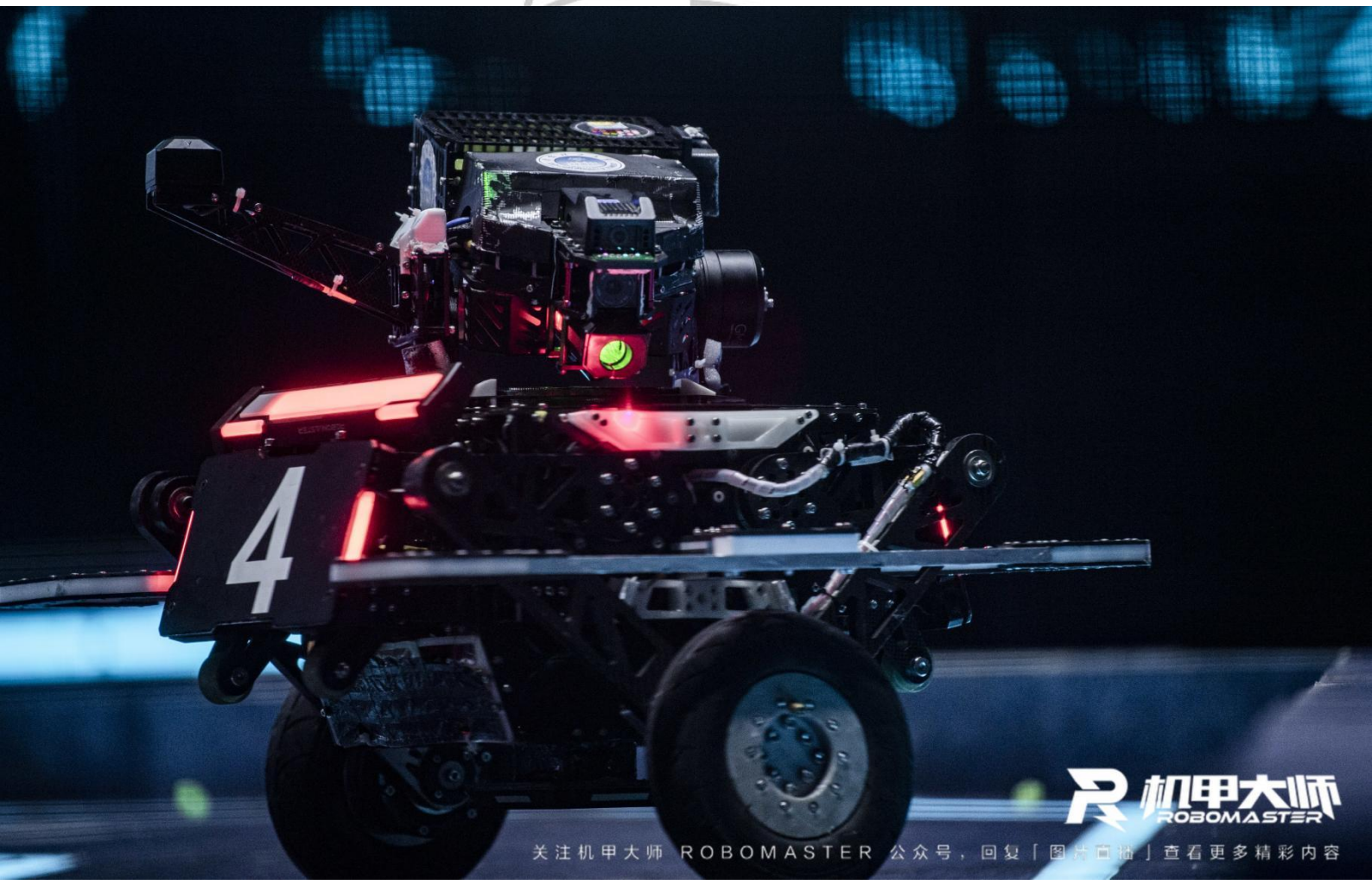
华南理工大學
South China University of Technology

华南虎战队 23 赛季轮足步兵电控代码框架设计思想分享

——华南虎 23 赛季平衡步兵电控 余俊晖 张至睿



摘要

无论是仿真与实车代码之间的相互移植、将旧车代码迁移到新车以及同类兵种的车间代码同步，都是一个花费时间并且容易出现 bug 的工作。为了减少这相关的开发时间，并且增强整车代码的模块化程度、提高可移植性等等，笔者提出了一套关于代码结构分层+模块抽象化的代码框架，并在此进行分享。

关键词：电控，代码框架，仿真

目录

摘要 2
关键词： 2
1. 背景和设计思想 4
2. 代码结构分层 4
3. 抽象电机模块 6
4. 抽象 IMU 模块 9
5. 总结思考 9
附录 1——中间层+抽象库代码仓库网址： 10
附录 2——抽象 IMU 模块代码： 10

1. 背景和设计思想

因为轮足步兵具有使用仿真软件验证控制算法可行性的新需求，在将仿真代码转化为实车代码时却发现工作量庞大且容易出错；将使用 3508 作为轮子电机的第一代平衡步兵代码，迁移到使用 9025 电机的二代平衡步兵上时的时间花销不少；在车间代码同步方面，因为代码与车辆本身强耦合，导致代码同步只能人工迁移同步.....

为了解决以上痛点，提高开发效率，使代码更加模块化，因此笔者设计了这一代码结构分层+模块抽象化的代码框架。

以下为该框架应用在 23 赛季轮足步兵上的示例图：

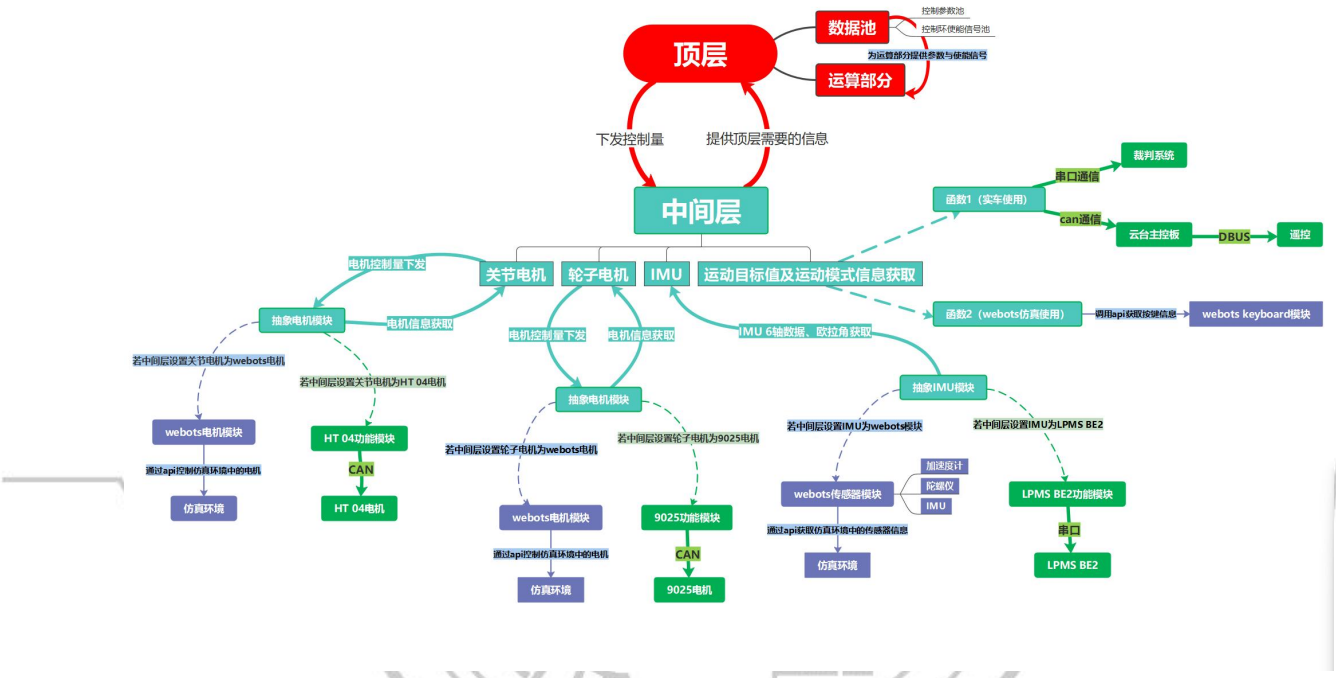


图 1 华南虎 23 赛季轮足步兵代码结构图

2. 代码结构分层

该设计将整车代码分为顶层、中间层、底层：

1. **顶层：**负责整车核心的控制算法运行，坐标系与电机转动方向均为“理想化”的，与 实车完全解耦，可以照搬不动的放到仿真软件、其他同类车上运行。

   为了方便在车间同步顶层代码，将顶层分为“数据池”与“计算部分”：

   - **数据池：**负责存储控制参数如 PID 参数，以及控制环的使能信号（若某辆车在顶层有某个特殊的策略，而另外一辆不需要，可以通过使能信号的设置来解决）
   - **计算部分：**为控制算法的具体计算流程。

   在代码管理时，将顶层的“计算部分”设为一个单独的仓库，作为 submodule 加入在整车的总仓库中，而“数据池”创建好模板文件以统一格式，每一辆车的总仓库中放入设置好数值的数据池模板文件，则顶层代码在车间同步将会变得简单。

2. **中间层：**用于对接顶层与底层，将底层的电机转速、电流以及 IMU 数据，转化为顶层的理想坐标系下的对应数据后，通过函数接口将理想数据传递给顶层；同时也负责为顶层提供电机控制指令，将顶层的电机控制指令对接到底层的电机中。

3. **底层：**对于实车来说，底层是对于实车上的 IMU、电机的数据收发的功能实现部分；对于 webots 等仿真软件来说，底层是调用软件提供的 API 来获取仿真软件中的 IMU、电机数据；

对于整车来说，底层是战队已经封装好的功能模块，而中间层只需为一个平台编写一次后便无需再改动，后续调试均为对于顶层代码的更改。因此，相对于不分层时的代码，分层的代码只需花费短暂的时间编写一个中间层后，无论顶层代码经历了多少改动，都可以在实车和仿真软件之间轻松移植。

以下为轮足步兵中间层的结构：

```
class MyMiddlewareClassdef
{
private:
#pragma pack(1)
  struct recDataStructdef
  {
    /*左右轮当前速度*/
    float linerSpeed = 0;
    /*速度向量*/
    float target_speed_y = 0;
    float target_speed_z = 0;
    float target_height = 0.14;
    /*功率系数*/
    float power_limit_scale = 0.;
    /*向心力系数（v^2/r)*/
    float centripetal_force_scale = 0.f;
    chassis_state_structdef chassis_state = {};
  } rec_data;
#pragma pack()
public:
  MyMiddlewareClassdef();
  void init(QueueHandle_t _USART_TxPort, uint8_t _port_num[2], LPMS_BE2_Typedef *_lmps);
  //连接状态控制
  void Link_Check();

  void setJointTorque(float tarTorque[4]);
  void sendDigitalBoard(float totalTorque[2], float _angularSpeed, uint16_t _is_balance, uint16_t _is_reset, uint16_t
_is_falldown, uint16_t _is_overturn, float _f_angle, float _b_angle, float _end_x, float _end_y);

  //关节初始化
  void jointInit(MotorHT04Classdef _motor[4]);
  //关节自动复位
  void jointReset();
  //关节自动标定
  void setJointOffset();
  //关节指令清零
  void jointCommandClear();

  float *getMotorAngle();
```

```
  float *getMotorTorque();

  void processRecData(Send2ControllerStructdef *recData);
  const recDataStructdef &getRecData() { return rec_data; }

  LinearDataStructdef *getAccData();
  AngularDataStructdef *getEulerData();
  AngularDataStructdef *getAngleVelData();
  abstractMotor<MotorHT04Classdef> jointMotor[4];

  int32_t crcErrCnt = 0;
private:
  abstractIMUClassdef<LPMS_BE2_Typedef> imu;

  RecFromControllerStructdef sendMessage;
  float jointAngle[4];
  float jointTorque[4];

  QueueHandle_t USART_TxPort;
  USART_COB Usart_TxCOB;
  uint8_t port_num[2];

  uint8_t LinkCount = 90;
  uint8_t continueError = 0;
  int16_t packetLossRate = 0;
};
```

3. 抽象电机模块

为了方便中间层的编写以及具体模块的切换，笔者想出了一个分层设计的伴生设计思想——模块抽象。就如同分层思想是为了将顶层算法与执行平台解耦一般，模块抽象是为了将代码中底层的功能模块与实际设备解耦。其主要工作为 1. 接口统一；2. 将模块的运行方向/坐标系抽象；

以下为抽象电机的实现代码：

```
/* 匿名命名空间，使基类无法被外部链接 */
namespace
{
    /**
     * @brief 电机抽象基类，用于对接顶层计算得控制量与底层电机的实际输出
     * 在继承类中，加入电机类的指针成员，绑定电机变量，定义几个纯虚函数，即可正常使用
     */
    template <class motorType>
    class abstractMotorBase
    {
    protected:
        //通过纯虚函数的形式，强制要求每个电机都二次封装对应的 api
        inline virtual float getRawMotorTotalAngle() = 0;//获取电机原生多圈角度数据，单位为 degree
        inline virtual float getRawMotorAngle() = 0;//获取电机原生单圈角度数据，单位为 degree
        inline virtual float getRawMotorSpeed() = 0;//获取电机原生速度数据，单位 RPM
        inline virtual void setRawMotorCurrentOut(float out) = 0;//下发电机实际电流输出
```

```
    inline virtual bool isInit()
    {
        if(this->motor == nullptr)
            return 0;
        else
            return 1;
    }; // 检测是否导入了电机指针、队列句柄等等，防止未导入时，因为指针为空指针，调用函数而进入硬件中断

public:
    motorType *motor = nullptr;
    inline void bindMotor(motorType *_motor) { motor = _motor; }//绑定电机指针

    //提供给顶层的接口，将电机原生多圈角度进行单位转换、极性对齐、基础角度偏移，再返回给顶层
    inline virtual float getMotorTotalAngle()
    {
        if (isInit() == 0)
            return 0;
        return getRawMotorTotalAngle() * angle_unit_convert * Polarity + baseAngle;
    }
    //提供给顶层的接口，将电机原生单圈角度进行单位转换、极性对齐、基础角度偏移，再返回给顶层
    inline virtual float getMotorAngle()
    {
        if (isInit() == 0)
            return 0;
        return getRawMotorAngle() * angle_unit_convert * Polarity + baseAngle;
    }
    //提供给顶层的接口，将电机原生速度进行单位转换、极性对齐，再返回给顶层
    inline virtual float getMotorSpeed()
    {
        if (isInit() == 0)
            return 0;
        return getRawMotorSpeed() * speed_unit_convert * Polarity;
    }
    //提供给顶层的接口，将顶层下发的力矩，进行单位转换、极性修正，再下发到底层
    inline virtual void setMotorCurrentOut(float out)
    {
        if (isInit() == 0)
            return;
        setRawMotorCurrentOut(out * out_unit_convert * Polarity);
    }

    // 极性，将电机实际转动方向的正负，对齐到人为设置的方向，自行设置
    // 对于绝大部分电机来说，输出正力矩时，速度增加，角度增加，所以三者极性是具有统一性的
    // 因此，只需要设置一个极性即可
    float Polarity = 1;

    float speed_unit_convert = 1; // 获取数据的单位转换，自行设置

    float angle_unit_convert = 1; // 获取数据的单位转换，自行设置
    float baseAngle = 0;          // 角度基础值，会在极性对齐、单位转换后，再加上这个值

    // 如果需要将顶层下发的力矩，转换为电流，再用电机类下发，就需要设置这个值
    // 如果不需要单位转换，就不用管
    float out_unit_convert = 1;
};
}
```

以下为将底层的 9025 功能模块适配到抽象电机模块的代码示例：

```
/**
 * @brief 电机抽象模板类，仅仅为了可以特化而写，并无实际作用
 * 用模板是为了类名统一，实现泛化
 * 模板主类不写实现，是为了避免在传入没有特化过的电机类型时，会出现无法设想的错误
 * @tparam motorType
 */
template <class motorType>
class abstractMotor
{
};
/**
 * @brief 模板特化，MF9025_v2 电机抽象类
 */
template <>
class abstractMotor<MotorMF9025v2Classdef> : public abstractMotorBase<MotorMF9025v2Classdef>
{
protected:
    inline virtual float getRawMotorTotalAngle()
    {
        return motor->getData().totalAngleLocal;
    }
    inline virtual float getRawMotorAngle()
    {
        return motor->getData().singleAngle;
    }
    inline virtual float getRawMotorSpeed()
    {
        return motor->getData().speed;
    }
    inline virtual void setRawMotorCurrentOut(float out)
    {
        motor->iqCloseControl_Current(out);
    }
};
```

通过构建一个抽象电机基类与抽象电机模板类，模板参数为已经封装好的电机模块类名，通过导入电机类成员的指针，使用“继承+纯虚函数+模板特化”的方式，强制对于以上说明的电机必备的几种功能封装二次封装成统一接口。由以上代码示例可以看出，将原有的底层电机功能模块适配到抽象电机模块只需很小的工作量。

使用该抽象电机模块的优势有三点：

1. **电机模块的更换变得简单：**比如使用 3508 的平衡底盘与使用 9025 的平衡底盘之间的中间层实现部分是一样的，只需将中间层中的 `abstractMotor<3508 电机> wheelMotor[2]` 变更为 `abstractMotor<9025 电机>  wheelMotor[2]`，即可完成代码迁移。
2. **将电机方向抽象化：**以底盘电机为例，通常电机的正方向为逆时针转动，这导致了一个问题——底盘向前运动时，右轮电机的输出、转速为负值，左轮却为正值，这无疑是十分“别扭”的，而且也会导致因为疏忽了电机极性导致 bug 的现象存在。通过抽象电机模块的极性设置，会在顶层与底层通过中间层进行数据交互时自动进行极性修正，减少了使用者的开发成本、写 bug 的可能性。

而且将电机方向抽象出来，有助于顶层的使用理想方向与坐标系进行计算，进一步与现实解耦，比如在平衡步兵建模中的理想坐标系为轮子电机施加正力矩时会使底盘往水平正方向加速，通过抽象电机的极性修正后，顶层处理的数据均为建模时的理想坐标系的数据，即减少了开发成本，也降低了出 bug 的概率。

3. **数据转化：**顶层与底层通过中间层进行数据交互时，抽象电机会自动进行单位转换、偏置修正。通过提前设置好的角度单位转换系数，顶层获取的角度数据可以方便的在 degree 与 rad 之间进行变更，且电机的零点设置也变得简单。

对于速度单位转换系数，将其设置为电机减速比的倒数后，即可方便的获得电机出轴的转速。比如将轮子电机为 19 减速比的舵轮底盘中间层迁移到 14 减速比的舵轮底盘中间层，则只需变更该系数即可完成迁移。

而输出单位转换系数，则是用于设置力矩输出对应到电机实际输出的系数，也是为了方便切换电机时使用。

4. 抽象 IMU 模块

除了抽象电机外，还设计了抽象 IMU 模块，用于将 IMU 的实际坐标系对接到顶层的理想坐标系，比如实际 IMU 的 roll 轴为建模时的 pitch 轴、因为 IMU 方向装反导致其建模时的极性与实际数据极性相反等等。

其设计理念与抽象电机的设计理念相同，因此代码放置在附录中，并不做过多解释。

5. 总结思考

通过该框架的搭建，使得代码的模块化程度提高，各模块的更换也变得容易，能够减少许多开发时间。因此该框架的推广也作为了华南虎 24 赛季电控组的重要推进方向之一，后面将计划为几种经典底盘都搭建一个中间层并加入开源当中，但该框架仍有不少可以改进的地方，欢迎大家为此框架提供建议，使单片机上的机器人控制程序开发能够越来越接近使用 ROS 那般的模块化与高效。

附录 1——中间层+抽象库代码仓库网址：

WhiteNightyu/23_WheelManipulator_Infantry_code_struct: RoboMaster 华南理工大学华南虎战队 23 赛季轮足步兵中间层+抽象模块代码开源 (github.com)

附录 2——抽象 IMU 模块代码：

```
/*线性参数*/
struct LinearDataStructdef
{
    float x;
    float y;
    float z;
};
/*角度参数*/
struct AngularDataStructdef
{
    float pitch;
    float yaw;
    float roll;
};

namespace abstractIMU
{
    /**
     * @brief 线性加速度极性结构体
     *
     */
    struct linerPolarityStruct
    {
        int8_t x = 1;
        int8_t y = 1;
        int8_t z = 1;
    };

    /**
     * @brief 角度极性结构体
     *
     */
    struct anglePolarityStruct
    {
        int8_t pitch = 1;
        int8_t roll = 1;
        int8_t yaw = 1;
    };
    /**
     * @brief 抽象加速度坐标结构体
     *
     */
    struct abstractAccCoordinateStruct
    {
        uint8_t x = 0;
        uint8_t y = 1;
        uint8_t z = 2;
    };
```

```
/**
 * @brief 抽象欧拉角坐标结构体
 *
 */
struct abstractEulerCoordinateStruct
{
    uint8_t Pitch = 0;
    uint8_t Yaw = 1;
    uint8_t Roll = 2;
};
/**
 * @brief IMU 欧拉角世界（现实）坐标系枚举
 *
 */
enum imuEulerCoordinate
{
    imuWorldPitch = 0,
    imuWorldYaw = 1,
    imuWorldRoll = 2
};
/**
 * @brief IMU 直角坐标系世界（现实）坐标系枚举
 *
 */
enum imuAccCoordinate
{
    imuWorldAccX = 0,
    imuWorldAccY = 1,
    imuWorldAccZ = 2
};
/**
 * @brief IMU 抽象类基类，完成了绝大部分的逻辑操作
 * 在继承类中，加入 imu 类的指针成员，绑定 imu 变量，定义 9 个获取 imu 数据的纯虚函数，即可正常使用
 */
class abstractIMUBaseClassdef
{
private:
    LinearDataStructdef accData;
    AngularDataStructdef eularData, angleVelData;

    // 抽象坐标系结构体，用于存储人定坐标系与 imu 坐标系的对应关系
    abstractAccCoordinateStruct abstractAccData;
    abstractEulerCoordinateStruct abstractEularData;

    // 获取 imu 在世界（现实）坐标系下的加速度
    inline virtual float getACCX() = 0;
    inline virtual float getACCY() = 0;
    inline virtual float getACCZ() = 0;

    // 获取 imu 在世界（现实）坐标系下的欧拉角、及对应的角速度
    inline virtual float getPitch() = 0;
    inline virtual float getYaw() = 0;
    inline virtual float getRoll() = 0;

    inline virtual float getPitchVel() = 0;
    inline virtual float getYawVel() = 0;
    inline virtual float getRollVel() = 0;
```

```
        inline virtual bool isInit() = 0; // 检测是否导入了 imu 指针，防止未导入时，因为指针为空指针，调用函数而进入硬件中断
    public:
        // 加速度极性，传入 1 或-1，用于将 imu 现实的极性对齐到人为定义的坐标系
        linerPolarityStruct accPolarity;

    // 欧拉角极性，传入 1 或-1，用于将 imu 现实的极性对齐到人为定义的坐标系
        anglePolarityStruct eularPolarity;
        // 欧拉角基准值，用于校准，会在校准极性后，加上此基准值
        AngularDataStructdef eularBaseData = {0, 0, 0};
        float angle_unit_convert = 1;

        // 绑定加速度坐标系，传入三个枚举值，分别代表 x,y,z 轴对应的 imu 坐标系
        // 若传入 imuWorldAccY,imuWorldAccX,imuWorldAccZ
        // 则代表将 imu 的 y 轴对应到人为定义的 x 轴，imu 的 x 轴对应到人为定义的 y 轴，imu 的 z 轴对应到人为定义的 z 轴
        inline void bindEulerCoordinate(imuEulerCoordinate abstractPitch, imuEulerCoordinate abstractYaw, imuEulerCoordinate abstractRoll)
        {
            abstractEularData.Pitch = abstractPitch;
            abstractEularData.Yaw = abstractYaw;
            abstractEularData.Roll = abstractRoll;
        }

        // 绑定加速度坐标系，传入三个枚举值，分别代表 pitch，yaw，roll 轴对应的 imu 坐标系
        // 若传入 imuWorldPitch,imuWorldRoll,imuWorldYaw
        // 则代表将 imu 的 pitch 轴对应到人为定义的 pitch 轴，imu 的 Roll 轴对应到人为定义的 yaw 轴，imu 的 Yaw 轴对应到人为定义的 roll 轴
        inline void bindAccCoordinate(imuAccCoordinate abstractX, imuAccCoordinate abstractY, imuAccCoordinate abstractZ)
        {
            abstractAccData.x = abstractX;
            abstractAccData.y = abstractY;
            abstractAccData.z = abstractZ;
        }

        // 获取 imu 世界坐标系数据，转化为人为定义的坐标系、对齐极性后的数据
        // 功能基本实现，继承类不用再写
        inline LinearDataStructdef *getAccData()
        {
            if (isInit() == 0)
                return &accData;
            float tempAccData[3];
            tempAccData[imuWorldAccX] = this->getACCX();
            tempAccData[imuWorldAccY] = this->getACCY();
            tempAccData[imuWorldAccZ] = this->getACCZ();

            accData.x = tempAccData[abstractAccData.x] * accPolarity.x * angle_unit_convert;
            accData.y = tempAccData[abstractAccData.y] * accPolarity.y * angle_unit_convert;
            accData.z = tempAccData[abstractAccData.z] * accPolarity.z * angle_unit_convert;
            return &accData;
        }
```

```
        inline AngularDataStructdef *getEularData()
        {
            if (isInit() == 0)
                return &eularData;
            float tempGyroData[3];
            tempGyroData[imuWorldPitch] = this->getPitch();
            tempGyroData[imuWorldYaw] = this->getYaw();
            tempGyroData[imuWorldRoll] = this->getRoll();

            eularData.pitch = tempGyroData[abstractEularData.Pitch] * eularPolarity.pitch * angle_unit_convert
+ eularBaseData.pitch;
            eularData.yaw = tempGyroData[abstractEularData.Yaw] * eularPolarity.yaw * angle_unit_convert +
eularBaseData.yaw;
            eularData.roll = tempGyroData[abstractEularData.Roll] * eularPolarity.roll * angle_unit_convert
+ eularBaseData.roll;
            return &eularData;
        }

         inline AngularDataStructdef *getAngleVelData()
        {
            if (isInit() == 0)
                return &angleVelData;
            float tempGyroData[3];
            tempGyroData[imuWorldPitch] = this->getPitchVel();
            tempGyroData[imuWorldYaw] = this->getYawVel();
            tempGyroData[imuWorldRoll] = this->getRollVel();

            angleVelData.pitch = tempGyroData[abstractEularData.Pitch] * angle_unit_convert *
eularPolarity.pitch;
            angleVelData.yaw = tempGyroData[abstractEularData.Yaw] * angle_unit_convert * eularPolarity.yaw;
            angleVelData.roll = tempGyroData[abstractEularData.Roll] * angle_unit_convert *
eularPolarity.roll;
            return &angleVelData;
        }
    };
};
/**
 * @brief IMU 抽象模板类，仅仅为了可以特化而写，并无实际作用
 * 模板主类不写实现，是为了避免在传入没有特化过的 IMU 类型时，会出现无法设想的错误
 * @tparam motorType
 */
template <class IMUtype>
class abstractIMUClassdef
{
};

#if USE_LPMS_BE2
/**
 * @brief 特化模板类，用于阿路比 imu
 */
template <>
class abstractIMUClassdef<LPMS_BE2_Typedef> : public abstractIMU::abstractIMUBaseClassdef
{
private:
    LPMS_BE2_Typedef *lpms = nullptr;
```

```
    inline virtual float getACCX()
    {
        return lpms->get_data().linearAccX;
    }
    inline virtual float getACCY()
    {
        return lpms->get_data().linearAccY;
    }
    inline virtual float getACCZ()
    {
        return lpms->get_data().linearAccZ;
    }

    inline virtual float getPitch()
    {
        return lpms->get_data().Euler_Pitch;
    }
    inline virtual float getYaw()
    {
        return lpms->get_data().Euler_Yaw;
    }
    inline virtual float getRoll()
    {
        return lpms->get_data().Euler_Roll;
    }

    /* 对于阿路比 imu */
    /* pitch 绑定 caliGyroY */
    /* yaw 绑定 caliGyroZ */
    /* roll 绑定 caliGyroX */
    inline virtual float getPitchVel()
    {
        return lpms->get_data().caliGyroY;
    }
    inline virtual float getYawVel()
    {
        return lpms->get_data().caliGyroZ;
    }
    inline virtual float getRollVel()
    {
        return lpms->get_data().caliGyroX;
    }

    inline virtual bool isInit()
    {
        if (lpms == nullptr)
            return 0;
        else
            return 1;
    }

public:
    inline void bindIMU(LPMS_BE2_Typedef *_lpms)
    {
        this->lpms = _lpms;
    }
```

```
    inline void update(uint8_t *data)
    {
        if (isInit() == 0)
            return;
        lpms->LPMS_BE2_Get_Data(data);
    }
};
```