1.项目概述

1.1 项目目标

- 本项目旨在探索通过ROS机器人操作系统和Gazebo仿真平台，建立一套实车和仿真一体化的体系，开发并优化在全国大学生机器人大赛(ROBOCON)中的调试效率

1.2 项目意义

由于机器人学院几年的单片机控制开发遇到了各种各样的瓶颈，在ROBOCON 2023的已有的基础下引入ROS机器人操作系统作为控制组开发的主要方向，以提升机器人的开发效率和避免重复造轮子的现象。

使用ROS的优点如下：

1. 调试界面简单直观
2. 能够在机械没出车的情况下完成代码的初调
3. 可以根据不同的硬件开发不同的硬件接口，而且硬件驱动只要写一次，便可以通过简单的配置文件使用对应的硬件
4. 代码的复用性极强，控制组的同学可以把更多的精力放在开发新技术上，而不是像单片机一样重新实现旧的功能。
5. 在全自动机器人的开发下拥有更多好用且可视化的工具

2.技术方案

2.1 系统架构

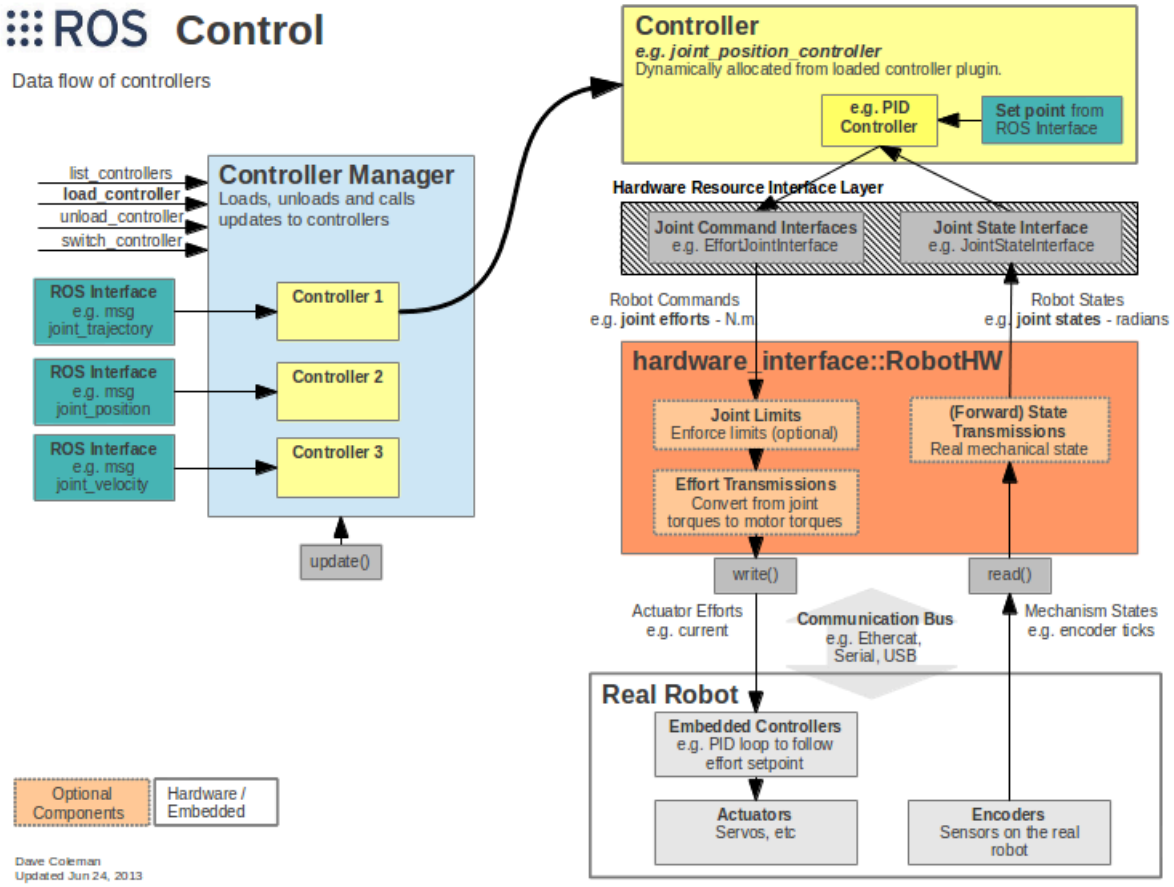
由于在ROBOCON实际的比赛中，并没有办法完全做到毫无单片机的干预，所以我们采用了ros-control的仿真环境+stm32单片机控制下层机构+ros路径规划器的开发模式。

1. 仿真框架：我们使用的仿真框架是基于ros_control进行开发控制器，然后在gazebo中进行运动控制规划仿真以及状态机仿真等
2. 硬件框架：使用stm32对机器人机构进行基础控制，例如底盘控制，夹爪控制、抬升机构和发射机构等等，通过通讯协议(etherCat)连接到ros中的控制器中进行管理。也就是说这部分相当于硬件抽象部分。

3. 导航框架：由于比赛功能并没有用上避障功能，这部分只使用了ros的全局规划器接口和跟踪器接口进行二次开发
4. 机器人状态机：使用了行为树(Behavior Tree)进行决策，接受视觉相机的信息订阅来处理机器人下一步的状态

2.2 ros-control中控制器的实现

以底盘控制为例：ros_control的数据流图如下



ros_control包将来自机器人执行器编码器的关节状态数据和输入设定点作为输入。它使用通用的控制回路反馈机制，通常是PID控制器，来控制发送到执行器的输出，通常是作用力。对于没有联合位置、努力等一对一映射的物理机制，ros_control变得更加复杂，但这些场景是使用传输来解释的。

2.2.1 官方常用的Ros_controller：

- **joint_state_controller**
    - 比较特殊，不是用来发指令的，而是用来读关节位置，并且发布在"/joint_states"话题上。"joint_state_controller/JointStateController"。
- **effort_controllers** 驱动器接受力矩指令

        - JointPositionController
            - "effort_controllers/JointPositionController". controller接受位置指令，Error=（期望位置-当前位置）。根据误差输出力指令（PID闭环)
        - JointVelocityController
            - "effort_controllers/JointVelocityController".controller接受速度指令，Error=（期望速度-当前速度）。根据误差输出力指令（PID闭环）。
        - JointEffortController
            - "effort_controllers/JointEffortController". 接受力矩指令，输出力指令（PID闭环在这里无效）。
    - **velocity_controllers** 驱动器接受速度指令
        - JointPositionController
            - "velocity_controllers/JointPositionController". controller接受位置指令，Error=（期望位置-当前位置）。根据误差输出速度指令（PID闭环）。
        - JointVelocityController
            - "velocity_controllers/JointVelocityController".接受速度指令，输出速度指令（PID闭环在这里无效）。
    - **position_controllers** 驱动器接受位置指令
        - JointPositionController
            - "position_controllers/JointPositionController". 接受位置指令，输出位置指令（PID闭环在这里无效）。

ros_control包提供了很多控制器（具体可以看ROS_WIKI）。你可以根据自己的驱动器接受指令的类型来选择。也可以根据需求重写自己的控制器。

2.2.2 自定义硬件抽象（HardwareInterface）

HardwareInterface就是ROS_Control与实体机器人之间沟通的桥梁，由上面的数据流图可以看出。而根据硬件抽象的具体功能又可以分为两类：

1. 向实体机器人发送指令--**Interfaces for Joint Actuators**
2. 从机器人的传感器读取机器人状态--**Interfaces for Joint Sensors**

1. Interfaces for Joint Actuators：

1. **EffortJointInterface**：用于接受力矩指令的驱动器。对应使用**effort_controllers**
2. **VelocityJointInterface**：用于接受速度指令的驱动器。对应使用**velocity_controllers**
3. *PositionJointInterface：用于接受位置指令的驱动器。对应使用position_controllers**

2. Interfaces for Joint Sensors:

1. **JointStateInterface**：当关节有 位置 / 速度 / 力 传感器时，用于读取传感器数据，对应使用 **joint_state_controller**
2. **ImuSensorInterface**：当有IMU传感器时使用，对应的controller为**imu_sensor_controller**

3. 成果展示：全向轮底盘控制器

下面我们假设我们需要控制四全向轮底盘。此时机器人拥有四个关节，每个关节都有位置传感器。机械臂支持力控和位置控制，即effort commands可以实现功能。我们只需要写一个头文件和cpp文件就可以实现：

我们已经有一个底盘基础信息的类，其中包含了一个机器人底盘所需要的所有信息和接口，我们只需要继承它。最后必须在控制器cpp中注册插件

chassis_base.h

```
#pragma once
#include <controller_interface/multi_interface_controller.h>
#include <hardware_interface/joint_command_interface.h>
#include <effort_controllers/joint_velocity_controller.h>
#include <realtime_tools/realtime_publisher.h>
#include <rc_common/filters/filters.h>
#include <rc_msgs/ChassisCmd.h>
#include <geometry_msgs/TwistStamped.h>

namespace chassis_controllers
{
struct Command
{
  geometry_msgs::Twist cmd_vel_;
  rc_msgs::ChassisCmd cmd_chassis_;
  ros::Time stamp_;
```

```
};

class ChassisBase : public
controller_interface::MultiInterfaceController<hardware_interface::
EffortJointInterface>
{
public:
  ChassisBase() = default;
  bool init(hardware_interface::RobotHW* robot_hw, ros::NodeHandle&
root_nh, ros::NodeHandle& controller_nh) override;
  void update(const ros::Time& time, const ros::Duration& period)
override;

protected:
  virtual void moveJoint(const ros::Time& time, const
ros::Duration& period) = 0;
  void recovery();
  void tfVelToBase(const std::string& from);
  void cmdChassisCallback(const rc_msgs::ChassisCmdConstPtr& msg);
  void cmdVelCallback(const geometry_msgs::Twist::ConstPtr& msg);
  hardware_interface::EffortJointInterface*
effort_joint_interface_{};
  std::vector<hardware_interface::JointHandle> joint_handles_{};

  double wheel_base_{}, wheel_track_{}, wheel_radius_{},
publish_rate_{}, twist_angular_{}, timeout_{},
effort_coeff_{},velocity_coeff_{}, power_offset_{};

  RampFilter<double>*ramp_x_{}, *ramp_y_{}, *ramp_w_{};
  std::string command_source_frame_{};
  geometry_msgs::Vector3 vel_cmd_{};  // x, y
  control_toolbox::Pid pid_follow_;
  ros::Subscriber cmd_chassis_sub_;
  ros::Subscriber cmd_vel_sub_;
  Command cmd_struct_;
  realtime_tools::RealtimeBuffer<Command> cmd_rt_buffer_;
};
}  // namespace chassis_controllers
```

chassis_base.cpp

```
```

```
#include "chassis_controllers/chassis_base.h"
#include <rc_common/ros_utilities.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.h>
#include <tf2_ros/transform_listener.h>

namespace chassis_controllers
{
bool ChassisBase::init(hardware_interface::RobotHW* robot_hw,
ros::NodeHandle& root_nh, ros::NodeHandle& controller_nh)
{
  if (!controller_nh.getParam("publish_rate", publish_rate_) ||
!controller_nh.getParam("timeout", timeout_))
  {
    ROS_ERROR("Some chassis params doesn't given (namespace: %s)",
controller_nh.getNamespace().c_str());
    return false;
  }

  wheel_radius_ = getParam(controller_nh, "wheel_radius", 0.02);
  wheel_track_ = getParam(controller_nh, "wheel_track", 0.410);
  wheel_base_ = getParam(controller_nh, "wheel_base", 0.320);

  effort_joint_interface_ = robot_hw-
>get<hardware_interface::EffortJointInterface>();

  ramp_x_ = new RampFilter<double>(0, 0.001);
  ramp_y_ = new RampFilter<double>(0, 0.001);
  ramp_w_ = new RampFilter<double>(0, 0.001);

  cmd_chassis_sub_ = controller_nh.subscribe<rc_msgs::ChassisCmd>
("command", 1, &ChassisBase::cmdChassisCallback, this);
  cmd_vel_sub_ = root_nh.subscribe<geometry_msgs::Twist>("cmd_vel",
1, &ChassisBase::cmdVelCallback, this);

  return true;
}

void ChassisBase::update(const ros::Time& time, const
ros::Duration& period)
{
  rc_msgs::ChassisCmd cmd_chassis = cmd_rt_buffer_.readFromRT()-
>cmd_chassis_;
```

```
  geometry_msgs::Twist cmd_vel = cmd_rt_buffer_.readFromRT()->cmd_vel_;

  if ((time - cmd_rt_buffer_.readFromRT()->stamp_).toSec() > timeout_)
  {
    vel_cmd_.x = 0.;
    vel_cmd_.y = 0.;
    vel_cmd_.z = 0.;
  }
  else
  {
    ramp_x_->setAcc(cmd_chassis.accel.linear.x);
    ramp_y_->setAcc(cmd_chassis.accel.linear.y);
    ramp_x_->input(cmd_vel.linear.x);
    ramp_y_->input(cmd_vel.linear.y);
    vel_cmd_.x = ramp_x_->output();
    vel_cmd_.y = ramp_y_->output();
    vel_cmd_.z = cmd_vel.angular.z;
  }

  if (cmd_rt_buffer_.readFromRT()->cmd_chassis_.command_source_frame.empty())
    command_source_frame_ = "base_link";
  else
    command_source_frame_ = cmd_rt_buffer_.readFromRT()->cmd_chassis_.command_source_frame;
  ramp_w_->setAcc(cmd_chassis.accel.angular.z);
  ramp_w_->input(vel_cmd_.z);
  vel_cmd_.z = ramp_w_->output();
  moveJoint(time, period);
}

void ChassisBase::recovery()
{
  ramp_x_->clear(vel_cmd_.x);
  ramp_y_->clear(vel_cmd_.y);
  ramp_w_->clear(vel_cmd_.z);
}

void ChassisBase::tfVelToBase(const std::string& from)
{
```

```
  try
  {
    tf2::doTransform(vel_cmd_, vel_cmd_,

tf2_ros::Buffer(ros::Duration(5)).lookupTransform("base_link",
from, ros::Time(0)));
  }
  catch (tf2::TransformException& ex)
  {
    ROS_WARN("%s", ex.what());
  }
}

void ChassisBase::cmdChassisCallback(const
rc_msgs::ChassisCmdConstPtr& msg)
{
  cmd_struct_.cmd_chassis_ = *msg;
  cmd_rt_buffer_.writeFromNonRT(cmd_struct_);
}

void ChassisBase::cmdVelCallback(const
geometry_msgs::Twist::ConstPtr& msg)
{
  cmd_struct_.cmd_vel_ = *msg;
  cmd_struct_.stamp_ = ros::Time::now();
  cmd_rt_buffer_.writeFromNonRT(cmd_struct_);
}
}
```

omni.h

```
#pragma once

#include <Eigen/Dense>

#include "chassis_base.h"

namespace chassis_controllers
{
class OmniController : public ChassisBase
{
```

```
public:
  OmniController() = default;
  bool init(hardware_interface::RobotHW* robot_hw, ros::NodeHandle&
root_nh, ros::NodeHandle& controller_nh) override;

private:
  void moveJoint(const ros::Time& time, const ros::Duration&
period) override;

 std::vector<std::shared_ptr<effort_controllers::JointVelocityContr
oller>> joints_;
  Eigen::MatrixXd chassis2joints_;
};

}  // namespace chassis_controllers
```

omni.cpp

```
#include <string>
#include <Eigen/QR>

#include <rc_common/ros_utilities.h>
#include <pluginlib/class_list_macros.hpp>

#include "chassis_controllers/omni.h"

namespace chassis_controllers
{
bool OmniController::init(hardware_interface::RobotHW* robot_hw,
ros::NodeHandle& root_nh,
                          ros::NodeHandle& controller_nh)
{
  ChassisBase::init(robot_hw, root_nh, controller_nh);

  XmlRpc::XmlRpcValue wheels;
  controller_nh.getParam("wheels", wheels);
  chassis2joints_.resize(wheels.size(), 3);

  size_t i = 0;
  for (const auto& wheel : wheels)
  {
```

```
    ROS_ASSERT(wheel.second.hasMember("pose"));
    ROS_ASSERT(wheel.second["pose"].getType() ==
XmlRpc::XmlRpcValue::TypeArray);
    ROS_ASSERT(wheel.second["pose"].size() == 3);
    ROS_ASSERT(wheel.second.hasMember("roller_angle"));
    ROS_ASSERT(wheel.second.hasMember("radius"));

    // Ref: Modern Robotics, Chapter 13.2: Omnidirectional Wheeled
Mobile Robots
    Eigen::MatrixXd direction(1, 2), in_wheel(2, 2), in_chassis(2,
3);
    double beta = (double)wheel.second["pose"][2];
    double roller_angle = (double)wheel.second["roller_angle"];
    direction << 1, tan(roller_angle);
    in_wheel << cos(beta), sin(beta), -sin(beta), cos(beta);
    in_chassis << -(double)wheel.second["pose"][1], 1., 0.,
(double)wheel.second["pose"][0], 0., 1.;
    Eigen::MatrixXd chassis2joint = 1. /
(double)wheel.second["radius"] * direction * in_wheel * in_chassis;
    chassis2joints_.block<1, 3>(i, 0) = chassis2joint;

    ros::NodeHandle nh_wheel = ros::NodeHandle(controller_nh,
"wheels/" + wheel.first);

 joints_.push_back(std::make_shared<effort_controllers::JointVeloci
tyController>());
    if (!joints_.back()->init(effort_joint_interface_, nh_wheel))
      return false;
    joint_handles_.push_back(joints_[i]->joint_);

    i++;
  }
  return true;
}

void OmniController::moveJoint(const ros::Time& time, const
ros::Duration& period)
{
  Eigen::Vector3d vel_chassis;
  vel_chassis << vel_cmd_.z, vel_cmd_.x, vel_cmd_.y;
  Eigen::VectorXd vel_joints = chassis2joints_ * vel_chassis;
  for (size_t i = 0; i < joints_.size(); i++)
```

```
  {
    joints_[i]->setCommand(vel_joints(i));
    joints_[i]->update(time, period);
  }
}

}  // namespace chassis_controllers

PLUGINLIB_EXPORT_CLASS(chassis_controllers::OmniController,
controller_interface::ControllerBase)
```

完成了上面这些工作后，我们需要写一些config.yaml文件去描述一下controller和Joint limits，由于我们有位置传感器，故文件如下：

config.yaml

```
controllers:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 100

  chassis_controller:
    # ChassisBase
    type: chassis_controllers/OmniController
    publish_rate: 100
    timeout: 0.1
    pid_follow: { p: 5.0, i: 0, d: 0.3, i_max: 0.0, i_min: 0.0,
antiwindup: true, publish_state: true }
    # OmniController
    wheels:
      left_front:
        pose: [ 0.254, 0.254, 2.356 ] # x y beta
        joint: left_front_wheel_joint
        <<: &wheel_setting
          roller_angle: 0.
          radius: 0.07625
          pid: { p: 0.6, i: 0, d: 0.0, i_max: 0.0, i_min: 0.0,
antiwindup: true, publish_state: true }
      right_front:
        pose: [ 0.254, -0.254, 0.785 ]
        joint: right_front_wheel_joint
```

```
        <<: *wheel_setting
      left_back:
        pose: [ -0.254, 0.254, -2.356 ]
        joint: left_back_wheel_joint
        <<: *wheel_setting
      right_back:
        pose: [ -0.254, -0.254, -0.785 ]
        joint: right_back_wheel_joint
        <<: *wheel_setting
```

最后一步，我们只要写一个launch文件即可

```
<launch>
    <!-- 将 Urdf 文件的内容加载到参数服务器 -->
    <param name="robot_description"
        command="$(find xacro)/xacro $(find
urdf01_gazebo)/urdf/omni_chassis/chassis.urdf.xacro" />
    <!-- 启动 gazebo -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="paused" value="false"/>
        <arg name="world_name" value="$(find
urdf01_gazebo)/worlds/8week_slam.world"/>
    </include>

    <rosparam file="$(find rc_gazebo)/config/actions.yaml"
command="load" if="true"/>
    <!-- 在 gazebo 中显示机器人模型 -->
    <!-- <node pkg="gazebo_ros" type="spawn_model" name="model"
args="-urdf -model Three-wheel_chassis -param robot_description"
/> -->
    <node name="spawn_model" pkg="gazebo_ros" type="spawn_model"
clear_params="true"
        args="-param robot_description -urdf -model Three-
wheel_chassis -x 5.5 -y 0.5 -z 0.5" output="screen" />
</launch>
```

2.3 机器人仿真环境

2.3.1 urdf

在ros中的仿真环境可以通过其自带的gazebo软件，其主要文件格式是urdf

ros中的urdf（Unified Robot Description Format）是一种基于xml的标准化格式，用于描述机器人的结构、外观、物理属性及运动学特性，其主要有两种基本结构，分为连杆(link)和关节(joint)

> <link>：描述机器人的刚性部件（如机械臂的连杆、轮式机器人的底盘），包含以下子标签：
>
> <visual>：定义外观（几何形状、颜色、材质），例如使用<box>、<cylinder>或导入STL文件。
>
> <collision>：简化后的几何形状，用于物理引擎的碰撞检测。
>
> <inertial>：定义质量、质心位置和惯性矩阵，用于动力学仿真。
>
> <\link>
>
> <joint>：连接两个<link>并定义运动方式，常见类型包括：
>
> fixed（固定）、revolute（旋转受限）、continuous（无限旋转，如轮子）、prismatic（平移关节）。
>
> 需指定<parent>和<child>链接，以及运动轴（<axis>）和限制（<limit>）
>
> <\joint>

一般我们使用简单的差速轮小车，我们可以自己编写，但是用到一些比较复杂的车模，还是需要使用soliwork或者fusion360等软件把机械文件导成urdf格式，包括我们的仿真场地也可以使用机械软件绘制。但是ros官方的一些传感器插件我们还是需要自己手动加入，此时我们使用xacro宏进行导入。

一般我们可以将仿真文件进行一个xacro的模块化封装，例如：雷达，相机、车模等等我们都可以作为一个单独模块放在一个xacro文件中。最后在一个总包文件中进行调用即可。例码如下

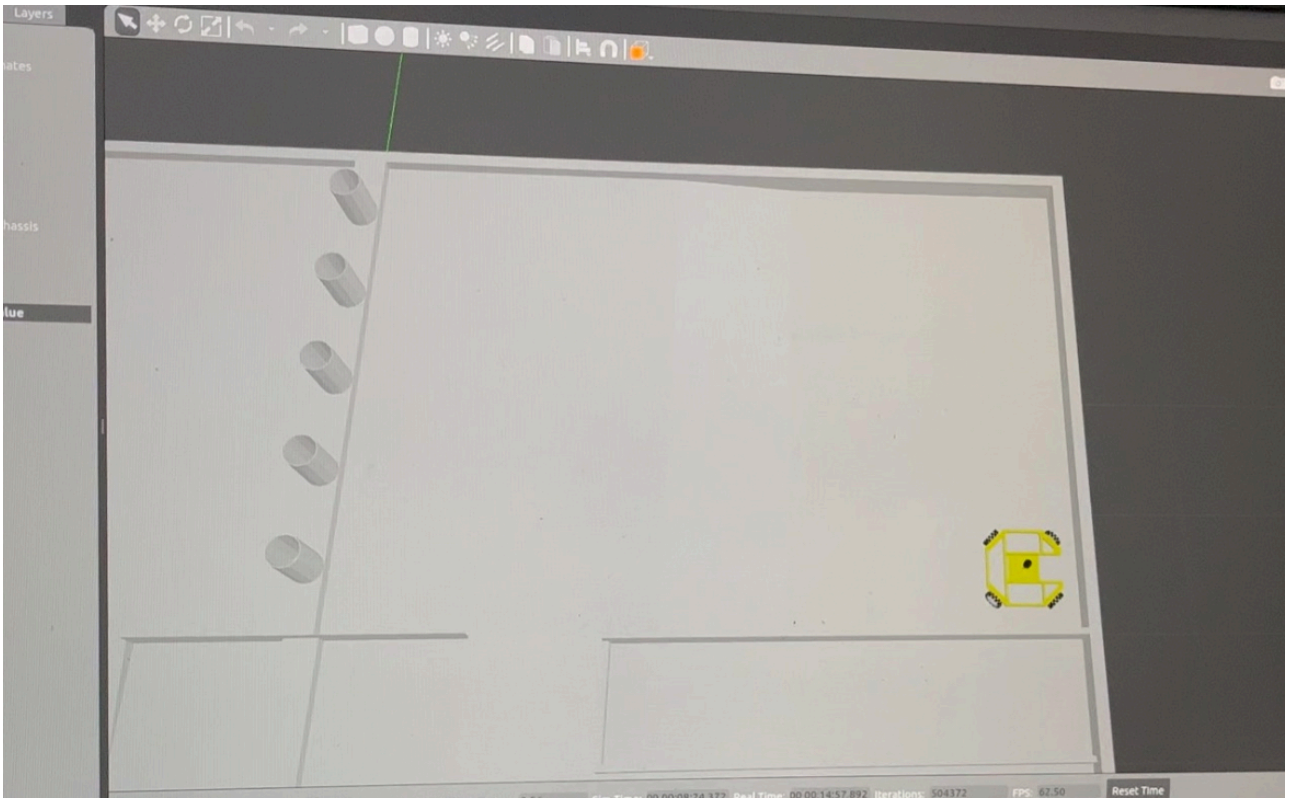
```
<xacro:ACTION
connected_to="base_link"action_name="location_action"use_sim="$(arg
use_sim)"
                        xyz="-0.035 0 0"
                        rpy="0.0 0.0 0.0"/>

<xacro:wheel_transmission prefix="left_front"
mechanical_reduction="-19.2032"/>
<xacro:wheel_transmission prefix="right_front"
mechanical_reduction="-19.2032"/>
<xacro:wheel_transmission prefix="left_back"
mechanical_reduction="-19.2032"/>
<xacro:wheel_transmission prefix="right_back"
mechanical_reduction="-19.2032"/>
```

最后在xacro文件中调用gazebo官方库以及我们自定义控制器的插件

```
<gazebo>
        <plugin name="rc_ros_control"
filename="librc_robot_hw_sim.so">
            <robotNamespace>/</robotNamespace>
            <robotSimType>rc_gazebo/rcRobotHWSim</robotSimType>
        </plugin>
</gazebo>
```

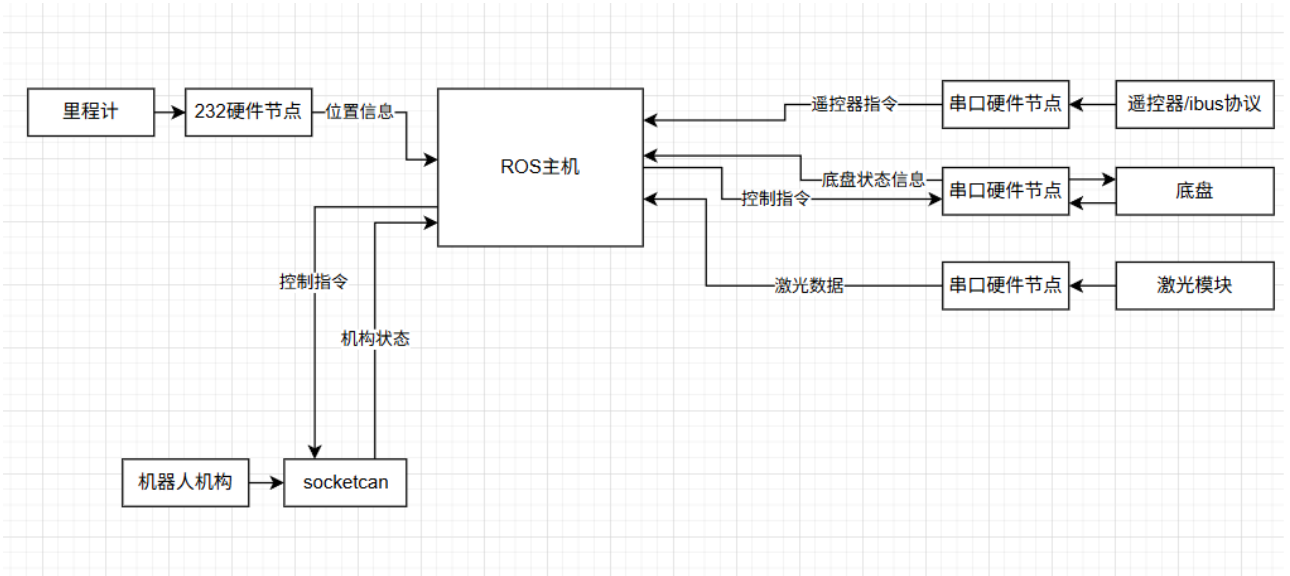
2.3.2 仿真表现

最终通过launch文件启动仿真，结果如下：



2.4 ROS系统与下位机的通讯连接

使用虚拟串口通讯与机器人各个机构进行通讯，由于一台机器人存在多个设备，会存在有多个通讯。ros主机和每个硬件的处理节点都使用话题通讯进行连接，统一消息类型都归于rc_msgs

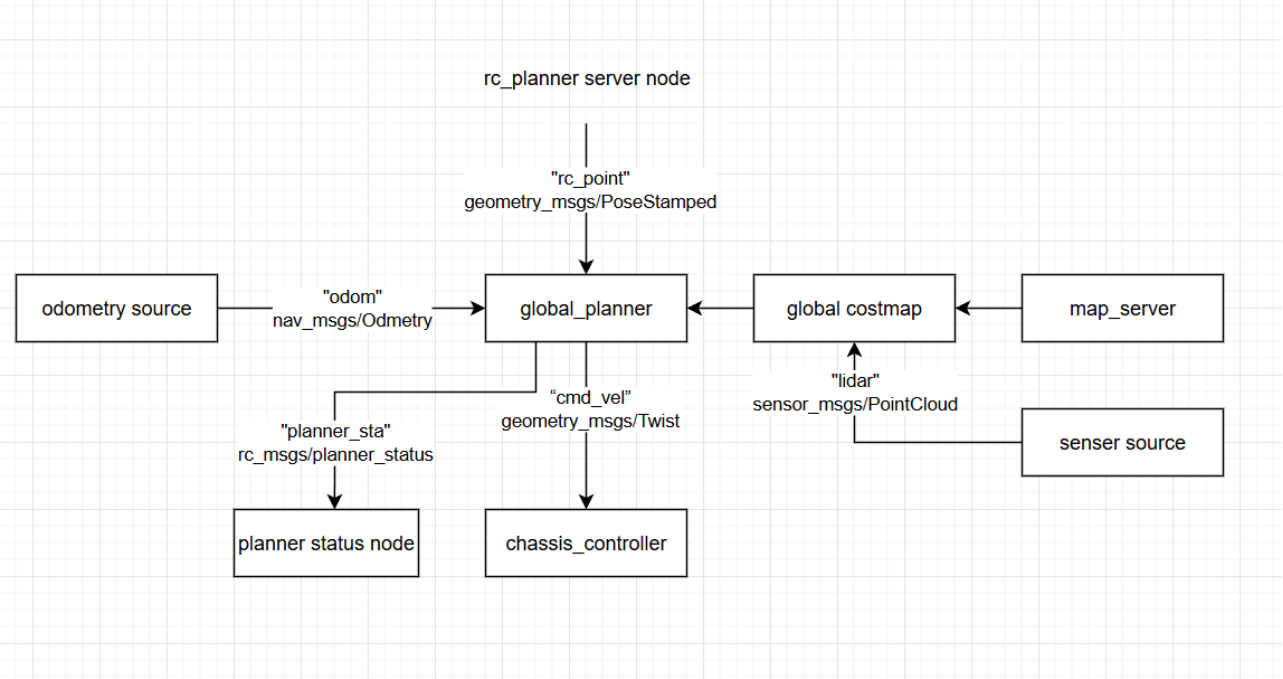


我们可以通过订阅每一个机构的相关话题来获取当前的状态信息，以及对某个话题广播指令对机器人进行控制

2.5 机器人路径规划

在ROBOCON比赛中，我们只有导航需求，对机器人底盘速度要求极高，为了追求极致的速度和稳定，我们在ros传统的导航框架下做了以下改动。删除了局部路径规划器，直接使用全局路径的规划路线进行跟踪，增加了Fuzzy PD跟踪器对全局路径进行跟踪。同时我们还在跟踪器中增加了机器人导航状态的发布节点，方便进行pid参数的可视化调试和底盘的速度位置信息打印

2.5.1 导航框图



2.5.2 参数配置

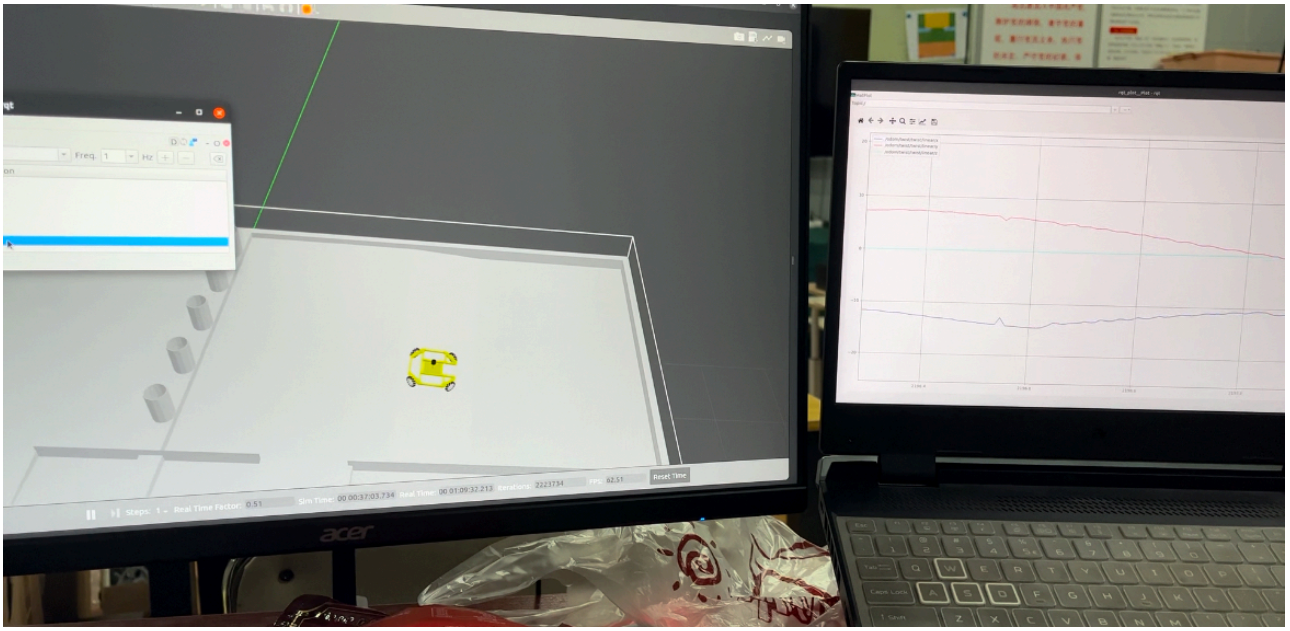
在参数管理以及调试方面，PID参数，路径参数都有配备的yaml文件进行管理。结合ROS本身自带的可视化界面，我们可以将规划后生成的曲线进行显示。使用rqt的系列工具可以查看机器人运动时的轨迹和我们定义的曲线拟合程度如何，还可以对PD跟踪器的参数进行实时调参。

```
if_use_gazebo: true  #是否使用gazebo进行仿真
if_use_debug: true  #是否开启路径调试模式
field_direction: "left"   #左右场地的镜像操作
joy_serial_port: "/dev/airRC" #手柄串口
stm32_serial_port: "/dev/upper" #stm32上层串口
chassis_serial_port: "/dev/chassis" #底盘串口
# joy_serial_port: "/dev/ttyUSB0"
# stm32_serial_port: "/dev/ttyUSB1"
control_frequence: 20  #Hz
```

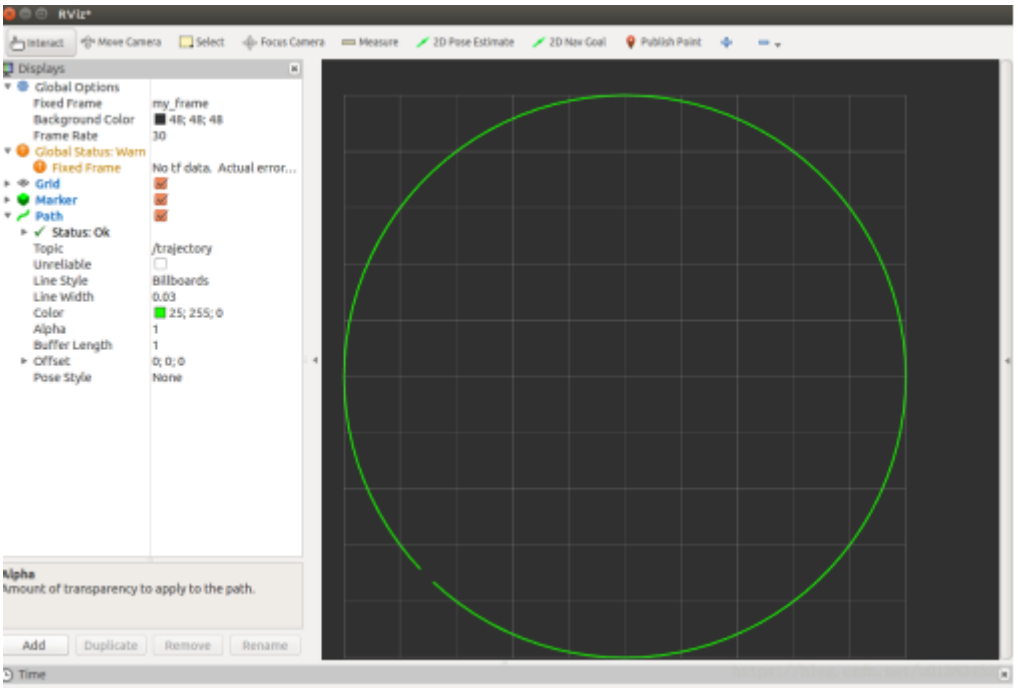
```yaml
take_ball:
  if_use_decision: false  #是否接入行为树的放球决策
  takeball_decision_num: 1  #决策方案 1 或 2
  decision_topic: "decision_topic"
  zone_decision_topic: "zone_decision_topic"
  y_max: 11 #自由取球的y坐标最大值
  y_min: 9 #自由取球的y坐标最小值
  x_abs_max: 5 #自由取球的x坐标绝对值最大值

path_plan:
  take_ball_move_speed: 1
  odom_topic: /robot_coordinate #最终的里程计话题，geometry_msgs/Point
  takeBall_ChangeDistance: 0.5 #目标点变化的阈值，大于这个阈值才会重新进行路径规划
  .......
```

2.5.3 成果

机器人参数可视化界面



在rviz中限制机器人实时路径



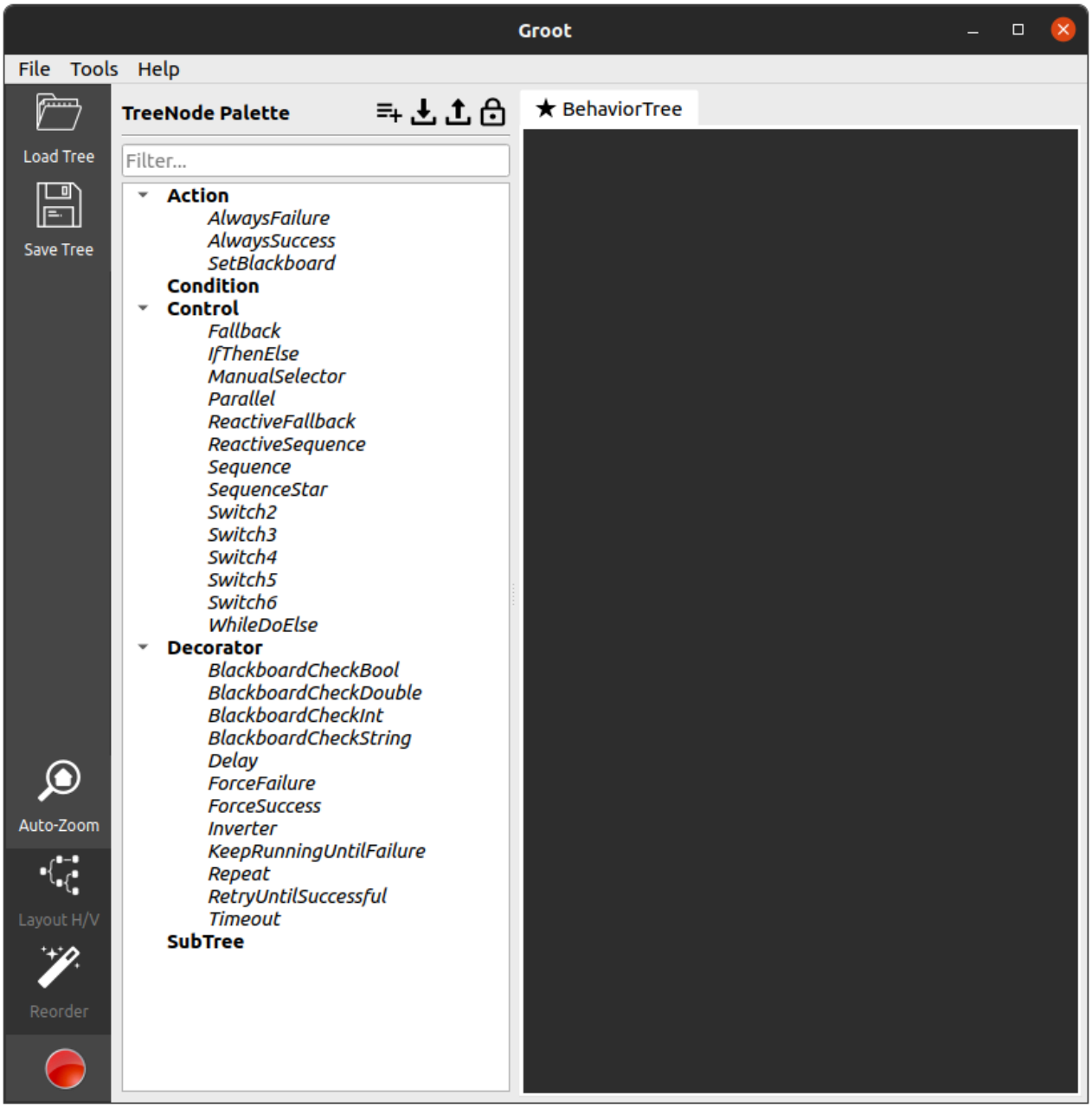
2.6 机器人行为树

在ROBCON 2024的比赛中，由于机器人是一个全自动状态且存在多个任务的切换，且任务的切换具有一定的逻辑性，故而使用BT行为树作为管理各个任务的工具。利用BT树，可以很方便地将每个任务作为一个节点，通过可视化软件Groot以某种逻辑关系连接起来，从而实现了各个任务的逻辑切换。以下便是比赛中各个任务切换的示意图，利用好BT树，能使得寻球、筛球、放球、移动与决策任务之间有一个很清晰的逻辑顺承关系，同时BT树还能很便捷地更改逻辑，便于调试使用。

2.6.1 行为树的主要优势

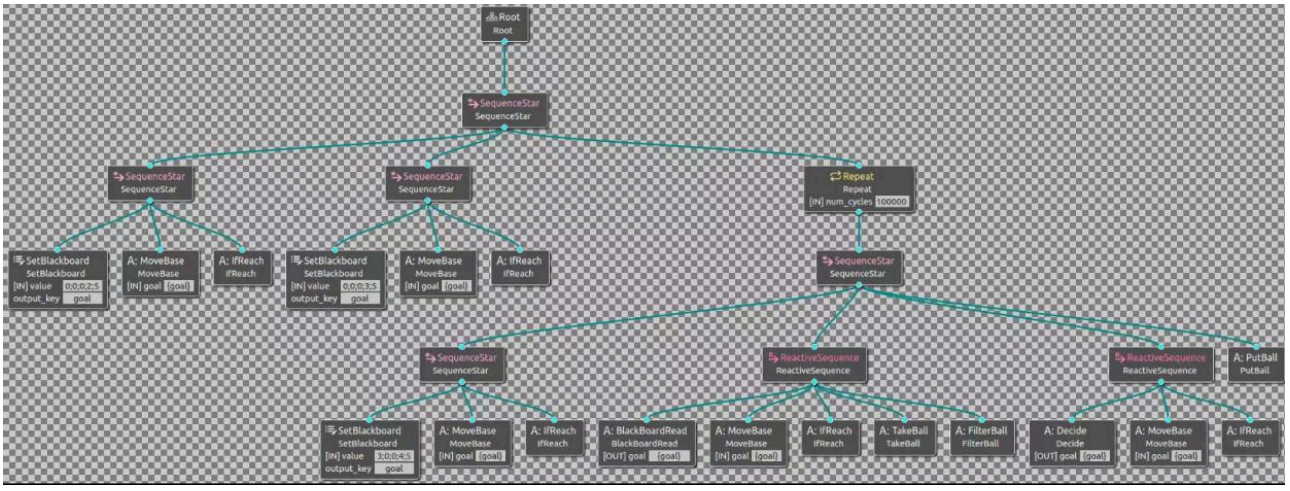
- 它们本质上是分层的: 我们可以组合复杂的行为，包括将整棵树作为更大的树的子树。例如，行为“ FetchBeer”可能会重用树“ GraspObject”。
- 它们的图形表示具有语义意义: 更容易“阅读”BT 并理解相应的工作流。通过比较，FSM 中的状态转换在文本和图形表示上都更难理解。

这里我们使用Groot进行树的编写，这是一个能够图形化构建BT树的软件。



2.6.2 成果

最终使用的BT树可视化逻辑如下图



3. 项目总结

3.1成果总结

在本次项目设计中，在电控方面上可以说是比较好的完成了本次RC的任务，并且取得了全国二等奖的好成绩，并且在机器人完成动作上也是较为流畅。不论在自动化的实现以及机器人的动作以及路径规划的调试上，都有这极高的表现。为后续机器人功能的迭代留下了良好的基础条件

技术实现上，我们采用了混合 A*算法作为全局路径规划策略，结合Fuzzy PD算法作为实时路径追踪器，确保了无人车在动态环境中的灵活应对与精确跟随

3.2 经验教训

- 关于自动路径：后续可以假如高阶的速度规划，将加速度体现到跟踪器中。建立一个完善的调参流程，引入Matlab仿真，对数据进行分析，已达到更好的效果
- 通讯连接：目前只有can通讯使用了socketCAN，并且将其封装在控制器体系下。后续应该考虑将串口通讯也进行封装，然后通过ros的话题进行接口暴露
- 仿真环境的完善，因为此次的仿真只引入了路径规划，但是机构执行的仿真并未合并在里面。后续应该考虑有一套完善的可视化仿真模块可以代替一些机构仿真