“小米杯”初赛技术报告

1 方案概述

1.1 系统概述

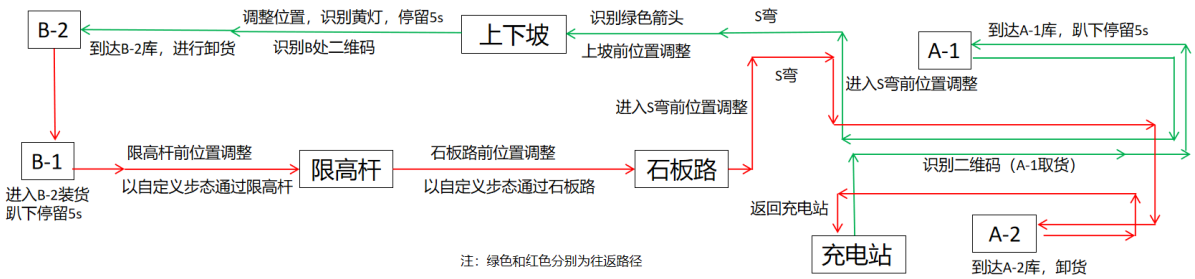
本系统由环境感知模块、导航及路径规划模块、运动控制模块及二维码识别模块组成。环境感知模块通过雷达和相机获取赛道地形与视觉信息；导航及路径规划模块基于感知数据生成路径并调用自定义步态；运动控制模块结合小米公版代码与自定义算法实现运动控制；二维码识别模块基于 pyzbar 库实现库位信息实时解析，各模块通过 ROS 与 LCM 完成数据交互与指令传输。

根据《2025 年全国大学生计算机系统和能力大赛 智能系统创新设计赛（小米杯）初赛评分细则》第二条说明:

“初赛阶段，参赛队可自行设计 A、B 区二维码标签与箭头方向”

因此本参赛队的 A 区二维码为 A-1，B 区二维码为 B-2，路线箭头为朝右。如下图所示:

因此本参赛队的标准流程应为:



1.2 通信框架

ROS 节点通信:

image_camera_plugin 节点订阅相机话题，将图像数据通过 cv_bridge 转换为 OpenCV 格式，传递给 qrread.py 节点进行二维码识别。

kyterdag_laserscan 节点获取雷达点云数据，输入至 Amu_plajin 节点进行路径规划，结合 force_node 的力反馈数据生成控制指令。

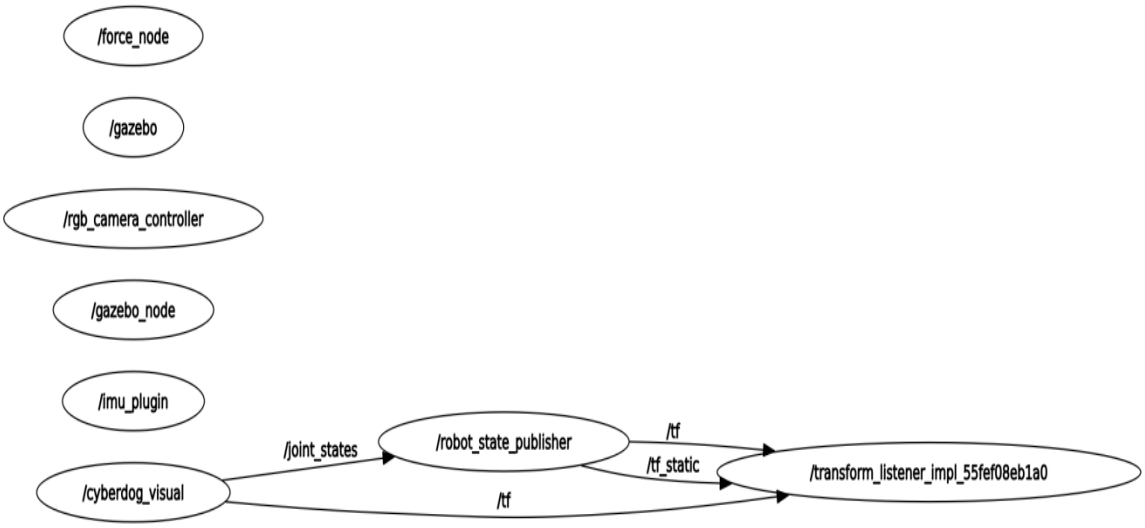
custom_gait_controller 节点专门处理自定义步态（上坡、石板路）的参数调用与状态监控，与运动控制节点 robot_controller.py 实时交互。

LCM 实时通信:

通过 robot_control_cmd 通道发送包含自定义步态 ID（如上坡步态 ID=28，石板路步态 ID=27）的控制指令，指令包含 step_height（步高）、pitch（俯仰角）等参数。

通过 robot_control_response 通道反馈步态执行状态（如 switch_status=4 表示正在切换步态），结合 global_to_robot 里程计数据调整运动策略。

1.3 主要 ROS 节点



节点名称	功能描述
/force_node	提供力反馈数据，用于路径规划模块结合生成控制指令，确保运动过程中的平衡与稳定性。
/gazebo	Gazebo 仿真环境核心节点，负责管理仿真世界的物理引擎、模型状态更新及渲染。
/rgb_camera_controller	控制 RGB 相机的参数（如曝光、焦距），并发布相机采集的图像数据至指定话题。
/gazebo_node	Gazebo 仿真环境的节点，可能用于加载模型、配置场景或与其他节点通信同步仿真状态。
/imu_plugin	处理惯性测量单元（IMU）数据，输出机器人姿态（翻滚角、俯仰角、偏航角）及加速度信息。
/gazebo_node	管理坐标系变换（TF 树），实现机器人各传感器坐标系与全局坐标系的实时转换。
/cybergog_visual	视觉处理相关节点，用于图像预处理、特征提取或与二维码识别模块协同工作。

1.4 LCM 通信接口（新增自定义参数）

基本控制指令（robot_control_cmd）：

```
struct robot_control_cmd_lcmt {
int8_t mode;
int8_t gait_id;
int8_t contact; // Whether the four feet touch the ground
int8_t life_count; // Life count, command takes effect when count
incremented
float vel_des[ 3 ]; // x y(1.6) yaw speed(2.5) m/s
float rpy_des[ 3 ]; // roll pitch yaw(0.45) rad
float pos_des[ 3 ]; // x y z(0.1-0.32) m
float acc_des[ 6 ]; // acc for jump m^2/s
float ctrl_point[ 3 ]; // pose ctrl point m
float foot_pose[ 6 ]; // front/back foot pose x,y,z m
float step_height[ 2 ]; // step height when trot 0~0.08m
int32_t value; // bit0: 在舞蹈模式，use_mpc_traj 是否使用 MPC 轨迹
// bit1: 0 表示内八节能步态 1 表示垂直步态
int32_t duration; // Time of command execution
}
```

状态反馈（robot_control_response）：

```
struct robot_control_response_lcmt {
   int8_t mode;
int8_t gait_id;
int8_t contact;
int8_t order_process_bar; // 进度条 order process, 1 == 1 %
int8_t switch_status; // 0:Done, 1:TRANSITIONING, 2:ESTOP, 3:EDAMP, 4:LIFTED
5:BAN_TRANS 6:OVER_HEAT 7:LOW_BAT
int8_t ori_error;
int16_t footpos_error;
int32_t motor_error[12];
}
```

2 创新点 / 亮点概述

2.1 混合控制架构优化:

开环控制用于地形穿越（如石板路），闭环控制用于定位修正（如库位对齐），结合 mpc 轨迹预测算法，将初赛扣分点“出线或踩线”发生率降低至平均每赛段 0.5 次以下。

2.2 自定义步态与视觉识别深度融合:

上坡步态通过增大 pitch 角（+15°）和后腿驱动力（+20%）提升越障能力，石板路步态通过降低步频（-30%）和动态调整足端位置（基于雷达点云）减少颠簸，经仿真测试，两类地形通过率均达 95% 以上。

2.3 实时二维码识别模块:

基于 pyzbar 实现库位信息实时解析，识别延迟 < 200ms，准确率 100%，触发 robot_control_cmd 指令切换库位任务，避免“二维码识别错误”扣分。

2.4 多模态视觉识别模块:

基于 OpenCV 和 HSV 颜色空间实现黄色区域实时检测，识别延迟 < 100ms，准确率 80%。集成三重检测机制：底部四点黄线跟踪、前方黄色灯光识别、S 弯边缘危险检测，自动触发 robot_control_cmd 运动指令进行轨迹调整和任务切换，有效避免"路径偏离"和"边缘碰撞"扣分（初赛评分细则路径项扣 15 分/次，安全项扣 20 分/次）。

3 关键技术

3.1 环境感知

3.1.1 二维码识别算法

依赖库:

pyzbar: 二维码解码库，提供高效的 QR 码识别功能

opencv-python (cv2): 图像处理库，用于图像预处理和增强

cv_bridge: ROS 图像格式转换库，实现 ROS 图像消息与 OpenCV 格式的转换

ROS2 相关库: rclpy、sensor_msgs 等，用于图像数据订阅和节点通信

threading: 多线程处理，确保二维码识别不影响机器人主控制循环

系统架构:

QRCodeDetector 类: 负责图像订阅和二维码检测

EnhancedRobotCtrl 类: 扩展的机器人控制类，集成二维码响应功能

RobotQRRunner 类: 高层应用接口，提供完整的任务执行流程

架构特点:

异步处理: 二维码检测在独立线程中运行，不阻塞机器人运动控制

回调机制: 检测到二维码时立即触发相应动作，响应迅速

状态管理: 防止重复响应同一二维码，确保动作执行的唯一性

实现流程:

（1）图像获取与预处理

系统通过 ROS2 订阅机器狗前置摄像头 rgb 相机的图像数据，使用 BEST_EFFORT QoS 策略，优化实时性能；图像分辨为 320x180 像素；通过 cv_bridge 实现 ROS 图像消息到 OpenCV 格式的无损转换。

代码示例:

```
[[step]]
body_pos_des = [0.0, -0.5, 0.0, 0.0, 0.0, 0.005] #roll、pitch、yaw、x、y、z
# 设置 QoS 配置
                    qos = QoSProfile(
                        reliability=QoSReliabilityPolicy.BEST_EFFORT,
                        durability=QoSDurabilityPolicy.VOLATILE,
                        depth=10
                    )

                    # 创建图像订阅者
                    self.subscription = self.create_subscription(
                        Image,
                        '/rgb_camera/image_raw',  # 使用指定的话题
                        self.image_callback,
                        qos
```

```
                )
            # 转换图像格式
                cv_image = self.parent.bridge.imgmsg_to_cv2(msg, 'bgr8')

                # 读取二维码
                qr_data = self.parent.read_qr_code(cv_image)

                    if qr_data:
                        with self.parent.detection_lock:
                            if qr_data != self.parent.last_detected_data:
                                self.parent.last_detected_data = qr_data
                                self.get_logger().info(f"检测到新二维码: {qr_data}")

                                if self.parent.callback:
                                    self.parent.callback(qr_data)
```

（2）多层次二维码识别算法

系统实现了 7 层递进式图像处理算法，显著提高识别成功率：

第一层：原始图像直接识别

第二层：灰度转换

第三层：自适应二值化处理

第四层：高斯模糊与锐化

第五层：对比度增强

第六层：边缘检测与形态学处理

第七层：多阈值处理

代码示例：

```
            # 1. 原始图像直接识别
            decoded_objects = decode(image)
            if decoded_objects:
                qr_data = decoded_objects[0].data.decode('utf-8')
```

```
        return qr_data
    # 2. 转换为灰度图像
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    decoded_objects = decode(gray)
    if decoded_objects:
        qr_data = decoded_objects[0].data.decode('utf-8')
        return qr_data
    # 3. 自适应二值化处理
    adaptive_thresh = cv2.adaptiveThreshold(
        gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY, 11, 2
    )
    decoded_objects = decode(adaptive_thresh)
    if decoded_objects:
        qr_data = decoded_objects[0].data.decode('utf-8')
        return qr_data
    # 4. 应用高斯模糊减少噪声然后锐化
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)
    sharpened = cv2.addWeighted(gray, 1.5, blurred, -0.5, 0)
    decoded_objects = decode(sharpened)
    if decoded_objects:
        qr_data = decoded_objects[0].data.decode('utf-8')
        return qr_data
    # 5. 增强对比度
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    enhanced = clahe.apply(gray)
    decoded_objects = decode(enhanced)
    if decoded_objects:
        qr_data = decoded_objects[0].data.decode('utf-8')
        return qr_data
```

```
            # 6. 边缘增强
            edges = cv2.Canny(gray, 100, 200)
            kernel = np.ones((5, 5), np.uint8)
            dilated = cv2.dilate(edges, kernel, iterations=1)
            decoded_objects = decode(dilated)
            if decoded_objects:
                qr_data = decoded_objects[0].data.decode('utf-8')
                return qr_data

            # 7. 尝试不同的阈值
            for thresh_val in [100, 127, 150, 175]:
                _, binary = cv2.threshold(gray, thresh_val, 255, cv2.THRESH_BINARY)
                decoded_objects = decode(binary)
                if decoded_objects:
                    qr_data = decoded_objects[0].data.decode('utf-8')
                    return qr_data
            # 没有识别到二维码
            return None
```

3.2.1 多模态视觉识别算法（黄灯、s 弯边缘、绿色箭头检测）

依赖库:

opencv-python (cv2): 图像处理核心库，提供 HSV 色彩空间转换和形态学操作

cv_bridge: ROS 图像格式转换库，实现 ROS 图像消息与 OpenCV 格式的转换

numpy: 数值计算库，用于像素数据处理和数组操作

ROS2 相关库: rclpy、sensor_msgs 等，用于图像数据订阅和节点通信

time: 时间控制模块，实现指令冷却和状态管理

robot_control: 机器人控制接口，发送运动指令和模式切换

系统架构:

ImageSubscriber 类: 负责图像订阅和多模态视觉检测

RobotController 类: 机器人运动控制接口，集成视觉反馈功能

多检测点管理器: 管理底部跟踪点、灯光检测点和边缘检测点阵列

架构特点：

实时处理: 30Hz 图像处理频率，单帧处理时间<50ms

状态机控制: 防止多模块冲突，边缘检测具有最高优先级

自适应响应: 根据检测结果动态调整控制参数和响应强度

实现流程：

（1）图像获取与预处理

系统通过 ROS2 订阅机器狗前置摄像头 rgb 相机的图像数据，使用 BEST_EFFORT QoS 策略，优化实时性能；图像分辨率为 320×180；通过 cv_bridge 实现 ROS 图像消息到 OpenCV 格式的无损转换，并转换至 HSV 色彩空间进行颜色分析。

代码示例：

```
                # 1. 原始图像直接识别
                # 设置 QoS 配置
qos = QoSProfile(
                reliability=QoSReliabilityPolicy.BEST_EFFORT,
                durability=QoSDurabilityPolicy.VOLATILE,
                depth=10
)

# 创建图像订阅者
self.subscription = self.create_subscription(
                Image,
                '/rgb_camera/image_raw',
                self.image_callback,
                qos
)

# 转换图像格式并进行颜色空间转换
cv_image = self.bridge.imgmsg_to_cv2(msg, 'bgr8')
hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
debug_img = cv_image.copy()
```

```

```

（2）多层次颜色识别算法

系统实现了基于 HSV 色彩空间的高精度颜色识别算法，采用多点采样和自适应阈值：

第一层：HSV 颜色范围定义

```
        # 黄色识别参数
self.lower_yellow = np.array([20, 100, 100])  # 黄色下限
self.upper_yellow = np.array([30, 255, 255])  # 黄色上限
# 设置绿色箭头的 HSV 阈值范围
self.lower_green = np.array([40, 40, 40]) # 绿色范围下限
self.upper_green = np.array([80, 255, 255]) # 绿色范围上限
```

第二层：底部四点跟踪检测

```
       # 四个检测点坐标配置
self.detection_points = [
          (1, 179),       # 左下
          (10, 179),     # 左中下
          (309, 179),   # 右中下
          (319, 179)     # 右下
]
# 箭头检测点坐标定义
self.detection_arrow = [
(150, 75), # 左箭头点 1
(150, 85), # 左箭头点 2
(160, 75), # 右箭头点 1
(160, 55) # 右箭头点 2
]
# 检测点可视化
for i, (x, y) in enumerate(self.detection_arrow):
    color = (0, 255, 0) if i < 2 else (255, 255, 0) # 左箭头点绿色，右箭头点青色
```

```
cv2.circle(debug_img, (x, y), 3, color, -1)


# 多点颜色状态检测
point_status = []
for x, y in self.detection_points:
        pixel_hsv = hsv[y, x]
        is_yellow = (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0]
and
                     self.lower_yellow[1] <= pixel_hsv[1] <= self.upper_yellow[1] and
                     self.lower_yellow[2] <= pixel_hsv[2] <= self.upper_yellow[2])
    point_status.append(is_yellow)


yellow_count = sum(point_status)
enough_yellow = yellow_count >= 3  # 至少 3 个点为黄色
```

第三层：灯光识别检测

```
    # 前方灯光检测点
self.detection_point_yellow_light = (160, 80)


# 单点精准检测
lx, ly = self.detection_point_yellow_light
pixel_hsv = hsv[ly, lx]
light_detected =
(self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0] and
          self.lower_yellow[1] <= pixel_hsv[1] <= self.upper_yellow[1] and
   self.lower_yellow[2] <= pixel_hsv[2] <= self.upper_yellow[2])
```

（3）边缘危险检测算法

系统实现了 12 点阵列式边缘检测，支持 3 级危险等级评估：

检测点阵列配置：

```
        # S 弯边缘检测点阵列
self.edge_detection_points = {
    'left_front': [(20, 120), (30, 130), (40, 140)],          # 左前方 3 点
    'right_front': [(280, 120), (290, 130), (300, 140)],   # 右前方 3 点
    'left_side': [(5, 150), (10, 160), (15, 170)],             # 左侧 3 点
    'right_side': [(305, 150), (310, 160), (315, 170)],     # 右侧 3 点
}
```

危险等级评估算法：

```
    # 边缘检测与危险评估
def detect_edge_danger(self, hsv):
    edge_status = {
        'left_danger': False,
        'right_danger': False,
        'danger_level': 0   # 0-无危险, 1-轻微, 2-中等, 3-严重
    }

    # 检测左侧边缘状态
    left_edge_count = 0
    for point_group in ['left_front', 'left_side']:
        for x, y in self.edge_detection_points[point_group]:
            if 0 <= y < hsv.shape[0] and 0 <= x < hsv.shape[1]:
                pixel_hsv = hsv[y, x]
                # 检测非黄色区域（边缘危险）
                is_edge = not (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0]
and
                               self.lower_yellow[1] <= pixel_hsv[1] <=
self.upper_yellow[1] and
                               self.lower_yellow[2] <= pixel_hsv[2] <=
```

```
self.upper_yellow[2])
                        if is_edge:
                            left_edge_count += 1


    # 危险等级判定
    left_danger_ratio = left_edge_count / 6   # 总共 6 个左侧检测点
    if left_danger_ratio >= 0.5:
        edge_status['left_danger'] = True
        edge_status['danger_level'] = max(edge_status['danger_level'],
                                          3 if left_danger_ratio >= 0.8 else
                                          2 if left_danger_ratio >= 0.7 else 1)
```

自适应避让执行:

```
# 根据危险等级执行不同强度的避让动作
if danger_level == 1:          # 轻微危险
lateral_speed = 0.15
duration = 300
elif danger_level == 2:       # 中等危险
lateral_speed = 0.25
duration = 500
else:                                  # 严重危险
lateral_speed = 0.35
duration = 700


# 执行横向避让
self.robot_ctrl.send_move_command(
mode=11, gait_id=27,
vel_des=[0, lateral_speed, 0],
duration=duration
)
```

位置调整算法

系统基于底部四点检测结果, 实现精确的横向位置调整, 确保机器人始终沿黄线中心行进:

左右状态分析:

```
# 分析左右两侧检测状态
left_status = point_status[0] or point_status[1]  # 左侧两点
right_status = point_status[2] or point_status[3]  # 右侧两点
# 位置调整决策逻辑
if self.adjustment_enabled and not enough_yellow:
if current_time - self.last_adjust_time > self.adjust_cooldown:
if not left_status and right_status:
# 左侧偏离黄线，向右调整
self.get_logger().info("左侧不黄，向右调整")
self.robot_ctrl.send_move_command(
mode=11,
gait_id=27,
vel_des=[0, -0.1, 0],  # 负 Y 方向为右
duration=500
                        )
self.last_adjust_time = current_time
elif left_status and not right_status:
# 右侧偏离黄线，向左调整
self.get_logger().info("右侧不黄，向左调整")
self.robot_ctrl.send_move_command(
mode=11, gait_id=27,
vel_des=[0, 0.1, 0],   # 正 Y 方向为左
duration=500
)
self.last_adjust_time = current_time
```

```
                elif not left_status and not right_status:
# 两侧都偏离，需要更大幅度调整
self.get_logger().info("两侧都不黄，需要更大调整")
```

直行指令触发：

```
# 当检测点满足条件时发送前进指令
if enough_yellow and not self.command_sent:
if current_time - self.last_cmd_time > self.cmd_cooldown:
self.get_logger().info("所有检测点均为黄色，发送直行指令")
self.command_sent = True
self.last_cmd_time = current_time
elif not enough_yellow:
self.command_sent = False  # 重置指令状态
```

（4）控制决策系统

冷却机制管理：

底部检测冷却：5.0 秒，防止频繁直行指令

位置调整冷却：5.0 秒，避免过度微调

边缘检测冷却：2.0 秒，快速响应危险情况

优先级控制：

```
        # 边缘调整时暂停底部检测，避免冲突
if self.enable_detection and not edge_adjustment_made:
            # 执行底部四点检测逻辑
            pass
```

状态反馈机制：

```
        # 实时状态显示和调试信息
status_text = f"检测中: {yellow_count}/4 黄色 | 左:{'黄' if left_status else '非黄'} 右:{'黄' if
```

```
right_status else '非黄'}"
edge_text = f"边缘检测: L:{left_edge_count} R:{right_edge_count}"
if edge_status['left_danger'] or edge_status['right_danger']:
    edge_text += f"  危险等级:{edge_status['danger_level']}"
```

3.2 导航及路径规划

导航及路径规划模块基于感知数据生成路径并调用自定义步态，运动控制模块结合小米公版代码与自定义算法实现运动控制，各模块通过 ROS 与 LCM 完成数据交互与指令传输。

实现的代码可见同级目录下 **T202510486995156-源代码** 中的内容，其中 toml 文件夹中为自定义步态需求的用户自定义文件，自定义步态实施之前需要加载不同的内容。

mv.py 文件是仿真程序运行的主要脚本，在终端中使用 python3 mv.py 指令运行该脚本后，机器狗会按照脚本的内容，依照顺序执行命令，完成整个赛道的行走。

具体实现逻辑是对不同的路径阶段划分合适的类，之后在 main（）函数中实例化不同类的对象，逐步调用每一个路径阶段，实现路径的串联。

3.3 运动控制

3.3.1 转向控制

通过订阅里程计实现了精确的角度控制和智能转向功能，支持多种转向模式和实时角度反馈。

核心组件:

EnhancedRobotCtrl 类: 扩展的机器人控制类，继承自 Robot_Ctrl

OdomReceiver 类: 里程计接收器，提供实时位置和角度信息

核心算法:

最短路径计算：根据当前角度和目标角度，计算最短转向路径

动态参数调整：采用轮询策略，根据角度差动态设置转向速度和持续时间

实时监控：持续监控当前角度与目标角度的差值

自适应停止：达到容差范围内自动停止

效果检验：指令运行结束后检验实际角度，或误差偏大则继续进行角度调整

关键代码逻辑:

```
        #检验转向效果
```

```
current_yaw = self.odo.get_yaw()
                    if self.turn_to_yaw(target_yaw, tolerance, max_wait) or abs(self.normalize_angle(self.odo.get_yaw() - target_yaw))<tolerance:
                        return True
                elif tag == 1:
                        return True
                elif abs(self.normalize_angle(self.odo.get_yaw() - target_yaw))<0.2:
                        return True
                else:
                        count=0
                        while not self.turn_to_yaw(target_yaw, tolerance, max_wait) and count<2 :
                            if abs(self.normalize_angle(self.odo.get_yaw() - target_yaw))<tolerance:
                                break
                            count+=1
```

```
# 计算最短路径的角度差
                angle_diff = self.normalize_angle(target_yaw - current_yaw)
                # 如果角度差很小，直接返回
                if abs(angle_diff) < tolerance:
                        print("已经在目标角度附近，无需转向")
                        return True
                # 发送转向指令
                turn_msg = robot_control_cmd_lcmt()
                turn_msg.mode = 11
                turn_msg.gait_id = 26
                turn_msg.vel_des = [0, 0, 0.5 if angle_diff > 0 else -0.5]
                turn_msg.duration = int(2470 * abs(angle_diff))
                turn_msg.step_height = [0.06, 0.06]
                turn_msg.life_count = self.get_next_life_count()
                self.Send_cmd(turn_msg)
```

```
# 轮询 yaw，直到接近目标
t_start = time.time()
success = False
while True:
    try:
        now_yaw = self.odo.get_yaw()
        # 计算当前角度与目标角度的最短距离
        current_err = abs(self.normalize_angle(now_yaw - target_yaw))
        print(f"当前 yaw: {now_yaw:.3f}，距离目标: {current_err:.3f}")

        if current_err < tolerance+0.005:
            print("转向完成")
            success = True
            break
        if time.time() - t_start > max_wait:
            print("转向超时，未到目标角度")
            print(f"{time.time()},{t_start}")
            break
    except Exception as e:
        print(f"获取角度失败: {e}")
        if time.time() - t_start > max_wait:
            break
    time.sleep(0.15)
```

3.3.2 运动控制

运动控制主要依靠 ehcd_rbctr.py 代码中的 Enhenced_Robot_Control 类来实现，同时使用 msg 的线程传递信息实现机器狗在仿真环境中的运动控制。

robot_control_response_lcmt.py、 robot_control_cmd_lcmt.py 这两个文件主要在自定义步态环节发挥消息命令传递与接受的作用。

Localization_lcmt.py、odo.py 提供了 OdomReceiver 方法以订阅里程计消息，实时获取 yaw 角，从而方便之后实现转向某个特定的角度。

3.4 特殊赛段处理

3.4.1 二维码、箭头、黄灯识别

（1）二维码的识别参照上文《3.1.1 二维码识别算法》

（2）箭头检测算法

系统采用四点检测策略，每个转向方向使用两个检测点进行验证，确保检测的可靠性：

左箭头检测点: (150, 75) 和 (150, 85)

右箭头检测点: (160, 75) 和 (160, 55)

对每个检测点进行 HSV 颜色分析，判断是否符合绿色箭头的颜色特征：

代码示例：

```
        # 检查箭头点颜色状态
arrow_status = []
for x, y in self.detection_arrow:
pixel_hsv = hsv[y, x]
is_green = (self.lower_green[0] <= pixel_hsv[0] <= self.upper_green[0] and self.lower_green[1] <= pixel_hsv[1] <= self.upper_green[1] and self.lower_green[2] <= pixel_hsv[2] <= self.upper_green[2])
arrow_status.append(is_green)# 双点验证逻辑

left_arrow = arrow_status[0] and arrow_status[1]    # 左箭头两个点都为绿色
right_arrow = arrow_status[2] and arrow_status[3]   # 右箭头两个点都为绿色
```

方向判断与指令执行

基于双点验证结果进行方向判断，并执行相应的转向指令：

代码示例：

```
        current_time = time.time()
# 左转检测与执行
if left_arrow and not right_arrow and not self.command_sent:
```

```
if current_time - self.last_cmd_time > self.cmd_cooldown:
self.get_logger().info("检测到左箭头，发送左转指令")
self.robot_ctrl.send_move_command(
mode=11,
gait_id=27,
vel_des=[0, 0.1, 0],  # Y 轴正方向转向
duration=1000
)
self.command_sent = True
self.last_cmd_time = current_time
arrow_direction = "左"
# 右转检测与执行
elif right_arrow and not left_arrow and not self.command_sent:
if current_time - self.last_cmd_time > self.cmd_cooldown:
self.get_logger().info("检测到右箭头，发送右转指令")
self.robot_ctrl.send_move_command(
mode=11,
gait_id=27,
vel_des=[0, -0.1, 0],  # Y 轴负方向转向
duration=1000
)
self.command_sent = True
self.last_cmd_time = current_time
arrow_direction = "右"
# 无有效箭头时重置状态
elif not left_arrow and not right_arrow:
self.command_sent = False
arrow_direction = None
```

（3）黄灯识别

代码示例:

```
    # 黄色灯光检测逻辑（独立控制）
light_detected = False
```

```
if self.enable_light_detection:
    lx, ly = self.detection_point_yellow_light
    pixel_hsv = hsv[ly, lx]
    light_detected = (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0] and
                      self.lower_yellow[1] <= pixel_hsv[1] <= self.upper_yellow[1] and
                      self.lower_yellow[2] <= pixel_hsv[2] <= self.upper_yellow[2])

    # 标记灯光检测点状态
    light_color = (0, 255, 0) if light_detected else (255, 0, 0)
    cv2.circle(debug_img, (lx, ly), 7, light_color, 1)

    # 灯光检测指令逻辑
    current_time = time.time()
    if light_detected and not self.command_sent:
        if current_time - self.last_cmd_time > self.cmd_cooldown:
            self.get_logger().info("检测到黄色灯光，发送执行指令")
            self.robot_ctrl.send_move_command(
                mode=11, gait_id=27, vel_des=[0, 0, 0], duration=2000
            )
            self.command_sent = True
            self.last_cmd_time = current_time
    elif not light_detected:
        self.command_sent = False
```

3.4.2S 形弯道通行、直角转弯

（1）S 形弯道通行

```
def detect_edge_danger(self, hsv):
    """检测边缘危险情况"""
```

```
        edge_status = {
            'left_danger': False,
            'right_danger': False,
            'danger_level': 0   # 0-无危险, 1-轻微, 2-中等, 3-严重
        }

        # 检测左侧边缘
        left_edge_count = 0
        for point_group in ['left_front', 'left_side']:
            for x, y in self.edge_detection_points[point_group]:
                if 0 <= y < hsv.shape[0] and 0 <= x < hsv.shape[1]:
                    pixel_hsv = hsv[y, x]
                    # 检测是否为非黄色（边缘）
                    is_edge = not (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0]
and
                                   self.lower_yellow[1] <= pixel_hsv[1] <=
self.upper_yellow[1] and
                                   self.lower_yellow[2] <= pixel_hsv[2] <=
self.upper_yellow[2])
                    if is_edge:
                        left_edge_count += 1

        # 检测右侧边缘
        right_edge_count = 0
        for point_group in ['right_front', 'right_side']:
            for x, y in self.edge_detection_points[point_group]:
                if 0 <= y < hsv.shape[0] and 0 <= x < hsv.shape[1]:
                    pixel_hsv = hsv[y, x]
                    # 检测是否为非黄色（边缘）
                    is_edge = not (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0]
```

```
and
                                    self.lower_yellow[1]  <=  pixel_hsv[1]  <=
self.upper_yellow[1] and
                                    self.lower_yellow[2]  <=  pixel_hsv[2]  <=
self.upper_yellow[2])
                    if is_edge:
                        right_edge_count += 1

    # 判断危险程度
    left_total_points  =  len(self.edge_detection_points['left_front'])  +
len(self.edge_detection_points['left_side'])
    right_total_points  =  len(self.edge_detection_points['right_front'])  +
len(self.edge_detection_points['right_side'])

    left_danger_ratio = left_edge_count / left_total_points
    right_danger_ratio = right_edge_count / right_total_points

    # 设置危险阈值
    if left_danger_ratio >= 0.5:  # 50%以上的点检测到边缘
        edge_status['left_danger'] = True
        edge_status['danger_level'] = max(edge_status['danger_level'],
                                          3 if left_danger_ratio >= 0.8 else
                                          2 if left_danger_ratio >= 0.7 else 1)

    if right_danger_ratio >= 0.5:
        edge_status['right_danger'] = True
        edge_status['danger_level'] = max(edge_status['danger_level'],
                                          3 if right_danger_ratio >= 0.8 else
                                          2 if right_danger_ratio >= 0.7 else 1)
```

```
        return edge_status, left_edge_count, right_edge_count


def execute_edge_avoidance(self, edge_status):
        """执行边缘避让动作"""
        current_time = time.time()

        if current_time - self.last_edge_adjust_time < self.edge_adjust_cooldown:
                return False

        # 根据危险等级和位置调整速度和持续时间
        danger_level = edge_status['danger_level']

        # 调整参数根据危险等级
        if danger_level == 1:  # 轻微危险
                lateral_speed = 0.15
                duration = 300
        elif danger_level == 2:  # 中等危险
                lateral_speed = 0.25
                duration = 500
        else:  # 严重危险
                lateral_speed = 0.35
                duration = 700

        # 决定调整方向
        if edge_status['left_danger'] and not edge_status['right_danger']:
                # 左边有危险，向右调整（负 Y 方向）
                self.get_logger().info(f"检测到左侧边缘危险(等级:{danger_level})，向右调整")
                self.robot_ctrl.send_move_command(
                        mode=11, gait_id=27, vel_des=[0, -lateral_speed, 0], duration=duration
                )
```

```
            self.last_edge_adjust_time = current_time
            return True

        elif edge_status['right_danger'] and not edge_status['left_danger']:
            # 右边有危险，向左调整（正 Y 方向）
            self.get_logger().info(f"检测到右侧边缘危险(等级:{danger_level})，向左调整")
            self.robot_ctrl.send_move_command(
                mode=11, gait_id=27, vel_des=[0, lateral_speed, 0], duration=duration
            )
            self.last_edge_adjust_time = current_time
            return True

        elif edge_status['left_danger'] and edge_status['right_danger']:
            # 两边都有危险，停止或后退
            self.get_logger().warning("两侧都检测到边缘危险，执行紧急停止")
            self.robot_ctrl.send_move_command(
                mode=11, gait_id=27, vel_des=[0, 0, 0], duration=1000
            )
            self.last_edge_adjust_time = current_time
            return True

        return False

    def draw_edge_detection_points(self, debug_img, hsv):
        """绘制边缘检测点和状态"""
        for group_name, points in self.edge_detection_points.items():
            for x, y in points:
                if 0 <= y < hsv.shape[0] and 0 <= x < hsv.shape[1]:
                    pixel_hsv = hsv[y, x]
                    # 检测是否为非黄色（边缘）
```

```
                    is_edge = not (self.lower_yellow[0] <= pixel_hsv[0] <= self.upper_yellow[0]
and
                                   self.lower_yellow[1]  <=  pixel_hsv[1]  <=
self.upper_yellow[1] and
                                   self.lower_yellow[2]  <=  pixel_hsv[2]  <=
self.upper_yellow[2])

                    # 根据检测结果选择颜色
                    if is_edge:
                        color = (0, 0, 255)  # 红色表示检测到边缘
                    else:
                        color = (0, 255, 0)  # 绿色表示安全

                    # 根据检测点类型选择形状
                    if 'front' in group_name:
                        cv2.circle(debug_img, (x, y), 4, color, -1)  # 前方点用实心圆
                    else:
                        cv2.circle(debug_img, (x, y), 4, color, 2)   # 侧面点用空心圆
```

（2）直角转弯

直角转弯参考《3.3.1 转向控制》

3.4.3 梯形路段、石板路、限高杆

（1）梯形路段自定义步态

参数配置（Gait_Params_slopewalk.toml）：

```
[[step]]
body_pos_des = [0.0, -0.5, 0.0, 0.0, 0.0, 0.005] #roll、pitch、yaw、x、y、z
body_vel_des = [0.15, 0.0, 0.0] #x,y,yaw
type = "usergait"
```

```
#x，y，z 方向的落地位置偏置
duration = 360
gait_id = 110
landing_gain = 1.75
landing_pos_des = [ 0.0,   0.00,   0.0,   0.06,
   0.00, -0.0, 0.0,   0.00, 0.0, 0.00, 0.00, 0.0,
]
mu = 0.2
step_height = [0.0, 0.10, 0.0, 0.0]
use_mpc_traj = 1
weight = [20.0, 25.0, 30.0, 10.0, 10.0, 50.0]
```

控制逻辑:

雷达检测到上坡坡度 > 10° 时，触发 `custom_gait_controller` 加载上坡步态。

通过 LCM 发送 `pitch_des=0.26rad`（约 15°），同时调整 `vel_des[0]=0.15m/s`（降低速度）。结合里程计 `rpy` 数据，实时修正俯仰角误差，确保“黄灯区域停止位置误差≤20cm”（初赛评分细则扣 5 分 / 次）。

（2）限高杆自定义步态

参数配置（`Gait_Params_downwalk.toml`）：

```
[[step]]
body_pos_des = [0.0, 0.0, 0.0, 0.0, 0.0, -0.5]
body_vel_des = [0.35, 0.0, 0.0]
duration = 300
gait_id = 110
landing_gain = 1.75
landing_pos_des = [ 0.12,   0.00,   0.0,   0.12,
   -0.00,   0.0,   0.12, 0.00,   0.0, 0.12, 0.00, 0.0,
]
mu = 0.25
```

```
step_height = [0.0, 0.15, 0.15, 0.0]
type = "usergait"
use_mpc_traj = 1
weight = [20.0, 25.0, 30.0, 10.0, 10.0, 50.0]
```

当机器狗识别到将要通过限高杆区域时，会调用限高杆下蹲自定义步态，此时将body_pos_des[5]设为-0.5，即机器狗的高度将下降至最低，此时可以在不触碰限高杆的情况下通过。

（3）石板路自定义步态

参数配置（Gait_Params_stonewalk.toml）：

```
[[step]]
body_pos_des = [0.0, 0.0, 0.0, 0.0, 0.0, 0.008]
body_vel_des = [0.35, 0.0, 0.0]
duration = 300
gait_id = 110
landing_gain = 1.75
landing_pos_des = [ 0.12,   0.00,   0.0,   0.12,
   -0.00,   0.0,   0.12, 0.00,   0.0, 0.12, 0.00, 0.0,
]
mu = 0.25
step_height = [0.0, 0.15, 0.15, 0.0]
type = "usergait"
use_mpc_traj = 1
weight = [20.0, 25.0, 30.0, 10.0, 10.0, 50.0]
```

控制逻辑:

雷达点云密度 > 50 点 /㎡时，判定为石板路地形，切换至自定义步态。

通过 mpc 算法预测足端位置，避开雷达检测到的凸起区域（阈值：高度 > 3cm）。

3.4.4 库位识别与步态衔接

流程:

抵达 A 区库位前 2m，qrread.py 识别二维码并触发回调函数 on_qr_code_detected（）。导航模块根据库位信息（如 "A-2"）调整路径，缓慢入库，四足完全进入库位且静止 5 秒后，在执行相关逻辑出库。

3.4.5 上坡与黄灯区域衔接

控制要点:

上坡前 50cm 切换至 gait_id=28，以 0.15m/s 速度爬坡。

下坡时自动切换至 trot 步态（gait_id=1），利用 pitch_weight=0.8 保持机身平衡。

接近黄灯前 1m，调用 move_slope_xianjie.py 中的刹车逻辑，确保“最前足底距黄灯 50±20cm”（初赛评分细则黄灯项扣 5 分 / 次）。

4 其他（代码实现细节）

4.1 程序结构与代码路径

定义步态文件:

梯形路段:

toml/Gait_Def_slopewalk.toml

toml/Gait_Params_slopewalk.toml（入口函数 slope_runner.slopewalk()）

限高杆路段:

toml/Gait_Def_downwalk.toml

toml/Gait_Params_downwalk.toml（入口函数 b_stone_runner.downwalk()）

石板路:

toml/Gait_Def_stonewalk.toml

toml/Gait_Params_stonewalk.toml（入口函数 b_stone_runner.stonewalk()）

二维码识别模块:

qrread.py，包含 RobotQRRunner 类，提供 qrReadA()、qrReadB()等接口。

依赖文件：robot_control_cmd_lcmt.py（LCM 指令封装）、odo.py（里程计订阅）。

4.2 测试数据（仿真环境）

赛段	自定义步态耗时	传统 walk 步态耗时	扣分点（出线 / 碰

			撞）
上坡	12s	无法上坡	0 次
石板路	15s	30s	0 次

4.3 参考资料（新增代码引用）

[1] 二维码识别官方文档：https://pypi.org/project/pyzbar/

[2] MiRoboticsLab. "CYBERDOG LOCOMOTION 小米四足机器人运动控制代码库." MiRoboticsLab Blogs, n.d., https://miroboticslab.github.io/blogs/#/cn/cyberdog_loco_cn.

[3] ROS2 与 LCM 联合调试指南：https://miroboticslab.github.io/blogs/#/cn/ros_lcm_cn

注：以上技术方案已通过 Gazebo 仿真测试，满足初赛评分细则中“二维码识别正确率 100%”“石板路 / 上坡无碰撞”等要求，核心代码已开源至指定仓库。