



智能遥控战车 设计说明书 ——麦轮

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](#_Toc16614)

[1.1 智能遥控战车机械分析 1](#_Toc12927)

[1.1.1 整体设计方案 1](#_Toc7129)

[1.1.2 底盘设计方案 1](#_Toc16391)

[1.1.2.1 井字型框架 2](#_Toc8893)

[1.1.2.3 内嵌轮轮组 4](#_Toc32015)

[1.1.3 云台设计方案 4](#_Toc22059)

[1.1.4 发射机构设计方案 5](#_Toc1109)

[1.1.4.1 储弹机构 6](#_Toc14412)

[1.1.4.2 拨弹仓 6](#_Toc24628)

[1.2 材料选择 7](#_Toc24452)

[1.3 结构装配 8](#_Toc29999)

[第二部分 嵌入式控制开发 9](#_Toc20295)

[2.1.英雄总体控制 9](#_Toc11748)

[2.1.1软件设计 9](#_Toc20693)

[2.1.2系统架构 9](#_Toc23384)

[2.1.3控制框架 10](#_Toc4793)

[2.1.4功能概述 10](#_Toc5384)

[2.1.5底盘任务 11](#_Toc19761)

[2.1.6云台任务 12](#_Toc32293)

[2.1.7射击任务 13](#_Toc23278)

[2.1.8控制参数及要求 14](#_Toc16852)

[2.1.8.1控制参数表 14](#_Toc28882)

[2.1.8.2控制要求 14](#_Toc26233)

[2.2底盘部分 14](#_Toc13456)

[2.2.1底盘模式判断 14](#_Toc22612)

[2.2.2麦轮速度解算 16](#_Toc12637)

[2.2.3底盘电机通讯控制 18](#_Toc19955)

[2.2.4底盘硬件离线检测初始化 20](#_Toc25360)

[2.2.5底盘通信线程初始化 20](#_Toc32072)

[2.3云台部分 21](#_Toc4966)

[2.3.1云台模式判断 21](#_Toc3686)

[2.3.2云台控制算法 22](#_Toc25159)

[2.3.3云台电机通讯控制 23](#_Toc24882)

[2.3.4自瞄模式 25](#_Toc22839)

[2.3.4.1自瞄数据获取 25](#_Toc9610)

[2.3.4.2视觉数据结构体 26](#_Toc11102)

[2.3.4.3视觉数据更新 27](#_Toc429)

[2.3.5云台硬件离线检测 27](#_Toc3789)

[2.3.6云台通信线程初始化 28](#_Toc8433)

[2.3.7视觉交互数据回调函数 29](#_Toc16619)

[2.4.射击部分 30](#_Toc5376)

[2.4.1 射击任务 30](#_Toc5924)

[2.4.2 射击判断 31](#_Toc1596)

[2.4.2.1 射击判断（遥控器控制段） 31](#_Toc18012)

[2.4.2.2 射击判断（键鼠控制段） 33](#_Toc21491)

[第三部分 视觉系统设计 35](#_Toc7861)

[3.1背景与目标 35](#_Toc2958)

[3.2视觉自动瞄准算法设计 35](#_Toc9881)

[3.3设计方案 36](#_Toc5602)

[3.3.1目标检测 36](#_Toc5948)

[3.3.2目标运动解算 42](#_Toc13377)

[3.3.3 火控策略 50](#_Toc3557)

[第四部分 硬件设计 53](#_Toc29455)

[4.1 英雄战车基础硬件设计 53](#_Toc8382)

[4.1.1 英雄战车器件 53](#_Toc9688)

[4.2 英雄战车战车特殊硬件设计 55](#_Toc15109)

[4.2.1 PITCH达妙电机 55](#_Toc24892)

[4.2.1.1 PITCH达妙电机特点 55](#_Toc971)

[4.2.1.2 PITCH达妙电机性能参数 55](#_Toc18741)

[4.2.2 超级电容系统 56](#_Toc27557)

[4.2.2.1总体方案 56](#_Toc32378)

[4.2.2.2控制部分 56](#_Toc8449)

[4.2.2.3功率部分 57](#_Toc3085)

[4.2.2.4 关键器件选型 58](#_Toc26615)

[4.2.2.5 相关参数计算 59](#_Toc32316)

**第一部分 机械结构设计**

1.1 智能遥控战车机械分析

1.1.1 整体设计方案

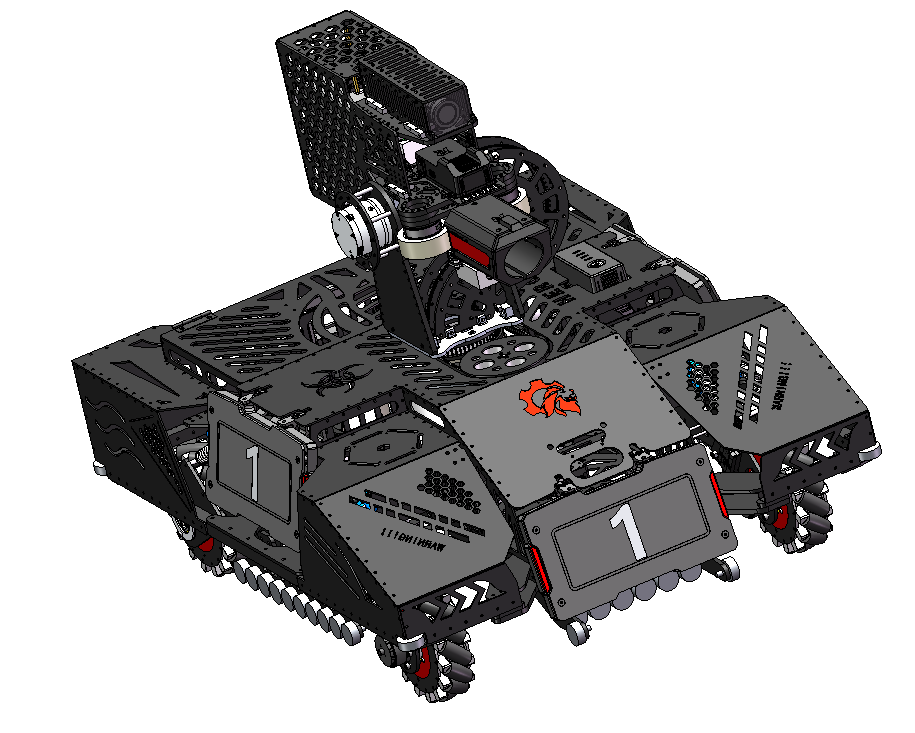
结合新工科人才培养试点班《智能遥控战车》项目制教学大纲的任务要求，以开发研制智能遥控战车为项目目标，筛选出机械组对智能遥控战车（如图1-1）的整体设计要求如下：

（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 30kg；

（3）功能要求：能实现自由移动并稳定发射 42mm 弹丸；

（4）设计制作周期：12 个月（组队）。

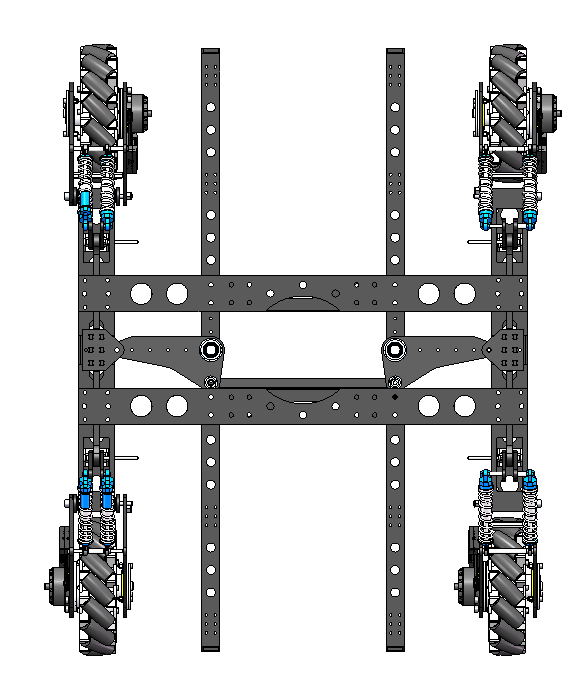


**图1-1 英雄机器人**

本项目旨在设计研发性能可靠优异的全方向自主移动机器人，对于机器人运作环境需求进行分析后，对英雄机器人的各项功能做出了功能需求与技术指标。在保证机器人稳定运行的前提下，扩大其的弹舱容量，提升最大弹容量；采用自适应悬挂机构，提高地形适应性；采用侧供弹机构，优化供弹性能。

1.1.2 底盘设计方案

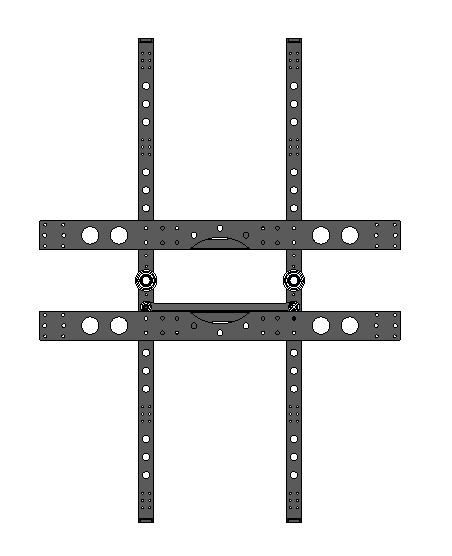
优化底盘结构，采用较为传统的铝合金方管框架井字形结构，确保机器人底盘拥有稳定的性能与紧凑的结构，同时也可确保机器人强度。设计时镂空铝方管为机器人自适应悬挂机构预留足够的空间，各个铝方管之间依靠碳纤维板，抗拉强度可达3500Mpa，由于碳纤维的比强度可达2000Mpa/（g/cm³），确保其重量不会因为扩大的体型严重超标的同时，具备可靠的强度。底盘框架及轮组悬挂如图1-2。



**图1-2 英雄机器人底盘**

1.1.2.1 井字型框架

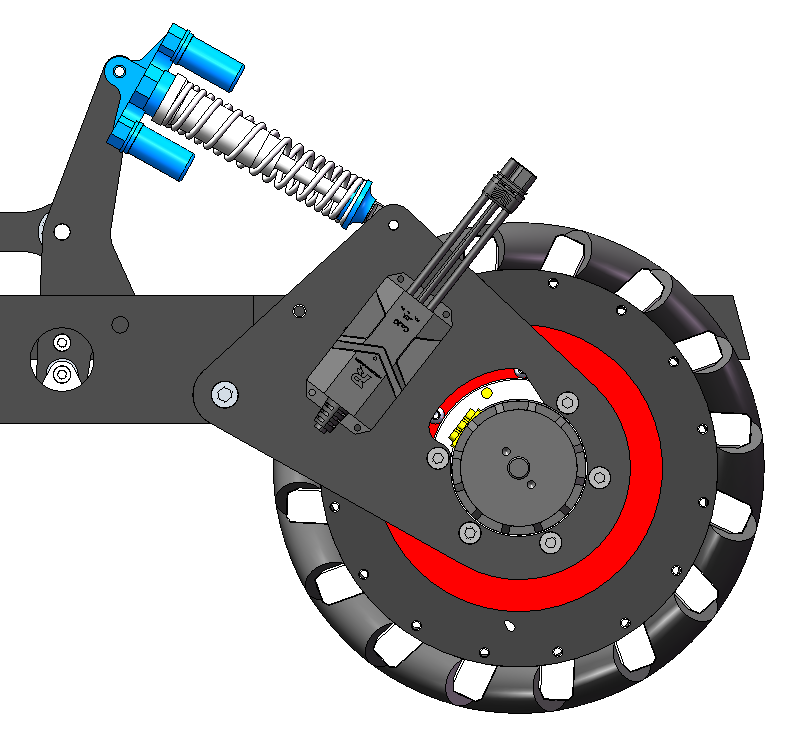
底盘大体框架为井字形铝合金方管，确保其具备优越的稳定性与较高的强度，同时，具备较多的空间安装电子元器件与连接用的板材。依靠井字形框架设计整车结构，可以更加清晰的分配剩余零件，使结构清晰明了，且更加紧凑。如图1-3所示：



**图 1-3 井字型框架**

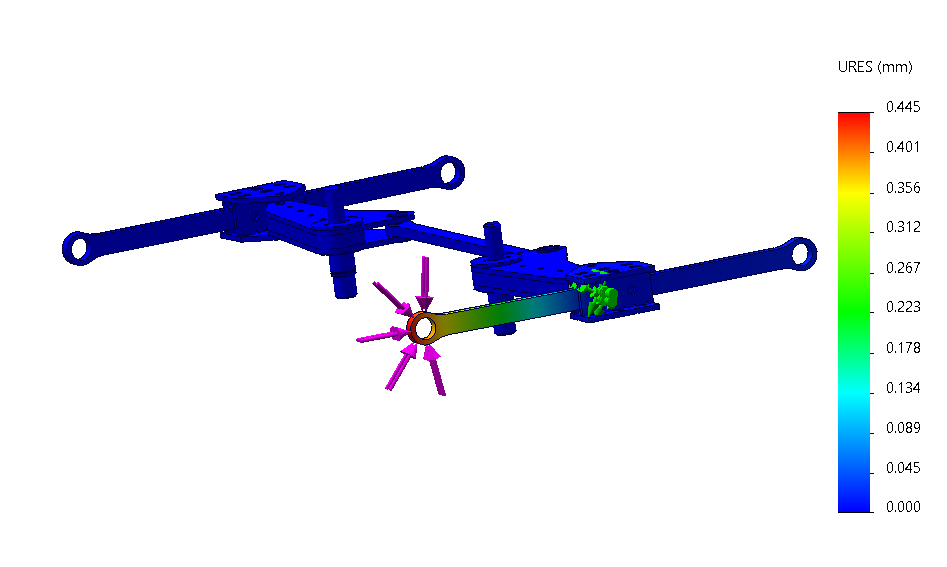
**1.1.2.2 避震悬挂**

在过去纵向避震的基础上进行改良，采用与车轮运动方向相切的斜向避震机构，能有效减少车辆在崎岖路面行驶时的冲击力。当车轮遇到障碍物时，避震器能够沿着与轮组相切的方向进行缓冲，分散冲击力，使车辆行驶更加平稳。同时使避震器的缓冲方向与车轮运动方向一致，提高了缓冲效率。避震器能够更精准地吸收震动，减少能量损失，从而延长避震器的使用寿命。并且可以在一定程度上提高车体的抓地性能。如图1-4所示：



**图1-4 斜向避震**

同时依靠由玻纤板构成的连杆机构连接轮组悬挂，使其具备一定的自适应功能，使轮组可以在同步实现上下40°的转动，同时对于其强度进行基于SOLIDWORKS Simulation的应力分析，如图1-5所示：



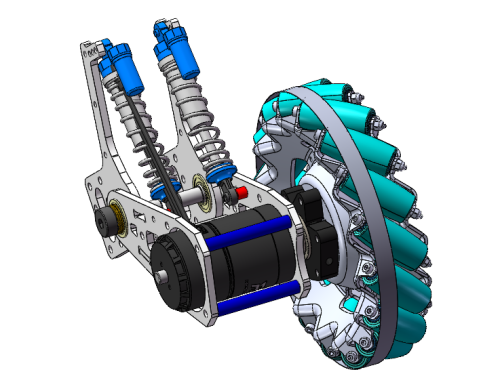
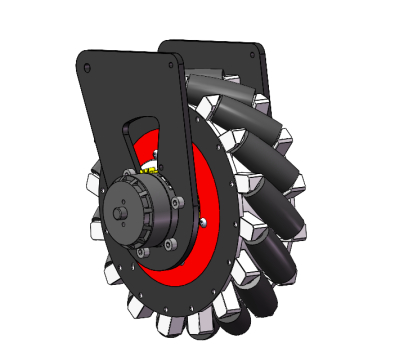
**图1-5 自适应悬挂静应力分析**

根据线应变公式 *ε*=*L*0/Δ*L*​以及高强玻璃纤维断裂时的型变量约为1%，计算出该连杆断裂时最大型变量约为0.8mm，根据仿真测试，其最大能承受10000N左右的冲击力，排除意外情况考虑机器人坠落或是直接冲撞。

根据动量守恒定理，F·t=M·V，其中已知M = 30kg，避震弹簧计算为*mx*¨+*kx*=*F*(*t*)，其中 *m* 为质量，*x*¨ 为加速度，*F*(*t*) 为外力。经过对避震器的选型并设计合适的切向角度之后，可以计算出该强度可以保证车体从约5m高处坠落，或是以20m/s的速度撞击墙面，考虑到该强度以远超轮组所能承受强度，最终确定改版设计方案。

1.1.2.3 内嵌轮轮组

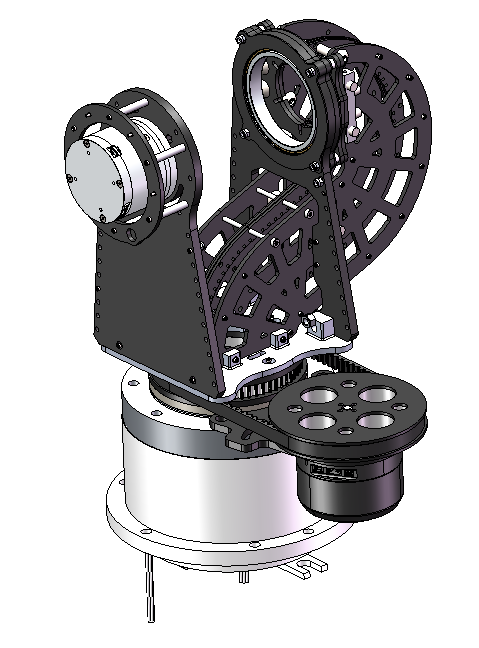
该版轮组将电机嵌入置麦克纳姆轮当中，通过镂空麦克纳姆轮的方式既达到了减重的效果又完成了节约空间的功能，方便为后续弹舱的设计腾出足够的空间（两侧各节约50mm宽度）。同时，由于轮组完全与悬挂的铝方连接，使其结构更加紧凑。与过去的轮组对比图如图1-6所示：



**图1-6 内嵌轮与传统麦轮**

1.1.3 云台设计方案

云台支持360°旋转，有YAW与PITCH两个转动方向，其中YAW轴由大疆GM6020电机驱动，而PITCH轴由达妙的4310电机驱动。其中GM6020具有较高的转动速度以及较高的控制精度，适合云台精准快速的定位，而达妙4310电机的大扭矩可以轻松的带动装有小电脑的云台，保证PICTH轴稳定的运行。云台结构如图1-7所示。



**图1-7 英雄机器人360°云台**

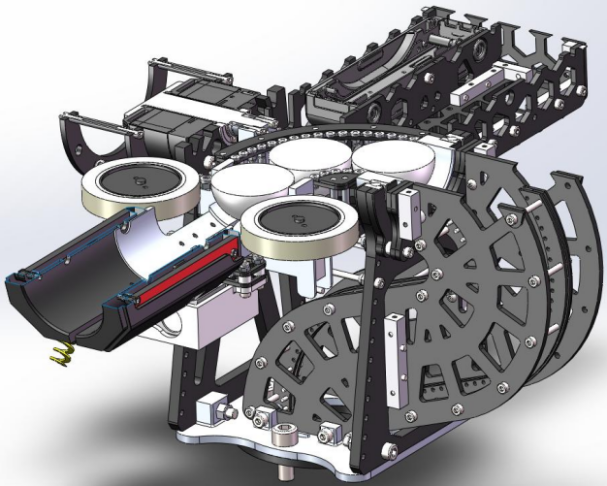
该云台YAW轴传动依靠电机带动M5 L150的同步带，与大多数电机直驱不同，该机构在确保传动可靠的前提下，为弹链穿过云台直达发射机构提供了空间，同时在云台转动处加装回转体轴承确保其转动效率以及反应速度。其PITCH轴则是采用了传统的电机直驱机构，并在云台PITCH轴转动处加装了薄壁轴承与法兰轴承，确保该机构转动足够顺畅。

考虑到机器人的日常维护与运作时的反应能力，PITCH轴采用的电机直驱具备高效率与高精度的特点，尽管较难维护，但优点带来的正面影响远超缺点带来的负面作用，遂采用电机直驱作为云台PITCH轴方案。

**1.1.4 发射机构设计方案**

由于该战车发射弹丸为42mm丁腈橡胶弹丸，其重量与体积较大，考虑到发射物物理特性，以前的上供弹机构已经不再适用于该机器人的制作需求，遂采用下供弹机构，以达到减轻云台重力减少PITCH轴阻力的效果。

该发射机构电机为去除减速箱的M3508电机，通过摩擦轮之间合适的间距，挤压推入摩擦轮之中的弹丸，进行发射，该发射机构（图1-8）采用半嵌入式枪管，在确保安装便利、连接紧密的前提下，有效提高枪管与发射器之间的同心度。其摩擦轮为包胶铝件，通过测试与计算，取到了包胶最适宜的硬度55°。

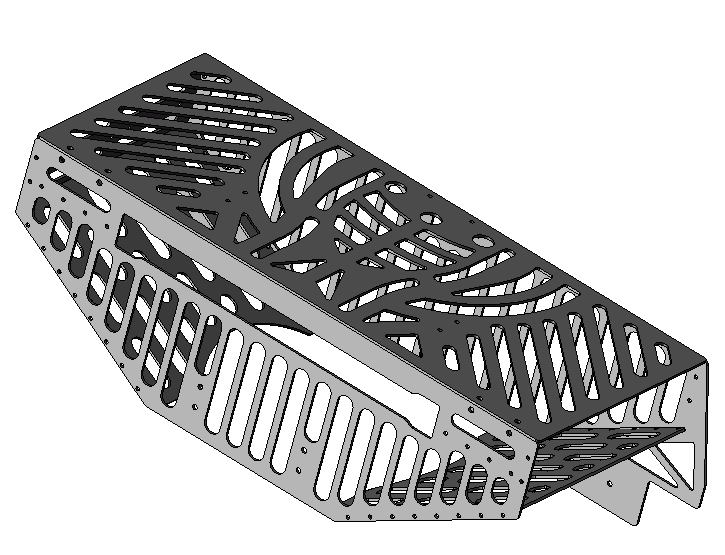


**图1-8 英雄机器人发射机构剖面图**

该发射机构以储存弹丸与发射弹丸为核心功能，其中弹丸被储存于弹链以及弹舱两个机构中，弹舱质量较大，而弹链与云台结构为一体化设计，该设计使其整体重心位于云台中心处。发射弹丸则分为拨弹与挤压发射两部分，其中拨弹机构位于弹舱底部，将弹链从弹舱供入弹链，而发射部分位于云台上方，将弹丸加速至16m/s发射。所以，该发射机构整体部分被设置为三个主要部分：拨弹机构、储弹机构、发射机构。

**1.1.4.1 储弹机构**

储弹机构的设计影响到拨弹机构的设计，其种类大体可被分为两种，一种是位于发射机构上方的上供弹机构，一种是位于发射机构下方的下供弹机构。两者最大的区别在于，上供弹机构以弹丸自身重量下坠，拨弹机构仅仅起到输送作用，对拨弹电机的力矩需求较小，而下供弹机构则需要拨弹电机克服弹丸自身重力，将弹丸向上拨动至发射机构，对电机的力矩需求较高。而上供弹机构由于位于云台上方，对云台自身的重量影响较大，带动PITCH轴转动所需的力矩也相对应的提升。下供弹将弹舱安装于底盘，可以极大程度的减少云台转动时产生的阻力。如图1-9所示，弹舱为一个上底为754mm，下底171.75mm，竖直边长59mm，总高148.5mm，宽193mm的集合体，其最大可容纳150枚42mm大弹丸（包括弹链）。

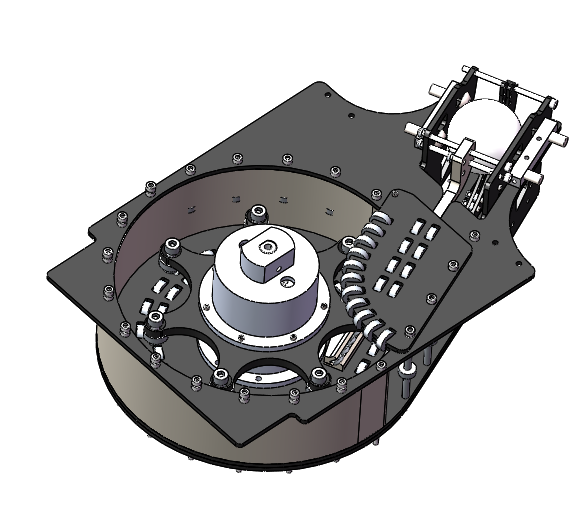


**图 1-9 储弹机构**

**1.1.4.2 拨弹仓**

拨弹仓最为重要的机构为拨弹盘，该机构拥有两个较为重要的指标，其一为单位时间内的供弹量，该数据被称为射频；其二为整体结构的可靠性，为确保其稳定性，通常拨弹盘内会装有大量轴承确保球体滚动顺滑，如果拨弹不够稳定，可能出现卡弹与发射延迟较高的现象，而发射延迟较高的现象较为普遍，通常是由于弹链整体阻力较大，电机无法第一时间转动导致。拨弹盘的运作方式为通过电机转动带动拨叉转动，将弹丸以一个较低的初速度供入弹链，整个弹链内部的弹丸相互推动，最终将离摩擦轮最近的一枚弹丸推入摩擦轮，使其发射。

整个过程可简化为由拨弹盘的曲面运动，转变为弹链内部的曲线传动，最后进入发射机构本体的直线运动，最终化作弹丸的落点，该运动需要稳定性与一定的规律。该机器人拨弹机构如图1-10所示：

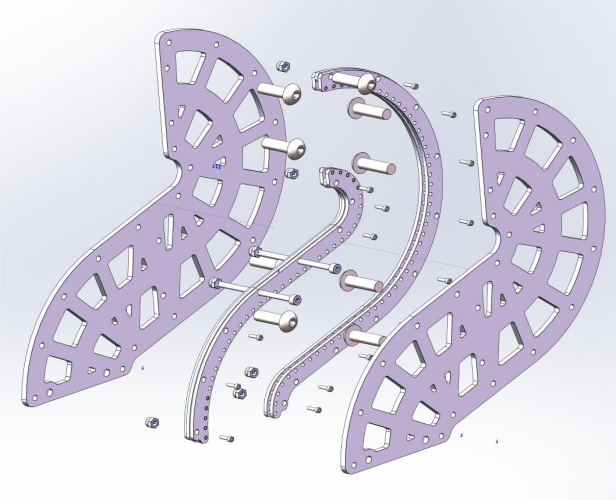


**图 1-10 拨弹机构**

设计时还需要考虑拨叉的高度与厚度，高度不宜高于弹丸，最佳位置为弹丸中心，厚度则不宜过薄，通常为4-5mm左右，改版拨叉高度离底面19mm，厚度为4mm，满足设计需求其各个角的末端装有轴承，确保其与弹丸接触的最终位置为滑动摩擦而非滚动摩擦。而拨弹仓的内壁则为树脂3D打印材料，该材质摩擦系数极低，有效减少拨弹仓内摩擦力。

该机器人发射机构创新点主要由两个方向，其一是预留了足够的空间安装视觉系统，可通过算法进行自动瞄准锁定；其二则是侧供弹机构，减少云台头部空间，为安装小电脑保留充足的空间位置。

侧供弹弹链区别于过去的鹅颈式弹链，无论是各个角度，弹丸整体所需位移量保持不变，设计时，采用了由小入大原则，其弹链宽度逐步拓宽，确保弹丸不会因为直径过大卡在弹链内部。如图1-11所示为弹链颈部爆炸图：



**图 1-11 侧供弹链颈部爆炸图**

**1.2 材料选择**

整车材料以玻璃纤维与碳纤维为主，由铝合金方管搭成整体框架，同时有少部分强度需求较低的材料为3D打印件，尽可能确保其结构可靠，并且极大程度减轻重量。

其外壳则以玻璃纤维为主，通过镂空等形式进行加工减少其重量，同时增加花纹确保外形美观，外壳前端参考坦克正面装甲，具有一定的倾斜角，可以弹开敌方弹丸的正面冲击，即使仅有2mm的厚度也足以确保其总体结构的强度，并且其玻璃纤维的材料价格较低，尽可能降低成本确保尽可能将经费花在刀刃上。

**1.3 结构装配**

英雄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，也可以在机器人运作后方便对损坏和磨损的零件进行更换。同时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。

在一些不能使用导体的地方，如电路板的安装，则采用尼龙螺柱的螺母的连接方式。另外本项目使用了大量标准件——四孔直角连接件（图 1-12），方便机器人壳体的拼接。



**图 1-12 四孔直角连接件**

**第二部分 嵌入式控制开发**

2.1.英雄总体控制

2.1.1软件设计

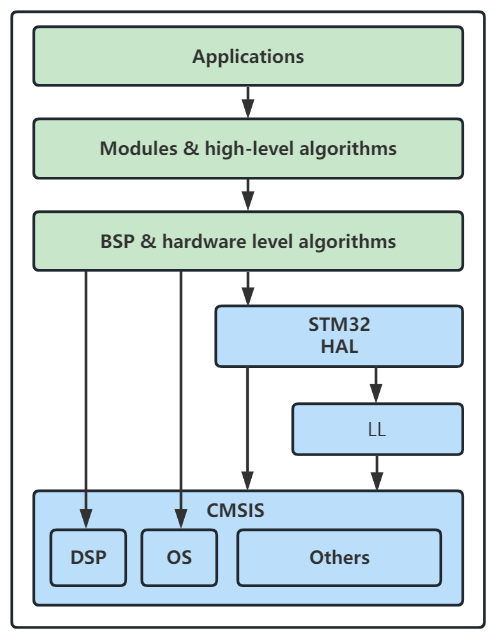
最初项目采用的开发工具包括 Keil uVision5、STM32CubeMX、STM32 Devices2022.01 和 MDK-ARM 5，如今已升级为以 ARM-GNU 工具链结合 VSCode、Ozone&Systemviewer、FreeMaster 及 OpenOCD 为核心的现代化开发体系。该开发流程配套有详尽的教程说明，在现代工具的支持下，能够显著提升电控系统在开发和调试环节的效率。

2.1.2系统架构

整个系统框架分为三大层级：BSP、Components 和 Application。采用的是结构化分层设计理念，其中上层模块会组合并调用下层模块的功能，通过模块嵌套构建出完整的功能体系。顶层的 Application 层之间则通过发布-订阅（pub-sub）机制进行通信，实现了解耦合，避免模块间的直接依赖与互相包含，使得代码维护与拓展更加清晰高效。

我们的目标是通过 BSP 层对底层硬件进行封装抽象，使开发者在编写 Components 层模块时，无需关注具体硬件的运行细节；而 Components 层进一步对外部设备进行模块化抽象，使 Application 层可以完全以硬件无关的方式进行逻辑编写。只需参考 Components 的接口说明文档，即可快速开发上层应用功能。BSP 与 Components 的设计初衷，是成为系统中实现软硬件隔离、提高复用性的中间件核心。

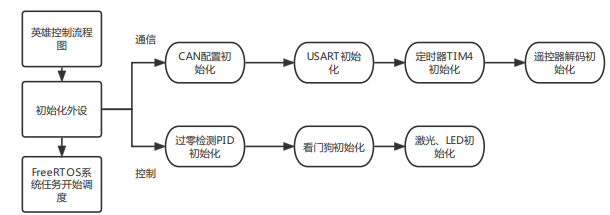
以英雄机器人中的软件架构为例，Application 层主要由 chassis 与 gimbal 两个核心模块构成，同时包含 Cmd 和 Shoot 两个功能任务。每个任务模块分别代表机器人上特定的功能单元。Cmd 任务模块负责接收来自遥控器、上位机或传感器的控制信号，解析为对应执行器（如电机、舵机）的操作指令，并将这些操作信息广播发布。chassis 和 gimbal 等模块中的执行单元则作为订阅者接收这些消息，并调用其内部的功能模块接口以完成具体动作的执行。

以 chassis 任务为例，其内部集成了 4 个底盘电机模块以及一个 shoot\_task 射击模块。当接收到来自 Cmd 任务的控制指令，例如要求以 1m/s 的速度向前运动时，chassis 会根据底盘的具体类型（如舵轮、麦克纳姆轮、全向轮或平衡式底盘）及其对应的动力学和运动学解算算法，计算出每个电机所需的目标转速。随后，chassis 将这些目标值通过 motor 组件的接口下发给对应电机模块。每个电机模块内部还包含 PID 控制器，并读取自身传感器（电流、速度、角度）数据以计算出实际需要输出的电流值。若该电机采用 CAN 总线与电调通信，则通过集成的 CAN\_TxMsg\_t（由 bsp\_can 提供）组织控制报文并下发给电调，实现闭环控制。此过程中，模块间的层级关系为 chassis → motor → bsp\_can。

**图2-1 整体系统架构示意图**

2.1.3控制框架

系统首先对各类硬件资源进行初始化配置，包括 CAN 总线通信、串口通信（USART）、编码器读取、定时器设置以及看门狗等功能模块的启动。随后，根据功能需求，将整个控制系统划分为多个独立任务模块，如云台控制、底盘控制、射击机构控制、遥控器模式选择与掉线检测等任务。所有任务基于 FreeRTOS 的优先级调度机制循环运行。同时，各任务借助 CAN 或串口等中断接口实时获取外部数据，并及时更新控制算法中的状态，最终输出指令至电机等执行部件，实现系统的完整闭环控制。此外，为增强系统的稳定性和抗干扰能力，还引入了滤波算法优化信号质量、减少噪声影响。



**图2-2 整体控制流程图**

2.1.4功能概述

（**1）底盘功率管理功能：**通过采集裁判系统提供的功率数据，动态调节英雄机器人的功率输出，确保系统运行不超过规定阈值以避免触发保护机制。在不平坦地形中，适当降低电机转速以维持输出扭矩，提高通过能力。

**（2）射击热控策略：**根据裁判系统实时反馈的枪口热量与冷却状态，对射击行为进行限制，防止连续发射导致系统因热量超限而进入停机状态。

**（3）云台模糊 PID 控制：**在传统 PID 控制的基础上引入模糊逻辑控制算法，使云台在各个角度区域内具备更强的稳定性。提升了快速响应性能并有效降低了静止状态下的角度误差，确保机器人在瞄准和射击任务中的稳定性。

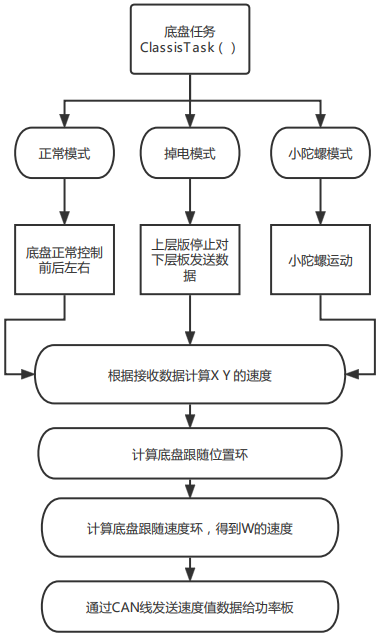
**（4）多模式底盘控制与超级电容应用：**底盘控制支持多种模式切换，可通过键盘实现如对角线前进、底盘随动、原地旋转等不同控制方式，使操控手能灵活应对复杂战局。同时引入超级电容模块，可在短时内释放大量功率而不违反裁判系统限制，从而显著增强机器人在高强度机动过程中的表现。

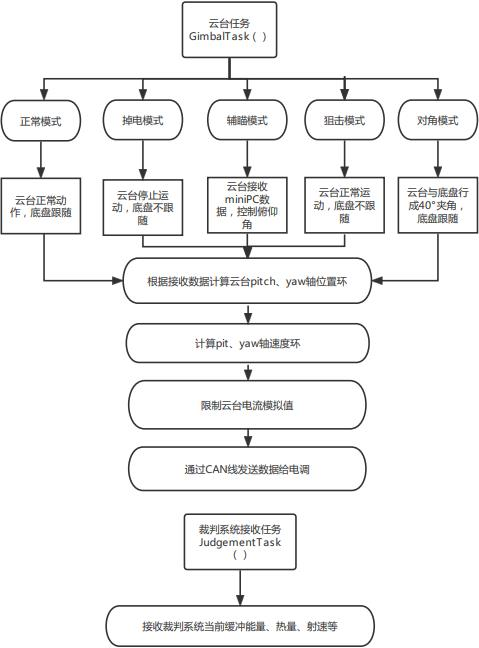
**表2-1 英雄机器人整体任务说明**

|  |  |  |
| --- | --- | --- |
| 参数 | 描述 | 频率(HZ) |
| START\_TASK\_PERIOD | 程序启动任务，用于区分云台和底盘控制板并执行对应任务初始化函数 | 单次 |
| CHASSIS\_TASK\_PERIOD | 英雄车底盘任务，包括麦轮基本运动和速度切换 | 200 |
| GIMBAL\_TASK\_PERIOD | 云台控制任务，包括 yaw，pitch 和云台抬升的控制 | 500 |
| IMU\_TASK\_PERIOD | 姿态解算任务，通过IMU获取Z姿态数据，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 | 200 |
| SHOOT\_TASK\_PERIOD | 发射任务，包括拨弹轮及摩擦轮的控制及热量检测 | 250 |
| COMM\_TASK\_PERIOD | 通讯任务，用于桥接云台及底盘两板，同时承担与视觉及各模块如电机等的数据交互 | 1000 |
| CONSOLE\_TASK\_PERIOD | 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 | 200 |

2.1.5底盘任务

该任务负责处理底盘的运行状态，主要控制模式包括“底盘无力”和“跟随云台”两种。流程为：首先完成底盘的初始化，包括 PID 参数设定与相关状态赋值。随后进入循环执行阶段，更新当前底盘数据（如电机速度、YAW 轴云台角度）。若当前模式为“无力”，则直接将电流设为 0；若处于控制模式下，则根据遥控器或键盘输入，计算期望的 XY 方向速度或底盘角速度。根据底盘类型进行运动解算，计算出四个底盘电机的目标转速，并通过速度环 PID 算法得出期望电流。最终依据超级电容状态和裁判系统反馈功率，对期望电流进行调整，完成功率限制下的闭环运动控制。

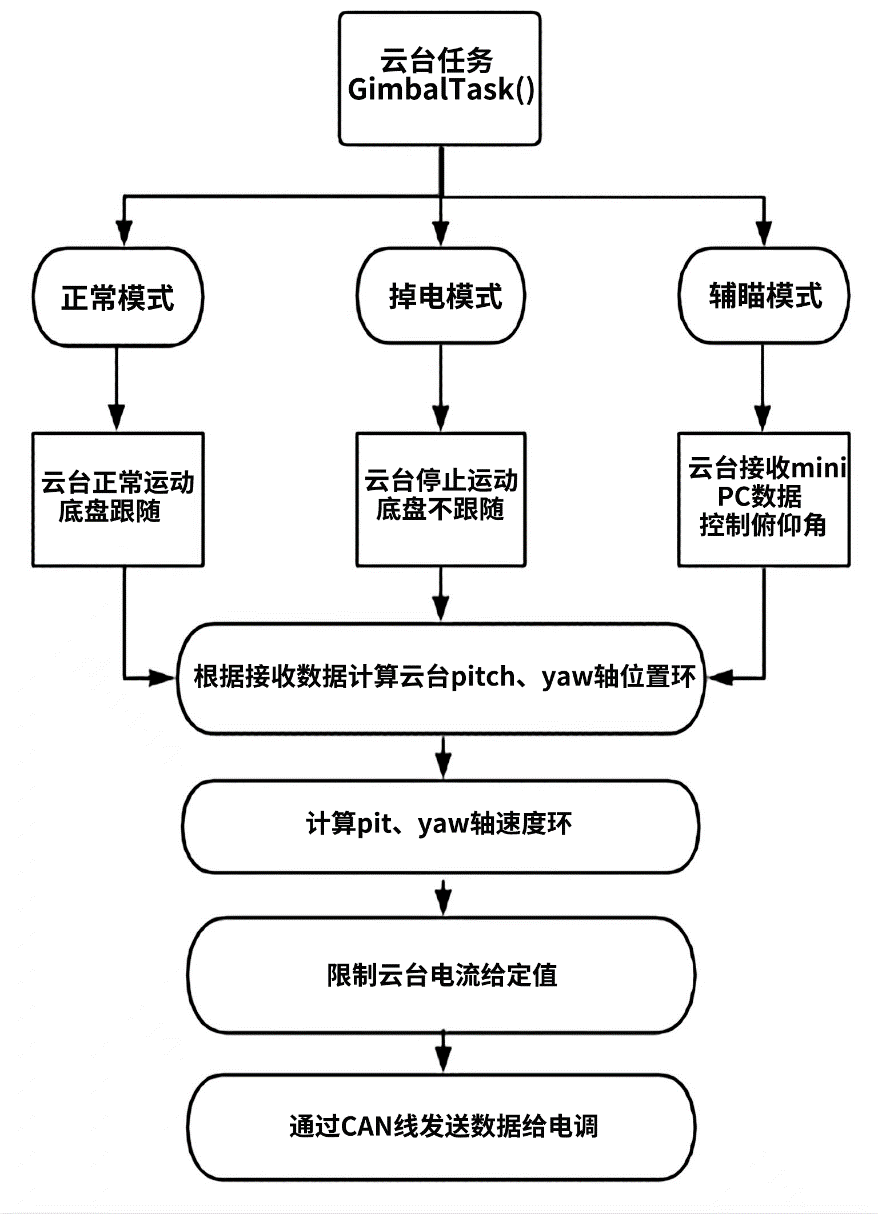




**图2-3 底盘任务控制流程图**

2.1.6云台任务

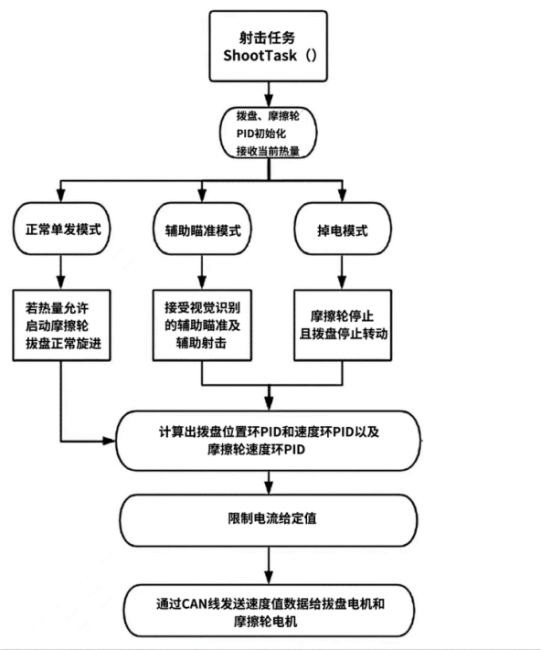
此任务主要负责云台的角度与角速度控制。初始阶段进行参数设定与 PID 控制器初始化，随后进入主循环。任务周期内，实时更新云台当前角度和角速度状态，并根据控制输入进行判断：若检测到鼠标或遥控拨杆信号，则直接将其转换为角速度指令并跳过角度环控制；若无主动控制信号，则默认维持前一时刻角度，通过角度 PID 计算期望角速度。最终依据目标角速度执行速度环 PID 计算，输出相应电流至云台电机，实现精准控制。



**图2-4 云台任务控制流程图**

2.1.7射击任务

此任务涵盖摩擦轮速度调节及拨弹盘运行控制。系统会首先读取裁判系统提供的弹丸速度信息，以设定摩擦轮的目标转速。任务初始化阶段会对 PID 参数进行配置。若操控为单发模式，则拨弹盘使用位置 PID 控制进行单步推进；若处于连发模式，则以固定摩擦轮转速持续射击。不论何种模式，最后均会使用速度环 PID 算法计算电机所需电流值，以确保射击动作的精准性与连贯性。



**图2-5 射击任务控制流程图**

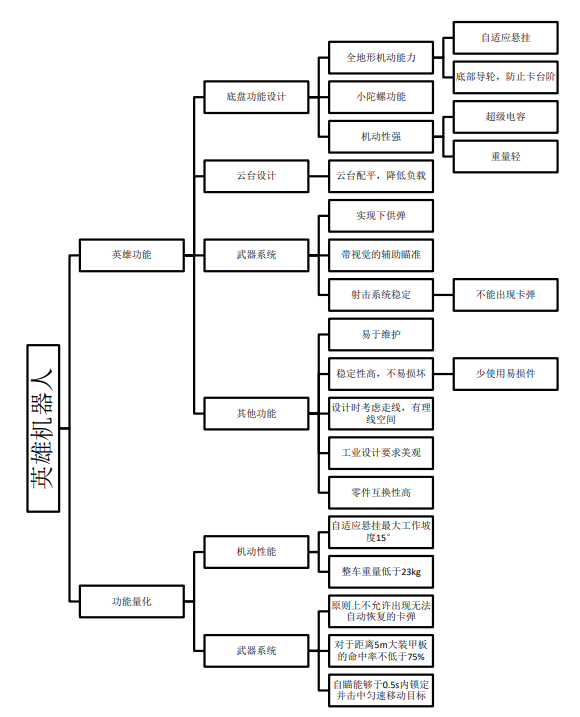
2.1.8控制参数及要求

2.1.8.1控制参数表

**表2-2 英雄机器人核心参数**

|  |  |
| --- | --- |
| 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 | GM6020×2：云台Pitch与Yaw轴驱动  M3508×7：底盘驱动、拨弹电机、摩擦轮电机 |

2.1.8.2控制要求



**图2-6 英雄机器人功能定义**

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麦轮速度解算

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);//YAW轴云台偏转正弦角（跟随云台模式）

    cos\_yaw = arm\_cos\_f32(chassis\_handle->gimbal\_yaw\_ecd\_angle / RADIAN\_COEF);//YAW轴云台偏转余弦角（跟随云台模式）

    \*chassis\_vx = cos\_yaw \* chassis\_handle->vx + sin\_yaw \* chassis\_handle->vy;//底盘x方向速度

    \*chassis\_vy =-sin\_yaw \* chassis\_handle->vx + cos\_yaw \* chassis\_handle->vy;//底盘y方向速度

}

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();

        VisionDataUpdate();     //处理视觉数据

        GimbalCtrlModeSwitch();

            GimbalRampInit();

        GimbalRampUpdate();

        switch (gimbal\_handle.ctrl\_mode)

        {

            case GIMBAL\_INIT:

            {

                GimbalInitMode();           //云台初始化

            }break;

            case GIMBAL\_GYRO:

            {

                GimbalGyroAngleMode();     //云台陀螺仪模式

            }break;

            case GIMBAL\_RELATIVE:           //云台分离模式

            {

                GimbalRelativeAngleMode();

            }break;

            case GIMBAL\_NORMAL:             //云台跟随模式

            {

                GimbalNormalMode();

            }break;

            case GIMBAL\_VISION:         //云台视觉跟踪模式

            {

                GimbalVisionAimMode();

            }break;

            default:

                break;

        }

2.3.2云台控制算法

void GimbalMotorControl(GimbalMotor\_t\* motor)

{

    GimbalMotorChangeProtect(motor);

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

    {

        motor->current\_set = motor->given\_value;

        Gimbal\_PID\_Clear(&motor->pid);

    }

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

    {

        if(motor->motor\_info->motor\_type == DJ\_MOTOR)

        {

            motor->current\_set = Gimbal\_PID\_Calc(&motor->pid,

                                             motor->given\_value,

                                             motor->sensor.gyro\_angle,

                                             motor->sensor.palstance);

        }

        // else if (motor->motor\_info->motor\_type == DM\_MOTOR)  //不是陀螺仪坐标系，之后需要转换成陀螺仪角度！！

        // {

        //     fp32 relative\_angle = DMMotor\_RelativePosition(motor->given\_value/180.0f\*PI, motor->motor\_info->offset\_value);//将给定的角度转换为相对角度

        //     motor->motor\_info->given\_value = DMMotor\_CalculateAbsolutePosition(relative\_angle,motor->motor\_info->offset\_value);

        // }

    }

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

    {

        if(motor->motor\_info->motor\_type == DJ\_MOTOR)

        {

        motor->current\_set = Gimbal\_PID\_Calc(&motor->pid,

                                             motor->given\_value,

                                             motor->sensor.relative\_angle,

                                             motor->sensor.palstance);

        }

        else if (motor->motor\_info->motor\_type == DM\_MOTOR)

        {

        // fp32 relative\_angle = DMMotor\_RelativePosition(motor->given\_value/180.0f\*PI, motor->motor\_info->offset\_value);//将给定的角度转换为相对角度

        motor->motor\_info->given\_value = motor->given\_value/180.0f\*PI+motor->motor\_info->offset\_value;

        }

    }

}

2.3.3云台电机通讯控制

if (gimbal\_handle.ctrl\_mode == GIMBAL\_RELAX)

        {

            pid\_clear(&gimbal\_handle.yaw\_motor.pid.outer\_pid);

            pid\_clear(&gimbal\_handle.yaw\_motor.pid.inter\_pid);

            pid\_clear(&gimbal\_handle.pitch\_motor.pid.outer\_pid);

            pid\_clear(&gimbal\_handle.pitch\_motor.pid.inter\_pid);

            gimbal\_handle.yaw\_motor.current\_set = 0;

            gimbal\_handle.pitch\_motor.current\_set = 0;

            lock\_motor(gimbal\_handle.gimbal\_can,CAN\_ID);

        }

        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);

//        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);

                float pitch\_pos = gimbal\_handle.pitch\_motor.motor\_info->given\_value;

    float pitch\_vel=calculate\_derivative(gimbal\_handle.pitch\_motor.motor\_info->given\_value,BSP\_GetTime\_ms());

        if(gimbal\_handle.ctrl\_mode == GIMBAL\_INIT)  //初始化模式时对pitch进行限速

                {

                    VAL\_LIMIT(pitch\_vel,0,0.02);

                    VAL\_LIMIT(pitch\_pos,gimbal\_dmmotor\_pitch.pos-0.05,gimbal\_dmmotor\_pitch.pos+0.05);

                }

        GimbalMotorSendPos(pitch\_pos,

                           pitch\_vel);//后续需要添加达妙信息结构体

        osDelay(GIMBAL\_TASK\_PERIOD);

static void GimbalMotorSendPos(float \_pos, float \_vel)

{

        Get\_MIT\_Pointer()->p\_int = \_pos;

        Get\_MIT\_Pointer()->v\_int = \_vel;

        Get\_MIT\_Pointer()->t\_int = -0.0001164\*powf(gimbal\_motor\_pitch.relative\_angle,2) + 0.02533\*gimbal\_motor\_pitch.relative\_angle+ 0.4923;              //前馈扭矩

      ctrl\_motor(gimbal\_handle.gimbal\_can,CAN\_ID,Get\_MIT\_Pointer());

}

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

 \* 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)

{

    gimbal\_handle.yaw\_motor.mode = GYRO\_MODE;                                       //使能云台yaw轴陀螺仪模式

    gimbal\_handle.pitch\_motor.mode = ENCONDE\_MODE;                                  //使能云台pitch轴编码器模式

    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 -= 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

{

    fp32    pitch;           //当前Pitch轴目标角度

    fp32    yaw;             //当前Yaw轴目标角度

    fp32    last\_pitch;      //上帧Pitch轴目标角度

    fp32    last\_yaw;        //上帧Yaw轴目标角度

    fp32    distance;        //目标距离

    fp32 pitch\_ratio;        //Pitch轴移动比例

    fp32 yaw\_ratio;          //Yaw轴移动比例

    fp32 yaw\_ang\_vel;        //Yaw轴角度时间微分

    VisionState\_e state;     //VisionState\_e  0x01->Comm\_Successed

} 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;}

    vision\_data.state = TwoBytesToInt(info->state);

    vision\_data.can\_shoot = TwoBytesToInt(info->can\_shoot);

    console.can\_shoot\_cmd = vision\_data.can\_shoot;//更新控制台的can\_shoot\_cmd

    vision\_data.enemy\_move\_state = TwoBytesToInt(info->enemy\_move\_state);

    vision\_data.last\_pitch = vision\_data.pitch;

//  ESO.y = vision\_data.yaw - vision\_data.last\_yaw;

    vision\_data.last\_yaw = vision\_data.yaw;

    vision\_data.last\_palstance = vision\_data.palstance;

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

    vision\_data.pitch = FourBytesToFloat(info->pitch);

    vision\_data.yaw = FourBytesToFloat(info->yaw);

//  vision\_data.rpm =   FourBytesToFloat(info->rpm);

    vision\_data.palstance = FourBytesToFloat(info->palstance);

    vision\_data.palstance = 180 \* vision\_data.palstance / PI;

    yaw\_aim.systeam\_time = BSP\_GetTime\_ms();//获取代码总运行时间

}

2.3.5云台硬件离线检测

**/\*--------------------event------------------------|-------enable-------|-offline time-|-beep\_times-\*/           //OfflineHandle离线处理**

OfflineHandle\_Init(OFFLINE\_GIMBAL\_PITCH,            OFFLINE\_ERROR\_LEVEL,       100,         5); **//云台P轴**

OfflineHandle\_Init(OFFLINE\_GIMBAL\_YAW,              OFFLINE\_ERROR\_LEVEL,       100,         4); **//云台Y轴**

OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR1,   OFFLINE\_ERROR\_LEVEL,       100,         1); **//一声，左摩擦轮**

OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR2,   OFFLINE\_ERROR\_LEVEL,       100,         2); **//两声，右摩擦轮**

**// OfflineHandle\_Init(OFFLINE\_PLUCK\_MOTOR,             OFFLINE\_ERROR\_LEVEL,       100,         0);        //拨弹轮**

**// OfflineHandle\_Init(OFFLINE\_COVER\_MOTOR,             OFFLINE\_ERROR\_LEVEL,       100,         0);        //**

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

**// OfflineHandle\_Init(OFFLINE\_CHASSIS\_INFO,            OFFLINE\_WARNING\_LEVEL,     100,         0);  //**

**//OfflineHandle\_Init(OFFLINE\_INFRARED\_INFO,           OFFLINE\_WARNING\_LEVEL,     100,         0);**

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

**// OfflineHandle\_Init(OFFLINE\_VISION\_INFO,             OFFLINE\_WARNING\_LEVEL,     500,         0);**

**// OfflineHandle\_Init(OFFLINE\_KEYBOARDINFO,            OFFLINE\_WARNING\_LEVEL,     100,         6);        //图传链路键鼠数据**

2.3.6云台通信线程初始化

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

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

SoftwareTimerRegister(GimbalInfoUploadCallback, (void\*)NULL, GIMBAL\_UPLOAD\_TIMER\_PERIOD);

Comm\_TransmitInit(&vision\_tx\_handle, vision\_tx\_fifo\_buffer, VISION\_DATA\_FIFO\_SIZE, Vision\_UploadDataHook); //视觉发送：视觉发送地址，视觉接受信息缓冲区，视觉数据信息大小，视觉数据上载。

Comm\_ReceiveInit(&vision\_rx\_handle, VISION\_PROTOCOL\_HEADER\_SOF, vision\_rx\_fifo\_buffer, VISION\_DATA\_FIFO\_SIZE, VisionProtocol\_ParseHandler); //视觉接收: 视觉接受地址，视觉协议帧头起始，视觉接受信息缓冲区，视觉数据信息大小，视觉协议、

SoftwareTimerRegister(Vision\_RobotInfoUploadCallback, (void\*)NULL, 5); //软时钟寄存器：视觉信息回调

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

    Console\_t\* console\_info = Console\_Pointer();

    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) \* ANGLE\_TO\_RAD; //

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

//        info->yaw\_palstance = gimbal\_handle.yaw\_motor.sensor.palstance;

          info->robot\_level = RefereeSystem\_RobotState\_Pointer()->robot\_level;

      //info->bullet\_speed = RefereeSystem\_SpeedData\_Pointer()->initial\_speed;

        append\_crc16\_check\_sum((uint8\_t\*)info, sizeof(Comm\_RobotInfo\_t) + 2);  //添加CRC16校验位

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

      HAL\_UART\_Transmit\_DMA(&huart6,(uint8\_t\*)info,sizeof(Comm\_RobotInfo\_t) + 2);

}

2.4.射击部分

2.4.1 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);

2.4.2 射击判断

2.4.2.1 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;

           }

2.4.2.2 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;

}

}

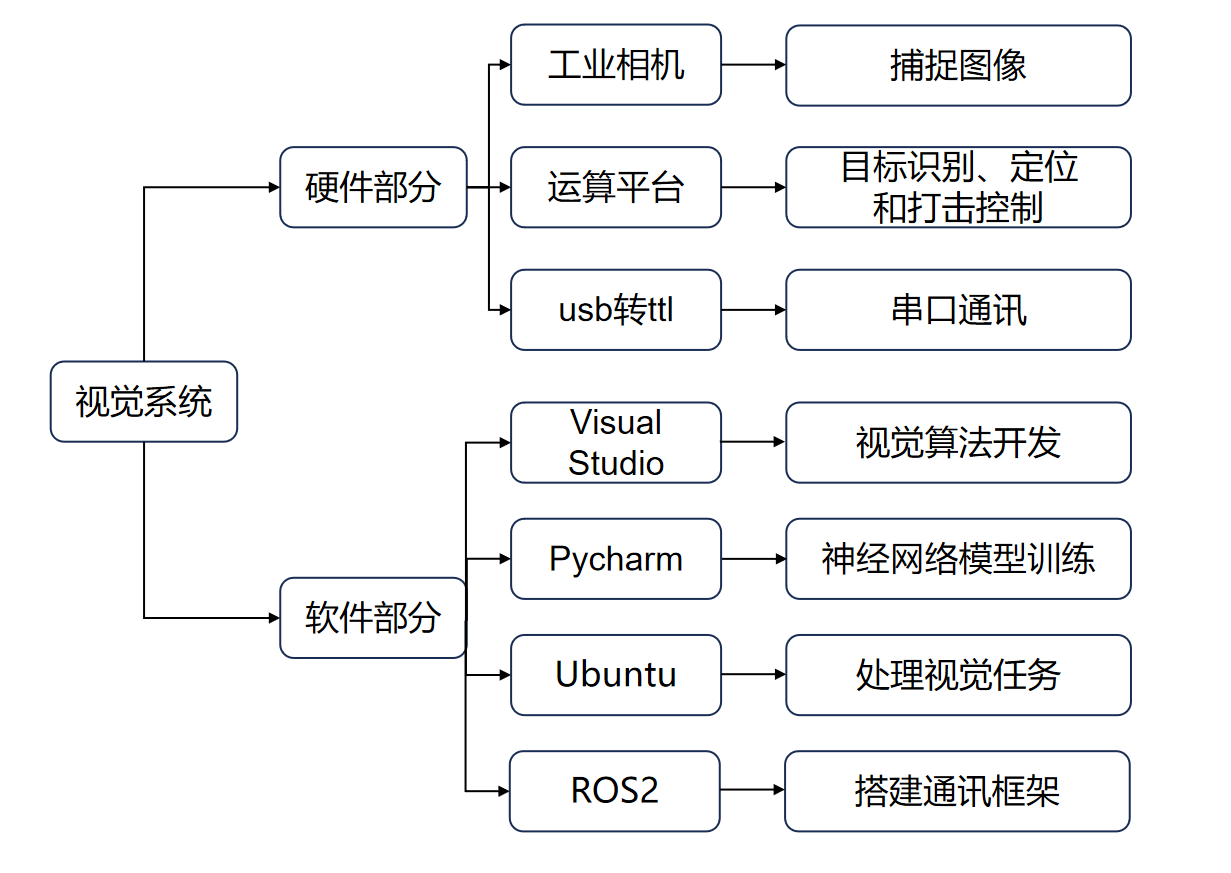
**第三部分 视觉系统设计**

**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**

**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)

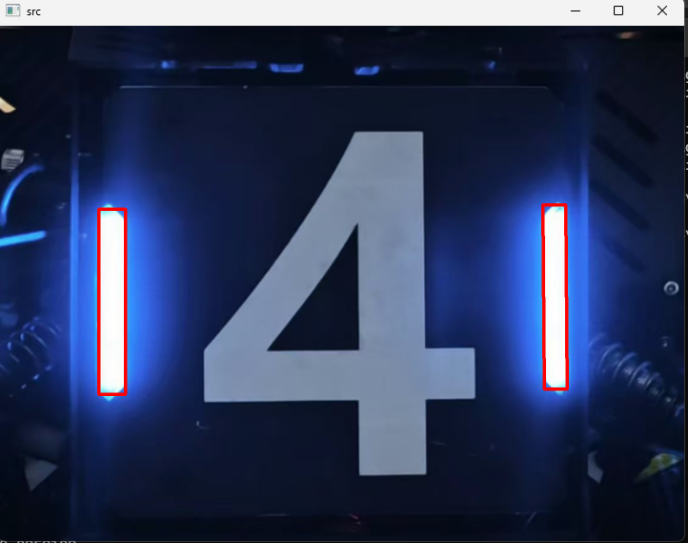
if (strip\_filter(rst\_.label, this->enemy\_blue))

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

return out\_rst;

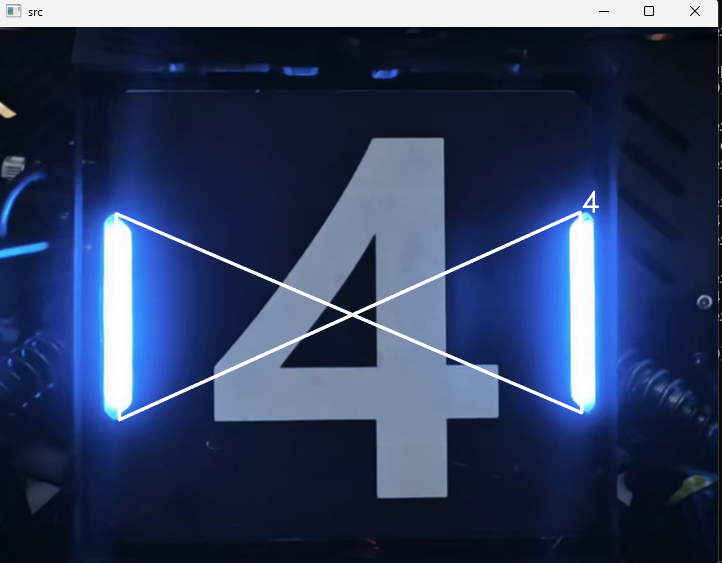
};

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，最终识别效果如图3-4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);

};

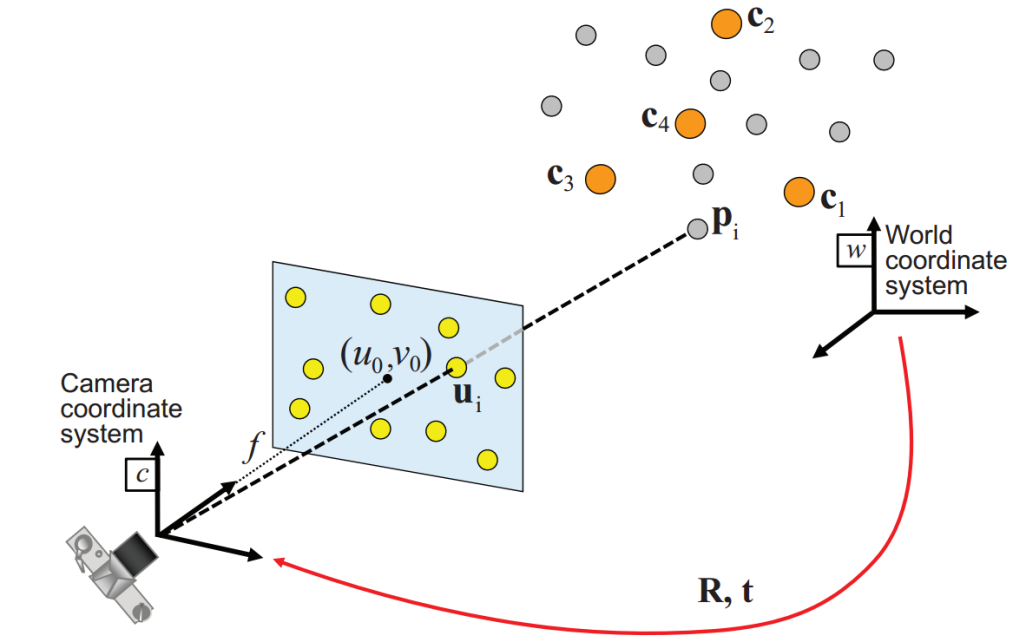
return sendarmors;

}

**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

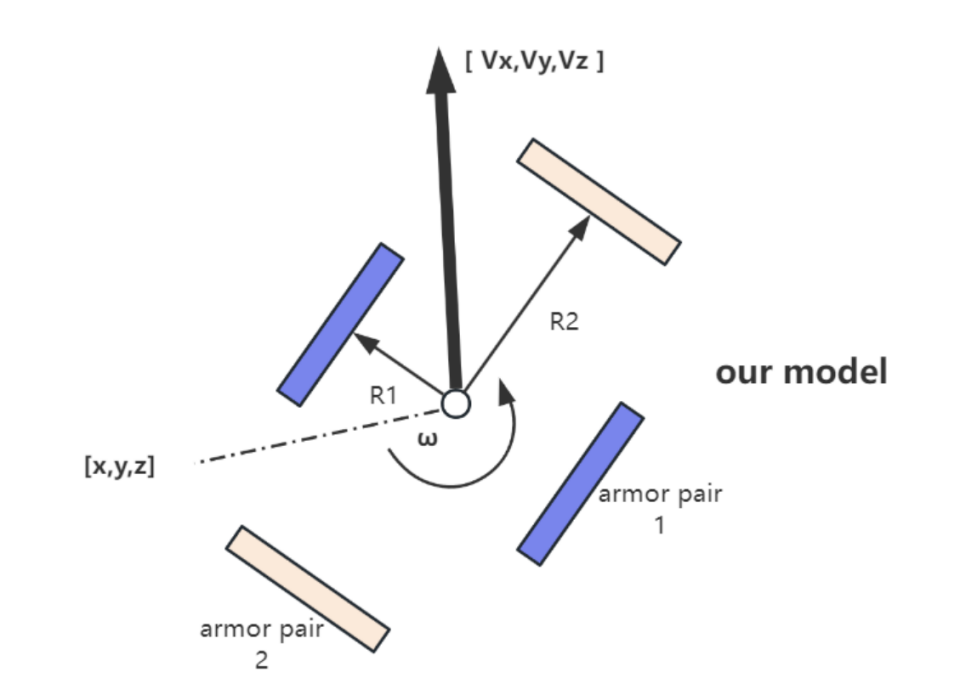
face\_angle = increase\_angle(face\_angle, car\_yaw, increase\_param);

return face\_angle;

}

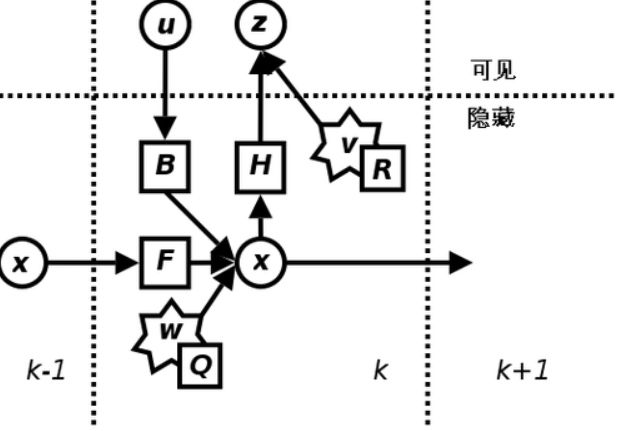
结合装甲板安装要求（pitch=15°，row=0°），我们就可以得到装甲板的位姿。

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。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

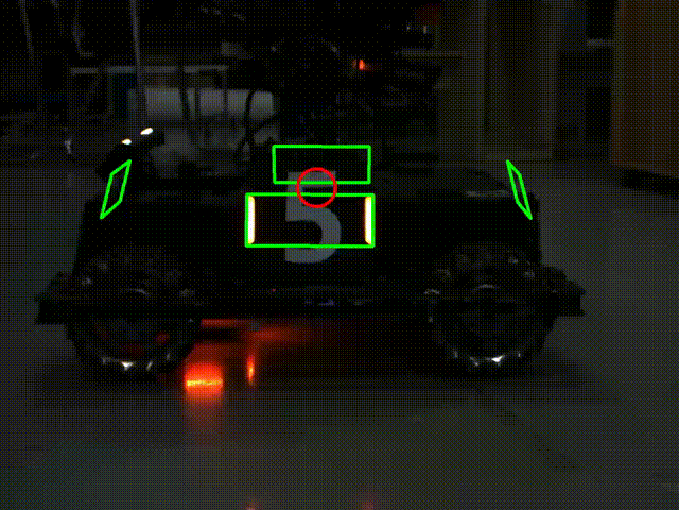
high\_low\_height\_differ = high\_armor\_pos.absolute\_middle\_point.y - low\_armor\_pos.absolute\_middle\_point.y;

// 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 硬件设计**

4.1 英雄战车基础硬件设计

4.1.1 英雄战车器件

**表4-1 英雄战车动力器件**

|  |  |  |
| --- | --- | --- |
| 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 | 2 | 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 |

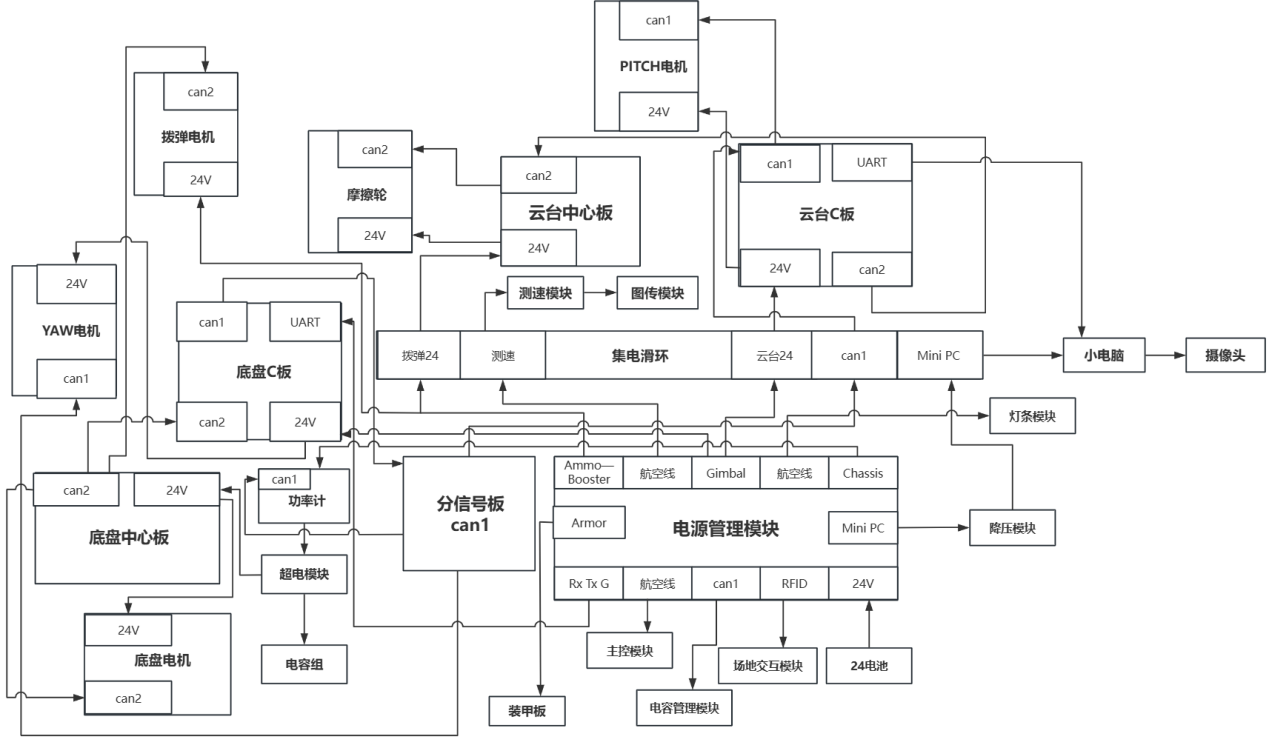
**表4-2 英雄战车控制器件**

|  |  |  |
| --- | --- | --- |
| 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 | 1 | 用于视觉接收信息处理 |

**表4-3 英雄战车其余器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| 分电板 | 2 | 用于对各电机之间供电分配 |
| 分信号板 | 3 | 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 | 1 | 单片机与视觉处理系统通信 |
| 24V转12V降压模块 | 1 | 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 |
| 电容组 | 1 | 快速充放电 |
| 超电模块 | 1 | 防止底盘功率超限的断电问题 |

英雄战车硬件接线线路可简化分为底盘和云台两部分，发射结构部分通过分线连接，底盘添加分信号板拓展通讯线路，并且通过导电滑环完成底s盘与云台的线路连接。



**图4-1 英雄战车电路拓扑图**

4.2 英雄战车战车特殊硬件设计

4.2.1 PITCH达妙电机

本次英雄战车使用了DM-J4310-2EC V1.0减速电机，它凭借自身卓越的减速性能，能够精准地控制云台运动，有效确保了云台在复杂多变的战场环境下的稳定性和精确性，为战车在执行各种战术任务时提供了坚实可靠的动力支持，极大地提升了战车的整体作战效能。

桌子上的游戏遥控器

低可信度描述已自动生成

**图4-2 DM-J4310-2EC电机实物图**

4.2.1.1 PITCH达妙电机**特点**

1. 电机与驱动器采用一体化设计，结构紧凑，集成度高，极大节省安装空间，便于设备布局。
2. 采用双编码器设计，可实现输出轴单圈绝对位置检测，即便在掉电情况下，也能确保输出轴绝对位置信息不丢失，为高精度定位提供可靠保障。
3. 支持通过 CAN 总线精准反馈电机的多项关键信息，包括速度、位置、转矩以及电机温度等，方便用户实时监测电机运行状态，实现精准控制。
4. 配备双温度保护功能，能实时监测电机及驱动器温度，一旦温度异常升高，立即触发保护机制，为设备稳定运行提供双重保障，有效延长使用寿命。
5. 支持上位机可视化调试，操作直观便捷，用户可轻松调整参数、监测运行状态；同时支持固件升级，不断优化性能、拓展功能，满足更多应用需求。
6. 在位置模式下，支持梯形加减速控制方式，可实现电机平稳启停，有效减少震动与冲击，提升设备运行的平稳性与精准性。

4.2.1.2 PITCH达妙电机性能参数

DM-J4310-2EC减速电机关键性能参数如下：

1. 速度与扭矩性能：额定电压24V，额定电流2.5A，峰值电流7.5A，额定扭矩3NM，峰值扭矩7NM，额定转速120rpm，空载最大转速200rpm。
2. 电机特征值：减速比为10:1，极对数为14，相电感340uH，相电阻650mΩ，外径56mm，高度46mm，重量约300g。
3. 编码器配置：配备两个14位磁编码器，单圈绝对位置检测，通讯接口CAN@1Mbps，调参接口UART@921600bps。

4.2.2 超级电容系统

4.2.2.1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。

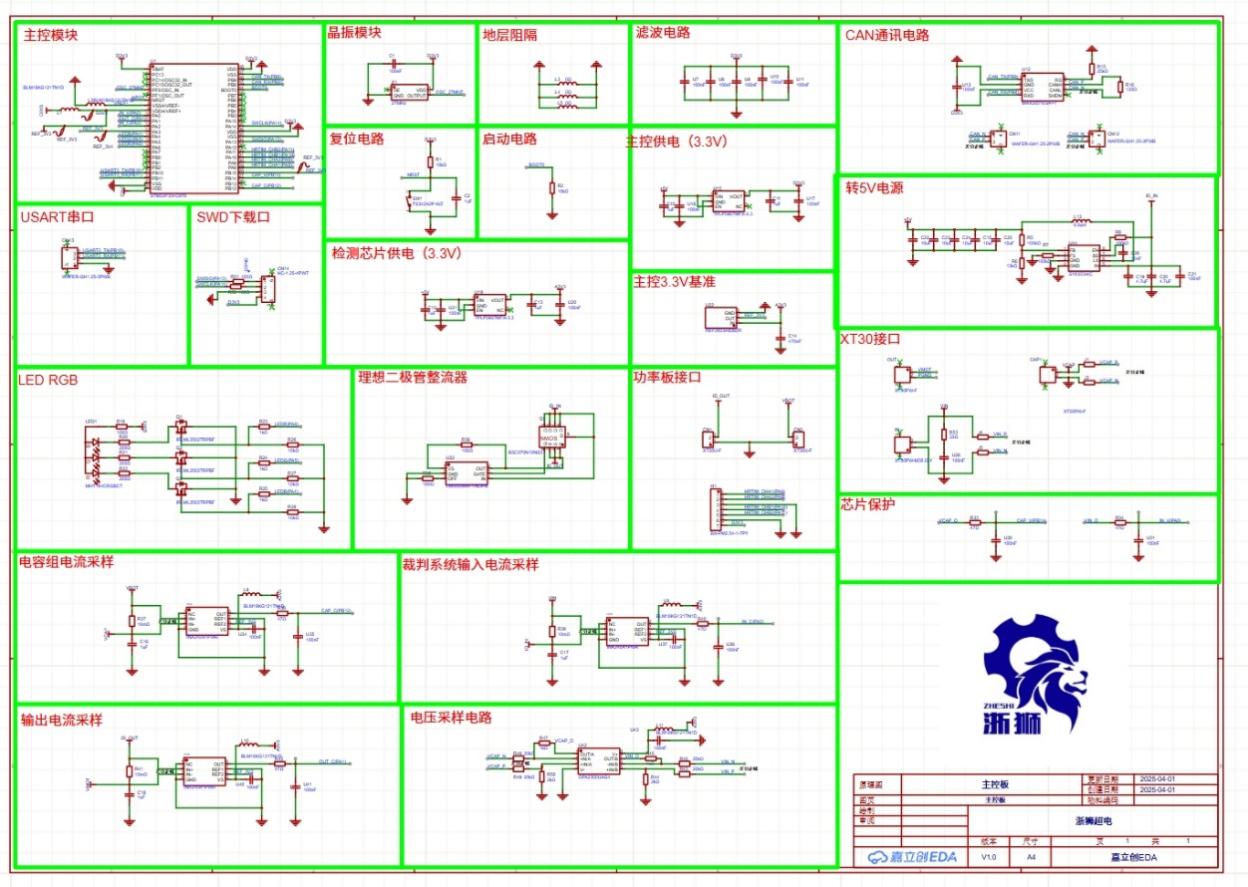
该模块采用双层板与四层板的组合设计方案。两块电路板通过2个MR30连接器（用于大电流传输）和1个7针排针（传输PWM信号）实现互连。具体布局如下：

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；

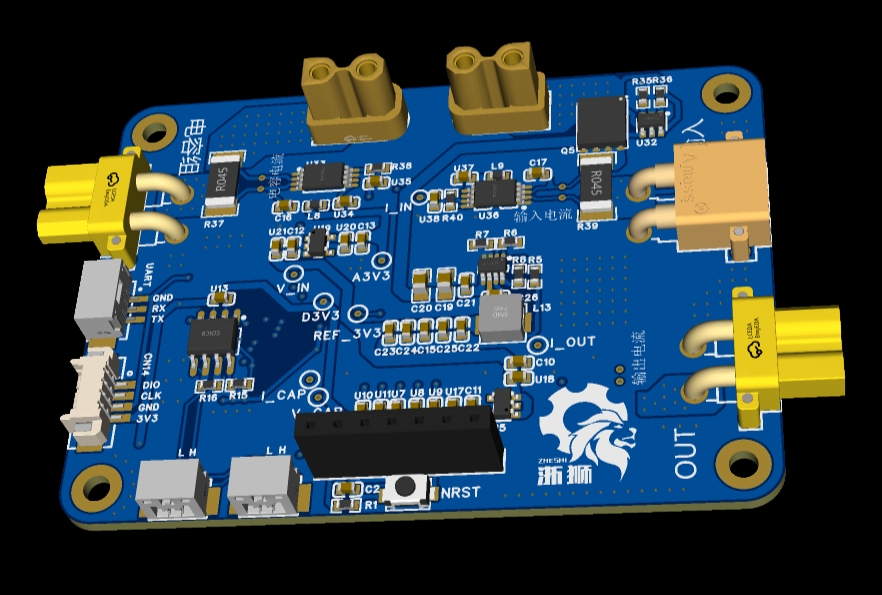
2. 控制层设计：将控制电路和采样电路集成在四层板上，充分发挥多层板的优势，实现更高密度的元器件布局，确保完整的地平面设计，提供更好的信号完整性和抗干扰能力。

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。

4.2.2.2控制部分

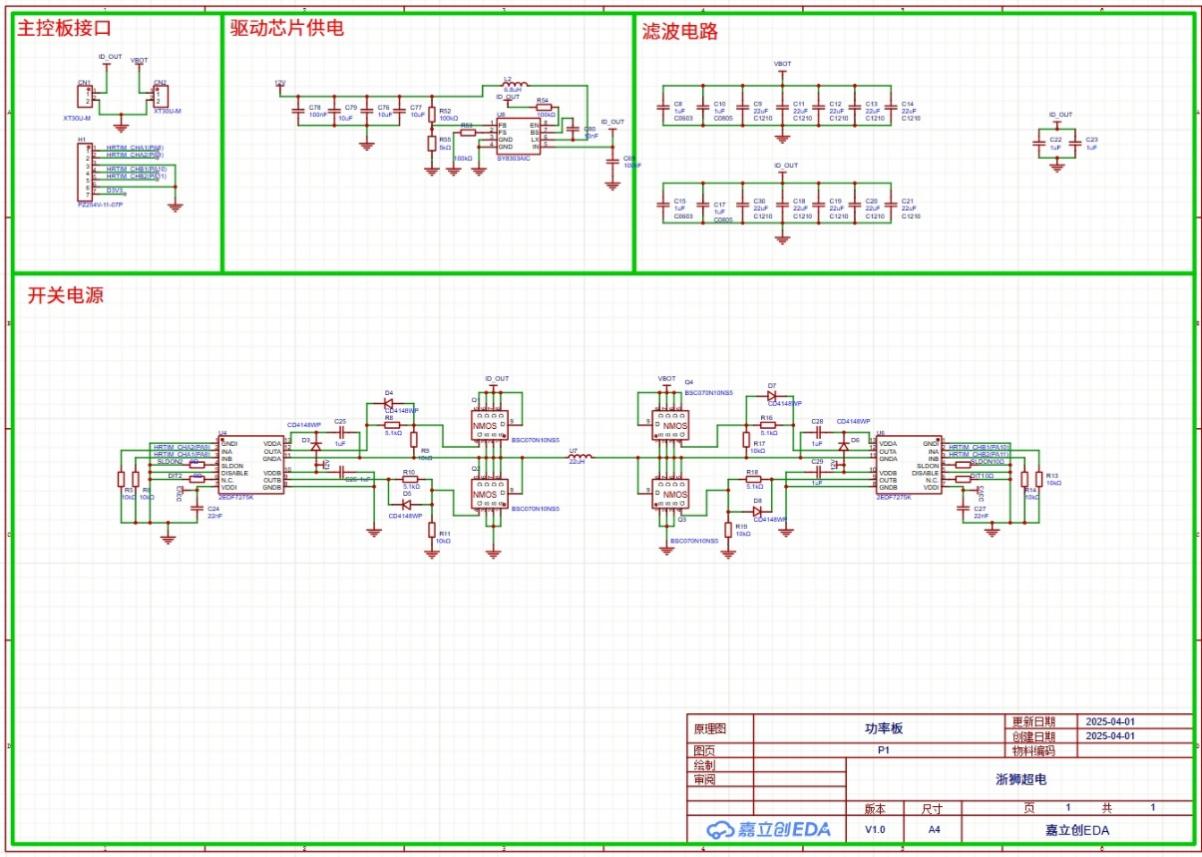


**图4-3 超级电容控制板原理图**

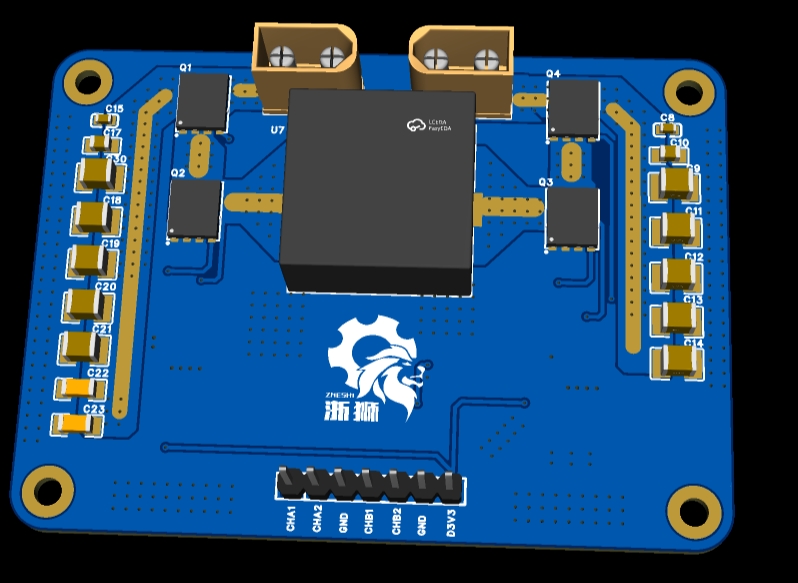


**图4-4 超级电容控制板模型图**

4.2.2.3功率部分



**图4-5 超级电容功率板原理图**



**图4-6 超级电容功率板模型图**

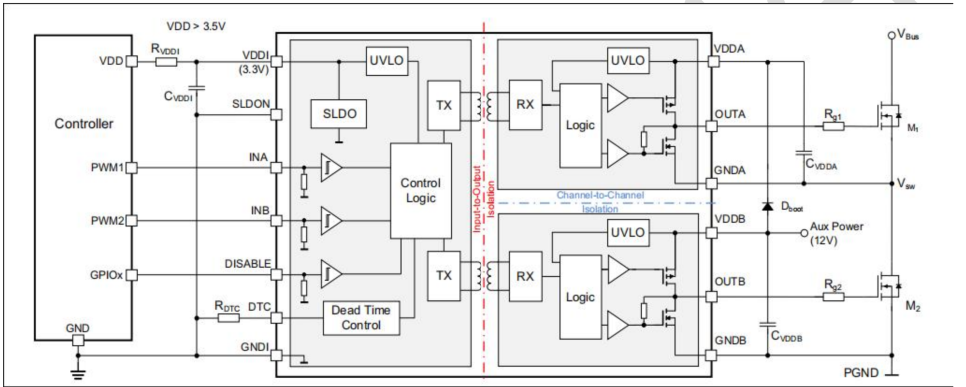
4.2.2.4 关键器件选型

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4-7 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。

4.2.2.5 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：

（4-1）

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：

（4-2）

那么我们可以得到电感电流增加量：

（4-3）

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：

（4-4）

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：

（4-5）

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：

（4-6）

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：

（4-7）

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：

（4-8）

那么我们可以得到电感电流增加量：

（4-9）

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：

（4-10）

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：

（4-11）

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：

（4-12）

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：

（4-13）

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。