1、mower_ctrl 功能包

1、heserial.py（串口文件）

1.1、涉及自定义消息类型

1.1.1 CtrlComand.msg

行驶速度和转向角度：

```
float32 driving_speed
float32 steering_angle
```

1.1.2 ModeSwitch.msg

目标模式：行驶、转换、旋转

```
uint8 target_mode
uint8 MOTION_MODE_STEERING = 0
uint8 MOTION_MODE_TRANSLATION = 1
uint8 MOTION_MODE_ROTATION = 2
```

1.1.3 SensorValue.msg

传感器的值：具体含义未知

```
uint8 col_data1
uint8 col_data2
uint8 col_data3
uint8 col_data4
uint16 ul_data1
uint16 ul_data2
uint16 ul_data3
uint16 ul_data4
uint32 distance
uint8 carpipe
uint8 rollpipe
float32 nowcar_speed
```

1.2、程序解析

1.2.1 类

类1：AUTOState（枚举类）

共计四个状态：WAITING（等待）、ALIGNMENT（对齐）、WORK（工作）、START（启动）

```
class AUTOState(Enum):
    WAITING = 0
    ALIGNMENT = 1
    WORK = 2
    START = 3
```

类2：串口控制

方法1：初始化函数

1. 初始化节点名：`hf_serial`，并设置无论如何启动该节点；
2. 初始化串口名称（serial_port，"/dev/ttyTHS1"）、波特率（serial_baudrate，115200）、控制速率或控制频率（control_rate）、ros循环频率（rate，由前者 control_rate 传入）；
3. 初始化时间（time）为0；
4. 初始化计数（ctrl_count_down）为0；
5. 定义发送指令集（tx_command）
   包括：摄像头未被正确检测、上位机自检无误、设置运动模式为横向、设置运动模式为竖向、上位机对行完成、上位机对行错误、获取当前车辆各模块各状态数据信息、通知上位机开始工作、通知下位机里程计清零、通知上位机退出自动工作、通知上位机暂停工作、通知上位机继续自动工作：

```
self.tx_command = [
            "#camera,ERROR*\n",
            "#jeston_check,OK*\n",
            "set_mode TS\n",
            "set_mode VT\n",
            "alignment_ready\n",
            "alignment_error\n",
            "read_carinfo\n",
            "#work_start,OK*\n",
            "distance_reset\n",
            "#work_stop,OK*\n",
            "#work_warning,OK*\n",
            "#work_continue,OK*\n"
            ]
```

6. 初始化传输计数（tx_count）、传输数量（tx_num）、命令行界面时间（CLItime）均为0；
7. 初始化当前工作模式为 start，即开始自动工作；
8. 串口初始化操作，定义串口变量ser，定义串口地址；（serial_port）、波特率（serial_baudrate）以及发送或等待数据的超时时间（timeout=0.5/self.control_rate）；若设置不成功则打印错误信息；
9. 定义发布者：
   1）RX_num_publisher：发送 Int8 类型消息，话题名为 “RX_num”；
   2）RX_info_publisher：发送自定义消息类型为 SensorValue 的消息，话题名为 “RX_info”；
   3）carinfo_publisher：发送 string 类型的消息，话题名为 “car_info”；

10. 定义订阅者：
    1. TX_num_subscriber：订阅话题名为“TX_num”，消息类型为 Int8 的消息，回调函数为 TXHandler；
       回调函数：
        1. 接收 Int8 消息类型的变量为 tx_num；
        2. 如果 类中变量tx_num 为 新传入的 tx_num 值：
            1. 类中变量tx_num 为 新传入的 tx_num 值
            2. tx_count 计数置零；
        3. 当接收到的 == == 大于 0 且计数（tx_count）为 0 时：
            1. 日志记录：tx_command[tx_num.data-1]的内容；
            2. 并将 tx_command 字符串集中 tx_num 对应的数据写入串口；
            3. CLItime 设为9，tx_count 计数自加1；
        4. 当计数 tx_count 为 3 时，清零。
    2. ctrl_subscriber：订阅话题名为“ctrl_command”，消息类型为自定义消息类型：CtrlCommand，调用回调函数为 CtrlCommandHandler：
       回调函数：
        1. 接收 CtrlCommand 消息类型的变量为 ctrl_msg；
        2. 如果当前 ctrl_count_down 为 0 且当前车辆状态为 自动工作：
            1. 设置 driving_speed 为 ctrl_msg 中 driving_speed 四舍五入且保留两位小数的结果；
            2. 设置 steering_angle为 ctrl_msg 中 steering_angle 四舍五入且保留两位小数的结果；
            3. 初始化字符串 ctrl_command_str 的内容为 “move \n”；
            4. 打印字符串 ctrl_command_str 内容，并写入到串口；
            5. 设置 CLItime 为 9， self.ctrl_count_down 为 1。
11. 初始化以下参数

```
self.last_speed = 0
self.last_angle = 0
self.last_rspeed = 0
self.last_wateren = 0
self.driving_speed = 0
self.steering_angle = 0
self.roll_speed = 0
self.water_en = 0
```

方法2：析构函数 del

对象销毁前关闭串口

方法3：命令接收函数 TXHandler（参数：tx_num，参数类型：Int8）

1. 接收 Int8 消息类型的变量为 tx_num；
2. 如果 类中变量tx_num 为 新传入的 tx_num 值：
    1. 类中变量tx_num 为 新传入的 tx_num 值
    2. tx_count 计数置零；
3. 当接收到的 == == 大于 0 且计数（tx_count）为 0 时：
    1. 日志记录：tx_command[tx_num.data-1]的内容；
    2. 并将 tx_command 字符串集中 tx_num 对应的数据写入串口；
    3. CLItime 设为9，tx_count 计数自加1；

4. 当计数 tx_count 为 3 时，清零。

方法4：命令接收函数 CtrlCommandHandler（参数：ctrl_msg，参数类型：CtrlCommand）

1. 接收 CtrlCommand 消息类型的变量为 ctrl_msg；
2. 如果当前 ctrl_count_down 为 0 且当前车辆状态为 自动工作：
    1. 设置 driving_speed 为 ctrl_msg 中 driving_speed 四舍五入且保留两位小数的结果；
    2. 设置 steering_angle为 ctrl_msg 中 steering_angle 四舍五入且保留两位小数的结果；
    3. 初始化字符串 ctrl_command_str 的内容为 “move \n”；
    4. 打印字符串 ctrl_command_str 内容，并写入到串口；
    5. 设置 CLItime 为 9， self.ctrl_count_down 为 1。

方法5：命令接收函数 RxCommand

1. 接收参数：num （类型未知）
2. 打印当前接收到的消息；
3. 初始化 RX_num 内容为 num；
4. 发布者RX_num_publisher 发布 RX_num 。

方法6：空闲指令函数 FreeCLITxCommand

1. 如果 CLItime 大于 0 ，则自减 1；
2. 否则：
    1. 发送空闲指令 free_command，内容为 “CLI_free\n”；
    2. 日志记录空闲指令内容；
    3. 串口写入空闲指令 free_command；
    4. CLItime 置 9。

方法7：主循环 MainLoop

运行前提：ros正在运行

1. 打印当前工作状态；
2. 读取串口信息并存入 feedback 中；
3. 当当前状态为 AUTOState.START （开始工作）时：
    1. 设置 start_command 内容为：`program_started\n`；
    2. 日志记录 start_command 的内容；
    3. 并在串口写入该信息，通知下位机：上位机已经准备就绪；CLItime 置 9；
    4. 当从串口读入的信息 （feedback）长度大于 1：
        1. 终端打印：`RX: + 接收到的内容`；
        2. 如果读取出来的信息为 `#program_started,OK*\n`，表示通知下位机：上位机决策程序正常启动；
        3. 将当前状态设为等待 （AUTOState.WAITING）
4. 当当前状态为等待 （AUTOState.WAITING） ：
    1. 如果接受到的串口信息数据长度大于 1：
        1. 终端打印：`RX: + 接收到的内容(feedback)`；
        2. 如果接受到的信息刚好为：`auto_work\n` （表示下位机通知上位机：进入自动工作模式） ：

                1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 1（num的值），当前状态设置为 对齐；
5. 当当前状态为对齐（AUTOState.ALIGNMENT）：
    1. 如果从串口接收到的信息长度大于 1：
        1. 打印：`RX：+ 接收到的内容`；
        2. 如果从串口读取的信息为：`work_start\n`，（表示下位机通知上位机：开始自动工作）：
            1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 2（num的值）；
            2. 将当前状态设置为 自动工作（AUTOState.WORK）；
        3. 如果从串口读取的信息为：`work_stop\n`，（表示下位机通知上位机：退出自动工作）：
            1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 3（num的值）；
            2. 将当前状态设置为 等待（AUTOState.WAITING）；
6. 当当前状态为自动工作（AUTOState.WORK）：
    1. 如果接受到的串口信息数据长度大于 1：
        1. 终端打印：`RX：+ 接收到的内容`；
        2. 如果从串口读取的信息为：`work_stop\n`（表示下位机通知上位机：退出自动工作）：
            1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 3（num的值）；
            2. 将当前状态设置为 等待（AUTOState.WAITING）；
            3. ctrl_count_down 计数清零；
        3. 如果从串口读取的信息为：`#set_mode,OK*\n`（表示下位机通知上位机：运动模式设置成功）：
            1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 4（num的值）；
        4. 如果从串口读取的信息为：`#distance_reset,OK*\n`（表示下位机通知上位机：里程计清零成功）：
            1. 通过 RxCommand 函数调用发布者 RX_num_publisher 发布 5（num的值）；
        5. 如果上述情况都不满足：
            1. 对从串口读取的数据进行处理：先使用 `*` 作为分隔符，并取拆分后的第一段，再对其使用分隔符 `,` 进行处理；
            2. 将经过两次拆分的字符串列表赋值给 RX_STR；
            3. 如果拆分后的第一段内容等于 `#move` && 拆分后的第二段内容等于 `OK` && 字符串列表的长度为 6：
                1. 将拆分后的第三段内容保留两位小数后，赋值给 last_speed；
                2. 将拆分后的第四段内容保留两位小数后，赋值给 last_angle；
                3. 将拆分后的第五段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data1；
                4. 将拆分后的第六段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data2；
                5. 调用发布者 RX_info_publisher 发布 RX_info 信息（RX_info）；
                6. 如果 last_speed 等于 driving_speed && last_angle 等于 steering_angle，打印：`move feedback ok`，

并将 ctrl_count_down 清零。

4. 拆分后的第一段内容等于 `#read_carinfo` && 拆分后的第二段内容等于 `OK` && 字符串列表的长度为 44：
    1. 将从串口读取的信息存入 carinfo，并使用发布者 carinfo_publisher 发布该信息（carinfo）；
    2. 将拆分后的第18段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 col_data1；
    3. 将拆分后的第19段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 col_data2；
    4. 将拆分后的第20段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 col_data3；
    5. 将拆分后的第21段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 col_data4；
    6. 将拆分后的第14段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data1；
    7. 将拆分后的第15段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data2；
    8. 将拆分后的第16段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data3；
    9. 将拆分后的第17段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 ul_data4；
    10. 将拆分后的第22段内容保留整数部分后，赋值给 RX_info（自定义消息类型SensorValue）的 distance；
    11. 使用发布者 RX_info_publisher发布该信息 (RX_info)
7. 执行空闲指令函数
8. 按照设定频率 rate 循环

2、hfflower.py

1、涉及的自定义消息类型

1.1 CtrlComand.msg

行驶速度和转向角度：

```
float32 driving_speed
float32 steering_angle
```

1.2 ModeSwitch.msg

目标模式：行驶、转换、旋转

```
uint8 target_mode
uint8 MOTION_MODE_STEERING = 0
uint8 MOTION_MODE_TRANSLATION = 1
uint8 MOTION_MODE_ROTATION = 2
```

1.3 SensorValue.msg

传感器的值：具体含义未知

```
uint8 col_data1
uint8 col_data2
uint8 col_data3
uint8 col_data4
uint16 ul_data1
uint16 ul_data2
uint16 ul_data3
uint16 ul_data4
uint32 distance
uint8 carpipe
uint8 rollpipe
float32 nowcar_speed
```

2 程序解析

2.1 类

类1：WHEEL（枚举类）

横向（TRANSVERSE）和垂直（VERTICAL）

```
class WHEEL(Enum):
    TRANSVERSE =0
    VERTICAL =1
```

类2：CarTowards（枚举类）

GO（前进）和BACK（后退）

```
class CarTowards(Enum):
    GO =0
    BACK =1
```

类3：AUTOState（枚举类）

在 heserial.py 文件的枚举类 AUTOState 基础上加了 FINISH（结束）、OVER () 和 ERROR ()

```
class AUTOState(Enum):
    WAITING = 0
    ALIGNMENT = 1
    GOWORK = 2
    BACKWORK = 3
    FINISH = 4
    OVER = 5
    ERROR = 6
```

类4：decision（决策）

方法1：初始化函数（init）

1. 初始化节点：hf_decision；
2. 通过外部参数 control_rate 初始化 control_rate，默认为 10；
3. 通过外部参数 linear_yk 初始化 linear_yk，默认为 0.75；
4. 通过外部参数 linear_zk 初始化 linear_zk，默认为 0.25；
5. 通过外部参数 linear_allk初始化 linear_allk，默认为 0.5；
6. 通过外部参数 near_ul初始化 near_ul，默认为 800；
7. 通过外部参数 far_ul初始化 far_ul，默认为 1200；
8. 定义一个 ROS 频率控制器对象，值为 control_rate；
9. 当前机器人的状态由 current_state 存储并初始化为 AUTOState.WAITING；
10. 初始化参数 distance 为 0；
11. 初始化参数 last_distance 为 0；
12. 初始化参数 dis_reset_flag 为 0；
13. 初始化参数 nav_num 为 0；
14. 初始化参数 nav_num1 为 0；
15. 初始化参数 delta_d 为 0；
16. 初始化参数 delta_d1为 0；
17. 初始化参数 speed 为 0；
18. 初始化参数 rx_num 为 0；
19. 初始化参数 setmodenum 为 0；
20. 初始化当前视觉对行情况标志位 leftflag 为 0；
21. 初始化当前视觉对行情况标志位 rightflag 为 0；
22. 初始化当前视觉对行情况标志位 leftnum 为 0；
23. 初始化当前视觉对行情况标志位 rightnum 为 0；
24. 初始化当前各传感器数值 ul_num1 为 0；
25. 初始化当前各传感器数值 ul_num2 为 0；
26. 初始化当前各传感器数值 ul_overtime 为 0；
27. 定义发布者：
    1. TX_num_publisher：发送 Int8 类型消息，话题名为 “TX_num”；
    2. ctrl_publisher：发送自定义 CtrlCommand 类型消息，话题名为 “ctrl_command”；
28. 定义订阅者：
    1. twist_subscriber：订阅话题名为 `duihang`，消息类型为 Twist 的消息，回调函数为 NavigationHandler；
       回调函数：
        1. 接收 Twist 消息类型的变量为 twist_msg；

2. 类中变量 nav_num 数值加1；
3. 类中变量 leftnum 数值等于 5；
4. 如果类中的变量 标志位 leftflag 小于 2：
    1. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
        1. leftflag 置 1；
        2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    2. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. leftflag 置零。

2. twist_subscriber2：订阅话题名为 `duihang2`，消息类型为 Twist 的消息，回调函数为 NavigationHandler2；
   回调函数：
    1. 接收 Twist 消息类型的变量为 twist_msg；
    2. 类中变量 nav_num1 数值加1；
    3. 类中变量 rightnum 数值等于 5；
    4. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
        1. rightflag 置 1；
        2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    1. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. rightflag 置零。
3. twist_subscriber3：订阅话题名为 `duihang3`，消息类型为 Twist 的消息，回调函数为 NavigationHandler3；
   回调函数：
    1. 接收 Twist 消息类型的变量为 twist_msg；
    2. 类中变量 nav_num 数值加1；
    3. 类中变量 leftnum 数值等于 5；
    4. 如果类中的变量 标志位 leftflag 小于 2：
        1. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
            1. leftflag 置 1；
            2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
        2. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
            1. leftflag 置零。
4. twist_subscriber4：订阅话题名为 `duihang4`，消息类型为 Twist 的消息，回调函数为 NavigationHandler4；
   回调函数：
    1. 接收 Twist 消息类型的变量为 twist_msg；
    2. 类中变量 nav_num1 数值加1；
    3. 类中变量 rightnum 数值等于 5；
    4. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
        1. rightflag 置 1；
        2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    1. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. rightflag 置零。
5. RX_num_subscriber：订阅话题名为 `RX_num`，消息类型为 Int8 的消息，回调函数为 RXHandler（接收串口信号数值）；

回调函数：

1. 接收 Int8 消息类型的变量为 rx_num；
2. 类中变量 rx_num 数值为订阅到的消息 rx_num 中的 data 部分；
3. 在终端打印：rx_num 的值；
4. 如果类中的 rx_num 等于 3：
    1. 类中的 rx_num 清零；
    2. 如果类中的 launch_state 为 AUTOState.GOWORK：
        1. 终止 launchgo 对应的节点；
    3. 如果类中的 launch_state 为 AUTOState.BACKWORK：
        1. 终止 launchback 对应的节点；
    4. 调用 StopToWaitInit() 函数，初始化以下参数：
        nav_num = 0；
        nav_num1 = 0；
        leftflag = 0；
        rightflag = 0；
        leftnum = 0；
        rightnum = 0；
        setmodenum = 0；
        distance = 0；
        dis_reset_flag = 0；
        delta_d = 0；
        delta_d1 = 0；
        speed = 0；
        rx_num = 0；
    5. 调用 TxCommand() 函数（命令发送）并传入参数 10：
        1. 打印日志：tx command num: 10；
        2. 将类中的变量 TX_num 初始化为 num（10）；
        3. 调用发布者 TX_num_publisher 发布 TX_num；
    6. 将当前状态 current_state 设为 AUTOState.WAITING：
5. 如果当前类中的 rx_num 等于 4：
    1. 将类中的 setmodenum 设为 1；
6. 如果当前类中的 rx_num 等于 5：
    1. 将类中的 dis_reset_flag 设为 1；

6. RX_info_subscriber：订阅话题名为 `RX_info`，消息类型为 SensorValue 的消息，回调函数为 InfoHandler（接收各传感器数值并处理）：

回调函数：

1. 接收 SensorValue 消息类型的变量为 msg；
2. 类中变量 ul_num1 为订阅的 msg 中的 ul_data1；
3. 类中变量 ul_num2 为订阅的 msg 中的 ul_data2；

方法2：启动关闭视觉导航程序（Startvisual）

1. 将临时生成的 UUID 赋值给变量 uuid；
2. 将生成的 UUID 传递给 ROS Launch 日志记录系统；
3. 如果接收到的 cartowards 内容为 CarTowards.GO：
    1. 使用 launchgo 这个变量来标识 hfvisual_go.launch 文件；
    2. 使用 launchgo 的 start() 方法来启动该文件；
    3. 将 launch_state 状态设置为 AUTOState.GOWORK；
    4. 终端打印日志：`go visual started`
4. 如果接收到的 cartowards 内容为 CarTowards.BACK：
    1. 使用 launchback 这个变量来标识 hfvisual_back.launch 文件；
    2. 使用 launchback 的 start() 方法来启动该文件；

3. 将 launch_state 状态设置为 AUTOState.BACKWORK；
4. 终端打印日志：`back visual started`。

方法3：RXHandler()（参数：rx_num，参数类型：Int8）

1. 接收 Int8 消息类型的变量为 rx_num；
2. 类中变量 rx_num 数值为订阅到的消息 rx_num 中的 data 部分；
3. 在终端打印：rx_num 的值；
4. 如果类中的 rx_num 等于 3：
    1. 类中的 rx_num 清零；
    2. 如果类中的 launch_state 为 AUTOState.GOWORK：
        1. 终止 launchgo 对应的节点；
    3. 如果类中的 launch_state 为 AUTOState.BACKWORK：
        1. 终止 launchback 对应的节点；
    4. 调用 StopToWaitInit() 函数，初始化以下参数：
        nav_num = 0；
        nav_num1 = 0；
        leftflag = 0；
        rightflag = 0；
        leftnum = 0；
        rightnum = 0；
        setmodenum = 0；
        distance = 0；
        dis_reset_flag = 0；
        delta_d = 0；
        delta_d1 = 0；
        speed = 0；
        rx_num = 0；
    5. 调用 TxCommand() 函数（命令发送）并传入参数 10：
        1. 打印日志：tx command num: 10；
        2. 将类中的变量 TX_num 初始化为 num（10）；
        3. 调用发布者 TX_num_publisher 发布 TX_num；
    6. 将当前状态 current_state 设为 AUTOState.WAITING；
5. 如果当前类中的 rx_num 等于 4：
    1. 将类中的 setmodenum 设为 1；
6. 如果当前类中的 rx_num 等于 5：
    1. 将类中的 dis_reset_flag 设为 1；

方法4：接收各传感器数值并处理：InfoHandler()（参数：msg，参数类型：SensorValue）

1. 接收 SensorValue 消息类型的变量为 msg；
2. 类中变量 ul_num1 为订阅的 msg 中的 ul_data1；
3. 类中变量 ul_num2 为订阅的 msg 中的 ul_data2；

方法5：接收视觉程序topic信息：NavigationHandler（参数：twist_msg，参数类型：）

1. 接收 Twist 消息类型的变量为 twist_msg；
2. 类中变量 nav_num 数值加1；
3. 类中变量 leftnum 数值等于 5；
4. 如果类中的变量 标志位 leftflag 小于 2：
    1. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
        1. leftflag 置 1；
        2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    2. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. leftflag 置零

方法6：接收视觉程序topic信息：NavigationHandler2（参数：twist_msg，参数类型：）

1. 接收 Twist 消息类型的变量为 twist_msg；
2. 类中变量 nav_num1 数值加1；
3. 类中变量 rightnum 数值等于 5；
4. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
    1. rightflag 置 1；
    2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    1. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. rightflag 置零。

方法7：接收视觉程序topic信息：NavigationHandler3（参数：twist_msg，参数类型：）

1. 接收 Twist 消息类型的变量为 twist_msg；
2. 类中变量 nav_num 数值加1；
3. 类中变量 leftnum 数值等于 5；
4. 如果类中的变量 标志位 leftflag 小于 2：
    1. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
        1. leftflag 置 1；
        2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    2. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. leftflag 置零。

方法8：接收视觉程序topic信息：NavigationHandler4（参数：twist_msg，参数类型：）

1. 接收 Twist 消息类型的变量为 twist_msg；
2. 类中变量 nav_num1 数值加1；
3. 类中变量 rightnum 数值等于 5；
4. 如果接收到的 twist_msg 的 x 方向上的线速度不等于 0：
    1. rightflag 置 1；
    2. delta_d 的变量赋值 `(twist_msg.linear.y * self.linear_yk + twist_msg.linear.z * self.linear_zk) * self.linear_allk`；
    1. 如果接收到的 twist_msg 的 x 方向上的线速度等于 0：
        1. rightflag 置零。

方法9：车辆运动控制（MoveCtrl）

1. 将订阅到的 speed 消息 赋值给类中变量 ctrl_command 的 driving_speed 分量；
2. 将订阅到的 angle 消息 赋值给类中变量 ctrl_command 的 steering_angle 分量；
3. 使用发布者 （ctrl_publisher） 发布 类中变量 ctrl_command；
4. 在终端打印：move 和 类中变量 ctrl_command 的 driving_speed 分量。

方法10：车辆模式切换（Setmode）

1. 如果接受订阅消息的 mode 为 WHEEL.TRANSVERSE：
    1. 调用 TxCommand 方法，传入参数 3，即：
        1. 终端打印日志：tx command num: 3；
        2. 类中变量 TX_num 赋值为 3，并通过发布者 TX_num_publisher 发布 TX_num。
2. 如果接受订阅消息的 mode 为 WHEEL.VERTICAL：
    1. 调用 TxCommand 方法，传入参数 4，即：
        1. 终端打印日志：tx command num: 4；
        2. 类中变量 TX_num 赋值为 4，并通过发布者 TX_num_publisher 发布 TX_num。

方法11：前进时的车辆速度决策处理（SpeedGoHandler）

1. 如果当前的 leftnum 大于等于 0：
    1. leftnum 做自减1操作；
2. 如果当前的 leftnum 小于 0：
    1. leftflag 置零；
3. 如果当前的 rightnum大于等于 0：
    1. rightnum做自减1操作；
4. 不满足上述情况：
    1. 类中变量 rightflag 置零；
    2. 终端打印：leftflag；
    3. 终端打印：rightflag；
    4. 如果类中变量 nav_num < 150 或者 类中变量 nav_num1 < 150 ，并且 类中变量 leftflag 和 类中变量 rightflag 均等于 0：
        1. 类中变量 ul_overtime 置 0；
        2. 类中变量 speed 取 1；
        3. 类中变量 delta_d 置 0.0；
    5. 如果类中变量 leftflag 等于 1并且 类中变量 rightflag 等于 1：
        1. 类中变量 ul_overtime 置0；
        2. 类中变量 speed 取1；
        3. 类中变量 delta_d 取 类中变量 delta_d 和类中变量 delta_d1的一半值；
    6. 如果类中变量 leftflag 等于 1并且 类中变量 rightflag 等于 0：
        1. 类中变量 ul_overtime 置0；
        2. 类中变量 speed 取1；
        3. 类中变量 delta_d 不变 ；
    7. 如果类中变量 leftflag 等于 0 并且 类中变量 rightflag 等于 1：
        1. 类中变量 ul_overtime 置0；
        2. 类中变量 speed 取1；
        3. 类中变量 delta_d 取类中变量 delta_d1；
    8. 如果类中变量 leftflag 等于 0 并且 类中变量 rightflag 等于 0：
        1. 类中变量 speed 置 0；

        2. 类中变量 delta_d 取 0.0；
    5. 如果类中变量 speed 为 0，且类中变量 ul_num1 小于等于 类中变量 far_ul 且类中变量 ul_num2 小于等于 类中变量 far_ul：
        1. 终端打印日志：`now gowork is finish,visual`；
        2. 类中变量 nav_num 置0；
        3. 类中变量 nav_num1置0；
        4. 关闭 launchgo 所对应的 launch 文件；
        5. 调用 Startvisual() 函数并将 CarTowards.BACK 传递给函数参数 cartowards：
            1. 使用 launchback 这个变量来标识 hfvisual_back.launch 文件；
            2. 使用 launchback 的 start() 方法来启动该文件；
            3. 将 launch_state 状态设置为 AUTOState.BACKWORK；
            4. 终端打印日志：`back visual started`。
        6. 类中变量 dis_reset_flag 置 0；
        7. 类中变量 current_state 设为 AUTOState.BACKWORK；
    6. 如果类中变量 speed 为 0，且满足（类中变量 ul_num1 大于等于类中变量 far_ul） 或者 （类中变量 ul_num2 大于类中变量 far_ul）之一：
        1. 类中变量 speed 设为 1.0；
        2. 类中变量 delta_d 设为 0.0；
    7. 类中变量 ul_overtime 自加 1；
    8. 如果类中变量 ul_overtime 大于等于 20：
        1. 类中变量 speed 设为 0.0；
        2. 关闭 launchgo 对应节点；
        3. 终端打印日志：now gowork is error；
        4. 类中变量 current_state 设为 AUTOState.ERROR；
    9. 如果 （类中变量 ul_num1 小于类中变量 near_ul 并且类中变量 ul_num1 大于 10）或者 （类中变量 ul_num2 小于类中变量 near_ul 并且类中变量 ul_num2 大于 10）：
        1. 类中变量 speed 等于 0.0；
        2. 类中变量 delta_d 等于 0.0；
        3. 关闭 launchgo 对应节点；
        4. 终端打印日志：now gowork is error；
        5. 类中变量 current_state 设为 AUTOState.ERROR；
    10. 调用 MoveCtrl() 函数，将类中变量 speed 和类中变量 delta_d 分别当作函数参数 speed 和 angle 传入：
        1. 将订阅到的 speed 消息 赋值给类中变量 ctrl_command 的 driving_speed 分量；
        2. 将订阅到的 angle 消息 赋值给类中变量 ctrl_command 的 steering_angle 分量；
        3. 使用发布者 （ctrl_publisher） 发布 类中变量 ctrl_command；
        4. 在终端打印： move 和 类中变量 ctrl_command 的 driving_speed 分量。

方法12：后退时的车辆速度决策处理（SpeedBackHandler）

1. 如果当前的 leftnum 大于等于 0：
    1. leftnum 做自减1操作；
2. 如果当前的 leftnum 小于 0：
    1. leftflag 置零；
3. 如果当前的 rightnum大于等于 0：
    1. rightnum做自减1操作；
4. 不满足上述情况：
    1. rightflag 置零；
    2. 终端打印： leftflag
    3. 终端打印： rightflag
5. 如果类中变量 nav_num < 300 或者 类中变量 nav_num1 < 300，并且 类中变量 leftflag 和 类中变量 rightflag 均等于 0：

    1. 类中变量 ul_overtime 置 0；
    2. 类中变量 speed 取 `-1`；
    3. 类中变量 delta_d 置 0.0；
6. 如果类中变量 leftflag 等于 1并且 类中变量 rightflag 等于 1：
    1. 类中变量 ul_overtime 置 0；
    2. 类中变量 speed 取 `-1`；
    3. 类中变量 delta_d 取 类中变量 delta_d 和类中变量 delta_d1的一半值；
7. 如果类中变量 leftflag 等于 1并且 类中变量 rightflag 等于 0：
    1. 类中变量 ul_overtime 置0；
    2. 类中变量 speed 取 `-1`；
    3. 类中变量 delta_d 不变；
8. 如果类中变量 leftflag 等于 0 并且 类中变量 rightflag 等于 1：
    1. 类中变量 ul_overtime 置0；
    2. 类中变量 speed 取 `-1`；
    3. 类中变量 delta_d 取类中变量 delta_d1；
9. 如果类中变量 nav_num 大于等于 300，且类中变量 nav_num1 大于等于300，且类中变量 leftflag 等于 0 并且 类中变量 rightflag 等于 0 且：
    1. 类中变量 speed 设为 0.0；
    2. 类中变量 delta_d 取 0.0；
10. 如果类中变量 speed 为 0，且类中变量 ul_num1 小于等于 类中变量 far_ul 且类中变量 ul_num2 小于等于 类中变量 far_ul：
    1. 终端打印日志：`now backwork is finish,visual`
    2. 类中变量 nav_num 置0；
    3. 类中变量 nav_num1置0；
    4. 关闭 launchback 所对应的 launch 文件；
    5. 将 launch_state 状态设置为 AUTOState.FINISH；

        终端打印日志：`back visual started`。
11. 如果类中变量 speed 为 0.0，且满足（类中变量 ul_num1 大于类中变量 far_ul） 或者 （类中变量 ul_num2 大于类中变量 far_ul）之一：
    1. 类中变量 speed 设为 `-1.0`；
    2. 类中变量 delta_d 设为 0.0；
12. 类中变量 ul_overtime 自加1；
13. 如果类中变量 ul_overtime 大于等于 20：
    1. 类中变量 speed 设为 0.0；
    2. 关闭 launchback 对应节点；
    3. 终端打印日志：`now backwork is error`；
    4. 类中变量 current_state 设为 AUTOState.ERROR；
14. 如果 （类中变量 ul_num1 小于类中变量 near_ul 并且类中变量 ul_num1 大于 10）或者 （类中变量 ul_num2 小于类中变量 near_ul 并且类中变量 ul_num2 大于 10） ：
    1. 类中变量 speed 等于 0.0；
    2. 类中变量 delta_d 等于 0.0；
    3. 关闭 launchback 对应节点；
    4. 终端打印日志：now backwork is error；
    5. 类中变量 current_state 设为 AUTOState.ERROR；
15. 调用 MoveCtrl() 函数，将类中变量 speed 和类中变量 `-delta_d` 分别当作函数参数 speed 和 angle 传入：
    1. 将订阅到的 speed 消息 赋值给类中变量 ctrl_command 的 driving_speed 分量；
    2. 将订阅到的 angle 消息 赋值给类中变量 ctrl_command 的 steering_angle 分量；
    3. 使用发布者 （ctrl_publisher） 发布 类中变量 ctrl_command；
    4. 在终端打印：move 和 类中变量 ctrl_command 的 driving_speed 分量。

方法13：StopToWaitInit()

初始化类中变量：

```
nav_num = 0;
nav_num1 = 0;
leftflag = 0;
rightflag = 0;
leftnum = 0;
rightnum = 0;
setmodenum = 0;
distance = 0;
dis_reset_flag = 0;
delta_d = 0;
delta_d1 = 0;
speed = 0;
rx_num = 0
```

方法 14：MainLoop()（主循环参数）

当 ros 节点没有退出：

1. 终端打印日志：current_state（类中变量）
2. 终端打印日志：near_ul（类中变量）；
3. 如果类中变量current_state （当前状态）等于 AUTOState.WAITING：
   如果类中变量 rx_num 等于 1：
    1. 将该变量置0；
    2. 使用 VideoCapture 类来初始化四个视频捕获对象，分别用于从四个不同的视频设备（摄像头）中捕获视频流，四个对象分别为：cap1、cap2、cap3 和 cap4；
    3. 如果有一个摄像头打开失败，终端打印：`camera is error`；
        1. 调用 TxCommand() 函数，传入参数 1：
        2. 终端打印日志：tx command num: 1；
        3. 类中变量 TX_num 赋值为 1，并通过发布者 TX_num_publisher 发布 TX_num。
    4. 如果全部连接成功：
        1. 终端打印日志：`camera is conntect`；
        2. 使用 release 方法释放摄像头资源；
        3. 调用 TxCommand() 函数，传入参数 2：
            1. 终端打印日志：tx command num: 2；
            2. 类中变量 TX_num 赋值为 2，并通过发布者 TX_num_publisher 发布 TX_num。
        4. 调用 Startvisual() 函数，并将 CarTowards.GO 作为参数传递给 cartowards：
            1. 使用 launchgo 这个变量来标识 hfvisual_back.launch 文件；
            2. 使用 launchgo 的 start() 方法来启动该文件；
            3. 将 launch_state 状态设置为 AUTOState.GOWORK；
            4. 终端打印日志：`go visual started`。
        5. 类中变量 current_state 设为 AUTOState.ALIGNMENT；
4. 如果类中变量 current_state （当前状态）等于 AUTOState.ALIGNMENT：
    1. 终端打印日志：`leftflag` 的值；
    2. 终端打印日志：`rightflag` 的值；
    3. 如果类中变量 rx_num 等于 2：
        1. 将该变量置0；

2. 调用 TxCommand() 函数，传入参数 8：
    1. 终端打印日志：tx command num: 8；
    2. 类中变量 TX_num 赋值为 8，并通过发布者 TX_num_publisher 发布 TX_num。
    3. 类中变量 current_state 设为 `AUTOState.GOWORK`；
4. 如果类中变量 nav_num 大于 50 且 类中变量 nav_num1 大于 50 且 类中变量 leftflag 等于0 且 类中变量 rightflag 等于0：
    1. 终端打印日志：`TX: alignment_error`；
    2. 调用 TxCommand() 函数，传入参数 6：
        1. 类中变量 TX_num 赋值为 6；
        2. 通过发布者 TX_num_publisher 发布 TX_num。
5. 如果类中变量 leftflag 等于 1 且 类中变量 rightflag 等于1：
    1. 终端打印日志：`TX: alignment_ready`；
    2. 调用 TxCommand() 函数，传入参数 5：
        1. 类中变量 TX_num 赋值为 5；
        2. 通过发布者 TX_num_publisher 发布 TX_num。

5. 如果类中变量 current_state （当前状态）等于 AUTOState.GOWORK：
    1. 终端打印日志：`dis_reset_flag`；
    2. 终端打印日志：dis_reset_flag（类中变量的值）；
    3. 如果类中变量 setmodenum 的值为 0：
        1. 调用方法 Setmode 并传入参数WHEEL.VERTICAL：
            1. 调用 TxCommand 方法，传入参数 4，即：
                1. 终端打印日志：tx command num: 4；
                2. 类中变量 TX_num 赋值为 4，并通过发布者 TX_num_publisher 发布 TX_num。
    4. 如果类中变量 setmodenum 的值为 1 且 类中变量 dis_reset_flag 的值为0：
        1. 设置类中变量 roll_speed 为 1；
        2. 设置类中变量 speed 为 1；
        3. 设置类中变量 ul_num1为 0；
        4. 设置类中变量 ul_num2 为 0；
        5. 设置类中变量 distance为 0；
        6. 调用 TxCommand() 函数，传入参数 9：
            1. 终端打印日志：tx command num: 9；
            2. 类中变量 TX_num 赋值为 9，并通过发布者 TX_num_publisher 发布 TX_num。
    5. 如果类中变量 dis_reset_flag 的值为 1：
        1. 调用函数 SpeedGoHandler();
6. 如果类中变量 current_state 为 AUTOState.BACKWORK：
    1. 如果类中变量 setmodenum 的值为 1 且 类中变量 dis_reset_flag 的值为0：
        1. 设置类中变量 roll_speed 为 `-1`；
        2. 设置类中变量 speed 为 `-1`；
        3. 设置类中变量 ul_num1为 0；
        4. 设置类中变量 ul_num2 为 0；
        5. 设置类中变量 distance为 0；
        6. 调用 TxCommand() 函数，传入参数 9：
            1. 终端打印日志：tx command num: 9；
            2. 类中变量 TX_num 赋值为 9，并通过发布者 TX_num_publisher 发布 TX_num。
    2. 如果类中变量 dis_reset_flag 的值为 1：
        1. 如果类中变量 nav_num 大于 100 且 类中变量 nav_num1 大于 100：
            1. 调用函数 SpeedBackHandler()
        2. 如果类中变量 current_state 等于 AUTOState.FINISH：

1. 调用方法 MoveCtrl，并将 speed = 0， angle = 0 当作参数传入：
    1. 将订阅到的 speed 消息 赋值给类中变量 ctrl_command 的 driving_speed 分量；
    2. 将订阅到的 angle 消息 赋值给类中变量 ctrl_command 的 steering_angle 分量；
    3. 使用发布者 （ctrl_publisher） 发布 类中变量 ctrl_command；
    4. 在终端打印：move 和 类中变量 ctrl_command 的 driving_speed 分量。
7. 如果类中变量 current_state 为 AUTOState.ERROR：
    1. 类中变量 nav_num 等于 0；
    2. 类中变量 nav_num1 等于 0；
    3. 调用方法 MoveCtrl，并将 speed = 0， angle = 0 当作参数传入：
        1. 将订阅到的 speed 消息 赋值给类中变量 ctrl_command 的 driving_speed 分量；
        2. 将订阅到的 angle 消息 赋值给类中变量 ctrl_command 的 steering_angle 分量；
        3. 使用发布者 （ctrl_publisher） 发布 类中变量 ctrl_command；
        4. 在终端打印：move 和 类中变量 ctrl_command 的 driving_speed 分量。
8. 终端打印：ul_num1:ul_num1 （类中变量的值）；
9. 终端打印：ul_num1:ul_num2 （类中变量的值）；
10. 终端打印：ul_num1:ul_overtime （类中变量的值）；
11. 循环休眠；

3、motion_ctrl.py

1、涉及的自定义消息类型

1.1 CtrlComand.msg

行驶速度和转向角度：

```
float32 driving_speed
float32 steering_angle
```

1.2 ModeSwitch.msg

目标模式：行驶、转换、旋转

```
uint8 target_mode
uint8 MOTION_MODE_STEERING = 0
uint8 MOTION_MODE_TRANSLATION = 1
uint8 MOTION_MODE_ROTATION = 2
```

2、程序解析

2.1 类

类1：MotionController

方法1：初始化方法（init）

1. 初始化节点名为：`motion_ctrl`；
2. 获取参数服务器中`~serial_port`的值，如果没有，则将类中变量 serial_port 的值设为`/dev/ttyTHS1`；
3. 获取参数服务器中`~serial_baudrate`的值，如果没有，则将类中变量 serial_baudrate 的值设为 115200；
4. 获取参数服务器中`~control_rate`的值，如果没有，则将类中变量 control_rate 的值设为 10；
5. 定义一个 ROS 频率器控制对象 rate，并设置其值为`control_rate`，即：10
6. 设置类中变量 ctrl_count_down 的值为 0；
7. 设置类中变量 ctrl_command_str 的值为空字符串；
8. 设置类中变量列表 target_mode 的值为`["ST", "TL", "RT"]`（TS：横向模式、VT：竖向模式和 RT）；
9. 设置类中变量 mode_num 的值为0；
10. 设置类中变量 mode_count 的值为0；
11. 设置类中变量 autoflag 的值为0；
12. 设置类中变量列表 tx_command的值为 ["#camera,ERROR*\n", "#jeston_check,OK*\n", "get_distance\n", "distance_reset\n", "alignment_error\n"]
13. 设置类中变量 tx_count 的值为0；
14. 设置类中变量 tx_num 的值为0；
15. 设置类中变量 infocnt 的值为0；
16. 初始化串口（名称、波特率、超时时间），如果失败，在日志中将错误记录为`ERROR: fail to open control serial port`；
17. 创建发布者：
    1. mode_fb_int8_publisher：发送 Int8 类型消息，话题名为 “mode_fb_num”；
    2. rx_command_publisher：发送 Int8 类型消息，话题名为 “rx_command”；
    3. distance_publisher：发送 Float32 类型消息，话题名为 “distance”；
    4. carinfo_publisher：发送 String 类型消息，话题名为 “car_info”；
18. 创建订阅者：
    1. ctrl_subscriber：
        1. 订阅话题名为 “ctrl_command”，消息类型为 CtrlCommand （自定义消息类型）的消息，回调函数为 CtrlCommandHandler；
           回调函数：
            1. 接收 CtrlCommand 消息类型的变量为 ctrl_msg；
            2. 如果类中变量 ctrl_count_down 等于 0：
                1. 对接收到的 ctrl_msg 中的 driving_speed 保留两位小数，赋值给类中变量driving_speed；
                2. 对接收到的 ctrl_msg 中的 steering_angle 保留两位小数，赋值给类中变量steering_angle；
                3. 对 driving_speed 和 steering_angle 进行格式处理，形成字符串`"move ${speed} ${angle}"`的格式并赋值给类中变量 ctrl_command_str；
                4. 如果类中变量 autoflag 的值等于 1：
                    1. 类中变量 ctrl_count_down 清 0；

2. 终端打印日志："TX:" + 类中变量：ctrl_command_str；
3. 串口写入类中变量 ctrl_command_str 的值

2. 订阅话题名为“tx_command”，消息类型为 Int8 的消息，回调函数为 TxCommandHandler；
回调函数：
1. 如果类中变量 tx_num 的值不等于 tx_num：
1. 将 tx_num 赋给类中的变量 tx_num ；
2. 类中变量 tx_count 清零；
2. 如果类中变量 tx_count 等于 0 且 tx_num的值大于 0：
1. 终端打印：tx_command 列表中的元素，元素下标为 tx_num.data-1；
2. 串口写入 tx_command 列表中的元素，元素下标为 tx_num.data-1；
3. 类中变量 tx_count 自加1；
4. 如果类中变量 tx_count 等于 3 ： 将该变量清0；

2. mode_subscriber：订阅话题名为“mode_switch”，消息类型为 ModeSwitch（自定义消息类型）的消息，回调函数为 ModeSwitchHandler；
回调函数：
1. 接收 CtrlCommand 消息类型的变量为 mode_index消息的 target_mode；
2. 如果类中变量 mode_num 不等于 mode_index：
1. 类中变量 mode_num 设为 mode_index 的 target_mode
2. 类中变量 mode_count 清零
3. 如果类中变量 mode_count 等于 0：
1. 类中变量 mode_num 设为 mode_index 消息的 target_mode；
2. mode_switch_command 的内容设置为 `"set_mode " + self.target_mode[self.mode_num]`（也就是 `"set_mode "` 加上 `"ST", "TL", "RT"` 其中之一）；
3. 终端打印 mode_switch_command 的内容；
4. 串口写入：mode_switch_command 的内容；
4. 类中变量 mode_count 的值自加1；
5. 如果类中变量 mode_count 的值为 3：对其进行清零操作。

19. 类中变量初始化：
last_speed、last_angle、driving_speed、steering_angle全部清零

方法2：析构函数（del）

程序结束关闭串口

方法3：SerialCtrlOnce()

终端打印日志：`"TX: " + self.ctrl_command_str`

方法4：CtrlCommandHandler()

1. 接收 CtrlCommand 消息类型的变量为 ctrl_msg；
2. 如果类中变量 ctrl_count_down 等于 0：
    1. 对接收到的 ctrl_msg 中的 driving_speed 保留两位小数，赋值给类中变量 driving_speed；
    2. 对接收到的 ctrl_msg 中的 steering_angle 保留两位小数，赋值给类中变量 steering_angle；
    3. 对 driving_speed 和 steering_angle 进行格式处理，形成字符串 `"move ${speed} ${angle}"` 的格式并赋值给类中变量 ctrl_command_str；
    4. 如果类中变量 autoflag 的值等于 1：
        1. 类中变量 ctrl_count_down 清 0；
        2. 终端打印日志："TX:" + 类中变量：ctrl_command_str；
        3. 串口写入类中变量 ctrl_command_str 的值

方法5：TxCommandHandler()

1. 如果类中变量 tx_num 的值不等于 tx_num：
    1. 将 tx_num 赋给类中的变量 tx_num ；
    2. 类中变量 tx_count 清零；
2. 如果类中变量 tx_count 等于 0 且 tx_num的值大于 0：
    1. 终端打印： tx_command 列表中的元素，元素下标为 tx_num.data-1；
    2. 串口写入 tx_command 列表中的元素，元素下标为 tx_num.data-1；
3. 类中变量 tx_count 自加1；
4. 如果类中变量 tx_count 等于 3： 将该变量清0；

方法6：ModeSwitchHandler()

1. 接收 CtrlCommand 消息类型的变量为 mode_index消息的 target_mode；
2. 如果类中变量 mode_num 不等于 mode_index：
    1. 类中变量 mode_num 设为 mode_index 的 target_mode
    2. 类中变量 mode_count 清零
3. 如果类中变量 mode_count 等于 0：
    1. 类中变量 mode_num 设为 mode_index 消息的 target_mode；
    2. mode_switch_command 的内容设置为 `"set_mode " + self.target_mode[self.mode_num]` （也就是 `"set_mode "` 加上 `"ST", "TL", "RT"` 其中之一）；
    3. 终端打印 mode_switch_command 的内容；
    4. 串口写入： mode_switch_command 的内容；
4. 类中变量 mode_count 的值自加1；
5. 如果类中变量 mode_count 的值为 3：对其进行清零操作。

方法7：MainLoop()

节点未关闭：

1. 如果类中变量 autoflag 等于 0：
    1. start_command 赋初值 `program_started\n`；
    2. 终端打印 start_command 的内容；
    3. 串口写入 start_command 的内容。
2. 从串口逐行读取内容，存放入 feedback；

3. 如果读取的 feedback 内容长度大于1：
    1. 终端打印：`"RX: "` + feedback 内容；
    2. 如果 feedback 内容是：`#program_started,OK*\n`：类中变量 autoflag 设为 1；
    3. 如果 feedback 内容是：`auto_work\n`：
        1. 终端打印：`auto work OK`；
        2. 类中变量 rx_command设为 1；
        3. 使用发布者 rx_command_publisher 发布 类中变量 rx_command；
    4. 如果 feedback 内容是：`stop_work\n`：
        1. 终端打印：`stop work OK`；
        2. 类中变量 rx_command设为 3；
        3. 使用发布者 rx_command_publisher 发布 类中变量 rx_command；
    5. 如果 feedback 内容是：`#set_mode,OK*\n`：
        1. 终端打印：`mode OK`；
        2. 类中变量 mode_fb_int8 设为 1；
        3. 使用发布者 mode_fb_int8_publisher 发布 类中变量 mode_fb_int8；
    6. 如果 feedback 内容是：`#distance_reset,OK*\n`：
        1. 终端打印：`reset OK`；
        2. 类中变量 rx_command 设为 2；
        3. 使用发布者 rx_command_publisher 发布 类中变量 rx_command；
    7. 除去以上情况：
        1. 对 feedback 先按 `*` 分段，并取第一段，按照 `,` 再次进行分段，最后将分段的结果赋值给列表 RX_DIS；
        2. 如果 RX_DIS 的第一段是 `#get_distance`，并且 RX_DIS 的第二段是 `OK`：
            1. 将 RX_DIS 的第三段转化为浮点型，并赋值给类中变量 distance；
            2. 终端打印：类中变量 distance 的值；
            3. 使用发布者 distance_publisher 发布类中变量 distance；
        3. 如果 RX_DIS 的第一段是 `#move`，并且 RX_DIS 的第二段是 `OK`：
            1. 将 RX_DIS 的第 3 段转化为浮点型，并保留两位小数，处理后的数据赋值给类中变量 last_speed；
            2. 将 RX_DIS 的第 4 段转化为浮点型，并保留两位小数，处理后的数据赋值给类中变量 last_angle；
            3. 终端打印：`speed:` 加上类中变量 last_speed 的值；
            4. 终端打印：`angle:` 加上类中变量 last_angle 的值；
            5. 如果类中变量 last_speed 等于类中变量 driving_speed 并且类中变量 last_angle 等于类中变量 steering_angle：类中变量 ctrl_count_down 清 0；
        4. 如果 RX_DIS 的第一段是 `#read_carinfo`，并且 RX_DIS 的第二段是 `OK`：
            1. 类中变量 carinfo 设为 feedback；
            2. 如果发布者 carinfo_publisher 发布类中变量 carinfo；（是否行对错？该文件第164行）
4. 如果类中变量 autoflag 的值为 1
    1. 如果类中变量 infocnt 的值为 0：
        1. carinfo_command 赋值为 `read_carinfo\n`；
        2. 终端打印 carinfo_command 的值；
        3. 向串口写入 carinfo_command 的值；
        4. 类中变量 infocnt 自加 1
    2. 如果类中变量 infocnt 的值不为 0：
        1. 类中变量 infocnt 自加 1
        2. 如果类中变量 infocnt 的值等于 15：

1. 类中变量 infocnt 的值清零

5. 按照 rate 设置的频率循环休眠

4、joystick_ctrl.py

1、涉及的自定义消息类型

1.1 CtrlComand.msg

行驶速度和转向角度：

```
float32 driving_speed
float32 steering_angle
```

1.2 ModeSwitch.msg

目标模式：行驶、转换、旋转

```
uint8 target_mode
uint8 MOTION_MODE_STEERING = 0
uint8 MOTION_MODE_TRANSLATION = 1
uint8 MOTION_MODE_ROTATION = 2
```

2、程序解析

类

类1：JoystickCtrl

方法1：初始化 (init)

1. 初始化节点，设为匿名，设置为：joystick_ctrl；
2. 类中列表变量 previous_buttons 设为 [0, 0, 0, 0]；
3. 类中变量 speed_range 从参数服务器中获取 `~speed_range`，默认为 1；
4. 类中变量 angle_range 从参数服务器中获取 `~angle_range`，默认为 1；
5. 创建订阅者：
    1. joystick_subscriber：订阅话题名为“joy”，消息类型为 Joy（sensor_msgs）的消息，回调函数为 JoystickMsgHandler；
       回调函数：
        1. 接收 Joy 消息类型的变量为 joystick_msg；
        2. 尝试：
            1. 将类中变量 ctrl_command 的分量 steering_angle 设为 `-1 * joystick_msg.axes[0] * self.angle_range` 的值；
            2. 将类中变量 ctrl_command 的分量 driving_speed 设为 `joystick_msg.axes[3] * self.speed_range` 的值；
            3. 调用 ctrl_publisher 发布者发布类中变量 ctrl_command 的内容；
        3. 设置失败则日志中记录：`ERROR: fail to control by joystick`
        4. 如果 joystick_msg 的分量 axes[5] 的值转换为 int 后 不等于 0：

1. 如果类中变量 joystick_msg 的分量 buttons[0] 的值等于 1 且类中变量 previous_buttons[0] 的值等于 0：
    1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_STEERING；
    2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
2. 如果类中变量 joystick_msg 的分量 buttons[1] 的值等于 1 且类中变量 previous_buttons[1] 的值等于 0：
    1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_TRANSLATION；
    2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
3. 如果类中变量 joystick_msg 的分量 buttons[2] 的值等于 1 且类中变量 previous_buttons[2] 的值等于 0：
    1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_ROTATION；
    2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
4. 不满足上述条件，不执行任何操作。

5. 更新类中变量 previous_buttons 中的值：具体为存储当前值为下一步的先前值：

```
self.previous_buttons[0] =
int(joystick_msg.buttons[0])
self.previous_buttons[1] =
int(joystick_msg.buttons[1])
self.previous_buttons[2] =
int(joystick_msg.buttons[2])
```

6. 定义类中变量 ctrl_command 的类型为 CtrlCommand()（自定义消息类型）
7. 定义类中变量 mode_switch 的类型为 ModeSwitch()（自定义消息类型）
8. 创建发布者：
    1. ctrl_publisher：发送 CtrlCommand 类型消息，话题名为 “ctrl_command”；
    2. mode_publisher：发送 ModeSwitch 类型消息，话题名为 “mode_switch”；

方法2：JoystickMsgHandler()

1. 接收 Joy 消息类型的变量为 joystick_msg；
2. 尝试：
    1. 将类中变量 ctrl_command 的分量 steering_angle 设为 `-1 * joystick_msg.axes[0] * self.angle_range` 的值；
    2. 将类中变量 ctrl_command 的分量 driving_speed 设为 `joystick_msg.axes[3] * self.speed_range` 的值；
    3. 调用 ctrl_publisher 发布者发布类中变量 ctrl_command 的内容；
3. 设置失败则日志中记录：`ERROR: fail to control by joystick`

如果 joystick_msg 的分量 axes[5] 的值转换为 int 后 不等于 0：

1. 如果类中变量 joystick_msg 的分量 buttons[0] 的值等于 1 且类中变量 previous_buttons[0] 的值等于 0：
    1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_STEERING；

            2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
        2. 如果类中变量 joystick_msg 的分量 buttons[1] 的值等于 1 且类中变量 previous_buttons[1] 的值等于 0：
            1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_TRANSLATION；
            2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
        3. 如果类中变量 joystick_msg 的分量 buttons[2] 的值等于 1 且类中变量 previous_buttons[2] 的值等于 0：
            1. 设置类中变量 mode_switch 的分量 target_mode 的值为 ModeSwitch.MOTION_MODE_ROTATION；
            2. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值；
        4. 不满足上述条件，不执行任何操作。
    4. 更新类中变量 previous_buttons 中的值：具体为存储当前值为下一步的先前值：

```
self.previous_buttons[0] = int(joystick_msg.buttons[0])
self.previous_buttons[1] = int(joystick_msg.buttons[1])
self.previous_buttons[2] = int(joystick_msg.buttons[2])
```

3、方法3：MainLoop()

循环

5、lining_ctrl.py (路线跟踪)

1、涉及的自定义消息类型

1.1 CtrlComand.msg

行驶速度和转向角度：

```
float32 driving_speed
float32 steering_angle
```

1.2 ModeSwitch.msg

目标模式：行驶、转换、旋转

```
uint8 target_mode
uint8 MOTION_MODE_STEERING = 0
uint8 MOTION_MODE_TRANSLATION = 1
uint8 MOTION_MODE_ROTATION = 2
```

2、程序解析

类

类1：PDController()

方法1：初始化（init）（参数列表：`kp`，`kd`，`output_max`）

1. 类中变量 kp 的值设为传入参数 kp；
2. 类中变量 kd 的值设为传入参数 kd；
3. 类中变量 output_max 的值设为传入参数 output_max；
4. 类中列表变量 error 的值初始化为 `[0，0，0]`；
5. 类中变量 feedback 的值设为 0；
6. 类中变量 reference 的值设为 0；
7. 类中变量 output 的值设为 0；

方法2：Clear()（清零方法）

1. 类中列表变量 error 的值设为 `[0，0，0]`；
2. 类中变量 feedback 的值设为传入参数 0；
3. 类中变量 reference 的值设为传入参数 0；

方法3：CalcError()

1. 类中列表变量 error 的第2个参数为列表变量 error 的第1个参数值；
2. 类中列表变量 error 的第1个参数为类中变量 reference 减去类中变量 feedback 的值；

方法4：CalcOutPut()

1. 类中列表变量 output 的值为 `self.error[0] * self.kp + (self.error[0] - self.error[1]) * self.kd`；
   这个公式是离散时间PID控制器的一部分。它用来计算PID控制器的输出。具体来说：
   1. self.error[0] 表示当前时刻的误差（偏差）值。
   2. self.error[1] 表示上一个时刻的误差（偏差）值。
   3. self.kp 是比例增益（Proportional gain）。
   4. self.kd 是微分增益（Derivative gain）。
   5. 公式的第一部分 self.error[0] * self.kp 计算了比例控制器的输出，即根据当前误差值计算出的控制量。

   公式的第二部分 (self.error[0] - self.error[1]) * self.kd 计算了微分控制器的输出，即当前误差值与上一个时刻的误差值之差乘以微分增益。
   这两部分的和即为PID控制器的输出，用来调节系统以减小误差。
2. 保持类中变量 output 的值始终在区间 `[-output_max，output_max]` 中，超出范围，则设为最接近的区间边界值。

类2：LiningState(枚举类)

```
WAITING = 0          # 等待路线信息
LINING = 1           # 跟随线路
LEAVE = 2            # 离开线路
SWITCHING = 3        # 切换线路
ERROR = 4            # 线路错误
```

类3：LiningController

方法1：初始化 (init)

1. 初始化 ros 匿名节点为 lining_ctrl；
2. 初始化类中变量 d_weight 的值为参数服务器中的 `~d_weight`，未找到设为 1；
3. 初始化类中变量 theta_weight 的值为 `1 - self.d_weight`；
4. 初始化类中变量 kp 的值为参数服务器中的 `~kp` 值，未找到设为 `-0.5`；
5. 初始化类中变量 kd 的值为参数服务器中的 `~kd` 值，未找到设为 `-13.3`；
6. 初始化类中变量 place_length 的值为参数服务器中的 `~place_length` 值，未找到设为 `5`；
7. 初始化类中变量 place_width 的值为参数服务器中的 `~place_width` 值，未找到设为 `4`；
8. 初始化类中变量 lining_speed 的值为参数服务器中的 `~lining_speed` 值，未找到设为 `1`；
9. 初始化类中变量 control_rate 的值为参数服务器中的 `~control_rate` 值，未找到设为 `10`；
10. 初始化ROS 频率控制器 rate 的值为类中变量 `control_rate` 值；
11. 初始化类中变量 timeout 的值为参数服务器中的 `~timeout` 值 * *1000*，未找到设为 `1`；
12. 初始化类中变量 timeout_cnt 的值为类中变量 `timeout` 值；
13. 初始化类中变量 autoflag 的值为 *0*；
14. 初始化类 PDController的 对象 PD_controller 的值为 PDController(self.kp, self.kd, 1)；
15. 初始化类中变量 lining_flag 的值为 `LiningState.WAITING`；
16. 初始化类中变量 distance 的值为 *0*；
17. 初始化类中变量 dis_reset_flag 的值为 *0*；
18. 初始化类中变量 rx_command_num 的值为 *0*；
19. 初始化类中变量 mode_change_value 的值为 *0*；
20. 初始化类中变量 switch_time 的值为当前时间戳；
21. 初始化类中变量 leaveflag 的值为 *0*；
22. 初始化类中变量 last_distance 的值为 *10000*；
23. 创建发布者：
    1. ctrl_publisher：发送 CtrlCommand 类型消息，话题名为 “ctrl_command”；
    2. mode_publisher：发送 ModeSwitch 类型消息，话题名为 “mode_switch”；
       将 mode_switch 的 target_mode 分量设为 `ModeSwitch.MOTION_MODE_STEERING`；
    3. tx_command_publisher：发送 ModeSwitch 类型消息，话题名为 “tx_command”；
24. 创建订阅者：
    1. twist_subscriber：订阅话题名为 “duihang”，消息类型为 Twist 的消息，回调函数为 LiningMsgHandler；
       回调函数：
       1. 接收 Twist 消息类型的变量为 twist_msg；
       2. 类中变量 timeout_cnt 设置为 类中变量 timeout 的值；
       3. 如果类中变量 lining_flag 等于 `LiningState.LINING`：
          1. 如果类中变量 twist_msg 的分量 linear.x 的值不等于 *0*：
             1. weighted_angle 的值为 `twist_msg.angular.y * self.d_weight + twist_msg.angular.z * self.theta_weight`；

2. 类 PDController 对象 PD_controller 的 feedback 值为上述的 weighted_angle；
3. 类中变量 leaveflag 的值为0；
4. 类中变量 lining_speed 的值为 0.7；

2. 否则：
1. 对象 PD_controller 中的部分值（error、feedback 和 reference）全部清0；
2. 如果 leaveflag 值为 0：
1. 将 leaveflag 置1；
2. 将 last_distance 置为类中变量 distance 的值；
3. 终端打印：`leave==========flag`；
4. 终端打印：类中变量 last_distance 的值
3. 类中变量 lining_speed 的值设为 0.7；

4. 如果类中变量 lining_flag 等于 `LiningState.WAITING`：
1. 如果类中变量 twist_msg 的分量 linear.x 的值不等于 0：
1. 类中变量 lining_flag 设为 `LiningState.LINING`
2. weighted_angle 的值为 `twist_msg.angular.y * self.d_weight + twist_msg.angular.z * self.theta_weight`；
3. 类 PDController 对象 PD_controller 的 feedback 值为上述的 weighted_angle；
4. 类中变量 leaveflag 的值为0；
5. 类中变量 lining_speed 的值为 0.7；
6. 终端打印：`ready lining`。

2. 否则：
1. 对象 PD_controller 中的部分值（error、feedback 和 reference）全部清0；
2. 调用类的 TxCommand() 方法，传入参数5：
1. 终端打印：`tx command num:`
2. 终端打印：`num` 的值；
3. 类中变量 tx_command 的值设为 num；
4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
3. 终端打印：`error state`。

2. sub_mode_fb：订阅话题名为“mode_fb_num”，消息类型为 Int8 的消息，回调函数为 ModeFbHandler；
回调函数：
1. 接收 Int8 消息类型的变量为 mode_fb；
2. 类中变量 mode_change_value 的值设为 mode_fb 中的data 内容；
3. 终端打印：`mode change ok`。

3. sub_rx_command：订阅话题名为“rx_command”，消息类型为 Int8 的消息，回调函数为 RxCommandHandler；
回调函数：
1. 接收 Int8 消息类型的变量为 rx_num；
2. 类中变量 rx_command_num 的值设为 rx_num 中的data 内容；
3. 终端打印：`rx command num ok`；
4. 终端打印：类中变量 rx_command_num 的值。

4. sub_distance：订阅话题名为“distance”，消息类型为 Float32 的消息，回调函数为 GetDistanceHandler；

回调函数：

1. 接收 Float32 消息类型的变量为 dis；
2. 类中变量 distance 的值设为 dis 中的data 内容；
3. 终端打印：`get distance:`
4. 终端打印：类中变量 distance 的值。

方法2：LiningMsgHandler()

1. 接收 Twist 消息类型的变量为 twist_msg；
2. 类中变量 timeout_cnt 设置为 类中变量 timeout 的值；
3. 如果类中变量 lining_flag 等于 `LiningState.LINING`：
    1. 如果类中变量 twist_msg 的分量 linear.x 的值不等于 0：
        1. weighted_angle 的值为 `twist_msg.angular.y * self.d_weight + twist_msg.angular.z * self.theta_weight`；
        2. 类 PDController 对象 PD_controller 的 feedback 值为上述的 weighted_angle；
        3. 类中变量 leaveflag 的值为0；
        4. 类中变量 lining_speed 的值为 0.7；
    2. 否则：
        1. 对象 PD_controller 中的部分值（error、feedback 和 reference）全部清 0；
        2. 如果 leaveflag 值为 0：
            1. 将 leaveflag 置1；
            2. 将 last_distance 置为类中变量 distance 的值；
            3. 终端打印：`leave==========flag`；
            4. 终端打印：类中变量 last_distance 的值
        3. 类中变量 lining_speed 的值设为 0.7；
4. 如果类中变量 lining_flag 等于 `LiningState.WAITING`：
    1. 如果类中变量 twist_msg 的分量 linear.x 的值不等于 0：
        1. 类中变量 lining_flag 设为 `LiningState.LINING`
        2. weighted_angle 的值为 `twist_msg.angular.y * self.d_weight + twist_msg.angular.z * self.theta_weight`；
        3. 类 PDController 对象 PD_controller 的 feedback 值为上述的 weighted_angle；
        4. 类中变量 leaveflag 的值为0；
        5. 类中变量 lining_speed 的值为 0.7；
        6. 终端打印：`ready lining`。
    2. 否则：
        1. 对象 PD_controller 中的部分值（error、feedback 和 reference）全部清 0；
        2. 调用类的 TxCommand() 方法，传入参数5：
            1. 终端打印：`tx command num:`
            2. 终端打印：`num` 的值；
            3. 类中变量 tx_command 的值设为 num；
            4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
        3. 终端打印：`error state`

方法3：ModeFbHandler()

1. 接收 Int8 消息类型的变量为 mode_fb；
2. 类中变量 mode_change_value 的值设为 mode_fb 中的data 内容；
3. 终端打印：`mode change ok`。

方法4：RxCommandHandler()

1. 接收 Int8 消息类型的变量为 rx_num；
2. 类中变量 rx_command_num 的值设为 rx_num 中的data 内容；
3. 终端打印：`rx command num ok`；
4. 终端打印：类中变量 rx_command_num 的值。

方法5：GetDistanceHandler()

1. 接收 Float32 消息类型的变量为 dis；
2. 类中变量 distance 的值设为 dis 中的data 内容；
3. 终端打印：`get distance:`
4. 终端打印：类中变量 distance 的值。

方法6：ControlOnce()

话题超时处理：

1. 如果类中变量 timeout_cnt 的值大于 *0*：
    1. 类中变量 timeout_cnt 的值 减去 *1000* / self.control_rate；
2. 否则：
    1. 类中变量 lining_flag 的值设为 `LiningState.WAITING`；
    2. 对象 PD_controller 中的部分值（error、feedback 和 reference）全部清*0*；
    3. 终端打印：`error state: lining topic timeout`

路线控制：

1. 如果类中变量 lining_flag 的值等于`LiningState.LINING`：
    1. 终端打印：`start lining`；
    2. 如果类中变量 rx_command_num 的值等于 2 ，并且类中变量 dis_reset_flag 的值等于 *0*，并且类中变量 mode_change_value 的值等于 *1*：
        1. 类中变量 dis_reset_flag 的值等于 *1*
    3. 如果类中变量 dis_reset_flag 的值等于 *0* ，并且类中变量 mode_change_value 的值等于 *1*：
        1. 类中变量 distance 的值等于 *0*;
        2. 调用类的 TxCommand() 方法，传入参数4：
            1. 终端打印：`tx command num:`
            2. 终端打印：`num` 的值；
            3. 类中变量 tx_command 的值设为 num；
            4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
    4. 如果类中变量 mode_change_value 的值等于 *0* ：
        1. 调用方法 `SwitchModeCtrl()`，传入参数 `ModeSwitch.MOTION_MODE_STEERING`：
            1. 接收 ModeSwitch (自定义消息类型)消息类型的变量为 mode；
            2. 终端打印：`set mode num:`；

3. 终端打印：mode 的值；
4. 设置类中变量 mode_switch 的分量 target_mode 的值为 mode；
5. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值。

5. 如果类中变量 distance 的值大于等于 6000：
    1. 调用 MoveCtrl() 方法，传入参数（0，0），即：速度(speed)和角度(angle)：
        1. 接收参数的变量为 speed 和 angle；
        2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
        3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
        4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
    2. 类中变量 lining_flag 的值设为 `LiningState.LEAVE`；
    3. 类中变量 rx_command_num 的值设为 0；
    4. 类中变量 dis_reset_flag 的值设为 0；
    5. 类中变量 mode_change_value 的值设为 0；
6. 如果类中变量 distance 的值大于等于 3000：
    1. 终端打印：`delta dis:`；
    2. 终端打印：类中变量 distance 的值减去 类中变量 last_distance；
    3. 类中变量 lining_flag 的值设为 `LiningState.LEAVE`；
    4. 类中变量 rx_command_num 的值设为 0；
    5. 类中变量 dis_reset_flag 的值设为 0；
    6. 类中变量 mode_change_value 的值设为 0；
    7. 调用 MoveCtrl() 方法，传入参数（0，0），即：速度(speed)和角度(angle)：
        1. 接收参数的变量为 speed 和 angle；
        2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
        3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
        4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
7. 不满足以上情况，则：
    1. 终端打印：`delta dis:`；
    2. 终端打印：类中变量 distance 的值减去 类中变量 last_distance；
    3. 调用对象 PD_controller 的 CalcError() 方法：
        1. 类中列表变量 error 的第2个参数 为 列表变量 error 的第1个参数值；
        2. 类中列表变量 error 的第1个参数为类中变量 reference 减去类中变量 feedback 的值。
    4. 调用对象 PD_controller 的 CalcOutPut() 方法：
        1. 类中列表变量 output 的值为 `self.error[0] * self.kp + (self.error[0] - self.error[1]) * self.kd`；
        2. 保持类中变量 output 的值始终在区间 `[-output_max, output_max]` 中，超出范围，则设为最接近的区间边界值。
    5. 调用 MoveCtrl() 方法，传入参数（类中变量 lining_speed，类中变量 PD_controller 的 output），即：速度(speed)和角度(angle)：
        1. 接收参数的变量为 speed 和 angle；
        2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
        3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
        4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
    6. 调用类的 TxCommand() 方法，传入参数 3：
        1. 终端打印：`tx command num:`
        2. 终端打印：`num` 的值；

                3. 类中变量 tx_command 的值设为 num；
                4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
2. 如果类中变量 lining_flag 的值等于`LiningState.LEAVE`：
    1. 终端打印：`leave the line`；
    2. 如果类中变量 rx_command_num 的值等于 2 ，并且类中变量 dis_reset_flag 的值等于 0，并且类中变量 mode_change_value 的值等于 1：
        1. 设置 dis_reset_flag 为1
    3. 如果类中变量 dis_reset_flag 的值等于 0 ，并且类中变量 mode_change_value 的值等于 1：
        1. 类中变量 distance 的值等于 0;
        2. 调用类的 TxCommand() 方法，传入参数4：
            1. 终端打印：`tx command num:`
            2. 终端打印：`num` 的值；
            3. 类中变量 tx_command 的值设为 num；
            4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
    4. 如果类中变量 mode_change_value 的值等于 0 ：
        1. 调用方法`SwitchModeCtrl()`，传入参数`ModeSwitch.MOTION_MODE_ROTATION`：
            1. 接收 ModeSwitch (自定义消息类型)消息类型的变量为 mode；
            2. 终端打印：`set mode num:`；
            3. 终端打印：mode 的值；
            4. 设置类中变量 mode_switch 的分量 target_mode 的值为 mode；
            5. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值。
    5. 如果类中变量 distance 的值大于等于 3100.0：
        1. 调用 MoveCtrl() 方法，传入参数 （0，0） ，即：速度(speed)和角度(angle)：
            1. 接收参数的变量为 speed 和 angle；
            2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
            3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
            4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
        2. 类中变量 lining_flag 的值设为`LiningState.SWITCHING`；
        3. 类中变量 rx_command_num 的值设为 0；
        4. 类中变量 dis_reset_flag 的值设为 0；
        5. 类中变量 mode_change_value 的值设为 0；
    6. 不满足以上情况，则：
        1. 调用 MoveCtrl() 方法，传入参数 （0.5，0） ，即：速度(speed)和角度(angle)：
            1. 接收参数的变量为 speed 和 angle；
            2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
            3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
            4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
        2. 调用类的 TxCommand() 方法，传入参数 3：
            1. 终端打印：`tx command num:`
            2. 终端打印：`num` 的值；
            3. 类中变量 tx_command 的值设为 num；
            4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。

3. 如果类中变量 lining_flag 的值等于`LiningState.SWITCHING`：
    1. 终端打印：`switching the line`；
    2. 如果类中变量 rx_command_num 的值等于 2 ，并且类中变量 dis_reset_flag 的值等于 0，并且类中变量 mode_change_value 的值等于 1：
        1. 设置 dis_reset_flag 为1
    3. 如果类中变量 dis_reset_flag 的值等于 0 ，并且类中变量 mode_change_value 的值等于 1：
        1. 类中变量 distance 的值等于 0;
        2. 调用类的 TxCommand() 方法，传入参数4：
            1. 终端打印：`tx command num:`
            2. 终端打印：`num` 的值；
            3. 类中变量 tx_command 的值设为 num；
            4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
    4. 如果类中变量 mode_change_value 的值等于 0 ：
        1. 调用方法 `SwitchModeCtrl()`，传入参数 `ModeSwitch.MOTION_MODE_TRANSLATION`：
            1. 接收 ModeSwitch (自定义消息类型)消息类型的变量为 mode；
            2. 终端打印：`set mode num:`；
            3. 终端打印：mode 的值；
            4. 设置类中变量 mode_switch 的分量 target_mode 的值为 mode；
            5. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值。
        2. 类中变量 switch_time 的值设为当前时间戳；
    5. 如果类中变量 mode_change_value 的值等于 1，并且类中变量 dis_reset_flag 的值等于 1，并且当前时间戳 - 类中变量switch_time 的结果小于等于2：
        1. 调用 MoveCtrl() 方法，传入参数`（0，1）`，即：速度(speed)和角度(angle)：
            1. 接收参数的变量为 speed 和 angle；
            2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
            3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
            4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
    6. 如果类中变量 distance 的值大于等于 1350.0：
        1. 调用 MoveCtrl() 方法，传入参数 （0，0） ，即：速度(speed)和角度(angle)：
            1. 接收参数的变量为 speed 和 angle；
            2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
            3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
            4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；
        2. 类中变量 lining_flag 的值设为 `LiningState.LINING`；
        3. 类中变量 rx_command_num 的值设为 0；
        4. 类中变量 dis_reset_flag 的值设为 0；
        5. 类中变量 mode_change_value 的值设为 0；
        6. 类中变量 switch_time 的值设为当前时间戳；
    7. 不满足以上情况，则：
        1. 调用 MoveCtrl() 方法，传入参数 （0.5，1） ，即：速度(speed)和角度(angle)：
            1. 接收参数的变量为 speed 和 angle；
            2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
            3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；

4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；

2. 调用类的 TxCommand() 方法，传入参数 3：
    1. 终端打印：`tx command num:`
    2. 终端打印：`num` 的值；
    3. 类中变量 tx_command 的值设为 num；
    4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。

4. 不满足上述情况：
    1. 调用 MoveCtrl() 方法，传入参数（0，0），即：速度(speed)和角度(angle)：
        1. 接收参数的变量为 speed 和 angle；
        2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
        3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
        4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；

方法7：MoveCtrl()（参数：speed，angle）

1. 接收参数的变量为 speed 和 angle；
2. 类中变量 ctrl_command 的分量 driving_speed 设为 speed；
3. 类中变量 ctrl_command 的分量 steering_angle 设为 angle；
4. 调用发布者 tx_command_publisher 发布类中变量 tx_command 的值；

方法8：TxCommand ()（参数：num）

1. 终端打印：`tx command num:`
2. 终端打印：`num` 的值；
3. 类中变量 tx_command 的值设为 num；
4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。

方法9：SwitchModeCtrl()（参数：mode）

1. 接收 ModeSwitch (自定义消息类型)消息类型的变量为 mode；
2. 终端打印：`set mode num:`；
3. 终端打印：mode 的值；
4. 设置类中变量 mode_switch 的分量 target_mode 的值为 mode；
5. 调用发布者 mode_publisher 发布类中变量 mode_switch 的值。

方法10：启动关闭视觉导航程序：Startvisual()

1. 将临时生成的 UUID 赋值给变量 uuid；
2. 将生成的 UUID 传递给 ROS Launch 日志记录系统；
3. 使用 launchgo 这个变量来标识 `/home/nvidia/catkin_ws/src/mower_ctrl/lining_visual.launch` 文件；
4. 使用 launchgo 的 start() 方法来启动该文件；
5. 终端打印日志：`go visual started`

方法11：MainLoop()

1. 如果节点未关闭：
    1. 如果类中变量 rx_command_num 的值为 1：
        1. rx_command_num 的值设为0；
        2. 检测摄像头是否正常连接：
            1. cap 获取路径：`/dev/video-mid` 中的视频；
                1. 如果打开失败
                    1. 终端打印：camera is error
                    2. 调用方法TxCommand()，传入参数 1：
                        1. 终端打印：`tx command num:`
                        2. 终端打印：`num` 的值；
                        3. 类中变量 tx_command 的值设为 num；
                        4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
                2. 不满足上述情况：
                    1. 终端打印：camera is conntect
                    2. 释放 cap 资源；
                    3. 调用方法TxCommand()，传入参数 2：
                        1. 终端打印：`tx command num:`
                        2. 终端打印：`num` 的值；
                        3. 类中变量 tx_command 的值设为 num；
                        4. 使用发布者 tx_command_publisher 发布类中变量 tx_command 的值。
                    4. 设置类中变量 autoflag 的值为 1；
                    5. 调用 Startvisual() 方法：
                        1. 将临时生成的 UUID 赋值给变量 uuid；
                        2. 将生成的 UUID 传递给 ROS Launch 日志记录系统；
                        3. 使用 launchgo 这个变量来标识 `/home/nvidia/catkin_ws/src/mower_ctrl/lining_visual.launch` 文件；
                        4. 使用 launchgo 的 start() 方法来启动该文件；
                        5. 终端打印日志：`go visual started`
    2. 如果类中变量 rx_command_num 的值为 3 并且类中变量 autoflag 的值为 1：
        1. 类中变量 rx_command_num 的值设为 0；
        2. 关闭 launchgo 对应的节点；
        3. 类中变量 timeout_cnt 的值设为类中变量 timeout；
        4. 类中变量 autoflag 的值设为 0；
        5. 类中变量 lining_flag 的值设为 `LiningState.WAITING`；
        6. 类中变量 distance 的值设为 0；
        7. 类中变量 mode_change_value 的值设为 0；
        8. 类中变量 switch_time 的值设为 当前时间戳；
        9. 对象PD_controller 调用方法 Clear()：

1. 类中列表变量 error 的值设为 `[0, 0, 0]`；
2. 类中变量 feedback 的值设为传入参数 0；
3. 类中变量 reference 的值设为传入参数 0；

10. 终端打印：`launch stop OK`

3. 如果类中变量 autoflag 的值为 1：
    1. 调用 ControlOnce() 方法
4. ros 休眠

6、BYGPS_reader.py

1、程序解析

类：

类1：GPSReader

方法1：初始化（init）

1. 初始化 匿名 ros 节点：BYGPS_reader；
2. 初始化类中变量 serial_port 的值为参数服务器中的 `~serial_port`，未找到设为 `/dev/ttyTHS1`；
3. 初始化类中变量 serial_baudrate 的值为参数服务器中的 `~serial_baudrate`，未找到设为 `115200`；
4. 创建一个正则表达式对象，用于匹配包含逗号，或星号 * 的任意单个字符；
5. 打开指定串口，若失败则日志记录：ERROR: fail to open GPS serial port, retrying...；
6. 设置类中变量 gpsx = 3.85，gpsy = 1.97；
7. 初始化类中变量 control_rate 的值为参数服务器中的 `~control_rate`，未找到设为 `10`；
8. 设置ROS 频率控制器 rate 的值为 类中变量 control_rate 的值；
9. 创建发布者：
    1. pose2D_publisher：发送 Pose2D 类型消息，话题名为 “pub_mid_pos”；
    2. path_pub：发送 Path 类型消息，话题名为 “trajectory”；
10. 创建类中变量：pose2D_msg，类型为 Pose2D；
11. 创建类中变量：path_msg，类型为 Path；

方法2：校验数据：CheckData（参数：by_string，参数类型：string）

1. 将 by_string 按照 `*` 进行分割，并将分割后的子串存储在列表 by_string 中；
2. 如果 by_string 的长度不为 2：
    1. 返回False
3. 将 split_string[1] 按照默认的空白字符（空格、制表符、换行符等）进行分割，并将分割后的子串存储在列表 check_string 中；
4. 获取 split_string[0] 字符串中去掉开头字符后的子串，并将该子串存储在变量 data_to_checksum 中；
5. 将 check_sum 的值设为 0；
6. 循环遍历了 split_string[0] 字符串中除第一个字符外的所有字符，将每个字符的 ASCII 值与 check_sum 进行异或操作，最终得到了校验和 check_sum；
7. 校验：`return ("%08X" % check_sum) == check_string.upper()`
    1. 将 check_sum 格式化为一个 8 位的十六进制字符串（左侧）；
    2. 将 check_string 转换为大写形式的字符串（右侧）；
    3. 检查左侧格式化后的字符串是否等于右侧转换为大写形式的字符串。
8. 如果这两个字符串相等，则整个条件表达式的结果为 True，表示校验通过；否则结果为 False，表示校验失败。

方法3：ParseData（参数：by_string，参数类型：string）

1. 使用类中正则表达式 `field_delimiter_regex` 对字符串 by_string 进行分割，并将分割后的结果存储在列表 fields 中；
2. 从列表 fields 的第一个元素中取出一个子串，并将该子串赋值给变量 sentence_type；
3. 如果 sentence_type 的值为 `KSXT` （表示接收到的数据符合特定的类型（KSXT 类型））：
    1. 取出列表 fields 中索引为 10 到 11 的元素（不包括索引 12），然后与 [3, 3] 进行比较。如果这个切片的值等于 [3, 3]，则执行以下操作：
        1. 类中变量 pose2D_msg 的分量 x 设为强制转化为 float 类型后的 fields[14]；
        2. 类中变量 pose2D_msg 的分量 y 设为强制转化为 float 类型后的 fields[15]；
        3. 类中变量 pose2D_msg 的分量 theta 设为强制转化为 float 类型后的 fields[5]；
        4. 使用发布者 pose2D_publisher 发布更新后的 pose2D_msg 消息；
        5. 返回 true，表示处理成功。
    2. 不满足上述条件，返回 Fasle；
4. 不满足上述条件，返回 False。

方法4：DataUpdating（参数：path_pub、path_record，参数类型：Path，nav_msgs/Path）

1. 设置 current_time 为当前时间戳；
2. 配置姿态：
    1. 定义 PoseStamped() 类型的变量 pose，用于表示具有时间戳的位姿信息（姿态）；
    2. 初始化 pose 的各个分量：
        1. 将消息 pose 的时间戳设置为 current_time；
        2. 将消息 pose 的参考坐标系（frame_id）设置为 map；
        3. 将消息 pose 中位姿的 x 坐标设置为 类中变量 gpsx，表示物体在地图坐标系中的 x 轴位置；
        4. 将消息 pose 中位姿的 y 坐标设置为 类中变量 gpsy，表示物体在地图坐标系中的 y 轴位置；
        5. 将消息 pose 中位姿的 x 轴方向的四元数分量设置为 0，表示物体的方向；
        6. 将消息 pose 中位姿的 y 轴方向的四元数分量设置为 0，表示物体的方向；
        7. 将消息 pose 中位姿的 z 轴方向的四元数分量设置为 0，表示物体的方向；
        8. 将消息 pose 中位姿的 w 轴方向的四元数分量设置为 1，表示物体的方向；
3. 配置路径：
    1. 标识 path_record 消息的时间戳和参考坐标系
    2. 存储 pose 消息进入 path_record 消息（path_record 用于存储路径信息，不断向其中添加新的位姿信息）
4. 如果路径记录 path_record 消息超过 1000，自动去除最前排的位姿信息；
5. 通过发布者 path_pub 发布消息 path_record 并在终端打印该信息内容；

方法5：MainLoop()

如果节点正常工作：

1. 逐行读取串口信息存入 data；
2. 类中变量 gpsx 自加 0.5；
3. 类中变量 gpsy 自加1；
4. 类中变量 pose2D_msg 的分量 x 设为强制转化为 float 类型后的 gpsx；
5. 类中变量 pose2D_msg 的分量 y 设为强制转化为 float 类型后的 gpsy；
6. 类中变量 pose2D_msg 的分量 theta 设为强制转化为 float 类型后的 (gpsx * gpsy)；
7. 使用发布者 pose2D_publisher 发布更新后的 pose2D_msg 消息；
8. 终端打印：pose2D_msg 的值
9. 调用 CheckData() 方法检查串口读取的数据是否符合预期，符合则继续循环，不符合则跳出本次循环；
10. 按设定的频率休眠

2、launch文件

1、hf.launch

1. 设置 `~control_rate` 的值为 10；
2. 启动 mower_ctrl 功能包的 hfflower.py 文件：
   参数 `~control_rate` 设置为 10；
   参数 `~linear_yk` 设置为 0.75；
   参数 `~linear_zk` 设置为 0.25；
   参数 `~linear_allk` 设置为 0.5；
   参数 `~near_ul` 设置为 1500；
   参数 `~far_ul` 设置为 2000。
3. 启动 mower_ctrl 功能包的 hfserial.py 文件：
   参数 `~control_rate` 设置为 10；

2、mower_ctrl.launch

1. 启动功能包 joy 的类型 joy_node，节点名为 joy_node，
   `~autorepeat_rate` 的值设为 10；
2. 启动功能包 mower_ctrl 的 motion_ctrl.py 文件，节点名为：motion_ctrl：
   设置参数 `~control_rate` 为 外部参数 `~control_rate` 的值；
3. 启动功能包 mower_ctrl 的 joystick_ctrl.py 文件，节点名为：joystick_ctrl：
   设置参数 `~speed_range` 为 0.5；
   设置参数 `~angle_range` 为1.0；

3、lining_ctrl.launch

1. 设置 `~control_rate` 的值为 10；
2. 启动功能包 mower_ctrl 的 motion_ctrl.py 文件，节点名为：motion_ctrl：
   设置参数 `~control_rate` 为 外部参数 `~control_rate` 的值；
3. 启动功能包 mower_ctrl 的 lining_ctrl.py 文件，节点名为：lining_ctrl：

设置参数 `~d_weight` 的值为 1；
设置参数 `~kp` 的值为 `-0.5`；
设置参数 `~kd` 的值为 `-13.3`；
设置参数 `~lining_speed` 的值为 1；
设置参数 `~control_rate` 的值为 `$(arg ~control_rate)`，即：值为 10；

4、gps_ctrl.launch

1. 设置 `~control_rate` 的值为 10；
2. 启动功能包 mower_ctrl 的 motion_ctrl.py 文件，节点名为：motion_ctrl：
   设置参数 `~control_rate` 为 外部参数 `~control_rate` 的值；
3. 启动功能包 mower_ctrl 的 BYGPS_reader.py 文件，节点名为：BYGPS_reader：
   设置参数 `~control_rate` 为 外部参数 `~control_rate` 的值；
4. 启动功能包 mower_ctrl 的 local_xy_nav.py 文件，节点名为：local_xy_nav：
   设置参数 `~kp` 为 `-0.5`；
   设置参数 `~kd` 为 `-13.3`；
   设置参数 `~lining_speed` 为 `1`；
   设置参数 `~control_rate` 为外部参数 `~control_rate` 的值；
   设置参数 `~land_long` 为 `8`；
   设置参数 `~land_wide` 为 `4`；
   设置参数 `~scan_wide` 为 `2`；