新框架的初步引导

- 善用搜索栏进行全局搜索和Ctrl+F进行局部搜索，看代码前先编译，便于转到函数定义，如果转到定义失败，可以用全局搜索功能搜索到。
- 前置知识：freertos（只需了解线程如何用、线程优先级以及线程频率）、面向对象的编程思想（代码中的xx_create相当于C++中的构造函数，填写XX_config相当于配置传如构造函数的初始化信息）、vector数据结构（很多功能是通过vector实现的）、循环队列数据结构。 先从Tutorials开始看，配环境可以不用看，有一些地方看不懂可以先不用看懂，留个印象即可。
- pub_sub机制看不懂暂时没关系。
- 这套代码框架可以理解为一套拼图，有些重要部分一旦确认就可以帮助更好地理解其他模块，有些细节部分即使理解也对其他模块没有较大帮助，推荐优先学习比较核心比较重要的部分。这里给出一些关键词：面向对象、vector、循环队列、去耦合、结构体、板极支持包、外设驱动、代码复用等。
- 下例将带大家看懂在这套代码框架中3508电机驱动是如何实现的，并且通过讲解面向对象、vector以及注册回调函数帮助大家凑齐一些重要拼图，后续的一些文档将会补上其他拼图。

实际案例：具体讲如何驱动一个3508电机

在app层，使用一个电机需要这样的流程，首先在结构体构造函数（可以看作参数配置和初始化）里面：

```
can_motor_config lf_config;//声明一个电机的config结构体
controller_config lf_controller_config;//声明一个控制器config结构体
lf_controller_config.control_type = PID_MODEL;//配置电机的控制模式为PID
lf_controller_config.control_depth = SPEED_CONTROL;//电机的控制深度为速度环
PID_SetConfig_Pos(&lf_controller_config.position_pid_config, 0, 0, 0, 0, 0);//位置环参数全设为0
PID_SetConfig_Pos(&lf_controller_config.speed_pid_config, 5, 0, 2, 0, 15000);//速度环参数设置（可调参）
lf_config.motor_model = MODEL_3508;//电机类型选择为3508
lf_config.bsp_can_index = 1;//can的bsp层的id是1
lf_config.motor_set_id = 1;//电机的id是1
lf_config.motor_controller_config = lf_controller_config;//把控制器config结构体的位置给电机config结构体里的控制器结构体
lf_config.position_fdb_model = MOTOR_FDB;//反馈模式为使用电机反馈
lf_config.speed_fdb_model = MOTOR_FDB; //反馈模式为使用电机反馈
lf_config.output_model = MOTOR_OUTPUT_NORMAL;//不采用输出反转
lf_config.lost_callback = chassis_motor_lost;//传入电机掉线函数
obj->lf = Can_Motor_Create(&lf_config);//把配置好的电机config结构体传给构造函数
```

然后在结构体更新函数里面改变电机预设值（速度或位置），就可以实现对这个电机的控制。

我们把配置好的参数全部扔进了最后调用的一个create函数，接下来看在以上代码最后一个函数里我们做了什么：

```
can_motor * Can_Motor_Create (can_motor_config *config) {
    can_motor *obj = (can_motor *)malloc(sizeof(can_motor));//首先新建一个电机结构体的指针并且分配对应内存
```

```
    memset(obj, 0, sizeof(can_motor));//初始化这段内存
    obj->position_queue = create_circular_queue(sizeof(float), VELOCITY_WINDOW);
    obj->config = *config;//把配置好的config给电机结构体的成员config存着，方便使用他的
参数
    obj->position_sum = 0;
    obj->fdbPosition = FDB_INIT_VALUE;
    motors_id[obj->config.bsp_can_index][obj->config.motor_model][obj-
>config.motor_set_id - 1] = 1;这是一个三维数组，把对应配置的值置1说明对应配置的电机已经
被注册
    switch (config->motor_model) {//因为所有电机都使用这个函数所以要分情况讨论
        case MODEL_2006:
            if (config->motor_set_id < 5) {
                motor_instances[config->bsp_can_index][0][config->motor_set_id -
1] = obj;
            } else {
                motor_instances[config->bsp_can_index][1][config->motor_set_id -
5] = obj;
            }
            BSP_CAN_AddFilter(obj->config.bsp_can_index, 0x200 + config-
>motor_set_id);
            break;
        case MODEL_3508://当使用3508电机时
            if (config->motor_set_id < 5) {
                motor_instances[config->bsp_can_index][0][config->motor_set_id -
1] = obj;
            } else {
                motor_instances[config->bsp_can_index][1][config->motor_set_id -
5] = obj;
            }// motor_instances是一组电机结构体数组，把在这一步把对应配置的数组赋值对应
结构体，便于后面进行的封装严格的操作
            BSP_CAN_AddFilter(obj->config.bsp_can_index, 0x200 + config-
>motor_set_id);//配置过滤器，便于接受反馈（具体如何配置可以点开对应函数查看）
            break;
        case MODEL_6020:
            if (config->motor_set_id < 5) {
                motor_instances[config->bsp_can_index][1][config->motor_set_id -
1] = obj;
            } else {
                motor_instances[config->bsp_can_index][2][config->motor_set_id -
5] = obj;
            }
            BSP_CAN_AddFilter(obj->config.bsp_can_index, 0x204 + config-
>motor_set_id);
            break;
        default:
            break;
    }
    if (obj->config.speed_fdb == NULL) {
        obj->config.speed_fdb = &obj->velocity;
    }//把velocity的地址给speed_fdb，velocity的值在电机反馈接受回调函数里面更新，于是
speed_fdb也同步更新了
    if (obj->config.position_fdb == NULL) {
        obj->config.position_fdb = &obj->real_position;
    }
```

```
    obj->motor_controller = create_controller(&obj-
>config.motor_controller_config); //把刚刚配置好的控制器config扔进控制器的构造函数
    obj->monitor = Monitor_Register(obj->config.lost_callback, 5, obj);//给电机注册
一个看门狗，如果掉线5ms就狗叫（电机反馈1ms更新一次）
    obj->enable = MOTOR_STOP;//暂时把电机模式设置为不能转
    return obj;//返回创建好的这个结构体的首地址
}
```

接下来我们看上面创建了这个电机结构体，会产生什么影响。

```
void StartMotorTask(void *argument)
{
  /* USER CODE BEGIN StartMotorTask */
    portTickType currentTimeMotor;
    currentTimeMotor = xTaskGetTickCount();
    /* Infinite loop */
    for (;;) {
        FrameRateStatistics(&task_frameRate.FPS_controller);//检测它的频率有没有寄掉
        HAL_Motor_Calc_Loop();
        vTaskDelayUntil(&currentTimeMotor, 1);//意思是在1ms以后令线程就绪
    }
  /* USER CODE END StartMotorTask */
}
```

这是freertos一直在跑的电机线程，继续深入查看HAL_Motor_Calc_Loop()这个函数：

```
void HAL_Motor_Calc_Loop() { Can_Motor_Calc_Send(); }
```

继续查看：

```
void Can_Motor_Calc_Send() {
    for (int can_index = 0; can_index < 2; can_index++) {
        for (int identifier = 0; identifier < 3; identifier++) {//套这两层确保遍历
所有的电机
            uint8_t identifier_send = 0;
            short buf[4] = {0};//初始化需要发送的字符为0
            for (int id = 0; id < 4; id++) {//在上面情况下遍历所有的id
                can_motor *obj = motor_instances[can_index][identifier][id];
                if (obj == NULL) continue;
                //电机掉线判断，如果同一个包内的所有的电机都掉线，那么这个包将不会发出
                if (is_Offline(obj->monitor)) continue;
                identifier_send = 1;
                if (obj->config.position_fdb_model == OTHER_FDB){
                    obj->motor_controller->fdb_position = *obj-
>config.position_fdb;
                } else {
                    obj->motor_controller->fdb_position = obj->real_position;
```

```
                }
                if(obj->config.speed_fdb_model == OTHER_FDB){
                    obj->motor_controller->fdb_speed = *obj->config.speed_fdb;
                } else {
                    obj->motor_controller->fdb_speed = obj->fdbSpeed * 6;
                }
//以上为给控制器需要调用的反馈赋值

                controller_calc(obj->motor_controller);
//这里的controller_calc（）相当于进行了一次pid计算（也可能是别的控制方式），我们app层把ref的值（目标值）配置好，在上面也获取了反馈值，经过这层调用，我们的output即输出值也更新了
                buf[id] = (short)obj->motor_controller->output;//把控制器输出值放入需要给电机发送的字符
                if (obj->config.output_model == MOTOR_OUTPUT_REVERSE) buf[id] *= -1;//是否反转
                if (obj->enable == MOTOR_STOP) {
                    buf[id] = 0;
                }//这个就是从底层让电机不转的办法，只要enable是MOTOR_STOP，电机就一定不会转
            }
            // 如果此标识符(identifier)对应的四个电机里至少有一个被注册，就发送这个标识符的报文，如果全部没有被注册，则这个标识符无需发送
            if (identifier_send) {
                Can_Motor_Send(can_index, identifiers[identifier], buf[0], buf[1], buf[2], buf[3]);
            }//最后调用函数发送这四个电流值
        }
    }
}
```

以上函数说明，只要电机被注册了，就会在这个函数里给对应的can发送对应的电流值， 我们看看这个发送电流的函数

```
void Can_Motor_Send(uint8_t can_id, uint16_t identifier, short motor_data_id1, short motor_data_id2, short motor_data_id3, short motor_data_id4) {
    static uint8_t data[8];
    data[0] = (uint8_t)(motor_data_id1 >> 8);
    data[1] = (uint8_t)(motor_data_id1 & 0x00FF);
    data[2] = (uint8_t)(motor_data_id2 >> 8);
    data[3] = (uint8_t)(motor_data_id2 & 0x00FF);
    data[4] = (uint8_t)(motor_data_id3 >> 8);
    data[5] = (uint8_t)(motor_data_id3 & 0x00FF);
    //对于标识符为0x2FF的情况，则data[6]和data[7]为NULL
    if (identifier == 0x200 || identifier == 0x1FF) {
        data[6] = (uint8_t)((motor_data_id4 & 0xFF00) >> 8);
        data[7] = (uint8_t)(motor_data_id4 & 0x00FF);
    }
    BSP_CAN_Send(can_id, identifier, data, 8);
}
```

首先进行了一些数据解算，然后调用bsp的can发送函数发送电流值，与考核上驱动电机的任务类似。BSP_CAN_Send只是根据c板的一些特点对hal层的can发送做了一层封装。

到这里我们一定会有一些疑点。最开始App层的config我们到底是怎么配置的呢？首先是bsp_can_index，提供一个思路，就是全局搜索他，我们会发现有很多结果 首先我们不看.h文件，也不看app层的文件，我们看到他主要是出现在Can_Motor_Create中，

```
case MODEL_3508://当使用3508电机时
            if (config->motor_set_id < 5) {
                motor_instances[config->bsp_can_index][0][config->motor_set_id -
1] = obj;
            } else {
                motor_instances[config->bsp_can_index][1][config->motor_set_id -
5] = obj;
            }
```

我们把配置了不同的bsp_can_index的电机放在了数组的不同位置，于是我们继续搜索这个数组，我们发现在Can_Motor_Calc_Send()里面，can_index与bsp_can_index的值是相同的，同时注意到，我们在最后调用Can_Motor_Send的第一个值传的是can_index。

在Can_Motor_Send这个函数里面最后调用BSP_CAN_Send的第一个值传的还是can_index，在BSP_CAN_Send这个函数的最后调用的HAL_CAN_AddTxMessage(can_devices[can_id].device,&txconf,data, &can_devices[can_id].tx_mailbox)中的can_id的值就是can_index，根据hal库的知识，我们知道这个函数第一个值传的是&hcan1或&hcan2，所以我们全局搜索can_devices，发现在BSP_CAN_Init()有

```
can_devices[0].device = &hcan1;
// can_devices[0].fifo = CAN_RX_FIFO0;
// can_devices[0].tx_mailbox = (uint32_t *)CAN_TX_MAILBOX0;
can_devices[0].bank_prefix = 0;
can_devices[0].call_backs = cvector_create(sizeof(can_rx_callback));  //创建用于储
存回调函数的向量组，存储方式为其头指针

can_devices[1].device = &hcan2;
// can_devices[1].fifo = CAN_RX_FIFO1;
// can_devices[1].tx_mailbox = (uint32_t *)CAN_TX_MAILBOX1;
can_devices[1].bank_prefix = 14;
can_devices[1].call_backs = cvector_create(sizeof(can_rx_callback));
```

因此可以知道，在令bsp_can_index为0时，我们用的是&hcan1，bsp_can_index为1时，我们用的是&hcan2 。其余参数配置方法也可以参照以上流程，或者也可以询问别人配置过的经验。

说到这里，我们大概已经知道了电机是怎么转起来的以及怎么配置他的一些参数，但是我们还没有搞懂它的反馈是怎么得到的，也不清楚是如何实现速度控制和位置控制的。

引入另一种看代码的方式，即直接从对应的外设驱动开始看。打开can_motor.c文件，我们可以看到一些重要的函数，Can_Motor_Create已经在上面看过了，就不讲了，我们看CanMotor_RxCallBack这个函数，通过全局检索发现，他BSP_CAN_RegisterRxCallback(0, CanMotor_RxCallBack);这样注册为两条can通道的回调函数了。为了防止思路断开，注册回调函数我们放在结尾讲。

这里默认我们知道当主控收到不被过滤的id发来的can数据时，在数据包接收完毕后会调用我们注册的回调函数。

在CanMotor_RxCallBack函数中分情况调用了Can_Motor_FeedbackData_Update，

```
void Can_Motor_FeedbackData_Update(can_motor *obj, uint8_t *data) {
    obj->last_fdbPosition = obj->fdbPosition;
    obj->fdbPosition = ((short)data[0]) << 8 | data[1];
    if(obj->last_fdbPosition == FDB_INIT_VALUE) obj->last_fdbPosition = obj-
>fdbPosition;
    obj->fdbSpeed = ((short)data[2]) << 8 | data[3];
    obj->electric_current = ((short)data[4]) << 8 | data[5];
    if (obj->config.motor_model == MODEL_2006) {
        obj->temperature = 0;
    } else {
        obj->temperature = data[6];
    }
    if (obj->fdbPosition - obj->last_fdbPosition > 4096)
        obj->round--;
    else if (obj->fdbPosition - obj->last_fdbPosition < -4096)
        obj->round++;
    obj->last_real_position_8192 = obj->real_position_8192;
    obj->real_position_8192 = obj->fdbPosition + obj->round * 8192;
    obj->real_position = obj->real_position_8192 * 360.0 / 8192.0;

    float position_delta = obj->real_position_8192 - obj->last_real_position_8192;
    if (obj->position_queue->cq_len == VELOCITY_WINDOW) {
        float *now = circular_queue_pop(obj->position_queue);
        obj->position_sum -= *now;
    }
    circular_queue_push(obj->position_queue, &position_delta);
    obj->position_sum += position_delta;
    float vnow = obj->position_sum * 43.9453125 / obj->position_queue->cq_len;
    obj->velocity = 0.2f * obj->velocity + 0.8f * vnow;
    obj->monitor->reset(obj->monitor);
}
```

传入的data是从注册的回调函数那里一步一步传回来的，就是电机反馈的原始数值，在这个函数中完成它的解算，同时也完成了一些控制器所需的反馈值的更新。

接下来讲控制器是如何实现的

前面我们配置了控制器的config，并且在电机的构造函数中把它丢入了一个控制器的构造函数：create_controller，而在这个函数中我们只是简单的分情况讨论，当控制器为pid时，只是简单的把config拷贝了过去。

```
    if (obj->config.control_type == PID_MODEL) {
        obj->pid_pos_data.config = obj->config.position_pid_config;
        obj->pid_speed_data.config = obj->config.speed_pid_config;
    }
```

至于怎么配置的config，还记得之前在app层调用的PID_SetConfig_Pos吗，我们传入了config结构体和一些参数，在这个函数中我们完成了这个结构体的配置。

```
void PID_SetConfig_Pos(struct PID_config_t* obj, float kp, float ki, float kd,
float errormax, float outputmax) {
    obj->PID_Mode = PID_POSITION;
    obj->KP = kp;
    obj->KI = ki;
    obj->KD = kd;
    obj->error_max = errormax;
    obj->outputMax = outputmax;
}
```

再次涉及到与控制器相关的是在Can_Motor_Calc_Send()中，调用了controller_calc函数，

```
void controller_calc(controller* obj) {
    if (obj->config.control_type == PID_MODEL) {
        if (obj->config.control_depth >= POS_CONTROL) {
            obj->pid_pos_data.fdb = obj->fdb_position;
            obj->pid_pos_data.ref = obj->ref_position;
            PID_Calc(&obj->pid_pos_data);
            obj->ref_speed = obj->pid_pos_data.output;
        }
        if (obj->config.control_depth >= SPEED_CONTROL) {
            obj->pid_speed_data.fdb = obj->fdb_speed;
            obj->pid_speed_data.ref = obj->ref_speed;
            PID_Calc(&obj->pid_speed_data);
            obj->output = obj->pid_speed_data.output;
        }
    } else if (obj->config.control_type == MRAC_MODEL) {
        if (obj->config.control_depth >= POS_CONTROL) {
            mrac_2d_calc(&obj->mrac_2d_data, obj->ref_position, obj->fdb_position,
obj->fdb_speed, 1);
            obj->output = obj->mrac_2d_data.output;
        }
    } else if( obj->config.control_type == ADRC_MODEL) {
        if (obj->config.control_depth >= POS_CONTROL) {
            obj->adrc_pos_data.prog.fdb = obj->fdb_position;
            obj->adrc_pos_data.prog.ref = obj->ref_position;
            ADRCFunction(&obj->adrc_pos_data);
            obj->ref_speed = obj->adrc_pos_data.prog.output;
        }
        if (obj->config.control_depth >= SPEED_CONTROL) {
            obj->adrc_speed_data.prog.fdb = obj->fdb_speed;
            obj->adrc_speed_data.prog.ref = obj->ref_speed;
            ADRCFunction(&obj->adrc_speed_data);
            obj->output = obj->adrc_speed_data.prog.output;
        }
```

```
      } else if (obj->config.control_type == SMC_MODEL) {
         if (obj->config.control_depth >= SPEED_CONTROL) {
             obj->smc_speed_data.fdb = obj->fdb_speed;
             obj->smc_speed_data.ref = obj->ref_speed;
             SMC_Calc(&obj->smc_speed_data);
             obj->output = obj->smc_speed_data.output;
         }
     }
     else if (obj->config.control_type == SMC_POSITION_MODEL) {

             obj->smc_2_position_data.fdb = obj->fdb_position;
             obj->smc_2_position_data.ref = obj->ref_position;
             SMC_2_Calc(&obj->smc_2_position_data);
             obj->output = obj->smc_2_position_data.output;
     }
     else if (obj->config.control_type == LQR_MODEL){
         obj->lqr_data.x1 = obj->fdb_position * 0.0174532f;  //转弧度制
         obj->lqr_data.x1_ref = obj->ref_position * 0.0174532f;
         obj->lqr_data.x2 = obj->fdb_speed * 0.0174532f;
         LQR_Calc(&obj->lqr_data);
         obj->output = obj->lqr_data.output;
     }
 }
```

当把控制器配置为pid时，我们调用的是PID_Calc，通过控制深度的配置，可以实现选择单速度环还是位置环速度环串级。而PID_Calc与第一轮考核差别不大，是你们熟悉的代码。

在使用控制器时，我们要时刻注意控制器极性，使用电机反馈一般不会有极性上的问题，如果使用其他反馈就要考虑反馈方向与输出方向是否相同，如果极性错误可能导致疯车。

注册回调函数

注册回调函数是这个代码框架一种比较神秘的写法，首先我们看他的典型函数BSP_CAN_RegisterRxCallback

```
void BSP_CAN_RegisterRxCallback(uint8_t can_id, can_rx_callback func) {
    cvector_pushback(can_devices[can_id].call_backs, &func);
}

typedef struct BSP_CanTypeDef_t {
    CAN_HandleTypeDef *device;  //自定义总线编号
    uint32_t tx_mailbox;
    cvector *call_backs;
    uint32_t bank_prefix;              //不同can对应的过滤器相关参数值
    uint16_t filters[FILTER_MAX_CNT];  //按标准帧ID定义
} BSP_CanTypeDef;
```

可以看到这份代码使用了c实现了vector，然后使用vector实现了注册回调函数 call_backs是vector变量，在注册回调函数的函数中，我们把函数CanMotor_RxCallBack的函数指针放入这个vector里面暂存。

在hal库定义的回调函数中，我们调用bsp层的回调函数

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    BSP_CAN_FifoMsg_Callback(hcan, CAN_RX_FIFO0);
}

void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    BSP_CAN_FifoMsg_Callback(hcan, CAN_RX_FIFO1);
}
```

以下是bsp层回调函数的写法

```
void BSP_CAN_FifoMsg_Callback(CAN_HandleTypeDef *hcan, uint32_t fifo) {
    for (int i = 0; i < DEVICE_CAN_CNT; ++i) {
        if (hcan == can_devices[i].device) {
            uint8_t bsp_can_rxbuf[8];
            CAN_RxHeaderTypeDef rxconf;
            HAL_CAN_GetRxMessage(hcan, fifo, &rxconf, bsp_can_rxbuf);
            for (size_t j = 0; j < can_devices[i].call_backs->cv_len; ++j) {
                can_rx_callback funcnow = *(can_rx_callback
*)cvector_val_at(can_devices[i].call_backs, j);
                funcnow(i, rxconf.StdId, bsp_can_rxbuf, rxconf.DLC);
            }
        }
    }
}
这段代码中最重要的部分是
for (size_t j = 0; j < can_devices[i].call_backs->cv_len; ++j) {
                can_rx_callback funcnow = *(can_rx_callback
*)cvector_val_at(can_devices[i].call_backs, j);
                funcnow(i, rxconf.StdId, bsp_can_rxbuf, rxconf.DLC);
            }
```

在这段代码中我们将函数指针所指的函数从vector中抽出来调用，并且把bsp_can_rxbuf即接收到的数据传进该函数，作为data。至此实现了注册回调函数的功能。

这套代码框架很多功能都是靠vector实现的，请大家认真了解vector并且仔细阅读cvector的两个文件

- 以上是这套代码一个外设实现其功能的全流程，其他外设的学习也可以参考以上步骤，融会贯通