\_light = rm::Light(cv::RotatedRect(left\_lights[0].center + cv::Point2f(L\_roi.x, L\_roi.y), left\_lights[0].size

, left\_lights[0].angle));

Light right\_light = rm::Light(cv::RotatedRect(right\_lights[0].center + cv::Point2f(R\_roi.x, R\_roi.y), right\_lights[0].size

, right\_lights[0].angle));

// 匹配筛选

std::vector<rm::Armor> combination\_armor = ArmorFinder\_\_->find\_armors({ left\_light , right\_light });

if (!combination\_armor.empty()) {

ret\_light\_loss = false;

combination\_armor[0].label = armors[i].label;

combination\_armor[0].armor\_small = armors[i].label != 1;

send\_armor = SendArmor(combination\_armor[0]);

};

};

};

// 对未通过的装甲板赋值

if (ret\_light\_loss) {

send\_armor.light\_loss = true;

};

sendarmors.push\_back(send\_armor);

};

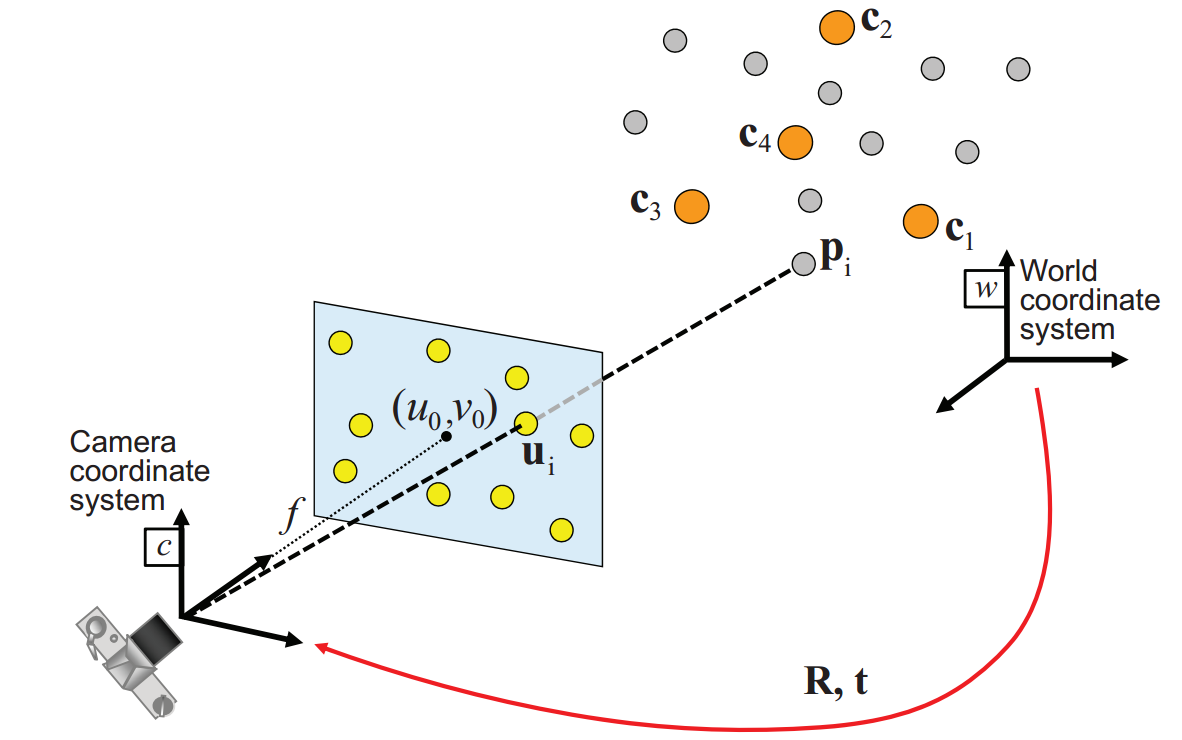
return sendarmors;

}

3.3.2目标运动解算

目标运动解算的目的是得到并预测敌方车辆在三维空间中的位置和运动状态，涉及到PnP相机位姿解算、多坐标系转换、朝向角解算、目标车辆状态估计等算法。

单目相机PnP解算的目的是得到目标装甲板相对于相机坐标系的位置，其原理如图3-5所示。



**图3-5 PnP算法原理图**

我们对OpenCV库中的solvePnP函数做了进一步封装，仅需提供灯条四点在像素坐标系的位置、灯条四点局部坐标系的位置、相机的内外参矩阵便可以得到tvec，即目标装甲板相对于相机坐标系的位置。

bool Coordinate::pnp(cv::Mat& Rvec, cv::Mat& Tvec, cv::Mat intrinsic, cv::Mat distcoeffs,

const std::vector<cv::Point2f>& four\_pixel\_points

, bool armor\_small\_size, bool SOLVEPNP\_IPPE)

{ //调用 OpenCV 的 solvePnP 函数，解算 Rvec 和 Tvec

return cv::solvePnP(armor\_small\_size ? small\_armor\_world\_points :

big\_armor\_world\_points,

four\_pixel\_points, intrinsic, distcoeffs, Rvec, Tvec, false,

SOLVEPNP\_IPPE ? cv::SOLVEPNP\_IPPE : cv::SOLVEPNP\_ITERATIVE);

};

然后我们把目标装甲板位置由相机坐标系转移到以以机器人云台固定点为原点的绝对坐标系上，这涉及到了坐标系的转换。

bool Coordinate::trans\_aromr\_pos(const float& car\_yaw, const float& car\_pitch,

ArmorPos& armorpos)

{

cv::Mat Rvec, Tvec; // Rvec: 旋转向量，Tvec: 平移向量（相机坐标系下）

// 使用 solvePnP 计算 Rvec 和 Tvec，失败则返回 false

if (!pnp(Rvec, Tvec, this->intrinsic, this->distcoeffs,

armorpos.pixel\_four\_point, armorpos.small\_armor, this->SOLVEPNP\_IPPE))

return false;

// 将 Tvec（cv::Mat）转换为 cv::Point3f，注意这里必须使用 double 类型访问元素

cv::Point3f tvec\_3f = cv::Point3f(

Tvec.at<double>(0),

Tvec.at<double>(1),

Tvec.at<double>(2)

);

// 打印位移向量（调试用）

std::cout << "Tvec\_3f: " << tvec\_3f << std::endl;

// 将相机坐标系下的位置转换为世界坐标（结合车体的 yaw 和 pitch）

armorpos.absolute\_middle\_point = cam2abs(tvec\_3f, car\_yaw, car\_pitch);

return true;

}

事实上我们在PnP解算时就可以得到装甲板的rvec，即位姿角，然而通过PnP解算的朝向角（yaw）存在很大误差，因此我们使用了装甲板映射算法来解算朝向角。

float IterationAngleSolver::absolute\_angle\_solve(const ArmorPos& armor\_pos,

const float& car\_yaw, const float& car\_pitch)

{

// 根据敌方颜色选择对应的角度补偿参数

float increase\_param = enemy\_blue ? increase\_param\_blue : increase\_param\_red;

// 设置朝向角的搜索范围为 [-车体朝向-90°, -车体朝向+90°]

float left = -car\_yaw - CV\_PI / 2;

float right = -car\_yaw + CV\_PI / 2;

// 设置迭代次数

int iteration = this->orientationangle\_iteration;

while (iteration--) {

// 三分法中的两个中间值

float m1 = left + (right - left) / 3;

float m2 = right - (right - left) / 3;

// 将装甲板绕 y 轴旋转 m1 和 m2 的角度，得到新的四个顶点的空间坐标

std::vector<cv::Point3f> points0 = rotate\_armor(armor\_pos.absolute\_middle\_point, m1, armor\_pos.small\_armor);

std::vector<cv::Point3f> points1 = rotate\_armor(armor\_pos.absolute\_middle\_point, m2, armor\_pos.small\_armor);

// 分别将两个姿态下的空间点投影回图像平面

std::vector<cv::Point2f> pointspix0(4), pointspix1(4);

for (int i = 0; i < 4; i++) {

pointspix0[i] = Coordinate\_\_->abs2pixel(points0[i], car\_yaw, car\_pitch);

pointspix1[i] = Coordinate\_\_->abs2pixel(points1[i], car\_yaw, car\_pitch);

};

// 比较投影后的四边形和原始像素位置之间的误差，保留误差小的一侧

if (cost(armor\_pos.pixel\_four\_point, pointspix0)

< cost(armor\_pos.pixel\_four\_point, pointspix1)) {

right = m2; // 舍弃右侧

}

else {

left = m1; // 舍弃左侧

};

};

// 得到最终估计的面朝角（在收缩后的区间中心）

float face\_angle = (right + left) / 2;

// 应用颜色对应的补偿参数，提升朝向角估计精度

face\_angle = increase\_angle(face\_angle, car\_yaw, increase\_param);

return face\_angle;

}

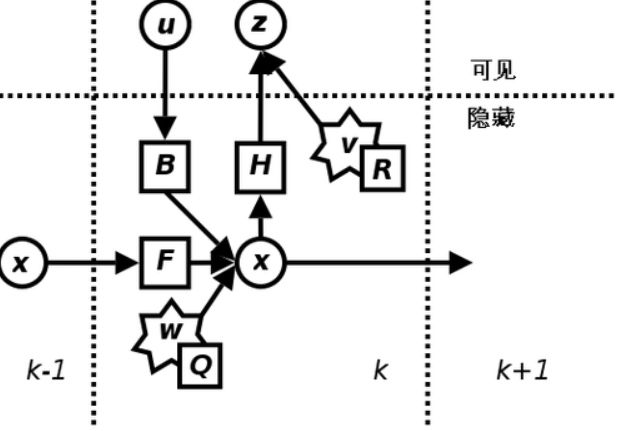
结合装甲板安装要求（pitch=15°，row=0°），我们就可以得到装甲板的位姿。

关于运动学建模，这里我们采用了匀速（CV）模型，将加速度视为零均值的高斯噪声，利用卡尔曼滤波器融合观测值和预测值，实现运动学建模（图3-6）。接下来以匀速转动+平动的运动学模型作为示例，构建机器人运动状态观测器。首先，规则手册规定机器人的装甲板安装必须成轴对称设计且对侧安装的装甲板两两之间的连线相互垂直。那么，机器人底盘的运动可以分解为绕中心的转动和底盘刚体的平动。



**图3-6机器人建模图**

根据统计这里一共有8个状态量：，分别为车辆中心的坐标、装甲板的朝向角、车辆在方向上的速度分量以及车辆自旋的角速度。的其中在前面位姿解算已经得到，而是通过把放入卡尔曼滤波观测得到的，卡尔曼滤波原理如图3-7所示。



**图3-7 卡尔曼滤波原理图**

MIddleCarState::MIddleCarState(const std::vector<double>& q\_vs, const std::vector<double>& r\_vs)

:CarStateBase()

{

d\_t = 0.01;

auto f0 = [this](const Eigen::VectorXf& x) {

Eigen::VectorXf x\_new = x;

x\_new(0) += x(4) \* d\_t;

x\_new(1) += x(5) \* d\_t;

x\_new(2) += x(6) \* d\_t;

x\_new(3) += x(7) \* d\_t;

return x\_new;

};

// J\_f - Jacobian of process function

auto j\_f0 = [this](const Eigen::VectorXf& x) {

Eigen::Matrix<float, 8, 8> f;

// clang-format off

f <<

1, 0, 0, 0, d\_t, 0, 0, 0,

0, 1, 0, 0, 0, d\_t, 0, 0,

0, 0, 1, 0, 0, 0, d\_t, 0,

0, 0, 0, 1, 0, 0, 0, d\_t,

0, 0, 0, 0, 1, 0, 0, 0,

0, 0, 0, 0, 0, 1, 0, 0,

0, 0, 0, 0, 0, 0, 1, 0,

0, 0, 0, 0, 0, 0, 0, 1;

// clang-format on

return f;

};

// h - Observation function

auto h0 = [this](const Eigen::VectorXf& x) {

Eigen::VectorXf z(4);

z(0) = x(0);

z(1) = x(1);

z(2) = x(2);

z(3) = x(3);

return z;

};

// J\_h - Jacobian of observation function

auto j\_h0 = [this](const Eigen::VectorXf& x) {

Eigen::Matrix<float, 4, 8> h;

h <<

1, 0, 0, 0, 0, 0, 0, 0,

0, 1, 0, 0, 0, 0, 0, 0,

0, 0, 1, 0, 0, 0, 0, 0,

0, 0, 0, 1, 0, 0, 0, 0;

return h;

};

Eigen::DiagonalMatrix<float, 8> q0;

q0.diagonal() << q\_vs[0], q\_vs[1], q\_vs[2], q\_vs[3], q\_vs[4], q\_vs[5],

q\_vs[6], q\_vs[7];

//xa ya za yaw

Eigen::DiagonalMatrix<float, 4> r0;

r0.diagonal() << r\_vs[0], r\_vs[1], r\_vs[2], r\_vs[3];

// P - error estimate covariance matrix

Eigen::DiagonalMatrix<float, 8> p0;

p0.setIdentity();

为建立起完整的车辆观测器，我们还需要得到车辆半径和高低装甲板高度差，我们的计算思路是：当敌方车辆在相机视野中同时露出两块装甲板时，作过两装甲板中心点的垂线得到交点，交点便是车辆中心点，再计算车辆中心点到两装甲板的欧氏距离，即可得到敌方车辆的长短轴半径，而高度差直接通过z轴坐标相减即可得到。

ArmorPos DoubleSolver::calculate\_car\_size(float& high\_armor\_radius, float& low\_armor\_radius, float& high\_low\_height\_differ,

const ArmorPos& left\_armor\_pos, const ArmorPos& right\_armor\_pos)

{

// 初始化高、低装甲板位置变量

ArmorPos high\_armor\_pos, low\_armor\_pos;

// 计算左装甲板和右装甲板的y坐标（图像中y值越小表示位置越高）

float left\_armor\_pos\_y = 0;

for (const auto& x : left\_armor\_pos.pixel\_four\_point)

left\_armor\_pos\_y += x.y; // 将左装甲板四个角点的y值加起来

left\_armor\_pos\_y /= 4.0; // 计算左装甲板的平均y坐标

float right\_armor\_pos\_y = 0;

for (const auto& x : right\_armor\_pos.pixel\_four\_point)

right\_armor\_pos\_y += x.y; // 将右装甲板四个角点的y值加起来

right\_armor\_pos\_y /= 4.0; // 计算右装甲板的平均y坐标

// 根据图像中的y坐标判断哪个装甲板位置更高（y值越小，装甲板越高）

if (left\_armor\_pos\_y < right\_armor\_pos\_y) {

high\_armor\_pos = left\_armor\_pos; // 左装甲板更高

low\_armor\_pos = right\_armor\_pos; // 右装甲板更低

}

else {

high\_armor\_pos = right\_armor\_pos; // 右装甲板更高

low\_armor\_pos = left\_armor\_pos; // 左装甲板更低

}

// 根据高装甲板的朝向角度（face\_angle）计算交点

float k\_next = tan(high\_armor\_pos.face\_angle); // 高装甲板的斜率

float k = -1.0 / k\_next; // 低装甲板的斜率，垂直于高装甲板的斜率

cv::Point2f high\_armor\_pos2f = cv::Point2f(high\_armor\_pos.absolute\_middle\_point.x, high\_armor\_pos.absolute\_middle\_point.z); // 高装甲板的中点坐标

cv::Point2f low\_armor\_pos2f = cv::Point2f(low\_armor\_pos.absolute\_middle\_point.x, low\_armor\_pos.absolute\_middle\_point.z); // 低装甲板的中点坐标

// 计算两个装甲板的交点，作为车体的中点

cv::Point2f car\_middle = get\_intersection\_point(k\_next, k, low\_armor\_pos2f, high\_armor\_pos2f);

// 计算高、低装甲板的半径（车体的半径）

high\_armor\_radius = cv::norm(car\_middle - high\_armor\_pos2f); // 车体中点到高装甲板的距离（半径）

low\_armor\_radius = cv::norm(car\_middle - low\_armor\_pos2f); // 车体中点到低装甲板的距离（半径）

// 计算高、低装甲板的高度差

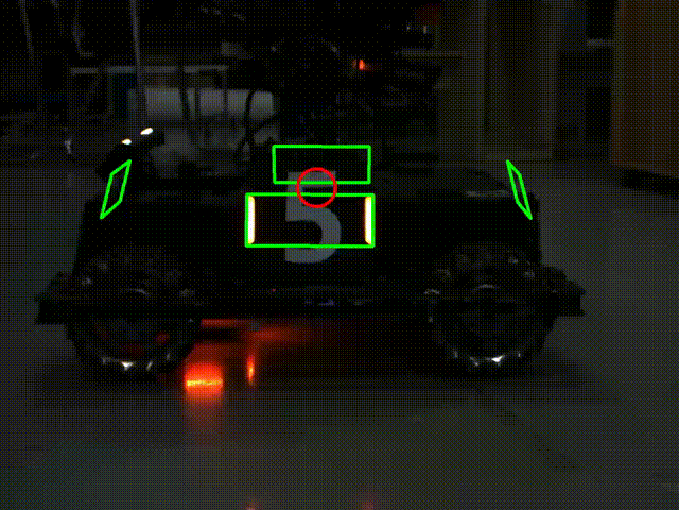
high\_low\_height\_differ = high\_armor\_pos.absolute\_middle\_point.y - low\_armor\_pos.absolute\_middle\_point.y;

// 返回高装甲板的位置信息

return high\_armor\_pos;

}

至此，我们已经得到了敌方车辆的所有运动状态量，建立起了完整的观测模型（图3-8）。



**图3-8 整车观测图**

3.3.3 火控策略

火控层作为机器人控制系统的关键组成部分，其核心作用在于精确控制射击过程，以确保在复杂多变的战场环境中能够精准、及时地击打敌方装甲板。火控层的主要功能包括击打策略选择以及开火判断，它通过一系列精心设计的算法和策略，对目标信息进行分析处理，并结合机器人的当前状态选择合适的击打策略，做出开火判断。

针对敌方机器人转速差异与距离远近，我们设计了不同的击打模式：平移击打模式、跟随击打模式、中心击打模式以及精准击打模式。

/\* 平移击打模式

适用场景：当目标转速过低（低于min\_shoot\_track\_rpm）时，直接瞄准最近的目标点，避免因目标移动过快丢失瞄准。\*/

case rm::StrategyPlanner::STATE::shoot\_translation:

{

// 选取击打目标

aim\_target = ShootTranslationStrategy\_\_->translation\_target\_choose(

Coordinate\_\_, pred\_car\_four\_points\_shoot, car\_yaw,car\_pitch);

shoot\_time = ShootTranslationStrategy\_\_->calculate\_target(aim\_target, { aim\_target }, pred\_state,car\_yaw);

shoot\_time = 0.0; // 一直开火

};

/\* 跟随击打模式

适用场景：中高转速（min\_shoot\_middle\_rpm < rpm\_abs < min\_shoot\_precise\_rpm）和目标在有效跟踪距离内（min\_shoot\_track\_distance < distance < max\_shoot\_track\_distance）\*/

case rm::StrategyPlanner::STATE::shoot\_track:

{

cv::Point3f absolute\_shoot\_target = ShootTrackStrategy\_\_->choose\_absolute\_target(

pred\_car\_four\_points\_absolute, pred\_car\_four\_points\_shoot, pred\_state, car\_yaw);

aim\_target = Coordinate\_\_->abs2sot(absolute\_shoot\_target, car\_yaw, car\_pitch, car\_yaw\_speed);

shoot\_time = ShootTrackStrategy\_\_->calculate\_target(aim\_target, pred\_car\_four\_points\_shoot, pred\_state, car\_yaw);

// shoot\_time = 0.0;

aim\_preview = ShootTrackStrategy\_\_->get\_aim\_preview();

};

/\* 中心击打模式

适用场景：转速适中（min\_shoot\_track\_rpm < rpm\_abs < min\_shoot\_precise\_rpm）及目标距离超出跟踪模式范围（distance > max\_shoot\_track\_distance 或 distance < min\_shoot\_track\_distance）\*/

case rm::StrategyPlanner::STATE::shoot\_middle:

{

if(last\_iteration) { // 处于迭代,仅仅计算最后一次目标的延迟

shoot\_time = ShootMiddleStrategy\_\_->calculate\_target(aim\_target, pred\_car\_four\_points\_shoot, pred\_state, car\_yaw);

}

else{

aim\_target = ShootMiddleStrategy\_\_->choose\_target(pred\_car\_four\_points\_shoot, pred\_state);

shoot\_time = -1.0;

};

};

/\* 精准击打模式

适用场景：高转速（rpm\_abs > min\_shoot\_precise\_rpm） \*/

case rm::StrategyPlanner::STATE::shoot\_precise:

{

if (last\_iteration) { // 处于迭代,仅仅计算最后一次目标的延迟

shoot\_time = PrecisionShootStrategy\_\_->calculate\_target(aim\_target, pred\_car\_four\_points\_shoot, pred\_state, car\_yaw);

}

else {

aim\_target = PrecisionShootStrategy\_\_->choose\_target(pred\_car\_four\_points\_shoot, pred\_state);

shoot\_time = -1.0;

};

}

}

**第四部分 导航算法**

**4.1 背景与目标**

智能遥控战车应具备自动规划路径，定点导航的功能。其设计旨在通过跨学科知识的融合对学生专业技能进行提高，激发学生的学习兴趣，打破专业壁垒，提升他们解决复杂工程问题的能力，从而有效应对具有复杂性和全局性的问题。同时针对导航这一实践性项目，能够以问题驱动学生学习，培养学生的创新思维与工程实践能力。在导航系统方面，本项目需要进行全面的导航系统设计，以实现精确扫描障碍物、定点导航的目的。这一设计不仅要求学生掌握多学科专业知识，还需具备将知识整合运用到实际的能力，从而确保战车在复杂的环境中能够快速避障并精确到达目标点。

**4.2 导航避障算法设计**

在导航系统的设计中，机器人需要灵活快速地到达目的地，这要求系统能够精确地获取周围障碍物的实时点云信息，从而快速规划出有效且安全的行驶路线。比赛中包含中心增益区和两个S形条状障碍物（如图1所示），可能被识别为障碍物的物体包含敌我双方机器人以及S形障碍物。在比赛场景中，本遥控机器人需要在比赛开始时快速到达中心增益区并且根据场上形势进行判断所需要到达的目的地，以保证机器人能够实现进攻和防守两端兼顾的作用。判断依据包括敌我双方机器人的血量以及比赛时间，在自身血量健康的时候能够发挥哨兵的等级和火力优势，在血量较少时也能够即使撤退，保证自身安全，达到团队利益最大化。

在实现过程中需要经过建立地图，重定位，发布目标点和规划路径的过程，具体实现方式可以参考以下设计方案。

**4.3 设计方案**

为满足雷达能够无死角扫描场地上的所有障碍物，所以在机体上采用的将雷达导致安装的方式，再对障碍物的高度进行过滤，防止出现由高度问题导致雷达扫描不到的问题。



**图4-1 雷达实际安装图**

**4.3.1 地图建立算法**

静态地图的建立主要采用fastlio算法，主要原理为通过雷达扫描获取点云信息对点云信息进行滤波过滤，正确处理并得到障碍物点云信息后再由三维点云信息映射获得二维地图。另外fastlio还为机器人定位提供了imu对机体速度和加速度的转换。



**图4-2 PCD点云转栅格地图**

//更新地图边界函数

void lasermap\_fov\_segment()

{

cub\_needrm.clear();

kdtree\_delete\_counter = 0;

kdtree\_delete\_time = 0.0;

//转化为世界坐标系

pointBodyToWorld(XAxisPoint\_body, XAxisPoint\_world);

V3D pos\_LiD = pos\_lid;

//初始化地图

if (!Localmap\_Initialized) {

for (int i = 0; i < 3; i++) {

LocalMap\_Points.vertex\_min[i] = pos\_LiD(i) - cube\_len / 2.0;

LocalMap\_Points.vertex\_max[i] = pos\_LiD(i) + cube\_len / 2.0;

}

Localmap\_Initialized = true;

return;

}

//计算雷达到地图边界的距离

float dist\_to\_map\_edge[3][2];

bool need\_move = false;

for (int i = 0; i < 3; i++) {

dist\_to\_map\_edge[i][0] = fabs(pos\_LiD(i) - LocalMap\_Points.vertex\_min[i]);

dist\_to\_map\_edge[i][1] = fabs(pos\_LiD(i) - LocalMap\_Points.vertex\_max[i]);

if (dist\_to\_map\_edge[i][0] <= MOV\_THRESHOLD \* DET\_RANGE || dist\_to\_map\_edge[i][1] <= MOV\_THRESHOLD \* DET\_RANGE) need\_move = true;

}

if (!need\_move) return;

BoxPointType New\_LocalMap\_Points, tmp\_boxpoints;

New\_LocalMap\_Points = LocalMap\_Points;

float mov\_dist = max((cube\_len - 2.0 \* MOV\_THRESHOLD \* DET\_RANGE) \* 0.5 \* 0.9, double(DET\_RANGE \* (MOV\_THRESHOLD - 1)));

//根据雷达的位置更新地图边界

for (int i = 0; i < 3; i++) {

tmp\_boxpoints = LocalMap\_Points;

if (dist\_to\_map\_edge[i][0] <= MOV\_THRESHOLD \* DET\_RANGE) {

New\_LocalMap\_Points.vertex\_max[i] -= mov\_dist;

New\_LocalMap\_Points.vertex\_min[i] -= mov\_dist;

tmp\_boxpoints.vertex\_min[i] = LocalMap\_Points.vertex\_max[i] - mov\_dist;

cub\_needrm.push\_back(tmp\_boxpoints);

// 将超出新边界的点加入删除列表

}

else if (dist\_to\_map\_edge[i][1] <= MOV\_THRESHOLD \* DET\_RANGE) {

New\_LocalMap\_Points.vertex\_max[i] += mov\_dist;

New\_LocalMap\_Points.vertex\_min[i] += mov\_dist;

tmp\_boxpoints.vertex\_max[i] = LocalMap\_Points.vertex\_min[i] + mov\_dist;

cub\_needrm.push\_back(tmp\_boxpoints);

// 将超出新边界的点加入删除列表

}

}

LocalMap\_Points = New\_LocalMap\_Points;

points\_cache\_collect();//收集缓存的点

double delete\_begin = omp\_get\_wtime();

if (cub\_needrm.size() > 0) kdtree\_delete\_counter = ikdtree.Delete\_Point\_Boxes(cub\_needrm);

kdtree\_delete\_time = omp\_get\_wtime() - delete\_begin;

}

//采集点云信息函数

void map\_incremental()

{

// 创建两个点云向量，用于存储需要添加到地图中的点和不需要下采样的点

PointVector PointToAdd;

PointVector PointNoNeedDownsample;

PointToAdd.reserve(feats\_down\_size);

PointNoNeedDownsample.reserve(feats\_down\_size);

for (int i = 0; i < feats\_down\_size; i++)

{

/\* 将点从体坐标系转换到世界坐标系 \*/

pointBodyToWorld(&(feats\_down\_body->points[i]), &(feats\_down\_world->points[i]));

/\* 决定是否需要将该点添加到地图中 \*/

if (!Nearest\_Points[i].empty() && flg\_EKF\_inited)

// 如果找到了邻近点且EKF已初始化

{

const PointVector &points\_near = Nearest\_Points[i];

bool need\_add = true;

// 计算当前点的中心点（用于下采样）

BoxPointType Box\_of\_Point;

PointType downsample\_result, mid\_point;

mid\_point.x = floor(feats\_down\_world->points[i].x/filter\_size\_map\_min)\*filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

mid\_point.y = floor(feats\_down\_world->points[i].y/filter\_size\_map\_min)\*filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

mid\_point.z = floor(feats\_down\_world->points[i].z/filter\_size\_map\_min)\*filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

float dist = calc\_dist(feats\_down\_world->points[i],mid\_point);

//若不处于以中心点为中心的滤波网格则不需要进行采集，此步判断是为了防止雷达将机体判断为障碍物而出现误识别扫描

if (

fabs(points\_near[0].x - mid\_point.x) > 0.5 \* filter\_size\_map\_min && fabs(points\_near[0].y - mid\_point.y) > 0.5 \* filter\_size\_map\_min && fabs(points\_near[0].z - mid\_point.z) > 0.5 \* filter\_size\_map\_min){

PointNoNeedDownsample.push\_back(feats\_down\_world->points[i]);

continue;

}

for (int readd\_i = 0; readd\_i < NUM\_MATCH\_POINTS; readd\_i ++)

{

if (points\_near.size() < NUM\_MATCH\_POINTS) break;

if (calc\_dist(points\_near[readd\_i], mid\_point) < dist)

{

need\_add = false;

break;

}

}

if (need\_add) PointToAdd.push\_back(feats\_down\_world->points[i]);

}

else

{

PointToAdd.push\_back(feats\_down\_world->points[i]);

}

}

//将点云信息保存下来作为PCD文件

void map\_incremental()

{

// 创建两个点云向量，用于存储需要添加到地图中的点和不需要下采样的点

PointVector PointToAdd;

PointVector PointNoNeedDownsample;

PointToAdd.reserve(feats\_down\_size);

PointNoNeedDownsample.reserve(feats\_down\_size);

for (int i = 0; i < feats\_down\_size; i++)

{

/\* 将点从体坐标系转换到世界坐标系 \*/

pointBodyToWorld(&(feats\_down\_body->points[i]), &(feats\_down\_world->points[i]));

/\* 决定是否需要将该点添加到地图中 \*/

if (!Nearest\_Points[i].empty() && flg\_EKF\_inited)

// 如果找到了邻近点且EKF已初始化

{

const PointVector& points\_near = Nearest\_Points[i];

bool need\_add = true;

// 计算当前点的中心点（用于下采样）

BoxPointType Box\_of\_Point;

PointType downsample\_result, mid\_point;

mid\_point.x = floor(feats\_down\_world->points[i].x / filter\_size\_map\_min) \* filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

mid\_point.y = floor(feats\_down\_world->points[i].y / filter\_size\_map\_min) \* filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

mid\_point.z = floor(feats\_down\_world->points[i].z / filter\_size\_map\_min) \* filter\_size\_map\_min + 0.5 \* filter\_size\_map\_min;

float dist = calc\_dist(feats\_down\_world->points[i], mid\_point);

//若不处于以中心点为中心的滤波网格则不需要进行采集，此步判断是为了防止雷达将机体判断为障碍物而出现误识别扫描

if (

fabs(points\_near[0].x - mid\_point.x) > 0.5 \* filter\_size\_map\_min && fabs(points\_near[0].y - mid\_point.y) > 0.5 \* filter\_size\_map\_min && fabs(points\_near[0].z - mid\_point.z) > 0.5 \* filter\_size\_map\_min) {

PointNoNeedDownsample.push\_back(feats\_down\_world->points[i]);

continue;

}

for (int readd\_i = 0; readd\_i < NUM\_MATCH\_POINTS; readd\_i++)

{

if (points\_near.size() < NUM\_MATCH\_POINTS) break;

if (calc\_dist(points\_near[readd\_i], mid\_point) < dist)

{

need\_add = false;

break;

}

}

if (need\_add) PointToAdd.push\_back(feats\_down\_world->points[i]);

}

else

{

PointToAdd.push\_back(feats\_down\_world->points[i]);

}

}

//由fastlio提供的imu转换与信息发布

void imu\_cbk(const sensor\_msgs::Imu::ConstPtr &msg\_in)

{

publish\_count++; // 增加发布计数

// 创建一个新的IMU消息指针，复制输入消息内容

sensor\_msgs::Imu::Ptr msg(new sensor\_msgs::Imu(\*msg\_in));

// 取反加速度数据的y和z轴

msg->linear\_acceleration.y = -msg->linear\_acceleration.y;

msg->linear\_acceleration.z = -msg->linear\_acceleration.z;

// 取反角速度数据的y和z轴

msg->angular\_velocity.y = -msg->angular\_velocity.y;

msg->angular\_velocity.z = -msg->angular\_velocity.z;

// 根据激光雷达和IMU之间的时间差调整IMU消息的时间戳

msg->header.stamp = ros::Time().fromSec(msg\_in->header.stamp.toSec() - time\_diff\_lidar\_to\_imu);

// 如果启用了时间同步，并且激光雷达和IMU之间的时间差超过阈值，则使用自动同步的时间差

if (abs(timediff\_lidar\_wrt\_imu) > 0.1 && time\_sync\_en)

{

msg->header.stamp = ros::Time().fromSec(timediff\_lidar\_wrt\_imu + msg\_in->header.stamp.toSec());

}

double timestamp = msg->header.stamp.toSec();

mtx\_buffer.lock();

// 检查时间戳，防止IMU数据回环

if (timestamp < last\_timestamp\_imu)

{

ROS\_WARN("imu loop back, clear buffer"); // 输出警告信息

imu\_buffer.clear(); // 清空IMU缓冲区

}

// 更新最后接收到的IMU时间戳

last\_timestamp\_imu = timestamp;

// 将处理后的IMU消息添加到缓冲区中

imu\_buffer.push\_back(msg);

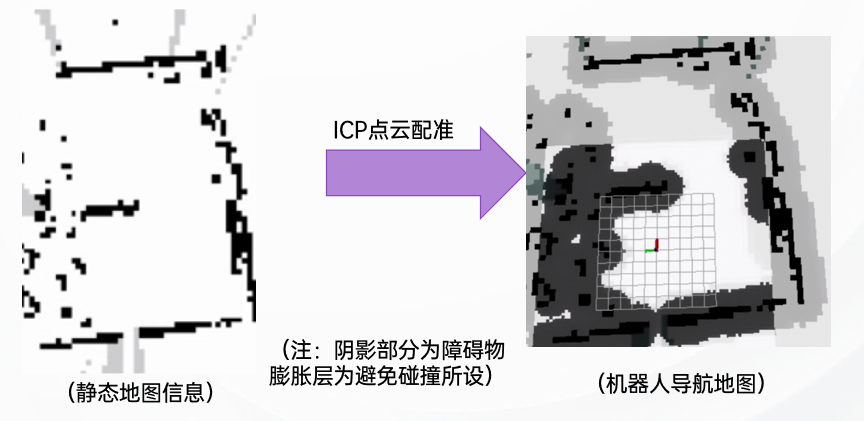
mtx\_buffer.unlock();

sig\_buffer.notify\_all();

}

**4.3.2 重定位算法**

重定位算法用于确认机器人的实时位置，其主要实现方式是通过ICP算法，即通过对雷达实时扫描获取的点云信息以及算法提供的里程计与静态地图进行比对，再计算点云与静态地图的匹配重合度来确认机器人的实时位置为后续的路径规划提供起始位置。



**图4-3 重定位效果图**

void h\_share\_model(state\_ikfom& s, esekfom::dyn\_share\_datastruct<double>& ekfom\_data)

{

double match\_start = omp\_get\_wtime(); // 记录匹配开始时间

laserCloudOri->clear(); // 清空原始激光点云

corr\_normvect->clear(); // 清空相关法向量点云

total\_residual = 0.0; // 初始化总残差为0

/\*\* 最邻近表面搜索和残差计算 \*\*/

#ifdef MP\_EN

omp\_set\_num\_threads(MP\_PROC\_NUM); // 如果启用多线程处理，设置线程数

#pragma omp parallel for // 使用OpenMP并行化for循环

#endif

for (int i = 0; i < feats\_down\_size; i++) // 遍历每个特征点

{

PointType& point\_body = feats\_down\_body->points[i]; // 获取体坐标系中的点

PointType& point\_world = feats\_down\_world->points[i]; // 获取世界坐标系中的点

/\* 将点从体坐标系转换到世界坐标系 \*/

V3D p\_body(point\_body.x, point\_body.y, point\_body.z);

V3D p\_global(s.rot \* (s.offset\_R\_L\_I \* p\_body + s.offset\_T\_L\_I) + s.pos);

point\_world.x = p\_global(0);

point\_world.y = p\_global(1);

point\_world.z = p\_global(2);

point\_world.intensity = point\_body.intensity;

vector<float> pointSearchSqDis(NUM\_MATCH\_POINTS); // 存储最近邻搜索的平方距离

auto& points\_near = Nearest\_Points[i]; // 获取最近邻点

if (ekfom\_data.converge) // 如果EKF已收敛

{

/\*\* 在地图中找到最近的表面 \*\*/

ikdtree.Nearest\_Search(point\_world, NUM\_MATCH\_POINTS, points\_near, pointSearchSqDis);

point\_selected\_surf[i] = points\_near.size() < NUM\_MATCH\_POINTS ? false : pointSearchSqDis[NUM\_MATCH\_POINTS - 1] > 5 ? false : true; // 判断点是否有效

}

if (!point\_selected\_surf[i]) continue; // 如果点无效，则跳过

VF(4) pabcd; // 存储平面方程系数

point\_selected\_surf[i] = false; // 重置选择标志

if (esti\_plane(pabcd, points\_near, 0.1f)) // 估计平面方程

{

float pd2 = pabcd(0) \* point\_world.x + pabcd(1) \* point\_world.y + pabcd(2) \* point\_world.z + pabcd(3); // 计算点到平面的距离

float s = 1 - 0.9 \* fabs(pd2) / sqrt(p\_body.norm()); // 计算残差

if (s > 0.9) // 如果残差小于阈值

{

point\_selected\_surf[i] = true; // 设置点有效

normvec->points[i].x = pabcd(0); // 存储法向量x分量

normvec->points[i].y = pabcd(1); // 存储法向量y分量

normvec->points[i].z = pabcd(2); // 存储法向量z分量

normvec->points[i].intensity = pd2; // 存储点到平面的距离

res\_last[i] = abs(pd2); // 存储残差的绝对值

}

}

}

effct\_feat\_num = 0; // 初始化有效特征点数

for (int i = 0; i < feats\_down\_size; i++) // 遍历每个特征点

{

if (point\_selected\_surf[i])

{

laserCloudOri->points[effct\_feat\_num] = feats\_down\_body->points[i];

corr\_normvect->points[effct\_feat\_num] = normvec->points[i]; // 存储相关法向量

total\_residual += res\_last[i]; // 累加残差

effct\_feat\_num++; // 增加有效特征点数

}

}

res\_mean\_last = total\_residual / effct\_feat\_num; // 计算平均残差

match\_time += omp\_get\_wtime() - match\_start; // 计算匹配时间

double solve\_start\_ = omp\_get\_wtime(); // 记录求解开始时间

/\*\*\* 计算测量雅可比矩阵H和测量向量h \*\*\*/

ekfom\_data.h\_x = MatrixXd::Zero(effct\_feat\_num, 12); // 初始化雅可比矩阵

ekfom\_data.h.resize(effct\_feat\_num);

for (int i = 0; i < effct\_feat\_num; i++) // 遍历每个有效特征点

{

const PointType& laser\_p = laserCloudOri->points[i];

V3D point\_this\_be(laser\_p.x, laser\_p.y, laser\_p.z);

M3D point\_be\_crossmat;

point\_be\_crossmat << SKEW\_SYM\_MATRX(point\_this\_be); // 计算体坐标系中点的叉乘矩阵

V3D point\_this = s.offset\_R\_L\_I \* point\_this\_be + s.offset\_T\_L\_I; // 将点从体坐标系转换到IMU坐标系

M3D point\_crossmat;

point\_crossmat << SKEW\_SYM\_MATRX(point\_this); // 计算IMU坐标系中点的叉乘矩阵

/\*\*\* 获取最近表面的法向量 \*\*\*/

const PointType& norm\_p = corr\_normvect->points[i];

V3D norm\_vec(norm\_p.x, norm\_p.y, norm\_p.z);

/\*\*\* 计算测量雅可比矩阵H \*\*\*/

V3D C(s.rot.conjugate() \* norm\_vec); // 旋转法向量到世界坐标系

V3D A(point\_crossmat \* C); // 计算雅可比矩阵的一部分

V3D B(point\_be\_crossmat \* s.offset\_R\_L\_I.conjugate() \* C); // 计算雅可比矩阵的另一部分

ekfom\_data.h\_x.block<1, 12>(i, 0) << norm\_p.x, norm\_p.y, norm\_p.z, VEC\_FROM\_ARRAY(A), VEC\_FROM\_ARRAY(B), VEC\_FROM\_ARRAY(C); // 存储雅可比矩阵行

/\*\*\* 测量向量：点到最近表面的距离 \*\*\*/

ekfom\_data.h(i) = -norm\_p.intensity; // 存储测量值

}

solve\_time += omp\_get\_wtime() - solve\_start\_; // 计算求解时间

}

其中，采用的核心算法为ICP算法，其主要原理是通过迭代的方式计算实时点云和地图点云的最小误差，其中还包含了最近点匹配、变换估计、应用变换和检查收敛的过程，最终输出两片匹配的最小误差就可以输出此时机器人所在的位置。

**4.3.3 目标点发布算法**

为了应对赛场上的复杂形式，我们同样需要接收不同的信息作为判断条件，再发布不同的目标点使机器人能够在适当的时间出现在适当的位置并且执行相应的任务。其中，在赛场上的判断信息主要包括敌我各机器人的血量，比赛阶段，针对不同的情况，我们还设计了追击模式，巡航模式，返航模式等等来实现占领中心增益点、追击残血目标、自身残血回家和保护其他机器人的功能。

//设定判断树将以下判断设为支判断

Judgment::Judgment(int branch\_size)

:branch\_size(branch\_size)

{}

Judgment::Judgment(int branch\_size,const std::string& name)

:branch\_size(branch\_size),name(name)

{}

int Judgment::get\_branch\_size() const

{

return this->branch\_size;

};

void Judgment::set\_name(const std::string& name)

{

this->name = name;

}

NameJudgment::NameJudgment(const std::string& name)

:Judgment(0,name)

{}

int NameJudgment::run(const perception\_msg& msg)

{

return -1;

};

//判断比赛进程，使机器人开始即可行动，结束即停

CompetitionProgress::CompetitionProgress()

:Judgment(2,"CompetitionProgress")

{

}

int CompetitionProgress::run(const perception\_msg& msg)

{

// 只要为4,即为比赛中

bool ret = msg.serial\_read\_data\_msg.competition\_situation == 4;

return ret;

};

SentryHp::SentryHp(int hp\_threshold)

:Judgment(2)

{

this->hp\_threshold = hp\_threshold;

this->set\_name("SentryHp-" + std::to\_string(hp\_threshold));

};

int SentryHp::run(const perception\_msg& msg)

{

bool ret = msg.serial\_read\_data\_msg.own\_sentry\_hp < hp\_threshold;

return ret;

}

TimeJudgment::TimeJudgment(double timing)

:Judgment(2)

{

this->timing = timing;

this->set\_name("TimeJudgment-" + std::to\_string(timing));

start = false;

};

int TimeJudgment::run(const perception\_msg& msg)

{

if(start){ // 已经启动,判断是否到计时时间

double now\_time = get\_now\_time();

return (now\_time - start\_time < timing);

}

else{

start = true;

start\_time = get\_now\_time();

return 1;

};

};

//判断是否需要保护其他机器人

ProtectJudgment::ProtectJudgment(int hp\_threshold)

:Judgment(3)

{

this->hp\_threshold = hp\_threshold;

this->set\_name("ProtectJudgment-" + std::to\_string(hp\_threshold));

};

int ProtectJudgment::run(const perception\_msg& msg)

{

// 子弹过低,不保护

if(msg.serial\_read\_data\_msg.bullets <= 0) return 0;

int hero\_hp = msg.serial\_read\_data\_msg.own\_hero\_hp;

int infantry\_hp = msg.serial\_read\_data\_msg.own\_infantry\_hp;

bool ret\_hero\_hp = hero\_hp < hp\_threshold && hero\_hp > 0;

bool ret\_infantry\_hp = infantry\_hp < hp\_threshold && infantry\_hp > 0;

if(ret\_hero\_hp && ret\_infantry\_hp){

return hero\_hp > infantry\_hp ? 1 : 2;

};

if(ret\_hero\_hp) return 2;

if(ret\_infantry\_hp) return 1;

return 0;

};

PursuitJudgment::PursuitJudgment(int hp\_threshold, float contain\_timer)

:Judgment(2),contain\_timer(contain\_timer),hp\_threshold(hp\_threshold)

{

first\_time = true;

this->set\_name("PursuitJudgment-" + std::to\_string(hp\_threshold));

};

//判断是否追击

int PursuitJudgment::run(const perception\_msg& msg)

{

// 子弹过低,不追击

if(msg.serial\_read\_data\_msg.bullets <= 0) return 0;

// 如果哨兵当前位置距离敌方补给区过近,不触发

float distance = sqrt((msg.sentry\_pos.x - competition\_enemy\_point.x) \* (msg.sentry\_pos.x - competition\_enemy\_point.x)

+ (msg.sentry\_pos.y - competition\_enemy\_point.y) \* (msg.sentry\_pos.y - competition\_enemy\_point.y));

if(distance < min\_pursuit\_to\_competition\_enemy\_point\_distance){

return 0;

};

//通过视觉传输敌方类型来判断追击对象

if(msg.serial\_read\_data\_msg.detect\_enemy\_kind > 0){

switch(msg.serial\_read\_data\_msg.detect\_enemy\_kind)

{

case 0:

return 0;

case 1: // 英雄

return msg.serial\_read\_data\_msg.enemy\_hero\_hp < this->hp\_threshold;

case 2: // 哨兵

return msg.serial\_read\_data\_msg.enemy\_sentry\_hp < this->hp\_threshold;

case 3: // 步兵

return msg.serial\_read\_data\_msg.enemy\_infantry\_hp < this->hp\_threshold;

default:

return 0;

};

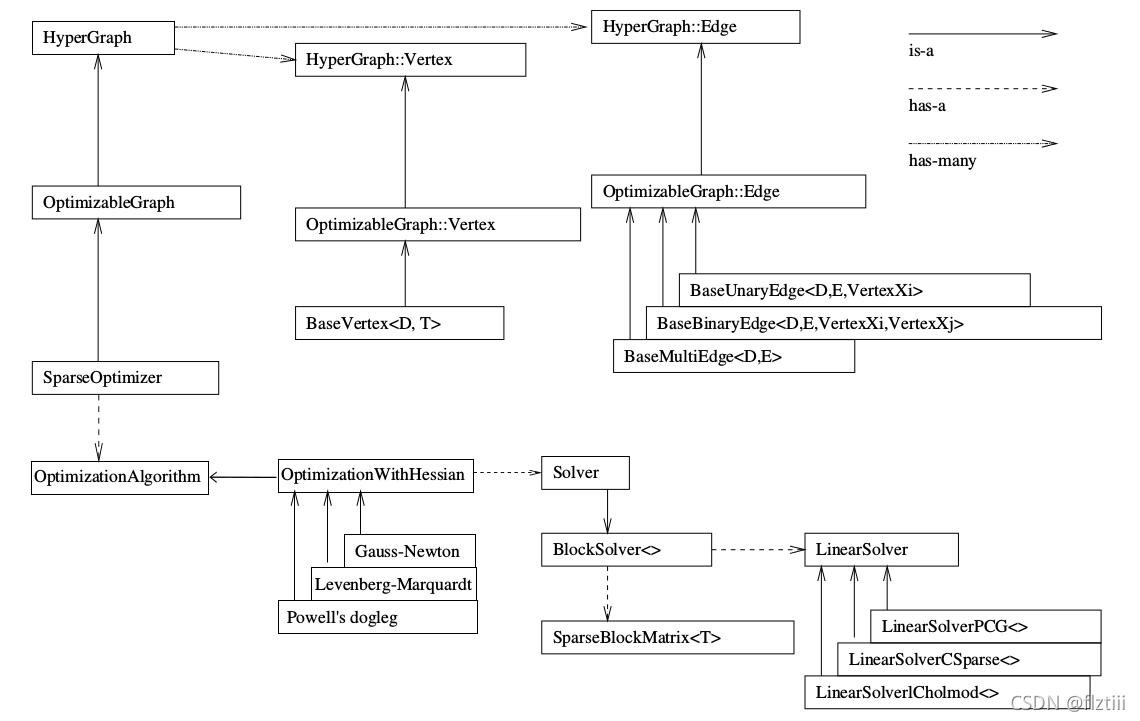
}

return 0;

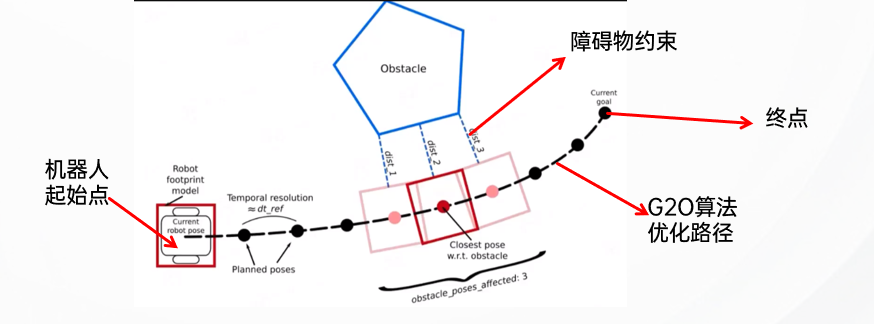
};

**4.3.4 路径规划算法（Teb算法）**

在拥有障碍物信息和自身机体的位置信息以及目标点坐标信息之后，就可以开始对机体需要的路径进行规划。本导航体系采用的是teb局部规划器，其原理将路径比作弹力带，再由周围的障碍物点云按照距离出现类似排斥力场，然后通过调整点云对路径的排斥力大小来调整出自己理想的路径以及位移速度。



**图4-4 Teb算法原理图**



**图4-5 Teb算法演示图**

// 从节点句柄加载ROS参数

config.loadRosParamFromNodeHandle(n);

// 创建两个定时器：一个用于主循环，一个用于发布周期

ros::Timer cycle\_timer = n.createTimer(ros::Duration(0.025), CB\_mainCycle);

ros::Timer publish\_timer = n.createTimer(ros::Duration(0.1), CB\_publishCycle);

// 设置规划器（基于同伦类规划或仅基于局部TEB规划器）

if (config.hcp.enable\_homotopy\_class\_planning)

planner = PlannerInterfacePtr(new HomotopyClassPlanner(config, &obst\_vector, robot\_model, visual, &via\_points));

else

planner = PlannerInterfacePtr(new TebOptimalPlanner(config, &obst\_vector, robot\_model, visual, &via\_points));

// TebOptimalPlanner的构造函数

TebOptimalPlanner::TebOptimalPlanner(const TebConfig & cfg, ObstContainer \* obstacles, RobotFootprintModelPtr robot\_model, TebVisualizationPtr visual, const ViaPointContainer \* via\_points)

{

initialize(cfg, obstacles, robot\_model, visual, via\_points);}

// 初始化优化器（设置求解器和块排序设置）

optimizer\_ = initOptimizer();

cfg\_ = &cfg;

obstacles\_ = obstacles;

robot\_model\_ = robot\_model;

via\_points\_ = via\_points;

cost\_ = HUGE\_VAL;

prefer\_rotdir\_ = RotType::none;

// 确保register\_g2o\_types只调用一次（线程安全）

static boost::once\_flag flag = BOOST\_ONCE\_INIT;

boost::call\_once(&registerG2OTypes, flag);

// 分配优化器

boost::shared\_ptr<g2o::SparseOptimizer> optimizer = boost::make\_shared<g2o::SparseOptimizer>();

std::unique\_ptr<TEBLinearSolver> linear\_solver(new TEBLinearSolver()); // 创建线性求解器（见optimization.h中的typedef）

linear\_solver->setBlockOrdering(true);

std::unique\_ptr<TEBBlockSolver> block\_solver(new TEBBlockSolver(std::move(linear\_solver)));

g2o::OptimizationAlgorithmLevenberg\* solver = new g2o::OptimizationAlgorithmLevenberg(std::move(block\_solver));

optimizer->setAlgorithm(solver);

optimizer->initMultiThreading();

// 如果轨迹未初始化，则初始化轨迹

if (!teb\_.isInit())

{

teb\_.initTrajectoryToGoal(start, goal, 0, cfg\_->robot.max\_vel\_x, cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion);

// 初始化到目标的轨迹，0个中间样本，但dt=1 -> autoResize将在第一次优化前添加更多样本

}

else // 如果轨迹已初始化，则尝试热启动

{

if (teb\_.sizePoses() > 0

&& (goal.position() - teb\_.BackPose().position()).norm() < cfg\_->trajectory.force\_reinit\_new\_goal\_dist

&& fabs(g2o::normalize\_theta(goal.theta() - teb\_.BackPose().theta())) < cfg\_->trajectory.force\_reinit\_new\_goal\_angular) // 检查是否可以热启动

teb\_.updateAndPruneTEB(start, goal, cfg\_->trajectory.min\_samples);

else

{

ROS\_DEBUG("New goal: distance to existing goal is higher than the specified threshold. Reinitalizing trajectories.");

teb\_.clearTimedElasticBand();

teb\_.initTrajectoryToGoal(start, goal, 0, cfg\_->robot.max\_vel\_x, cfg\_->trajectory.min\_samples, cfg\_->trajectory.allow\_init\_with\_backwards\_motion);

}

}

// 设置起始速度

if (start\_vel)

setVelocityStart(\*start\_vel);

// 设置目标速度自由

if (free\_goal\_vel)

setVelocityGoalFree();

else

vel\_goal\_.first = true;

// 优化TEB

return optimizeTEB(cfg\_->optim.no\_inner\_iterations, cfg\_->optim.no\_outer\_iterations);

// 自动调整TEB大小

teb\_.autoResize(cfg\_->trajectory.dt\_ref, cfg\_->trajectory.dt\_hysteresis, cfg\_->trajectory.min\_samples, cfg\_->trajectory.max\_samples, fast\_mode);

// 构建图

success = buildGraph(weight\_multiplier);

if (!optimizer\_->edges().empty() || !optimizer\_->vertices().empty())

{

ROS\_WARN("Cannot build graph, because it is not empty. Call graphClear()!");

return false;

}

// 设置计算批量统计信息

optimizer\_→setComputeBatchStatistics(cfg\_→recovery.divergence\_detection\_enable);

// 添加TEB顶点

AddTEBVertices();

unsigned int id\_counter = 0; // 用于顶点ID

obstacles\_per\_vertex\_.resize(teb\_.sizePoses()); // 调整障碍物列表大小

auto iter\_obstacle = obstacles\_per\_vertex\_.begin();

for (int i = 0; i < teb\_.sizePoses(); ++i)

{

teb\_.PoseVertex(i)->setId(id\_counter++); // 设置顶点ID

optimizer\_->addVertex(teb\_.PoseVertex(i)); // 添加顶点

if (teb\_.sizeTimeDiffs() != 0 && i < teb\_.sizeTimeDiffs())

{

teb\_.TimeDiffVertex(i)->setId(id\_counter++); // 设置时间差顶点ID

optimizer\_->addVertex(teb\_.TimeDiffVertex(i)); // 添加时间差顶点

}

iter\_obstacle->clear();

(iter\_obstacle++)->reserve(obstacles\_->size()); // 预留障碍物空间

}

// 添加障碍物边

AddEdgesObstacles(weight\_multiplier);

// 添加动态障碍物边

AddEdgesDynamicObstacles();

// 添加通过点边

AddEdgesViaPoints();

for (ViaPointContainer::const\_iterator vp\_it = via\_points\_->begin(); vp\_it != via\_points\_->end(); ++vp\_it)

{

int index = teb\_.findClosestTrajectoryPose(\*vp\_it, NULL, start\_pose\_idx); // 找到最接近的轨迹点

if (cfg\_->trajectory.via\_points\_ordered)

start\_pose\_idx = index + 2; // 跳过一个点，以便为后续通过点留出自由度

// 检查通过点是否与目标重合或位于目标之后

if (index > n - 2)

index = n - 2; // 设置为目标前一个点，因为可以移动它！

// 检查通过点是否与起点重合或位于起点之前

if (index < 1)

{

if (cfg\_->trajectory.via\_points\_ordered)

{

index = 1; // 尝试将通过点连接到第二个（且非固定）姿态。之后autoresize可能会在中间添加新姿态。

}

else

{

ROS\_DEBUG("TebOptimalPlanner::AddEdgesViaPoints(): skipping a via-point that is close or behind the current robot pose.");

continue; // 跳过非常接近或位于机器人当前姿态之后的通过点

}

}

Eigen::Matrix<double, 1, 1> information;

information.fill(cfg\_->optim.weight\_viapoint); // 设置通过点权重

EdgeViaPoint\* edge\_viapoint = new EdgeViaPoint;

edge\_viapoint->setVertex(0, teb\_.PoseVertex(index)); // 设置顶点

edge\_viapoint->setInformation(information); // 设置信息矩阵

edge\_viapoint->setParameters(\*cfg\_, &(\*vp\_it)); // 设置参数

optimizer\_->addEdge(edge\_viapoint); // 添加边

}

// 添加速度边

AddEdgesVelocity();

int n = teb\_.sizePoses();

Eigen::Matrix<double, 2, 2> information;

information(0, 0) = cfg\_->optim.weight\_max\_vel\_x; // 设置X方向速度权重

information(1, 1) = cfg\_->optim.weight\_max\_vel\_theta; // 设置角速度权重

information(0, 1) = 0.0;

information(1, 0) = 0.0;

for (int i = 0; i < n - 1; ++i)

{

EdgeVelocity\* velocity\_edge = new EdgeVelocity;

velocity\_edge->setVertex(0, teb\_.PoseVertex(i)); // 设置起始顶点

velocity\_edge->setVertex(1, teb\_.PoseVertex(i + 1)); // 设置目标顶点

velocity\_edge->setVertex(2, teb\_.TimeDiffVertex(i)); // 设置时间差顶点

velocity\_edge->setInformation(information); // 设置信息矩阵

velocity\_edge->setTebConfig(\*cfg\_); // 设置TEB配置

optimizer\_->addEdge(velocity\_edge); // 添加边

}

// 添加加速度边

AddEdgesAcceleration();

EdgeAcceleration\* acceleration\_edge = new EdgeAcceleration;

acceleration\_edge->setVertex(0, teb\_.PoseVertex(i));

acceleration\_edge->setVertex(1, teb\_.PoseVertex(i + 1));

acceleration\_edge->setVertex(2, teb\_.PoseVertex(i + 2));

acceleration\_edge->setVertex(3, teb\_.TimeDiffVertex(i));

acceleration\_edge->setVertex(4, teb\_.TimeDiffVertex(i + 1));

acceleration\_edge->setInformation(information);

acceleration\_edge->setTebConfig(\*cfg\_);

optimizer\_→addEdge(acceleration\_edge);

// 添加最短路径边

AddEdgesShortestPath();

for (int i = 0; i < teb\_.sizePoses() - 1; ++i)

{

EdgeShortestPath\* shortest\_path\_edge = new EdgeShortestPath;

shortest\_path\_edge->setVertex(0, teb\_.PoseVertex(i));

shortest\_path\_edge->setVertex(1, teb\_.PoseVertex(i + 1));

shortest\_path\_edge->setInformation(information);

shortest\_path\_edge->setTebConfig(\*cfg\_);

optimizer\_->addEdge(shortest\_path\_edge);

}

**第五部分 硬件设计**

5.1 全向轮战车基础硬件设计

5.1.1 全向轮战车器件

**表5-1 全向轮战车动力器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| RoboMaster M3508 直流无刷减速电机 | 6 | 用于底盘动力输出和摩擦轮动力输出 |
| RoboMaster GM6020 直流无刷云台电机 | 3 | 用于底盘转舵动力输出和云台动力输出 |
| RoboMaster M2006 直流无刷减速电机 | 1 | 拨盘动力输出 |

**表5-2 全向轮战车控制器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| RoboMaster开发板 C 型 | 2 | 用于控制底盘和云台 |
| RoboMaster C620无刷电机调速器 | 6 | 用于控制RoboMaster M3508 直流无刷减速电机 |
| RoboMaster C610无刷电机调速器 | 1 | 用于控制RoboMaster M2006 直流无刷减速电机 |
| Mini PC24V | 2 | 用于视觉接收信息处理 |

**表5-3 全向轮战车其余器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| 分电板 | 5 | 用于对各电机之间供电分配 |
| 分信号板 | 5 | 用于对信号的分配中转 |
| 线材 | 若干 | 用于连接，使信号和供电稳定 |
| DT7 遥控器 | 1 | 用于控制战车 |
| DR16 接收机 | 1 | 配合DT7遥控器使用 |
| USB转TTL模块 | 2 | 单片机与视觉处理系统通信 |
| 24V转12V降压模块 | 2 | 视觉处理系统供电 |
| 线扎 | 若干 | 梳理线路 |
| 热缩管绝缘胶管 | 若干 | 保护线路 |
| 功率计 | 1 | 实时监测底盘功率 |

全向轮战车硬件线路可简化分为底盘和云台两部分，同时具备双YAW轴，战车模块板间通信为 CAN 通信，通过底盘接线板与滑环转PCB板简化线路，完成通信连接。

图示

描述已自动生成

**图5-1 全向轮战车电路接线图**

5.2 **全向轮战车**特殊硬件设计

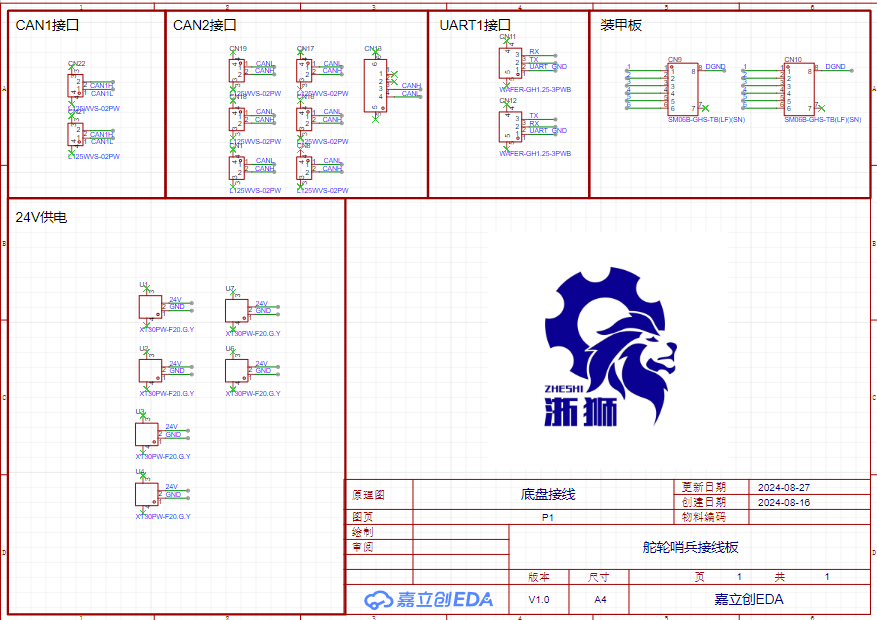
5.2.1 总体方案

整个硬件系统采用模块化设计，集成底盘PCB、滑环转接PCB和超级电容系统，形成一个高效、稳定的智能遥控战车平台。底盘PCB和滑环转接PCB通过CAN总线实现无缝通信，确保数据传输的实时性和准确性。超级电容系统为整个平台提供额外的功率支持，确保在高负载情况下仍能保持稳定的性能。模块化设计不仅便于后续的维护和功能扩展，还提高了系统的可靠性和可维护性。

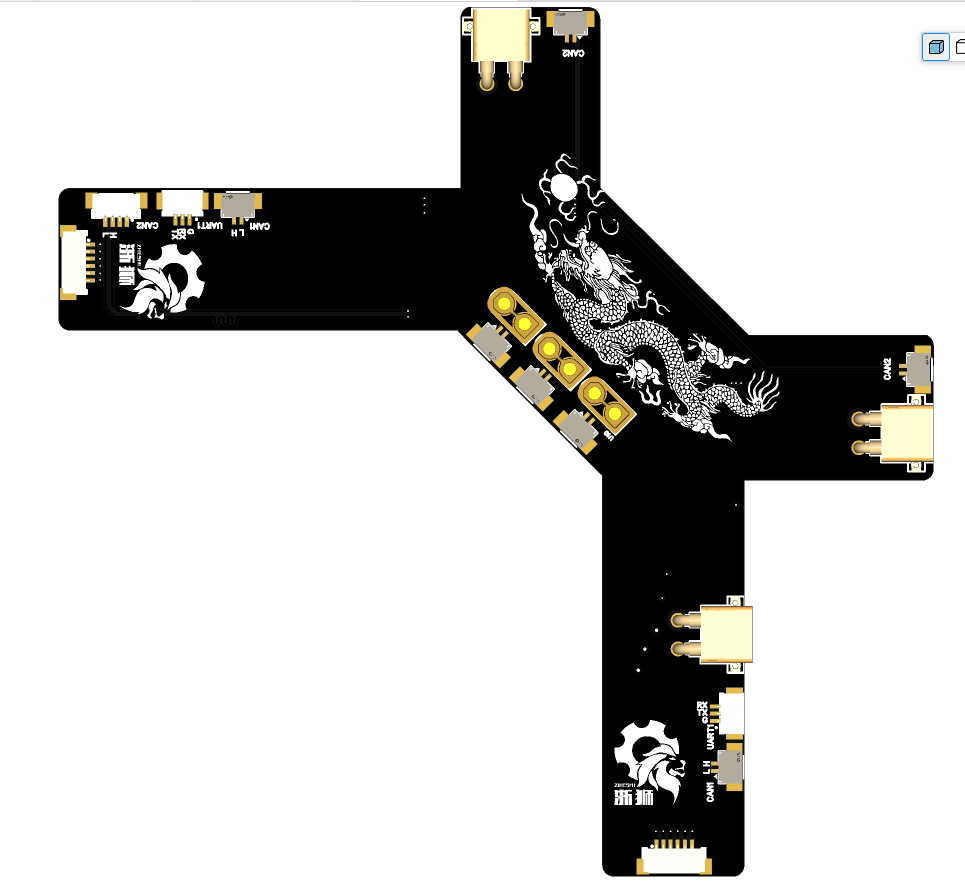
通过合理的电源管理和高效的散热设计，整个系统在复杂环境下仍能保持优异的性能。这种设计不仅满足了当前赛季的需求，还为未来的迭代提供了灵活的扩展空间。

5.2.2底盘接线板PCB设计

底盘PCB采用模块化设计，将电机控制、通讯和信号传输功能模块化，便于后续的维护和功能扩展。为减少电磁干扰，确保信号传输的稳定性，设计中采用了屏蔽层和滤波电路。底盘PCB的额定电压为24V，最大电流为20A，支持CAN总线通信（1Mbps）和UART通信（115200bps）。

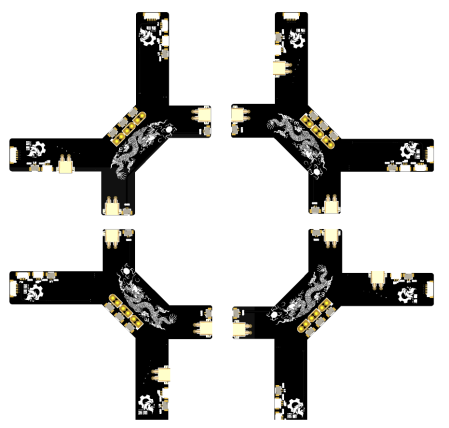
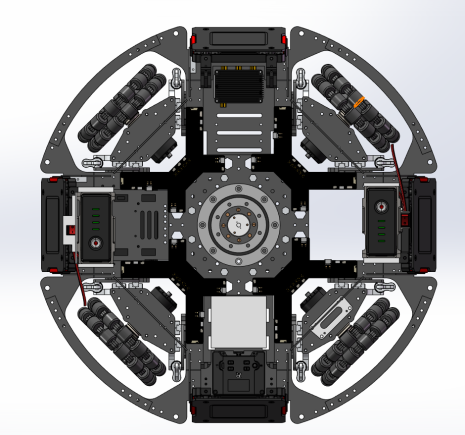


**图5-2 底盘接线板原理图**



**图5-3 底盘接线板模型图**

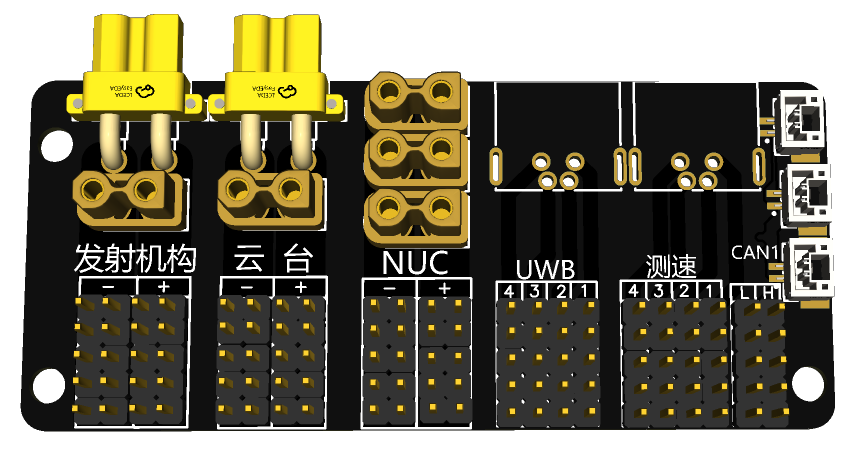
底盘PCB采用井字形布局（图5-3），装配到底盘铝管上，提供稳定的结构支持。其设计集成了电机控制、云台通讯和裁判系统交互功能。通过CAN2接口实现对四组全向轮电机（M3508）的精确控制，确保机器人能够灵活移动。通过CAN1接口与云台控制板进行数据交换，实现云台的精确控制。通过UART1接口与裁判系统进行通信，确保比赛数据的实时传输。

**图5-4 底盘接线板安装效果图**

5.2.3滑环转接PCB设计

滑环转接PCB用于连接底盘和云台，支持360°旋转通信。其主要功能包括通过24线导电滑环实现云台与底盘之间的信号和电源传输，为发射机构电机、Yaw和Pitch电机提供24V供电，为雷达和NUC提供20V供电，并通过航空插头与裁判系统进行供电和通信。滑环转接PCB的额定电压为24V和20V，最大电流为20A，支持CAN总线通信（1Mbps）。



**图5-5 滑环转接PCB模型图**

5.2.4超级电容系统

5.2.4.1总体方案

超级电容（Super Capacitor）是一种重要的能量存储装置，主要应用于机器人的电力系统中。它的核心作用是在功率未满时将多余功率储存起来，从而在需要时短时间内提供大电流放电，以支持机器人执行高功率动作，同时保护电池和优化能量管理。

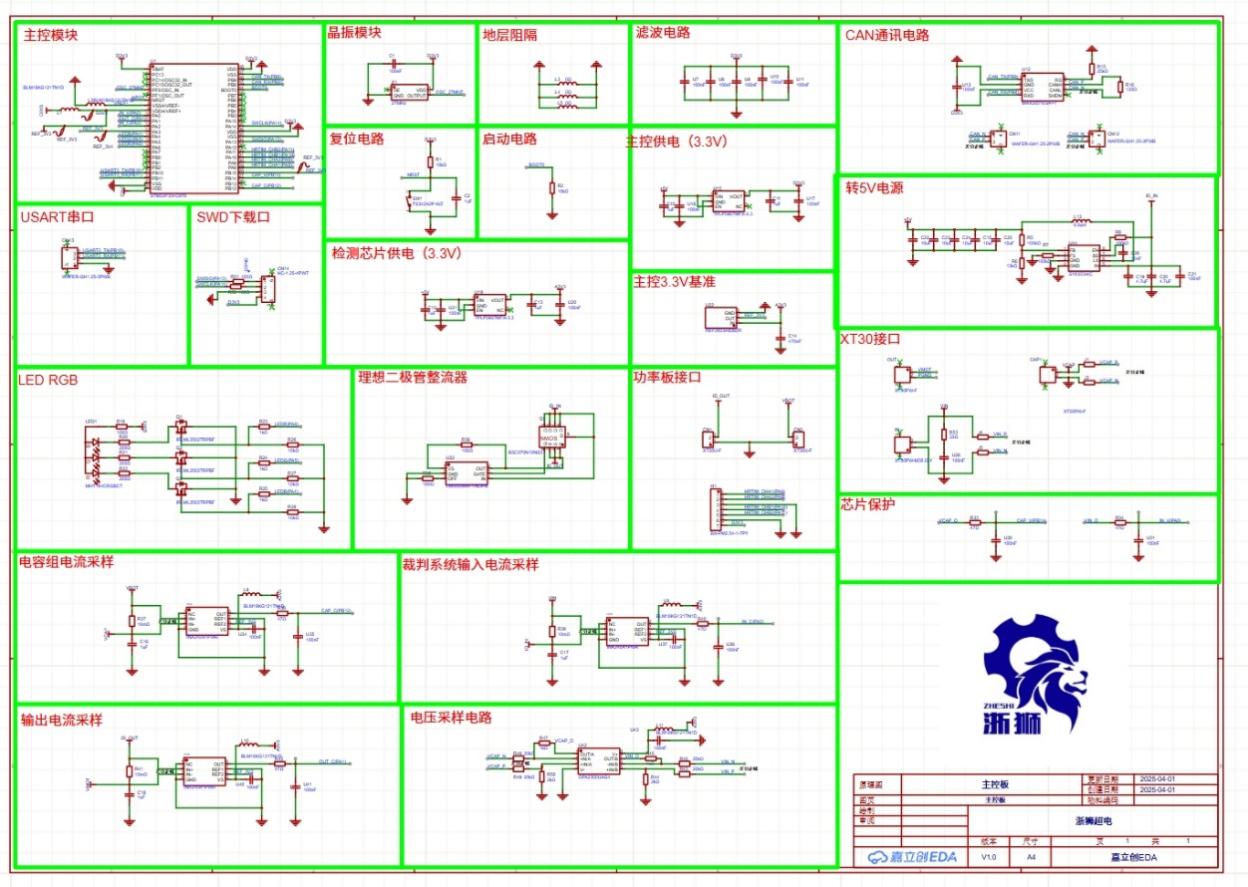
该模块采用双层板与四层板的组合设计方案。两块电路板通过2个MR30连接器（用于大电流传输）和1个7针排针（传输PWM信号）实现互连。具体布局如下：

1. 功率层设计：主要功率器件集中布置在双层板上，利用其简单的层叠结构和较少的元器件数量，实现良好的散热性能；

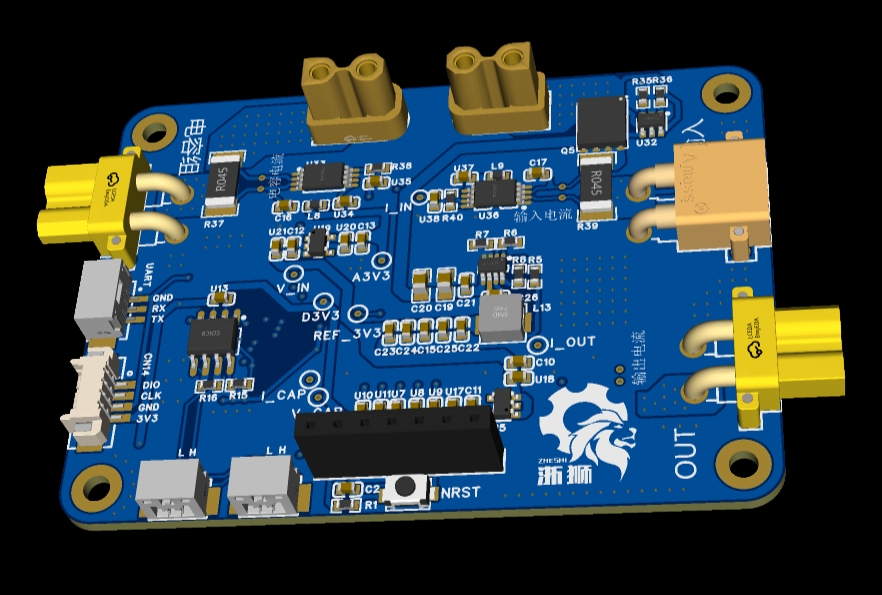
2. 控制层设计：将控制电路和采样电路集成在四层板上，充分发挥多层板的优势，实现更高密度的元器件布局，确保完整的地平面设计，提供更好的信号完整性和抗干扰能力。

这种分层设计方案既优化了功率器件的散热性能，又保证了控制电路的稳定性和集成度，实现了功能与可靠性的最佳平衡。

5.2.4.2控制部分

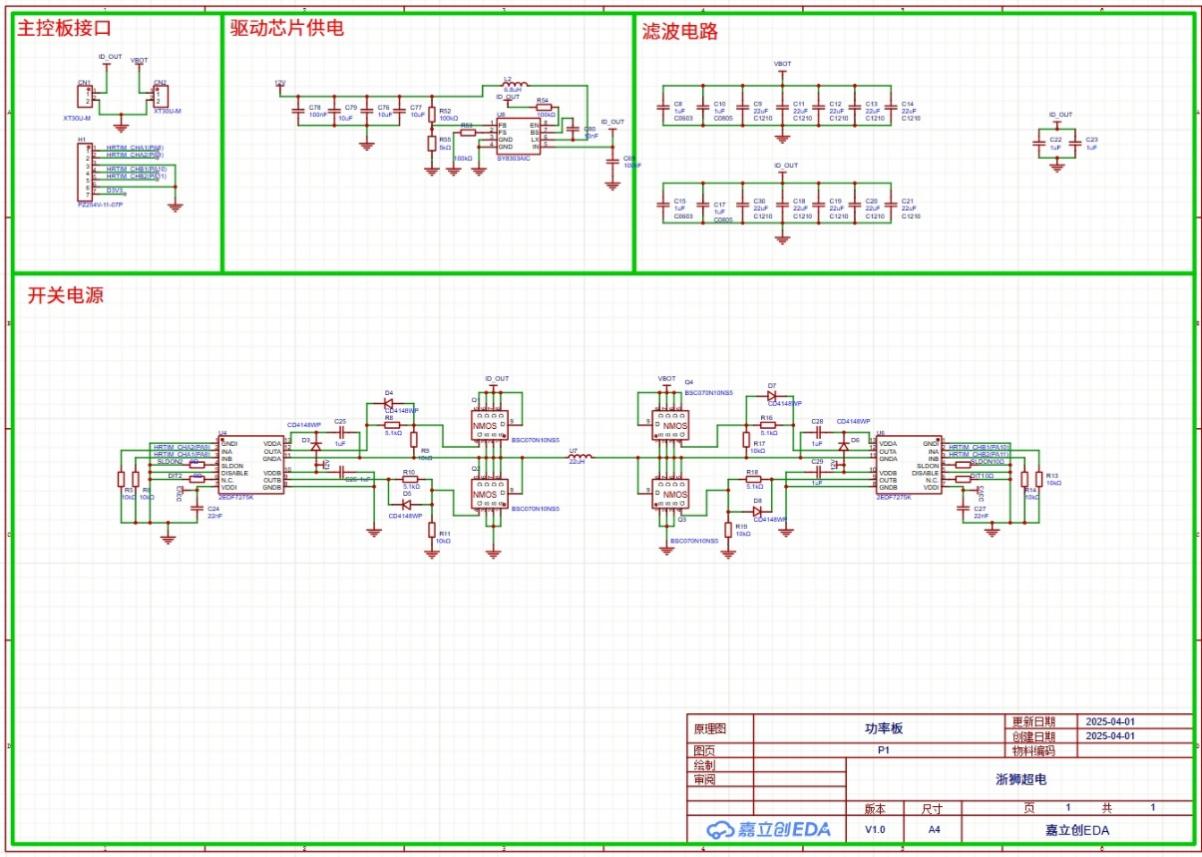


**图5-6 超级电容控制板原理图**

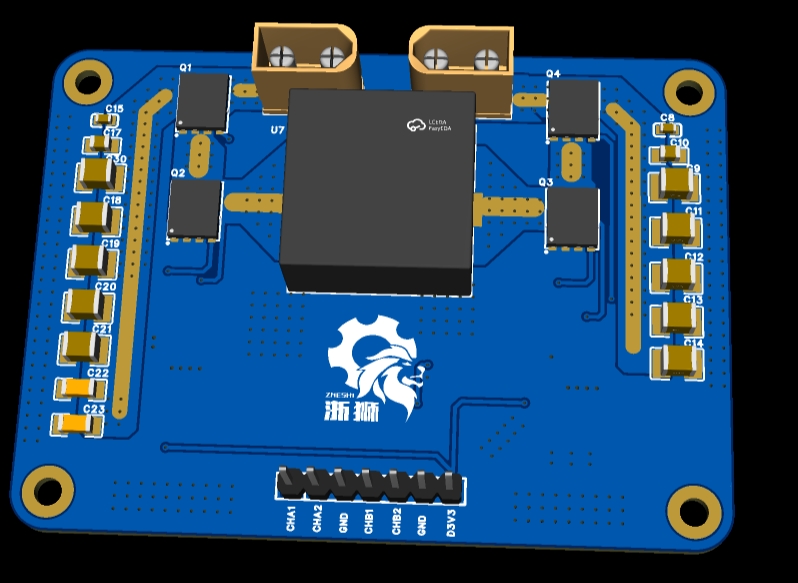


**图5-7 超级电容控制板模型图**

5.2.4.3功率部分



**图5-8 超级电容功率板原理图**



**图5-9 超级电容功率板模型图**

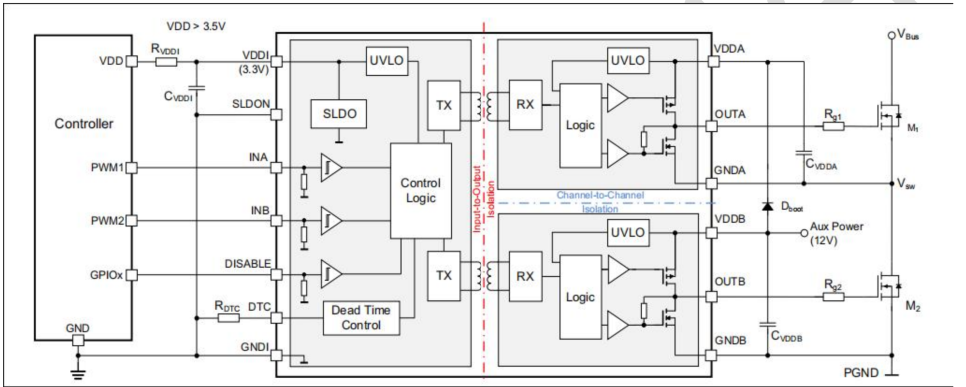
5.2.4.4 关键器件选型

MCU：意法半导体推出的 STM32F334C8T6 微控制器，搭载 Cortex-M4 核心，配备 32 位 CPU 和浮点运算单元（最高主频 72MHz），支持单周期乘法运算及硬件除法 DSP 指令，内置 64KB Flash 存储器和 16KB SRAM。芯片集成 2 个模数转换器（ADC）、3 个数模转换器（DAC）、3 个比较器、1 个运算放大器以及 1 个 4 通道高分辨率定时器（HRTIM）；其功能完全满足设计要求。尽管在运算能力和片上资源方面不及 STM32G474 型号，但该型号供货稳定且具有更高的性价比优势。

CAN收发器：为了实现控制模块与整车中心控制（C板）之间的通信，控制板上搭载了MAX3051 CAN信号收发器，使其通过CAN信号与中心控制板之间实时信息交流。

电流与电压检测方案：为精确监测充放电功率，系统采用STM32内置的16位高速ADC，配合电流检测芯片INA240\_A1PWR和运算放大器OPA2350UA/2K5，实现3路电流信号和2路电压信号的自动循环采集与数据传输。在软件实现上，需配置MPU保护区域、禁用缓存区（cache）以及调整链接脚本（ld文件）以确保数据采集的实时性和可靠性。

MOSFET驱动方案：采用英飞凌2EDF7275K双通道隔离驱动芯片，该器件基于无磁芯变压器(CT)技术，提供基础型(2EDFx)或增强型(2EDSx)输入输出隔离。凭借其37ns超短传输延迟、1.5kVDC/10ms的隔离耐压、强大的驱动电流（±4A峰值）以及优异的共模抑制特性，特别适合高频开关电源系统中高压MOSFET的可靠驱动，有效防止MOS管击穿引发的系统故障。



**图5-10 IC封装图**

高压MOSFET选型：采用英飞凌BSC070N10NS5 N沟道MOSFET搭建H桥，该器件具有100V耐压/80A电流能力，极短的13-24ns死区时间。其PQFN8封装展现出优异的散热性能，在无散热器条件下测试，可确保超级电容系统瞬态工作温度不超过140℃。

驱动电源设计：通过MP2451 DC-DC降压芯片将输入电压转换至12V，为驱动电路提供稳定工作电源。

电感选型：选用伍尔特(WE)74435582200型号的22μH一体成型电感，该器件具有高集成度和优良的电磁特性。

5.2.4.5 相关参数计算

BUCK电感参数计算：定义占空比为 , 其中和为MOS通态和关态时间，那么在BUCK电路中输入与输出的关系如下：

（5-1）

在上臂MOS导通(ton)，电感充能期间，电感两端电压为 , 电感电流线性增加：

（5-2）

那么我们可以得到电感电流增加量：

（5-3）

在上臂MOS关断()，电感释放能量，电感两端电压为，电感电流线性减小，那么我们 得到电感电流减小量：

（5-4）

理想地，BUCK电路稳定工作时，有，并且，其中K 通常取10~20%经验值。

那么电感值可以计算为：

（5-5）

在整个开关周期中电感量都要满足要求，电感要取最大值，我们取D=50%来计算，假设以120W 的功率向电容组充电，且电池供电电压为25V，则电流为4A，那么以输入电压 25V，输出 电流 4.8A，电源纹波均为200mV，开关频率 288kHz为 例，K取20%，计算得：

（5-6）

BOOST电感参数计算 ：定义占空比为，其中和为MOS通态和关态时间，那么在BOOST电路 中输入与输出的关系如下：

（5-7）

在上臂MOS导通()，电感充能期间，电感两端电压为，电感电流线性增加：

（5-8）

那么我们可以得到电感电流增加量：

（5-9）

在上臂MOS关断()，电感释放能量，电感两端电压为，电感电流线性减小，那么我们得到电感电流减小量：

（5-10）

同样理想地，BOOST电路稳定工作时，有

电感值：根据对于的推导，有：

（5-11）

在整个开关周期中电感量都要满足要求，电感要取最大值，我们取D=50%来计算，假设机器人地盘在大多数工况下，功耗在200-300W范围内，且电池供电电压为25-26V，则电流范围主要集中在8-12A范围，那么以输出电压 =25V，输出电流，电源纹波均为200mV，开关频率 288kHz为例，K取10%，计算得到。

可见电路在特定工况下是对称的，两种工况下所需的L值非常相近。为了满足所有工况的要求，L应该取最大值，即，然而市面上无法购买到或的电感器，因此我们最终选择的电感值为。

电容：本方案的滤波电容为多个 50V 陶瓷电容并联，体积小，发热小；总容值为：110uF。

相关参数计算： 输入输出电容参数计算，输入电容的作用主要是保障输入电压的稳定，若输入电容过小，当BUCK 输出重载时，可能 拉低输入电压；输入电容的纹波大，将污染输入电源；所以输入电容要求足够大，在电容上 产生的纹波小于输入电源纹波。

将滤波电容等效为一个理想电容C和一个等效串联电阻(ESR)，那么输入/输出纹波电压计算为：

（5-12）

其中，ΔVC为理想电容两端的纹波电压，ΔVR为ESR两端的电压，为了方便计算，同时减小 体积提高性能，我们采用了单体ESR低于10mΩ的陶瓷电容，多个并联后ESR可忽略不计，则C计算为：

（5-13）

根据前面的参数，计算得出C=41.7μF，在我们的设计中，由于BUCK-BOOST电路需要持续为 电机供电，而电机属于功率会大范围变动的动态负载，BUCK-BOOST电路的工况较为恶劣，因此为了保证稳定性，我们保留150%余量，则为110μF。