

智能遥控战车 设计说明书 ——全向轮

2024年5月21日



浙狮战队

**浙江师范大学新工科试点班（中级）**

**人员名单**

（按姓氏拼音排序）

**学小心生**：

陈心强 贺怡佳 黄澄森 李学瑞 李予洋 李兴龙 洛桑尼玛 牟心舒 马下众 潘震翊 屠关阳 徐钧林 姚茂涛 朱心宁

周杭韬 周浩伟 宗骋昊 张下昊 张下泓

**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](#_Toc19568)

[1.1 智能遥控战车机械分析 1](#_Toc17911)

[1.1.1 底盘设计方案 1](#_Toc7292)

[1.1.1.1 井字型框架 2](#_Toc13583)

[1.1.1.2 避震悬挂 2](#_Toc9893)

[1.1.1.3 全向轮轮组 3](#_Toc12517)

[1.1.2 云台设计方案 4](#_Toc592)

[1.1.3 发射机构设计方案 6](#_Toc22012)

[1.1.3.1 储弹机构 6](#_Toc31199)

[1.1.3.2 拨弹仓 7](#_Toc3443)

[1.2 材料选择 8](#_Toc304)

[1.3 结构装配 8](#_Toc22016)

[第二部分 全向轮嵌入式控制开发 9](#_Toc21800)

[2.1 总体控制 9](#_Toc30165)

[2.1.1 诉求分析 9](#_Toc13457)

[2.1.2 软件设计 10](#_Toc25737)

[2.1.3 系统架构 10](#_Toc25801)

[2.1.4 控制框架 12](#_Toc30083)

[2.1.5 功能概述 13](#_Toc2330)

[2.1.6 底盘任务 14](#_Toc20083)

[2.1.7 云台任务 15](#_Toc18874)

[2.1.8 射击任务 16](#_Toc21215)

[2.2 底盘部分 17](#_Toc28598)

[2.2.1 底盘模式判断 17](#_Toc13519)

[2.2.2 全向轮速度解算 19](#_Toc1372)

[2.2.3 底盘电机通讯控制 21](#_Toc7192)

[2.2.4 底盘硬件离线检测初始化 23](#_Toc18151)

[2.2.5 底盘通信线程初始化 23](#_Toc8419)

[2.3 云台部分 24](#_Toc19818)

[2.3.1 云台模式判断 24](#_Toc31567)

[2.3.2 云台控制算法 25](#_Toc28253)

[2.3.3 云台电机通讯控制 26](#_Toc6513)

[2.3.4 自瞄模式 27](#_Toc23597)

[2.3.4.1 自瞄数据获取 27](#_Toc5625)

[2.3.4.2 视觉数据结构体 28](#_Toc14535)

[2.3.4.3 视觉数据更新 28](#_Toc7485)

[2.3.5 云台硬件离线检测 30](#_Toc4336)

[2.3.6 云台通信线程初始化 31](#_Toc31621)

[2.3.7 视觉交互数据回调函数 32](#_Toc12265)

[2.4 射击部分 33](#_Toc9929)

[2.4.1 射击任务 33](#_Toc29876)

[2.4.2 射击判断 34](#_Toc26430)

[第三部分 自瞄算法 37](#_Toc24302)

[3.1 背景与目标 37](#_Toc7523)

[3.2 视觉自动瞄准算法设计 37](#_Toc30501)

[3.3 设计方案 37](#_Toc22898)

[3.3.1 装甲板传统识别算法 37](#_Toc5888)

[3.3.2 深度学习识别算法 43](#_Toc22369)

[3.3.3 三维重建算法 48](#_Toc30668)

[3.3.4 状态估计算法 52](#_Toc19412)

[第四部分 硬件设计 55](#_Toc25111)

[4.1 全向轮战车基础硬件设计 55](#_Toc4524)

[4.1.1 全向轮战车器件 55](#_Toc28549)

[4.2 全向轮战车特殊硬件设计 56](#_Toc1311)

[4.2.1 总体方案 56](#_Toc6158)

[4.2.2 控制部分 57](#_Toc123)

[4.2.3 功率部分 58](#_Toc20020)

[4.2.4 关键器件选型 59](#_Toc20476)

[4.2.5 相关参数计算 60](#_Toc18940)

**第一部分 机械结构设计**

1.1 智能遥控战车机械分析

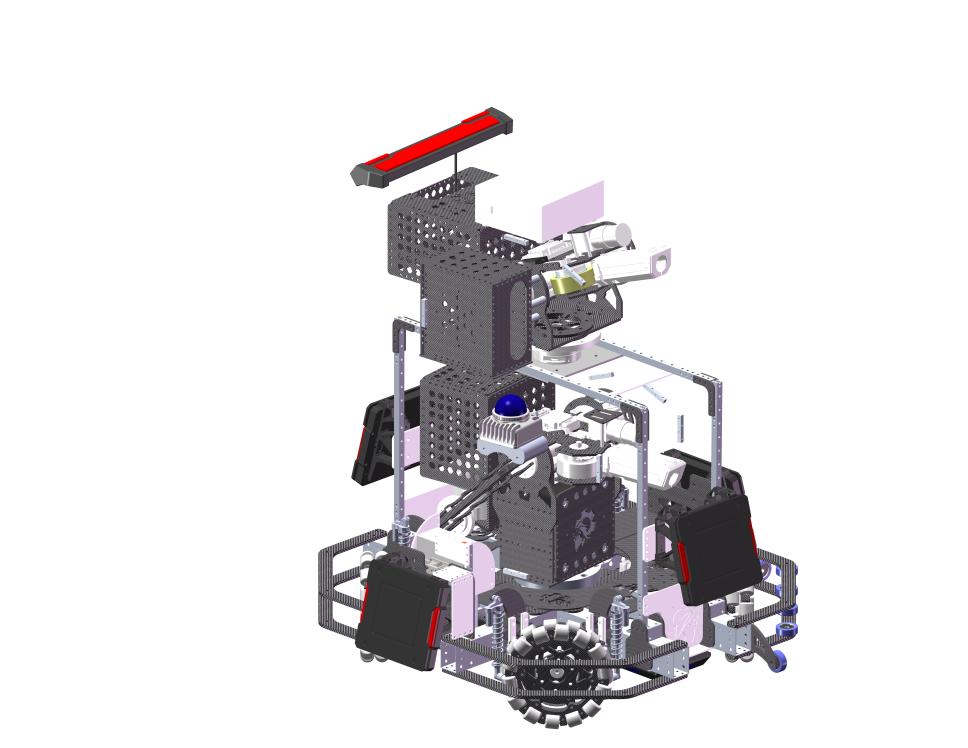
结合中级班《智能遥控战车》项目制课程教学大纲的任务要求，以设计制造智能遥控战车为项目目标，筛选出机械组对智能遥控战车（如图1-1）的整体设计要求如下：

（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28.7kg；

（3）功能要求：能实现自由移动并稳定发射17mm弹丸；

（4）设计制作周期：12个月（组队）。

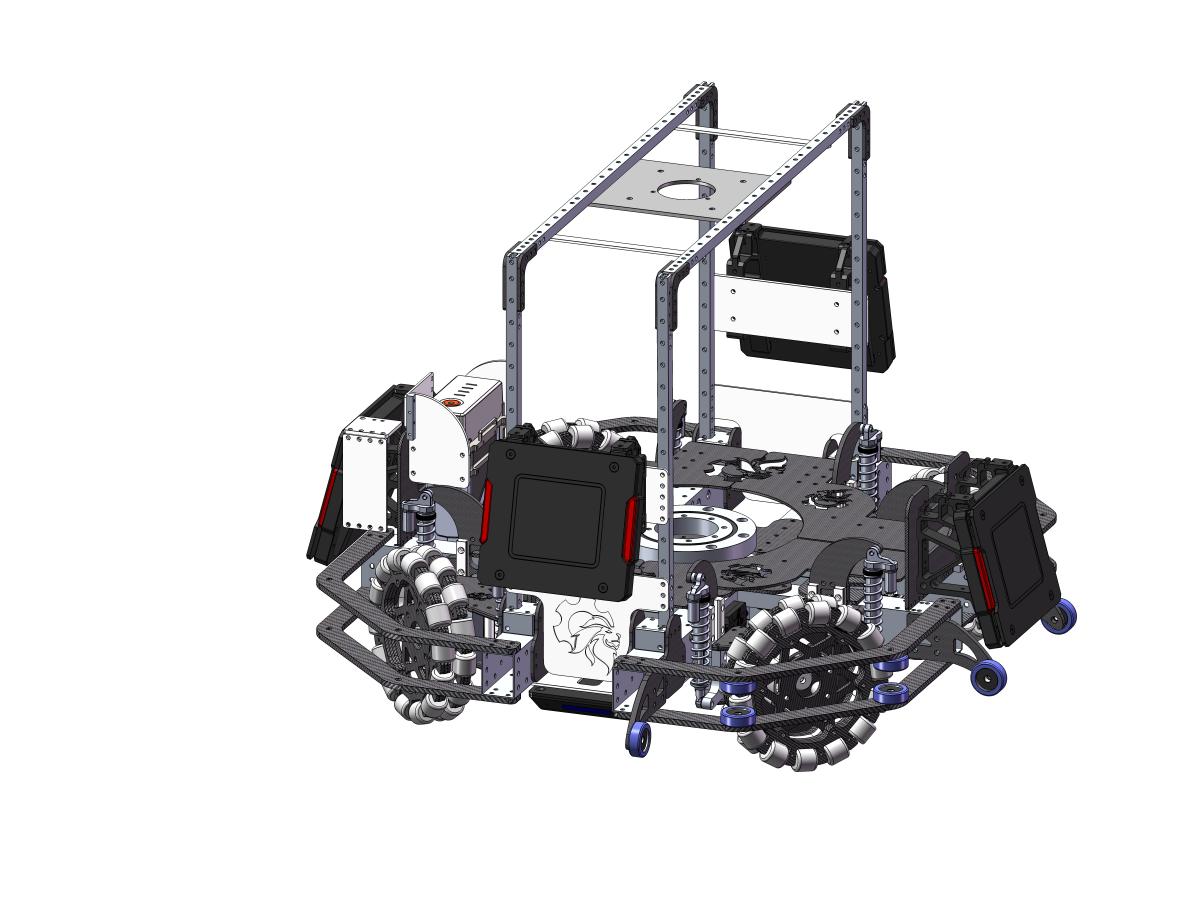


**图1-1全向轮机器**

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。即保证稳定性的前提下缩减尺寸，底盘尺寸；整车采用碳纤维板材与铝材，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

优化底盘结构，井字型的结构增加了整体的稳定性，这样的优点是使整个结构的稳定性更高，强度更高，空间预留位置更大，可以有更多的装配空间，结构简洁清晰。采用铝方管搭建框架，碳纤维连接板，保证结构强度的同时尽可能轻量化，调整重心以保证飞坡不会出现栽头或过度翘头现象。底盘整体结构如图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

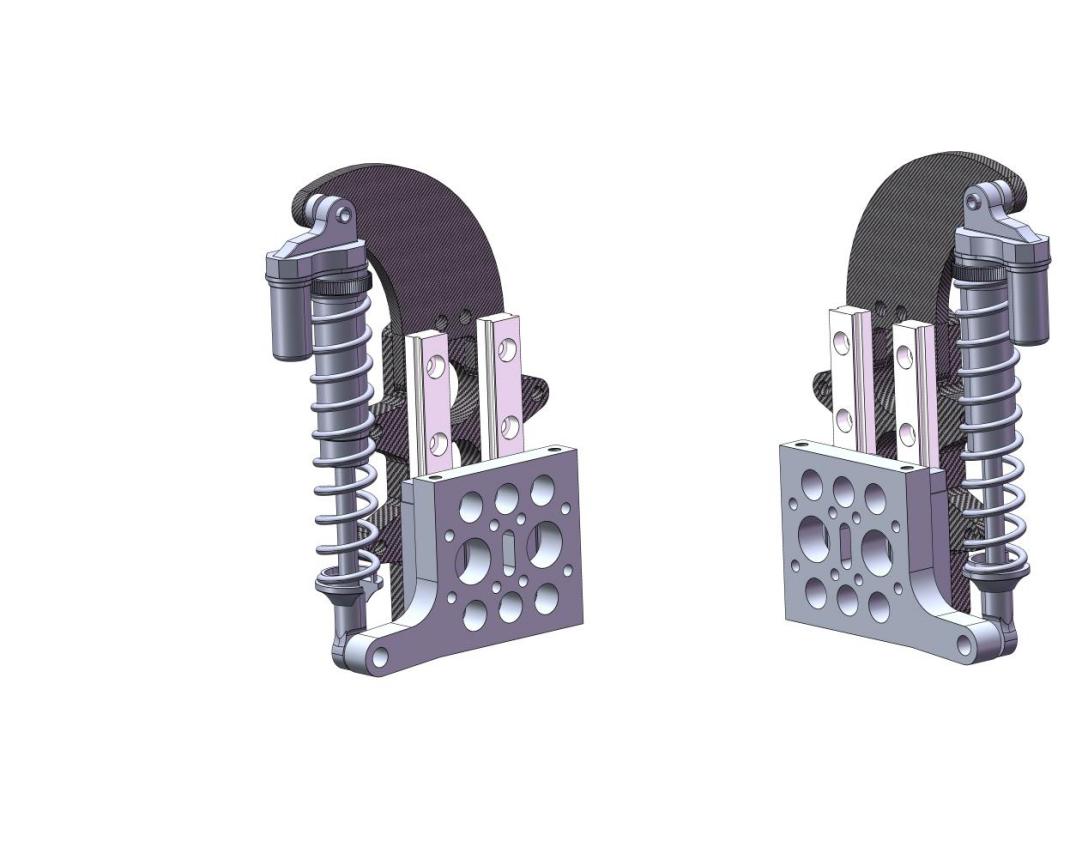
底盘的整体搭建是采用井字型铝管搭建，这样的优点是使整个结构的稳定性更高，强度更高，空间预留位置更大，可以有更多的装配空间，结构简洁清晰。同时在井字结构基础之上采用铝方管再搭建了一个平台，有利于第二个发射机构的安装，达到最后火力强大的目的。在井字型的外围开始搭建和绘制其他部分的零件，在井字型的底盘上装配，零件更加清晰。如图1-3所示：



**图1-3 井字型框架**

1.1.1.2 避震悬挂

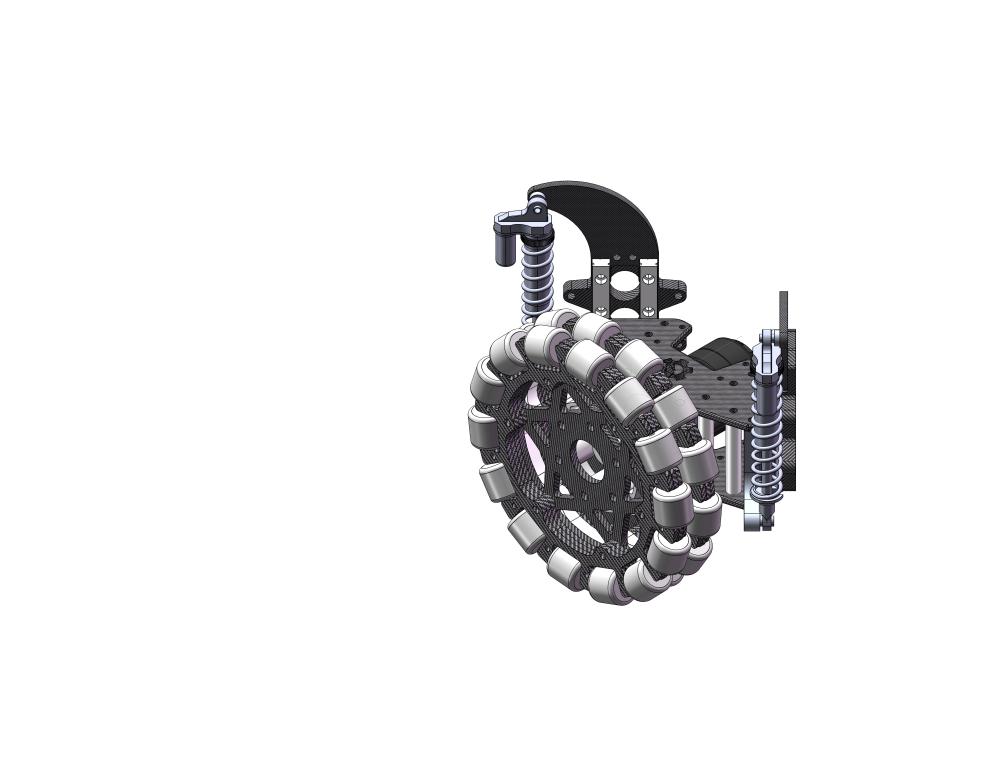
线性滑轨避震设计：机器人在移动过程中避震技术的应用至关重要，线性滑轨避震设计是一种常见的避震技术。对于全向轮底盘，比较适合的悬挂需要保证的是，无论悬挂的弹簧如何受压变形，都要保证轮组相对于地面的直立（或者相对于斜坡要垂直于斜坡），线性滑轨避震可以保持在垂直方向的作用力缓冲，滑块加上滑轨一方面可以做到校正受力方向始终竖直方，另一方面可以缓冲一部分力，以此保持轮组和车身yaw轴的平行，也即和地面垂直。该设计通过弹簧以及避震器内部的液压缸来消除机器人在运动过程中的振动和冲击，具有稳定性好、精度高、寿命长等优点。采用线性滑轨避震设计的机器人能够在高速、高精度运动中保持平稳，提高机器人的稳定性和精度，减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

全向轮设计在机器人领域具有重要意义。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。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，通过控制轮子的运动实现高精度的定位和导航。因此，采用全向轮设计的机器人在自动化生产线、智能仓储和服务机器人等领域有着广泛应用前景，为人们创造了更大的可能性和便利性。如图1-5所示：

****

**图1-5 全向轮**

采用M3508电机驱动整个全向轮轮组运动，全向轮使用碳纤维板材、PLA3D打印材料以及购买的小轮子组成，通过小轮子之间的转动以及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1-6所示：



**图1-6 轮组设计（驱动部分）**

1.1.2 云台设计方案

360°云台（如图1-7），支持小陀螺，采用2A、24线小滑环，将滑环安装在 6020内部，降低云台整体高度。云台Yaw轴和Pitch轴电机均为GM6020电机，降低整车的控制难度；云台与地盘采用交叉滚子轴承连接，在轴承的上下加装4mm碳板，增加结构强度，减小轴心晃动，同时可以抵抗飞坡与撞击时的冲击，防止“断头”情况发生。



**图1-7 360°云台**

另外Pitch轴驱动电机采用直驱形式，电机直驱有以下优点促使本项目去选择了电机直驱：

（1）高效能：由于直驱电机无需通过传动装置传递动力，可以减少传动装置带来的传动损失，提高能量利用率。

（2）精度高：由于没有传动装置的影响，直驱电机转动的精度和稳定性较高，适用于对位置和速度要求较高的设备。

（3）维护简便：直驱电机结构简单，减少了传动部分，减少了机械故障的概率，同时也方便维护。

（4）响应速度快：由于直驱电机无需传动装置，可以直接响应驱动信号，转速调节响应速度快。

但是，相应的也会有以下缺点：

（1）配重难：转轴中心前后的配重难控制，导致电机消耗大，损耗大。

（2）重心偏高：相较于传统的连杆机构，直驱电机安装位置普遍偏高，导致重心上移。

（3）拆卸不方便：由于直接与整个云台连接，拆卸相较于连杆机构会更难拆卸

同时我们还采用了侧供弹的供弹方式，侧供弹有以下优点使我们选择了侧供弹：

（1）提高射击精度：侧供弹可以让机器人的重心更低，降低机器人的重量分布中心，提高了机器人的稳定性和定位精度。

（2）减小机器人尺寸：传统的上方或后方装填需要在机器人的上部或后部增加特定的弹药储存区域，因此会增加机器人的体积和重量，而采用侧供弹方式则可以减小机器人的尺寸和重量。

但是同时也会有以下缺点：

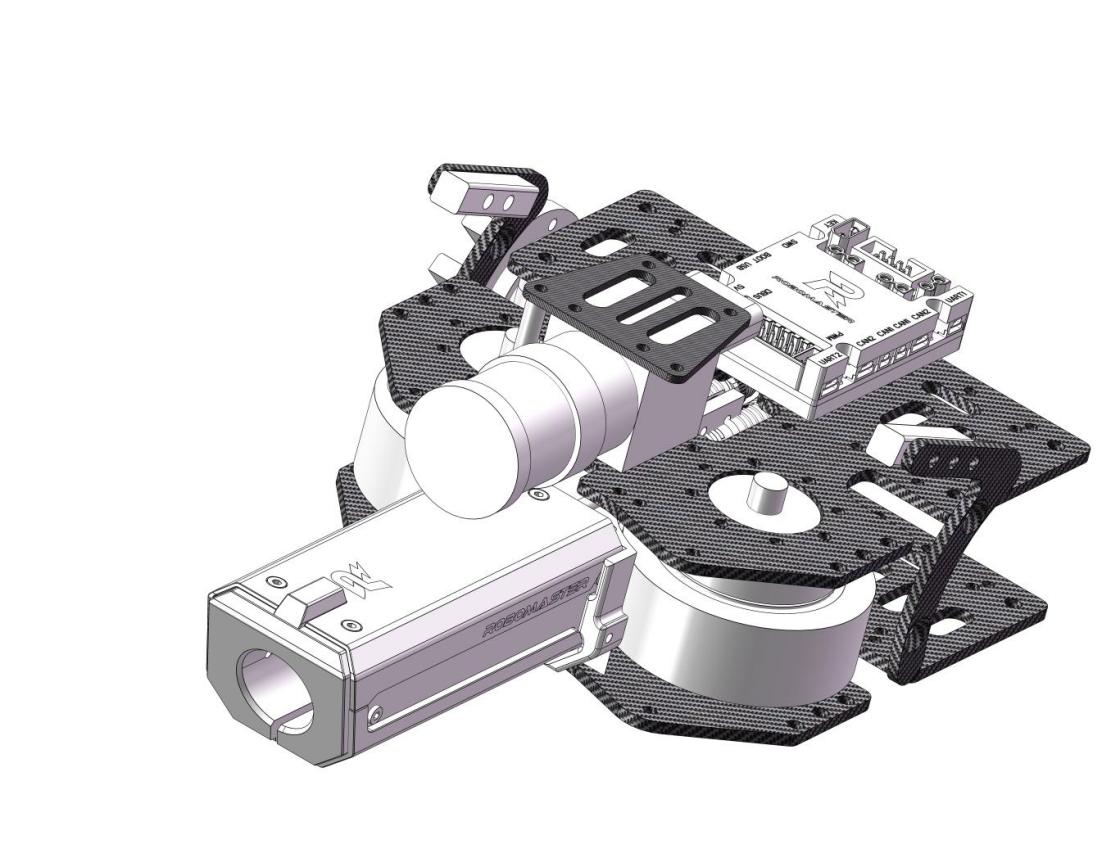
（1）增加供弹电机的负载，使电机维护成本高。

（2）增加复杂性：侧供弹系统相对于其他供弹方式可能更加复杂，需要更多的机械部件和控制系统来确保弹药的正常供给和装填。这增加了机器人的维护和故障排除的难度。

（3）增加了卡弹的可能性，链路越长，越难提供弹丸。

1.1.3 发射机构设计方案

由于智能遥控战车底盘空间紧缺，采用侧供弹云台。使用拆除减速箱的3508作为摩擦轮电机，通过调整摩擦轮间距，使弹丸经过拨弹舱电机的驱动经过挤压发射，经过测试达到了12m小装甲命中率大于90%的点射精度。重新设计了拨弹盘，利用类似官方车拨弹盘的拨叉和盘壁轴承设计，达到了25发/s的射频下满舱情况下连射不卡弹。



**图1-8 发射机构**

1.1.3.1 储弹机构

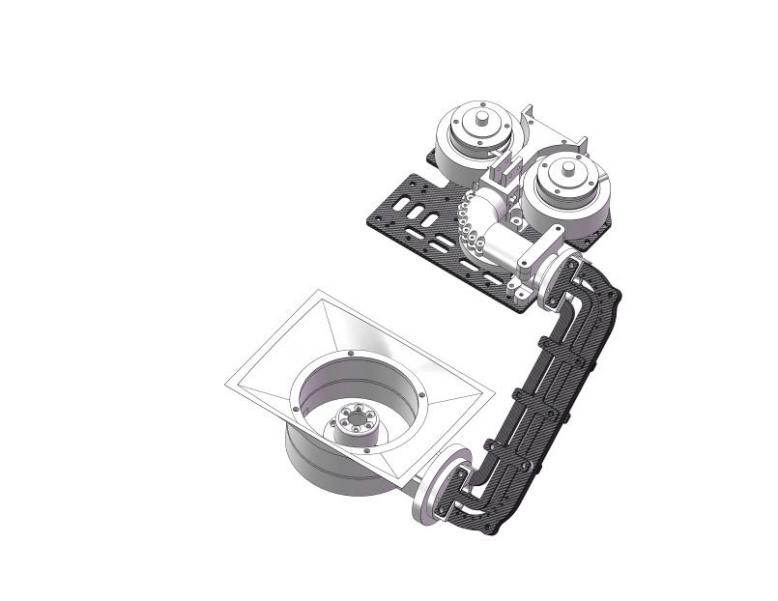
弹药存储位置与供弹方式联系紧密，供弹方式主要有上供弹与下供弹两种，下供弹可以将弹丸存储位置设置在云台之下，将极大减轻云台的负担，但关键设计点在弹道，较为复杂。上供弹方式将弹丸储存在云台之上，利用弹丸在自身重力作用下落入拨弹仓进行供弹。此种供弹方式弹道简洁，只要合理控制弹药舱重心质量，就不用额外设计弹道。如图1-9所示，弹药舱为整体为长方体，各面板由四角转接件连接，尺寸为154mm×115mm×145mm。

****

**图1-9 储弹机构**

1.1.3.2 拨弹仓

拨弹仓中关键结构为拨弹盘，而拨弹盘有两个重要指标，一个是单位时间内的供弹数量，这直接影响着弹丸输出率，也称射频，另外一个指标则是可靠性，这个指标比射频更加重要，如果结构不够可靠，拨弹仓容易卡弹，则发射机构直接丧失输出能力，功能失效。拨弹盘通过自身旋转赋予落入拨弹仓的弹丸一定的初速度从而推入弹道，进入摩擦轮。整个过程其实就是将一定数量组成的弹丸整体先分散在拨弹轮的同一平面上，再沿设定角度线推出，摩擦轮加速射出，这样就实现了由整体到平面，再到直线，最后成为发射出去的一点，是一个有规律的过程，如图1-10所示。



**图1-10 拨弹机构和弹道**

因此，拨弹轮要具有一定的厚度，至少要高于弹丸直径，并且能将堆叠密集的弹丸有效分层。拨弹轮旋转拨动弹丸时，弹丸同时受到拨弹仓内地面和内壁的摩擦力，因此拨弹仓内壁设计为圆形并尽可能使底部光滑。

1.2 材料选择

由于整车大部分零件为碳板和铝合金加工件，含少量3D打印件，所以主要工艺为2D雕刻以及车铣加工。在机器人底盘处主要由铝方管搭建骨架，轻量化的同时强度也得到了保证；

在机器人外壳部分就主要使用碳板，因地制宜地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，本项目选择使用铝合金机加工，高品质的发射头是机器人射击精准与否的重要前提；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，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1-11），方便机器人壳体的拼接。



**图1-11 四孔直角连接**

**第二部分 嵌入式控制开发**

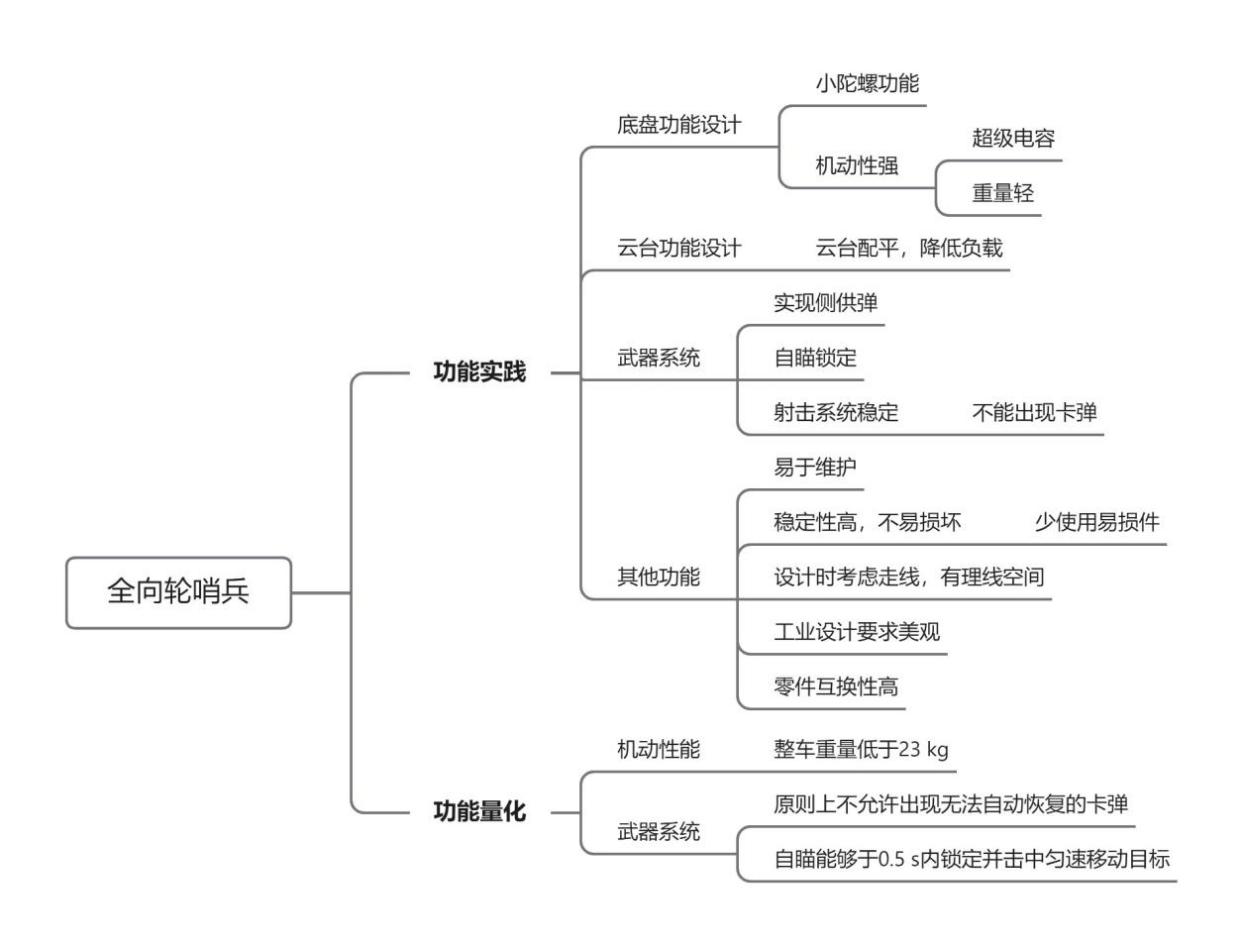
2.1 全向轮总体控制

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。将全向轮战车的功能拆分为底盘功能、云台功能、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：

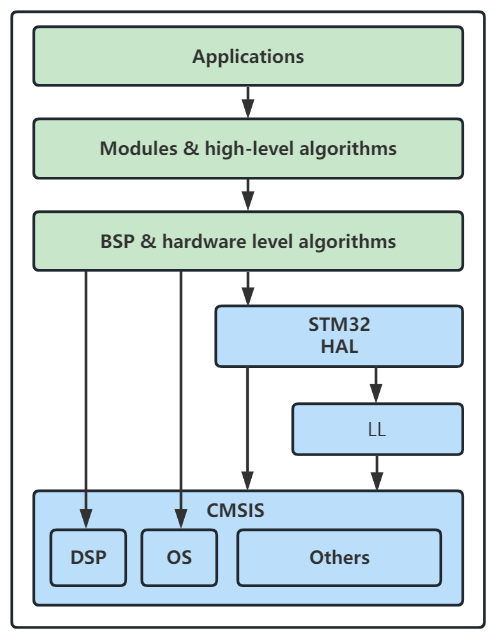
框架在结构上分为三层:bsp/module/app。整体使用的设计模式是结构层级模式，即每个“类”包含需要使用的底层“类”，通过组装不同的基础模块实现更强大的功能。而最顶层的app之间则过pub-sub消息机制进行解耦，使得在编写代码时不会出现相互包含的情况。

我们希望通过bsp对硬件的抽象使得module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module的外接模块的抽象，使得app的编写可以通过完全硬件无关的方式考虑，达到只阅读module说明文档就能迅速开发应用的程度。bsp和module的设计愿景，就是成为人们常说的中间件。

pub-sub机制的体现:以全向轮战车软件的app层为例，含了chassis、gimbal、shoot、cmd四个应用，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、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 module)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。

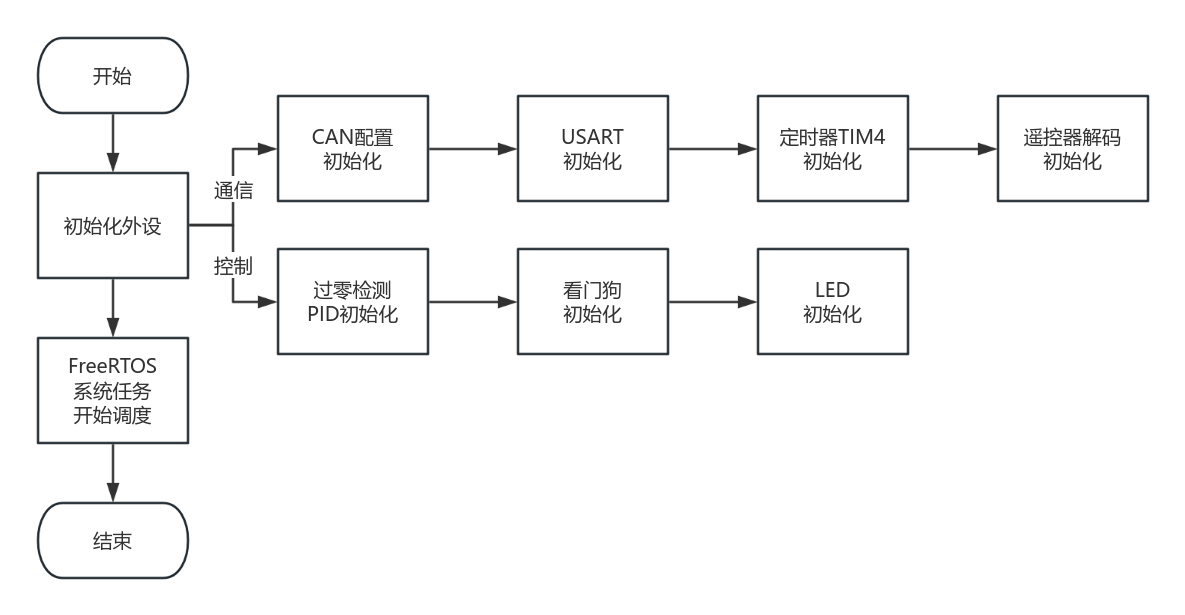
整体架构如图2-2所示。



**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. **双云台锁定功能** 上下云台可以单独执行相关指令，根据预设轨迹分别对周围环境进行识别扫描。当上云台检测到目标后，下云台能自动转向上云台所在方位。两者共同锁定同一目标，进行快速打击。
2. **单云台扫描功能** 上云台能根据预设轨迹进行360°无死角扫描，能实现自动识别和跟踪功能。当视野中出现目标，枪口会快速锁定目标，并进行精准打击。
3. **底盘控制模式切换** 底盘可以接收裁判系统的相关数据，并根据这些数据来判断当前比赛状态。在比赛正式开始之前，底盘不会执行预设的移动轨迹；而一旦比赛正式开始，底盘将按照预设的轨迹进行移动。当比赛正式结束时，底盘会停止移动，以确保比赛结束后的安全状态。

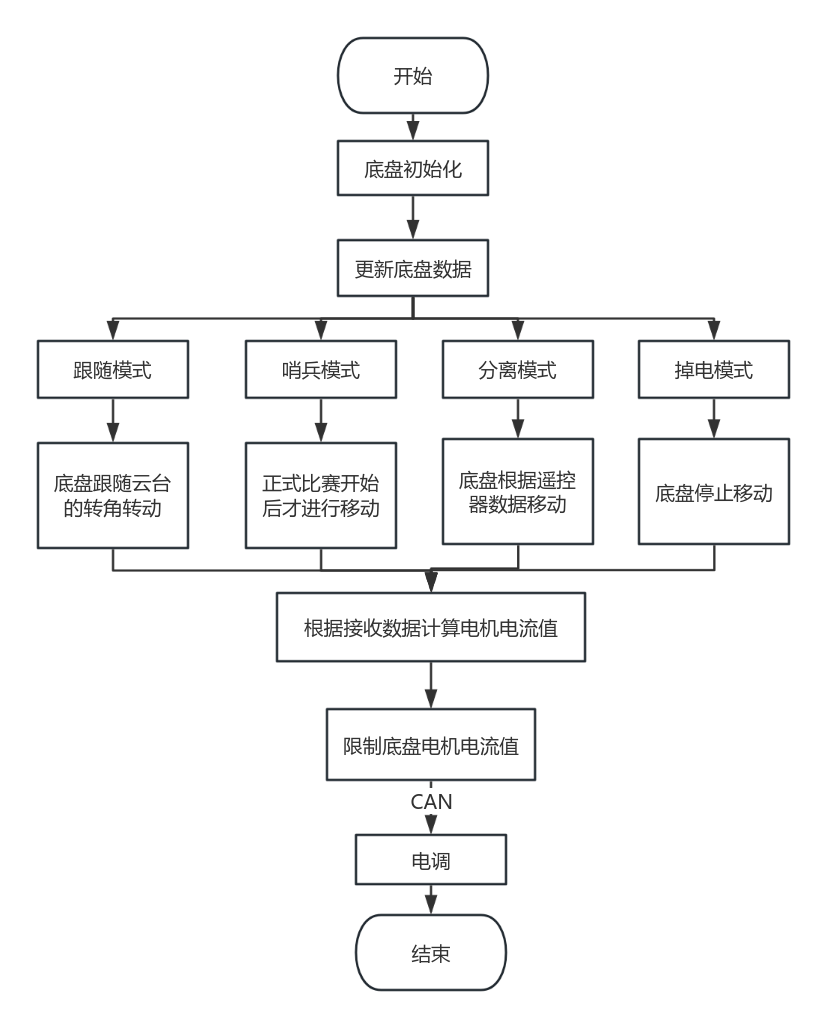
其余功能见表2-2。

**表2-2 其余功能**

|  |  |  |
| --- | --- | --- |
| 参数 | 描述 | 频率/HZ |
| **Start\_Task()** | 程序启动任务，用于区分云台和底盘控制板并执行对应任务初始化函数 | 单次 |
| **Chassis\_Task()** | 英雄车底盘任务，包括麦轮基本运动和速度切换 | 500 |
| **Gimbal\_Task()** | 下云台控制任务，包括 yaw，pitch和云台抬升的控制 | 1000 |
| **Gimbal\_\_Other\_Task()** | 上云台控制任务，包括 yaw，pitch和云台抬升的控制 | 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()** | 下云台发射任务，包括拨弹轮及摩擦轮的控制及热量检测 | 1000 |
| **Shoot\_Other\_Task()** | 上云台发射任务，包括拨弹轮及摩擦轮的控制及热量检测 | 1000 |
| **Comm\_Task()** | 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()** | 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

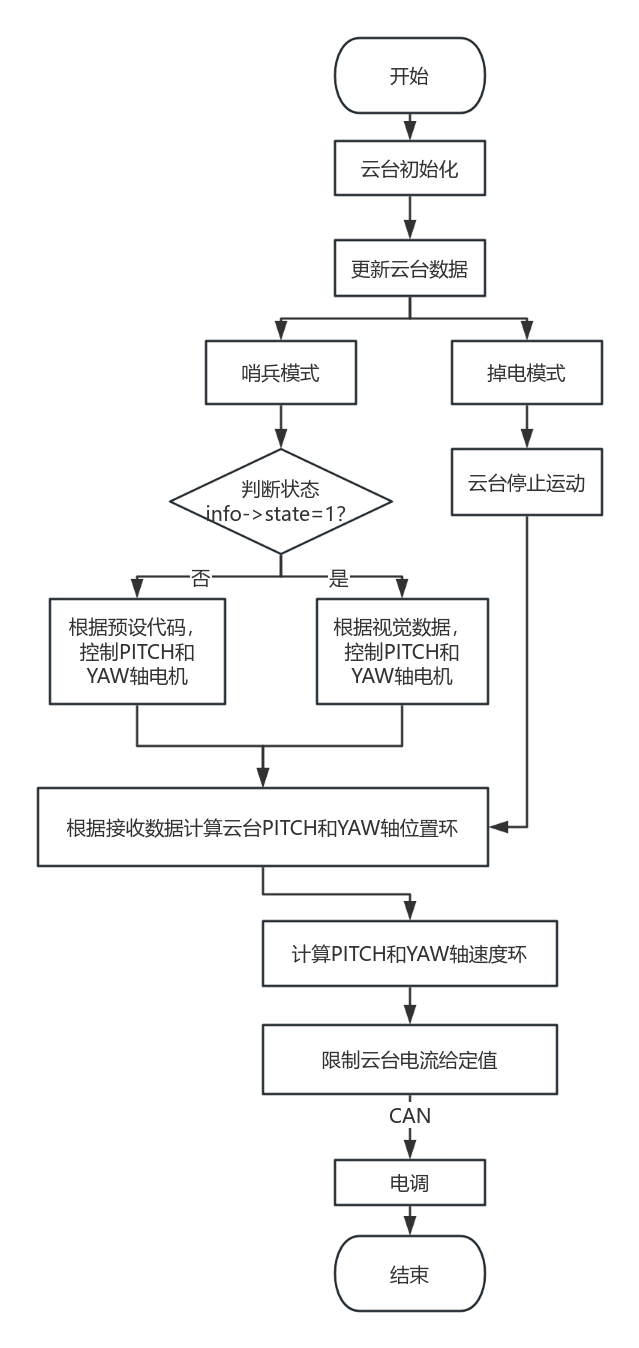
在Chassis\_Task中,底盘控制，主要模式为底盘无力或跟随云台，控制底盘速度方向，以及功率限制，底盘控制流程如图2-4。首先进行底盘初始化，主要为设置初值和PID参数初始化，然后进入主循环。循环中，首先更新底盘数据，包括底盘电机转速、YAW轴云台电机编码器角度(用于底盘跟随云台)。然后根据不同的控制模式进行处理。如果底盘为无力状态，则直接设置底盘电机电流值为0。否则，根据键盘或遥控拨杆的控制，计算机器人XY方向期望速度。如果机器人在小陀螺模式，则设置底盘旋转速度；否则，根据YAW轴云台电机编码器角度进行PID计算期望的底盘旋转速度。之后根据XY方向速度和底盘旋转速度计算四个底盘电机分别的期望转速。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

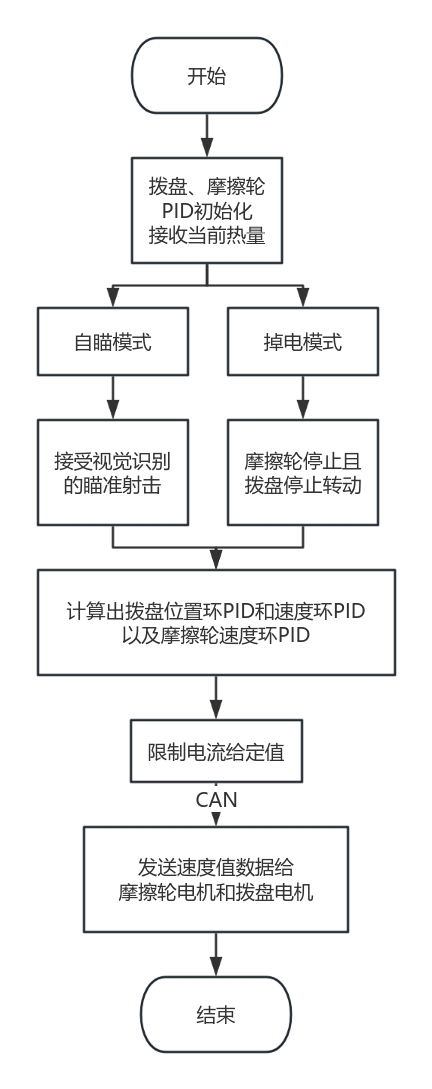
全向轮战车的云台采用双云台，云台任务分为两块：上云台任务和下云台任务。在Gimbal\_Task和Gimbal\_Other\_Task中，进行上下云台的独立控制，但总体控制流程类似。云台控制包括使用遥控对云台角速度、角度控制，以及开启自瞄时的角度控制，云台控制流程如图2-5。首先进行云台初始化，主要为设置初值和PID参数初始化，然后进入主循环。循环中，首先更新云台数据，包括角度、角速度。然后根据不同的控制模式进行处理。如果鼠标或遥控拨杆有控制，则根据这个控制信号设置角速度，不进行角度环计算。如果没有控制数据，则保持上一刻的角度，根据这个角度进行角度环PID计算。角度环PID计算得出期望角速度。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 射击任务

全向轮战车的云台设计采用单个云台搭配单个枪管，所以射击任务也分为上云台射击任务和下云台射击任务，但总体控制类似。在Shoot\_Task和Shoot\_\_Other\_Task中，进行摩擦轮转速控制，以及拨弹盘的控制，射击控制流程如图2-6。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。根据预设代码，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);//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();//IMU、码盘数据读取

        GimbalCtrlModeSwitch();//与控制台(Console)云台模式进行比较并切换

        GimbalDirection();//暂时无用

        switch (gimbal\_handle.ctrl\_mode)

        {

             case GIMBAL\_ASSIST:

            {

                GimbalAssistMode();  //辅瞄模式ver0.1

            }

            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:

            {

                GimbalRelativeAngleMode();//云台分离模式

            }break;

            case GIMBAL\_NORMAL:

            {

                GimbalNormalMode();//云台正常模式(底盘跟随)

            }break;

            case GIMBAL\_BACK:

            {

                GimbalBackMode();//掉头模式(以尾为头)

            }break;

            case GIMBAL\_VISION\_AIM:

            {

                GimbalVisionAimMode();//自瞄模式

            }break;

            case GIMBAL\_VISION\_AIM2:

            {

                GimbalVisionAimMode2();

            }break;

            case GIMBAL\_SENTRY:

            {

                GimbalSentryMode();

            }break;

case GIMBAL\_ADD:

            {

                GimbalADDMode();

            }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)//陀螺仪模式

    {

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

                                motor->given\_value,

                                 motor->sensor.gyro\_angle,

                                 motor->sensor.palstance);//PID计算角度控制

    }

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

    {

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

                               motor->given\_value,

                                motor->sensor.relative\_angle,

                               motor->sensor.palstance);//PID计算角度控制

    }

}

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)      //自瞄（自开火手移动）

{

    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

    VisionDataUpdate();     //处理视觉数据

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

VAL\_LIMIT(gimbal\_handle.pitch\_motor.given\_value, gimbal\_handle.pitch\_motor.min\_relative\_angle, gimbal\_handle.pitch\_motor.max\_relative\_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

VisionAim\_e yaw\_success;//YAW轴数据传输成功

VisionAim\_e pitch\_success;//PITCH轴数据传输成功

} VisionDatabase\_t;

2.3.4.3 视觉数据更新

static void VisionDataUpdate(void)

{

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

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

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

    vision\_data.pitch = info->pitch\_int + (info->pitch\_dec)\*0.01;

    if ((int)vision\_data.pitch >=30){

    vision\_data.pitch = 30.0f - vision\_data.pitch;

    }

    vision\_data.yaw = info->yaw\_int + (info->yaw\_dec)\*0.01;

    if ((int)vision\_data.yaw >=60){

    vision\_data.yaw = 60.0f - vision\_data.yaw;

    }

    vision\_data.distance = info->distance\_1 + info->distance\_2;

    vision\_data.state = info->state;

    vision\_data.yaw\_ang\_vel = (vision\_data.yaw - vision\_data.last\_yaw);

}

static void Aim\_contorl(AutoAim\_t\* Aim)

{

   if (vision\_other\_data.state)//视觉NUC在线时，此条件一直成立

    {

    if (ABS(vision\_other\_data.yaw)<Aim->tol\_angle)//自瞄已经达到一定角度之内

      {  //thought to be aimed right认为自瞄正确

        if (vision\_other\_data.yaw\_success == AIM\_NO&&!Aim->first\_aim)

        {

        /\*\*\*\*\*\*\*\*\*\*\*\*记录未进入第一次自瞄的时间\*\*\*\*\*\*\*\*\*\*/

        Aim->Sstart\_time = Aim->systeam\_time;

        Aim->first\_aim = 1, Aim->aiming\_time = 0;//进入第一次自瞄

        Aim->Vision\_PID.p = 0.011;//user set

        }

        else if(vision\_other\_data.yaw\_success == AIM\_NO&&Aim->first\_aim)

        {

            Aim->stay\_time = Aim->systeam\_time-Aim->Sstart\_time;//comparing aimed right time

            if(Aim->stay\_time>Aim->tol\_time)//自瞄持续时间达到tol\_time的标准

            {

              Aim->first\_aim = 0,Aim->aim\_flag = 0;//认为已完成瞄准，退出此次自瞄及消抖

              if(ABS(vision\_other\_data.yaw)<0.5f)vision\_other\_data.yaw\_success = AIM\_RIGHT;//if long enough ,thought to be aimed stable

            }

        }

      }

    else{  //自瞄开始，此时还未瞄准到规定角度差

         vision\_other\_data.yaw\_success = AIM\_NO,Aim->Vision\_PID.p = 0.013;//user set

      if (Aim->aim\_flag == 0)

      {

            Aim->Astart\_time = Aim->systeam\_time,Aim->aim\_flag = 1;//记录自瞄开始时间，并进入新的单次自瞄

      }

      else{

            Aim->aiming\_time = Aim->systeam\_time - Aim->Astart\_time;//记录自瞄持续时间

      }

    }

   }

  else{//视觉NUC离线，无法被视觉控制自瞄

      vision\_other\_data.yaw\_success = AIM\_NO;

      Aim->aiming\_time = 0 , Aim->stay\_time = 0;

       Aim->Vision\_PID.i=0.00f;

  }

    VAL\_LIMIT(vision\_other\_data.yaw,-15.0f,15.0f);

}

2.3.5 云台硬件离线检测

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

    OfflineHandle\_Init(OFFLINE\_GIMBAL\_PITCH,            OFFLINE\_ERROR\_LEVEL,       100,         0);//5

    OfflineHandle\_Init(OFFLINE\_GIMBAL\_YAW,              OFFLINE\_ERROR\_LEVEL,       100,         0);//6

    OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR1,   OFFLINE\_ERROR\_LEVEL,       100,         1);//1

    OfflineHandle\_Init(OFFLINE\_FRICTION\_WHEEL\_MOTOR2,   OFFLINE\_ERROR\_LEVEL,       100,         2);

    OfflineHandle\_Init(OFFLINE\_TRIGGER\_MOTOR,           OFFLINE\_ERROR\_LEVEL,       100,         3);

    OfflineHandle\_Init(OFFLINE\_MAGAZINE\_MOTOR,          OFFLINE\_ERROR\_LEVEL,       100,         0);//4

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

 // OfflineHandle\_Init(OFFLINE\_CHASSIS\_INFO,            OFFLINE\_WARNING\_LEVEL,     100,         2);

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

    OfflineHandle\_Init(OFFLINE\_VISION\_INFO,           OFFLINE\_WARNING\_LEVEL,     1000,        0);

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

    /\*裁判系统\*/

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

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

    SoftwareTimerRegister(Vision\_RobotInfoUploadCallback, (void\*)NULL, 10);

    /\*VisionProtocol\_ParseHandler：

        vision\_rx\_handle视觉结构体的 UnpackDataSolve(); 解包处理回调函数\*/

    /\*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);

2.3.7 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();

    gimbal\_handle.vision\_tx\_data->data\_head = 0xAA;//头帧

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

    {

       gimbal\_handle.vision\_tx\_data->enemy\_color = Red;

    }

    else if (robot\_id > 1)

    {

        gimbal\_handle.vision\_tx\_data->enemy\_color = Blue;

    }

    else

    {

       gimbal\_handle.vision\_tx\_data->enemy\_color = AllColor;

    }

      gimbal\_handle.vision\_tx\_data->speed = sqrt(pow(console.chassis.vx,2) + pow(console.chassis.vy,2));//机体当前速度(合成速度)

      gimbal\_handle.vision\_tx\_data->yaw\_relative\_angle = gimbal\_handle.yaw\_motor.sensor.relative\_angle;//当前yaw轴偏角(相对上电时)

      gimbal\_handle.vision\_tx\_data->pitch\_relative\_angle = gimbal\_handle.pitch\_motor.sensor.relative\_angle;//当前pitch轴偏角(相对上电时)

      gimbal\_handle.vision\_tx\_data->bullet\_speed = shoot\_handle.shooter\_speed\_limit;//子弹速度

      gimbal\_handle.vision\_tx\_data->data\_tail = 0xA5;//尾帧

    //  VISION\_DATA\_CMD\_ID          = 0x0001

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

     Comm\_TransmitData(&vision\_tx\_handle, VISION\_PROTOCOL\_HEADER\_SOF, VISION\_DATA\_CMD\_ID, (uint8\_t\*)gimbal\_handle.vision\_tx\_data, sizeof(Comm\_RobotInfo\_t));

                                                  //有改动，本为 ROBOT\_DATA\_CMD\_ID ↑

    return 0;

}

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

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;

           }

**第三部分 自瞄算法**

3.1 背景与目标

智能遥控战车具备自动识别、跟踪并击打目标的功能。其设计旨在通过多学科融合激发学生的学习兴趣，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。在缺乏成熟工具的条件下，本项目鼓励学生创新性地应用基础知识和基本理论，综合考虑多种因素，通过领导多学科团队高效达成工程目标。同时，本项目还强调自主学习和终身学习的意识。在视觉系统方面，本项目需要进行全面的视觉系统设计，以实现精确的目标识别和跟踪。这一设计不仅要求学生掌握先进的视觉技术，还需具备整合多学科知识的能力，从而确保战车在复杂环境中能够精准执行任务。

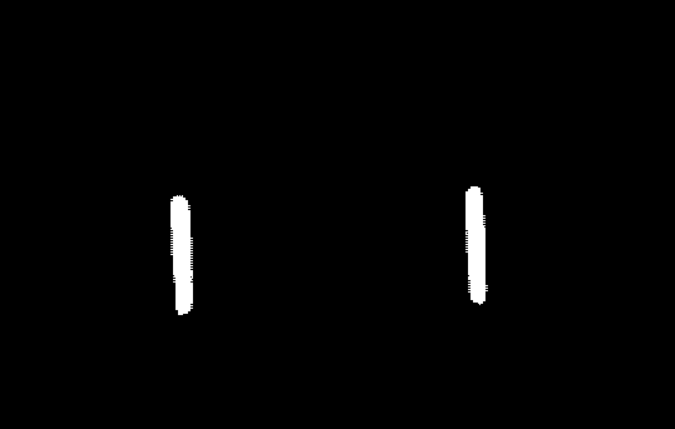
3.2 视觉自动瞄准算法设计

在视觉系统的设计中，机器人需要精准地命中敌方机器人身上四块装甲板，以造成有效伤害。这要求系统能够精确地获取目标装甲板的位置，从而计算发射时的云台姿态和调整射击参数。比赛中存在两种不同尺寸的装甲板：大型装甲板（尺寸为230mm x 127mm）和小型装甲板（尺寸为135mm x 125mm）。每块装甲板上都贴有数字标签（数字范围从1到5）以及一些特殊的图案。这些标签和图案不仅用于装甲板的识别，还帮助机器人区分不同的目标。同时，在装甲板的边缘装有两个灯条，这些灯条可以发出红色或蓝色的光。在比赛场景中，当己方团队属于红队时，己方机器人上的灯条将显示红色，而敌方机器人上的灯条则显示蓝色。相应地，如果己方团队为蓝队，则己方机器人上的灯条为蓝色，敌方灯条为红色。这种颜色标识有助于机器人快速区分敌我，从而更有效地瞄准敌方目标。装甲板上的数字标签也在标识不同机器人上发挥了重要作用。每个敌方机器人都配置了不同的数字标签，通过这些数字，机器人可以区分目标并制定相应的攻击策略。为了实现这一目标，需要对视觉系统进行全面设计。

3.3 设计方案

3.3.1 装甲板传统识别算法

装甲板的传统识别算法主要基于图像处理和计算机视觉技术，包含图像预处理、灯条检测、装甲板识别、数字标签识别、颜色识别、目标位置计算及多目标处理等多个步骤。首先，通过高分辨率摄像头实时采集图像并将其灰度化处理，结合高斯滤波或中值滤波去除噪声，提高图像的清晰度。接下来，利用Canny边缘检测或Sobel算子找到图像中的边缘，通过形态学操作进一步强化灯条特征，再通过轮廓提取算法筛选出符合装甲板特征的灯条轮廓。将相邻灯条配对后，根据长宽比和对称性识别装甲板候选区域。随后，裁剪出装甲板图像，通过对比度调整和二值化等方法增强数字标签，并使用OCR技术读取标签内容。



**图3-1 图像处理后效果**

这一整套方法不仅涵盖了传统的图像处理技术，还融入了机器学习与几何变换处理，确保在复杂动态的竞技环境中，机器人可以高效、精准地识别和定位敌方装甲板，从而提供重要的射击数据支持，提高比赛的整体表现和胜率。这种综合性的处理流程，不仅提高了系统的鲁棒性和识别率，同时也为未来的智能视觉系统提供了良好的理论和实践基础。



**图3-2 几何变换处理后效果**



**图3-3 传统识别效果**

// 尝试识别并返回最合适的装甲板，利用深度学习模型和预设参数

bool ArmorFinder::findTarget(cv::Mat src, int min\_threshold, cv::dnn::Net model,

    int whole\_h, double min\_confince, int num\_min\_threshold, double min\_iteration, int min\_area)

{

// 尝试查找符合阈值条件的真实装甲板

    if (findTrueArmors(src, min\_threshold, model, whole\_h, min\_confince,

        num\_min\_threshold, min\_iteration, min\_area)) {

        this->target = getBestArmors(Armors);

        return true;

    }

    return false;

}

// 用于车辆的装甲板识别函数

bool ArmorFinder::findCarTargets(cv::Mat src)

{

    if (findCarMayArmors(src,230,0.5)) {

        sortArmors(Armors);

        MatchArmors(Armors);

        if (FindBestChain(Armors)) {

            return true;

        }

    }

    return false;

}

// 查找可能是装甲板的部分,进行初筛

bool ArmorFinder::findMayArmors(cv::Mat src, int min\_threshold,double min\_iteration,

    int min\_area)

{

    Armors.clear();

    Armors.shrink\_to\_fit();

    if(findlights(src, min\_threshold, min\_iteration,min\_area))

    {

        //if (paired\_light\_strips(Lights)) {

            sortLights(Lights); // 灯条排序

            for (int i = 0; i < Lights.size(); i++)

            {

                for (int j = i + 1; j < Lights.size(); j++)

                {

                    Armor armor(Lights[i], Lights[j]);

                    if (determine\_if\_the\_light\_strip\_matches(armor)) {

                        Armors.push\_back(armor);

                    }

                }

            }

        //}

    }

    remove\_noise\_by\_position(Armors, Lights);

    return !Armors.empty();

}

// 在多个目标中获取最优目标

Armor ArmorFinder::getBestArmors(std::vector<Armor>& Armors)

{

    double min\_distance = 10000;

    int min\_id = 0;

    for (int i = 0; i < Armors.size(); i++) {

        cv::Point2f point = Armors[i].get\_MiddlePoint();

        double distance = get\_distance(point, cv::Point2f((int)whole\_w / 2.0,(int)whole\_h / 2.0));

        if (distance < min\_distance) {

            min\_distance = distance;

            min\_id = i;

        }

    }

    return Armors[min\_id];

}

void ArmorFinder::clear()

{

    target = Armor();

    Armors.clear();

    Armors.shrink\_to\_fit();

    Lights.clear();

    Lights.shrink\_to\_fit();

}

// 连接装甲板

void ArmorFinder::MatchArmors(std::vector<Armor>& Armors)

{

    if (Armors.size() == 1)return;

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

    {

        for (int j = i + 1, J = Armors.size(); j < J; j++)

        {

            if (Armors[i].R.center.x == Armors[j].L.center.x

            && Armors[i].R.center.y == Armors[j].L.center.y) {

                Armors[i].R\_\_ = &Armors[j];

                Armors[j].L\_\_ = &Armors[i];

            }

        }

    }

}

// 装甲板排序

void ArmorFinder::sortArmors(std::vector<Armor>& Armors)

{

    std::sort(Armors.begin(), Armors.end(), [&](Armor& i, Armor& j)->bool {

        return i.get\_MiddlePoint().x < j.get\_MiddlePoint().x;

        });

}

// 获取最优链式装甲板,以链表形式相连

bool ArmorFinder::FindBestChain(std::vector<Armor>& Armors)

{

    if (Armors.size() == 1)return true;

    std::vector<Armor> OutArmors;

    int longest = 0;

    Armor\* armor = NULL;

    for (auto& A : Armors) {

        if (A.L\_\_ == NULL && A.R\_\_ != NULL) {

            Armor\* armor\_ = &A;

            int length = 0;

            while (armor\_) {

                length++;

                armor\_ = armor\_->R\_\_;

            };

            if (length > longest) {

                longest = length;

                armor = &A;

            };

        }

    };

    if (longest == 0) return false;

    while (armor) {

        OutArmors.push\_back(\*armor);

        armor = armor->R\_\_;

    }

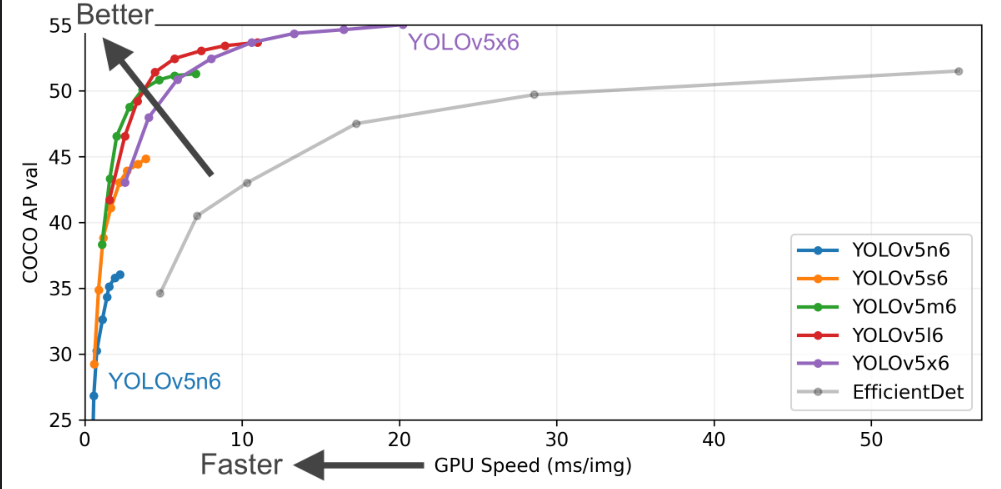
    Armors = OutArmors;

    return OutArmors.size() > 1;

}

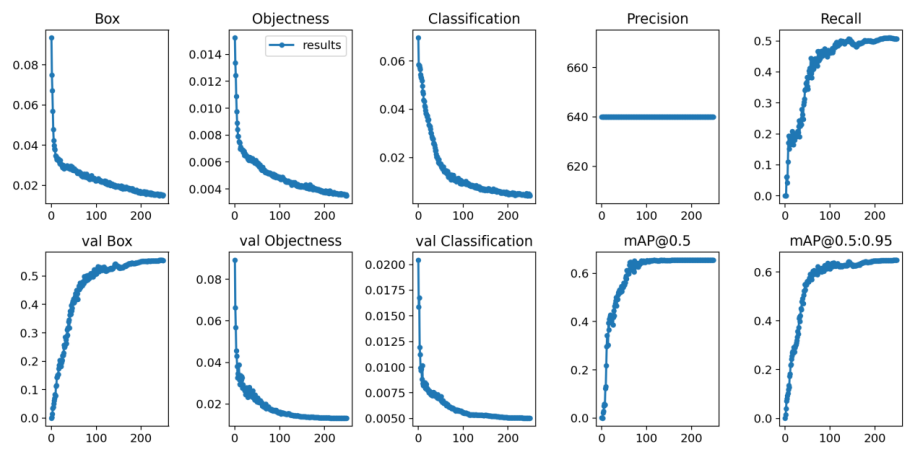
3.3.2 深度学习识别算法

装甲板的深度学习识别算法利用深度神经网络（DNN），特别是卷积神经网络（CNN），在图像识别领域的强大特性，实现高效、精确的目标识别和位置计算。这一过程主要包括数据采集与预处理、模型构建、训练与验证、实时检测与追踪等多个环节，融合了现代图像处理与机器学习技术。首先，使用高分辨率摄像头采集大量包含装甲板的图像和视频数据，并进行手动或自动标注，确保训练数据集的丰富性和多样性。为了提高模型的泛化能力，应用图像旋转、缩放、平移、翻转等数据增强技术扩展数据集。在模型构建阶段，设计适合装甲板识别的CNN架构，例如基于YOLO、SSD或Faster R-CNN等先进目标检测算法，并可能采用预训练模型进行迁移学习，进一步增强模型的性能。



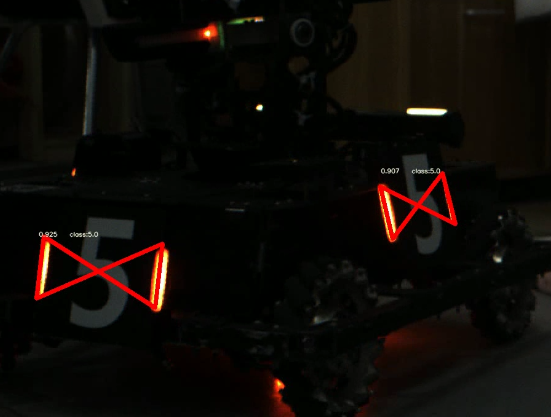
**图3-4 YOLOV5性能对比图**

训练阶段，将数据集划分为训练集、验证集和测试集，并设置合适的超参数如学习率、批量大小、优化器和损失函数，以确保训练过程的稳健性和模型评估的公正性。通过反向传播算法训练模型，并在验证集上测试，利用精确率、召回率、F1分数和平均精度均值等指标进行评估与优化。



**图3-5 模型性能数据变化曲线图**

不断采集新数据，通过在线学习或周期性重新训练来持续改进模型性能，确保适应环境变化和新目标特性。综合而言，深度学习识别算法赋予装甲板检测系统强大的自适应能力和特征提取能力，使其在复杂的竞技环境中能够高效准确地识别和定位目标，显著提升机器人的战斗力和智能化水平。这一方法不仅在实时动态场景中表现卓越，还为智能视觉系统的未来发展提供了坚实的理论和实践基础。



**图3-6 神经网络识别效果图**

    # 初始化关键变量，如保存目录、训练轮次、批处理大小等，并从命令行参数中提取

    save\_dir, epochs, batch\_size, weights, single\_cls, evolve, data, cfg, resume, noval, nosave, workers, freeze = \

        Path(opt.save\_dir), opt.epochs, opt.batch\_size, opt.weights, opt.single\_cls, opt.evolve, opt.data, opt.cfg, \

        opt.resume, opt.noval, opt.nosave, opt.workers, opt.freeze

    callbacks.run('on\_pretrain\_routine\_start')#回调函数

    # Directories下载路径

    w = save\_dir / 'weights'  # weights dir

    (w.parent if evolve else w).mkdir(parents=True, exist\_ok=True)  # make dir

    last, best = w / 'last.pt', w / 'best.pt'

    # Hyperparameters

    if isinstance(hyp, str):

        with open(hyp, errors='ignore') as f:

            hyp = yaml.safe\_load(f)  # load hyps dict

    LOGGER.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k, v in hyp.items()))

    opt.hyp = hyp.copy()  # for saving hyps to checkpoints

    # Save run settings

    if not evolve:

        yaml\_save(save\_dir / 'hyp.yaml', hyp)

        yaml\_save(save\_dir / 'opt.yaml', vars(opt))

    # Config    plots:True

    plots = not evolve and not opt.noplots  # create plots

    cuda = device.type != 'cpu'

    init\_seeds(opt.seed + 1 + RANK, deterministic=True)

    with torch\_distributed\_zero\_first(LOCAL\_RANK):

        data\_dict = data\_dict or check\_dataset(data)  # check if None

    #读取路径

    train\_path, val\_path = data\_dict['train'], data\_dict['val']

    nc = 1 if single\_cls else int(data\_dict['nc'])  # number of classes #4

    names = {0: 'item'} if single\_cls and len(data\_dict['names']) != 1 else data\_dict['names']  # class names   #

    is\_coco = isinstance(val\_path, str) and val\_path.endswith('coco/val2017.txt')  # COCO dataset

    # Model

    check\_suffix(weights, '.pt')  # check weights

    pretrained = weights.endswith('.pt')#是否以。pt结尾

    if pretrained:

        with torch\_distributed\_zero\_first(LOCAL\_RANK):#这是一个上下文管理器，用于在分布式训练中将第一个 GPU 的数据全部设置为零

        #初始化模型

            weights = attempt\_download(weights)  # 配置文件参数

        ckpt = torch.load(weights, map\_location='cpu')  # 加载模型权重文件，避免cpu内存泄露 # load checkpoint to CPU to avoid CUDA memory leak

        #创建模型对象

        model = Model(cfg or ckpt['model'].yaml, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create

        #决定是否排除 (‘anchor’) 键，这是是鲜艳框

        exclude = ['anchor'] if (cfg or hyp.get('anchors')) and not resume else []  # exclude keys

        #将模型转化为float形获取状态字典（state\_dict）

        csd = ckpt['model'].float().state\_dict()  # checkpoint state\_dict as FP32

        #交集操作

        csd = intersect\_dicts(csd, model.state\_dict(), exclude=exclude)  # intersect

        #将模型字典加载到model中

        model.load\_state\_dict(csd, strict=False)  # load

        LOGGER.info(f'Transferred {len(csd)}/{len(model.state\_dict())} items from {weights}')  # report

    else:

        model = Model(cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device)  # create

    amp = check\_amp(model)  # check AMP

    # Freeze 冻结模型层不更新数据

    freeze = [f'model.{x}.' for x in (freeze if len(freeze) > 1 else range(freeze[0]))]  # layers to freeze

    #更新梯度

    for k, v in model.named\_parameters():#named\_parameters是迭代器

        v.requires\_grad = True  # train all layers

        # v.register\_hook(lambda x: torch.nan\_to\_num(x))  # NaN to 0 (commented for erratic training results)

        if any(x in k for x in freeze):

            LOGGER.info(f'freezing {k}')

            v.requires\_grad = False

    # Image size 获取图像大小，确保网络大小大于32  384

    gs = max(int(model.stride.max()), 32)  # grid size (max stride)最大步幅

    imgsz = check\_img\_size(opt.imgsz, gs, floor=gs \* 2)  # verify imgsz is gs-multiple

    # Optimizer 优化器

    nbs = 64  # nominal batch size 每次期望的样本数

    accumulate = max(round(nbs / batch\_size), 1)  # accumulate loss before optimizing

    hyp['weight\_decay'] \*= batch\_size \* accumulate / nbs  # scale weight\_decay

    #获取权重缩放比例

    optimizer = smart\_optimizer(model, opt.optimizer, hyp['lr0'], hyp['momentum'], hyp['weight\_decay'])

    # Scheduler

    if opt.cos\_lr: #是否采用余弦退火的参数#XXX

        lf = one\_cycle(1, hyp['lrf'], epochs)  # cosine 1->hyp['lrf']

    else:

        lf = lambda x: (1 - x / epochs) \* (1.0 - hyp['lrf']) + hyp['lrf']  # linear

    scheduler = lr\_scheduler.LambdaLR(optimizer, lr\_lambda=lf)  # plot\_lr\_scheduler(optimizer, scheduler, epochs)

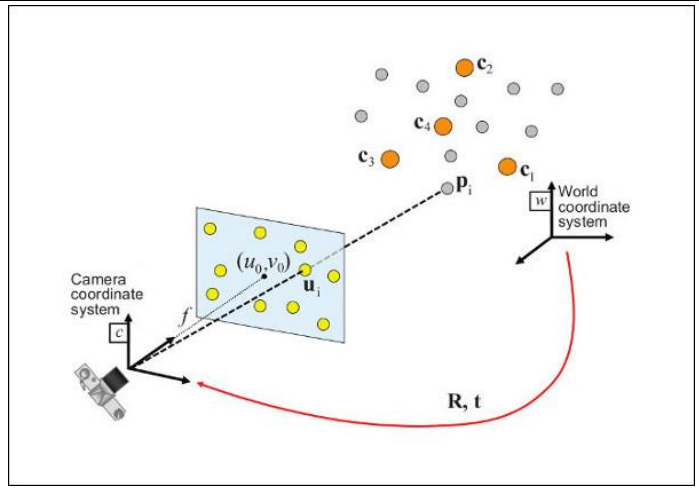
    # EMA #指数移动平均模型在训练过程中通过维护另一个模型的参数平均值，并使用该平均值作为预测时的模型。通过使用指数移动平均，可以降低模型的波动性，并有助于提高模型的泛化能力

    ema = ModelEMA(model) if RANK in {-1, 0} else None

3.3.3 三维重建算法

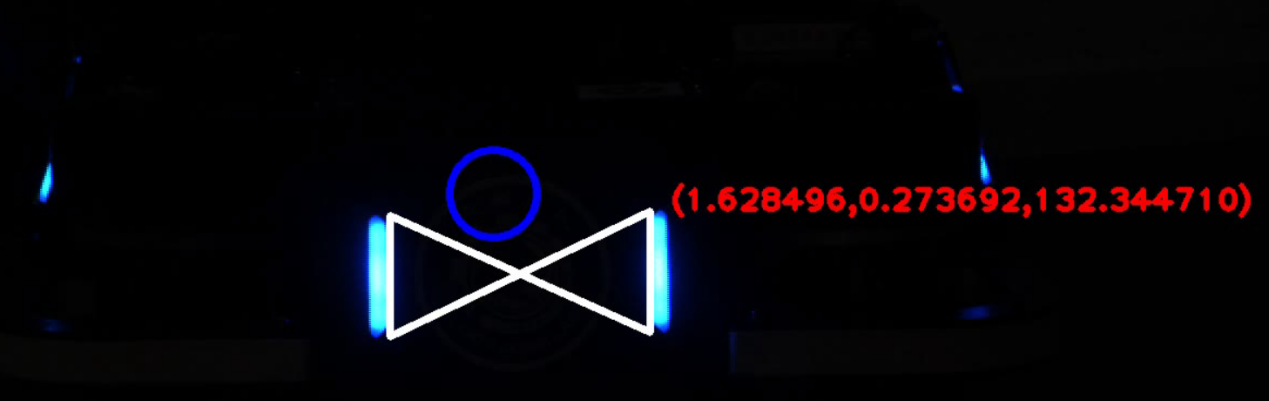
在自动瞄准系统中，基于深度学习的深度估计和单目PnP方法起到了关键作用。基于深度学习的绝对深度估计法虽然能提供深度信息，但其泛化能力较差且推理延时较高，不利于自动瞄准任务。

在这样的背景下，单目PnP方法表现出显著优势，它无需先验信息，仅通过单张RGB图像即可恢复目标的三维位置姿态。由于目标在视觉任务场景中是固定且可知的，单目PnP方法更适合此类应用。



**图3-7 PnP算法原理图**

此外，一般工业相机能够提供高帧率的RGB图像，并且可以根据不同的识别距离需求更换不同焦距的镜头。因此，视觉组选择了单目PnP方法作为自动瞄准系统的主要三维重建算法，以其高帧率、灵活性和可靠性的特点，为机器人系统提供了准确且高效的自动瞄准功能。



**图3-8 坐标解算效果图**

// 构造函数，初始化类的成员变量distCoeffs（畸变系数）、intrinsic（内参矩阵）和worldPoints（世界坐标点向量）

// 构造函数，初始化类的成员变量distCoeffs（畸变系数）、intrinsic（内参矩阵）和worldPoints（世界坐标点向量）

Coordinate3::Coordinate3(const cv::Mat& distCoeffs, const cv::Mat& intrinsic, const std::vector<cv::Point3d>& worldPoints)

    : distCoeffs(distCoeffs), intrinsic(intrinsic), worldPoints(worldPoints)

{

}

// 使用PnP算法计算旋转和平移向量，返回是否成功找到解。默认情况下同时计算Rvec和Tvec

bool Coordinate3::PnP(const std::vector<cv::Point2d>& PixelPoints)

{

    return solvePnP(worldPoints, PixelPoints, intrinsic, distCoeffs, Rvec, Tvec, 0, cv::SOLVEPNP\_P3P);

}

// 特殊化的PnP方法，仅计算并返回平移向量Tvec

bool Coordinate3::PnP(const std::vector<cv::Point2d>& PixelPoints, cv::Mat& Tvec)

{

    cv::Mat \_; // 未使用的变量，用于占位

    return solvePnP(worldPoints, PixelPoints, intrinsic, distCoeffs, \_, Tvec, 0, cv::SOLVEPNP\_P3P);

}

// 根据给定的新平移向量NewTvec，计算并返回新的坐标变换后的平移向量（仅平移）

cv::Mat Coordinate3::CoordinateTranslation(const cv::Mat& NewTvec)

{

    return NewTvec \* Tvec;

}

cv::Mat Coordinate3::CoordinateRotation(const cv::Mat& NewTvec)

{

    return NewTvec \* Tvec;

}

// 计算绕X轴旋转指定角度后的坐标变换矩阵，并应用于当前的平移向量Tvec

cv::Mat Coordinate3::CoordinateRotationX(const double X)

{

    auto pitch = X \* CV\_PI / 180; // 将角度转换为弧度

    return (cv::Mat\_<double>(3, 3) << 1, 0, 0,

        0, cos(pitch), -sin(pitch),

        0, sin(pitch), cos(pitch)) \* Tvec;

}

// 计算绕Y轴旋转指定角度后的坐标变换矩阵，并应用于当前的平移向量Tvec

cv::Mat Coordinate3::CoordinateRotationY(const double Y)

{

    auto yaw = Y \* CV\_PI / 180; // 将角度转换为弧度

    return (cv::Mat\_<double>(3, 3) << cos(yaw), 0, -sin(yaw),

        0, 1, 0,

        sin(yaw), 0, cos(yaw)) \* Tvec;

}

// 下面的方法重复了上面的逻辑，但直接接收额外的Tvec作为参数，可能是为了灵活性，允许基于不同的Tvec进行旋转操作

cv::Mat Coordinate3::CoordinateRotation(const cv::Mat& NewTvec, const cv::Mat& Tvec)

{

    return NewTvec \* Tvec;

}

cv::Mat Coordinate3::CoordinateRotationX(const double X, const cv::Mat& Tvec)

{

    auto pitch = X \* CV\_PI / 180;

    return (cv::Mat\_<double>(3, 3) << 1, 0, 0,

        0, cos(pitch), -sin(pitch),

        0, sin(pitch), cos(pitch)) \* Tvec;

}

cv::Mat Coordinate3::CoordinateRotationY(const double Y, const cv::Mat& Tvec)

{

    auto yaw = Y \* CV\_PI / 180;

    return (cv::Mat\_<double>(3, 3) << cos(yaw), 0, -sin(yaw),

        0, 1, 0,

        sin(yaw), 0, cos(yaw)) \* Tvec;

}

// 计算绕X轴逆旋转（即反向旋转）后的新坐标变换矩阵，并应用于给定的Tvec

cv::Mat Coordinate3::BackCoordinateRotationX(const double X, const cv::Mat& Tvec)

{

    auto pitch = X \* CV\_PI / 180;

    cv::Mat pitch\_ = (cv::Mat\_<double>(3, 3) << 1, 0, 0,

        0, cos(pitch), -sin(pitch),

        0, sin(pitch), cos(pitch));

    return pitch\_.inv() \* Tvec; // 使用逆矩阵进行逆旋转操作

}

// 将三维相机坐标系下的点转换为像素坐标系下的点

cv::Point2d Calculate::ConvertPixelCoordinate(const cv::Mat& CameraPoint)

{

    // 将输入的Mat转换为Point3d形式

    std::vector<cv::Point3d> objectPoints;

    objectPoints.push\_back(cv::Point3d(static\_cast<double>(CameraPoint.at<float>(0)), static\_cast<double>(CameraPoint.at<float>(1)), static\_cast<double>(CameraPoint.at<float>(2))));

    // 使用projectPoints函数进行投影变换

    std::vector<cv::Point2d> imagePoints;

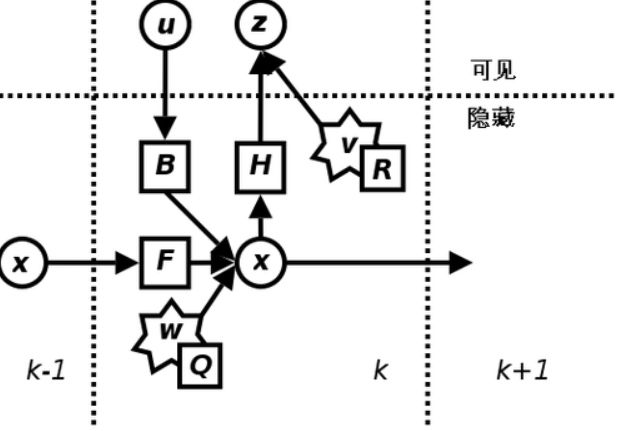
    cv::projectPoints(objectPoints, cv::Vec3d(0, 0, 0), cv::Vec3d(0, 0, 0), intrinsic, distCoeffs, imagePoints);

    return imagePoints[0]; // 返回转换后的像素坐标点

}

3.3.4 状态估计算法

在本项目中，针对运动目标的预测至关重要，尤其是对目标运动状态（如速度）的准确估计。然而，除了位置姿态外，其他状态无法直接由系统观测到，因此需要使用状态估计器从观测数据中推断系统状态。常见的状态估计算法包括卡尔曼滤波、扩展卡尔曼滤波、无迹卡尔曼滤波、粒子滤波以及多变量状态估计技术。考虑到视觉任务中的目标机器人通常进行旋转和平移运动的组合，而项目要观测和预测的是目标机器人上搭载的装甲板，为了实现最佳击打效果，必须估计目标机器人的整体运动状态.此时将观测量转换为目标的位移和旋转角以及对应的移速和转速即可。



**图3-9 卡尔曼滤波原理图**

// 构造函数初始化扩展卡尔曼滤波器(Extended Kalman Filter)类的对象

ExtendedKalmanFilter2::ExtendedKalmanFilter2(

    const NonlinearFunc& f, const NonlinearFunc& h, const JacobianFunc& Jf,

    const JacobianFunc& Jh, const Eigen::MatrixXd& Q, const Eigen::MatrixXd& R,

    const Eigen::MatrixXd& P0)

    : f(f), // 非线性状态转移函数

    h(h), // 非线性观测模型函数

    Jf(Jf), // 状态转移函数的雅可比矩阵计算函数

    Jh(Jh), // 观测模型的雅可比矩阵计算函数

    Q(Q), // 过程噪声协方差矩阵

    R(R), // 观测噪声协方差矩阵

    P\_post(P0), // 后验估计的协方差矩阵初始值

    n(Q.rows()), // 系统状态维度

    I(Eigen::MatrixXd::Identity(n, n)), // 单位矩阵，维度为n\*n

    x\_pri(n), // 预测状态向量，初始化为n维

    x\_post(n) // 后验状态向量，初始化为n维

{

}

// 设置滤波器的初始状态估计

void ExtendedKalmanFilter2::setState(const Eigen::VectorXd& x0) {

    x\_post = x0; // 将输入的状态向量设为当前的后验状态估计

}

// 预测步骤，根据非线性动态模型推进状态估计

Eigen::MatrixXd ExtendedKalmanFilter2::predict()

{

    x\_pri = f(x\_post); // 使用非线性状态转移函数预测下一个状态

    F = Jf(x\_post); // 计算预测状态的雅可比矩阵

    P\_pri = F \* P\_post \* F.transpose() + Q; // 计算预测协方差矩阵

    x\_post = x\_pri; // 临时预测状态成为新的后验状态（通常这一步在更新步骤之后，此处可能需调整）

    P\_post = P\_pri; // 同上，临时协方差矩阵成为新的后验协方差矩阵

    return x\_pri; // 返回预测的状态

}

// 更新步骤，融合测量值以修正状态估计

Eigen::MatrixXd ExtendedKalmanFilter2::update(const Eigen::VectorXd& z)

{

    H = Jh(x\_pri); // 计算观测模型的雅可比矩阵

    K = P\_pri \* H.transpose() \* (H \* P\_pri \* H.transpose() + R).inverse(); // 计算卡尔曼增益

    x\_post = x\_pri + K \* (z - h(x\_pri)); // 修正状态估计

    P\_post = (I - K \* H) \* P\_pri; // 更新后验协方差矩阵

    return x\_post; // 返回更新后的状态估计

}

**第四部分 硬件设计**

4.1 全向轮战车基础硬件设计

4.1.1 全向轮战车器件

**表4-1 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 | 8 | 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 | 4 | 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 | 2 | 拨盘动力输出 |

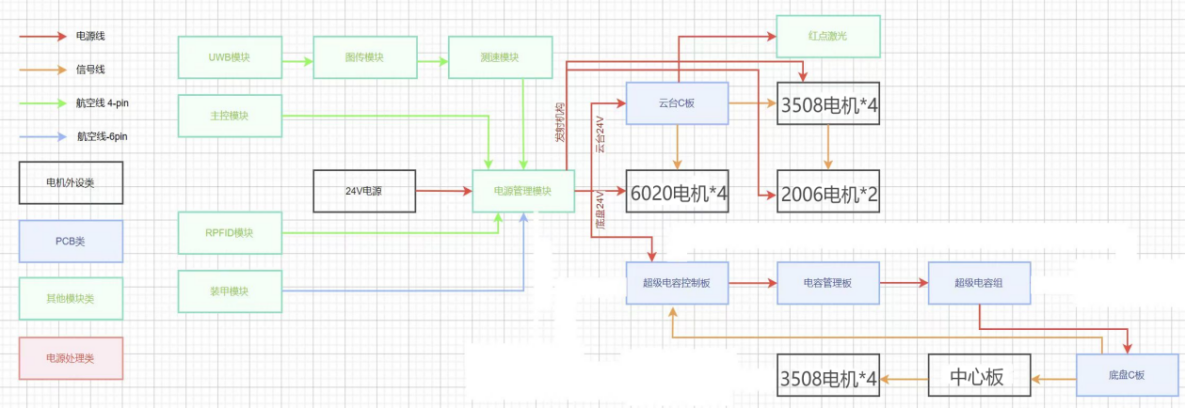
**表4-2 全向轮战车控制器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| RoboMaster开发板 C 型 | 3 | 用于控制底盘和云台 |
| RoboMaster C620无刷电机调速器 | 8 | 用于控制RoboMaster M3508 直流无刷减速电机 |
| RoboMaster C610无刷电机调速器 | 2 | 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4-3 全向轮战车其余器件**

|  |  |  |
| --- | --- | --- |
| 器件名称 | 数量 | 用途 |
| 分电板 | 3 | 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 | 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 |

机器人硬件拓扑图可简化分为底盘和下云台与上云台三部分，其中的板间通信为 CAN 通信，通过导电滑环完成底盘与云台的通信连接。



**图4-1 全向轮战车电路拓扑图**

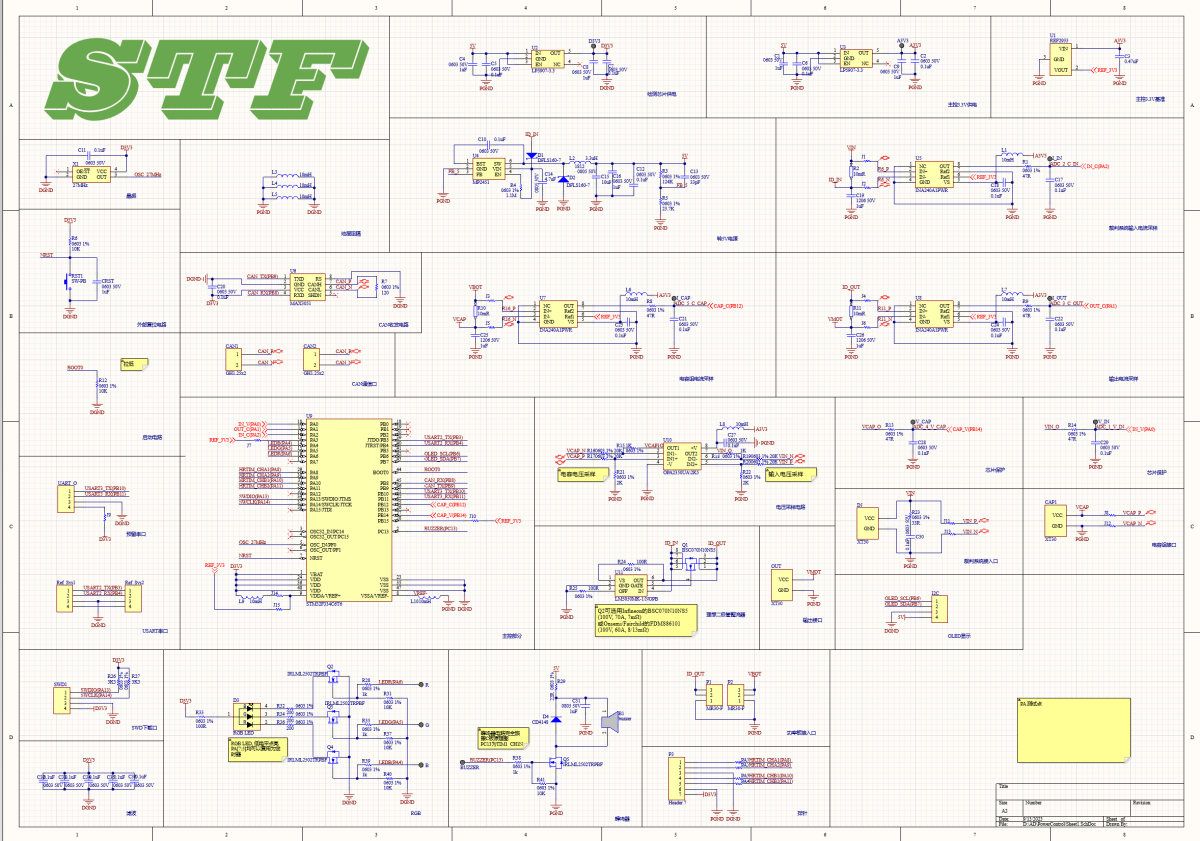
4.2 **全向轮战车**特殊硬件设计

4.2.1 总体方案

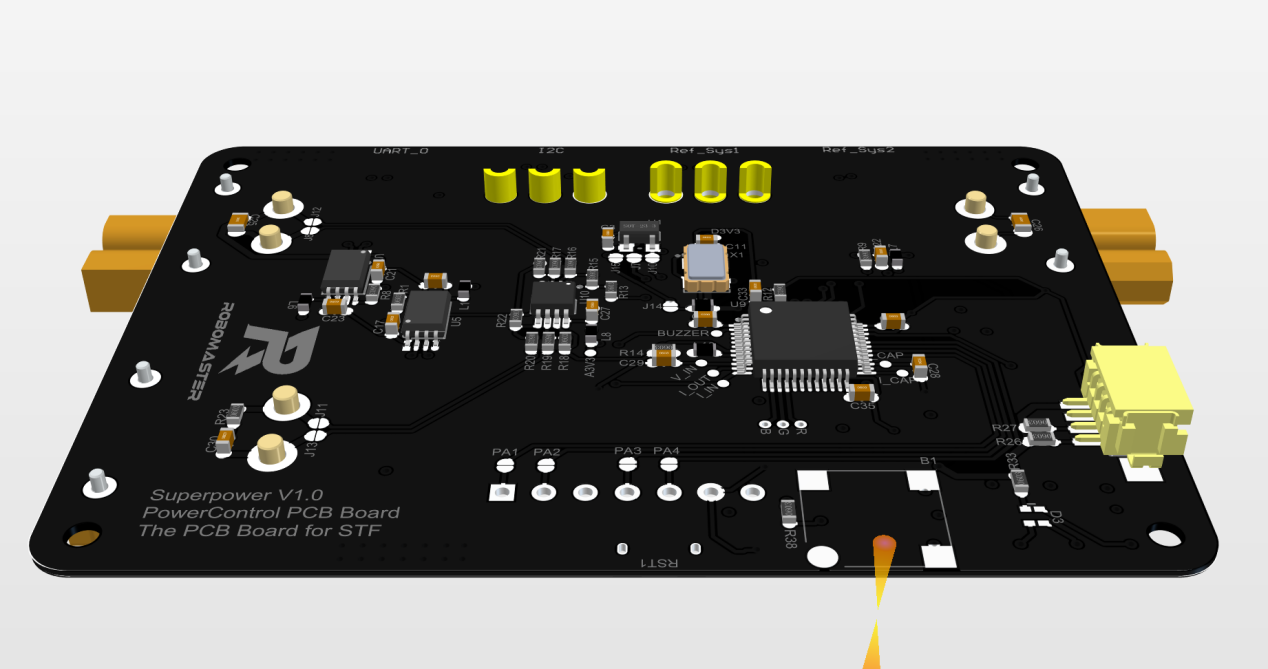
为遥控战车提供额外功率，实现超功率运作，用作后备能源，为战车底盘提供更强劲驱动力。通过超级电容控制板对超级电容组充放电，将战车在没有全功率输出时多余的功率储存，当需要时，开启超级电容释放储存的能量从而实现超功率输出。

这里模块采用四层板和二层板的结合方案。两板之间通过两个MR30和一个7PIN排针连接 （MR30过大电流，排针过PWM信号）主要功率器件放在二层板上，控制部分和采样部分的电 路安置在四层板上。大面积、少器件的二层板散热性良好，四层板可以在相同面积下放置更多的电子元件，同时可以保证接地地层的完整性，提高PCB抗干扰能力。

4.2.2 控制部分

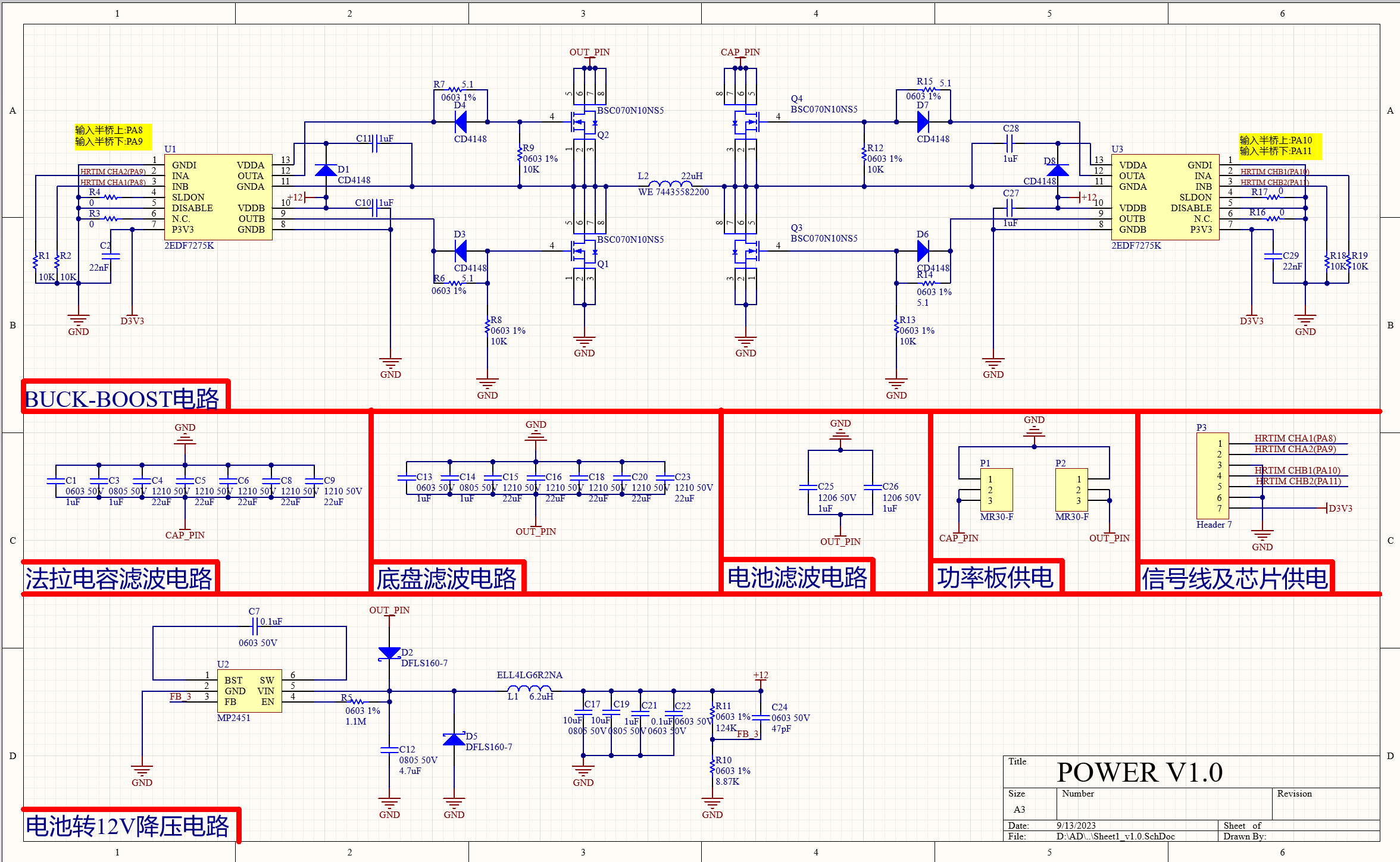


**图4-2 超级电容控制板原理图**

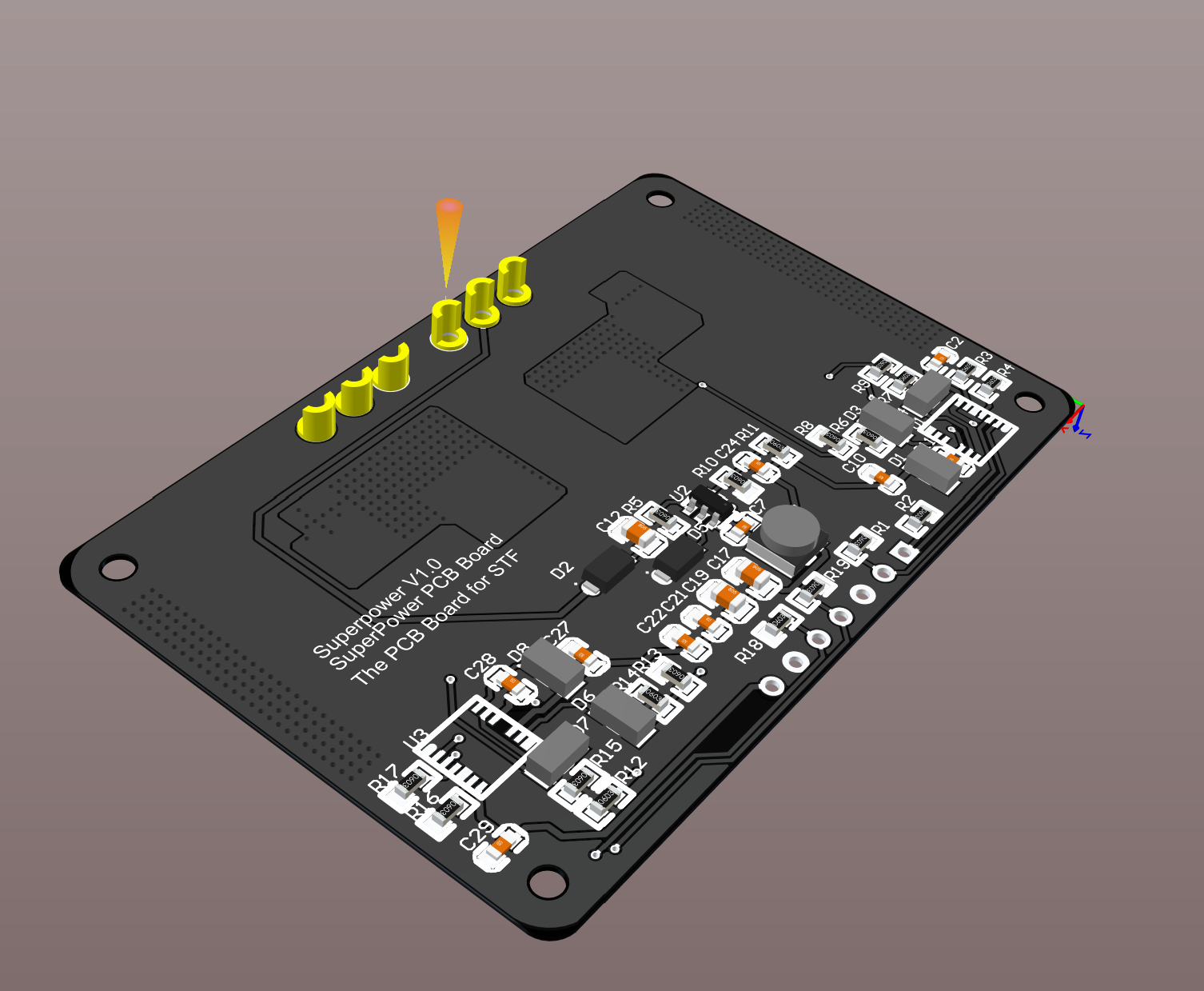


**图4-3 超级电容控制板模型图**

4.2.3 功率部分



**图4-4 超级电容功率板原理图**



**图4-5 超级电容功率板模型图**

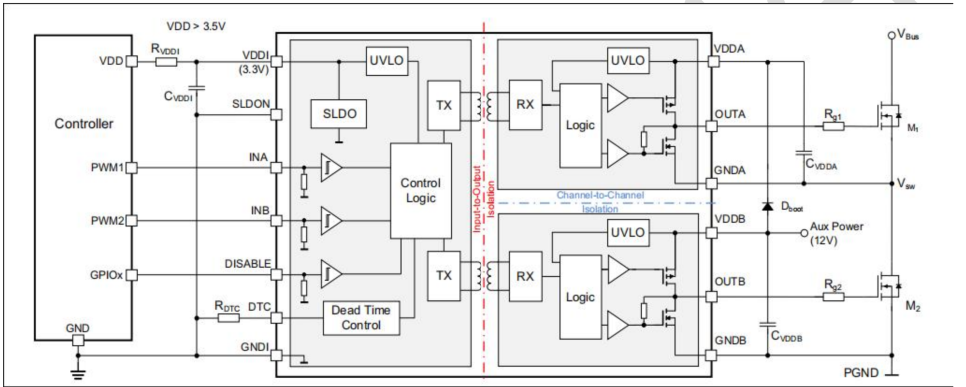
4.2.4 关键器件选型

MCU：意法半导体公司的 STM32F334C8T6 芯片，该芯片采用 Cortex-M4 内核，32位CPU和 FPU(72mhz max)，单周期乘法和硬件分割DSP指令，具有64KB Flash与16KB SRAM。内部拥有 2个ADC, 3个DAC, 3个comp、一个运放和1个4通道HRTIM；足够满足需求，虽然运算性能和片上资源少于STM32G474，但不缺货而且价格相对便宜。

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。

电流以及电压采样芯片：为了实现充电、放电的精准功率监测，我们使用 STM32 自带的高速 16 位 ADC 配合INA240\_A1PWR与OPA2350UA\2K5外设实现对 3路电流采样窗口、2路电压采样窗 口进行自动循环采样和数据转移。期间还需要配置 MPU 保护区域、cache 禁止缓存区域、 ld 链接文件等。

IC驱动：英飞凌2EDF7275K——一款快速双通道隔离MOSFET门驱 动芯片，通过无芯变压器(CT)技术提供功能(2EDFx)或增强(2EDSx)输入到输出隔离。由于高 驱动电流，优秀的共模抑制和快速的信号传播，2EDi特别适合于驱动在快速开关电源系统中 高压MOSFET。37ns延迟，隔离能力1.5 kVDC~10 ms，能够避免MOS击穿导致灾难性后果。



**图4-6 IC封装图**

高压电流MOS:本方案选择英飞凌公司的BSC070N10NS5 N型MOSFET进行H桥的构建。该MOSFET 有着100V的额定电压和80A的额定电流，死区时间仅在13-24ns。其PQFN8的封装也极大的解 决了散热问题，使得超级电容的瞬时工作最高温度在140℃以下（无散热器裸板测试情况下）。

驱动供电：采用 DCDC 降压芯片MP2451降压到12V为驱动芯片供电。

电感：本方案选用22uH的一体式电感 WE 74435582200。

4.2.5 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。