ROS2 学习笔记

——从入门到入土

Created with Typst

vkyuxr, Qwen

ROS2（Robot Operating System 2）是一个开源的机器人操作系统，旨在提供一个灵活的框架来开发机器人软件。 它支持多种编程语言和平台，具有分布式计算、实时性能和安全性等特点。

本文档的内容来源于我学习 ROS2 过程中的笔记，其来源于学习过程中的各个小工程。 本文档的知识框架可能并不全面，但是如果只是想让机器人动起来的话，应该是够用了。

Contents

1. ROS2 原理说明 3
   - 1.1. ROS2 主要构成 3
   - 1.2. Topic 的使用 4
   - 1.3. Service 的使用 5
   - 1.4. Action 的使用 5
2. ROS2 项目结构 6
   - 2.1. 创建 ROS2 项目的完整流程 6
     - 2.1.1. 创建并初始化工作空间 6
     - 2.1.2. 创建描述性资源 6
     - 2.1.3. 创建算法功能包 7
     - 2.1.4. 编译整个工作空间 8
     - 2.1.5. 启动目标节点 8
   - 2.2. `.urdf` 文件与 `.xacro` 的编写 8
     - 2.2.1. `.urdf` 文件的基本结构 8
     - 2.2.2. `.xacro` 文件的基本结构 10
     - 2.2.3. `.xacro` 文件的使用 12
   - 2.3. `.launch.py` 文件的编写 12
   - 2.4. 算法功能包的编写 13
     - 2.4.1. 功能包的基本结构 13
     - 2.4.2. 节点文件的具体实现 14

3. joint 的控制 16
3.1. URDF 文件的配置 16
3.2. yaml 文件的配置 17
3.3. 控制器的启动 18
4. sensor 的使用 20
4.1. 机器人姿态的获取 20
4.2. 激光雷达的使用 21
4.3. 对碰撞进行检测 24
5. 使用 ROS2 进行强化学习 26
5.1. 将 ROS2 封装为 Gymnasium 环境 26
5.2. 使用 Stable Baselines3 进行训练和评估 27
5.3. 使用 RLlib 进行训练和评估 29

1. ROS2 原理说明

1.1. ROS2 主要构成

ROS2 本质上是一个分布式通信框架，主要由 Node（节点）与通信机制组成。Node 是 ROS2 的基本执行单元，负责处理特定任务。 通信机制则包括 Topic（话题）、Service（服务）和 Action（动作），用于实现节点间的数据交换。

ROS2 的工作是通过使用通信机制在不同的 Node 之间传递消息来实现的。 每个 Node 可以发布消息到一个或多个消息，也可以订阅其他 Node 发布的消息。 此外，Node 还可以提供 Service 供其他 Node 调用，或者执行 Action 以处理复杂任务。

为了便于数据的交换，ROS2 提供了一些常用的基础数据类型：

基础数据类型	描述
`std_msgs/msg/Bool`	布尔类型
`std_msgs/msg/Byte`	字节类型
`std_msgs/msg/Char`	字符类型
`std_msgs/msg/Float32`, `Float64`	浮点数类型
`std_msgs/msg/Int8`, `UInt8`	有符号和无符号 8 位整数类型
`std_msgs/msg/Int16`, `UInt16`	有符号和无符号 16 位整数类型
`std_msgs/msg/Int32`, `UInt32`	有符号和无符号 32 位整数类型
`std_msgs/msg/Int64`, `UInt64`	有符号和无符号 64 位整数类型
`std_msgs/msg/String`	字符串类型

进一步的，有如下复杂数据类型（这里仅列举了一部分）：

复杂数据类型	描述
`sensor_msgs/msg/Image`	图像数据类型，用于表示图像信息
`sensor_msgs/msg/PointCloud2`	点云数据类型，用于表示三维点云信息
`nav_msgs/msg/Odometry`	里程计数据类型，用于表示机器人位置和速度

对于静态的数据（如 URDF 模型），ROS2 提供了参数服务器机制对其进行管理。可以通过命令行的方方式对参数进行发布和提取：

```
ros2 param set /node_name param_name value
```

```
ros2 param get /node_name param_name
```

在 Python 脚本中，可以在 Node 中通过如下方式进行参数的声明和获取：

```
self.declare_parameter("my_int_param", 42)

int_val =
self.get_parameter("my_int_param").get_parameter_value().integer_value
```

如果要在 Python 中进行跨节点的参数服务器访问，需要实现对应的服务端和客户端，较为复杂，不推荐。推荐使用 Python 创建命令行子进程进行访问。

1.2. Topic 的使用

Topic 是一种异步通信机制，使用发布订阅模型（Publish-Subscribe Model）来实现节点间的数据交换。 节点可以发布消息到一个 Topic，也可以订阅其他节点发布的消息。

可以在终端中使用`ros2 topic`命令查看当前系统中的 Topic 信息。

在 Python 中可以使用如下方法创建一个 Node 并发布消息到 Topic：

```
from rclpy.node import Node
from std_msgs.msg import Float64MultiArray

class WheelVeloctyPublisher(Node):
    def __init__(self):
        super().__init__('wheel_velocty_publisher')
        self.wheel_velocity_publisher = self.create_publisher(
            Float64MultiArray,
            "/wheel/commands",
            10
        )

    def publish_wheel_velocity(self, velocity):
        msg = Float64MultiArray()
        msg.data = array('d', velocity)
        self.wheel_velocity_publisher.publish(msg)
```

上面这个发布者例子中，定义了一个继承自`Node`的类`WheelVeloctyPublisher`，在其构造函数中创建了一个发布者`wheel_velocity_publisher`，向名为`/wheel/commands`的 Topic 发送`Float64MultiArray`类型的数据。 在类函数`publish_wheel_velocity`中，创建了一个消息对象`msg`，并将传入的速度数据赋

值给`msg.data`，最后调用`self.wheel_velocity_publisher.publish(msg)`将消息发布到 Topic。

可以使用如下方式创建一个订阅者：

```
from rclpy.node import Node
from std_msgs.msg import Float64MultiArray

class WheelVeloctySubscriber(Node):
    def __init__(self):
        super().__init__('wheel_velocity_subscriber')
        self.wheel_velocity_subscriber = self.create_subscription(
            Float64MultiArray,
            "/wheel/commands",
            self.wheel_velocity_handler,
            10
        )

    def wheel_velocity_handler(self, msg):
        // 处理接收到的消息
        pass
```

上面这个订阅者例子通过`self.create_subscription()`创建了一个订阅者`wheel_velocity_subscriber`，订阅名为`/wheel/commands`的 Topic，并指定消息类型为`Float64MultiArray`。当接收到消息时，会调用`wheel_velocity_handler`方法，在该方法中处理接收到的消息。

以上便是 ROS2 中 Topic 的基本使用方法。

1.3. Service 的使用

1.4. Action 的使用

2. ROS2 项目结构

由于 ROS2 并没有集成开发环境，所以理清楚 ROS2 项目的结构对 ROS2 的使用非常重要的。 为了记录完整的 ROS2 学习流程，我们将从 ROS2 项目的创建开始。

ROS2 的安装过程在 https://docs.ros.org/ 中有详细的说明。 这里跳过了 ROS2 的安装过程，假设你已经安装好了 ROS2，并且可以使用`ros2`命令。

2.1. 创建 ROS2 项目的完整流程

ROS2 项目包含多个包（Package），每个包可以包含多个节点（Node）。 每个节点可以实现特定的功能，如传感器数据处理、控制算法等。

2.1.1. 创建并初始化工作空间

在 ROS2 中，工作空间是一个包含多个包的目录结构。 我们可以使用以下命令创建一个新的工作空间：

```
cd <path-of-workspace>
mkdir -p <name-of-workspace>
```

然后需要创建一个文件夹用于放置功能包的源代码：

```
mkdir -p <name-of-workspace>/src
```

接下来，使用`colcon`命令初始化工作空间：

```
cd <name-of-workspace>
colcon build
source install/setup.bash
```

这将编译所有的包，并将它们安装到工作空间的`install`目录中。 编译完成后，我们需要使用`source`命令来设置环境变量，以便 ROS2 能够找到我们编译的包。第一次时可能会报错，因为还没有编译内容，这一步主要是初始化环境。

2.1.2. 创建描述性资源

为了保持工作空间的整洁和有序，我们通常会将描述性资源（如机器人模型、环境等）放在一个单独的包中。 在这个包中，我们只需要实现描述性资源的存放和加载功能，而不需要实现具体的算法或控制逻辑。

在工作空间的`src`目录下创建一个新的包来存放这些资源。可以使用以下命令：

```
cd src
ros2 pkg create <name-of-package> --build-type ament_cmake
```

1. `<name-of-package>`是你要创建的包的名称，建议将其命名为`<name>_description`，表示这是一个描述性包。
2. `--build-type`参数指定了包的构建类型，`ament_cmake`是 ROS2 中常用的构建类型，表示使用 CMake 进行构建。`--build-type`参数的值可以是`ament_cmake`或`ament_python`，具体取决于你要实现的功能。`ament_cmake`适用于 C++包，而`ament_python`适用于 Python 包。通常描述性资源包使用`ament_cmake`。

在使用 ROS2 进行仿真时，有两种资源必不可少：一是控制对象，即机器人的模型。在 ROS2 中，通常使用 URDF（Unified Robot Description Format）来描述机器人的模型。二是环境，即机器人的工作场景，通常使用 Gazebo 等仿真工具来创建。

在`<name-of-package>`中添加`urdf`目录和`worlds`目录，并放入对应的资源文件。`urdf`目录用于存放机器人的 URDF 文件，而`worlds`目录用于存放 Gazebo 的世界文件。

然后，在`<name-of-package>/launch`中创建`.launch.py`文件用于加载资源。

上述资源只是我们在创建源代码时的内容. 在 ROS2 的实际工作过程中，ROS2 并不会在我们存放文件的目录中直接使用这些资源，而是在使用`colcon build`命令的时候将其安装到工作空间的`install`目录中。 因此我们还需要在包的根目录下创建一个`CMakeLists.txt`文件来描述如何安装这些资源。

在我们使用`ros2 pkg create`命令时，ROS2 会自动生成一个基本的`CMakeLists.txt`文件。 在这个文件中，我们需要添加如下指令来安装 URDF 和 launch 文件：

```
install(
  DIRECTORY urdf launch
  DESTINATION share/${PROJECT_NAME}/
)
```

2.1.3. 创建算法功能包

回到`src`目录，并创建算法功能包：

```
ros2 pkg create <name-of-package> --build-type <ament_type>
```

其中`<ament_type>`可以是`ament_cmake`或`ament_python`，具体取决于你要实现的功能。

在这里仅创建一个空的包，而忽略这个包中具体的算法实现。

2.1.4. 编译整个工作空间

为了运行 ROS2 项目，我们需要编译整个工作空间。 在工作空间的根目录下，使用以下命令编译整个工作空间：

```
cd <name-of-workspace>
colcon build
source install/setup.bash
```

2.1.5. 启动目标节点

ROS2 的具体运行方式通常是通过`ros2 launch`命令来启动一个或多个节点。 在运行之前，我们需要确保工作空间已经编译成功，并且环境变量已经设置好。 然后使用以下命令来运行一个 launch 文件：

```
ros2 launch <name-of-package> <name-of-launch>
```

2.2. `.urdf`文件与`.xacro`的编写

URDF（Unified Robot Description Format）是 ROS2 中用于描述机器人的模型文件格式，通常包含机器人的几何形状、关节、传感器等信息。 其通常位于包的`urdf`目录下，文件名通常以`.urdf`或`.xacro`结尾。 URDF 文件可以使用 XML 格式编写，也可以使用 Xacro（XML Macros）格式编写。 Xacro 是一种 ROS2 提供的宏语言，可以简化 URDF 文件的编写。

2.2.1. `.urdf`文件的基本结构

`.urdf`文件可以理解为是由多个模块组成的，每个模块描述了机器人的一个部分。 下面是一些 URDF 文件的基本元素：

元素	作用
link	描述机器人的一个部分，如手臂、腿等
joint	描述机器人的关节，如旋转关节、滑动关节等

link 的参数使用一对`<link>`标签来定义，通常包含以下部分：

```
<link name="link_name">

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
    <material name="material_name"/>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="1 1 1"/>
    </geometry>
  </collision>

  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <mass value="1.0"/>
    <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0"
izz="0.1"/>
  </inertial>

</link>
```

- `<visual>`：用于描述机器人的外观。`name`属性指定了链接的名称。
  - `<origin>`：指定了几何形状的原点位置`xyz`和朝向`rpy`。
  - `<geometry>`：指定几何形状，可以是`box`、`cylinder`、`sphere`等。
  - `<material>`：指定几何形状的材质，可以指定颜色、纹理等。
- `<collision>`：指定机器人的碰撞体积，用于物理仿真。
  - `<origin>`：指定了碰撞体积的原点位置`xyz`和朝向`rpy`。
  - `<geometry>`：指定了碰撞体积的几何形状，通常与`<visual>`元素相同。
- `<inertial>`：描述了机器人的惯性属性。
  - `<origin>`：指定了惯性属性的原点位置`xyz`和朝向`rpy`。
  - `<mass>`：指定了机器人的质量。
  - `<inertia>`：指定了机器人的惯性矩阵，包括`ixx`、`iyy`、`izz`等元素。

joint 的参数使用一对`<joint>`标签来定义，通常包含以下部分：

```
<joint name="joint_name" type="revolute">
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <parent link="parent_link_name"/>
  <child link="child_link_name"/>
```

```
  <axis xyz="1 0 0"/>
  <limit effort="10.0" velocity="1.0" lower="-1.57" upper="1.57"/>
  <dynamics damping="0.1" friction="0.1"/>
</joint>
```

`name`用于定义关节的名称，`type`用于定义关节的类型。`type`可以是以下几种类型：

类型	自由度	描述
`fixed`	0	固定关节，不允许移动
`prismatic`	1	滑动关节，允许沿一个轴滑动
`revolute`	1	旋转关节，允许绕一个轴旋转
`continuous`	1	连续旋转关节，允许绕一个轴无限旋转
`planar`	3	平面关节，允许在一个平面内移动
`floating`	6	浮动关节，允许在三维空间内自由移动

- `<origin>`：指定了关节的原点位置`xyz`和朝向`rpy`。
- `<parent>`：指定了关节的父链接`link`。
- `<child>`：指定了关节的子链接`link`。
- `<axis>`：指定了关节的旋转轴或滑动轴`xyz`。
- `<limit>`：指定了关节的限制条件，如最大力矩`effort`（N 或 N · m）、最大速度`velocity`（m/s 或 rad/s）、最小位移`lower`（m 或 rad）和最大位移`upper`（m 或 rad）。
- `<dynamics>`：指定了关节的动力学属性，如阻尼`damping`和摩擦`friction`。

URDF 可以看作是一个将 link 作为边，joint 作为节点的图结构。

可以使用`urdf_check`命令来检查 URDF 文件的语法是否正确：

```
urdf_check <urdf-file>
```

这里只介绍最简单的 URDF 文件结构，更多的 URDF 元素和属性会在之后的具体实例中介绍。

2.2.2. `.xacro`文件的基本结构

`.xacro`文件是 ROS2 中用于生成 URDF 文件的宏语言文件，其基本结构与 URDF 文件类似，但它支持宏定义和参数化，可以更方便地生成复杂的 URDF 文件。`.xacro`文件可以使用以下标签对象来定义：

元素	作用
`<xacro:property>`	定义一个属性，可以在 URDF 中使用
`<xacro:macro>`	定义一个宏，可以在 URDF 中调用
`<xacro:include>`	包含其他 Xacro 文件

`.xacro`的使用方法用下面的例子来说明：

```
<?xml version="1.0"?>
<xacro:property name="PI" value="3.141592653589793"/>
<xacro:property name="wheel_radius" value="0.05"/>
<xacro:property name="wheel_width" value="0.03"/>
<xacro:property name="wheel_mass" value="0.5"/>

<xacro:macro name="wheel" params="name prefix parent xyz rpy">
  <link name="${name}">
    <visual>
      <geometry>
        <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
      </geometry>
      <origin xyz="0 0 0" rpy="${PI/2} 0 0"/>
      <material name="black">
        <color rgba="0 0 0 1"/>
      </material>
    </visual>

    <collision>
      <geometry>
        <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
      </geometry>
      <origin xyz="0 0 0" rpy="${PI/2} 0 0"/>
    </collision>

    <inertial>
      <mass value="${wheel_mass}"/>
      <inertia ixx="0.001" ixy="0.0" ixz="0.0"
               iyy="0.001" iyz="0.0"
               izz="0.001"/>
    </inertial>
  </link>

  <joint name="${prefix}_wheel_joint" type="continuous">
    <parent link="${parent}"/>
    <child link="${name}"/>
    <origin xyz="${xyz}" rpy="${rpy}"/>
    <axis xyz="0 1 0"/>
```

```
  </joint>
</xacro:macro>

<xacro:wheel name="left_wheel" prefix="left" parent="base_link"
  xyz="${base_length/2} ${-base_width/2} 0" rpy="0 0 0"/>
```

在这个例子中，我们定义了一个轮子的宏`<xacro:macro>`，它接受四个参数：`name`、`prefix`、`parent`、`xyz`和`rpy`。这个宏生成了一个轮子的链接和一个关节。我们可以在 URDF 中多次调用这个宏来生成多个轮子。`.xacro`文件的语法与 URDF 文件类似，但它支持更多的功能，如宏定义、属性定义和条件判断等。

2.2.3. `.xacro`文件的使用

可以使用`xacro`命令将`.xacro`文件转换为`.urdf`文件：

```
xacro <xacro-file> -o <urdf-file>
```

也可以在 ROS2 的 launch 文件中直接使用`.xacro`文件，而不需要手动转换为`.urdf`文件：

```
import os
import xacro

xacro_file = os.path.join(path, 'urdf', 'my_xacro.xacro')
robot_model = xacro.parse(open(xacro_file)).toxml()

node_robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    output='screen',
    parameters=[{'robot_description': robot_model}]
)
```

2.3. `.launch.py`文件的编写

在 ROS2 中，`.launch.py`文件用于定义如何启动一个或多个节点，以及它们之间的关系。`.launch.py`文件通常位于包的`launch`目录下，用于描述如何启动 ROS2 节点、加载参数、设置环境变量等。通常一个`.launch.py`文件都包含有下面三个 Python 包：

```
import os # 用于操作系统路径操作。
from launch import LaunchDescription # 启动描述类，用来定义要启动的节点。
from launch_ros.actions import Node # 定义一个 ROS 节点。

# 其他常用
from ament_index_python.packages import get_package_share_directory #
获取指定 ROS 包的共享目录路径（通常存放配置、URDF、meshes 等资源文件）。
```

每个`.launch.py`文件都需要实现`generate_launch_description()`函数。 这个函数返回一个`LaunchDescription`对象，描述了要启动的节点和其他相关配置。

使用如下方式实例化 ROS2 节点描述：

```
NODE_INSTENCE_NAME = Node(
    package=PACKAGE_NAME_STR,
    executable=PROGRAM_IN_PACKAGE_STR,
    name=NODE_NAME_STR,
    output='both', # 输出日志信息到控制台和日志文件。
    parameters=[{PARAMETER_NAME_STR: PARAMETER_VAR}],
)
```

最后需要将实例化的节点描述传给 ROS2：

```
return LaunchDescription([
    NODE_INSTENCE_NAME
])
```

2.4. 算法功能包的编写

功能包是 ROS2 中实现具体功能的核心部分。 功能包通常包含一个或多个节点，每个节点实现特定的功能。

2.4.1. 功能包的基本结构

功能包可以使用 C++或 Python 编写。 在这里以 Python 为例，介绍功能包的基本结构。 功能包目录通常如下：

```
package_name/
├── package.xml
├── setup.py
└── package_name/
    ├── __init__.py
```

```
    └── node_1_name.py
    └── node_2_name.py
```

在`setup.py`中注册对应的功能：

```
entry_points={
    'console_scripts': [
        'my_node_1_exe = package_name.node_1_name:main',
        'my_node_2_exe = package_name.node_2_name:main',
    ],
},
```

`my_node_1_exe`和`my_node_2_exe`的标识符是任意定义的，用于调用，比如在`launch.py`的`Node()`中，将其作为`executable`参数传递给`Node()`：

```
executable='my_node_1_exe'
```

2.4.2. 节点文件的具体实现

节点文件中，都需要实现一个继承自`rclpy.node.Node`的类，用于获取节点的基本能力。如：

```
import rclpy
from rclpy.node import Node

class MyNode(Node):
    def __init__(self):
        super().__init__('node_name')

        # 初始化逻辑
        self.declare_parameter('a', 0)
        a =
self.get_parameter('a').get_parameter_value().integer_value

        self.get_logger().info(f'a = {a}')

def main():
    rclpy.init()
    node = MyNode()
    rclpy.spin(node)
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

其中，`self.declare_parameter('<name-of-parameter>', <value-of-default>)`用于声明一个节点可以接受的参数。这个参数可以从`launch.py`文件或命令行传入：

```
ros2 run my_package my_node --ros-args -p a:=5
```

或者在`launch.py`文件里，使用下面代码将参数传递给`Node()`：

```
parameters=[{'a': 5}]
```

函数	作用
`self.get_parameter('a')`	获取参数对象
`.get_parameter_value()`	取参数值部分（可能是 int、double、str 等类型）
`.integer_value`	获取其整数类型的值

3. joint 的控制

joint 是机器人控制的核心，对机器人的控制就是对 joint 的控制。 用于 ROS2 本身只是一个分布式的多节点通信框架，并不能实现控制器的功能，所以需要安装额外的控制器功能包`ros2_control`。 该功能包提供了控制器的接口和实现，支持多种类型的控制器，如位置控制器、速度控制器和力控制器等。

要使用`ros2_control`，需要依次完成以下步骤：

1. 安装`ros2_control`和相关依赖包。
2. 创建控制器配置文件，定义需要使用的控制器类型和参数。
3. 在机器人描述文件中添加控制器配置。
4. 启动控制器管理器节点，加载控制器配置。
5. 启动控制器，开始控制机器人。

安装`ros2_control`和相关依赖包，可以使用以下命令安装：

```
sudo apt install ros-<distro>-ros2-control ros-<distro>-ros2-
controllers
```

其中`<distro>`是 ROS2 的发行版名称，如`humble`。

要使用控制器对 joint 进行控制，主要是进行控制器的配置和启动。

3.1. URDF 文件的配置

在 URDF 文件中，需要添加控制器的配置。以下是一个示例配置：

```
<!-- 配置 ROS2 Control -->
<ros2_control name="GazeboSystem" type="system">
    <hardware>
        <plugin>gazebo_ros2_control/GazeboSystem</plugin>
    </hardware>
    <joint name="wheelRR_continuous">
        <command_interface name="velocity">
            <param name="min">-10</param>   <!-- rad/s -->
            <param name="max">10</param>    <!-- rad/s -->
        </command_interface>
        <state_interface name="position"/>
        <state_interface name="velocity"/>
    </joint>
</ros2_control>
```

上面的例子中，`<ros2_control>`标签定义了控制器的配置，`<hardware>`标签指定了使用的硬件插件，`<joint>`标签定义了需要控制的关节。对`wheelFL_continuous`关节，配置了速度控制接口，并设置了速度的最小值和最大值，并且定义了位置和速度的状态接口。

然后需要启用控制器管理器插件，以便在 Gazebo 中加载和管理控制器。以下是一个示例配置：

```
<!-- Gazebo 通用插件 -->
<gazebo>
    <!-- 控制输入 -->
    <plugin filename="libgazebo_ros2_control.so"
name="gazebo_ros2_control">
        <!-- /{controller_name}/commands -->
        <parameters>$(find agent_description)/config/controller.yaml</
parameters>
    </plugin>
</gazebo>
```

在这个配置中，`<plugin>`标签指定了使用的 Gazebo 插件，并通过`<parameters>`标签指定了控制器的配置文件路径。

3.2. yaml 文件的配置

在上一节的 URDF 文件中，我们已经配置了对`wheelRR_continuous`关节的控制接口。接下来，我们需要创建一个 yaml 文件来定义控制器的参数。

```
controller_manager:
  ros__parameters:
    update_rate: 100
    use_sim_time: true

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    wheel_controller:
      type: velocity_controllers/JointGroupVelocityController

wheel_controller:
  ros__parameters:
    joints:
      - wheelRR_continuous
    command_interfaces:
      - velocity
```

```
    state_interfaces:
      - position
      - velocity
```

在这个 yaml 文件中，我们定义了控制器管理器的参数，包括更新频率和是否使用模拟时间。然后，我们定义了一个关节状态广播器`joint_state_broadcaster`，以及一个速度控制器`wheel_controller`。

3.3. 控制器的启动

在配置好 URDF 文件和 yaml 文件后，我们需要在 launch.py 文件中启动控制器管理器和控制器。以下是一个示例的 launch.py 文件：

```
from launch.event_handlers import OnProcessExit
from launch.actions import ExecuteProcess, RegisterEventHandler
from launch import LaunchDescription

import xacro

def generate_launch_description():
    load_joint_state_broadcaster = ExecuteProcess(
        cmd=['ros2', 'control', 'load_controller', '--set-state',
'start', 'joint_state_broadcaster'],
        output='screen'
    )

    load_wheel_controller = ExecuteProcess(
        cmd=['ros2', 'control', 'load_controller', '--set-state',
'start', 'wheel_controller'],
        output='screen'
    )

    load_steering_controller = ExecuteProcess(
        cmd=['ros2', 'control', 'load_controller', '--set-state',
'start', 'steering_controller'],
        output='screen'
    )

    return LaunchDescription([
        RegisterEventHandler(
            event_handler=OnProcessExit(
                target_action=spawn_entity_node,
                on_exit=[load_joint_state_broadcaster],
            )
        ),
```

```
        RegisterEventHandler(
            event_handler=OnProcessExit(
                target_action=load_joint_state_broadcaster,
                on_exit=[load_wheel_controller],
            )
        ),
        RegisterEventHandler(
            event_handler=OnProcessExit(
                target_action=load_joint_state_broadcaster,
                on_exit=[load_steering_controller],
            )
        )
    ])
```

在上面的例子中，我们使用`ExecuteProcess`来执行 ROS2 控制器的加载命令。首先加载关节状态广播器，然后加载轮子控制器和转向控制器。在`OnProcessExit`事件处理器中，我们确保在实体节点生成后加载关节状态广播器，然后加载轮子控制器和转向控制器。

4. sensor 的使用

sensor 是实现反馈控制的关键组件。它们提供了关于机器人状态和环境的信息，使得控制器能够做出相应的调整。 在 ROS2 中，传感器通常通过发布消息到特定的主题来工作。控制器可以订阅这些主题，以获取传感器数据并进行处理。 传感器可以是各种类型的，例如激光雷达、摄像头、IMU 等。每种传感器都有其特定的消息类型和数据格式。

4.1. 机器人姿态的获取

首先需要在 URDF 文件中添加一个 Gazebo 插件，用于输出机器人的位姿信息。

```
  <!-- Gazebo 通用插件 -->
  <gazebo>
      <!-- 位姿输出 -->
      <plugin name="gazebo_ros_p3d" filename="libgazebo_ros_p3d.so">
          <!-- /odom -->
          <alwaysOn>true</alwaysOn>
          <updateRate>50.0</updateRate>
          <body_name>wheelFC_steering_link</body_name>
          <gaussianNoise>0.0</gaussianNoise>
      </plugin>
  </gazebo>
```

然后在 Node 中订阅对应的主题来获取位姿信息，并对其进行处理。

```
from nav_msgs.msg import Odometry

# 订阅机器人位姿
self.odometry_subscription = self.create_subscription(
    Odometry,
    "/odom",
    self.odometry_handler,
    10
)

def odometry_handler(self, msg):
    # 数据获取
    position = np.array([
        msg.pose.pose.position.x,
        msg.pose.pose.position.y,
        msg.pose.pose.position.z
    ])
```

```
    orientation = np.array([
        msg.pose.pose.orientation.x,
        msg.pose.pose.orientation.y,
        msg.pose.pose.orientation.z,
        msg.pose.pose.orientation.w,
    ])
    linear = np.array([
        msg.twist.twist.linear.x,
        msg.twist.twist.linear.y,
        msg.twist.twist.linear.z
    ])
    angular = np.array([
        msg.twist.twist.angular.x,
        msg.twist.twist.angular.y,
        msg.twist.twist.angular.z
    ])
    # 其他处理
```

4.2. 激光雷达的使用

首先需要在 URDF 文件中添加一个 Gazebo 插件，用于模拟雷达。

```
<!-- 激光雷达插件 -->
<gazebo reference="laser_link">
    <sensor type="ray" name="lidar_sensor">
        <pose>0 0 0 0 0 0</pose>
        <visualize>true</visualize>
        <update_rate>40</update_rate>               <!-- 更新率 40Hz --
>
        <ray>
            <scan>
                <horizontal>                        <!-- 水平方向 -->
                    <samples>180</samples>          <!-- 180 线 -->
                    <min_angle>-1.5708</min_angle>  <!-- -180 度 -->
                    <max_angle>1.5708</max_angle>   <!-- 180 度 -->
                </horizontal>
                <vertical>                          <!-- 垂直方向 -->
                    <samples>16</samples>           <!-- 16 线 -->
                    <min_angle>-0.2618</min_angle>  <!-- -15 度 -->
                    <max_angle>0.2618</max_angle>   <!-- 15 度 -->
                </vertical>
            </scan>
            <range>
                <min>0.15</min>                     <!-- 最小探测距离
0.14m -->
```

```
                <max>10.0</max>                     <!-- 最大探测距离
10m -->
                <resolution>0.01</resolution>       <!-- 分辨率 0.01m
-->
            </range>
        </ray>
        <plugin name="gazebo_ros_lidar_controller"
filename="libgazebo_ros_ray_sensor.so">
            <!-- /gazebo_ros_lidar_controller/out -->
            <frame_name>laser_link</frame_name>
        </plugin>
    </sensor>
</gazebo>
```

雷达的输出是点云数据，我们可以直接对其处理，或者转换为深度图。

```
from sensor_msgs.msg import PointCloud2

# 创建订阅者
self.laserScan_subscription = self.create_subscription(
    PointCloud2,
    "/gazebo_ros_lidar_controller/out",
    self.laserScan_processor,
    10,
)
```

```
from sensor_msgs.msg import Image

# 创建发布者
self.depthImage_publisher = self.create_publisher(
    Image,
    "/depth_image",
    10,
)

def laserScan_processor(self, msg):
    # 初始化深度图像（使用最大值填充）
    depth_array = np.full((self.depthImage_height,
self.depthImage_width),
                          self.distance_max,
                          dtype=np.float32)

    # 批量处理点云
    points = np.array(list(pc2.read_points(msg, field_names=("x", "y",
"z"), skip_nans=True)))
```

```
    if len(points) == 0:
        return

    # 提取坐标并计算距离
    x, y, z = points.T
    distances = np.sqrt(x**2 + y**2 + z**2)

    # 距离过滤
    valid_mask = (distances >= self.distance_min) & (distances <=
self.distance_max)
    x, y, z, distances = x[valid_mask], y[valid_mask], z[valid_mask],
distances[valid_mask]

    # 计算角度坐标
    azimuth = -np.arctan2(y, x)
    elevation = -np.arctan2(z, np.sqrt(x**2 + y**2))

    # 修改坐标计算
    u = (azimuth + self.radian_horizontal/2) *
(self.depthImage_width / self.radian_horizontal)
    v = (elevation + self.radian_vertical/2) *
(self.depthImage_height / self.radian_vertical)

    # 箝位到有效范围
    u = np.clip(u, 0, self.depthImage_width - 1).astype(np.int32)
    v = np.clip(v, 0, self.depthImage_height - 1).astype(np.int32)

    # 创建临时数组存储最小距离
    temp_array = np.full_like(depth_array, np.inf)
    np.minimum.at(temp_array, (v, u), distances)  # 原子最小化操作

    # 仅更新比当前值小的距离
    update_mask = temp_array < depth_array
    depth_array[update_mask] = temp_array[update_mask]

    # === 图像归一化 ===
    # 创建8UC1图像
    depth_image = np.zeros((self.depthImage_height,
self.depthImage_width), dtype=np.uint8)

    # 有效区域掩码
    valid_mask = (depth_array >= self.distance_min) & (depth_array <=
self.distance_max)

    if np.any(valid_mask):
        # 归一化并转换为0-255
```

```
        normalized = (depth_array[valid_mask] - self.distance_min)
        normalized /= (self.distance_max - self.distance_min)
        depth_image[valid_mask] = (normalized * 255).astype(np.uint8)

    # 构建 Image 消息
    image_msg = Image()
    image_msg.header = msg.header
    image_msg.height = self.depthImage_height
    image_msg.width = self.depthImage_width
    image_msg.encoding = "8UC1"
    image_msg.step = self.depthImage_width
    image_msg.data = depth_image.flatten().tolist()

    # 发布图像
    self.depthImage_publisher.publish(image_msg)
```

4.3. 对碰撞进行检测

首先需要在 URDF 文件中添加一个 Gazebo 插件，用于检测接触。

```
<!-- 碰撞检测插件 -->
<gazebo reference="body_link">
    <selfCollide>false</selfCollide>
    <sensor name="base_contact_sensor" type="contact">
        <always_on>true</always_on>
        <update_rate>100.0</update_rate>
        <contact>

<collision>body_link_fixed_joint_lump__body_collision_collision</
collision> <!-- 这个名字需要将xacro/urdf 转换为 sdf，使用 sdf 中的
collision 名字 -->
        </contact>
        <plugin filename="libgazebo_ros_bumper.so"
name="base_gazebo_ros_bumper_controller">
            <!-- /bumper_states -->
        </plugin>
    </sensor>
</gazebo>
```

碰撞信息的内容较为复杂，这里只对是否发生碰撞进行检测。

```
from gazebo_msgs.msg import ContactsState

# 订阅碰撞检测
```

```
self.collision_sub = self.create_subscription(
    ContactsState,
    "/bumper_states",
    self.collision_handler,
    10,
)

def collision_handler(self, msg):
    if msg.states:
        self.get_logger().info("检测到碰撞")
        self.collision = True
    else:
        self.collision = False
```

5. 使用 ROS2 进行强化学习

5.1. 将 ROS2 封装为 Gymnasium 环境

将 ROS2 封装为 Gymnasium 环境的核心是在 ROS2 的 Node 中实现`step()`和`reset()`方法，然后在继承自`gymnasium.Env`的类中编写对应的处理。

```
import gymnasium
from gymnasium import spaces
import numpy as np
from trainer.trainer_bridge import TrainerBridge
import rclpy
from typing import Optional, Dict, Any
import torch
import time

class GazeboEnv(gymnasium.Env):
    metadata = {'render_modes': ['human']}

    def __init__(self, render_mode: Optional[str] = None,
max_episode_steps: int = 2048):
        super().__init__()

        self.max_episode_steps = max_episode_steps

        # 延迟初始化 ROS，避免多进程冲突
        self.bridge_initialized = False
        self._init_bridge()

        # 定义 observation_space 为 Dict 类型
        self.observation_space = spaces.Dict({
            "depth_image": spaces.Box(
                low=0,
                high=1,
                shape=(1, 16, 180),
                dtype=np.float64
            ),
            "goal": spaces.Box(
                low=-np.inf,
                high=np.inf,
                shape=(2,),
                dtype=np.float64
            ),
        })
        self.action_space = spaces.Box(
```

```
            low=np.array([0.0, -10], dtype=np.float64),
            high=np.array([10.0, 10], dtype=np.float64),
            shape=(2,),
            dtype=np.float64
        )

    def _init_bridge(self):
        if not self.bridge_initialized:
            if not rclpy.ok():
                rclpy.init()
            self.agent = TrainerBridge()
            self.bridge_initialized = True

    def reset(self, seed: Optional[int] = None, options:
Optional[dict] = None):
        super().reset(seed=seed)
        obs = self.agent.reset()
        self.steps_in_episode = 0  # 重置计数器
        return obs, {}

    def step(self, action):
        time.sleep(0.01)

        obs, reward, done, info = self.agent.step(action)
        self.steps_in_episode += 1

        # 如果达到最大步数，则设置 done=True 并 reset
        if self.steps_in_episode >= self.max_episode_steps:
            done = True

        truncated = False
        return obs, reward, done, truncated, info

    def close(self):
        if self.bridge_initialized:
            self.agent.destroy_node()
            rclpy.shutdown()
            self.bridge_initialized = False
```

5.2. 使用 Stable Baselines3 进行训练和评估

```
import rclpy
from rclpy.node import Node
import gymnasium as gym
from stable_baselines3 import PPO, SAC, TD3
```

```
from .gazebo_env import GazeboEnv
from stable_baselines3.common.callbacks import CheckpointCallback
from .extractor import Extractor

class TrainingNode(Node):
    def __init__(self):
        super().__init__('training_node')
        self.get_logger().info('Training Node Started')

        # 注册 Gym 环境
        gym.register("GazeboSimple-v0",
entry_point="trainer.gazebo_env:GazeboEnv")

        self.env = GazeboEnv()

        # 添加回调：每 10,000 步保存一次模型
        checkpoint_callback = CheckpointCallback(
            save_freq=10000,
            save_path='./checkpoints/',
            name_prefix='model' # 文件名前缀，例如 model_10000_steps
        )

        self.get_logger().info('[TrainingNode] initialize trainer...')
        policy_kwargs = dict(
            features_extractor_class=Extractor,
            features_extractor_kwargs=dict(features_dim=256),
            normalize_images=False,
        )

        self.model = SAC(
            policy="MultiInputPolicy",
            env=self.env,
            policy_kwargs=policy_kwargs,
            verbose=1,
            learning_rate=3e-4,
            device="cuda",
            buffer_size=100000,
            train_freq=10,
            tensorboard_log="./tensorboard/"
        )

        self.get_logger().info('[TrainingNode] Starting training...')
        self.model.learn(
            total_timesteps=1000000,
            callback=checkpoint_callback
        )
        self.get_logger().info('[TrainingNode] training end...')
```

```
        self.model.save("model_end")
        self.get_logger().info('Training complete and model saved.')

def main(args=None):
    rclpy.init()
    node = TrainingNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

5.3. 使用 RLlib 进行训练和评估

ROS2

学习笔记

Version 0.3.1

2025 年 07 月 22 日

vkyuxr, Qwen